

# Deep Learning - The Straight Dope

This repo contains an incremental sequence of notebooks designed to teach deep learning, [Apache MXNet \(incubating\)](#), and the gluon interface. Our goal is to leverage the strengths of Jupyter notebooks to present prose, graphics, equations, and code together in one place. If we're successful, the result will be a resource that could be simultaneously a book, course material, a prop for live tutorials, and a resource for plagiarising (with our blessing) useful code. To our knowledge there's no source out there that teaches either (1) the full breadth of concepts in modern deep learning or (2) interleaves an engaging textbook with runnable code. We'll find out by the end of this venture whether or not that void exists for a good reason.

Another unique aspect of this book is its authorship process. We are developing this resource fully in the public view and are making it available for free in its entirety. While the book has a few primary authors to set the tone and shape the content, we welcome contributions from the community and hope to coauthor chapters and entire sections with experts and community members. Already we've received contributions spanning typo corrections through full working examples.

## How to contribute

To clone or contribute, visit [Deep Learning - The Straight Dope](#) on Github.

## Dependencies

To run these notebooks, a recent version of MXNet is required. The easiest way is to install the nightly build MXNet through `pip`. E.g.:

```
$ pip install mxnet --pre --user
```

More detailed instructions are available [here](#)

## Part 1: Deep Learning Fundamentals

### Crash course

- [Preface](#)
- [Introduction](#)
- [Manipulate data the MXNet way with `ndarray`](#)
- [Linear algebra](#)
- [Intermediate linear algebra](#)
- [Probability and statistics](#)
- [Automatic differentiation with `autograd`](#)

## Introduction to supervised learning

- [Linear regression from scratch](#)
- [Linear regression with `gluon`](#)
- [The Perceptron](#)
- [Multiclass logistic regression from scratch](#)
- [Multiclass logistic regression with `gluon`](#)
- [Overfitting and regularization](#)
- [Overfitting and regularization \(with `gluon`\)](#)
- [Environment](#)

## Deep neural networks

- [Multilayer perceptrons from scratch](#)
- [Multilayer perceptrons in `gluon`](#)
- [Dropout regularization from scratch](#)
- [Dropout regularization with `gluon`](#)
- [Plumbing: A look under the hood of `gluon`](#)
- [Designing a custom layer with `gluon`](#)
- [Serialization - saving, loading and checkpointing](#)

## Convolutional neural networks

- [Convolutional neural networks from scratch](#)
- [Convolutional Neural Networks in `gluon`](#)
- [Deep convolutional neural networks](#)
- [Very deep networks with repeating elements](#)
- [Batch Normalization from scratch](#)
- [Batch Normalization in `gluon`](#)

## Recurrent neural networks

- [Recurrent Neural Networks \(RNNs\) for Language Modeling](#)

- Long short-term memory (LSTM) RNNs
- Gated recurrent unit (GRU) RNNs
- Recurrent Neural Networks with `gluon`

## Optimization

- Introduction
- Optimization by gradient descent
- Stochastic gradient descent with momentum

## High-performance and distributed training

- Fast, portable neural networks with Gluon HybridBlocks
- Training with multiple GPUs from scratch
- Training on multiple GPUs with `gluon`
- Distributed training with multiple machines

## Part 2: Applications

### Computer vision

- Object Detection Using Convolutional Neural Networks
- Transferring knowledge through finetuning
- Visual Question Answering in gluon

### Natural language processing

- Tree LSTM modeling for semantic relatedness

### Recommender systems

- Introduction to recommender systems

### Time series

- Linear Dynamical Systems with MXNet
- Filtering
- Generating Synthetic Dataset
- Exponential Smoothing and Innovation State Space Model (ISSM)
- Filtering

# Part 3: Advanced Topics

## Generative adversarial networks

- [Generative Adversarial Networks](#)
- [Deep Convolutional Generative Adversarial Networks](#)
- [Pixel to Pixel Generative Adversarial Networks](#)

## Variational methods

- [Bayes by Backprop from scratch \(NN, classification\)](#)
- [Bayes by Backprop with `gluon` \(NN, classification\)](#)

## Cheat sheets

- [Kaggle house price prediction with `Gluon` and k-fold cross-validation](#)

## Developer documents

- [Run these tutorials](#)
- [How to contribute](#)



# Preface

If you're a reasonable person, you might ask, "what is *mxnet-the-straight-dope*?" You might also ask, "why does it have such an ostentatious name?" Speaking to the former question, *mxnet-the-straight-dope* is an attempt to create a new kind of educational resource for deep learning. Our goal is to leverage the strengths of Jupyter notebooks to present prose, graphics, equations, and (importantly) code together in one place. If we're successful, the result will be a resource that could be simultaneously a book, course material, a prop for live tutorials, and a resource for plagiarising (with our blessing) useful code. To our knowledge, few available resources aim to teach either (1) the full breadth of concepts in modern machine learning or (2) interleave an engaging textbook with runnable code. We'll find out by the end of this venture whether or not that void exists for a good reason.

Regarding the name, we are cognizant that the machine learning community and the ecosystem in which we operate have lurched into an absurd place. In the early 2000s, comparatively few tasks in machine learning had been conquered, but we felt that we understood *how* and *why* those models worked (with some caveats). By contrast, today's machine learning systems are extremely powerful and *actually work* for a growing list of tasks, but huge open questions remain as to precisely *why* they are so effective.

This new world offers enormous opportunity, but has also given rise to considerable buffoonery. Research preprints like [the arXiv](#) are flooded by clickbait, AI startups have sometimes received overly optimistic valuations, and the blogosphere is flooded with thought leadership pieces written by marketers bereft of any technical knowledge. Amid the chaos, easy money, and lax standards, we believe it's important not to take our models or the environment in which they are worshipped too seriously. Also, in order to both explain, visualize, and code the full breadth of models that we aim to address, it's important that the authors do not get bored while writing.

## Organization

At present, we're aiming for the following format: aside from a few (optional) notebooks providing a crash course in the basic mathematical background, each subsequent notebook will both:

1. Introduce a reasonable number (perhaps one) of new concepts
2. Provide a single self-contained working example, using a real dataset

This presents an organizational challenge. Some models might logically be grouped together in a single notebook. And some ideas might be best taught by executing several models in succession. On the other hand, there's a big advantage to adhering to a policy of *1 working example, 1 notebook*. This makes it as easy as possible for you to start your own research projects by plagiarising our code. Just copy a single notebook and start modifying it.

We will interleave the runnable code with background material as needed. In general, we will often err on the side of making tools available before explaining them fully (and we will follow up by explaining the background later). For instance, we might use *stochastic gradient descent* before fully explaining why it is useful or why it works. This helps to give practitioners the necessary ammunition to solve problems quickly, at the expense of requiring the reader to trust us with some decisions, at least in the short term. Throughout, we'll be working with the MXNet library, which has the rare property of being flexible enough for research while being fast enough for production. Our more advanced chapters will mostly rely on MXNet's new high-level imperative interface `gluon`. Note that this is not the same as `mxnet.module`, an older, symbolic interface supported by MXNet.

This book will teach deep learning concepts from scratch. Sometimes, we'll want to delve into fine details about the models that are hidden from the user by `gluon`'s advanced features. This comes up especially in the basic tutorials, where we'll want you to understand everything that happens in a given layer. In these cases, we'll generally present two versions of the example: one where we implement everything from scratch, relying only on NDArray and automatic differentiation, and another where we show how to do things succinctly with `gluon`. Once we've taught you how a layer works, we can just use the `gluon` version in subsequent tutorials.

## Learning by doing

Many textbooks teach a series of topics, each in exhaustive detail. For example, Chris Bishop's excellent textbook, [Pattern Recognition and Machine Learning](#), teaches each topic so thoroughly, that getting to the chapter on linear regression requires a non-trivial amount of work. When I (Zack) was first learning machine learning, this actually limited the book's usefulness as an introductory text. When I rediscovered it a couple years later, I loved it precisely for its thoroughness, and I hope you check it out after working through this material! But perhaps the traditional textbook approach is not the easiest way to get started in the first place.

Instead, in this book, we'll teach most concepts just in time. For the fundamental preliminaries like linear algebra and probability, we'll provide a brief crash course from the outset, but we want you to taste the satisfaction of training your first model before worrying about exotic probability distributions.

## Next steps

If you're ready to get started, head over to [the introduction](#) or go straight to [our basic primer on NDArray](#), MXNet's workhorse data structure.

For whinges or inquiries, [open an issue on GitHub](#).

# Introduction

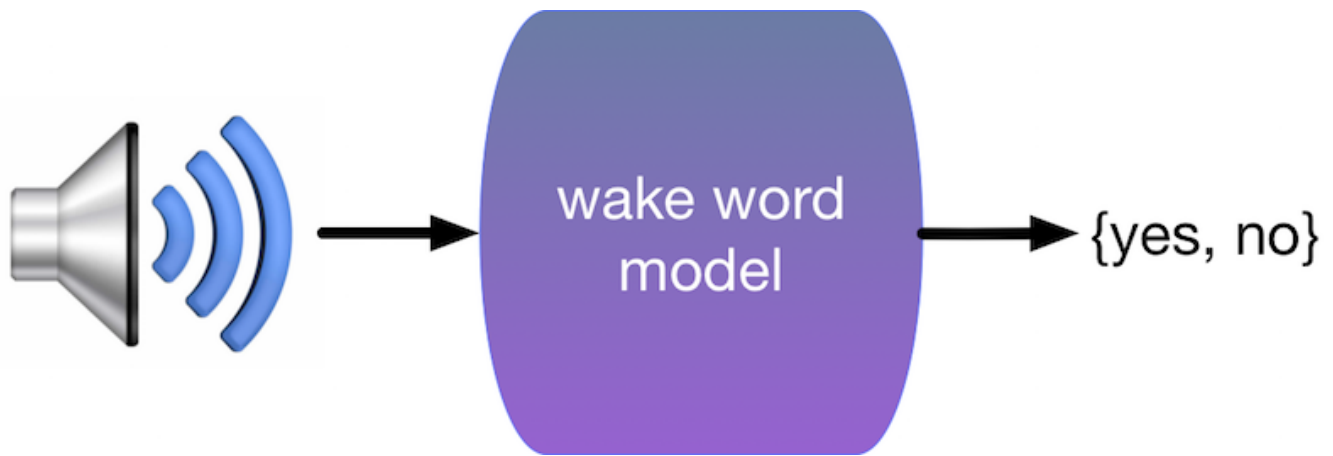
Before we could begin writing, the authors of this book, like much of the work force, had to become caffeinated. We hopped in the car and started driving. Having an Android, Alex called out “Okay Google”, awakening the phone’s voice recognition system. Then Mu commanded “directions to Blue Bottle coffee shop”. The phone quickly displayed the transcription of his command. It also recognized that we were asking for directions and launched the Maps application to fulfill our request. Once launched, the Maps app identified a number of routes. Next to each route, the phone displayed a predicted transit time. While we fabricated this story for pedagogical convenience, it demonstrates that in the span of just a few seconds, our everyday interactions with a smartphone can engage several machine learning models.

If you’ve never worked with machine learning before, you might be wondering what the hell we’re talking about. You might ask, “isn’t that just programming?” or “what does *machine learning* even mean?” First, to be clear, we implement all machine learning algorithms by writing computer programs. Indeed, we use the same languages and hardware as other fields of computer science, but not all computer programs involve machine learning. In response to the second question, precisely defining a field of study as vast as machine learning is hard. It’s a bit like answering, “what is math?”. But we’ll try to give you enough intuition to get started.

## A motivating example

Most of the computer programs we interact with every day can be coded up from first principles. When you add an item to your shopping cart, you trigger an e-commerce application to store an entry in a *shopping cart* database table, associating your user ID with the product’s ID. We can write such a program from first principles, launch without ever having seen a real customer. When it’s this easy to write an application *you should not be using machine learning*.

Fortunately (for the community of ML scientists), however, for many problems, solutions aren’t so easy. Returning to our fake story about going to get coffee, imagine just writing a program to respond to a *wake word* like “Alexa”, “Okay, Google” or “Siri”. Try coding it up in a room by yourself with nothing but a computer and a code editor. How would you write such a program from first principles? Think about it... the problem is hard. Every second, the microphone will collect roughly 44,000 samples. What rule could map reliably from a snippet of raw audio to confident predictions `{yes, no}` on whether the snippet contains the wake word? If you’re stuck, don’t worry. We don’t know how to write such a program from scratch either. That’s why we use machine learning.



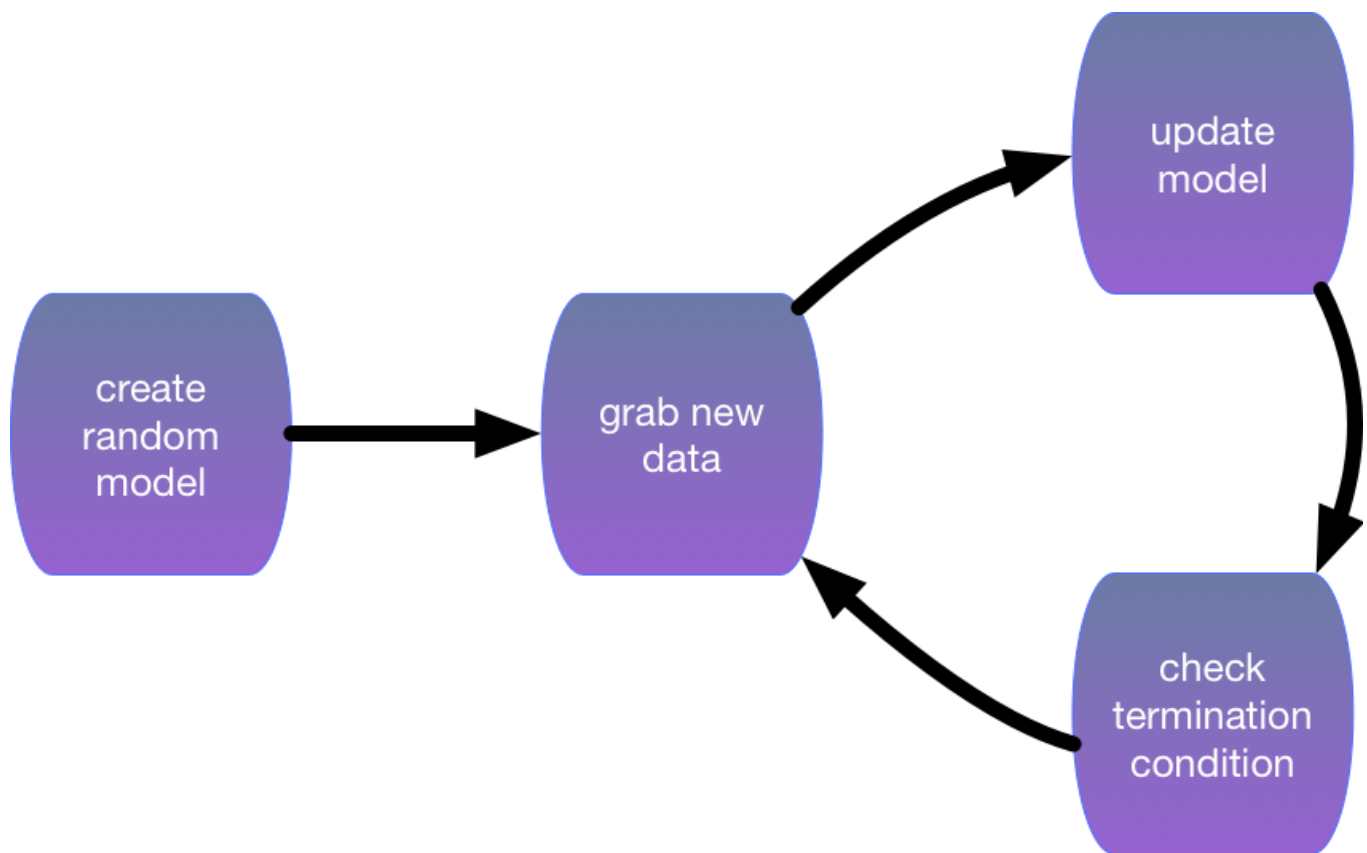
Here's the trick. Often, even when we don't know how to tell a computer explicitly how to map from inputs to outputs, we ourselves are nonetheless capable of performing the cognitive feat ourselves. In other words, even if you don't know *how to program a computer* to recognize the word "Alexa", you yourself *are able* to recognize the word "Alexa". Armed with this ability, we can collect a huge *data set* containing examples of audio and label those that *do* and that *do not* contain the wake word. In the machine learning approach, we do not design a system *explicitly* to recognize wake words right away. Instead, we define a flexible program with a number of *parameters*. These are knobs that we can tune to change the behavior of the program. We call this program a model. Generally, our model is just a machine that transforms its input into some output. In this case, the model receives as *input* a snippet of audio, and it generates as output an answer `{yes, no}`, which we hope reflects whether (or not) the snippet contains the wake word.

If we choose the right kind of model, then there should exist one setting of the knobs such that the model fires `yes` every time it hears the word "Alexa". There should also be another setting of the knobs that might fire `yes` on the word "Apricot". We expect that the same model should apply to "Alexa" recognition and "Apricot" recognition because these are similar tasks. However, we might need a different model to deal with fundamentally different inputs or outputs. For example, we might choose a different sort of machine to map from images to captions, or from English sentences to Chinese sentences.

As you might guess, if we just set the knobs randomly, the model will probably recognize neither "Alexa", "Apricot", nor any other word in the English language. In most deep learning, the *learning* refers precisely to updating the model's behavior (by twisting the knobs) over the course of a *training period*.





The training process usually looks like this:

1. Start off with a randomly initialized model that can't do anything useful
2. Grab some of your labeled data (e.g. audio snippets and corresponding `{yes, no}` labels)
3. Tweak the knobs so the model sucks less with respect to those examples
4. Repeat until the model is dope



To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, *if we present it with a large labeled dataset*. You can think of this act of determining a program’s behavior by presenting it with a dataset as *programming with data*.

We can ‘program’ a cat detector by providing our machine learning system with many examples of cats and dogs, such as the images below:

			
cat	cat	dog	dog

This way the detector will eventually learn to emit a very large positive number if it’s a cat, a very large negative number if it’s a dog, and something closer to zero if it isn’t sure, but this is just barely scratching the surface of what machine learning can do.

## The dizzying versatility of machine learning

This is the core idea behind machine learning: Rather than code programs with fixed behavior, we design programs with the ability to improve as they acquire more experience. This basic idea can take many forms. Machine learning can address many different application domains, involve

many different types of models, and update them according to many different learning algorithms. In this particular case, we described an instance of *supervised learning* applied to a problem in automated speech recognition.

Machine Learning is a versatile set of tools that lets you work with data in many different situations where simple rule-based systems would fail or might be very difficult to build. Due to its versatility, machine learning can be quite confusing to newcomers. For example, machine learning techniques are already widely used in applications as diverse as search engines, self driving cars, machine translation, medical diagnosis, spam filtering, game playing (*chess, go*), face recognition, data matching, calculating insurance premiums, and adding filters to photos.

Despite the superficial differences between these problems many of them share a common structure and are addressable with deep learning tools. They're mostly similar because they are problems where we wouldn't be able to program their behavior directly in code, but we can *program them with data*. Often times the most direct language for communicating these kinds of programs is *math*. In this book, we'll introduce a minimal amount of mathematical notation, but unlike other books on machine learning and neural networks, we'll always keep the conversation grounded in real examples and real code.

## Basics of machine learning

When we considered the task of recognizing wake-words, we put together a dataset consisting of snippets and labels. We then described (albeit abstractly) how you might train a machine learning model to predict the label given a snippet. This set-up, predicting labels from examples, is just one flavor of ML and it's called *supervised learning*. Even within deep learning, there are many other approaches, and we'll discuss each in subsequent sections. To get going with machine learning, we need four things:

1. Data
2. A model of how to transform the data
3. A loss function to measure how well we're doing
4. An algorithm to tweak the model parameters such that the loss function is minimized

## Data

Generally, the more data we have, the easier our job as modelers. When we have more data, we can train more powerful models. Data is at the heart of the resurgence of deep learning and many of most exciting models in deep learning don't work without large data sets. Here are some examples of the kinds of data machine learning practitioners often engage with:

- **Images:** Pictures taken by smartphones or harvested from the web, satellite images, photographs of medical conditions, ultrasounds, and radiologic images like CT scans and

MRIs, etc.

- **Text:** Emails, high school essays, tweets, news articles, doctor's notes, books, and corpora of translated sentences, etc.
- **Audio:** Voice commands sent to smart devices like Amazon Echo, or iPhone or Android phones, audio books, phone calls, music recordings, etc.
- **Video:** Television programs and movies, YouTube videos, cell phone footage, home surveillance, multi-camera tracking, etc.
- **Structured data:** This Jupyter notebook (it contains text, images, code), webpages, electronic medical records, car rental records, electricity bills, etc.

## Models

Usually the data looks quite different from what we want to accomplish with it. For example, we might have photos of people and want to know whether they appear to be happy. We might desire a model capable of ingesting a high-resolution image and outputting a happiness score. While some simple problems might be addressable with simple models, we're asking a lot in this case. To do its job, our happiness detector needs to transform hundreds of thousands of low-level features (pixel values) into something quite abstract on the other end (happiness scores). Choosing the right model is hard, and different models are better suited to different datasets. In this book, we'll be focusing mostly on deep neural networks. These models consist of many successive transformations of the data that are chained together top to bottom, thus the name *deep learning*. On our way to discussing deep nets, we'll also discuss some simpler, shallower models.

## Loss functions

To assess how well we're doing we need to compare the output from the model with the truth. Loss functions give us a way of measuring how *bad* our output is. For example, say we trained a model to infer a patient's heart rate from images. If the model predicted that a patient's heart rate was 100bpm, when the ground truth was actually 60bpm, we need a way to communicate to the model that it's doing a lousy job.

Similarly if the model was assigning scores to emails indicating the probability that they are spam, we'd need a way of telling the model when its predictions are bad. Typically the *learning* part of machine learning consists of minimizing this loss function. Usually, models have many parameters. These are the ones that we need to 'learn', by minimizing the loss incurred on training data. Unfortunately, doing well on the latter doesn't guarantee that we will do well on (unseen) test data, so we'll want to keep track of two quantities.

- **Training Error:** This is the error on the dataset used to find  $f$  by minimizing the loss on the training set. This is equivalent to doing well on all the practice exams that a student might use to prepare for the real exam. The results are encouraging, but by no means a guarantee.



- **Test Error:** This is the error incurred on an unseen test set. This can be off by quite a bit (statisticians call this overfitting). In real-life terms, this is the equivalent of screwing up the real exam despite doing well on the practice exams.

## Optimization algorithms

Finally, to minimize the loss, we'll need some way of taking the model and its loss functions, and searching for a set of parameters that minimizes the loss. The most popular optimization algorithms for work on neural networks follow an approach called gradient descent. In short, they look to see, for each parameter which way the training set loss would move if you jiggled the parameter a little bit. They then update the parameter in the direction that reduces the loss.

In the following sections, we will discuss a few types of machine learning in some more detail. This helps to understand what exactly one aims to do. We begin with a list of *objectives*, i.e. a list of things that machine learning can do. Note that the objectives are complemented with a set of techniques of *how* to accomplish them, i.e. training, types of data, etc. The list below is really only sufficient to whet the readers' appetite and to give us a common language when we talk about problems. We will introduce a larger number of such problems as we go along.

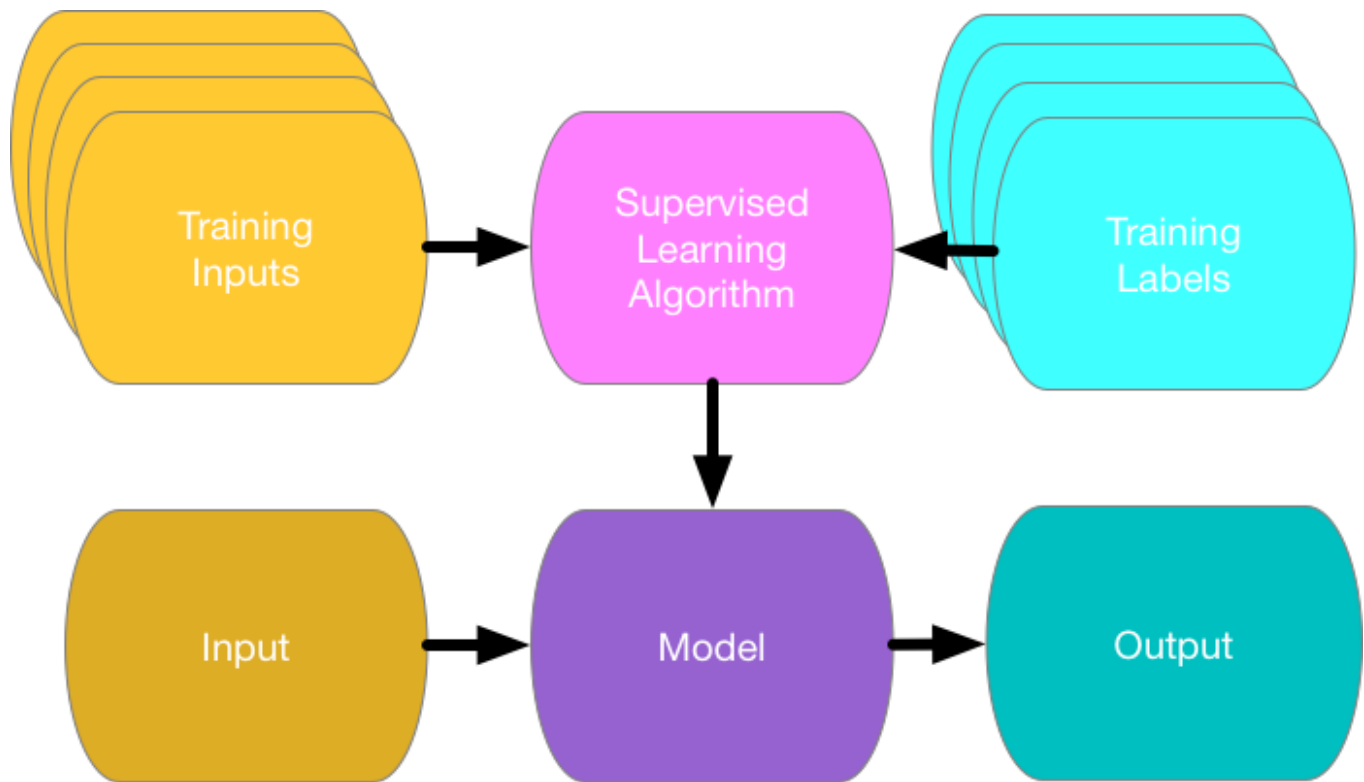
## Supervised learning

Supervised learning describes the task of predicting targets  $y$  given inputs  $x$  by training on labeled examples. In probabilistic terms, supervised learning is concerned with estimating the conditional probability  $P(y|x)$ . While it's just one among several approaches to machine learning, supervised learning accounts for the majority of machine learning in practice. Partly, that's because many important tasks can be described crisply as predicting something unknown from something known: \* Predict cancer vs not cancer, given a CT image \* Predict the correct translation in French, given a sentence in English \* Predict the price of a stock next month based on this month's financial reporting data

Even with the simple description "predict targets from inputs" supervised learning can take a great many forms and require a great many modeling decisions, depending on the type, size, and the number of inputs and outputs. For example, we use different models to process sequences (like strings of text or time series data) and for processing fixed-length vector representations. We'll visit many of these problems in depth throughout the first 9 parts of this book.

Put plainly, the learning process looks something like this. Grab a big pile of example inputs, selecting them randomly. Acquire the ground truth labels for each. Together, these inputs and corresponding labels (the desired outputs) comprise the training set. We feed the training dataset into a supervised learning algorithm. So here the *supervised learning algorithm* is a

function that takes as input a dataset, and outputs another function, *the learned model*. Then, given a learned model, we can take a new previously unseen input, and predict the corresponding label.



## Regression

Perhaps the simplest supervised learning task to wrap your head around is Regression. Consider, for example a set of data harvested from a database of home sales. We might construct a table, where each row corresponds to a different house, and each column corresponds to some relevant attribute, such as the square footage of a house, the number of bedrooms, the number of bathrooms, and the number of minutes (walking) to the center of town. Formally, we call one row in this dataset a *feature vector*, and the object (e.g. a house) it's associated with an *example*.

If you live in New York or San Francisco, and you are not the CEO of Amazon, Google, Microsoft, or Facebook, the (sq. footage, no. of bedrooms, no. of bathrooms, walking distance) feature vector for your home might look something like: [100, 0, .5, 60]. However, if you live in Pittsburgh, it might look more like [3000, 4, 3, 10]. Feature vectors like this are essential for all the classic machine learning problems. We'll typically denote the feature vector for any one example  $\mathbf{x}_i$  and the set of feature vectors for all our examples  $X$ .

What makes a problem *regression* is actually the outputs. Say that you're in the market for a new home, you might want to estimate the fair market value of a house, given some features like these. The target value, the price of sale, is a *real number*. We denote any individual target  $y_i$  (corresponding to example  $\mathbf{x}_i$ ) and the set of all targets  $\mathbf{y}$  (corresponding to all examples  $X$ ). When our targets take on arbitrary real values in some range, we call this a

regression problem. The goal of our model is to produce predictions (guesses of the price, in our example) that closely approximate the actual target values.

We denote these predictions  $\hat{y}_i$  and if the notation seems whacky, then just ignore it for now. We'll unpack it more thoroughly in the subsequent chapters.

Lots of practical problems are well-described regression problems. Predicting the rating that a user will assign to a movie is a regression problem, and if you designed a great algorithm to accomplish this feat in 2009, you might have won the [\\$1 million Netflix prize](#). Predicting the length of stay for patients in the hospital is also a regression problem. A good rule of thumb is that any *How much?* or *How many?* problem should suggest regression. \* “How many hours will this surgery take?”... *regression* \* “How many dogs are in this photo?” ... *regression*. However, if you can easily pose your problem as “Is this a \_\_\_?”, then it's likely, classification, a different fundamental problem type that we'll cover next.

Even if you've never worked with machine learning before, you've probably worked through a regression problem informally. Imagine, for example, that you had your drains repaired and that your contractor, spent  $x_1 = 3$  hours removing gunk from your sewage pipes. Then she sent you a bill of  $y_1 = \$350$ . Now imagine that your friend hired the same contractor for  $x_2 = 2$  hours and that she received a bill of  $y_2 = \$250$ . If someone then asked you how much to expect on their upcoming gunk-removal invoice you might make some reasonable assumptions, such as more hours worked costs more dollars. You might also assume that there's some base charge and that the contractor then charges per hour. If these assumptions held, then given these two data points, you could already identify the contractor's pricing structure: \$100 per hour plus \$50 to show up at your house. If you followed that much then you already understand the high-level idea behind linear regression.

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes that's not possible, e.g., if some of the variance owes to some factors besides your two features. In these cases, we'll try to learn models that minimize the distance between our predictions and the observed values. In most of our chapters, we'll focus on one of two very common losses, the [L1 loss](#) where  $l(y, y') = \sum_i |y_i - y'_i|$  and the [L2 loss](#) where  $l(y, y') = \sum_i (y_i - y'_i)^2$ . As we will see later, the  $L_2$  loss corresponds to the assumption that our data was corrupted by Gaussian noise, whereas the  $L_1$  loss corresponds to an assumption of noise from a Laplace distribution.

## Classification

While regression models are great for addressing *how many?* questions, lots of problems don't bend comfortably to this template. For example, a bank wants to add check scanning to their mobile app. This would involve the customer snapping a photo of a check with their smartphone's camera and the machine learning model would need to be able to automatically understand text seen in the image. It would also need to understand hand-written text to be

even more robust. This kind of system is referred to as optical character recognition (OCR), and the kind of problem it solves is called a classification. It's treated with a distinct set of algorithms than those that are used for regression.

In classification, we want to look at a feature vector, like the pixel values in an image, and then predict which category (formally called *classes*), among some set of options, an example belongs to. For hand-written digits, we might have 10 classes, corresponding to the digits 0 through 9. The simplest form of classification is when there are only two classes, a problem which we call binary classification. For example, our dataset  $X$  could consist of images of animals and our *labels*  $Y$  might be the classes  $\{\text{cat}, \text{dog}\}$ . While in regression, we sought a regressor to output a real value  $\hat{y}$ , in classification, we seek a *classifier*, whose output  $\hat{y}$  is the predicted class assignment.

For reasons that we'll get into as the book gets more technical, it's pretty hard to optimize a model that can only output a hard categorical assignment, e.g. either *cat* or *dog*. It's a lot easier instead to express the model in the language of probabilities. Given an example  $x$ , the model assigns a probability  $\hat{y}_k$  to each label  $k$ . Because these are probabilities, they need to be positive numbers and add up to 1. This means that we only need  $K - 1$  numbers to give the probabilities of  $K$  categories. This is easy to see for binary classification. If there's a 0.6 (60%) probability that an unfair coin comes up heads, then there's a 0.4 (40%) probability that it comes up tails. Returning to our animal classification example, a classifier might see an image and output the probability that the image is a cat  $\Pr(y = \text{cat} \mid x) = 0.9$ . We can interpret this number by saying that the classifier is 90% sure that the image depicts a cat. The magnitude of the probability for the predicted class is one notion of confidence. It's not the only notion of confidence and we'll discuss different notions of uncertainty in more advanced chapters.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include hand-written character recognition `[0, 1, 2, 3 ... 9, a, b, c, ...]`. While we attacked regression problems by trying to minimize the L1 or L2 loss functions, the common loss function for classification problems is called cross-entropy. In `MXNet Gluon`, the corresponding loss function can be found [here](#).

Note that the most likely class is not necessarily the one that you're going to use for your decision. Assume that you find this beautiful mushroom in your backyard:





Death cap - do not eat!

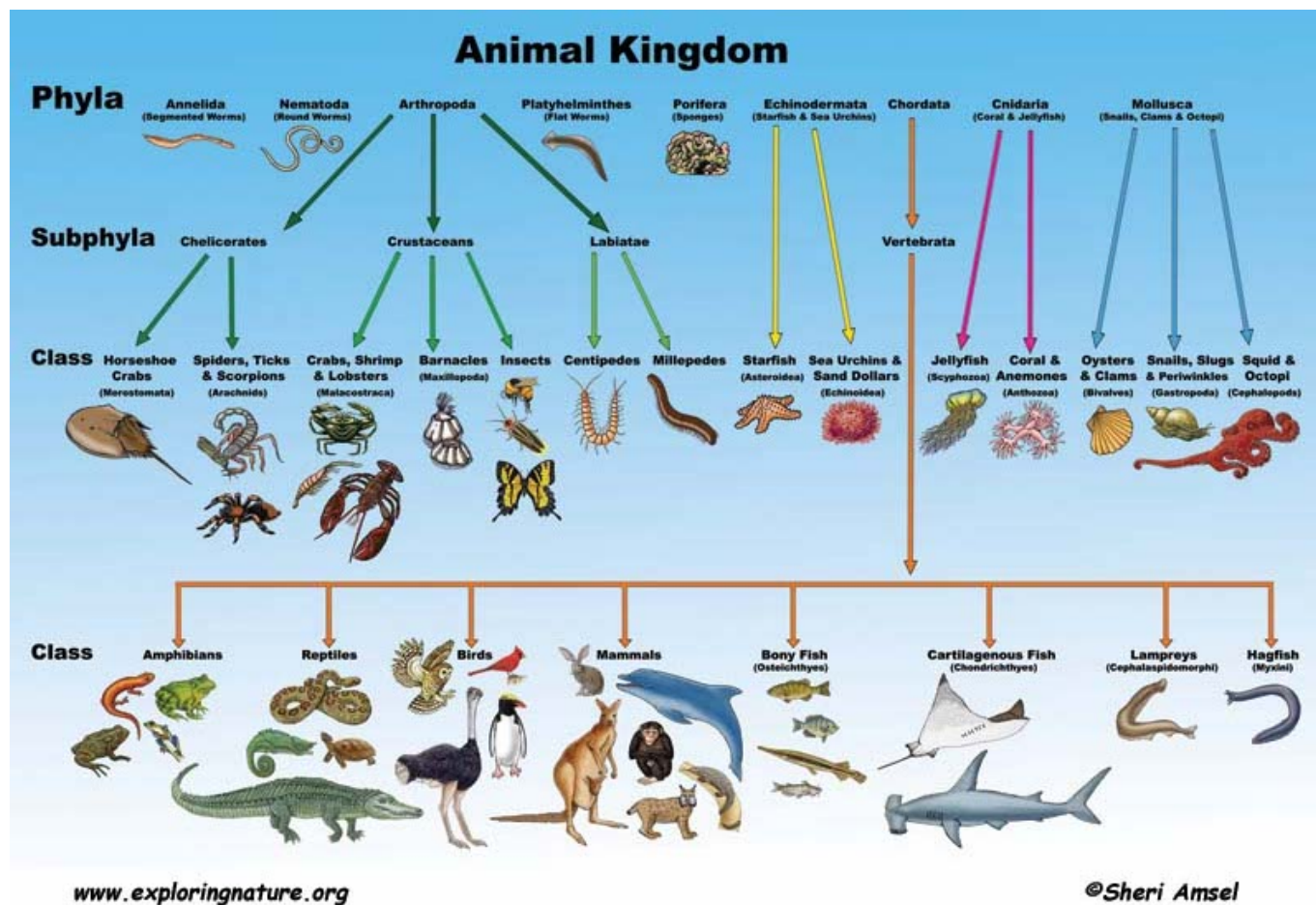
Now, assume that you built a classifier and trained it to predict if a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs

$\Pr(y = \text{deathcap} \mid \text{image}) = 0.2$  In other words, the classifier is 80% confident that our mushroom *is not* a death cap. Still, you'd have to be a fool to eat it. That's because the certain benefit of a delicious dinner isn't worth a 20% chance of dying from it. In other words, the effect of the *uncertain risk* by far outweighs the benefit. Let's look at this in math. Basically, we need to compute the expected risk that we incur, i.e. we need to multiply the probability of the outcome with the benefit (or harm) associated with it:

$$L(\text{action} \mid x) = \mathbf{E}_{y \sim p(y|x)}[\text{loss}(\text{action}, y)]$$

Hence, the loss  $L$  incurred by eating the mushroom is  $L(a = \text{eat} \mid x) = 0.2 * \infty + 0.8 * 0 = \infty$  whereas the cost of discarding it is  $L(a = \text{discard} \mid x) = 0.2 * 0 + 0.8 * 1 = 0.8$

We got lucky: as any mycologist would tell us, the above actually *is* a death cap. Classification can get much more complicated than just binary, multiclass, or even multi-label classification. For instance, there are some variants of classification for addressing hierarchies. Hierarchies assume that there exist some relationships among the many classes. So not all errors are equal - we prefer to misclassify to a related class than to a distant class. Usually, this is referred to as *hierarchical classification*. One early example is due to [Linnaeus](#), who organized the animals in a hierarchy..



In the case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle for a dinosaur. What hierarchy is relevant might depend on how you plan to use the model. For example, rattlesnakes and garter snakes might be close on the phylogenetic tree, but mistaking a rattler for a garter could be deadly.

## Tagging

Some classification problems don't fit neatly into the binary or multiclass classification setups. For example, we could train a normal binary classifier to distinguish cats from dogs. Given the current state of computer vision, we can do this easily, with off-the-shelf tools. Nonetheless, no matter how accurate our model gets, we might find ourselves in trouble when the classifier encounters an image like this:





As you can see, there's a cat in the picture. There is also a dog, a tire, some grass, a door, concrete, rust, individual grass leaves, etc. Depending on what we want to do with our model ultimately, treating this as a binary classification problem might not make a lot of sense. Instead, we might want to give the model the option of saying the image depicts a cat *and* a dog, or *neither* a cat *nor* a dog.

The problem of learning to predict classes that are *not mutually exclusive* is called multi-label classification. Auto-tagging problems are typically best described as multi-label classification problems. Think of the tags people might apply to posts on a tech blog, e.g., "machine learning", "technology", "gadgets", "programming languages", "linux", "cloud computing", "AWS". A typical article might have 5-10 tags applied because these concepts are correlated. Posts about "cloud computing" are likely to mention "AWS" and posts about "machine learning" could also deal with "programming languages".

This problem also emerges in the biomedical literature, where correctly tagging articles is important because it allows researchers to do exhaustive reviews of the literature. At the National Library of Medicine, a number of professional annotators go over each article that gets indexed in PubMed to associate each with the relevant terms from MeSH, a collection of roughly 28k tags. This is a time-consuming process and the annotators typically have a one year lag

between archiving and tagging. Machine learning can be used here to provide provisional tags until each article can have a proper manual review. Indeed, for several years, the BioASQ organization has [hosted a competition](#) to do precisely this.

## Search and ranking

Sometimes we don't just want to assign each example to a bucket or to a real value. In the field of information retrieval, we want to impose a ranking on a set of items. Take web search for example, the goal is less to determine whether a particular page is relevant for a query, but rather, which one of the plethora of search results should be displayed for the user. We really care about the ordering of the relevant search results and our learning algorithm needs to produce ordered subsets of elements from a larger set. In other words, if we are asked to produce the first 5 letters from the alphabet, there is a difference between returning 

A	B	C	D	E
---	---	---	---	---

 and 

C	A	B	E	D
---	---	---	---	---

. Even if the result set is the same, the ordering within the set matters nonetheless.

A possible solution to this problem is to score every element in the set of possible sets with a relevance score and then retrieve the top-rated elements. [PageRank](#) is an early example of such a relevance score. One of the peculiarities is that it didn't depend on the actual query. Instead, it simply helped to order the results that contained the query terms. Nowadays search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire conferences devoted to this subject.

## Recommender systems

Recommender systems are another problem setting that is related to search and ranking. The problems are similar insofar as the goal is to display a set of relevant items to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a SciFi fan and the results page for a connoisseur of Woody Allen comedies might differ significantly.

Such problems occur, e.g. for movie, product or music recommendation. In some cases, customers will provide explicit details about how much they liked the product (e.g. Amazon product reviews). In some other cases, they might simply provide feedback if they are dissatisfied with the result (skipping titles on a playlist). Generally, such systems strive to estimate some score  $y_{ij}$  as a function of user  $u_i$  and object  $o_j$ . The objects  $o_j$  with the largest scores  $y_{ij}$  are then used as a recommendation. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. The following image is an example of deep learning books recommended by Amazon based on personalization algorithms tuned to the author's preferences.





Best Seller



## Sequence Learning

So far we've looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. Before we considered predicting home prices from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. We also discussed mapping from an image (of fixed dimension), to the predicted probabilities that it belongs to each of a fixed number of classes, or taking a user ID and a product ID, and predicting a star rating. In these cases, once we feed our fixed-length input into the model to generate an output, the model immediately forgets what it just saw.

This might be fine if our inputs truly all have the same dimensions and if successive inputs truly have nothing to do with each other. But how would we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what's going on in each frame might be much stronger if we take into account the previous or succeeding frames. Same goes for language. One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language.

These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely wouldn't want this model to throw away everything it knows about the patient history each hour, and just make its predictions based on the most recent measurements.

These problems are among the more exciting applications of machine learning and they are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both!). These latter problems are sometimes referred to as `seq2seq` problems. Language translation is a `seq2seq` problem. Transcribing text from spoken speech is also a `seq2seq` problem. While it is impossible to consider all types of sequence transformations, a number of special cases are worth mentioning:

## Tagging and Parsing

This involves annotating a text sequence with attributes. In other words, the number of inputs and outputs is essentially the same. For instance, we might want to know where the verbs and subjects are. Alternatively, we might want to know which words are the named entities. In general, the goal is to decompose and annotate text based on structural and grammatical assumptions to get some annotation. This sounds more complex than it actually is. Below is a very simple example of annotating a sentence with tags indicating which words refer to named entities.

Tom
Ent

## Automatic Speech Recognition

With speech recognition, the input sequence  $x$  is the sound of a speaker, and the output  $y$  is the textual transcript of what the speaker said. The challenge is that there are many more audio frames (sound is typically sampled at 8kHz or 16kHz) than text, i.e. there is no 1:1 correspondence between audio and text, since thousands of samples correspond to a single spoken word. These are `seq2seq` problems where the output is much shorter than the input.



## Text to Speech

Text to Speech (TTS) is the inverse of speech recognition. In other words, the input  $x$  is text and the output  $y$  is an audio file. In this case, the output is *much longer* than the input. While it is easy for *humans* to recognize a bad audio file, this isn't quite so trivial for computers.

## Machine Translation

Unlike the previous cases where the order of the inputs was preserved, in machine translation, order inversion can be vital. In other words, while we are still converting one sequence into another, neither the number of inputs and outputs nor the order of corresponding data points are assumed to be the same. Consider the following illustrative example of the obnoxious tendency of Germans (*Alex writing here*) to place the verbs at the end of sentences.

German	Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English	Did you already check out this excellent tutorial?
Wrong alignment	Did you yourself already this excellent tutorial looked-at?

A number of related problems exist. For instance, determining the order in which a user reads a webpage is a two-dimensional layout analysis problem. Likewise, for dialogue problems, we need to take world-knowledge and prior state into account. This is an active area of research.

## Unsupervised learning

All the examples so far were related to *Supervised Learning*, i.e. situations where we feed the model a bunch of examples and a bunch of *corresponding target values*. You could think of supervised learning as having an extremely specialized job and an extremely anal boss. The boss stands over your shoulder and tells you exactly what to do in every situation until you learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, it's easy to please this boss. You just recognize the pattern as quickly as possible and imitate their actions.

In a completely opposite way, it could be frustrating to work for a boss who has no idea what they want you to do. However, if you plan to be a data scientist, you had better get used to it. The boss might just hand you a giant dump of data and tell you to *do some data science with it!* This sounds vague because it is. We call this class of problems *unsupervised learning*, and the type and number of questions we could ask is limited only by our creativity. We will address a number of unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the questions you might ask:

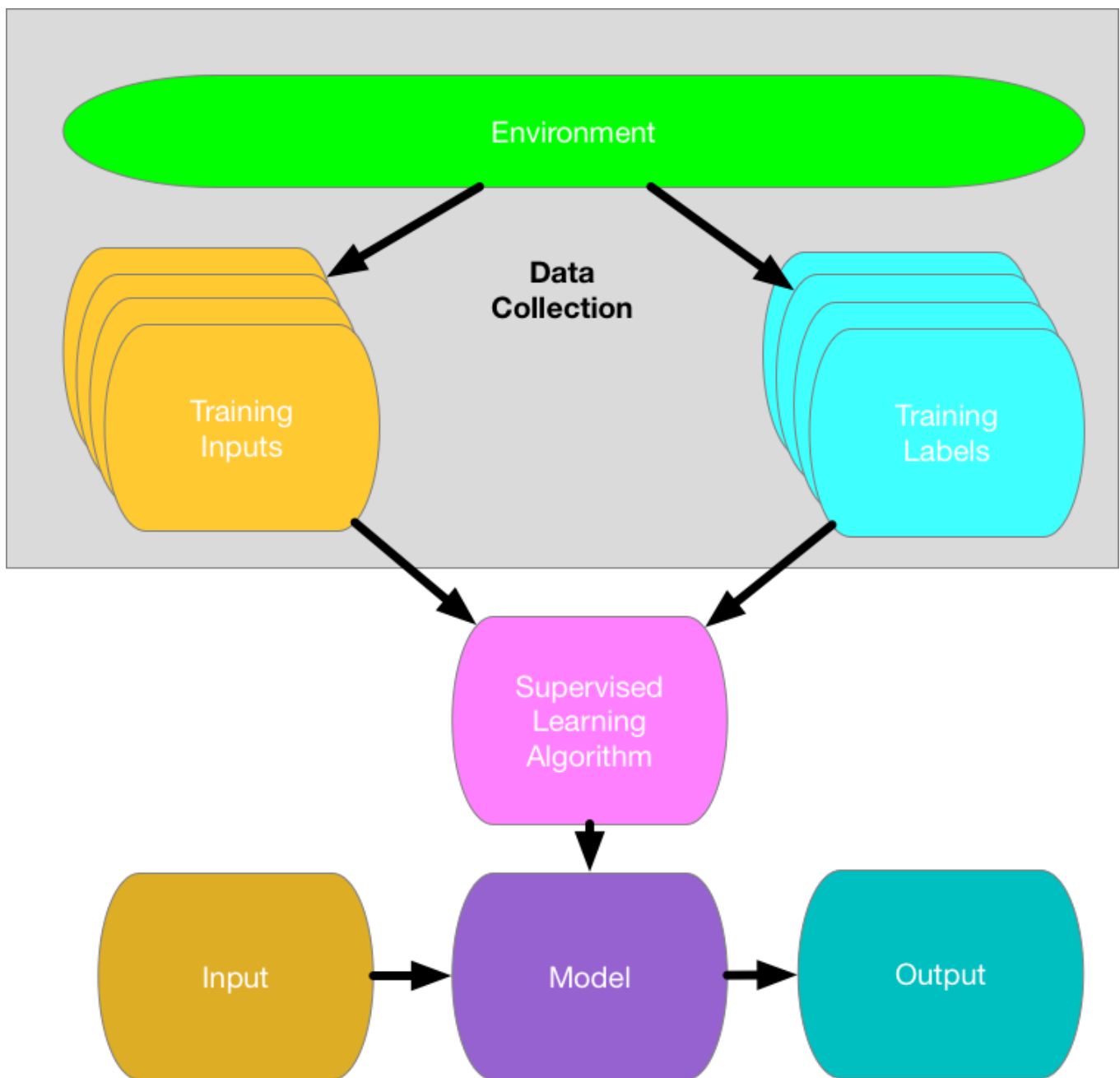
- Can we find a small number of prototypes that accurately summarize the data? Given a set of photos, can we group them into landscape photos, pictures of dogs, babies, cats, mountain peaks, etc.? Likewise, given a collection of users' browsing activity, can we group them into users with similar behavior? This problem is typically known as **clustering**.
- Can we find a small number of parameters that accurately capture the relevant properties of the data? The trajectories of a ball are quite well described by velocity, diameter, and mass of

the ball. Tailors have developed a small number of parameters that describe human body shape fairly accurately for the purpose of fitting clothes. These problems are referred to as **subspace estimation** problems. If the dependence is linear, it is called **principal component analysis**.

- Is there a representation of (arbitrarily structured) objects in Euclidean space (i.e. the space of vectors in  $\mathbb{R}^n$ ) such that symbolic properties can be well matched? This is called **representation learning** and it is used to describe entities and their relations, such as Rome - Italy + France = Paris.
- Is there a description of the root causes of much of the data that we observe? For instance, if we have demographic data about house prices, pollution, crime, location, education, salaries, etc., can we discover how they are related simply based on empirical data? The field of **directed graphical models** and **causality** deals with this.
- An important and exciting recent development is **generative adversarial networks**. They are basically a procedural way of synthesizing data. The underlying statistical mechanisms are tests to check whether real and fake data are the same. We will devote a few notebooks to them.

## Interacting with an environment

So far, we haven't discussed where data actually comes from, or what actually *happens* when a machine learning model generates an output. That's because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, we grab a big pile of data up front, then do our pattern recognition without ever interacting with the environment again. Because all of the learning takes place after the algorithm is disconnected from the environment, this is called *offline learning*. For supervised learning, the process looks like this:



This simplicity of offline learning has its charms. The upside is we can worry about pattern recognition in isolation without these other problems to deal with, but the downside is that the problem formulation is quite limiting. If you are more ambitious, or if you grew up reading Asimov's Robot Series, then you might imagine artificially intelligent bots capable not only of making predictions, but of taking actions in the world. We want to think about intelligent *agents*, not just predictive *models*. That means we need to think about choosing *actions*, not just making *predictions*. Moreover, unlike predictions, actions actually impact the environment. If we want to train an intelligent agent, we must account for the way its actions might impact the future observations of the agent.

Considering the interaction with an environment that opens a whole set of new modeling questions. Does the environment:

- remember what we did previously?

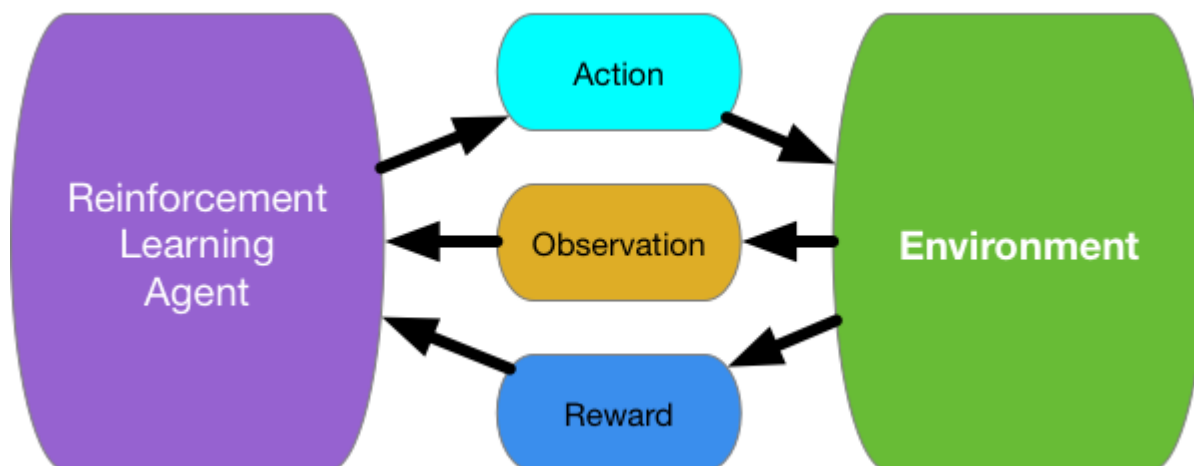
- want to help us, e.g. a user reading text into a speech recognizer?
- want to beat us, i.e. an adversarial setting like spam filtering (against spammers) or playing a game (vs an opponent)?
- not care (as in most cases)?
- have shifting dynamics (steady vs shifting over time)?

This last question raises the problem of *covariate shift*, (when training and test data are different). It's a problem that most of us have experienced when taking exams written by a lecturer, while the homeworks were composed by his TAs. We'll briefly describe reinforcement learning, and adversarial learning, two settings that explicitly consider interaction with an environment.

## Reinforcement learning

If you're interested in using machine learning to develop an agent that interacts with an environment and takes actions, then you're probably going to wind up focusing on *reinforcement learning* (RL). This might include applications to robotics, to dialogue systems, and even to developing AI for video games. *Deep reinforcement learning* (DRL), which applies deep neural networks to RL problems, has surged in popularity. The breakthrough [deep Q-network that beat humans at Atari games using only the visual input](#), and the [AlphaGo program that dethroned the world champion at the board game Go](#) are two prominent examples.

Reinforcement learning gives a very general statement of a problem, in which an agent interacts with an environment over a series of *time steps*. At each time step  $t$ , the agent receives some observation  $o_t$  from the environment, and must choose an action  $a_t$  which is then transmitted back to the environment. Finally, the agent receives a reward  $r_t$  from the environment. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of an RL agent is governed by a *policy*. In short, a *policy* is just a function that maps from observations (of the environment) to actions. The goal of reinforcement learning is to produce a good policy.



It's hard to overstate the generality of the RL framework. For example, we can cast any supervised learning problem as an RL problem. Say we had a classification problem. We could create an RL agent with one *action* corresponding to each class. We could then create an environment which gave a reward that was exactly equal to the loss function from the original supervised problem.

That being said, RL can also address many problems that supervised learning cannot. For example, in supervised learning we always expect that the training input comes associated with the correct label. But in RL, we don't assume that for each observation, the environment tells us the optimal action. In general, we just get some reward. Moreover, the environment may not even tell us which actions led to the reward.

Consider for example the game of chess. The only real reward signal comes at the end of the game when we either win, which we might assign a reward of 1, or when we lose, which we could assign a reward of -1. So reinforcement learners must deal with the *credit assignment problem*. The same goes for an employee who gets a promotion on October 11. That promotion likely reflects a large number of well-chosen actions over the previous year. Getting more promotions in the future requires figuring out what actions along the way led to the promotion.

Reinforcement learners may also have to deal with the problem of partial observability. That is, the current observation might not tell you everything about your current state. Say a cleaning robot found itself trapped in one of many identical closets in a house. Inferring the precise location (and thus state) of the robot might require considering its previous observations before entering the closet.

Finally, at any given point, reinforcement learners might know of one good policy, but there might be many other better policies that the agent has never tried. The reinforcement learner must constantly choose whether to *exploit* the best currently-known strategy as a policy, or to *explore* the space of strategies, potentially giving up some short-run reward in exchange for knowledge.

## MDPs, bandits, and friends

The general reinforcement learning problem is a very general setting. Actions affect subsequent observations. Rewards are only observed corresponding to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may ask too much of researchers. Moreover not every practical problem exhibits all this complexity. As a result, researchers have studied a number of *special cases* of reinforcement learning problems.

When the environment is fully observed, we call the RL problem a *Markov Decision Process* (MDP). When the state does not depend on the previous actions, we call the problem a *contextual bandit problem*. When there is no state, just a set of available actions with initially

unknown rewards, this problem is the classic *multi-armed bandit problem*.

## When *not* to use machine learning

Let's take a closer look at the idea of programming data by considering an interaction that [Joel Grus](#) reported experiencing in a [job interview](#). The interviewer asked him to code up Fizz Buzz. This is a children's game where the players count from 1 to 100 and will say 'fizz' whenever the number is divisible by 3, 'buzz' whenever it is divisible by 5, and 'fizzbuzz' whenever it satisfies both criteria. Otherwise, they will just state the number. It looks like this:

```
1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 ...
```

The conventional way to solve such a task is quite simple.

```
In [1]: res = []
        for i in range(1, 101):
            if i % 15 == 0:
                res.append('fizzbuzz')
            elif i % 3 == 0:
                res.append('fizz')
            elif i % 5 == 0:
                res.append('buzz')
            else:
                res.append(str(i))
        print(' '.join(res))

1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 17 fizz 19 buzz fizz 22 23
fizz buzz 26 fizz 28 29 fizzbuzz 31 32 fizz 34 buzz fizz 37 38 fizz buzz 41 fizz 43 44
fizzbuzz 46 47 fizz 49 buzz fizz 52 53 fizz buzz 56 fizz 58 59 fizzbuzz 61 62 fizz 64 buzz
fizz 67 68 fizz buzz 71 fizz 73 74 fizzbuzz 76 77 fizz 79 buzz fizz 82 83 fizz buzz 86
fizz 88 89 fizzbuzz 91 92 fizz 94 buzz fizz 97 98 fizz buzz
```

Needless to say, this isn't very exciting if you're a good programmer. Joel proceeded to 'implement' this problem in Machine Learning instead. For that to succeed, he needed a number of pieces:

- Data X `[1, 2, 3, 4, ...]` and labels Y `['fizz', 'buzz', 'fizzbuzz', identity]`
- Training data, i.e. examples of what the system is supposed to do. Such as `[(2, 2), (6, fizz), (15, fizzbuzz), (23, 23), (40, buzz)]`
- Features that map the data into something that the computer can handle more easily, e.g. `x -> [(x % 3), (x % 5), (x % 15)]`. This is optional but helps a lot if you have it.

Armed with this, Joel wrote a classifier in TensorFlow ([code](#)). The interviewer was nonplussed ... and the classifier didn't have perfect accuracy.



Quite obviously, this is silly. Why would you go through the trouble of replacing a few lines of Python with something much more complicated and error prone? However, there are many cases where a simple Python script simply does not exist, yet a 3-year-old child will solve the problem perfectly. Fortunately, this is precisely where machine learning comes to the rescue.

## Conclusion

Machine Learning is vast. We cannot possibly cover it all. On the other hand, neural networks are simple and only require elementary mathematics. So let's get started.

## Next

[Manipulate data the MXNet way with NDArray](#)

For whinges or inquiries, [open an issue on GitHub](#).

# Manipulate data the MXNet way with `ndarray`

It's impossible to get anything done if we can't manipulate data. Generally, there are two important things we need to do with: (i) acquire it! and (ii) process it once it's inside the computer. There's no point in trying to acquire data if we don't even know how to store it, so let's get our hands dirty first by playing with synthetic data.

We'll start by introducing NDArrays, MXNet's primary tool for storing and transforming data. If you've worked with NumPy before, you'll notice that NDArrays are, by design, similar to NumPy's multi-dimensional array. However, they confer a few key advantages. First, NDArrays support asynchronous computation on CPU, GPU, and distributed cloud architectures. Second, they provide support for automatic differentiation. These properties make NDArray an ideal library for machine learning, both for researchers and engineers launching production systems.

## Getting started

In this chapter, we'll get you going with the basic functionality. Don't worry if you don't understand any of the basic math, like element-wise operations or normal distributions. In the next two chapters we'll take another pass at NDArray, teaching you both the math you'll need and how to realize it in code.

To get started, let's import `mxnet`. We'll also import `ndarray` from `mxnet` for convenience. We'll make a habit of setting a random seed so that you always get the same results that we do.

```
In [2]: import mxnet as mx
        from mxnet import nd
        mx.random.seed(1)
```

Next, let's see how to create an NDArray, without any values initialized. Specifically, we'll create a 2D array (also called a *matrix*) with 3 rows and 4 columns.

```
In [3]: x = nd.empty((3, 4))
        print(x)
```

```
[[ 0.00000000e+00  0.00000000e+00  3.23540399e+21  4.58070455e-41]
 [ 1.38654559e-38  0.00000000e+00  5.56431345e+19  4.58070455e-41]
 [ 2.00971939e-37  0.00000000e+00  6.34710382e+23  4.58070455e-41]]
<NDArray 3x4 @cpu(0)>
```

The `empty` method just grabs some memory and hands us back a matrix without setting the values of any of its entries. This means that the entries can have any form of values, including very big ones! But typically, we'll want our matrices initialized. Commonly, we want a matrix of all zeros.

```
In [4]: x = nd.zeros((3, 5))  
x
```

```
Out[4]:  
[[ 0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.]  
 [ 0.  0.  0.  0.  0.]]  
<NDArray 3x5 @cpu(0)>
```

Similarly, `ndarray` has a function to create a matrix of all ones.

```
In [5]: x = nd.ones((3, 4))  
x
```

```
Out[5]:  
[[ 1.  1.  1.  1.]  
 [ 1.  1.  1.  1.]  
 [ 1.  1.  1.  1.]]  
<NDArray 3x4 @cpu(0)>
```

Often, we'll want to create arrays whose values are sampled randomly. This is especially common when we intend to use the array as a parameter in a neural network. In this snippet, we initialize with values drawn from a standard normal distribution with zero mean and unit variance.

```
In [6]: y = nd.random_normal(0, 1, shape=(3, 4))  
y
```

```
Out[6]:  
[[-0.67765152  0.03629481  0.10073948 -0.49024421]  
 [ 0.57595438 -0.95017916 -0.3469252   0.03751944]  
 [-0.22134334 -0.72984636 -1.80471897 -2.04010558]]  
<NDArray 3x4 @cpu(0)>
```

As in NumPy, the dimensions of each NDArray are accessible via the `.shape` attribute.

```
In [7]: y.shape
```

```
Out[7]: (3, 4)
```

We can also query its size, which is equal to the product of the components of the shape. Together with the precision of the stored values, this tells us how much memory the array occupies.

```
In [8]: y.size
```

```
Out[8]: 12
```

## Operations

NDArray supports a large number of standard mathematical operations. Such as element-wise addition:

```
In [9]: x + y
```

```
Out[9]:
[[ 0.32234848  1.03629482  1.10073948  0.50975579]
 [ 1.57595444  0.04982084  0.6530748  1.03751945]
 [ 0.77865666  0.27015364 -0.80471897 -1.04010558]]
<NDArray 3x4 @cpu(0)>
```

Multiplication:

```
In [10]: x * y
```

```
Out[10]:
[[-0.67765152  0.03629481  0.10073948 -0.49024421]
 [ 0.57595438 -0.95017916 -0.3469252  0.03751944]
 [-0.22134334 -0.72984636 -1.80471897 -2.04010558]]
<NDArray 3x4 @cpu(0)>
```

And exponentiation:

```
In [11]: nd.exp(y)
```

```
Out[11]:
[[ 0.50780815  1.03696156  1.1059885  0.61247683]
 [ 1.77882743  0.38667175  0.70685822  1.03823221]
 [ 0.80144149  0.48198304  0.16452068  0.13001499]]
<NDArray 3x4 @cpu(0)>
```

We can also grab a matrix's transpose to compute a proper matrix-matrix product.

```
In [12]: nd.dot(x, y.T)
```

```
Out[12]:
[[-1.03086138 -0.68363053 -4.79601431]
 [-1.03086138 -0.68363053 -4.79601431]
 [-1.03086138 -0.68363053 -4.79601431]]
<NDArray 3x3 @cpu(0)>
```

We'll explain these operations and present even more operators in the [linear algebra](#) chapter. But for now, we'll stick with the mechanics of working with NDArrays.

## In-place operations

In the previous example, every time we ran an operation, we allocated new memory to host its results. For example, if we write `y = x + y`, we will dereference the matrix that `y` used to point to and instead point it at the newly allocated memory. We can show this using Python's `id()` function, which tells us precisely which object a variable refers to.

```
In [13]: print('id(y):', id(y))
y = y + x
print('id(y):', id(y))

id(y): 140399741227416
id(y): 140395721280760
```

We can assign the result to a previously allocated array with slice notation, e.g., `result[:] = ...`.

```
In [14]: z = nd.zeros_like(x)
print('id(z):', id(z))
z[:] = x + y
print('id(z):', id(z))

id(z): 140395721281488
id(z): 140395721281488
```

However, `x+y` here will still allocate a temporary buffer to store the result before copying it to `z`. To make better use of memory, we can perform operations in place, avoiding temporary buffers. To do this we specify the `out` keyword argument every operator supports:

```
In [15]: nd.elemwise_add(x, y, out=z)

Out[15]:
[[ 1.32234848  2.03629494  2.10073948  1.50975585]
 [ 2.57595444  1.0498209   1.65307474  2.03751945]
 [ 1.77865672  1.27015364  0.19528103 -0.04010558]]
<NDArray 3x4 @cpu(0)>
```

If we're not planning to re-use `x`, then we can assign the result to `x` itself. There are two ways to do this in MXNet. 1. By using slice notation `x[:] = x op y` 2. By using the op-equals operators like `+=`

```
In [16]: print('id(x):', id(x))
x += y
x
print('id(x):', id(x))

id(x): 140395721278072
id(x): 140395721278072
```

## Slicing

MXNet NDArrays support slicing in all the ridiculous ways you might imagine accessing your data. Here's an example of reading the second and third rows from `x`.

```
In [17]: x[1:3]
```

```
Out[17]: [[ 2.57595444  1.0498209   1.65307474  2.03751945]
 [ 1.77865672  1.27015364  0.19528103 -0.04010558]]
<NDArray 2x4 @cpu(0)>
```

Now let's try writing to a specific element.

```
In [18]: x[1,2] = 9.0
x
```

```
Out[18]: [[ 1.32234848  2.03629494  2.10073948  1.50975585]
 [ 2.57595444  1.0498209   9.          2.03751945]
 [ 1.77865672  1.27015364  0.19528103 -0.04010558]]
<NDArray 3x4 @cpu(0)>
```

Multi-dimensional slicing is also supported.

```
In [19]: x[1:2,1:3]
```

```
Out[19]: [[ 1.0498209  9.          ]
<NDArray 1x2 @cpu(0)>
```

```
In [20]: x[1:2,1:3] = 5.0
x
```

```
Out[20]: [[ 1.32234848  2.03629494  2.10073948  1.50975585]
 [ 2.57595444  5.          5.          2.03751945]
 [ 1.77865672  1.27015364  0.19528103 -0.04010558]]
<NDArray 3x4 @cpu(0)>
```

## Broadcasting

You might wonder, what happens if you add a vector `y` to a matrix `x`? These operations, where we compose a low dimensional array `y` with a high-dimensional array `x` invoke a functionality called broadcasting. Here, the low-dimensional array is duplicated along any axis with dimension 1 to match the shape of the high dimensional array. Consider the following example.

```
In [21]: x = nd.ones(shape=(3,3))
print('x = ', x)
y = nd.arange(3)
print('y = ', y)
print('x + y = ', x + y)
```

```
x =
[[ 1.  1.  1.]
 [ 1.  1.  1.]
```

```
[ 1.  1.  1.]]
<NDArray 3x3 @cpu(0)>
y =
[ 0.  1.  2.]
<NDArray 3 @cpu(0)>
x + y =
[[ 1.  2.  3.]
 [ 1.  2.  3.]
 [ 1.  2.  3.]]
<NDArray 3x3 @cpu(0)>
```

While `y` is initially of shape (3), MXNet infers its shape to be (1,3), and then broadcasts along the rows to form a (3,3) matrix. You might wonder, why did MXNet choose to interpret `y` as a (1,3) matrix and not (3,1). That's because broadcasting prefers to duplicate along the left most axis. We can alter this behavior by explicitly giving `y` a 2D shape.

```
In [22]: y = y.reshape((3,1))
         print('y = ', y)
         print('x + y = ', x+y)
```

```
y =
[[ 0.]
 [ 1.]
 [ 2.]]
<NDArray 3x1 @cpu(0)>
x + y =
[[ 1.  1.  1.]
 [ 2.  2.  2.]
 [ 3.  3.  3.]]
<NDArray 3x3 @cpu(0)>
```

## Converting from MXNet NDArray to NumPy

Converting MXNet NDArrays to and from NumPy is easy. The converted arrays do not share memory.

```
In [23]: a = x.asnumpy()
         type(a)
```

```
Out[23]: numpy.ndarray
```

```
In [24]: y = nd.array(a)
         y
```

```
Out[24]:
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 3x3 @cpu(0)>
```

## Managing context

You might have noticed that MXNet NDArray looks almost identical to NumPy. But there are a few crucial differences. One of the key features that differentiates MXNet from NumPy is its support for diverse hardware devices.

In MXNet, every array has a context. One context could be the CPU. Other contexts might be various GPUs. Things can get even hairier when we deploy jobs across multiple servers. By assigning arrays to contexts intelligently, we can minimize the time spent transferring data between devices. For example, when training neural networks on a server with a GPU, we typically prefer for the model's parameters to live on the GPU. To start, let's try initializing an array on the first GPU.

```
In [25]: from mxnet import gpu
z = nd.ones(shape=(3,3), ctx=gpu(0))
z
```

```
Out[25]:
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 3x3 @gpu(0)>
```

Given an NDArray on a given context, we can copy it to another context by using the `copyto()` method.

```
In [26]: x_gpu = x.copyto(gpu(0))
print(x_gpu)
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 3x3 @gpu(0)>
```

The result of an operator will have the same context as the inputs.

```
In [29]: x_gpu + z
```

```
Out[29]:
[[ 2.  2.  2.]
 [ 2.  2.  2.]
 [ 2.  2.  2.]]
<NDArray 3x3 @gpu(0)>
```

If we ever want to check the context of an NDArray programmatically, we can just call its `.context` attribute.

```
In [30]: print(x_gpu.context)
print(z.context)
```

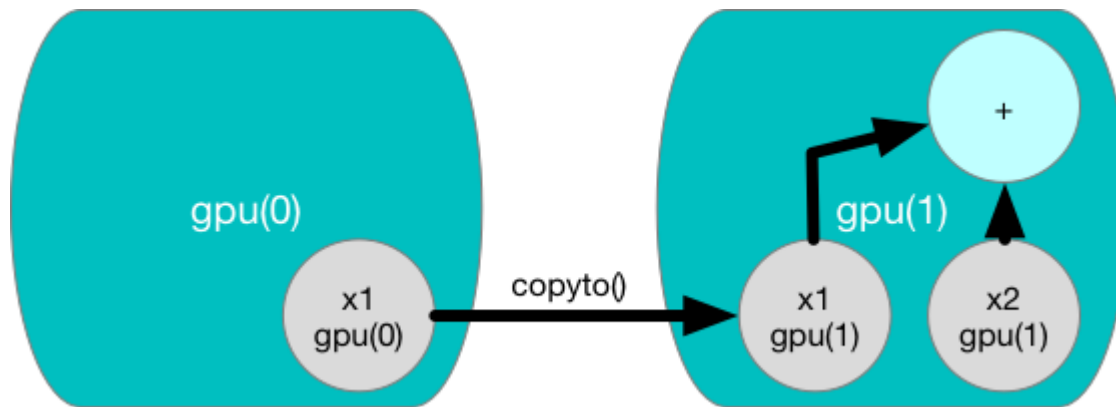
```
gpu(0)
gpu(0)
```

In order to perform an operation on two ndarrays `x1` and `x2`, we need them both to live on the same context. And if they don't already, we may need to explicitly copy data from one context to another. You might think that's annoying. After all, we just demonstrated that MXNet knows



where each NDArry lives. So why can't MXNet just automatically copy `x1` to `x2.context` and then add them?

In short, people use MXNet to do machine learning because they expect it to be fast. But transferring variables between different contexts is slow. So we want you to be 100% certain that you want to do something slow before we let you do it. If MXNet just did the copy automatically without crashing then you might not realize that you had written some slow code. We don't want you to spend your entire life on StackOverflow, so we make some mistakes impossible.



## Watch out!

Imagine that your variable `z` already lives on your second GPU (`gpu(1)`). What happens if we call `z.copyto(gpu(0))`? It will make a copy and allocate new memory, even though that variable already lives on the desired device!

There are times where depending on the environment our code is running in, two variables may already live on the same device. So we only want to make a copy if the variables currently lives on different contexts. In these cases, we can call `as_in_context()`. If the variable is already the specified context then this is a no-op.

```
In [32]: print('id(z):', id(z))
         z = z.copyto(gpu(0))
         print('id(z):', id(z))
         z = z.as_in_context(gpu(0))
         print('id(z):', id(z))
         print(z)
```

```
id(z): 140395721279640
id(z): 140395639123472
id(z): 140395639123472
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 3x3 @gpu(0)>
```

# Next

[Linear algebra](#)

For whinges or inquiries, [open an issue on GitHub](#).

# Linear algebra

Now that you can store and manipulate data, let's briefly review the subset of basic linear algebra that you'll need to understand most of the models. We'll introduce all the basic concepts, the corresponding mathematical notation, and their realization in code all in one place. If you're already confident in your basic linear algebra, feel free to skim or skip this chapter.

```
In [2]: from mxnet import nd
```

## Scalars

If you never studied linear algebra or machine learning, you're probably used to working with one number at a time. And know how to do basic things like add them together or multiply them. For example, in Palo Alto, the temperature is 52 degrees Fahrenheit. Formally, we call these values *scalars*. If you wanted to convert this value to the more sensible metric system, you'd evaluate the expression  $c = (52 - 32) * 5/9$  setting  $f$  to 52. In this equation, each of the terms 32, 5, and 9 is a scalar value. The placeholders  $c$  and  $f$  that we use are called variables and they stand in for unknown scalar values.

In mathematical notation, we represent scalars with ordinary lower cased letters ( $x$ ,  $y$ ,  $z$ ). We also denote the space of all scalars as  $\mathcal{R}$ . For expedience, we're going to punt a bit on what precisely a space is, but for now, remember that if you want to say that  $x$  is a scalar, you can simply say  $x \in \mathcal{R}$ . The symbol  $\in$  can be pronounced "in" and just denotes membership in a set.

In MXNet, we work with scalars by creating NDArrays with just one element. In this snippet, we instantiate two scalars and perform some familiar arithmetic operations with them.

```
In [3]: #####
# Instantiate two scalars
#####
x = nd.array([3.0])
y = nd.array([2.0])

#####
# Add them
#####
print('x + y = ', x + y)

#####
# Multiply them
#####
print('x * y = ', x * y)
```

```
#####
# Divide x by y
#####
print('x / y = ', x / y)

#####
# Raise x to the power y.
#####
print('x ** y = ', nd.power(x,y))

x + y =
[ 5.]
<NDArray 1 @cpu(0)>
x * y =
[ 6.]
<NDArray 1 @cpu(0)>
x / y =
[ 1.5]
<NDArray 1 @cpu(0)>
x ** y =
[ 9.]
<NDArray 1 @cpu(0)>
```

We can convert any NDArray to a Python float by calling its `asscalar` method

```
In [4]: x.asscalar()

Out[4]: 3.0
```

## Vectors

You can think of a vector as simply a list of numbers, for example `[1.0, 3.0, 4.0, 2.0]`. Each of the numbers in the vector consists of a single scalar value. We call these values the *entries* or *components* of the vector. Often, we're interested in vectors whose values hold some real-world significance. For example, if we're studying the risk that loans default, we might associate each applicant with a vector whose components correspond to their income, length of employment, number of previous defaults, etc. If we were studying the risk of heart attack in hospital patients, we might represent each patient with a vector whose components capture their most recent vital signs, cholesterol levels, minutes of exercise per day, etc. In math notation, we'll usually denote vectors as bold-faced, lower-cased letters (**u**, **v**, **w**). In MXNet, we work with vectors via 1D NDArrays with an arbitrary number of components.

```
In [5]: u = nd.arange(4)
print('u = ', u)

u =
[ 0.  1.  2.  3.]
<NDArray 4 @cpu(0)>
```

We can refer to any element of a vector by using a subscript. For example, we can refer to the 4th element of **u** by  $u_4$ . Note that the element  $u_4$  is a scalar, so we don't bold-face the font when referring to it. In code, we access any element  $i$  by indexing into the `NDArray`.

```
In [6]: u[3]
```

```
Out[6]: [ 3.]  
<NDArray 1 @cpu(0)>
```

## Length, dimensionality, and, shape

A vector is just an array of numbers. And just as every array has a length, so does every vector. In math notation, if we want to say that a vector  $x$  consists of  $n$  real-valued scalars, we can express this as  $\mathbf{x} \in \mathcal{R}^n$ . The length of a vector is commonly called its *dimension*. As with an ordinary Python array, we can access the length of an NDArray by calling Python's in-built `len()` function.

```
In [7]: len(u)
```

```
Out[7]: 4
```

We can also access a vector's length via its `.shape` attribute. The shape is a tuple that lists the dimensionality of the NDArray along each of its axes. Because a vector can only be indexed along one axis, its shape has just one element.

```
In [8]: u.shape
```

```
Out[8]: (4,)
```

Note that the word dimension is overloaded and this tends to confuse people. Some use the *dimensionality* of a vector to refer to its length (the number of components). However some use the word *dimensionality* to refer to the number of axes that an array has. In this sense, a scalar *would have* 0 dimensions and a vector *would have* 1 dimension. **To avoid confusion, when we say \*2D\* array or \*3D\* array, we mean an array with 2 or 3 axes respectively. But if we say *\*math:~n~-dimensional\* vector*, we mean a vector of length *\*math:~n~\**.**

```
In [ ]: a = 2  
x = nd.array([1,2,3])  
y = nd.array([10,20,30])  
print(a * x)  
print(a * x + y)
```

## Matrices

Just as vectors generalize scalars from order 0 to order 1, matrices generalize vectors from 1D to 2D. Matrices, which we'll denote with capital letters ( $A$ ,  $B$ ,  $C$ ), are represented in code as arrays with 2 axes.

Visually, we can draw a matrix as a table, where each entry  $a_{ij}$  belongs to the  $i$ -th row and  $j$ -th column.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

We can create a matrix with  $n$  rows and  $m$  columns in MXNet by specifying a shape with two components `(n,m)` when calling any of our favorite functions for instantiating an `ndarray` such as `ones`, or `zeros`.

```
In [10]: A = nd.zeros((5,4))
A
```

```
Out[10]:
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
<NDArray 5x4 @cpu(0)>
```

We can also reshape any 1D array into a 2D ndarray by calling `ndarray`'s `reshape` method and passing in the desired shape. Note that the product of shape components `n * m` must be equal to the length of the original vector.

```
In [12]: x = nd.arange(20)
A = x.reshape((5, 4))
A
```

```
Out[12]:
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]]
<NDArray 5x4 @cpu(0)>
```

Matrices are useful data structures: they allow us to organize data that has different modalities of variation. For example, returning to the example of medical data, rows in our matrix might correspond to different patients, while columns might correspond to different attributes.

We can access the scalar elements  $a_{ij}$  of a matrix  $A$  by specifying the indices for the row ( $i$ ) and column ( $j$ ) respectively. Let's grab the element  $a_{2,3}$  from the random matrix we initialized above.

```
In [13]: print('A[2, 3] = ', A[2, 3])

A[2, 3] =
[ 11.]
<NDArray 1 @cpu(0)>
```

We can also grab the vectors corresponding to an entire row  $\mathbf{a}_{i,:}$  or a column  $\mathbf{a}_{:,j}$ .

```
In [14]: print('row 2', A[2, :])
print('column 3', A[:, 3])

row 2
[ 8.  9. 10. 11.]
<NDArray 4 @cpu(0)>
column 3
[ 3.  7. 11. 15. 19.]
<NDArray 5 @cpu(0)>
```

We can transpose the matrix through `T`. That is, if  $B = A^T$ , then  $b_{ij} = a_{ji}$  for any  $i$  and  $j$ .

```
In [15]: A.T

Out[15]:
[[ 0.  4.  8. 12. 16.]
 [ 1.  5.  9. 13. 17.]
 [ 2.  6. 10. 14. 18.]
 [ 3.  7. 11. 15. 19.]]
<NDArray 4x5 @cpu(0)>
```

## Tensors

Just as vectors generalize scalars, and matrices generalize vectors, we can actually build data structures with even more axes. Tensors give us a generic way of discussing arrays with an arbitrary number of axes. Vectors, for example, are first-order tensors, and matrices are second-order tensors.

Using tensors will become more important when we start working with images, which arrive as 3D data structures, with axes corresponding to the height, width, and the three (RGB) color channels. But in this chapter, we're going to skip past and make sure you know the basics.

```
In [16]: X = nd.arange(24).reshape((2, 3, 4))
print('X.shape =', X.shape)
print('X =', X)

X.shape = (2, 3, 4)
X =
[[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]

 [[ 12. 13. 14. 15.]
 [ 16. 17. 18. 19.]
 [ 20. 21. 22. 23.]]]
<NDArray 2x3x4 @cpu(0)>
```

## Element-wise operations

Oftentimes, we want to apply functions to arrays. Some of the simplest and most useful functions are the element-wise functions. These operate by performing a single scalar operation on the corresponding elements of two arrays. We can create an element-wise function from any

function that maps from the scalars to the scalars. In math notations we would denote such a function as  $f : \mathcal{R} \rightarrow \mathcal{R}$ . Given any two vectors  $\mathbf{u}$  and  $\mathbf{v}$  of *the same shape*, and the function  $f$ , we can produce a vector  $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$  by setting  $c_i \leftarrow f(u_i, v_i)$  for all  $i$ . Here, we produced the vector-valued  $F : \mathcal{R}^d \rightarrow \mathcal{R}^d$  by *lifting* the scalar function to an element-wise vector operation. In MXNet, the common standard arithmetic operators (+, -, /, \*, \*\*) have all been *lifted* to element-wise operations for identically-shaped tensors of arbitrary shape.

```
In [17]: u = nd.array([1, 2, 4, 8])
v = nd.ones_like(u) * 2
print('v =', v)
print('u + v', u + v)
print('u - v', u - v)
print('u * v', u * v)
print('u / v', u / v)
```

```
v =
[ 2.  2.  2.  2.]
<NDArray 4 @cpu(0)>
u + v
[ 3.  4.  6. 10.]
<NDArray 4 @cpu(0)>
u - v
[-1.  0.  2.  6.]
<NDArray 4 @cpu(0)>
u * v
[ 2.  4.  8. 16.]
<NDArray 4 @cpu(0)>
u / v
[ 0.5  1.  2.  4. ]
<NDArray 4 @cpu(0)>
```

We can call element-wise operations on any two tensors of the same shape, including matrices.

```
In [18]: B = nd.ones_like(A) * 3
print('B =', B)
print('A + B =', A + B)
print('A * B =', A * B)
```

```
B =
[[ 3.  3.  3.  3.]
 [ 3.  3.  3.  3.]
 [ 3.  3.  3.  3.]
 [ 3.  3.  3.  3.]
 [ 3.  3.  3.  3.]]
<NDArray 5x4 @cpu(0)>
A + B =
[[ 3.  4.  5.  6.]
 [ 7.  8.  9. 10.]
 [11. 12. 13. 14.]
 [15. 16. 17. 18.]
 [19. 20. 21. 22.]]
<NDArray 5x4 @cpu(0)>
A * B =
[[ 0.  3.  6.  9.]
 [12. 15. 18. 21.]
 [24. 27. 30. 33.]
 [36. 39. 42. 45.]
 [48. 51. 54. 57.]]
<NDArray 5x4 @cpu(0)>
```

## Basic properties of tensor arithmetic



Scalars, vectors, matrices, and tensors of any order have some nice properties that we'll often rely on. For example, as you might have noticed from the definition of an element-wise operation, given operands with the same shape, the result of any element-wise operation is a tensor of that same shape. Another convenient property is that for all tensors, multiplication by a scalar produces a tensor of the same shape. In math, given two tensors  $X$  and  $Y$  with the same shape,  $\alpha X + Y$  has the same shape. (numerical mathematicians call this the AXPY operation).

```
In [19]: a = 2
         x = nd.ones(3)
         y = nd.zeros(3)
         print(x.shape)
         print(y.shape)
         print((a * x).shape)
         print((a * x + y).shape)

(3,)
(3,)
(3,)
(3,)
```

Shape is not the the only property preserved under addition and multiplication by a scalar. These operations also preserve membership in a vector space. But we'll postpone this discussion for the second half of this chapter because it's not critical to getting your first models up and running.

## Sums and means

The next more sophisticated thing we can do with arbitrary tensors is to calculate the sum of their elements. In mathematical notation, we express sums using the  $\sum$  symbol. To express the sum of the elements in a vector  $\mathbf{u}$  of length  $d$ , we can write  $\sum_{i=1}^d u_i$ . In code, we can just call

```
nd.sum()
```

```
In [ ]: nd.sum(u)
```

We can similarly express sums over the elements of tensors of arbitrary shape. For example, the sum of the elements of an  $m \times n$  matrix  $A$  could be written  $\sum_{i=1}^m \sum_{j=1}^n a_{ij}$ .

```
In [ ]: nd.sum(A)
```

A related quantity is the *mean*, which is also called the *average*. We calculate the mean by dividing the sum by the total number of elements. With mathematical notation, we could write the average over a vector  $\mathbf{u}$  as  $\frac{1}{d} \sum_{i=1}^d u_i$  and the average over a matrix  $A$  as  $\frac{1}{n \cdot m} \sum_{i=1}^m \sum_{j=1}^n a_{ij}$ . In code, we could just call `nd.mean()` on tensors of arbitrary shape:

```
In [ ]: print(nd.mean(A))
        print(nd.sum(A) / A.size)
```

## Dot products

One of the most fundamental operations is the dot product. Given two vectors  $\mathbf{u}$  and  $\mathbf{v}$ , the dot product  $\mathbf{u}^T \mathbf{v}$  is a sum over the products of the corresponding elements:  $\mathbf{u}^T \mathbf{v} = \sum_{i=1}^d u_i \cdot v_i$ .

```
In [ ]: nd.dot(u, v)
```

Note that we can express the dot product of two vectors `nd.dot(u, v)` equivalently by performing an element-wise multiplication and then a sum:

```
In [ ]: nd.sum(u * v)
```

Dot products are useful in a wide range of contexts. For example, given a set of weights  $\mathbf{w}$ , the weighted sum of some values  $u$  could be expressed as the dot product  $\mathbf{u}^T \mathbf{w}$ . When the weights are non-negative and sum to one ( $\sum_{i=1}^d w_i = 1$ ), the dot product expresses a *weighted average*. When two vectors each have length one (we'll discuss what *length* means below in the section on norms), dot products can also capture the cosine of the angle between them.

## Matrix-vector products

Now that we know how to calculate dot products we can begin to understand matrix-vector products. Let's start off by visualizing a matrix  $A$  and a column vector  $\mathbf{x}$ .

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

We can visualize the matrix in terms of its row vectors

$$A = \begin{pmatrix} \cdots & \mathbf{a}_1^T & \cdots \\ \cdots & \mathbf{a}_2^T & \cdots \\ & \vdots & \\ \cdots & \mathbf{a}_n^T & \cdots \end{pmatrix},$$

where each  $\mathbf{a}_i^T \in \mathbb{R}^m$  is a row vector representing the  $i$ -th row of the matrix  $A$ .

Then the matrix vector product  $\mathbf{y} = A\mathbf{x}$  is simply a column vector  $\mathbf{y} \in \mathbb{R}^n$  where each entry  $y_i$  is the dot product  $\mathbf{a}_i^T \mathbf{x}$ .

$$A\mathbf{x} = \begin{pmatrix} \cdots & \mathbf{a}_1^T & \cdots \\ \cdots & \mathbf{a}_2^T & \cdots \\ & \vdots & \\ \cdots & \mathbf{a}_n^T & \cdots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1^T \mathbf{x} \\ \mathbf{a}_2^T \mathbf{x} \\ \vdots \\ \mathbf{a}_n^T \mathbf{x} \end{pmatrix}$$

So you can think of multiplication by a matrix  $A \in \mathbb{R}^{m \times n}$  as a transformation that projects vectors from  $\mathbb{R}^m$  to  $\mathbb{R}^n$ .

These transformations turn out to be quite useful. For example, we can represent rotations as multiplications by a square matrix. As we'll see in subsequent chapters, we can also use matrix-vector products to describe the calculations of each layer in a neural network.

Expressing matrix-vector products in code with `ndarray`, we use the same `nd.dot()` function as for dot products. When we call `nd.dot(A, x)` with a matrix `A` and a vector `x`, `MXNet` knows to perform a matrix-vector product. Note that the column dimension of `A` must be the same as the dimension of `x`.

```
In [ ]: nd.dot(A, u)
```

## Matrix-matrix multiplication

If you've gotten the hang of dot products and matrix-vector multiplication, then matrix-matrix multiplications should be pretty straightforward.

Say we have two matrices,  $A \in \mathbb{R}^{n \times k}$  and  $B \in \mathbb{R}^{k \times m}$ :

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{pmatrix}$$

To produce the matrix product  $C = AB$ , it's easiest to think of  $A$  in terms of its row vectors and  $B$  in terms of its column vectors:

$$A = \begin{pmatrix} \cdots & \mathbf{a}_1^T & \cdots \\ \cdots & \mathbf{a}_2^T & \cdots \\ & \vdots & \\ \cdots & \mathbf{a}_n^T & \cdots \end{pmatrix}, \quad B = \begin{pmatrix} \vdots & \vdots & \cdots & \vdots \\ \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \\ \vdots & \vdots & & \vdots \end{pmatrix}.$$

Note here that each row vector  $\mathbf{a}_i^T$  lies in  $\mathbb{R}^k$  and that each column vector  $\mathbf{b}_j$  also lies in  $\mathbb{R}^k$ .

Then to produce the matrix product  $C \in \mathbb{R}^{n \times m}$  we simply compute each entry  $c_{ij}$  as the dot product  $\mathbf{a}_i^T \mathbf{b}_j$ .

$$C = AB = \begin{pmatrix} \cdots & \mathbf{a}_1^T & \cdots \\ \cdots & \mathbf{a}_2^T & \cdots \\ & \vdots & \\ \cdots & \mathbf{a}_n^T & \cdots \end{pmatrix} \begin{pmatrix} \vdots & \vdots & \cdots & \vdots \\ \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \\ \vdots & \vdots & & \vdots \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1^T \mathbf{b}_1 & \mathbf{a}_1^T \mathbf{b}_2 & \cdots & \mathbf{a}_1^T \mathbf{b}_m \\ \mathbf{a}_2^T \mathbf{b}_1 & \mathbf{a}_2^T \mathbf{b}_2 & \cdots & \mathbf{a}_2^T \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^T \mathbf{b}_1 & \mathbf{a}_n^T \mathbf{b}_2 & \cdots & \mathbf{a}_n^T \mathbf{b}_m \end{pmatrix}$$

You can think of the matrix-matrix multiplication  $AB$  as simply performing  $m$  matrix-vector products and stitching the results together to form an  $n \times m$  matrix. Just as with ordinary dot products and matrix-vector products, we can compute matrix-matrix products in `MXNet` by using

`nd.dot()`.

```
In [20]: A = nd.ones(shape=(3, 4))
         B = nd.ones(shape=(4, 5))
         nd.dot(A, B)
```

```
Out[20]: [[ 4.  4.  4.  4.  4.]
          [ 4.  4.  4.  4.  4.]
          [ 4.  4.  4.  4.  4.]]
<NDArray 3x5 @cpu(0)>
```

## Norms

Before we can start implementing models, there's one last concept we're going to introduce. Some of the most useful operators in linear algebra are norms. Informally, they tell us how big a vector or matrix is. We represent norms with the notation  $\|\cdot\|$ . The  $\cdot$  in this expression is just a placeholder. For example, we would represent the norm of a vector  $\mathbf{x}$  or matrix  $A$  as  $\|\mathbf{x}\|$  or  $\|A\|$ , respectively.

All norms must satisfy a handful of properties: 1.  $\|\alpha A\| = |\alpha| \|A\|$  2.  $\|A + B\| \leq \|A\| + \|B\|$  3.  $\|A\| \geq 0$  4. If  $\forall i, j, a_{ij} = 0$ , then  $\|A\| = 0$

To put it in words, the first rule says that if we scale all the components of a matrix or vector by a constant factor  $\alpha$ , its norm also scales by the *absolute value* of the same constant factor. The second rule is the familiar triangle inequality. The third rule simply says that the norm must be

non-negative. That makes sense, in most contexts the smallest *size* for anything is 0. The final rule basically says that the smallest norm is achieved by a matrix or vector consisting of all zeros. It's possible to define a norm that gives zero norm to nonzero matrices, but you can't give nonzero norm to zero matrices. That's a mouthful, but if you digest it then you probably have grepped the important concepts here.

If you remember Euclidean distances (think Pythagoras' theorem) from grade school, then non-negativity and the triangle inequality might ring a bell. You might notice that norms sound a lot like measures of distance.

In fact, the Euclidean distance  $\sqrt{x_1^2 + \dots + x_n^2}$  is a norm. Specifically it's the  $\ell_2$ -norm. An analogous computation, performed over the entries of a matrix, e.g.  $\sqrt{\sum_{i,j} a_{ij}^2}$ , is called the Frobenius norm. More often, in machine learning we work with the squared  $\ell_2$  norm (notated  $\ell_2^2$ ). We also commonly work with the  $\ell_1$  norm. The  $\ell_1$  norm is simply the sum of the absolute values. It has the convenient property of placing less emphasis on outliers.

To calculate the  $\ell_2$  norm, we can just call `nd.norm()`.

```
In [ ]: nd.norm(u)
```

To calculate the L1-norm we can simply perform the absolute value and then sum over the elements.

```
In [ ]: nd.sum(nd.abs(u))
```

## Norms and objectives

While we don't want to get too far ahead of ourselves, we do want you to anticipate why these concepts are useful. In machine learning we're often trying to solve optimization problems: *Maximize* the probability assigned to observed data. *Minimize* the distance between predictions and the ground-truth observations. Assign vector representations to items (like words, products, or news articles) such that the distance between similar items is minimized, and the distance between dissimilar items is maximized. Oftentimes, these objectives, perhaps the most important component of a machine learning algorithm (besides the data itself), are expressed as norms.

## Intermediate linear algebra

If you've made it this far, and understand everything that we've covered, then honestly, you *are* ready to begin modeling. If you're feeling antsy, this is a perfectly reasonable place to move on. You already know nearly all of the linear algebra required to implement a number of many practically useful models and you can always circle back when you want to learn more.

But there's a lot more to linear algebra, even as concerns machine learning. At some point, if you plan to make a career of machine learning, you'll need to know more than we've covered so far. In the rest of this chapter, we introduce some useful, more advanced concepts.

## Basic vector properties

Vectors are useful beyond being data structures to carry numbers. In addition to reading and writing values to the components of a vector, and performing some useful mathematical operations, we can analyze vectors in some interesting ways.

One important concept is the notion of a vector space. Here are the conditions that make a vector space:

- **Additive axioms** (we assume that  $x, y, z$  are all vectors):  $x + y = y + x$  and  $(x + y) + z = x + (y + z)$  and  $0 + x = x + 0 = x$  and  $(-x) + x = x + (-x) = 0$ .
- **Multiplicative axioms** (we assume that  $x$  is a vector and  $a, b$  are scalars):  $0 \cdot x = 0$  and  $1 \cdot x = x$  and  $(ab)x = a(bx)$ .
- **Distributive axioms** (we assume that  $x$  and  $y$  are vectors and  $a, b$  are scalars):  $a(x + y) = ax + ay$  and  $(a + b)x = ax + bx$ .

## Special matrices

There are a number of special matrices that we will use throughout this tutorial. Let's look at them in a bit of detail:

- **Symmetric Matrix** These are matrices where the entries below and above the diagonal are the same. In other words, we have that  $M^T = M$ . An example of such matrices are those that describe pairwise distances, i.e.  $M_{ij} = \|x_i - x_j\|$ . Likewise, the Facebook friendship graph can be written as a symmetric matrix where  $M_{ij} = 1$  if  $i$  and  $j$  are friends and  $M_{ij} = 0$  if they are not. Note that the *Twitter* graph is asymmetric -  $M_{ij} = 1$ , i.e.  $i$  following  $j$  does not imply that  $M_{ji} = 1$ , i.e.  $j$  following  $i$ .
- **Antisymmetric Matrix** These matrices satisfy  $M^T = -M$ . Note that any arbitrary matrix can always be decomposed into a symmetric and into an antisymmetric matrix by using  $M = \frac{1}{2}(M + M^T) + \frac{1}{2}(M - M^T)$
- **Diagonally Dominant Matrix** These are matrices where the off-diagonal elements are small relative to the main diagonal elements. In particular we have that  $M_{ii} \geq \sum_{j \neq i} M_{ij}$  and

$M_{ii} \geq \sum_{j \neq i} M_{ji}$ . If a matrix has this property, we can often approximate  $M$  by its diagonal. This is often expressed as  $\text{diag}(M)$ .

- **Positive Definite Matrix** These are matrices that have the nice property where  $x^T M x > 0$  whenever  $x \neq 0$ . Intuitively, they are a generalization of the squared norm of a vector  $\|x\|^2 = x^T x$ . It is easy to check that whenever  $M = A^T A$ , this holds since there  $x^T M x = x^T A^T A x = \|Ax\|^2$ . There is a somewhat more profound theorem which states that all positive definite matrices can be written in this form.

## Conclusions

In just a few pages (or one Jupyter notebook) we've taught you all the linear algebra you'll need to understand a good chunk of neural networks. Of course there's a *lot* more to linear algebra. And a lot of that math *is* useful for machine learning. For example, matrices can be decomposed into factors, and these decompositions can reveal low-dimensional structure in real-world datasets. There are entire subfields of machine learning that focus on using matrix decompositions and their generalizations to high-order tensors to discover structure in datasets and solve prediction problems. But this book focuses on deep learning. And we believe you'll be much more inclined to learn more mathematics once you've gotten your hands dirty deploying useful machine learning models on real datasets. So while we reserve the right to introduce more math much later on, we'll wrap up this chapter here.

If you're eager to learn more about linear algebra, here are some of our favorite resources on the topic \* For a solid primer on basics, check out Gilbert Strang's book [Introduction to Linear Algebra](#) \* Zico Kolter's [Linear Algebra Reivew and Reference](#)

## Next











[Probability and statistics](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Probability and statistics

In some form or another, machine learning is all about making predictions. We might want to predict the *probability* of a patient suffering a heart attack in the next year, given their clinical history. In anomaly detection, we might want to assess how *likely* a set of readings from an airplane's jet engine would be, were it operating normally. In reinforcement learning, we want an agent to act intelligently in an environment. This means we need to think about the probability of getting a high reward under each of the available action. And when we build recommender systems we also need to think about probability. For example, if we *hypothetically* worked for a large online bookseller, we might want to estimate the probability that a particular user would buy a particular book, if prompted. For this we need to use the language of probability and statistics. Entire courses, majors, theses, careers, and even departments, are devoted to probability. So our goal here isn't to teach the whole subject. Instead we hope to get you off the ground, to teach you just enough that you know everything necessary to start building your first machine learning models and to have enough of a flavor for the subject that you can begin to explore it on your own if you wish.

We've talked a lot about probabilities so far without articulating what precisely they are or giving a concrete example. Let's get more serious by considering the problem of distinguishing cats and dogs based on photographs. This might sound simpler but it's actually a formidable challenge. To start with, the difficulty of the problem may depend on the resolution of the image.

20px	40px	80px	160px	320px
				
				

While it's easy for humans to recognize cats and dogs at 320 pixel resolution, it becomes challenging at 40 pixels and next to impossible at 20 pixels. In other words, our ability to tell cats and dogs apart at a large distance (and thus low resolution) might approach uninformed guessing. Probability gives us a formal way of reasoning about our level of certainty. If we are completely sure that the image depicts a cat, we say that the *probability* that the corresponding label  $l$  is cat, denoted  $P(l = \text{cat})$  equals 1.0. If we had no evidence to suggest that  $l = \text{cat}$  or that  $l = \text{dog}$ , then we might say that the two possibilities were equally *likely* expressing this as  $P(l = \text{cat}) = 0.5$ . If we were reasonably confident, but not sure that the image depicted a cat, we might assign a probability  $.5 < P(l = \text{cat}) < 1.0$

Now consider a second case: given some weather monitoring data, we want to predict the probability that it will rain in Taipei tomorrow. If it's summertime, the rain might come with probability .5 In both cases, we have some value of interest. And in both cases we are uncertain about the outcome. But There's a key difference between the two cases. In this first case, the image is in fact either a dog or a cat, we just don't know which. In the second case, the outcome may



actually be a random event, if you believe in such things (and most physicists do). So probability is a flexible language for reasoning about our level of certainty, and it can be applied effectively in a broad set of contexts.

## Basic probability theory

Say that we cast a die and want to know what the chance is of seeing a 1 rather than another digit. If the die is fair, all six outcomes  $\mathcal{X} = \{1, \dots, 6\}$  are equally likely to occur, hence we would see a 1 in 1 out of 6 cases. Formally we state that 1 occurs with probability  $\frac{1}{6}$ .

For a real die that we receive from a factory, we might not know those proportions and we would need to check whether it is tainted. The only way to investigate the die is by casting it many times and recording the outcomes. For each cast of the die, we'll observe a value  $\{1, 2, \dots, 6\}$ . Given these outcomes, we want to investigate the probability of observing each outcome.

One natural approach for each value is to take the individual count for that value and to divide it by the total number of tosses. This gives us an *estimate* of the probability of a given event. The law of large numbers tell us that as the number of tosses grows this estimate will draw closer and closer to the true underlying probability. Before going into the details of what's going here, let's try it out.

To start, let's import the necessary packages:

```
In [66]: import mxnet as mx
        from mxnet import nd
```

Next, we'll want to be able to cast the die. In statistics we call this process of drawing examples from probability distributions *sampling*. The distribution which assigns probabilities to a number of discrete choices is called the *multinomial* distribution. We'll give a more formal definition of *distribution* later, but at a high level, think of it as just an assignment of probabilities to events. In MXNet, we can sample from the multinomial distribution via the aptly named `nd.sample_multinomial` function. The function can be called in many ways, but we'll focus on the simplest. To draw a single sample, we simply give pass in a vector of probabilities.

```
In [67]: probabilities = nd.ones(6) / 6
        nd.sample_multinomial(probabilities)
```

```
Out[67]:
[1]
<NDArray 1 @cpu(0)>
```

If you run this line (`nd.sample_multinomial(probabilities)`) a bunch of times, you'll find that you get out random values each time. As with estimating the fairness of a die, we often want to generate many samples from the same distribution. It would be really slow to do this with a Python `for` loop, so `sample_multinomial` supports drawing multiple samples at once, returning an array of independent samples in any shape we might desire.

```
In [68]: print(nd.sample_multinomial(probabilities, shape=(10)))
        print(nd.sample_multinomial(probabilities, shape=(5,10)))
```

```
[4 4 2 1 1 1 5 4 0 4]
<NDArray 10 @cpu(0)>
```

```
[[2 3 0 4 1 2 4 0 3 0]
 [3 3 4 0 0 0 5 5 3 5]
 [3 0 0 0 5 5 0 1 1 1]
 [5 4 0 0 2 1 4 5 3 3]]
```

```
[4 3 1 2 2 4 2 0 1 5]]
<NDArray 5x10 @cpu(0)>
```

Now that we know how to sample rolls of a die, we can simulate 1000 rolls.

```
In [69]: rolls = nd.sample_multinomial(probabilities, shape=(1000))
```

We can then go through and count, after each of the 1000 rolls, how many times each number was rolled.

```
In [70]: counts = nd.zeros((6,1000))
totals = nd.zeros(6)
for i, roll in enumerate(rolls):
    totals[int(roll.asscalar())] += 1
    counts[:, i] = totals
```

To start, we can inspect the final tally at the end of 1000 rolls.

```
In [71]: totals / 1000
```

```
Out[71]: [ 0.168      0.15800001  0.15099999  0.155      0.18799999  0.18000001]
<NDArray 6 @cpu(0)>
```

As you can see, the lowest estimated probability for any of the numbers is about .15 and the highest estimated probability is 0.188. Because we generated the data from a fair die, we know that each number actually has probability of 1/6, roughly .167, so these estimates are pretty good. We can also visualize how these probabilities converge over time towards reasonable estimates.

To start let's take a look at the `counts` array which has shape `(6, 1000)`. For each time step (out of 1000), counts, says how many times each of the numbers has shown up. So we can normalize each  $j$ -th column of the counts vector by the number of tosses to give the `current` estimated probabilities at that time. The counts object looks like this:

```
In [72]: counts
```

```
Out[72]: [[ 0.  0.  0. ..., 168. 168. 168.]
 [ 0.  0.  0. ..., 157. 158. 158.]
 [ 0.  0.  0. ..., 151. 151. 151.]
 [ 0.  0.  0. ..., 155. 155. 155.]
 [ 0.  0.  1. ..., 188. 188. 188.]
 [ 1.  2.  2. ..., 179. 179. 180.]]
<NDArray 6x1000 @cpu(0)>
```

Normalizing by the number of tosses, we get:

```
In [73]: x = nd.arange(1000).reshape((1,1000)) + 1
estimates = counts / x
print(estimates[:,0])
print(estimates[:,1])
print(estimates[:,100])
```

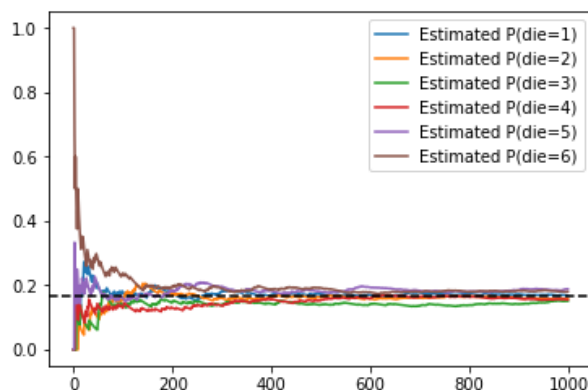
```
[ 0.  0.  0.  0.  0.  1.]
<NDArray 6 @cpu(0)>
```

```
[ 0.  0.  0.  0.  0.  1.]
<NDArray 6 @cpu(0)>
```

```
[ 0.17821783  0.15841584  0.13861386  0.12871288  0.15841584  0.23762377]
<NDArray 6 @cpu(0)>
```

As you can see, after the first toss of the die, we get the extreme estimate that one of the numbers will be rolled with probability 1.0 and that the others have probability 0. After 100 rolls, things already look a bit more reasonable. We can visualize this convergence by using the plotting package `matplotlib`. If you don't have it installed, now would be a good time to [install it](#).

```
In [74]: from matplotlib import pyplot as plt
plt.plot(estimates[0, :].asnumpy(), label="Estimated P(die=1)")
plt.plot(estimates[1, :].asnumpy(), label="Estimated P(die=2)")
plt.plot(estimates[2, :].asnumpy(), label="Estimated P(die=3)")
plt.plot(estimates[3, :].asnumpy(), label="Estimated P(die=4)")
plt.plot(estimates[4, :].asnumpy(), label="Estimated P(die=5)")
plt.plot(estimates[5, :].asnumpy(), label="Estimated P(die=6)")
plt.axhline(y=0.16666, color='black', linestyle='dashed')
plt.legend()
plt.show()
```



Each solid curve corresponds to one of the six values of the die and gives our estimated probability that the die turns up that value as assessed after each of the 1000 turns. The dashed black line gives the true underlying probability. As we get more data, the solid curves converge towards the true answer.

In our example of casting a die, we introduced the notion of a **random variable**. A random variable, which we denote here as  $X$  can be pretty much any quantity is not deterministic. Random variables could take one value among a set of possibilities. We denote sets with brackets, e.g.,  $\{\text{cat}, \text{dog}, \text{rabbit}\}$ . The items contained in the set are called *elements*, and we can say that an element  $x$  is *in* the set  $S$ , by writing  $x \in S$ . The symbol  $\in$  is read as “in” and denotes membership. For instance, we could truthfully say  $\text{dog} \in \{\text{cat}, \text{dog}, \text{rabbit}\}$ . When dealing with the rolls of die, we are concerned with a variable  $X \in \{1, 2, 3, 4, 5, 6\}$ .

Note that there is a subtle difference between discrete random variables, like the sides of a dice, and continuous ones, like the weight and the height of a person. There's little point in asking whether two people have exactly the same height. If we take precise enough measurements you'll find that no two people on the planet have the exact same height. In fact, if we take a fine enough measurement, you will not have the same height when you wake up and when you go to sleep. So there's no purpose in asking about the probability that some one is 2.00139278291028719210196740527486202 meters tall. The probability is 0. It makes more sense in this case to ask whether someone's height falls into a given interval, say between 1.99 and 2.01 meters. In these cases we quantify the likelihood that we see a value as a density. The height of exactly 2.0 meters has no probability, but nonzero density. Between any two different heights we have nonzero probability.

There are a few important axioms of probability that you'll want to remember:

- For any event  $z$ , the probability is never negative, i.e.  $\Pr(Z = z) \geq 0$ .

- For any two events  $Z = z$  and  $X = x$  the union is no more likely than the sum of the individual events, i.e.  $\Pr(Z = z \cup X = x) \leq \Pr(Z = z) + \Pr(X = x)$
- For any random variable, the probabilities of all the values it can take must sum to 1  

$$\sum_{i=1}^n P(Z = z_i) = 1.$$
- For any two mutually exclusive events  $Z = z$  and  $X = x$ , the probability that either happens is equal to the sum of their individual probabilities that  

$$\Pr(Z = z \cup X = x) = \Pr(Z = z) + \Pr(X = x)$$

## Dealing with multiple random variables

Very often, we'll want consider more than one random variable at a time. For instance, we may want to model the relationship between diseases and symptoms. Given a disease and symptom, say 'flu' and 'cough', either may or may not occur in a patient with some probability. While we hope that the probability of both would be close to zero, we may want to estimate these probabilities and their relationships to each other so that we may apply our inferences to effect better medical care.

As a more complicated example, images contain millions of pixels, thus millions of random variables. And in many cases images will come with a label, identifying objects in the image. We can also think of the label as a random variable. We can even get crazy and think of all the metadata as random variables such as location, time, aperture, focal length, ISO, focus distance, camera type, etc. All of these are random variables that occur jointly. When we deal with multiple random variables, there are several quantities of interest. The first is called the joint distribution  $\Pr(A, B)$ . Given any elements  $a$  and  $b$ , the joint distribution lets us answer, what is the probability that  $A = a$  and  $B = b$  simultaneously? It might be clear that for any values  $a$  and  $b$ ,  $\Pr(A, B) \leq \Pr(A = a)$ .

This has to be the case, since for  $A$  and  $B$  to happen,  $A$  has to happen *and*  $B$  also has to happen (and vice versa). Thus  $A, B$  cannot be more likely than  $A$  or  $B$  individually. This brings us to an interesting ratio:  $0 \leq \frac{\Pr(A, B)}{\Pr(A)} \leq 1$ . We call this a **conditional probability** and denote it by  $\Pr(B|A)$ , the probability that  $B$  happens, provided that  $A$  has happened.

Using the definition of conditional probabilities, we can derive one of the most useful and celebrated equations in statistics - Bayes' theorem. It goes as follows: By construction, we have that  $\Pr(A, B) = \Pr(B|A) \Pr(A)$ . By symmetry, this also holds for  $\Pr(A, B) = \Pr(A|B) \Pr(B)$ . Solving for one of the conditional variables we get:

$$\Pr(A|B) = \frac{\Pr(B|A) \Pr(A)}{\Pr(B)}$$

This is very useful if we want to infer one thing from another, say cause and effect but we only know the properties in the reverse direction. One important operation that we need to make this work is **marginalization**, i.e., the operation of determining  $\Pr(A)$  and  $\Pr(B)$  from  $\Pr(A, B)$ . We can see that the probability of seeing  $A$  amounts to accounting for all possible choices of  $B$  and aggregating the joint probabilities over all of them, i.e.

$$\Pr(A) = \sum_{B'} \Pr(A, B') \text{ and } \Pr(B) = \sum_{A'} \Pr(A', B)$$

A really useful property to check is for **dependence** and **independence**. Independence is when the occurrence of one event does not influence the occurrence of the other. In this case  $\Pr(B|A) = \Pr(B)$ . Statisticians typically use  $A \perp B$  to express this. From Bayes Theorem it follows immediately that also  $\Pr(A|B) = \Pr(A)$ . In all other cases we call  $A$  and  $B$  dependent. For instance,

two successive rolls of a dice are independent. On the other hand, the position of a light switch and the brightness in the room are not (they are not perfectly deterministic, though, since we could always have a broken lightbulb, power failure, or a broken switch).

Let's put our skills to the test. Assume that a doctor administers an AIDS test to a patient. This test is fairly accurate and fails only with 1% probability if the patient is healthy by reporting him as diseased, and that it never fails to detect HIV if the patient actually has it. We use  $D$  to indicate the diagnosis and  $H$  to denote the HIV status. Written as a table the outcome  $\Pr(D|H)$  looks as follows:

	Patient is HIV positive	Patient is HIV negative
Test positive	1	0.01
Test negative	0	0.99

Note that the column sums are all one (but the row sums aren't), since the conditional probability needs to sum up to 1, just like the probability. Let us work out the probability of the patient having AIDS if the test comes back positive. Obviously this is going to depend on how common the disease is, since it affects the number of false alarms. Assume that the population is quite healthy, e.g.  $\Pr(\text{HIV positive}) = 0.0015$ . To apply Bayes Theorem we need to determine

$$\Pr(\text{Test positive}) = \Pr(D = 1|H = 0) \Pr(H = 0) + \Pr(D = 1|H = 1) \Pr(H = 1) = 0.01 \cdot 0.9985 + 1 \cdot 0.0015 = 0.011485$$

Hence we get  $\Pr(H = 1|D = 1) = \frac{\Pr(D=1|H=1) \Pr(H=1)}{\Pr(D=1)} = \frac{1 \cdot 0.0015}{0.011485} = 0.131$ , in other words, there's only a 13.1% chance that the patient actually has AIDS, despite using a test that is 99% accurate! As we can see, statistics can be quite counterintuitive.

## Conditional independence

What should a patient do upon receiving such terrifying news? Likely, he/she would ask the physician to administer another test to get clarity. The second test has different characteristics (it isn't as good as the first one).

	Patient is HIV positive	Patient is HIV negative
Test positive	0.98	0.03
Test negative	0.02	0.97

Unfortunately, the second test comes back positive, too. Let us work out the requisite probabilities to invoke Bayes' Theorem.

- $\Pr(D_1 = 1 \text{ and } D_2 = 1|H = 0) = 0.01 \cdot 0.03 = 0.0001$
- $\Pr(D_1 = 1 \text{ and } D_2 = 1|H = 1) = 1 \cdot 0.98 = 0.98$
- $\Pr(D_1 = 1 \text{ and } D_2 = 1) = 0.0001 \cdot 0.9985 + 0.98 \cdot 0.0015 = 0.00156985$
- $\Pr(H = 1|D_1 = 1 \text{ and } D_2 = 1) = \frac{0.98 \cdot 0.0015}{0.00156985} = 0.936$

That is, the second test allowed us to gain much higher confidence that not all is well. Despite the second test being considerably less accurate than the first one, it still improved our estimate quite a bit. *Why couldn't we just run the first test a second time?* After all, the first test was more accurate. The reason is that we needed a second test that confirmed *independently* of the first test that

things were dire, indeed. In other words, we made the tacit assumption that  $\Pr(D_1, D_2|H) = \Pr(D_1|H) \Pr(D_2|H)$ . Statisticians call such random variables **conditionally independent**. This is expressed as  $D_1 \perp\!\!\!\perp D_2|H$ .

## Naive Bayes classification

Conditional independence is useful when dealing with data, since it simplifies a lot of equations. A popular algorithm is the Naive Bayes Classifier. The key assumption in it is that the attributes are all independent of each other, given the labels. In other words, we have:

$$p(x|y) = \prod_i p(x_i|y)$$

Using Bayes Theorem this leads to the classifier  $p(y|x) = \frac{\prod_i p(x_i|y)p(y)}{p(x)}$ . Unfortunately, this is still intractable, since we don't know  $p(x)$ . Fortunately, we don't need it, since we know that  $\sum_y p(y|x) = 1$ , hence we can always recover the normalization from  $p(y|x) \propto \prod_i p(x_i|y)p(y)$ . After all that math, it's time for some code to show how to use a Naive Bayes classifier for distinguishing digits on the MNIST classification dataset.

The problem is that we don't actually know  $p(y)$  and  $p(x_i|y)$ . So we need to *estimate* it given some training data first. This is what is called *training* the model. In the case of 10 possible classes we simply compute  $n_y$ , i.e. the number of occurrences of class  $y$  and then divide it by the total number of occurrences. E.g. if we have a total of 60,000 pictures of digits and digit 4 occurs 5800 times, we estimate its probability as  $\frac{5800}{60000}$ . Likewise, to get an idea of  $p(x_i|y)$  we count how many times pixel  $i$  is set for digit  $y$  and then divide it by the number of occurrences of digit  $y$ . This is the probability that that very pixel will be switched on.

```
In [76]: import numpy as np

# we go over one observation at a time (speed doesn't matter here)
def transform(data, label):
    return data.astype(np.float32)/255, label.astype(np.float32)
mnist_train = mx.gluon.data.vision.MNIST(train=True, transform=transform)
mnist_test = mx.gluon.data.vision.MNIST(train=False, transform=transform)

# Initialize the count statistics for p(y) and p(x_i|y)
# We initialize all numbers with a count of 1 to ensure that we don't get a
# division by zero. Statisticians call this Laplace smoothing.
ycount = nd.ones(shape=(10))
xcount = nd.ones(shape=(784, 10))

# Aggregate count statistics of how frequently a pixel is on (or off) for
# zeros and ones.
for data, label in mnist_train:
    x = data.reshape((784,))
    y = int(label)
    ycount[y] += 1
    xcount[:, y] += x

# normalize the probabilities p(x_i|y) (divide per pixel counts by total
# count)
for i in range(10):
    xcount[:, i] = xcount[:, i]/ycount[i]

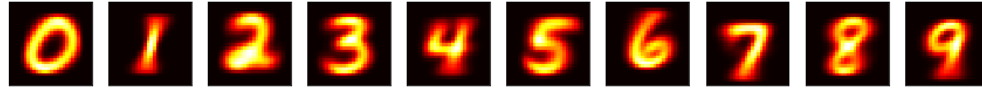
# Likewise, compute the probability p(y)
py = ycount / nd.sum(ycount)
```

Now that we computed per-pixel counts of occurrence for all pixels, it's time to see how our model behaves. Time to plot it. We show the estimated probabilities of observing a switched-on pixel. These are some mean looking digits.

```
In [77]: import matplotlib.pyplot as plt
fig, figarr = plt.subplots(1, 10, figsize=(15, 15))
for i in range(10):
```

```
figarr[i].imshow(xcount[:, i].reshape((28, 28)).asnumpy(), cmap='hot')
figarr[i].axes.get_xaxis().set_visible(False)
figarr[i].axes.get_yaxis().set_visible(False)

plt.show()
print(py)
```



```
[ 0.09871688  0.11236461  0.09930012  0.10218297  0.09736711  0.09035161
  0.09863356  0.10441593  0.09751708  0.09915014]
<NDArray 10 @cpu(0)>
```

Now we can compute the likelihoods of an image, given the model. This is statistical speak for  $p(x|y)$ , i.e. how likely it is to see a particular image under certain conditions (such as the label). Since this is computationally awkward (we might have to multiply many small numbers if many pixels have a small probability of occurring), we are better off computing its logarithm instead. That is, instead of  $p(x|y) = \prod_i p(x_i|y)$  we compute  $\log p(x|y) = \sum_i \log p(x_i|y)$ .

$$l_y := \sum_i \log p(x_i|y) = \sum_i x_i \log p(x_i = 1|y) + (1 - x_i) \log(1 - p(x_i = 1|y))$$

To avoid recomputing logarithms all the time, we precompute them for all pixels.

```
In [78]: logxcount = nd.log(xcount)
logxcountneg = nd.log(1-xcount)
logpy = nd.log(py)

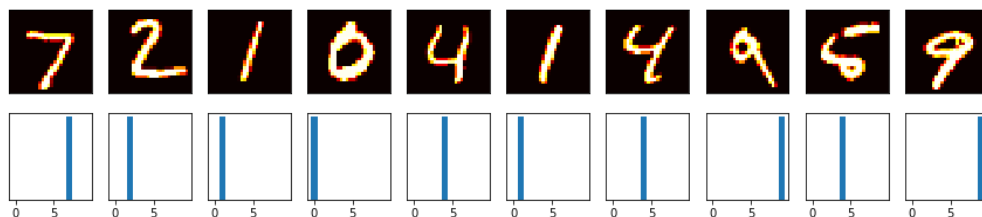
fig, figarr = plt.subplots(2, 10, figsize=(15, 3))

# show 10 images
ctr = 0
for data, label in mnist_test:
    x = data.reshape((784,))
    y = int(label)

    # we need to incorporate the prior probability p(y) since p(y|x) is
    # proportional to p(x|y) p(y)
    logpx = logpy.copy()
    for i in range(10):
        # compute the log probability for a digit
        logpx[i] = nd.dot(logxcount[:, i], x) + nd.dot(logxcountneg[:, i], 1-x)
    # normalize to prevent overflow or underflow by subtracting the largest
    # value
    logpx -= nd.max(logpx)
    # and compute the softmax using logpx
    px = nd.exp(logpx).asnumpy()
    px /= np.sum(px)

    # bar chart and image of digit
    figarr[1, ctr].bar(range(10), px)
    figarr[1, ctr].axes.get_yaxis().set_visible(False)
    figarr[0, ctr].imshow(x.reshape((28, 28)).asnumpy(), cmap='hot')
    figarr[0, ctr].axes.get_xaxis().set_visible(False)
    figarr[0, ctr].axes.get_yaxis().set_visible(False)
    ctr += 1
    if ctr == 10:
        break

plt.show()
```



As we can see, this classifier is both incompetent and overly confident of its incorrect estimates. That is, even if it is horribly wrong, it generates probabilities close to 1 or 0. Not a classifier we should use very much nowadays any longer. While Naive Bayes classifiers used to be popular in the 80s and 90s, e.g. for spam filtering, their heydays are over. The poor performance is due to the incorrect statistical assumptions that we made in our model: we assumed that each and every pixel are *independently* generated, depending only on the label. This is clearly not how humans write digits, and this wrong assumption led to the downfall of our overly naive (Bayes) classifier.

## Sampling

Random numbers are just one form of random variables, and since computers are particularly good with numbers, pretty much everything else in code ultimately gets converted to numbers anyway. One of the basic tools needed to generate random numbers is to sample from a distribution. Let's start with what happens when we use a random number generator.

```
In [79]: import random
         for i in range(10):
           print(random.random())
```

```
0.36555613019852196
0.4723632667363914
0.11762827222278505
0.28997926538432717
0.26585562005905383
0.03264057741193649
0.9300030175944551
0.7837865124001734
0.10177959804227388
0.7111418651090528
```

## Uniform Distribution

These are some pretty random numbers. As we can see, their range is between 0 and 1, and they are evenly distributed. That is, there is (actually, should be, since this is not a *real* random number generator) no interval in which numbers are more likely than in any other. In other words, the chances of any of these numbers to fall into the interval, say  $[0.2, 0.3)$  are as high as in the interval  $[.593264, .693264)$ . The way they are generated internally is to produce a random integer first, and then divide it by its maximum range. If we want to have integers directly, try the following instead. It generates random numbers between 0 and 100.

```
In [80]: for i in range(10):
         print(random.randint(1, 100))
```

```
23
66
75
34
65
23
74
32
72
25
```

What if we wanted to check that `randint` is actually really uniform. Intuitively the best strategy would be to run it, say 1 million times, count how many times it generates each one of the values and to ensure that the result is uniform.

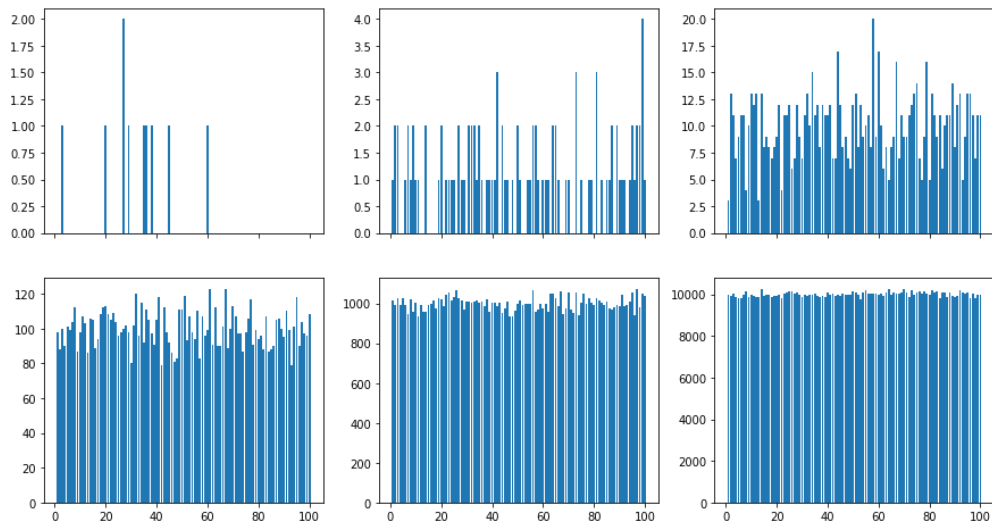
```
In [81]: import math

         counts = np.zeros(100)
         fig, axes = plt.subplots(2, 3, figsize=(15, 8), sharex=True)
         axes = axes.reshape(6)
```



```
# mangle subplots such that we can index them in a linear fashion rather than
# a 2d grid
```

```
for i in range(1, 1000001):
    counts[random.randint(0, 99)] += 1
    if i in [10, 100, 1000, 10000, 100000, 1000000]:
        axes[int(math.log10(i))-1].bar(np.arange(1, 101), counts)
plt.show()
```



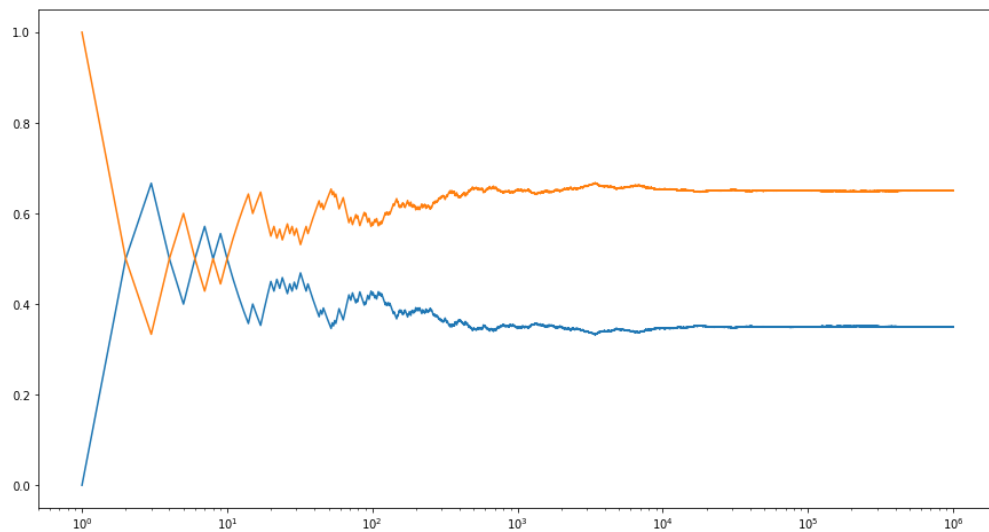
What we can see from the above figures is that the initial number of counts looks *very* uneven. If we sample fewer than 100 draws from a distribution over 100 outcomes this is pretty much expected. But even for 1000 samples there is a significant variability between the draws. What we are really aiming for is a situation where the probability of drawing a number  $x$  is given by  $p(x)$ .

## The categorical distribution

Quite obviously, drawing from a uniform distribution over a set of 100 outcomes is quite simple. But what if we have nonuniform probabilities? Let's start with a simple case, a biased coin which comes up heads with probability 0.35 and tails with probability 0.65. A simple way to sample from that is to generate a uniform random variable over  $[0, 1]$  and if the number is less than 0.35, we output heads and otherwise we generate tails. Let's try this out.

```
In [82]: # number of samples
n = 1000000
y = np.random.uniform(0, 1, n)
x = np.arange(1, n+1)
# count number of occurrences and divide by the number of total draws
p0 = np.cumsum(y < 0.35) / x
p1 = np.cumsum(y >= 0.35) / x

plt.figure(figsize=(15, 8))
plt.semilogx(x, p0)
plt.semilogx(x, p1)
plt.show()
```



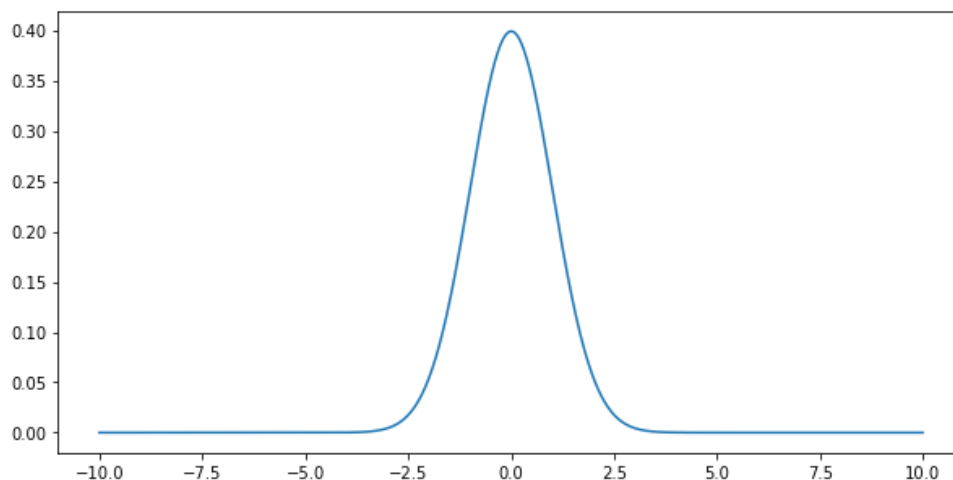
As we can see, on average this sampler will generate 35% zeros and 65% ones. Now what if we have more than two possible outcomes? We can simply generalize this idea as follows. Given any probability distribution, e.g.  $p = [0.1, 0.2, 0.05, 0.3, 0.25, 0.1]$  we can compute its cumulative distribution (python's `cumsum` will do this for you)  $F = [0.1, 0.3, 0.35, 0.65, 0.9, 1]$ . Once we have this we draw a random variable  $x$  from the uniform distribution  $U[0, 1]$  and then find the interval where  $F[i - 1] \leq x < F[i]$ . We then return  $i$  as the sample. By construction, the chances of hitting interval  $[F[i - 1], F[i])$  has probability  $p(i)$ .

Note that there are many more efficient algorithms for sampling than the one above. For instance, binary search over  $F$  will run in  $O(\log n)$  time for  $n$  random variables. There are even more clever algorithms, such as the [Alias Method](#) to sample in constant time, after  $O(n)$  preprocessing.

## The normal distribution

The normal distribution (aka the Gaussian distribution) is given by  $p(x) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}x^2)$ . Let's plot it to get a feel for it.

```
In [83]: x = np.arange(-10, 10, 0.01)
p = (1/math.sqrt(2 * math.pi)) * np.exp(-0.5 * x**2)
plt.figure(figsize=(10, 5))
plt.plot(x, p)
plt.show()
```



Sampling from this distribution is a lot less trivial. First off, the support is infinite, that is, for any  $x$  the density  $p(x)$  is positive. Secondly, the density is nonuniform. There are many tricks for sampling from it - the key idea in all algorithms is to stratify  $p(x)$  in such a way as to map it to the uniform distribution  $U[0, 1]$ . One way to do this is with the probability integral transform.

Denote by  $F(x) = \int_{-\infty}^x p(z)dz$  the cumulative distribution function (CDF) of  $p$ . This is in a way the continuous version of the cumulative sum that we used previously. In the same way we can now define the inverse map  $F^{-1}(\xi)$ , where  $\xi$  is drawn uniformly. Unlike previously where we needed to find the correct interval for the vector  $F$  (i.e. for the piecewise constant function), we now invert the function  $F(x)$ .

In practice, this is slightly more tricky since inverting the CDF is hard in the case of a Gaussian. It turns out that the *twodimensional* integral is much easier to deal with, thus yielding two normal random variables than one, albeit at the price of two uniformly distributed ones. For now, suffice it to say that there are built-in algorithms to address this.

The normal distribution has yet another desirable property. In a way all distributions converge to it, if we only average over a sufficiently large number of draws from any other distribution. To understand this in a bit more detail, we need to introduce three important things: expected values, means and variances.

- The expected value  $\mathbb{E}_{x \sim p(x)}[f(x)]$  of a function  $f$  under a distribution  $p$  is given by the integral  $\int_x p(x)f(x)dx$ . That is, we average over all possible outcomes, as given by  $p$ .
- A particularly important expected value is that for the function  $f(x) = x$ , i.e.  $\mu := \mathbb{E}_{x \sim p(x)}[x]$ . It provides us with some idea about the typical values of  $x$ .
- Another important quantity is the variance, i.e. the typical deviation from the mean  $\sigma^2 := \mathbb{E}_{x \sim p(x)}[(x - \mu)^2]$ . Simple math shows (check it as an exercise) that  $\sigma^2 = \mathbb{E}_{x \sim p(x)}[x^2] - \mathbb{E}_{x \sim p(x)}^2[x]$ .

The above allows us to change both mean and variance of random variables. Quite obviously for some random variable  $x$  with mean  $\mu$ , the random variable  $x + c$  has mean  $\mu + c$ . Moreover,  $\gamma x$  has the variance  $\gamma^2 \sigma^2$ . Applying this to the normal distribution we see that one with mean  $\mu$  and variance  $\sigma^2$  has the form  $p(x) = \frac{1}{\sqrt{2\sigma^2\pi}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right)$ . Note the scaling factor  $\frac{1}{\sigma}$  - it arises from the fact that if we stretch the distribution by  $\sigma$ , we need to lower it by  $\frac{1}{\sigma}$  to retain the same probability mass (i.e. the weight under the distribution always needs to integrate out to 1).

Now we are ready to state one of the most fundamental theorems in statistics, the [Central Limit Theorem](#). It states that for sufficiently well-behaved random variables, in particular random variables with well-defined mean and variance, the sum tends toward a normal distribution. To get some idea, let's repeat the experiment described in the beginning, but now using random variables with integer values of  $\{0, 1, 2\}$ .

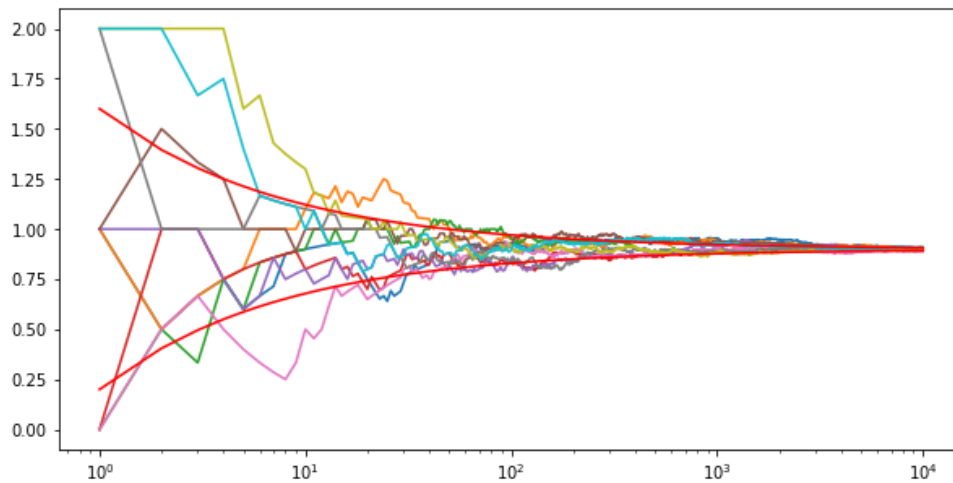
```
In [84]: # generate 10 random sequences of 10,000 random normal variables N(0,1)
tmp = np.random.uniform(size=(10000,10))
x = 1.0 * (tmp > 0.3) + 1.0 * (tmp > 0.8)
mean = 1 * 0.5 + 2 * 0.2
variance = 1 * 0.5 + 4 * 0.2 - mean**2
print('mean {}, variance {}'.format(mean, variance))
# cumulative sum and normalization
y = np.arange(1,10001).reshape(10000,1)
z = np.cumsum(x,axis=0) / y

plt.figure(figsize=(10,5))
for i in range(10):
    plt.semilogx(y,z[:,i])

plt.semilogx(y,(variance**0.5) * np.power(y,-0.5) + mean,'r')
```

```
plt.semilogx(y, -(variance**0.5) * np.power(y, -0.5) + mean, 'r')
plt.show()
```

mean 0.9, variance 0.49



This looks very similar to the initial example, at least in the limit of averages of large numbers of variables. This is confirmed by theory. Denote by mean and variance of a random variable the quantities

$$\mu[p] := \mathbb{E}_{x \sim p(x)}[x] \text{ and } \sigma^2[p] := \mathbb{E}_{x \sim p(x)}[(x - \mu[p])^2]$$

Then we have that  $\lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} \sum_{i=1}^n \frac{x_i - \mu}{\sigma} \rightarrow \mathcal{N}(0, 1)$ . In other words, regardless of what we started out with, we will always converge to a Gaussian. This is one of the reasons why Gaussians are so popular in statistics.

## More distributions

Many more useful distributions exist. We recommend consulting a statistics book or looking some of them up on Wikipedia for further detail.

- **Multinomial Distribution** It is used to describe the distribution over multiple draws from the same distribution, e.g. the number of heads when tossing a biased coin (i.e. a coin with probability  $\pi$  of returning heads) 10 times. The probability is given by  $p(x) = \binom{n}{x} \pi^x (1 - \pi)^{n-x}$ .
- **Multivariate** Obviously we can have more than two outcomes, e.g. when rolling a dice multiple times. In this case the distribution is given by  $p(x) = \frac{n!}{\prod_{i=1}^k x_i!} \prod_{i=1}^k \pi_i^{x_i}$ .
- **Poisson Distribution** It is used to model the occurrence of point events that happen with a given rate, e.g. the number of raindrops arriving within a given amount of time in an area (weird fact - the number of Prussian soldiers being killed by horses kicking them followed that distribution). Given a rate  $\lambda$ , the number of occurrences is given by  $p(x) = \frac{1}{x!} \lambda^x e^{-\lambda}$ .
- **Beta, Dirichlet, Gamma, and Wishart Distributions** They are what statisticians call *conjugate* to the Binomial, Multinomial, Poisson and Gaussian respectively. Without going into detail, these distributions are often used as priors for coefficients of the latter set of distributions, e.g. a Beta distribution as a prior for modeling the probability for binomial outcomes.

## Next

[Autograd](#)

For whinges or inquiries, [open an issue on GitHub](#).



## Automatic differentiation with `autograd`

In machine learning, we *train* models to get better and better as a function of experience. Usually, *getting better* means minimizing a *loss function*, i.e. a score that answers “how *bad* is our model?” With neural networks, we choose loss functions to be differentiable with respect to our parameters. Put simply, this means that for each of the model’s parameters, we can determine how much *increasing* or *decreasing* it might affect the loss. While the calculations are straightforward, for complex models, working it out by hand can be a pain.

*MXNet*’s `autograd` package expedites this work by automatically calculating derivatives. And while most other libraries require that we compile a symbolic graph to take automatic derivatives, `mxnet.autograd`, like PyTorch, allows you to take derivatives while writing ordinary imperative code. Every time you make pass through your model, `autograd` builds a graph on the fly, through which it can immediately backpropagate gradients.

Let’s go through it step by step. For this tutorial, we’ll only need to import `mxnet.ndarray`, and `mxnet.autograd`.

```
In [1]: import mxnet as mx
        from mxnet import nd, autograd
        mx.random.seed(1)
```

## Attaching gradients

As a toy example, Let’s say that we are interested in differentiating a function `f = 2 * (x ** 2)` with respect to parameter `x`. We can start by assigning an initial value of `x`.

```
In [2]: x = nd.array([[1, 2], [3, 4]])
```

Once we compute the gradient of `f` with respect to `x`, we’ll need a place to store it. In *MXNet*, we can tell an `NDArray` that we plan to store a gradient by invoking its `attach_grad()` method.

```
In [3]: x.attach_grad()
```

Now we're going to define the function `f` and `MXNet` will generate a computation graph on the fly. It's as if `MXNet` turned on a recording device and captured the exact path by which each variable was generated.

Note that building the computation graph requires a nontrivial amount of computation. So `MXNet` will only build the graph when explicitly told to do so. We can instruct `MXNet` to start recording by placing code inside a `with autograd.record():` block.

```
In [4]: with autograd.record():
        y = x * 2
        z = y * x
```

Let's backprop by calling `z.backward()`. When `z` has more than one entry, `z.backward()` is equivalent to `mx.nd.sum(z).backward()`.

```
In [5]: z.backward()
```

Now, let's see if this is the expected output. Remember that `y = x * 2`, and `z = x * y`, so `z` should be equal to `2 * x * x`. After, doing backprop with `z.backward()`, we expect to get back gradient  $dz/dx$  as follows:  $dy/dx = 2$ ,  $dz/dx = 4 * x$ . So, if everything went according to plan, `x.grad` should consist of an NDArray with the values `[[4, 8], [12, 16]]`.

```
In [6]: print(x.grad)
```

```
[[ 4.  8.]
 [12. 16.]]
<NDArray 2x2 @cpu(0)>
```

## Head gradients and the chain rule

*Warning: This part is tricky, but not necessary to understanding subsequent sections.*

Sometimes when we call the backward method on an NDArray, e.g. `y.backward()`, where `y` is a function of `x` we are just interested in the derivative of `y` with respect to `x`. Mathematicians write this as  $\frac{dy(x)}{dx}$ . At other times, we may be interested in the gradient of `z` with respect to `x`, where `z` is a function of `y`, which in turn, is a function of `x`. That is, we are interested in  $\frac{d}{dx} z(y(x))$ . Recall that by the chain rule  $\frac{d}{dx} z(y(x)) = \frac{dz(y)}{dy} \frac{dy(x)}{dx}$ . So, when `y` is part of a larger function `z`, and we want `x.grad` to store  $\frac{dz}{dx}$ , we can pass in the *head gradient*  $\frac{dz}{dy}$  as an input to `backward()`. The default argument is `nd.ones_like(y)`. See [Wikipedia](#) for more details.

```
In [7]: with autograd.record():
        y = x * 2
        z = y * x

        head_gradient = nd.array([[10, 1.], [.1, .01]])
        z.backward(head_gradient)
        print(x.grad)

[[ 40.      8.      ]
 [ 1.20000005  0.16   ]]
<NDArray 2x2 @cpu(0)>
```

Now that we know the basics, we can do some wild things with autograd, including building differentiable functions using Pythonic control flow.

```
In [8]: a = nd.random_normal(shape=3)
        a.attach_grad()

        with autograd.record():
            b = a * 2
            while (nd.norm(b) < 1000).asscalar():
                b = b * 2

            if (mx.nd.sum(b) > 0).asscalar():
                c = b
            else:
                c = 100 * b
```

```
In [9]: head_gradient = nd.array([0.01, 1.0, .1])
        c.backward(head_gradient)
```

```
In [10]: print(a.grad)
```

```
[ 2048. 204800. 20480.]
<NDArray 3 @cpu(0)>
```

## Next

[Linear regression from scratch](#)

For whinges or inquiries, [open an issue on GitHub](#).

```
In [ ]:
```



# Linear regression from scratch

Powerful ML libraries can eliminate repetitive work, but if you rely too much on abstractions, you might never learn how neural networks really work under the hood. So for this first example, let's get our hands dirty and build everything from scratch, relying only on autograd and NDAarray. First, we'll import the same dependencies as in the [autograd chapter](#):

```
In [1]: import mxnet as mx
        from mxnet import nd, autograd
        mx.random.seed(1)
```

## Linear regression

We'll focus on the problem of linear regression. Given a collection of data points  $x$ , and corresponding target values  $y$ , we'll try to find the line, parameterized by a vector  $w$  and intercept  $b$  that approximately best associates data points  $x[i]$  with their corresponding labels  $y[i]$ . Using some proper math notation, we want to learn a prediction

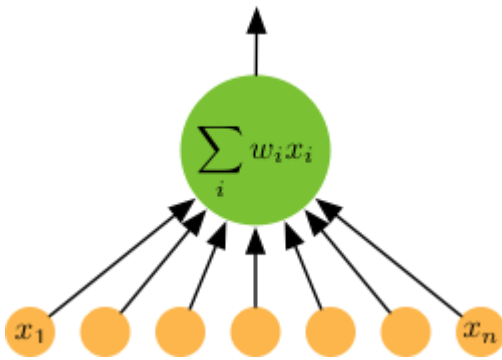
$$\hat{y} = X \cdot w + b$$

that minimizes the squared error across all examples

$$\sum_{i=1}^n (\hat{y}_i - y_i)^2.$$

You might notice that linear regression is an ancient model and wonder why we would present a linear model as the first example in a tutorial series on neural networks. Well it turns out that we can express linear regression as the simplest possible (useful) neural network. A neural network is just a collection of nodes (aka neurons) connected by directed edges. In most networks, we arrange the nodes into layers with each taking input from the nodes below. To calculate the value of any node, we first perform a weighted sum of the inputs (according to weights  $w$ ) and then apply an *activation function*. For linear regression, we have two layers, the input (depicted in orange) and a single output node (depicted in green) and the activation function is just the identity function.

In this picture, we visualize all of the components of each input as orange circles.



To make things easy, we're going to work with synthetic data where we know the solution, by generating random data points `X[i]` and labels `y[i] = 2 * X[i][0] - 3.4 * X[i][1] + 4.2 + noise` where the noise is drawn from a random gaussian with mean `0` and variance `.1`.

In mathematical notation we'd say that the true labeling function is

$$y = X \cdot w + b + \eta, \quad \text{for } \eta \sim \mathcal{N}(0, \sigma^2)$$

```
In [2]: num_inputs = 2
num_outputs = 1
num_examples = 10000

def real_fn(X):
    return 2 * X[:, 0] - 3.4 * X[:, 1] + 4.2

X = nd.random_normal(shape=(num_examples, num_inputs))
noise = .01 * nd.random_normal(shape=(num_examples,))
y = real_fn(X) + noise
```

Notice that each row in `X` consists of a 2-dimensional data point and that each row in `Y` consists of a 1-dimensional target value.

```
In [3]: print(X[0])
print(y[0])

[-0.67765152  0.03629481]
<NDArray 2 @cpu(0)>

[ 2.74159384]
<NDArray 1 @cpu(0)>
```

We can confirm that for any randomly chosen point, a linear combination with the (known) optimal parameters produces a prediction that is indeed close to the target value

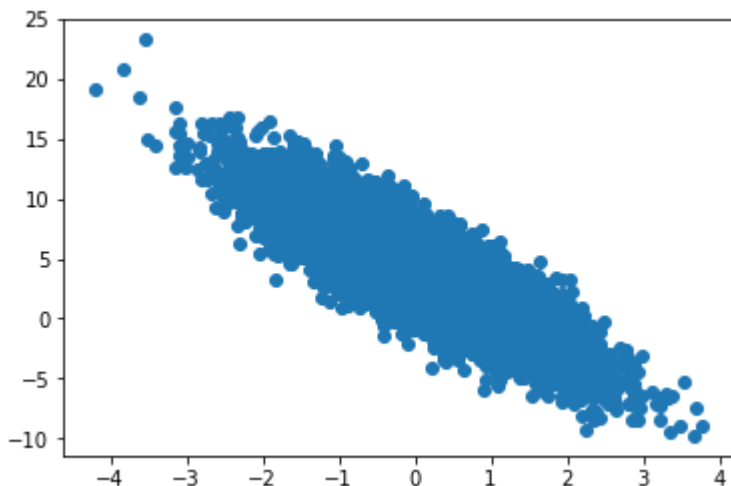
```
In [4]: print(2 * X[0, 0] - 3.4 * X[0, 1] + 4.2)

[ 2.7212944]
<NDArray 1 @cpu(0)>
```

We can visualize the correspondence between our second feature (`x[:, 1]`) and the target values `y` by generating a scatter plot with the Python plotting package `matplotlib`.

Make sure that `matplotlib` is installed. Otherwise, you may install it by running `pip2 install matplotlib` (for Python 2) or `pip3 install matplotlib` (for Python 3) on your command line.

```
In [5]: import matplotlib.pyplot as plt
plt.scatter(X[:, 1].asnumpy(), y.asnumpy())
plt.show()
```



## Data iterators

Once we start working with neural networks, we're going to need to iterate through our data points quickly. We'll also want to be able to grab batches of `k` data points at a time, to shuffle our data. In MXNet, data iterators give us a nice set of utilities for fetching and manipulating data. In particular, we'll work with the simple `DataLoader` class, that provides an intuitive way to use an `ArrayDataset` for training models.

```
In [6]: batch_size = 4
train_data = mx.gluon.data.DataLoader(mx.gluon.data.ArrayDataset(X, y),
                                       batch_size=batch_size, shuffle=True)
```

Once we've initialized our `DataLoader` (`train_data`), we can easily fetch batches by calling `train_data.next()`. `batch.data` gives us a list of inputs. Because our model has only one input (`x`), we'll just be grabbing `batch.data[0]`.

```
In [7]: for data, label in train_data:
        print(data, label)
        break
```

```
[[ -1.00448346  0.40962201]
 [  0.25873452 -0.89008772]
 [  0.85362488 -0.77108061]
 [-2.29745603  2.35324502]]
<NDArray 4x2 @cpu(0)>
[  0.77388477  7.72972631  8.53470325 -8.38178253]
<NDArray 4 @cpu(0)>
```

Finally, we can iterate over `train_data` just as though it were an ordinary Python list:

```
In [8]: counter = 0
        for data, label in train_data:
            counter += 1
        print(counter)
```

2500

## Model parameters

Now let's allocate some memory for our parameters and set their initial values.

```
In [9]: w = nd.random_normal(shape=(num_inputs, num_outputs))
        b = nd.random_normal(shape=num_outputs)
        params = [w, b]
```

In the succeeding cells, we're going to update these parameters to better fit our data. This will involve taking the gradient (a multi-dimensional derivative) of some *loss function* with respect to the parameters. We'll update each parameter in the direction that reduces the loss. But first, let's just allocate some memory for each gradient.

```
In [10]: for param in params:
          param.attach_grad()
```

## Neural networks

Next we'll want to define our model. In this case, we'll be working with linear models, the simplest possible *useful* neural network. To calculate the output of the linear model, we simply multiply a given input with the model's weights (`w`), and add the offset `b`.

```
In [11]: def net(X):
          return mx.nd.dot(X, w) + b
```

Ok, that was easy.

## Loss function

Train a model means making it better and better over the course of a period of training. But in order for this goal to make any sense at all, we first need to define what *better* means in the first place. In this case, we'll use the squared distance between our prediction and the true value.

```
In [12]: def square_loss(yhat, y):  
         return nd.mean((yhat - y) ** 2)
```

## Optimizer

It turns out that linear regression actually has a closed-form solution. However, most interesting models that we'll care about cannot be solved analytically. So we'll solve this problem by stochastic gradient descent. At each step, we'll estimate the gradient of the loss with respect to our weights, using one batch randomly drawn from our dataset. Then, we'll update our parameters a small amount in the direction that reduces the loss. The size of the step is determined by the *learning rate* `lr`.

```
In [13]: def SGD(params, lr):  
         for param in params:  
             param[:] = param - lr * param.grad
```

## Execute training loop

Now that we have all the pieces all we have to do is wire them together by writing a training loop. First we'll define `epochs`, the number of passes to make over the dataset. Then for each pass, we'll iterate through `train_data`, grabbing batches of examples and their corresponding labels.

For each batch, we'll go through the following ritual: \* Generate predictions (`yhat`) and the loss (`loss`) by executing a forward pass through the network. \* Calculate gradients by making a backwards pass through the network (`loss.backward()`). \* Update the model parameters by invoking our SGD optimizer.

```
In [14]: def plot(losses, X, sample_size=100):  
         xs = list(range(len(losses)))  
         f, (fg1, fg2) = plt.subplots(1, 2)  
         fg1.set_title('Loss during training')  
         fg1.plot(xs, losses, '-r')  
         fg2.set_title('Estimated vs real function')  
         fg2.plot(X[:sample_size, 1].asnumpy(),  
                  net(X[:sample_size, :]).asnumpy(), 'or', label='Estimated')  
         fg2.plot(X[:sample_size, 1].asnumpy(),  
                  real_fn(X[:sample_size, :]).asnumpy(), '*g', label='Real')  
         fg2.legend()  
         plt.show()
```

```
In [15]: epochs = 2  
         ctx = mx.cpu()
```

```

learning_rate = .001
smoothing_constant = .01
moving_loss = 0
niter = 0
losses = []
plot(losses, X)

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx)
        label = label.as_in_context(ctx).reshape((-1, 1))
        with autograd.record():
            output = net(data)
            loss = square_loss(output, label)
        loss.backward()
        SGD(params, learning_rate)

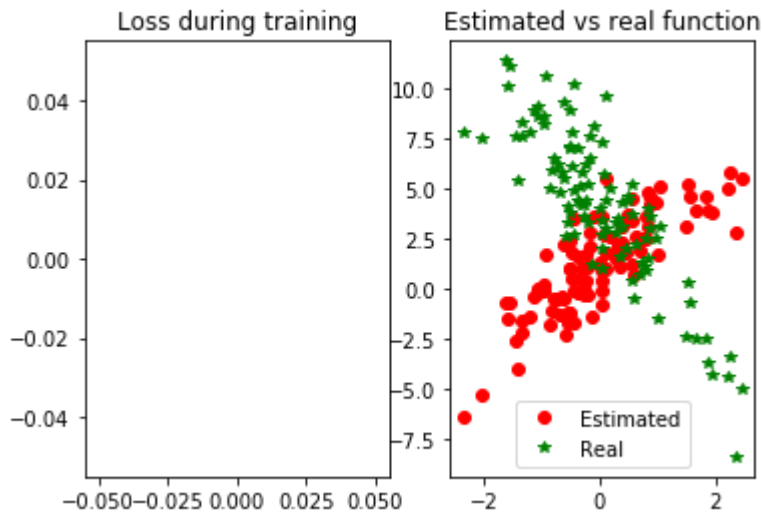
        #####
        # Keep a moving average of the losses
        #####
        niter += 1
        curr_loss = nd.mean(loss).asscalar()
        moving_loss = (1 - smoothing_constant) * moving_loss + (smoothing_constant) *
curr_loss

        # correct the bias from the moving averages
        est_loss = moving_loss / (1 - (1 - smoothing_constant) ** niter)

        if (i + 1) % 500 == 0:
            print("Epoch %s, batch %s. Moving avg of loss: %s" % (e, i, est_loss))
            losses.append(est_loss)

plot(losses, X)

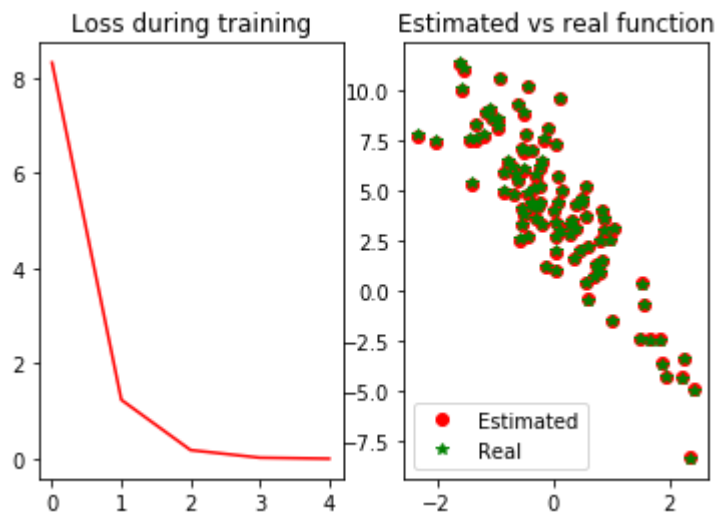
```



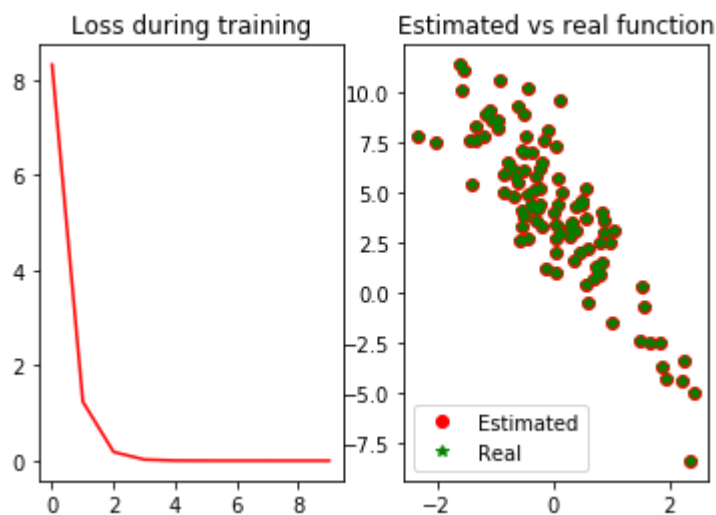
```

Epoch 0, batch 499. Moving avg of loss: 8.31916862514
Epoch 0, batch 999. Moving avg of loss: 1.23927701496
Epoch 0, batch 1499. Moving avg of loss: 0.183482519856
Epoch 0, batch 1999. Moving avg of loss: 0.0238096606951
Epoch 0, batch 2499. Moving avg of loss: 0.00301649309496

```



Epoch 1, batch 499. Moving avg of loss: 0.000478033840371  
Epoch 1, batch 999. Moving avg of loss: 0.000149816343283  
Epoch 1, batch 1499. Moving avg of loss: 0.000108049033043  
Epoch 1, batch 1999. Moving avg of loss: 9.81455696598e-05  
Epoch 1, batch 2499. Moving avg of loss: 9.96083186262e-05



## Conclusion

You've seen that using just `mxnet.ndarray` and `mxnet.autograd`, we can build statistical models from scratch. In the following tutorials, we'll build on this foundation, introducing the basic ideas between modern neural networks and powerful abstractions in MXNet for building complex models with little code.

## Next

[Linear regression with gluon](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Linear regression with gluon

Now that we've implemented a whole neural network from scratch, using nothing but `mx.ndarray` and `mxnet.autograd`, let's see how we can make the same model while doing a lot less work.

Again, let's import some packages, this time adding `mxnet.gluon` to the list of dependencies.

```
In [1]: from __future__ import print_function
import mxnet as mx
import mxnet.ndarray as nd
from mxnet import autograd
from mxnet import gluon
```

### Set the context

We'll also want to set a context to tell gluon where to do most of the computation.

```
In [2]: ctx = mx.cpu()
```

### Build the dataset

Again we'll look at the problem of linear regression and stick with the same synthetic data.

```
In [3]: num_inputs = 2
num_outputs = 1
num_examples = 10000

def real_fn(X):
    return 2 * X[:, 0] - 3.4 * X[:, 1] + 4.2

X = nd.random_normal(shape=(num_examples, num_inputs))
noise = 0.01 * nd.random_normal(shape=(num_examples,))
y = real_fn(X) + noise
```

### Load the data iterator

We'll stick with the `DataLoader` for handling out data batching.

```
In [4]: batch_size = 4
train_data = gluon.data.DataLoader(gluon.data.ArrayDataset(X, y),
                                   batch_size=batch_size, shuffle=True)
```

### Define the model

When we implemented things from scratch, we had to individually allocate parameters and then compose them together as a model. While it's good to know how to do things from scratch, with `gluon`, we can just compose a network from predefined layers. For a linear model, the appropriate layer is called `Dense`. It's called a *dense* layer because every node in the input is connected to every node in the subsequent layer. That description seems excessive because we only have one output here. But in most subsequent chapters we'll work with networks that have multiple outputs.

Unless we're planning to make some wild decisions (and at some point, we will!), the easiest way to throw together a neural network is to rely on the `gluon.nn.Sequential`. Once instantiated, a `Sequential` just stores a chain of layers. Presented with data, the `Sequential` executes each layer in turn, using the output of one layer as the input for the next. We'll delve deeper into these details later when we actually have more than one layer to work with (we could have multiple parallel branches, long chains, etc.). For now let's just instantiate the `Sequential`.

```
In [5]: net = gluon.nn.Sequential()
```



Recall that in our linear regression example, the number of inputs is 2 and the number of outputs is 1. We can then add on a single `Dense` layer. The most direct way to do this is to specify the number of inputs and the number of outputs.

```
In [6]: with net.name_scope():
        net.add(gluon.nn.Dense(1, in_units=2))
```

This tells `gluon` all that it needs in order to allocate memory for the weights. The `net.name_scope` tells `gluon` that it should name all parameters in a consistent way within `net`, such that we could reference individual weights explicitly at a later stage. Now all we need to do is initialize the weights, instantiate a loss and an optimizer, and we can start training.

## Shape inference

One slick feature that we can take advantage of in `gluon` is shape inference on parameters. Instead of explicitly declaring the number of inputs to a layer, we can simply state the number of outputs.

```
In [7]: net = gluon.nn.Sequential()
        with net.name_scope():
            net.add(gluon.nn.Dense(1))
```

You might wonder, how can `gluon` allocate our parameters if it doesn't know what shape they should take? We'll elaborate on this and more of `gluon`'s internal workings in [our chapter on plumbing](#), but here's the short version: In fact, `gluon` doesn't allocate our parameters at that very moment. Instead it defers allocation to the first time we actually make a forward pass through the model with real data. Then, when `gluon` sees the shape of our data, it can infer the shapes of all of the parameters.

## Initialize parameters

This all we need to do to define our network. However, we're not ready to pass it data just yet. If you try calling `net(nd.array([[0,1]]))`, you'll find the following hideous error message:

```
RuntimeError: Parameter dense1_weight has not been initialized. Note that you should initialize parameters and create Trainer with Block.cc
```

That's because we haven't yet told `gluon` what the *initial values* for our parameters should be. Also note that we need not tell our network about the *input dimensionality* and it still works. This is because the dimensions are bound the first time `net(x)` is called. This is a common theme in MxNet - stuff is evaluated only when needed (called lazy evaluation), using all the information available at the time when the results is needed.

Before we can do anything with this model, we must initialize its parameters. *MXNet* provides a variety of common initializers in `mxnet.init`. To keep things consistent with the model we built by hand, we'll choose to initialize each parameter by sampling from a standard normal distribution. Note that we pass the initializer a *context*. This is how we tell `gluon` model where to store our parameters. Once we start training deep nets, we'll generally want to keep parameters on one or more GPUs (and on more than one computer).

```
In [8]: net.collect_params().initialize(mx.init.Normal(sigma=1.), ctx=ctx)
```

## Deferred Initialization

Since `gluon` doesn't know the shape of our net's parameters, and we haven't even allocated memory for them yet, it might seem bizarre that we can initialize them. This is where `gluon` does a little more magic to make our lives easier. When we call `initialize`, `gluon` associates each parameter with an initializer. However, the *actual initialization* is deferred until the shapes have been inferred.

## Define loss

Instead of writing our own loss function we're just going to call down to `gluon.loss.L2Loss`

```
In [9]: square_loss = gluon.loss.L2Loss()
```

## Optimizer

Instead of writing stochastic gradient descent from scratch every time, we can instantiate a `gluon.Trainer`, passing it a dictionary of parameters. Note that the `sgd` optimizer in `gluon` actually uses SGD with momentum and clipping (both can be switched off if needed), since these modifications make it converge rather much better. We will discuss this later when we go over a range of optimization algorithms in detail.

```
In [10]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

## Execute training loop

You might have noticed that it was a bit more concise to express our model in `gluon`. For example, we didn't have to individually allocate parameters, define our loss function, or implement stochastic gradient descent. The benefits of relying on `gluon`'s abstractions will grow substantially once we start working with much more complex models. But once we have all the basic pieces in place, the training loop itself is quite similar to what we would do if implementing everything from scratch.

To refresh your memory. For some number of `epochs`, we'll make a complete pass over the dataset (`train_data`), grabbing one mini-batch of inputs and the corresponding ground-truth labels at a time.

Then, for each batch, we'll go through the following ritual. So that this process becomes maximally ritualistic, we'll repeat it verbatim: \* Generate predictions (`yhat`) and the loss (`loss`) by executing a forward pass through the network. \* Calculate gradients by making a backwards pass through the network via `loss.backward()`. \* Update the model parameters by invoking our SGD optimizer (note that we need not tell `trainer.step` about which parameters but rather just the amount of data, since we already performed that in the initialization of `trainer`).

```
In [11]: epochs = 1
smoothing_constant = .01
moving_loss = 0
niter = 0
loss_seq = []

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx)
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            loss = square_loss(output, label)
        loss.backward()
        trainer.step(batch_size)

        #####
        # Keep a moving average of the losses
        #####
        niter += 1
        curr_loss = nd.mean(loss).asscalar()
        moving_loss = (1 - smoothing_constant) * moving_loss + (smoothing_constant) * curr_loss

        # correct the bias from the moving averages
        est_loss = moving_loss / (1 - (1 - smoothing_constant) ** niter)
        loss_seq.append(est_loss)

    print("Epoch %s. Moving avg of MSE: %s" % (e, est_loss))
```

Epoch 0. Moving avg of MSE: 4.94698964984e-05

## Visualizing the learning curve

Now let's check how quickly SGD learns the linear regression model by plotting the learning curve.

```
In [12]: # plot the convergence of the estimated loss function
%matplotlib inline

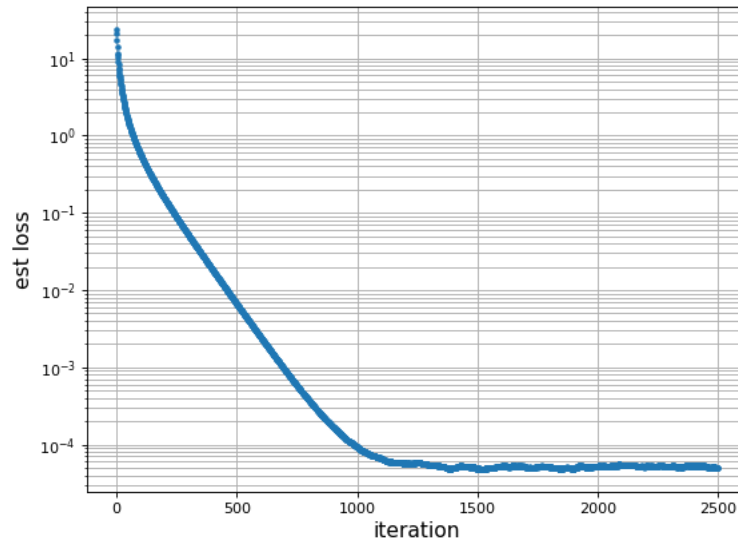
import matplotlib
import matplotlib.pyplot as plt

plt.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
plt.semilogy(range(niter), loss_seq, '.')

# adding some additional bells and whistles to the plot
```

```
plt.grid(True, which="both")
plt.xlabel('iteration', fontsize=14)
plt.ylabel('est loss', fontsize=14)
```

Out[12]: Text(0,0.5, 'est loss')



As we can see, the loss function converges linearly (exponentially) to the optimal solution.

## Getting the learned model parameters

As an additional sanity check, since we generated the data from a Gaussian linear regression model, we want to make sure that the learner managed to recover the model parameters, which were set to weight 2,  $-3.4$  with an offset of 4.2.

```
In [13]: params = net.collect_params() # this returns a ParameterDict
print('The type of "params" is a ', type(params))

# A ParameterDict is a dictionary of Parameter class objects
# therefore, here is how we can read off the parameters from it.

for param in params.values():
    print(param.name, param.data())

The type of "params" is a <class 'mxnet.gluon.parameter.ParameterDict'>
sequential1_dense0_weight
[[ 1.99977756 -3.4003276]]
<NDArray 1x2 @cpu(0)>
sequential1_dense0_bias
[ 4.19792223]
<NDArray 1 @cpu(0)>
```

## Conclusion

As you can see, even for a simple example like linear regression, `gluon` can help you to write quick, clean, code. Next, we'll repeat this exercise for multi-layer perceptrons, extending these lessons to deep neural networks and (comparatively) real datasets.

## Next

[The perceptron algorithm](#)

For whinges or inquiries, [open an issue on GitHub](#).

# The Perceptron

We just employed an optimization method - stochastic gradient descent, without really thinking twice about why it should work at all. It's probably worth while to pause and see whether we can gain some intuition about why this should actually work at all. We start with considering the E. Coli of machine learning algorithms - the Perceptron. After that, we'll give a simple convergence proof for SGD. This chapter is not really needed for practitioners but will help to understand why the algorithms that we use are working at all.

```
In [1]: import mxnet as mx
        from mxnet import nd, autograd
        import matplotlib.pyplot as plt
        import numpy as np
        mx.random.seed(1)
```

## A Separable Classification Problem

The Perceptron algorithm aims to solve the following problem: given some classification problem of data  $x \in \mathbb{R}^d$  and labels  $y \in \{\pm 1\}$ , can we find a linear function  $f(x) = w^T x + b$  such that  $f(x) > 0$  whenever  $y = 1$  and  $f(x) < 0$  for  $y = -1$ . Obviously not all classification problems fall into this category but it's a very good baseline for what can be solved easily. It's also the kind of problems computers could solve in the 1960s. The easiest way to ensure that we have such a problem is to fake it by generating such data. We are going to make the problem a bit more interesting by specifying how well the data is separated.

```
In [2]: # generate fake data that is linearly separable with a margin epsilon given the data
def getfake(samples, dimensions, epsilon):
    wfake = nd.random_normal(shape=(dimensions)) # fake weight vector for separation
    bfake = nd.random_normal(shape=(1)) # fake bias
    wfake = wfake / nd.norm(wfake) # rescale to unit length

    # making some linearly separable data, simply by choosing the labels accordingly
    X = nd.zeros(shape=(samples, dimensions))
    Y = nd.zeros(shape=(samples))

    i = 0
    while (i < samples):
        tmp = nd.random_normal(shape=(1, dimensions))
        margin = nd.dot(tmp, wfake) + bfake
        if (nd.norm(tmp).asscalar() < 3) & (abs(margin.asscalar()) > epsilon):
            X[i,:] = tmp
            Y[i] = 1 if margin > 0 else -1
            i += 1
    return X, Y

# plot the data with colors chosen according to the labels
def plotdata(X,Y):
```

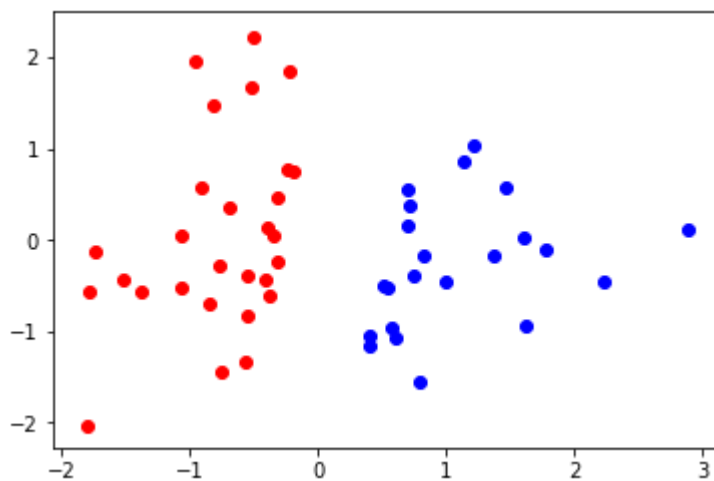
```

for (x,y) in zip(X,Y):
    if (y.asscalar() == 1):
        plt.scatter(x[0].asscalar(), x[1].asscalar(), color='r')
    else:
        plt.scatter(x[0].asscalar(), x[1].asscalar(), color='b')

# plot contour plots on a [-3,3] x [-3,3] grid
def plotscore(w,d):
    xgrid = np.arange(-3, 3, 0.02)
    ygrid = np.arange(-3, 3, 0.02)
    xx, yy = np.meshgrid(xgrid, ygrid)
    zz = nd.zeros(shape=(xgrid.size, ygrid.size, 2))
    zz[:, :, 0] = nd.array(xx)
    zz[:, :, 1] = nd.array(yy)
    vv = nd.dot(zz, w) + b
    CS = plt.contour(xgrid, ygrid, vv.asnumpy())
    plt.clabel(CS, inline=1, fontsize=10)

X, Y = getfake(50, 2, 0.3)
plotdata(X,Y)
plt.show()

```



Now we are going to use the simplest possible algorithm to learn parameters. It's inspired by the [Hebbian Learning Rule](#) which suggests that positive events should be reinforced and negative ones diminished. The analysis of the algorithm is due to Rosenblatt and we will give a detailed proof of it after illustrating how it works. In a nutshell, after initializing parameters  $w = 0$  and  $b = 0$  it updates them by  $yx$  and  $y$  respectively to ensure that they are properly aligned with the data. Let's see how well it works:

```

In [3]: def perceptron(w,b,x,y):
    if (y * (nd.dot(w,x) + b)).asscalar() <= 0:
        w += y * x
        b += y
        return 1
    else:
        return 0

w = nd.zeros(shape=(2))
b = nd.zeros(shape=(1))
for (x,y) in zip(X,Y):
    res = perceptron(w,b,x,y)
    if (res == 1):
        print('Encountered an error and updated parameters')
        print('data {}, label {}'.format(x.asnumpy(),y.asscalar()))
        print('weight {}, bias {}'.format(w.asnumpy(),b.asscalar()))
        plotscore(w,b)

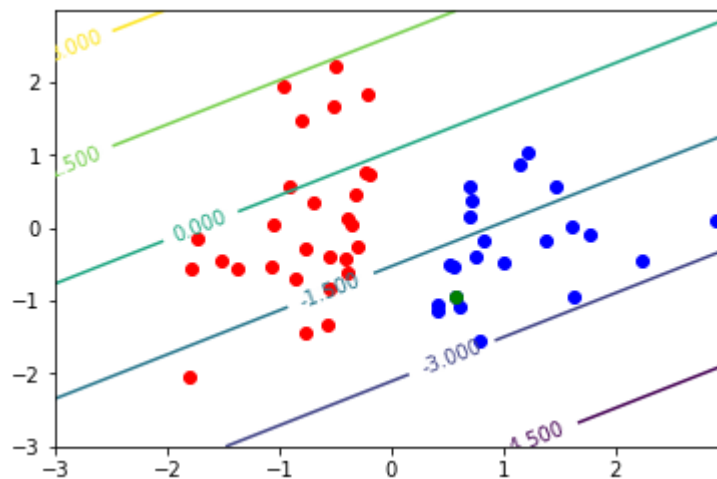
```

```

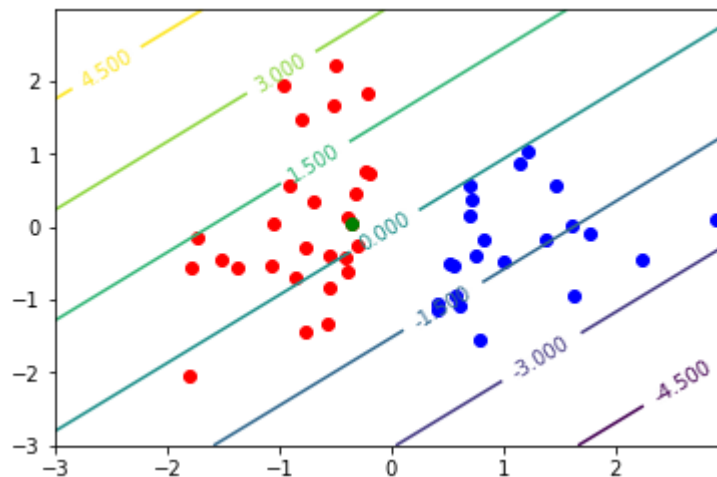
plotdata(X,Y)
plt.scatter(x[0].asscalar(), x[1].asscalar(), color='g')
plt.show()

```

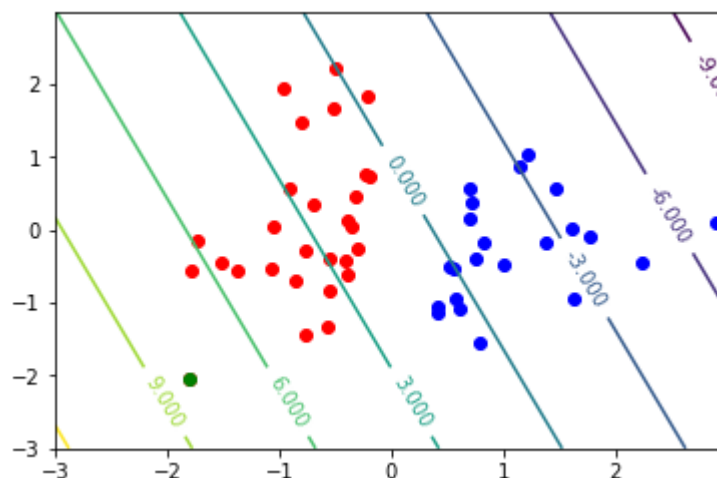
Encountered an error and updated parameters  
data [ 0.57595438 -0.95017916], label -1.0  
weight [-0.57595438 0.95017916], bias -1.0



Encountered an error and updated parameters  
data [-0.3469252 0.03751944], label 1.0  
weight [-0.92287958 0.98769861], bias 0.0



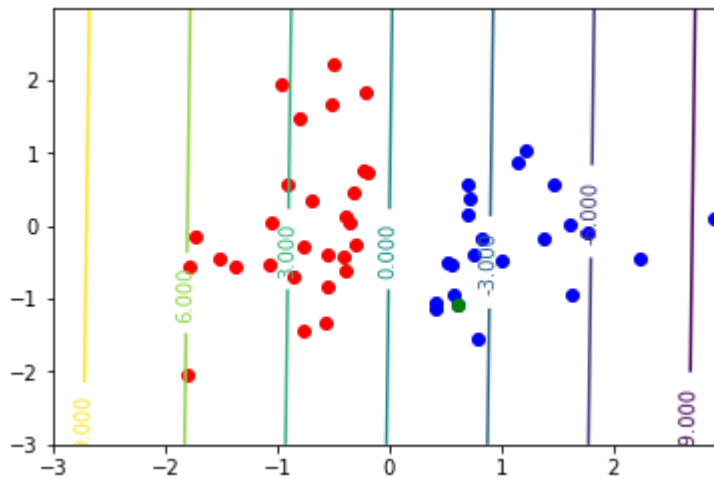
Encountered an error and updated parameters  
data [-1.80471897 -2.04010558], label 1.0  
weight [-2.72759867 -1.05240703], bias 1.0



```

Encountered an error and updated parameters
data [ 0.60334933 -1.08074296], label -1.0
weight [-3.33094788  0.02833593], bias 0.0

```



As we can see, the model has learned something - all the red dots are positive and all the blue dots correspond to a negative value. Moreover, we saw that the values for  $w^T x + b$  became more extreme as values over the grid. Did we just get lucky in terms of classification or is there any math behind it? Obviously there is, and there's actually a nice theorem to go with this. It's the perceptron convergence theorem.

## The Perceptron Convergence Theorem

**Theorem** Given data  $x_i$  with  $\|x_i\| \leq R$  and labels  $y_i \in \{\pm 1\}$  for which there exists some pair of parameters  $(w^*, b^*)$  such that  $y_i((w^*)^T x_i + b) \geq \epsilon$  for all data, and for which  $\|w^*\| \leq 1$  and  $b^2 \leq 1$ , then the perceptron algorithm converges after at most  $2(R^2 + 1)/\epsilon^2$  iterations.

The cool thing is that this theorem is *independent of the dimensionality of the data*. Moreover, it is *independent of the number of observations*. Lastly, looking at the algorithm itself, we see that we only need to store the mistakes that the algorithm made - for the data that was classified correctly no update on  $(w, b)$  happened. As a first step, let's check how accurate the theorem is.

```

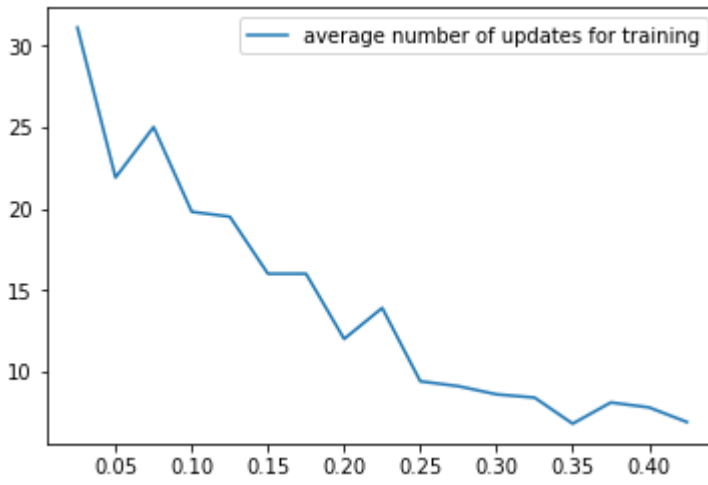
In [4]: Eps = np.arange(0.025, 0.45, 0.025)
        Err = np.zeros(shape=(Eps.size))

        for j in range(10):
            for (i,epsilon) in enumerate(Eps):
                X, Y = getfake(1000, 2, epsilon)

                for (x,y) in zip(X,Y):
                    Err[i] += perceptron(w,b,x,y)

        Err = Err / 10.0
        plt.plot(Eps, Err, label='average number of updates for training')
        plt.legend()
        plt.show()

```



As we can see, the number of errors (and with it, updates) decreases inversely with the width of the margin. Let's see whether we can put this into equations. The first thing to consider is the size of the inner product between  $(w, b)$  and  $(w^*, b^*)$ , the parameter that solves the classification problem with margin  $\epsilon$ . Note that we do not need explicit knowledge of  $(w^*, b^*)$  for this, just know about its existence. For convenience, we will index  $w$  and  $b$  by  $t$ , the number of updates on the parameters. Moreover, whenever convenient we will treat  $(w, b)$  as a new vector with an extra dimension and with the appropriate terms such as norms  $\|(w, b)\|$  and inner products.

First off,  $w_0^\top w^* + b_0 b^* = 0$  by construction. Second, by the update rule we have that

$$\begin{aligned}
 (w_{t+1}, b_{t+1})^\top (w^*, b^*) &= (w_t, b_t)^\top (w^*, b^*) + y_t (x_t^\top w^* + b^*) \\
 &\geq (w_t, b_t)^\top (w^*, b^*) + \epsilon \\
 &\geq (t+1)\epsilon
 \end{aligned}$$

Here the first equality follows from the definition of the weight updates. The next inequality follows from the fact that  $(w^*, b^*)$  separate the problem with margin at least  $\epsilon$ , and the last inequality is simply a consequence of iterating this inequality  $t+1$  times. Growing alignment between the 'ideal' and the actual weight vectors is great, but only if the actual weight vectors don't grow too rapidly. So we need a bound on their length:

$$\begin{aligned}
 \|(w_{t+1}, b_{t+1})\|^2 &\geq \|(w_t, b_t)\|^2 + 2y_t x_t^\top w_t + 2y_t b_t + \|(x_t, 1)\|^2 \\
 &= \|(w_t, b_t)\|^2 + 2y_t (x_t^\top w_t + b_t) + \|(x_t, 1)\|^2 \\
 &\geq \|(w_t, b_t)\|^2 + R^2 + 1 \\
 &\geq (t+1)(R^2 + 1)
 \end{aligned}$$

Now let's combine both inequalities. By Cauchy-Schwartz, i.e.  $\|a\| \cdot \|b\| \geq a^\top b$  and the first inequality we have that  $t\epsilon \leq (w_t, b_t)^\top (w^*, b^*) \leq \|(w_t, b_t)\| \sqrt{2}$ . Using the second inequality we furthermore get  $\|(w_t, b_t)\| \leq \sqrt{t(R^2 + 1)}$ . Combined this yields



$$t\epsilon \leq \sqrt{2t(R^2 + 1)}$$

This is a strange equation - we have a linear term on the left and a sublinear term on the right. So this inequality clearly cannot hold indefinitely for large  $t$ . The only logical conclusion is that there must never be updates beyond when the inequality is no longer satisfied. We have  $t \leq 2(R^2 + 1)/\epsilon^2$ , which proves our claim.

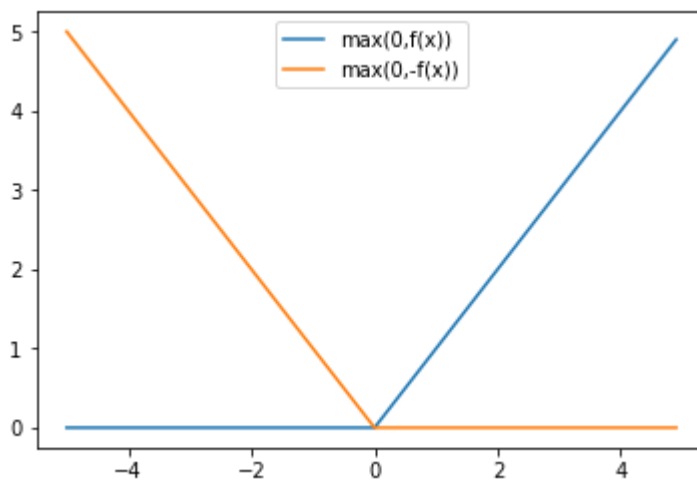
**Note** - sometimes the perceptron convergence theorem is written without bias  $b$ . In this case a lot of things get simplified both in the proof and in the bound, since we can do away with the constant terms. Without going through details, the theorem becomes  $t \leq R^2/\epsilon^2$ .

**Note** - the perceptron convergence proof crucially relied on the fact that the data is actually separable. If this is not the case, the perceptron algorithm will diverge. It will simply keep on trying to get things right by updating  $(w, b)$ . And since it has no safeguard to keep the parameters bounded, the solution will get worse. This sounds like an ‘academic’ concern, alas it is not. The avatar in the computer game [Black and White] ([https://en.wikipedia.org/wiki/Black\\_%26\\_White\\_\(video\\_game%29\)](https://en.wikipedia.org/wiki/Black_%26_White_(video_game%29))) used the perceptron to adjust the avatar. Due to the poorly implemented update rule the game quickly became unplayable after a few hours (as one of the authors can confirm).

## Stochastic Gradient Descent

The perceptron algorithm also can be viewed as a stochastic gradient descent algorithm, albeit with a rather strange loss function:  $\max(0, -yf(x))$ . This is commonly called the hinge loss. As can be checked quite easily, its gradient is 0 whenever  $yf(x) > 0$ , i.e. whenever  $x$  is classified correctly, and gradient  $-y$  for incorrect classification. For a linear function, this leads exactly to the updates that we have (with the minor difference that we consider  $f(x) = 0$  as an example of incorrect classification). To get some intuition, let’s plot the loss function.

```
In [5]: f = np.arange(-5,5,0.1)
        zero = np.zeros(shape=(f.size))
        lplus = np.max(np.array([f,zero]), axis=0)
        lminus = np.max(np.array([-f,zero]), axis=0)
        plt.plot(f,lplus, label='max(0,f(x))')
        plt.plot(f,lminus, label='max(0,-f(x))')
        plt.legend()
        plt.show()
```



More generally, a stochastic gradient descent algorithm uses the following template:

```
initialize w
loop over data and labels (x,y):
    compute f(x)
    compute loss gradient g = partial_w l(y, f(x))
    w = w - eta g
```

Here the learning rate  $\eta$  may well change as we iterate over the data. Moreover, we may traverse the data in nonlinear order (e.g. we might reshuffle the data), depending on the specific choices of the algorithm. The issue is that as we go over the data, sometimes the gradient might point us into the right direction and sometimes it might not. Intuitively, on average things *should* get better. But to be really sure, there's only one way to find out - we need to prove it. We pick a simple and elegant (albeit a bit restrictive) proof of [Nesterov and Vial](#).

The situation we consider are *convex* losses. This is a bit restrictive in the age of deep networks but still quite instructive (in addition to that, nonconvex convergence proofs are a lot messier). For recap - a convex function  $f(x)$  satisfies  $f(\lambda x + (1 - \lambda)x') \leq \lambda f(x) + (1 - \lambda)f(x')$ , that is, the linear interpolant between function values is *larger* than the function values in between. Likewise, a convex set  $S$  is a set where for any points  $x, x' \in S$  the line  $[x, x']$  is in the set, i.e.  $\lambda x + (1 - \lambda)x' \in S$  for all  $\lambda \in [0, 1]$ . Now assume that  $w^*$  is the minimizer of the expected loss that we are trying to minimize, e.g.

$$w^* = \operatorname{argmin}_w R(w) \text{ where } R(w) = \frac{1}{m} \sum_{i=1}^m l(y_i, f(x_i, w))$$

Let's assume that we actually *know* that  $w^*$  is contained in some set convex set  $S$ , e.g. a ball of radius  $R$  around the origin. This is convenient since we want to make sure that during optimization our parameter  $w$  doesn't accidentally diverge. We can ensure that, e.g. by shrinking it back to such a ball whenever needed.

Secondly, assume that we have an upper bound on the magnitude of the gradient  $g_i := \partial_w l(y_i, f(x_i, w))$  for all  $i$  by some constant  $L$  (it's called so since this is often referred to as the Lipschitz constant). Again, this is super useful since we don't want  $w$  to diverge while we're optimizing. In practice, many algorithms employ e.g. *gradient clipping* to force our gradients to be well behaved, by shrinking the gradients back to something tractable.

Third, to get rid of variance in the parameter  $w_t$  that is obtained during the optimization, we use the weighted average over the entire optimization process as our solution, i.e. we use  $\bar{w} := \sum_t \eta_t w_t / \sum_t \eta_t$ .

Let's look at the distance  $r_t := \|w_t - w^*\|$ , i.e. the distance between the optimal solution vector  $w^*$  and what we currently have. It is bounded as follows:

$$\begin{aligned} \|w_{t+1} - w^*\|^2 &= \|w_t - w^*\|^2 + \eta_t^2 \|g_t\|^2 - 2\eta_t g_t^\top (w_t - w^*) \\ &\leq \|w_t - w^*\|^2 + \eta_t^2 L^2 - 2\eta_t g_t^\top (w_t - w^*) \end{aligned}$$

Next we use convexity of  $R(w)$ . We know that  $R(w^*) \geq R(w_t) + \partial_w R(w_t)^\top (w^* - w_t)$  and moreover that the average of function values is larger than the function value of the average, i.e.  $\sum_{t=1}^T \eta_t R(w_t) / \sum_t \eta_t \geq R(\bar{w})$ . The first inequality allows us to bound the expected decrease in distance to optimality via

$$\mathbf{E}[r_{t+1} - r_t] \leq \eta_t^2 L^2 - 2\eta_t \mathbf{E}[g_t^\top (w_t - w^*)] \leq \eta_t^2 L^2 - 2\eta_t \mathbf{E}[R[w_t] - R[w^*]]$$

Summing over  $t$  and using the facts that  $r_T \geq 0$  and that  $w$  is contained inside a ball of radius  $R$  yields:

$$-R^2 \leq L^2 \sum_{t=1}^T \eta_t^2 - 2 \sum_t \eta_t \mathbf{E}[R[w_t] - R[w^*]]$$

Rearranging terms, using convexity of  $R$  the second time, and dividing by  $\sum_t \eta_t$  yields a bound on how far we are likely to stray from the best possible solution:

$$\mathbf{E}[R[\bar{w}]] - R[w^*] \leq \frac{R^2 + L^2 \sum_{t=1}^T \eta_t^2}{2 \sum_{t=1}^T \eta_t}$$

Depending on how we choose  $\eta_t$  we will get different bounds. For instance, if we make  $\eta$  constant, i.e. if we use a constant learning rate, we get the bounds  $(R^2 + L^2 \eta^2 T) / (2\eta T)$ . This is minimized for  $\eta = R / L \sqrt{T}$ , yielding a bound of  $RL / \sqrt{T}$ . A few things are interesting in this context:

- If we are potentially far away from the optimal solution, we should use a large learning rate (the  $O(R)$  dependency).

- If the gradients are potentially large, we should use a smaller learning rate (the  $O(1/L)$  dependency).
- If we have a long time to converge, we should use a smaller learning rate, but not too small.
- Large gradients and a large degree of uncertainty as to how far we are away from the optimal solution lead to poor convergence.
- More optimization steps make things better.

None of these insights are terribly surprising, albeit useful to keep in mind when we use SGD in the wild. And this was the very point of going through this somewhat tedious proof.

Furthermore, if we use a decreasing learning rate, e.g.  $\eta_t = O(1/\sqrt{t})$ , then our bounds are somewhat less tight, and we get a bound of  $O(\log T/\sqrt{T})$  bound on how far away from optimality we might be. The key difference is that for the decreasing learning rate we need not know when to stop. In other words, we get an anytime algorithm that provides a good result at any time, albeit not as good as what we could expect if we knew how much time to optimize we have right from the beginning.

## Next

[Softmax regression from scratch](#)


For whinges or inquiries, [open an issue on GitHub](#).

## Multiclass logistic regression from scratch

If you've made it through our tutorial on linear regression from scratch, then you're past the hardest part. You already know how to load and manipulate data, build computation graphs on the fly, and take derivatives. You also know how to define a loss function, construct a model, and write your own optimizer.

Nearly all neural networks that we'll build in the real world consist of these same fundamental parts. The main differences will be the type and scale of the data, and the complexity of the models. And every year or two, a new hipster optimizer comes around, but at their core they're all subtle variations of stochastic gradient descent.

So let's work on a more interesting problem now. We're going to classify images of handwritten

digits like these:  We're

going to implement a model called multiclass logistic regression. Other common names for this model include softmax regression and multinomial regression. To start, let's import our bag of libraries.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
import numpy as np
```

We'll also want to set the compute context for our modeling. Feel free to go ahead and change this to `mx.gpu(0)` if you're running on an appropriately endowed machine.

```
In [2]: ctx = mx.cpu()
```

## The MNIST dataset

This time we're going to work with real data, each a 28 by 28 centrally cropped black & white photograph of a handwritten digit. Our task will be come up with a model that can associate each image with the digit (0-9) that it depicts.

To start, we'll use MXNet's utility for grabbing a copy of this dataset. The datasets accept a transform callback that can preprocess each item. Here we cast data and label to floats and normalize data to range [0, 1]:

```
In [3]: def transform(data, label):  
        return data.astype(np.float32)/255, label.astype(np.float32)  
mnist_train = mx.gluon.data.vision.MNIST(train=True, transform=transform)  
mnist_test = mx.gluon.data.vision.MNIST(train=False, transform=transform)
```

There are two parts of the dataset for training and testing. Each part has N items and each item is a tuple of an image and a label:

```
In [4]: image, label = mnist_train[0]  
        print(image.shape, label)  
  
(28, 28, 1) 5.0
```

Note that each image has been formatted as a 3-tuple (height, width, channel). For color images, the channel would have 3 dimensions (red, green and blue).

## Record the data and label shapes

Generally, we don't want our model code to care too much about the exact shape of our input data. This way we could switch in a different dataset without changing the code that follows. Let's define variables to hold the number of inputs and outputs.

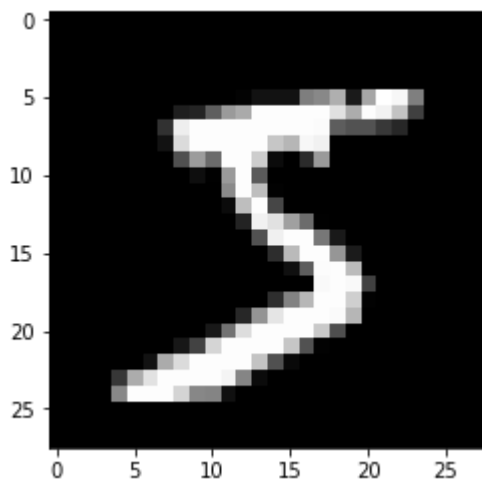
```
In [5]: num_inputs = 784  
        num_outputs = 10
```

Machine learning libraries generally expect to find images in (batch, channel, height, width) format. However, most libraries for visualization prefer (height, width, channel). Let's transpose our image into the expected shape. In this case, matplotlib expects either (height, width) or (height, width, channel) with RGB channels, so let's broadcast our single channel to 3.

```
In [6]: im = mx.nd.tile(image, (1,1,3))  
        print(im.shape)  
  
(28, 28, 3)
```

Now we can visualize our image and make sure that our data and labels line up.

```
In [7]: import matplotlib.pyplot as plt  
        plt.imshow(im.asnumpy())  
        plt.show()
```



Ok, that's a beautiful five.

## Load the data iterator

Now let's load these images into a data iterator so we don't have to do the heavy lifting.

```
In [8]: batch_size = 64
        train_data = mx.gluon.data.DataLoader(mnist_train, batch_size, shuffle=True)
```

We're also going to want to load up an iterator with *test* data. After we train on the training dataset we're going to want to test our model on the test data. Otherwise, for all we know, our model could be doing something stupid (or treacherous?) like memorizing the training examples and regurgitating the labels on command.

```
In [9]: test_data = mx.gluon.data.DataLoader(mnist_test, batch_size, shuffle=False)
```

## Allocate model parameters

Now we're going to define our model. For this example, we're going to ignore the multimodal structure of our data and just flatten each image into a single 1D vector with  $28 \times 28 = 784$  components.

Because our task is multiclass classification, we want to assign a probability to each of the classes  $P(Y=c|X)$  given the input  $X$ . In order to do this we're going to need one vector of 784 weights for each class, connecting each feature to the corresponding output. Because there are 10 classes, we can collect these weights together in a 784 by 10 matrix.

We'll also want to allocate one offset for each of the outputs. We call these offsets the *bias term* and collect them in the 10-dimensional array `b`.

```
In [10]: W = nd.random_normal(shape=(num_inputs, num_outputs))
         b = nd.random_normal(shape=num_outputs)

         params = [W, b]
```

As before, we need to let MXNet know that we'll be expecting gradients corresponding to each of these parameters during training.

```
In [11]: for param in params:
         param.attach_grad()
```

## Multiclass logistic regression

In the linear regression tutorial, we performed regression, so we had just one output  $\hat{y}$  and tried to push this value as close as possible to the true target  $y$ . Here, instead of regression, we are performing *classification*, where we want to assign each input  $X$  to one of  $L$  classes.

The basic modeling idea is that we're going to linearly map our input  $X$  onto 10 different real valued outputs `y_linear`. Then before, outputting these values, we'll want to normalize them so that they are non-negative and sum to 1. This normalization allows us to interpret the output  $\hat{y}$  as a valid probability distribution.

```
In [12]: def softmax(y_linear):
         exp = nd.exp(y_linear-nd.max(y_linear))
         norms = nd.sum(exp, axis=0, exclude=True).reshape((-1,1))
         return exp / norms
```

```
In [13]: sample_y_linear = nd.random_normal(shape=(2,10))
         sample_yhat = softmax(sample_y_linear)
         print(sample_yhat)
```

```
[[ 0.01466005  0.03104205  0.09487285  0.11615293  0.07316667  0.01516553
   0.44094777  0.08199082  0.0917872  0.04021411]
 [ 0.0309542  0.07588483  0.37230074  0.03313261  0.0499984  0.13276106
   0.14566724  0.02354518  0.08515968  0.05059606]]
<NDArray 2x10 @cpu(0)>
```

Let's confirm that indeed all of our rows sum to 1.

```
In [14]: print(nd.sum(sample_yhat, axis=1))
```

```
[ 1.  1.]
<NDArray 2 @cpu(0)>
```

But for small rounding errors, the function works as expected.

## Define the model



Now we're ready to define our model

```
In [15]: def net(X):  
         y_linear = nd.dot(X, W) + b  
         yhat = softmax(y_linear)  
         return yhat
```

## The cross-entropy loss function

Before we can start training, we're going to need to define a loss function that makes sense when our prediction is a probability distribution.

The relevant loss function here is called cross-entropy and it may be the most common loss function you'll find in all of deep learning. That's because at the moment, classification problems tend to be far more abundant than regression problems.

The basic idea is that we're going to take a target  $Y$  that has been formatted as a one-hot vector, meaning one value corresponding to the correct label is set to 1 and the others are set to 0, e.g.

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

The basic idea of cross-entropy loss is that we only care about how much probability the prediction assigned to the correct label. In other words, for true label 2, we only care about the component of  $yhat$  corresponding to 2. Cross-entropy attempts to maximize the log-likelihood given to the correct labels.

```
In [16]: def cross_entropy(yhat, y):  
         return - nd.sum(y * nd.log(yhat), axis=0, exclude=True)
```

## Optimizer

For this example we'll be using the same stochastic gradient descent (SGD) optimizer as last time.

```
In [17]: def SGD(params, lr):  
         for param in params:  
             param[:] = param - lr * param.grad
```

## Write evaluation loop to calculate accuracy

While cross-entropy is nice, differentiable loss function, it's not the way humans usually evaluate performance on multiple choice tasks. More commonly we look at accuracy, the number of correct answers divided by the total number of questions. Let's write an evaluation loop that will

take a data iterator and a network, returning the model's accuracy averaged over the entire dataset.

```
In [18]: def evaluate_accuracy(data_iterator, net):
    numerator = 0.
    denominator = 0.
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1,784))
        label = label.as_in_context(ctx)
        label_one_hot = nd.one_hot(label, 10)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        numerator += nd.sum(predictions == label)
        denominator += data.shape[0]
    return (numerator / denominator).asscalar()
```

Because we initialized our model randomly, and because roughly one tenth of all examples belong to each of the ten classes, we should have an accuracy in the ball park of .10.

```
In [19]: evaluate_accuracy(test_data, net)
```

```
Out[19]: 0.079499997
```

## Execute training loop

```
In [20]: epochs = 10
moving_loss = 0.
learning_rate = .001
smoothing_constant = .01
niter=0

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx).reshape((-1,784))
        label = label.as_in_context(ctx)
        label_one_hot = nd.one_hot(label, 10)
        with autograd.record():
            output = net(data)
            loss = cross_entropy(output, label_one_hot)
        loss.backward()
        SGD(params, learning_rate)

        #####
        # Keep a moving average of the losses
        #####
        niter +=1
        moving_loss = (1 - smoothing_constant) * moving_loss + (smoothing_constant) *
nd.mean(loss).asscalar()
        est_loss = moving_loss/(1-(1-smoothing_constant)**niter)

    test_accuracy = evaluate_accuracy(test_data, net)
    train_accuracy = evaluate_accuracy(train_data, net)
    print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, est_loss, train_accuracy,
test_accuracy))
```

```
Epoch 0. Loss: 1.33693315503, Train_acc 0.755217, Test_acc 0.7644
Epoch 1. Loss: 0.974225272841, Train_acc 0.81185, Test_acc 0.8202
Epoch 2. Loss: 0.832080314418, Train_acc 0.8358, Test_acc 0.8438
Epoch 3. Loss: 0.719344859828, Train_acc 0.850433, Test_acc 0.8566
Epoch 4. Loss: 0.684221684508, Train_acc 0.8584, Test_acc 0.8655
Epoch 5. Loss: 0.613997728956, Train_acc 0.864667, Test_acc 0.8698
```

Epoch 6. Loss: 0.585307060391, Train\_acc 0.8701, Test\_acc 0.8753  
Epoch 7. Loss: 0.586430716543, Train\_acc 0.874333, Test\_acc 0.8786  
Epoch 8. Loss: 0.559677588725, Train\_acc 0.877017, Test\_acc 0.8807  
Epoch 9. Loss: 0.50624773834, Train\_acc 0.880817, Test\_acc 0.884

## Using the model for prediction

Let's make it more intuitive by picking 10 random data points from the test set and use the trained model for predictions.

```
In [21]: # Define the function to do prediction
def model_predict(net,data):
    output = net(data)
    return nd.argmax(output, axis=1)

# Let's sample 10 random data points from the test set
sample_data = mx.gluon.data.DataLoader(mnist_test, 10, shuffle=True)
for i, (data, label) in enumerate(sample_data):
    data = data.as_in_context(ctx)
    print(data.shape)
    im = nd.transpose(data,(1,0,2,3))
    im = nd.reshape(im,(28,10*28,1))
    imtiles = nd.tile(im, (1,1,3))

    plt.imshow(imtiles.asnumpy())
    plt.show()
    pred=model_predict(net,data.reshape((-1,784)))
    print('model predictions are:', pred)
    break
```

(10, 28, 28, 1)



```
model predictions are:
[ 9.  5.  6.  6.  8.  5.  6.  6.  7.  3.]
<NDArray 10 @cpu(0)>
```

## Conclusion

Jeepers. We can get nearly 90% accuracy at this task just by training a linear model for a few seconds! You might reasonably conclude that this problem is too easy to be taken seriously by experts.

But until recently, many papers (Google Scholar says 13,800) were published using results obtained on this data. Even this year, I reviewed a paper whose primary achievement was an (imagined) improvement in performance. While MNIST can be a nice toy dataset for testing new ideas, we don't recommend writing papers with it.

## Next

## Softmax regression with gluon

For whinges or inquiries, [open an issue on GitHub](#).

In [22]:

## Multiclass logistic regression with `gluon`

Now that we've built a [logistic regression model from scratch](#), let's make this more efficient with `gluon`. If you completed the corresponding chapters on linear regression, you might be tempted rest your eyes a little in this one. We'll be using `gluon` in a rather similar way and since the interface is reasonably well designed, you won't have to do much work. To keep you awake we'll introduce a few subtle tricks.

Let's start by importing the standard packages.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
from mxnet import gluon
import numpy as np
```

### Set the context

Now, let's set the context. In the linear regression tutorial we did all of our computation on the cpu (`mx.cpu()`) just to keep things simple. When you've got 2-dimensional data and scalar labels, a smartwatch can probably handle the job. Already, in this tutorial we'll be working with a considerably larger dataset. If you happen to be running this code on a server with a GPU and installed the GPU-enabled version of MXNet (or remembered to build MXNet with `CUDA=1`), you might want to substitute the following line for its commented-out counterpart.

```
In [2]: # Set the context to CPU
ctx = mx.cpu()

# To set the context to GPU use this
# ctx = mx.gpu()
```

### The MNIST Dataset

We won't suck up too much wind describing the MNIST dataset for a second time. If you're unfamiliar with the dataset and are reading these chapters out of sequence, take a look at the data section in the previous chapter on [softmax regression from scratch](#).

We'll load up data iterators corresponding to the training and test splits of MNIST dataset.

```
In [3]: batch_size = 64
        num_inputs = 784
        num_outputs = 10
        def transform(data, label):
            return data.astype(np.float32)/255, label.astype(np.float32)
        train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
                                transform=transform),
                                batch_size, shuffle=True)
        test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
                                transform=transform),
                                batch_size, shuffle=False)
```

We're also going to want to load up an iterator with *test* data. After we train on the training dataset we're going to want to test our model on the test data. Otherwise, for all we know, our model could be doing something stupid (or treacherous?) like memorizing the training examples and regurgitating the labels on command.

## Multiclass Logistic Regression

Now we're going to define our model. Remember from *our tutorial on linear regression with ``gluon``* <./P02-C02-linear-regression-gluon>`\_\_ that we add `Dense` layers by calling `net.add(gluon.nn.Dense(num_outputs))`. This leaves the parameter shapes under-specified, but `gluon` will infer the desired shapes the first time we pass real data through the network.

```
In [4]: net = gluon.nn.Sequential()
        with net.name_scope():
            net.add(gluon.nn.Dense(num_outputs))
```

## Parameter initialization

As before, we're going to register an initializer for our parameters. Remember that `gluon` doesn't even know what shape the parameters have because we never specified the input dimension. The parameters will get initialized during the first call to the forward method.

```
In [5]: net.collect_params().initialize(mx.init.Normal(sigma=1.), ctx=ctx)
```

## Softmax Cross Entropy Loss

Note, we didn't have to include the softmax layer because MXNet's has an efficient function that simultaneously computes the softmax activation and cross-entropy loss. However, if ever need to get the output probabilities,

```
In [6]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

# Optimizer

And let's instantiate an optimizer to make our updates

```
In [7]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

## Evaluation Metric

This time, let's simplify the evaluation code by relying on MXNet's built-in `metric` package.

```
In [8]: def evaluate_accuracy(data_iterator, net):
    acc = mx.metric.Accuracy()
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1,784))
        label = label.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

Because we initialized our model randomly, and because roughly one tenth of all examples belong to each of the ten classes, we should have an accuracy in the ball park of .10.

```
In [9]: evaluate_accuracy(test_data, net)
```

```
Out[9]: 0.1177
```

## Execute training loop

```
In [10]: epochs = 4
moving_loss = 0.
smoothing_constant = .01
niter = 0

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx).reshape((-1,784))
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output, label)
        loss.backward()
        trainer.step(batch_size)

        #####
        # Keep a moving average of the losses
        #####
        niter += 1
        moving_loss = (1 - smoothing_constant) * moving_loss + (smoothing_constant) *
nd.mean(loss).asscalar()
        est_loss = moving_loss/(1-(1-smoothing_constant)**niter)

    test_accuracy = evaluate_accuracy(test_data, net)
    train_accuracy = evaluate_accuracy(train_data, net)
```

```
print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, est_loss, train_accuracy, test_accuracy))
```

```
Epoch 0. Loss: 1.09775956875, Train_acc 0.792066666667, Test_acc 0.8023  
Epoch 1. Loss: 0.836000709358, Train_acc 0.837816666667, Test_acc 0.8452  
Epoch 2. Loss: 0.672541667936, Train_acc 0.856316666667, Test_acc 0.8616  
Epoch 3. Loss: 0.630035108029, Train_acc 0.866616666667, Test_acc 0.8731
```

```
In [11]: import matplotlib.pyplot as plt  
  
def model_predict(net,data):  
    output = net(data)  
    return nd.argmax(output, axis=1)  
  
# Let's sample 10 random data points from the test set  
sample_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,  
transform=transform),  
                                         10, shuffle=True)  
for i, (data, label) in enumerate(sample_data):  
    data = data.as_in_context(ctx)  
    print(data.shape)  
    im = nd.transpose(data,(1,0,2,3))  
    im = nd.reshape(im,(28,10*28,1))  
    imtiles = nd.tile(im, (1,1,3))  
  
    plt.imshow(imtiles.asnumpy())  
    plt.show()  
    pred=model_predict(net,data.reshape((-1,784)))  
    print('model predictions are:', pred)  
    break
```

```
(10, 28, 28, 1)
```



```
model predictions are:  
[ 3.  9.  1.  9.  2.  0.  5.  3.  1.  4.]  
<NDArray 10 @cpu(0)>
```

## Next

Overfitting and regularization from scratch

For whinges or inquiries, [open an issue on GitHub](#).



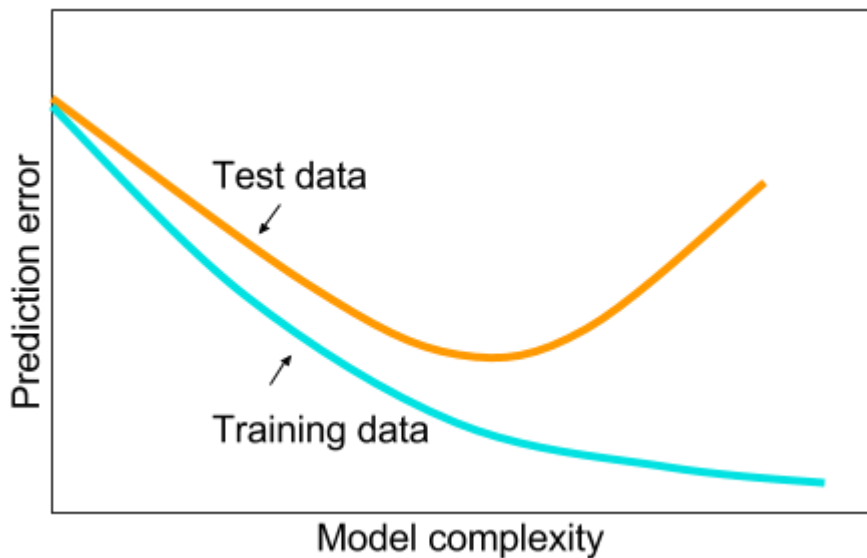
## Overfitting and regularization

In [the last tutorial](#), we introduced the task of multiclass classification. We showed how you can tackle this problem with a linear model called logistic regression. Owing to some amount of randomness, you might get slightly different results, but when I ran the notebook, the model achieved 88.1% accuracy on the training data and actually did slightly (but not significantly) better on the test data than on the training data.

Not every algorithm that performs well on training data will also perform well on test data. Take, for example, a trivial algorithm that memorizes its inputs and stores the associated labels. This model would have 100% accuracy on training data but would have no way of making any prediction at all on previously unseen data.

The goal of supervised learning is to produce models that *generalize* to previously unseen data. When a model achieves low error on training data but performs much worse on test data, we say that the model has *overfit*. This means that the model has caught on to idiosyncratic features of the training data (e.g. one “2” happened to have a white pixel in the top-right corner), but hasn’t really picked up on general patterns.

We can express this more formally. The quantity we really care about is the test error  $e$ . Because this quantity reflects the error of our model when generalized to previously unseen data, we commonly call it the *generalization error*. When we have simple models and abundant data, we expect the generalization error to resemble the training error. When we work with more complex models and fewer examples, we expect the training error to go down but the generalization gap to grow. Fixing the size of the dataset, the following graph should give you some intuition about what we generally expect to see.



What precisely constitutes model complexity is a complex matter. Many factors govern whether a model will generalize well. For example a model with more parameters might be considered more complex. A model whose parameters can take a wider range of values might be more complex. Often with neural networks, we think of a model that takes more training steps as more complex, and one subject to *early stopping* as less complex.

It can be difficult to compare the complexity among members of very different model classes (say decision trees versus neural networks). Researchers in the field of statistical learning theory have developed a large body of mathematical analysis that formulizes the notion of model complexity and provides guarantees on the generalization error for simple classes of models. *We won't get into this theory but may delve deeper in a future chapter.* For now a simple rule of thumb is quite useful: A model that can readily explain *arbitrary* facts is what statisticians view as complex, whereas one that has only a limited expressive power but still manages to explain the data well is probably closer to the truth. In philosophy this is closely related to Popper's criterion of [falsifiability](#) of a scientific theory: a theory is good if it fits data and if there are specific tests which can be used to disprove it. This is important since all statistical estimation is [post hoc](#), i.e. we estimate after we observe the facts, hence vulnerable to the associated fallacy. Ok, enough of philosophy, let's get to more tangible issues.

To give you some intuition in this chapter, we'll focus on a few factors that tend to influence the generalizability of a model class:

1. **The number of tunable parameters.** When the number of tunable parameters, sometimes denoted as the number of degrees of freedom, is large, models tend to be more susceptible to overfitting.
2. **The values taken by the parameters.** When weights can take a wider range of values, models can be more susceptible to over fitting.
3. **The number of training examples.** It's trivially easy to overfit a dataset containing only one or two examples even if your model is simple. But overfitting a dataset with millions of

examples requires an extremely flexible model.

When classifying handwritten digits before, we didn't overfit because our 60,000 training examples far outnumbered the  $784 \times 10 = 7,840$  weights plus 10 bias terms, which gave us far fewer parameters than training examples. Let's see how things can go wrong. We begin with our import ritual.

```
In [1]: from __future__ import print_function
import mxnet as mx
import mxnet.ndarray as nd
from mxnet import autograd
import numpy as np
ctx = mx.cpu()
mx.random.seed(1)

# for plotting purposes
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

## Load the MNIST dataset

```
In [2]: mnist = mx.test_utils.get_mnist()
num_examples = 1000
batch_size = 64
train_data = mx.gluon.data.DataLoader(
    mx.gluon.data.ArrayDataset(mnist["train_data"][:num_examples],
                               mnist["train_label"][:num_examples].astype(np.float32)),
    batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(
    mx.gluon.data.ArrayDataset(mnist["test_data"][:num_examples],
                               mnist["test_label"][:num_examples].astype(np.float32)),
    batch_size, shuffle=False)
```

## Allocate model parameters and define model

We pick a simple linear model  $f(x) = Wx + b$  with subsequent softmax, i.e.  $p(y|x) \propto \exp(f(x)_y)$ . This is about as simple as it gets.

```
In [3]: W = nd.random_normal(shape=(784,10))
b = nd.random_normal(shape=10)

params = [W, b]

for param in params:
    param.attach_grad()

def net(X):
    y_linear = nd.dot(X, W) + b
    yhat = nd.softmax(y_linear, axis=1)
    return yhat
```

## Define loss function and optimizer

A sensible thing to do is to minimize the negative log-likelihood of the data, i.e.  $-\log p(y|x)$ . Statisticians have proven that this is actually the most *efficient* estimator, i.e. the one that makes the most use of the data provided. This is why it is so popular.

```
In [4]: def cross_entropy(yhat, y):
        return - nd.sum(y * nd.log(yhat), axis=0, exclude=True)

def SGD(params, lr):
    for param in params:
        param[:] = param - lr * param.grad
```

## Write evaluation loop to calculate accuracy

Ultimately we want to recognize digits. This is a bit different from knowing the *probability* of a digit - when given an image we need to *decide* what digit we are seeing, *regardless* of how uncertain we are. Hence we measure the number of actual misclassifications.

For diagnosis purposes, it is always a good idea to calculate the average loss function.

```
In [5]: def evaluate_accuracy(data_iterator, net):
        numerator = 0.
        denominator = 0.
        loss_avg = 0.
        for i, (data, label) in enumerate(data_iterator):
            data = data.as_in_context(ctx).reshape((-1,784))
            label = label.as_in_context(ctx)
            label_one_hot = nd.one_hot(label, 10)
            output = net(data)
            loss = cross_entropy(output, label_one_hot)
            predictions = nd.argmax(output, axis=1)
            numerator += nd.sum(predictions == label)
            denominator += data.shape[0]
            loss_avg = loss_avg*i/(i+1) + nd.mean(loss).asscalar()/(i+1)
        return (numerator / denominator).asscalar(), loss_avg
```

## Write a utility function to plot the learning curves

Just to visualize how loss functions and accuracy changes over the number of iterations.

```
In [6]: def plot_learningcurves(loss_tr, loss_ts, acc_tr, acc_ts):
        xs = list(range(len(loss_tr)))

        f = plt.figure(figsize=(12,6))
        fg1 = f.add_subplot(121)
        fg2 = f.add_subplot(122)

        fg1.set_xlabel('epoch', fontsize=14)
        fg1.set_title('Comparing loss functions')
        fg1.semilogy(xs, loss_tr)
        fg1.semilogy(xs, loss_ts)
        fg1.grid(True, which="both")
```

```

fg1.legend(['training loss', 'testing loss'], fontsize=14)

fg2.set_title('Comparing accuracy')
fg1.set_xlabel('epoch', fontsize=14)
fg2.plot(xs, acc_tr)
fg2.plot(xs, acc_ts)
fg2.grid(True, which="both")
fg2.legend(['training accuracy', 'testing accuracy'], fontsize=14)

```

## Execute training loop

We now train the model until there is no further improvement. Our approach is actually a bit naive since we will keep the learning rate unchanged but it fits the purpose (we want to keep the code simple and avoid confusing anyone with further tricks for adjusting learning rate schedules).

```

In [7]: epochs = 1000
        moving_loss = 0.
        niter=0

        loss_seq_train = []
        loss_seq_test = []
        acc_seq_train = []
        acc_seq_test = []

        for e in range(epochs):
            for i, (data, label) in enumerate(train_data):
                data = data.as_in_context(ctx).reshape((-1, 784))
                label = label.as_in_context(ctx)
                label_one_hot = nd.one_hot(label, 10)
                with autograd.record():
                    output = net(data)
                    loss = cross_entropy(output, label_one_hot)
                loss.backward()
                SGD(params, .001)

                #####
                # Keep a moving average of the losses
                #####
                niter +=1
                moving_loss = .99 * moving_loss + .01 * nd.mean(loss).asscalar()
                est_loss = moving_loss/(1-0.99**niter)

            test_accuracy, test_loss = evaluate_accuracy(test_data, net)
            train_accuracy, train_loss = evaluate_accuracy(train_data, net)

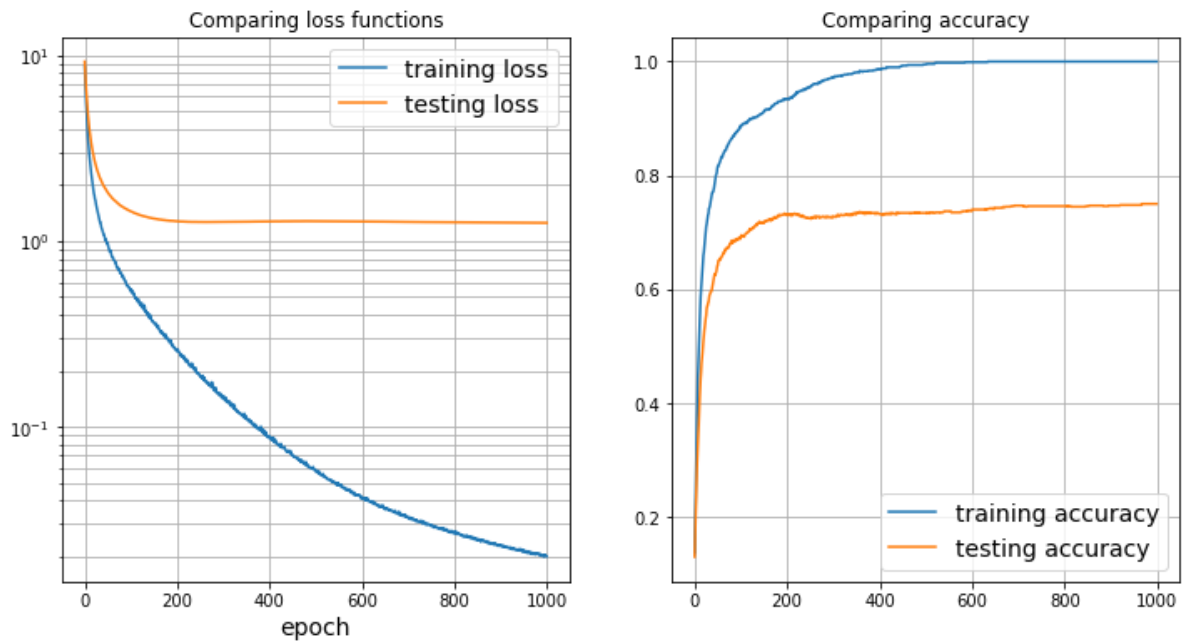
            # save them for later
            loss_seq_train.append(train_loss)
            loss_seq_test.append(test_loss)
            acc_seq_train.append(train_accuracy)
            acc_seq_test.append(test_accuracy)

            if e % 100 == 99:
                print("Completed epoch %. Train Loss: %, Test Loss %, Train_acc %, Test_acc
                %s" %
                    (e+1, train_loss, test_loss, train_accuracy, test_accuracy))

        ## Plotting the Learning curves
        plot_learningcurves(loss_seq_train, loss_seq_test, acc_seq_train, acc_seq_test)

```

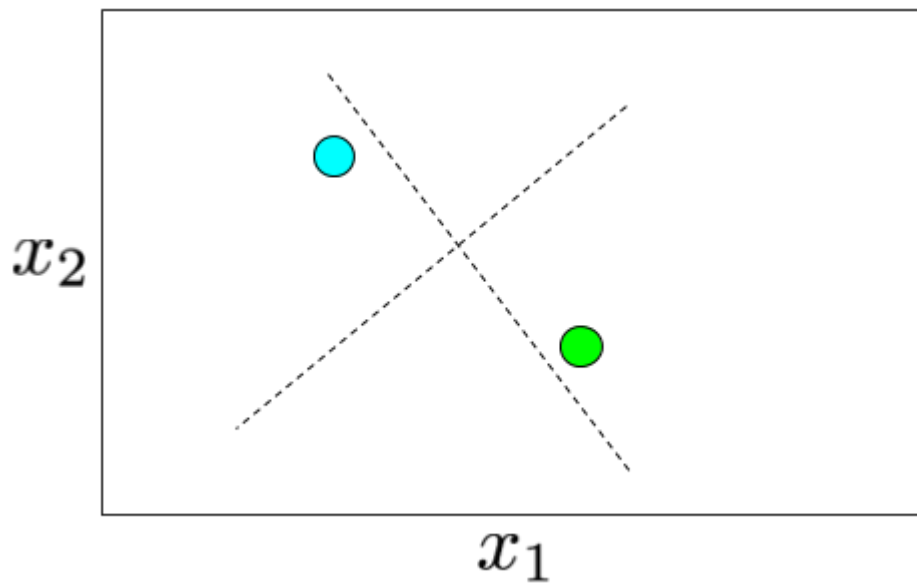
Completed epoch 100. Train Loss: 0.541354929097, Test Loss 1.4438945204, Train\_acc 0.887, Test\_acc 0.69  
Completed epoch 200. Train Loss: 0.259953523055, Test Loss 1.27532487363, Train\_acc 0.935, Test\_acc 0.731  
Completed epoch 300. Train Loss: 0.142286404734, Test Loss 1.26391389966, Train\_acc 0.973, Test\_acc 0.728  
Completed epoch 400. Train Loss: 0.0880716806278, Test Loss 1.27251135558, Train\_acc 0.987, Test\_acc 0.733  
Completed epoch 500. Train Loss: 0.0578876060899, Test Loss 1.27707066014, Train\_acc 0.995, Test\_acc 0.734  
Completed epoch 600. Train Loss: 0.0420041342732, Test Loss 1.27324004471, Train\_acc 0.999, Test\_acc 0.739  
Completed epoch 700. Train Loss: 0.0328008912038, Test Loss 1.2654389888, Train\_acc 1.0, Test\_acc 0.747  
Completed epoch 800. Train Loss: 0.026653088571, Test Loss 1.25905798748, Train\_acc 1.0, Test\_acc 0.746  
Completed epoch 900. Train Loss: 0.0231189786573, Test Loss 1.25432690606, Train\_acc 1.0, Test\_acc 0.747  
Completed epoch 1000. Train Loss: 0.0200853450806, Test Loss 1.25094169378, Train\_acc 1.0, Test\_acc 0.75



## What Happened?

By the 700th epoch, our model achieves 100% accuracy on the training data. However, it only classifies 75% of the test examples accurately. This is a clear case of overfitting. At a high level, there's a reason this went wrong. Because we have 7450 parameters and only 1000 data points, there are actually many settings of the parameters that could produce 100% accuracy on training data.

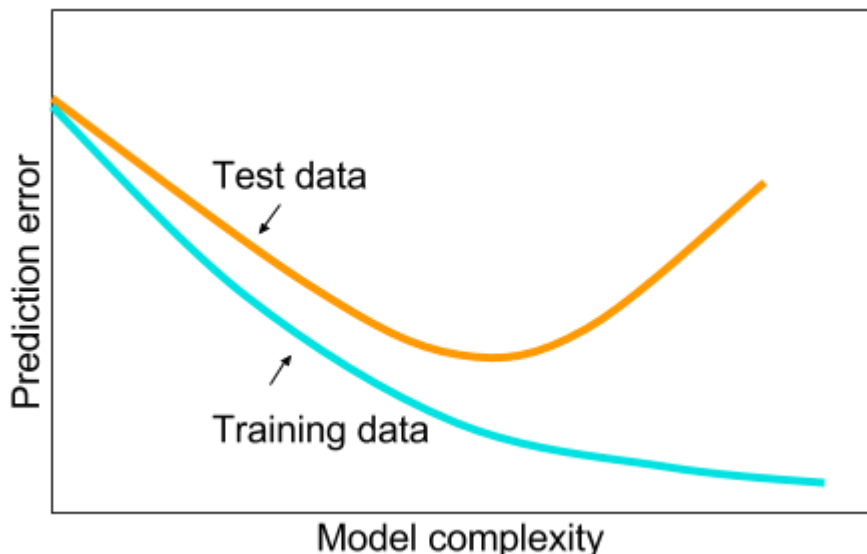
To get some intuition imagine that we wanted to fit a dataset with 2 dimensional data and 2 data points. Our model has three degrees of freedom, and thus for any dataset can find an arbitrary number of separators that will perfectly classify our training points. Note below that we can produce completely orthogonal separators that both classify our training data perfectly. Even if it seems preposterous that they could both describe our training data well.



## Regularization

Now that we've characterized the problem of overfitting, we can begin talking about some solutions. Broadly speaking the family of techniques geared towards mitigating overfitting are referred to as *regularization*. The core idea is this: when a model is overfitting, its training error is substantially lower than its test error. We're already doing as well as we possibly can on the training data, but our test data performance leaves something to be desired. Typically, regularization techniques attempt to trade off our training performance in exchange for lowering our test error.

There are several straightforward techniques we might employ. Given the intuition from the previous chart, we might attempt to make our model less complex. One way to do this would be to lower the number of free parameters. For example, we could throw away some subset of our input features (and thus the corresponding parameters) that we thought were least informative.



Another approach is to limit the values that our weights might take. One common approach is to force the weights to take small values. [give more intuition with example of polynomial curve fitting] We can accomplish this by changing our optimization objective to penalize the value of our weights. The most popular regularizer is the  $\ell_2^2$  norm. For linear models,  $\ell_2^2$  regularization has the additional benefit that it makes the solution unique, even when our model is overparametrized.

$$\sum_i (\hat{y} - y)^2 + \lambda \|\mathbf{w}\|_2^2$$

Here,  $\|\mathbf{w}\|$  is the  $\ell_2$  norm and  $\lambda$  is a hyper-parameter that determines how aggressively we want to push the weights towards 0. In code, we can express the  $\ell_2^2$  penalty succinctly:

```
In [8]: def l2_penalty(params):
        penalty = nd.zeros(shape=1)
        for param in params:
            penalty = penalty + nd.sum(param ** 2)
        return penalty
```

## Re-initializing the parameters

Just for good measure to ensure that the results in the second training run don't depend on the first one.

```
In [9]: for param in params:
        param[:] = nd.random_normal(shape=param.shape)
```

## Training L2-regularized logistic regression

```
In [10]: epochs = 1000
        moving_loss = 0.
        l2_strength = .1
        niter=0

        loss_seq_train = []
        loss_seq_test = []
        acc_seq_train = []
        acc_seq_test = []

        for e in range(epochs):
            for i, (data, label) in enumerate(train_data):
                data = data.as_in_context(ctx).reshape((-1,784))
                label = label.as_in_context(ctx)
                label_one_hot = nd.one_hot(label, 10)
                with autograd.record():
                    output = net(data)
                    loss = nd.sum(cross_entropy(output, label_one_hot)) + l2_strength *
l2_penalty(params)
                    loss.backward()
                    SGD(params, .001)

                #####
```



```

# Keep a moving average of the losses
#####
niter +=1
moving_loss = .99 * moving_loss + .01 * nd.mean(loss).asscalar()
est_loss = moving_loss/(1-0.99**niter)

test_accuracy, test_loss = evaluate_accuracy(test_data, net)
train_accuracy, train_loss = evaluate_accuracy(train_data, net)

# save them for later
loss_seq_train.append(train_loss)
loss_seq_test.append(test_loss)
acc_seq_train.append(train_accuracy)
acc_seq_test.append(test_accuracy)

if e % 100 == 99:
    print("Completed epoch %s. Train Loss: %s, Test Loss %s, Train_acc %s, Test_acc %s" %
          (e+1, train_loss, test_loss, train_accuracy, test_accuracy))

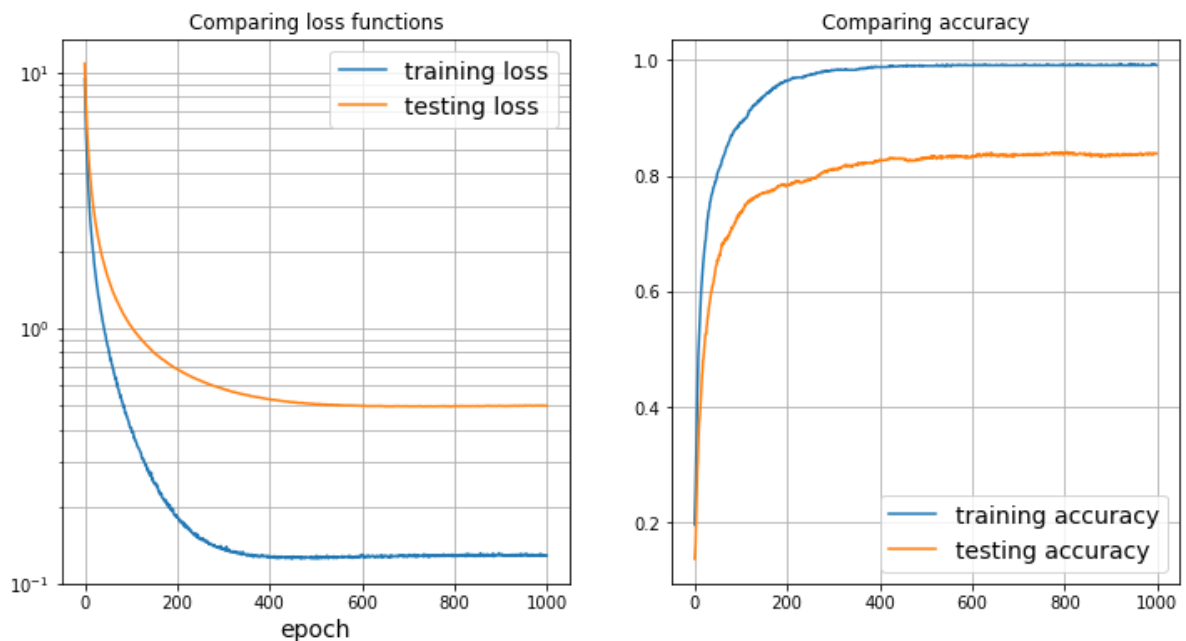
```

```

Completed epoch 100. Train Loss: 0.408492157236, Test Loss 1.02146248147, Train_acc 0.892, Test_acc 0.734
Completed epoch 200. Train Loss: 0.183608927764, Test Loss 0.689952459186, Train_acc 0.965, Test_acc 0.783
Completed epoch 300. Train Loss: 0.13577057654, Test Loss 0.576444735751, Train_acc 0.982, Test_acc 0.811
Completed epoch 400. Train Loss: 0.126813526731, Test Loss 0.524908654392, Train_acc 0.988, Test_acc 0.826
Completed epoch 500. Train Loss: 0.126559415367, Test Loss 0.503038179129, Train_acc 0.99, Test_acc 0.83
Completed epoch 600. Train Loss: 0.129612701479, Test Loss 0.494899718091, Train_acc 0.991, Test_acc 0.833
Completed epoch 700. Train Loss: 0.127363444306, Test Loss 0.49323198013, Train_acc 0.991, Test_acc 0.835
Completed epoch 800. Train Loss: 0.129239805974, Test Loss 0.493483386934, Train_acc 0.991, Test_acc 0.839
Completed epoch 900. Train Loss: 0.12854804704, Test Loss 0.493301877752, Train_acc 0.991, Test_acc 0.837
Completed epoch 1000. Train Loss: 0.128394702449, Test Loss 0.494407396764, Train_acc 0.991, Test_acc 0.838

```

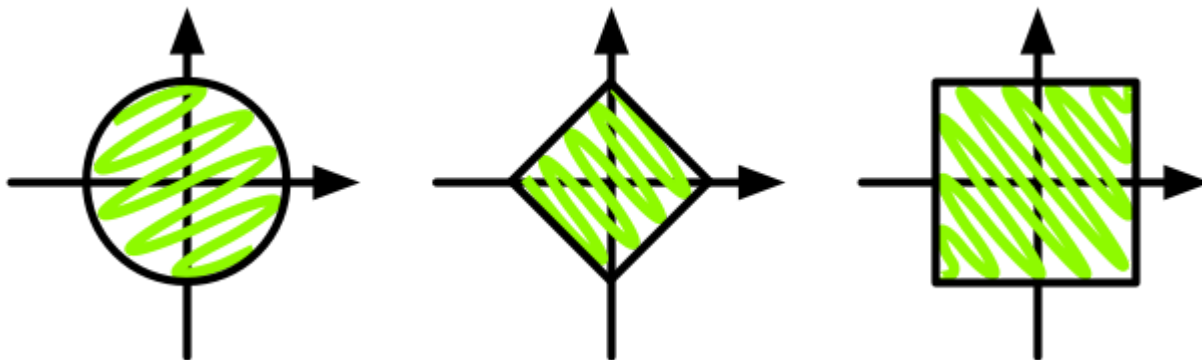
In [11]: `## Plotting the Learning curves`  
`plot_learningcurves(loss_seq_train,loss_seq_test,acc_seq_train,acc_seq_test)`



# Analysis

By adding  $L_2$  regularization we were able to increase the performance on test data from 75% accuracy to 83% accuracy. That's a 32% reduction in error. In a lot of applications, this big an improvement can make the difference between a viable product and useless system. Note that  $L_2$  regularization is just one of many ways of controlling capacity. Basically we assumed that small weight values are good. But there are many more ways to constrain the values of the weights:

- We could require that the total sum of the weights is small. That is what  $L_1$  regularization does via the penalty  $\sum_i |w_i|$ .
- We could require that the largest weight is not too large. This is what  $L_\infty$  regularization does via the penalty  $\max_i |w_i|$ .
- We could require that the number of nonzero weights is small, i.e. that the weight vectors are *sparse*. This is what the  $L_0$  penalty does, i.e.  $\sum_i I\{w_i \neq 0\}$ . This penalty is quite difficult to deal with explicitly since it is nonsmooth. There is a lot of research that shows how to solve this problem approximately using an  $L_1$  penalty.



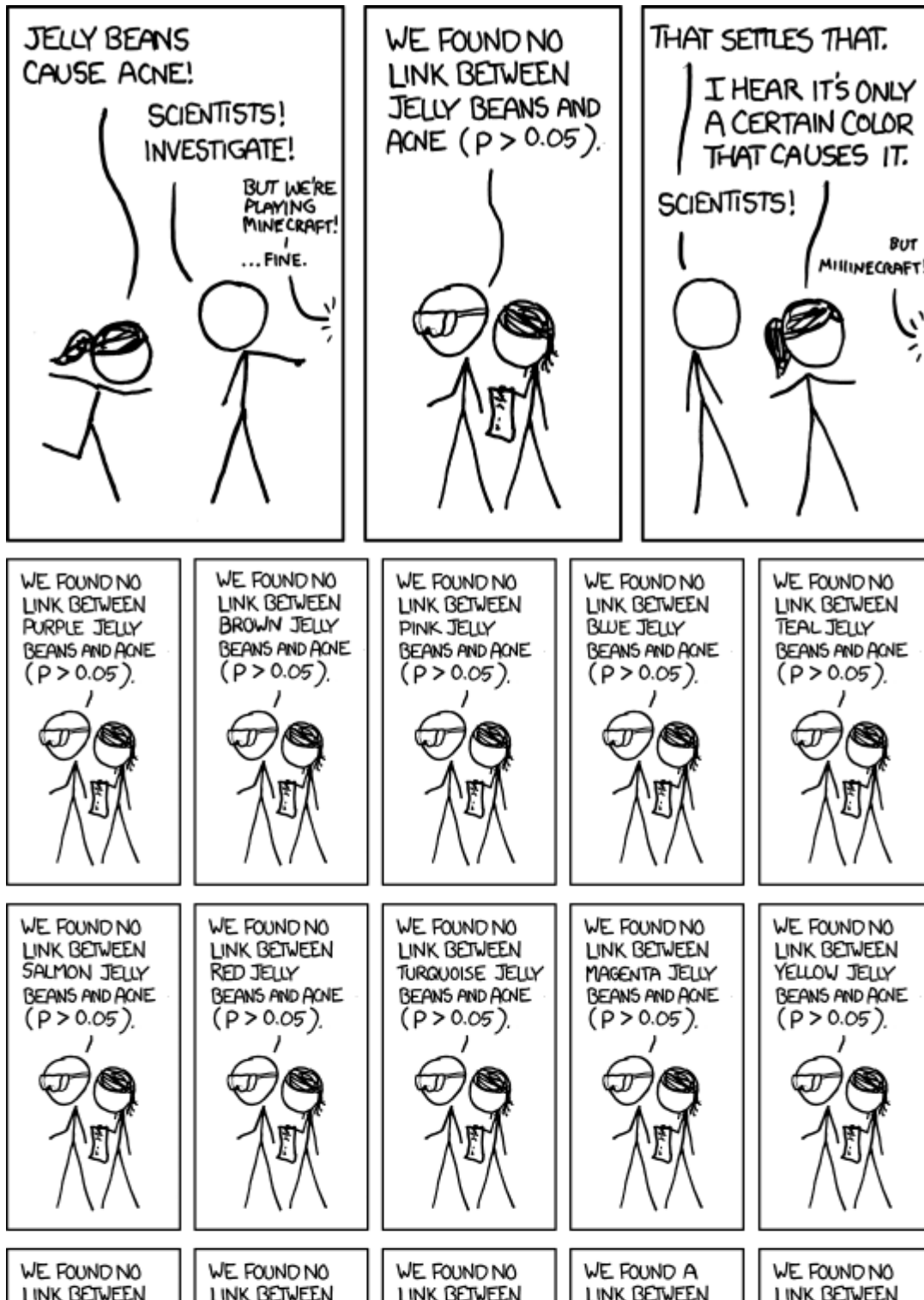
From left to right:  $L_2$  regularization, which constrains the parameters to a ball,  $L_1$  regularization, which constrains the parameters to a diamond (for lack of a better name, this is often referred to as an  $L_1$ -ball), and  $L_\infty$  regularization, which constrains the parameters to a hypercube.

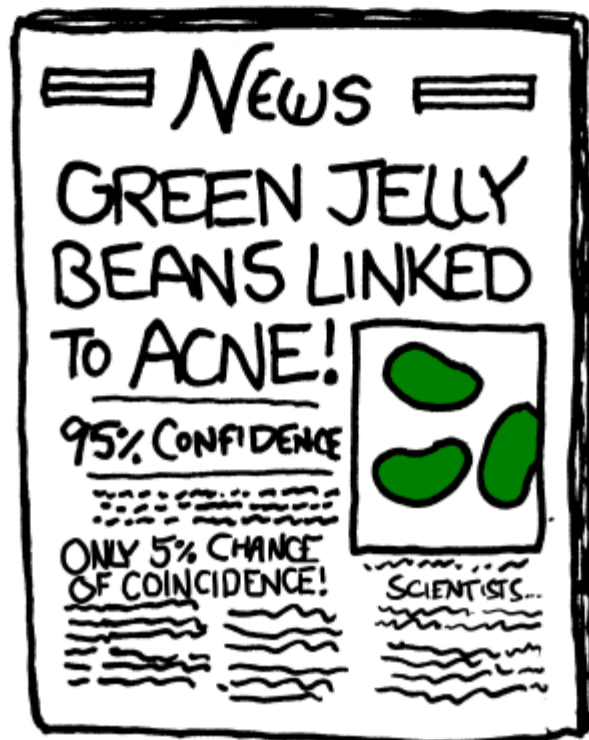
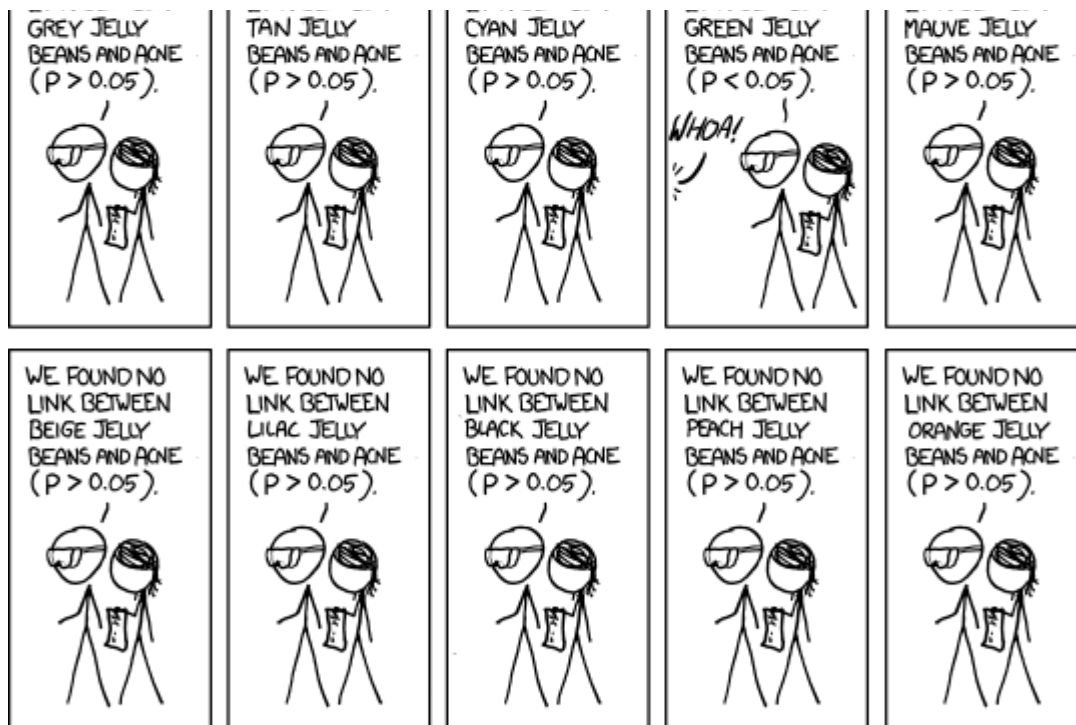
All of this raises the question of **why** regularization is any good. After all, choice is good and giving our model more flexibility *ought* to be better (e.g. there are plenty of papers which show improvements on ImageNet using deeper networks). What is happening is somewhat more subtle. Allowing for many different parameter values allows our model to cherry pick a combination that is *just right* for all the training data it sees, without really learning the underlying mechanism. Since our observations are likely noisy, this means that we are trying to approximate the errors at least as much as we're learning what the relation between data and labels actually is. There is an entire field of statistics devoted to this issue - Statistical Learning Theory. For now, a few simple rules of thumb suffice:

- Fewer parameters tend to be better than more parameters.

- Better engineering for a specific problem that takes the actual problem into account will lead to better models, due to the prior knowledge that data scientists have about the problem at hand.
- $L_2$  is easier to optimize for than  $L_1$ . In particular, many optimizers will not work well out of the box for  $L_1$ . Using the latter requires something called *proximal operators*.
- Dropout and other methods to make the model robust to perturbations in the data often work better than off-the-shelf  $L_2$  regularization.

We conclude with an [XKCD Cartoon](#) which captures the entire situation more succinctly than the proceeding paragraph.





## Next

Overfitting and regularization with gluon

For whinges or inquiries, [open an issue on GitHub](#).

## Overfitting and regularization (with `gluon`)

Now that we've built a [regularized logistic regression model from scratch](#), let's make this more efficient with `gluon`. We recommend that you read that section for a description as to why regularization is a good idea. As always, we begin by loading libraries and some data.

[REFINED DRAFT - RELEASE STAGE: CATFOOD]

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import autograd
from mxnet import gluon
import mxnet.ndarray as nd
import numpy as np
ctx = mx.cpu()

# for plotting purposes
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

## The MNIST Dataset

```
In [2]: mnist = mx.test_utils.get_mnist()
num_examples = 1000
batch_size = 64
train_data = mx.gluon.data.DataLoader(
    mx.gluon.data.ArrayDataset(mnist["train_data"][:num_examples],
                               mnist["train_label"][:num_examples].astype(np.float32)),
    batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(
    mx.gluon.data.ArrayDataset(mnist["test_data"][:num_examples],
                               mnist["test_label"][:num_examples].astype(np.float32)),
    batch_size, shuffle=False)
```

## Multiclass Logistic Regression

```
In [3]: net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Dense(10))
```

## Parameter initialization

```
In [4]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

# Softmax Cross Entropy Loss

```
In [5]: loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Optimizer

By default `gluon` tries to keep the coefficients from diverging by using a *weight decay* penalty. So, to get the real overfitting experience we need to switch it off. We do this by passing `'wd': 0.0` when we instantiate the trainer.

```
In [6]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.01, 'wd': 0.0})
```

## Evaluation Metric

```
In [7]: def evaluate_accuracy(data_iterator, net, loss_fun):
    acc = mx.metric.Accuracy()
    loss_avg = 0.
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1,784))
        label = label.as_in_context(ctx)
        output = net(data)
        loss = loss_fun(output, label)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
        loss_avg = loss_avg*i/(i+1) + nd.mean(loss).asscalar()/(i+1)
    return acc.get()[1], loss_avg

def plot_learningcurves(loss_tr,loss_ts, acc_tr,acc_ts):
    xs = list(range(len(loss_tr)))

    f = plt.figure(figsize=(12,6))
    fg1 = f.add_subplot(121)
    fg2 = f.add_subplot(122)

    fg1.set_xlabel('epoch',fontsize=14)
    fg1.set_title('Comparing loss functions')
    fg1.semilogy(xs, loss_tr)
    fg1.semilogy(xs, loss_ts)
    fg1.grid(True,which="both")

    fg1.legend(['training loss', 'testing loss'],fontsize=14)

    fg2.set_title('Comparing accuracy')
    fg1.set_xlabel('epoch',fontsize=14)
    fg2.plot(xs, acc_tr)
    fg2.plot(xs, acc_ts)
    fg2.grid(True,which="both")
    fg2.legend(['training accuracy', 'testing accuracy'],fontsize=14)
```

## Execute training loop

```
In [8]: epochs = 700
    moving_loss = 0.
    niter=0
```

```

loss_seq_train = []
loss_seq_test = []
acc_seq_train = []
acc_seq_test = []

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx).reshape((-1,784))
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            cross_entropy = loss(output, label)
            cross_entropy.backward()
            trainer.step(data.shape[0])

        #####
        # Keep a moving average of the Losses
        #####
        niter +=1
        moving_loss = .99 * moving_loss + .01 * nd.mean(cross_entropy).asscalar()
        est_loss = moving_loss/(1-0.99**niter)

    test_accuracy, test_loss = evaluate_accuracy(test_data, net, loss)
    train_accuracy, train_loss = evaluate_accuracy(train_data, net, loss)

    # save them for later
    loss_seq_train.append(train_loss)
    loss_seq_test.append(test_loss)
    acc_seq_train.append(train_accuracy)
    acc_seq_test.append(test_accuracy)

    if e % 20 == 0:
        print("Completed epoch %. Train Loss: %, Test Loss %, Train_acc %, Test_acc
        %s" %
              (e+1, train_loss, test_loss, train_accuracy, test_accuracy))

## Plotting the Learning curves
plot_learningcurves(loss_seq_train,loss_seq_test,acc_seq_train,acc_seq_test)

```

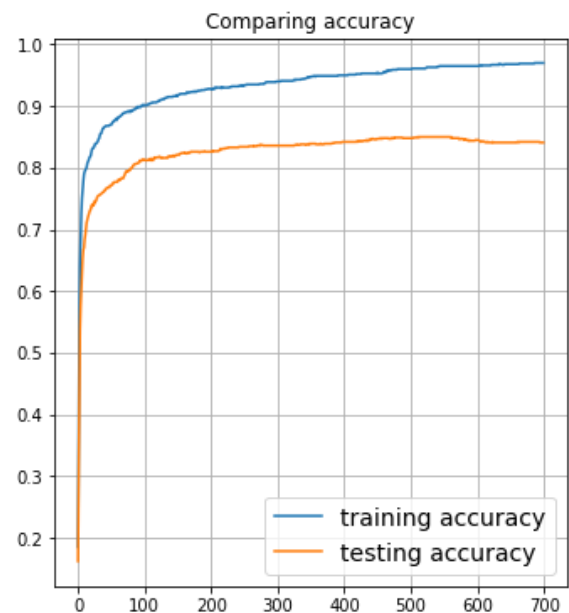
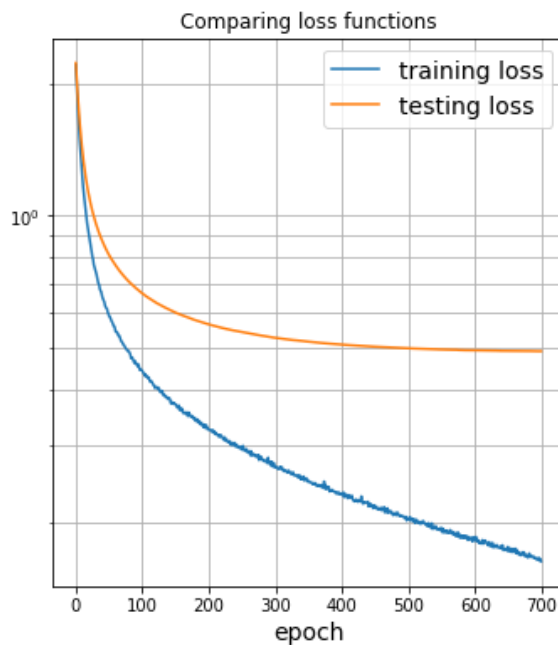
```

Completed epoch 1. Train Loss: 2.20206177235, Test Loss 2.22747650743, Train_acc 0.185,
Test_acc 0.162
Completed epoch 21. Train Loss: 0.892146475613, Test Loss 1.10410450026, Train_acc 0.825,
Test_acc 0.737
Completed epoch 41. Train Loss: 0.650202345103, Test Loss 0.872422374785, Train_acc 0.866,
Test_acc 0.763
Completed epoch 61. Train Loss: 0.542248453945, Test Loss 0.76846794039, Train_acc 0.881,
Test_acc 0.778
Completed epoch 81. Train Loss: 0.489410074428, Test Loss 0.705862168223, Train_acc 0.893,
Test_acc 0.799
Completed epoch 101. Train Loss: 0.441433861852, Test Loss 0.663885675371, Train_acc
0.902, Test_acc 0.813
Completed epoch 121. Train Loss: 0.405532337725, Test Loss 0.633128682151, Train_acc
0.908, Test_acc 0.817
Completed epoch 141. Train Loss: 0.383627502248, Test Loss 0.610212739557, Train_acc
0.915, Test_acc 0.818
Completed epoch 161. Train Loss: 0.362067368813, Test Loss 0.591509068385, Train_acc 0.92,
Test_acc 0.825
Completed epoch 181. Train Loss: 0.339679084718, Test Loss 0.576706366614, Train_acc
0.925, Test_acc 0.826
Completed epoch 201. Train Loss: 0.325259311125, Test Loss 0.564017400146, Train_acc
0.927, Test_acc 0.826
Completed epoch 221. Train Loss: 0.31234044861, Test Loss 0.553559621796, Train_acc 0.93,
Test_acc 0.831
Completed epoch 241. Train Loss: 0.297074861825, Test Loss 0.545117178932, Train_acc
0.932, Test_acc 0.833
Completed epoch 261. Train Loss: 0.2894834904, Test Loss 0.537920514122, Train_acc 0.935,
Test_acc 0.835
Completed epoch 281. Train Loss: 0.277775473893, Test Loss 0.531740020961, Train_acc
0.936, Test_acc 0.836
Completed epoch 301. Train Loss: 0.267405152321, Test Loss 0.525510722771, Train_acc 0.94,
Test_acc 0.836

```



Completed epoch 321. Train Loss: 0.26191042643, Test Loss 0.521078709513, Train\_acc 0.941, Test\_acc 0.836  
 Completed epoch 341. Train Loss: 0.2508082157, Test Loss 0.517240958288, Train\_acc 0.944, Test\_acc 0.838  
 Completed epoch 361. Train Loss: 0.244502888992, Test Loss 0.513158256188, Train\_acc 0.949, Test\_acc 0.838  
 Completed epoch 381. Train Loss: 0.239984786138, Test Loss 0.509872548282, Train\_acc 0.949, Test\_acc 0.839  
 Completed epoch 401. Train Loss: 0.230127763934, Test Loss 0.50716057606, Train\_acc 0.95, Test\_acc 0.842  
 Completed epoch 421. Train Loss: 0.222901177593, Test Loss 0.504965415224, Train\_acc 0.952, Test\_acc 0.842  
 Completed epoch 441. Train Loss: 0.21668790374, Test Loss 0.502555353567, Train\_acc 0.953, Test\_acc 0.847  
 Completed epoch 461. Train Loss: 0.213474844582, Test Loss 0.500988578424, Train\_acc 0.956, Test\_acc 0.848  
 Completed epoch 481. Train Loss: 0.209507481195, Test Loss 0.49926232174, Train\_acc 0.96, Test\_acc 0.849  
 Completed epoch 501. Train Loss: 0.202280035242, Test Loss 0.497560294345, Train\_acc 0.96, Test\_acc 0.848  
 Completed epoch 521. Train Loss: 0.198728222866, Test Loss 0.496033722535, Train\_acc 0.961, Test\_acc 0.85  
 Completed epoch 541. Train Loss: 0.195377404802, Test Loss 0.495215365663, Train\_acc 0.964, Test\_acc 0.85  
 Completed epoch 561. Train Loss: 0.188592088409, Test Loss 0.494167156518, Train\_acc 0.965, Test\_acc 0.849  
 Completed epoch 581. Train Loss: 0.184347858187, Test Loss 0.492983272299, Train\_acc 0.965, Test\_acc 0.845  
 Completed epoch 601. Train Loss: 0.181168745272, Test Loss 0.492297751829, Train\_acc 0.965, Test\_acc 0.845  
 Completed epoch 621. Train Loss: 0.176521503832, Test Loss 0.491771969944, Train\_acc 0.966, Test\_acc 0.841  
 Completed epoch 641. Train Loss: 0.174657453783, Test Loss 0.491076562554, Train\_acc 0.967, Test\_acc 0.841  
 Completed epoch 661. Train Loss: 0.173845630139, Test Loss 0.490820756182, Train\_acc 0.968, Test\_acc 0.842  
 Completed epoch 681. Train Loss: 0.167031467427, Test Loss 0.490575453267, Train\_acc 0.969, Test\_acc 0.842



## Regularization



Now let's see what this mysterious *weight decay* is all about. We begin with a bit of math. When we add an L2 penalty to the weights we are effectively adding  $\frac{\lambda}{2} \|w\|^2$  to the loss. Hence, every time we compute the gradient it gets an additional  $\lambda w$  term that is added to  $g_t$ , since this is the very derivative of the L2 penalty. As a result we end up taking a descent step not in the direction  $-\eta g_t$  but rather in the direction  $-\eta(g_t + \lambda w)$ . This effectively shrinks  $w$  at each step by  $\eta \lambda w$ , thus the name weight decay. To make this work in practice we just need to set the weight decay to something nonzero.

```
In [9]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx,
force_reinit=True)
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.01, 'wd': 0.001})

moving_loss = 0.
niter=0
loss_seq_train = []
loss_seq_test = []
acc_seq_train = []
acc_seq_test = []

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx).reshape((-1,784))
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            cross_entropy = loss(output, label)
        cross_entropy.backward()
        trainer.step(data.shape[0])

        #####
        # Keep a moving average of the losses
        #####
        niter +=1
        moving_loss = .99 * moving_loss + .01 * nd.mean(cross_entropy).asscalar()
        est_loss = moving_loss/(1-0.99**niter)

    test_accuracy, test_loss = evaluate_accuracy(test_data, net,loss)
    train_accuracy, train_loss = evaluate_accuracy(train_data, net, loss)

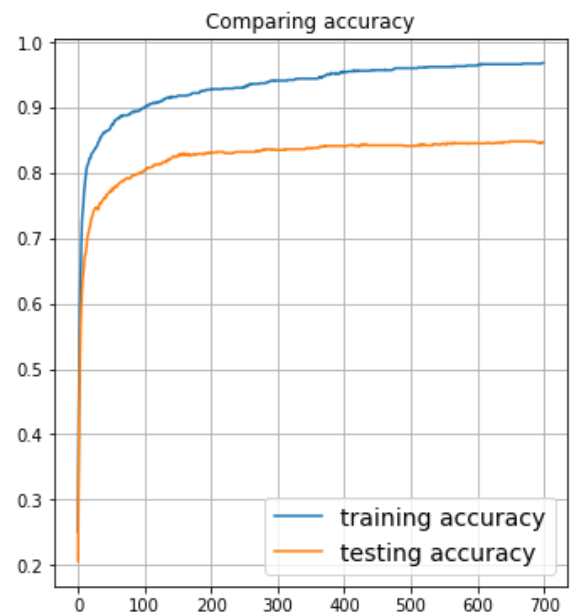
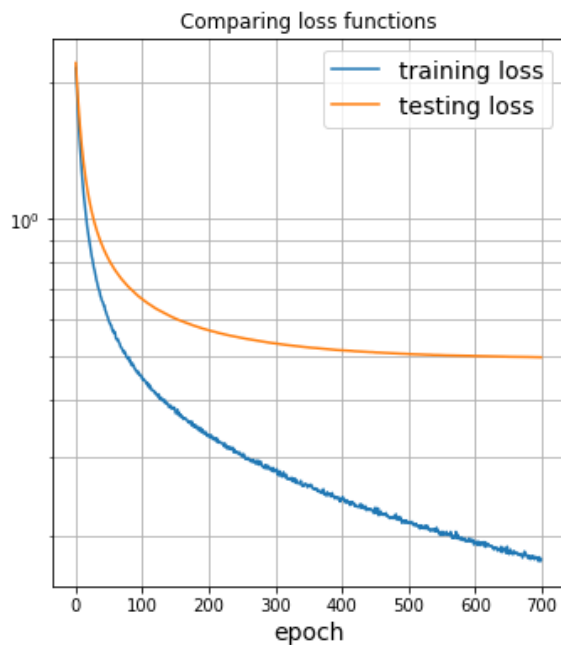
    # save them for later
    loss_seq_train.append(train_loss)
    loss_seq_test.append(test_loss)
    acc_seq_train.append(train_accuracy)
    acc_seq_test.append(test_accuracy)

    if e % 20 == 0:
        print("Completed epoch %s. Train Loss: %s, Test Loss %s, Train_acc %s, Test_acc
%s" %
              (e+1, train_loss, test_loss, train_accuracy, test_accuracy))

## Plotting the learning curves
plot_learningcurves(loss_seq_train,loss_seq_test,acc_seq_train,acc_seq_test)
```

```
Completed epoch 1. Train Loss: 2.14882323146, Test Loss 2.20137365162, Train_acc 0.251,
Test_acc 0.206
Completed epoch 21. Train Loss: 0.90038299188, Test Loss 1.10326142609, Train_acc 0.828,
Test_acc 0.728
Completed epoch 41. Train Loss: 0.656786840409, Test Loss 0.870345477015, Train_acc 0.861,
Test_acc 0.762
Completed epoch 61. Train Loss: 0.548336107284, Test Loss 0.767046701163, Train_acc 0.883,
Test_acc 0.782
Completed epoch 81. Train Loss: 0.496384473518, Test Loss 0.7064377442, Train_acc 0.892,
Test_acc 0.794
Completed epoch 101. Train Loss: 0.44359844178, Test Loss 0.664950948209, Train_acc 0.899,
Test_acc 0.804
Completed epoch 121. Train Loss: 0.414084114134, Test Loss 0.635051801801, Train_acc
```

0.908, Test\_acc 0.813  
Completed epoch 141. Train Loss: 0.385088480078, Test Loss 0.612390810624, Train\_acc 0.915, Test\_acc 0.822  
Completed epoch 161. Train Loss: 0.367607819848, Test Loss 0.594696460292, Train\_acc 0.918, Test\_acc 0.828  
Completed epoch 181. Train Loss: 0.351749705151, Test Loss 0.580225821584, Train\_acc 0.922, Test\_acc 0.829  
Completed epoch 201. Train Loss: 0.331399998628, Test Loss 0.568705741316, Train\_acc 0.928, Test\_acc 0.832  
Completed epoch 221. Train Loss: 0.321220521815, Test Loss 0.558648433536, Train\_acc 0.929, Test\_acc 0.831  
Completed epoch 241. Train Loss: 0.305758283474, Test Loss 0.550140928477, Train\_acc 0.93, Test\_acc 0.832  
Completed epoch 261. Train Loss: 0.297194710933, Test Loss 0.543482977897, Train\_acc 0.935, Test\_acc 0.832  
Completed epoch 281. Train Loss: 0.285452499054, Test Loss 0.53716141358, Train\_acc 0.937, Test\_acc 0.836  
Completed epoch 301. Train Loss: 0.278049795888, Test Loss 0.531760372221, Train\_acc 0.941, Test\_acc 0.835  
Completed epoch 321. Train Loss: 0.26735914126, Test Loss 0.527073308825, Train\_acc 0.942, Test\_acc 0.836  
Completed epoch 341. Train Loss: 0.258510973305, Test Loss 0.522905476391, Train\_acc 0.944, Test\_acc 0.836  
Completed epoch 361. Train Loss: 0.253523058258, Test Loss 0.519650578499, Train\_acc 0.944, Test\_acc 0.838  
Completed epoch 381. Train Loss: 0.248453027569, Test Loss 0.516450336203, Train\_acc 0.953, Test\_acc 0.841  
Completed epoch 401. Train Loss: 0.243487037718, Test Loss 0.513755075634, Train\_acc 0.955, Test\_acc 0.84  
Completed epoch 421. Train Loss: 0.235058872961, Test Loss 0.511168016121, Train\_acc 0.956, Test\_acc 0.841  
Completed epoch 441. Train Loss: 0.22945627477, Test Loss 0.509243799374, Train\_acc 0.957, Test\_acc 0.842  
Completed epoch 461. Train Loss: 0.223737638444, Test Loss 0.507475787774, Train\_acc 0.957, Test\_acc 0.842  
Completed epoch 481. Train Loss: 0.220507668331, Test Loss 0.505692290142, Train\_acc 0.96, Test\_acc 0.842  
Completed epoch 501. Train Loss: 0.215138303582, Test Loss 0.504200616851, Train\_acc 0.96, Test\_acc 0.841  
Completed epoch 521. Train Loss: 0.211557823233, Test Loss 0.502833517268, Train\_acc 0.961, Test\_acc 0.842  
Completed epoch 541. Train Loss: 0.205731691793, Test Loss 0.501563321799, Train\_acc 0.962, Test\_acc 0.843  
Completed epoch 561. Train Loss: 0.202632978559, Test Loss 0.500430552289, Train\_acc 0.962, Test\_acc 0.845  
Completed epoch 581. Train Loss: 0.197685314342, Test Loss 0.49991202727, Train\_acc 0.963, Test\_acc 0.845  
Completed epoch 601. Train Loss: 0.19489345653, Test Loss 0.498907541856, Train\_acc 0.965, Test\_acc 0.845  
Completed epoch 621. Train Loss: 0.192665177863, Test Loss 0.498069874942, Train\_acc 0.966, Test\_acc 0.845  
Completed epoch 641. Train Loss: 0.184659756254, Test Loss 0.497514432296, Train\_acc 0.966, Test\_acc 0.847  
Completed epoch 661. Train Loss: 0.184148575179, Test Loss 0.496921436861, Train\_acc 0.966, Test\_acc 0.848  
Completed epoch 681. Train Loss: 0.179120627698, Test Loss 0.496557332575, Train\_acc 0.967, Test\_acc 0.848



As we can see, the test accuracy improves a bit. Note that the amount by which it improves actually depends on the amount of weight decay. We recommend that you try and experiment with different extents of weight decay. For instance, a larger weight decay (e.g. 0.01) will lead to inferior performance, one that's larger still (0.1) will lead to terrible results. This is one of the reasons why tuning parameters is quite so important in getting good experimental results in practice.

## Next

[Learning environments](#)

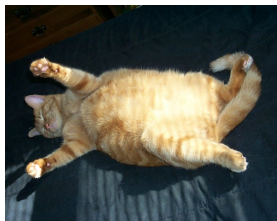

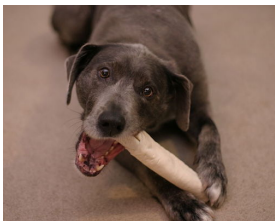

For whinges or inquiries, [open an issue on GitHub](#).

# Environment





So far we did not worry very much about where the data came from and how the models that we build get deployed. Not caring about it can be problematic. Many failed machine learning deployments can be traced back to this situation. This chapter is meant to help with detecting such situations early and points out how to mitigate them. Depending on the case this might be rather simple (ask for the ‘right’ data) or really difficult (implement a reinforcement learning system).

## Covariate Shift

At its heart is a problem that is easy to understand but also equally easy to miss. Consider being given the challenge of distinguishing cats and dogs. Our training data consists of images of the following kind:

			
cat	cat	dog	dog

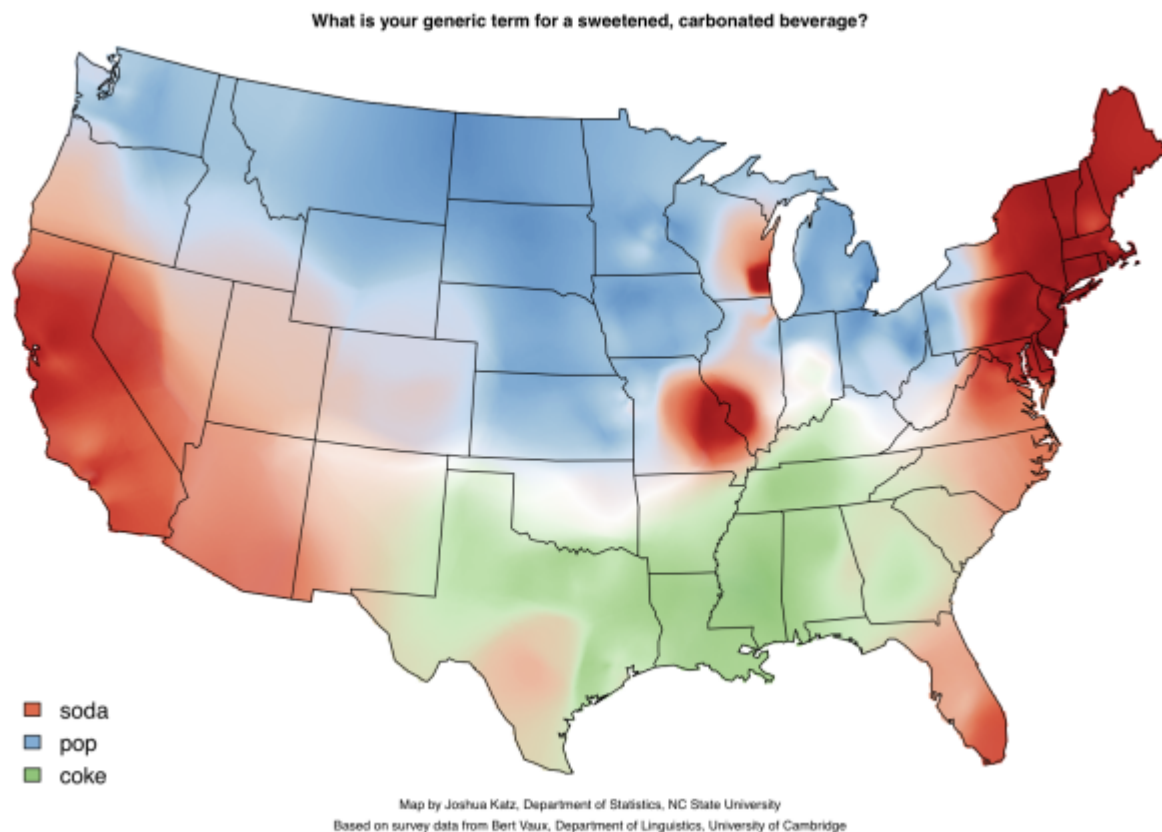
At test time we are asked to classify the following images:

			
cat	cat	dog	dog

Obviously this is unlikely to work well. The training set consists of photos, while the test set contains only cartoons. The colors aren't even accurate. Training on a dataset that looks substantially different from the test set without some plan for how to adapt to the new domain is a bad idea. Unfortunately, this is a very common pitfall. Statisticians call this **Covariate Shift**, i.e. the situation where the distribution over the covariates (aka training data) is shifted on test data relative to the training case. Mathematically speaking, we are referring the case where  $p(x)$  changes but  $p(y|x)$  remains unchanged.

## Concept Shift

A related problem is that of concept shift. This is the situation where the labels change. This sounds weird - after all, a cat is a cat is a cat. Well, cats maybe but not soft drinks. There is considerable concept shift throughout the USA, even for such a simple term:



If we were to build a machine translation system, the distribution  $p(y|x)$  would be different, e.g. depending on our location. This problem can be quite tricky to spot. A saving grace is that quite often the  $p(y|x)$  only shifts gradually (e.g. the click-through rate for NOKIA phone ads). Before we go into further details, let us discuss a number of situations where covariate and concept shift are not quite as blatantly obvious.

## Examples

## Medical Diagnostics

Imagine you want to design some algorithm to detect cancer. You get data of healthy and sick people; you train your algorithm; it works fine, giving you high accuracy and you conclude that you're ready for a successful career in medical diagnostics. Not so fast ...

Many things could go wrong. In particular, the distributions that you work with for training and those in the wild might differ considerably. This happened to an unfortunate startup I had the opportunity to consult for many years ago. They were developing a blood test for a disease that affects mainly older men and they'd managed to obtain a fair amount of blood samples from patients. It is considerably more difficult, though, to obtain blood samples from healthy men (mainly for ethical reasons). To compensate for that, they asked a large number of students on campus to donate blood and they performed their test. Then they asked me whether I could help them build a classifier to detect the disease. I told them that it would be very easy to distinguish between both datasets with probably near perfect accuracy. After all, the test subjects differed in age, hormone level, physical activity, diet, alcohol consumption, and many more factors unrelated to the disease. This was unlikely to be the case with real patients: Their sampling procedure had caused an extreme case of covariate shift that couldn't be corrected by conventional means. In other words, training and test data were so different that nothing useful could be done and they had wasted significant amounts of money.

## Self Driving Cars

A company wanted to build a machine learning system for self-driving cars. One of the key components is a roadside detector. Since real annotated data is expensive to get, they had the (smart and questionable) idea to use synthetic data from a game rendering engine as additional training data. This worked really well on 'test data' drawn from the rendering engine. Alas, inside a real car it was a disaster. As it turned out, the roadside had been rendered with a very simplistic texture. More importantly, *all* the roadside had been rendered with the *same* texture and the roadside detector learned about this 'feature' very quickly.

A similar thing happened to the US Army when they first tried to detect tanks in the forest. They took aerial photographs of the forest without tanks, then drove the tanks into the forest and took another set of pictures. The so-trained classifier worked 'perfectly'. Unfortunately, all it had learned was to distinguish trees with shadows from trees without shadows - the first set of pictures was taken in the early morning, the second one at noon.

## Nonstationary distributions

A much more subtle situation is where the distribution changes slowly and the model is not updated adequately. Here are a number of typical cases:

- We train a computational advertising model and then fail to update it frequently (e.g. we forget to incorporate that an obscure new device called an iPad was just launched).
- We build a spam filter. It works well at detecting all spam that we've seen so far. But then the spammers wisen up and craft new messages that look quite unlike anything we've seen before.
- We build a product recommendation system. It works well for the winter. But then it keeps on recommending Santa hats after Christmas.

## More Anecdotes

- We build a classifier for "Not suitable/safe for work" (NSFW) images. To make our life easy, we scrape a few seedy Subreddits. Unfortunately the accuracy on real life data is lacking (the pictures posted on Reddit are mostly 'remarkable' in some way, e.g. being taken by skilled photographers, whereas most real NSFW images are fairly unremarkable ...). Quite unsurprisingly the accuracy is not very high on real data.
- We build a face detector. It works well on all benchmarks. Unfortunately it fails on test data - the offending examples are close-ups where the face fills the entire image (no such data was in the training set).
- We build a web search engine for the USA market and want to deploy it in the UK.

In short, there are many cases where training and test distribution  $\mathcal{p}(x)$  are different. In some cases, we get lucky and the models work despite the covariate shift. We now discuss principled solution strategies. Warning - this will require some math and statistics.

## Covariate Shift Correction

Assume that we want to estimate some dependency  $\mathcal{p}(y|x)$  for which we have labeled data  $\mathcal{D} = \{(x_i, y_i)\}$ . Alas, the observations  $\mathcal{D}$  are drawn from some distribution  $\mathcal{q}(x)$  rather than the 'proper' distribution  $\mathcal{p}(x)$ . To make progress, we need to reflect about what exactly is happening during training: we iterate over training data and associated labels  $\{(x_1, y_1), \dots, (x_m, y_m)\}$  and update the weight vectors of the model after every minibatch. Depending on the situation we also apply some penalty to the parameters, e.g.  $L_2$  regularization. In other words, we want to solve

$$\underset{w}{\operatorname{minimize}} \frac{1}{m} \sum_{i=1}^m l(x_i, y_i, f(x_i)) + \frac{\lambda}{2} \|w\|_2^2$$

Statisticians call the first term an *empirical average*, that is an average computed over the data drawn from  $\mathcal{p}(x) \mathcal{p}(y|x)$ . If the data is drawn from the 'wrong' distribution  $\mathcal{q}$ , we can correct for that by using the following simple identity:

$$\mathbb{E}_{x \sim \mathcal{p}(x)} [f(x)] = \int f(x) \mathcal{p}(x) dx = \int f(x) \frac{\mathcal{p}(x)}{\mathcal{q}(x)} \mathcal{q}(x) dx = \mathbb{E}_{x \sim \mathcal{q}(x)} \left[ f(x) \frac{\mathcal{p}(x)}{\mathcal{q}(x)} \right]$$

In other words, we need to re-weight each instance by the ratio of probabilities that it would have been drawn from the correct distribution  $\beta(x) := p(x)/q(x)$ . Alas, we do not know that ratio, so before we can do anything useful we need to estimate it. Many methods are available, e.g. some rather fancy operator theoretic ones which try to recalibrate the expectation operator directly using a minimum-norm or a maximum entropy principle. Note that for any such approach, we need samples drawn from both distributions - the 'true'  $p$ , e.g. by access to training data, and the one used for generating the training set  $q$  (the latter is trivially available).

In this case there exists a very effective approach that will give almost as good results: logistic regression. This is all that is needed to compute estimate probability ratios. We learn a classifier to distinguish between data drawn from  $p(x)$  and data drawn from  $q(x)$ . If it is impossible to distinguish between the two distributions then it means that the associated instances are equally likely to come from either one of the two distributions. On the other hand, any instances that can be well discriminated should be significantly over/underweighted accordingly. For simplicity's sake assume that we have an equal number of instances from both distributions, denoted by  $x_i \sim p(x)$  and  $x'_i \sim q(x)$  respectively. Now denote by  $z_i$  labels which are 1 for data drawn from  $p$  and -1 for data drawn from  $q$ . Then the probability in a mixed dataset is given by

$p(z=1|x) = \frac{p(x)}{p(x)+q(x)}$  and hence  $\frac{p(z=1|x)}{p(z=-1|x)} = \frac{p(x)}{q(x)}$ . Hence, if we use a logistic regression approach where  $p(z=1|x) = \frac{1}{1+\exp(-f(x))}$  it follows (after some simple algebra) that  $\beta(x) = \exp(f(x))$ . In summary, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a reweighted minimization problem where we weigh terms by  $\beta$ , e.g. via the head gradients. Here's a prototypical algorithm for that purpose:

```
CovariateShiftCorrector(X, Z)
  X: Training dataset (without labels)
  Z: Test dataset (without labels)

  generate training set with {(x_i, -1) ... (z_j, 1)}
  train binary classifier using logistic regression to get function f
  weigh data using beta_i = exp(f(x_i)) or
                        beta_i = min(exp(f(x_i)), c)
  use weights beta_i for training on X with labels Y
```

**Generative Adversarial Networks** use the very idea described above to engineer a *data generator* such that it cannot be distinguished from a reference dataset. For this, we use one network, say  $f$  to distinguish real and fake data and a second network  $g$  that tries to fool the discriminator  $f$  into accepting fake data as real. We will discuss this in much more detail later.

## Concept Shift Correction



Concept shift is much harder to fix in a principled manner. For instance, in a situation where suddenly the problem changes from distinguishing cats from dogs to one of distinguishing white from black animals, it will be unreasonable to assume that we can do much better than just training from scratch using the new labels. Fortunately, in practice, such extreme shifts almost never happen. Instead, what usually happens is that the task keeps on changing slowly. To make things more concrete, here are some examples:

- In computational advertising, new products are launched, old products become less popular. This means that the distribution over ads and their popularity changes gradually and any click-through rate predictor needs to change gradually with it.
- Traffic cameras lenses degrade gradually due to environmental wear, affecting image quality progressively.
- News content changes gradually (i.e. most of the news remains unchanged but new stories appear).

In such cases, we can use the same approach that we used for training networks to make them adapt to the change in the data. In other words, we use the existing network weights and simply perform a few update steps with the new data rather than training from scratch.

## A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in  $p(x)$  and in  $p(y|x)$ , let us consider a number of problems that we can solve using machine learning.

- **Batch Learning.** Here we have access to training data and labels  $\{(x_1, y_1), \dots, (x_n, y_n)\}$ , which we use to train a network  $f(x, w)$ . Later on, we deploy this network to score new data  $\{(x, y)\}$  drawn from the same distribution. This is the default assumption for any of the problems that we discuss here. For instance, we might train a cat detector based on lots of pictures of cats and dogs. Once we trained it, we ship it as part of a smart catdoor computer vision system that lets only cats in. This is then installed in a customer's home and is never updated again (barring extreme circumstances).
- **Online Learning.** Now imagine that the data  $(x_i, y_i)$  arrives one sample at a time. More specifically, assume that we first observe  $x_i$ , then we need to come up with an estimate  $f(x_i, w)$  and only once we've done this, we observe  $y_i$  and with it, we receive a reward (or incur a loss), given our decision. Many real problems fall into this category. E.g. we need to predict tomorrow's stock price, this allows us to trade based on that estimate and at the end of the day we find out whether our estimate allowed us to make a profit. In other words, we have the following cycle where we are continuously improving our model given new observations.

$$\begin{aligned} [\text{model}] \sim f_t &\rightarrow \text{data} \sim x_t \rightarrow \text{estimate} \\ &\sim f_t(x_t) \rightarrow \text{observation} \sim y_t \rightarrow \text{loss} \sim l(y_t, \end{aligned}$$

$$f_t(x_t) \rightarrow \mathrm{model} \sim f_{t+1}$$

- **Bandits.** They are a *special case* of the problem above. While in most learning problems we have a continuously parametrized function  $f$  where we want to learn its parameters (e.g. a deep network), in a bandit problem we only have a finite number of arms that we can pull (i.e. a finite number of actions that we can take). It is not very surprising that for this simpler problem stronger theoretical guarantees in terms of optimality can be obtained. We list it mainly since this problem is often (confusingly) treated as if it were a distinct learning setting.
- **Control (and nonadversarial Reinforcement Learning).** In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it'll just remember and the response will depend on what happened before. E.g. a coffee boiler controller will observe different temperatures depending on whether it was heating the boiler previously. PID (proportional integral derivative) controller algorithms are a [popular choice](#) there. Likewise, a user's behavior on a news site will depend on what we showed him previously (e.g. he will read most news only once). Many such algorithms form a model of the environment in which they act such as to make their decisions appear less random (i.e. to reduce variance).
- **Reinforcement Learning.** In the more general case of an environment with memory, we may encounter situations where the environment is trying to *cooperate* with us (cooperative games, in particular for non-zero-sum games), or others where the environment will try to *win*. Chess, Go, Backgammon or StarCraft are some of the cases. Likewise, we might want to build a good controller for autonomous cars. The other cars are likely to respond to the autonomous car's driving style in nontrivial ways, e.g. trying to avoid it, trying to cause an accident, trying to cooperate with it, etc.

One key distinction between the different situations above is that the same strategy that might have worked throughout in the case of a stationary environment, might not work throughout when the environment can adapt. For instance, an arbitrage opportunity discovered by a trader is likely to disappear once he starts exploiting it. The speed and manner at which the environment changes determines to a large extent the type of algorithms that we can bring to bear. For instance, if we *know* that things may only change slowly, we can force any estimate to change only slowly, too. If we know that the environment might change instantaneously, but only very infrequently, we can make allowances for that. These types of knowledge are crucial for the aspiring data scientist to deal with concept shift, i.e. when the problem that he is trying to solve changes over time.

For whinges or inquiries, [open an issue on GitHub](#).

## Multilayer perceptrons from scratch

Now that we've covered all the preliminaries, extending to deep neural networks is actually quite easy.

```
In [1]: from __future__ import print_function
import mxnet as mx
import numpy as np
from mxnet import nd, autograd
ctx = mx.cpu()
```

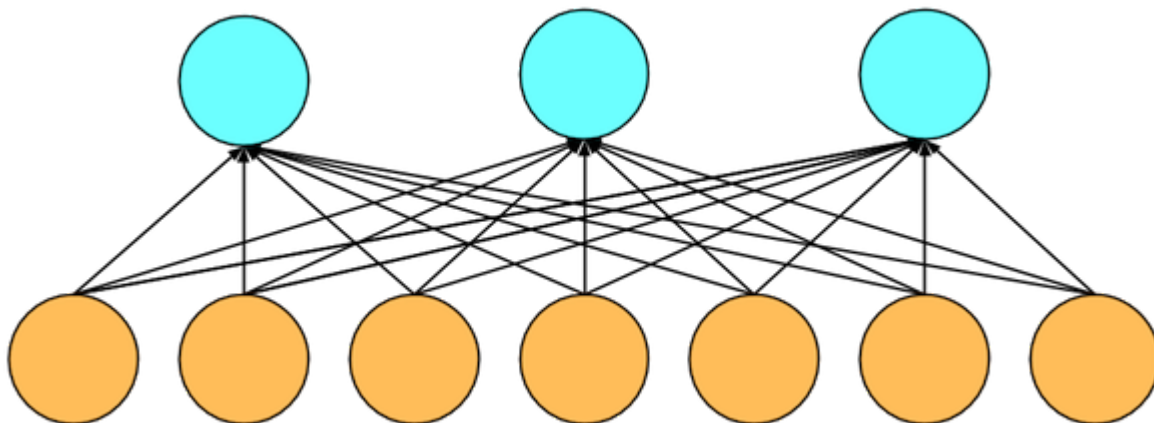
## MNIST data (surprise!)

Let's go ahead and grab our data.

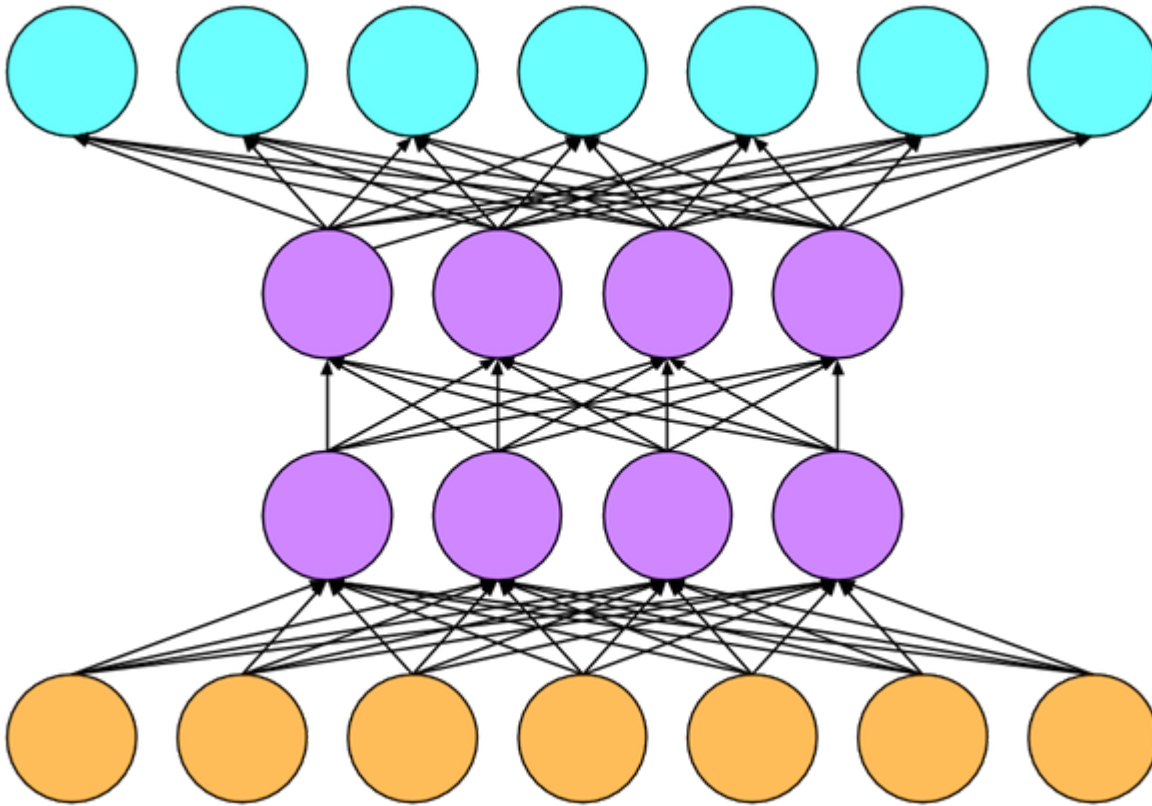
```
In [2]: num_inputs = 784
num_outputs = 10
batch_size = 64
def transform(data, label):
    return data.astype(np.float32)/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
transform=transform),
                                     batch_size, shuffle=False)
```

## Multilayer perceptrons

Here's where things start to get interesting. Before, we mapped our inputs directly onto our outputs through a single linear transformation.



This model is perfectly adequate when the underlying relationship between our data points and labels is approximately linear. When our data points and targets are characterized by a more complex relationship, a linear model will produce results with insufficient accuracy. We can model a more general class of functions by incorporating one or more *hidden layers*.



Here, each layer will require it's own set of parameters. To make things simple here, we'll assume two hidden layers of computation.

```
In [3]: #####
# Set some constants so it's easy to modify the network later
#####
num_hidden = 256
weight_scale = .01

#####
# Allocate parameters for the first hidden layer
#####
W1 = nd.random_normal(shape=(num_inputs, num_hidden), scale=weight_scale, ctx=ctx)
b1 = nd.random_normal(shape=num_hidden, scale=weight_scale, ctx=ctx)

#####
# Allocate parameters for the second hidden layer
#####
W2 = nd.random_normal(shape=(num_hidden, num_hidden), scale=weight_scale, ctx=ctx)
b2 = nd.random_normal(shape=num_hidden, scale=weight_scale, ctx=ctx)

#####
# Allocate parameters for the output layer
#####
W3 = nd.random_normal(shape=(num_hidden, num_outputs), scale=weight_scale, ctx=ctx)
b3 = nd.random_normal(shape=num_outputs, scale=weight_scale, ctx=ctx)

params = [W1, b1, W2, b2, W3, b3]
```

Again, let's allocate space for each parameter's gradients.

```
In [4]: for param in params:
        param.attach_grad()
```

## Activation functions

If we compose a multi-layer network but use only linear operations, then our entire network will still be a linear function. That's because  $\hat{y} = X \cdot W_1 \cdot W_2 = X \cdot W_4$  for  $W_4 = W_1 \cdot W_2 \cdot W_3$ . To give our model the capacity to capture nonlinear functions, we'll need to interleave our linear operations with activation functions. In this case, we'll use the rectified linear unit (ReLU):

```
In [5]: def relu(X):
        return nd.maximum(X, nd.zeros_like(X))
```

## Softmax output

As with multiclass logistic regression, we'll want the outputs to constitute a valid probability distribution. We'll use the same softmax activation function on our output to make sure that our outputs sum to one and are non-negative.

```
In [6]: def softmax(y_linear):
        exp = nd.exp(y_linear - nd.max(y_linear))
        partition = nd.nansum(exp, axis=0, exclude=True).reshape((-1, 1))
        return exp / partition
```

## The *softmax* cross-entropy loss function

In the previous example, we calculated our model's output and then ran this output through the cross-entropy loss function:

```
In [7]: def cross_entropy(yhat, y):
        return - nd.nansum(y * nd.log(yhat), axis=0, exclude=True)
```

Mathematically, that's a perfectly reasonable thing to do. However, computationally, things can get hairy. We'll revisit the issue at length in a chapter more dedicated to implementation and less interested in statistical modeling. But we're going to make a change here so we want to give you the gist of why.

When we calculate the softmax partition function, we take a sum of exponential functions:  $\sum_{i=1}^n e^{z_i}$ . When we also calculate our numerators as exponential functions, then this can give rise to some big numbers in our intermediate calculations. The pairing of big numbers and low precision mathematics tends to make things go crazy. As a result, if we use our naive softmax implementation, we might get horrific not a number (`nan`) results printed to screen.

Our salvation is that even though we're computing these exponential functions, we ultimately plan to take their log in the cross-entropy functions. It turns out that by combining these two operators `softmax` and `cross_entropy` together, we can elude the numerical stability issues that might otherwise plague us during backpropagation. We'll want to keep the conventional softmax function handy in case we ever want to evaluate the probabilities output by our model.

But instead of passing softmax probabilities into our loss function - we'll just pass our `yhat_linear` and compute the softmax and its log all at once inside the `softmax_cross_entropy` loss function simultaneously, which does smart things like the log-sum-exp trick ([see on Wikipedia](#)).

```
In [8]: def softmax_cross_entropy(yhat_linear, y):
        return - nd.nansum(y * nd.log_softmax(yhat_linear), axis=0, exclude=True)
```

## Define the model

Now we're ready to define our model

```
In [9]: def net(X):
        #####
        # Compute the first hidden layer
        #####
        h1_linear = nd.dot(X, W1) + b1
        h1 = relu(h1_linear)

        #####
        # Compute the second hidden layer
        #####
        h2_linear = nd.dot(h1, W2) + b2
        h2 = relu(h2_linear)

        #####
        # Compute the output layer.
        # We will omit the softmax function here
        # because it will be applied
        # in the softmax_cross_entropy loss
        #####
        yhat_linear = nd.dot(h2, W3) + b3
        return yhat_linear
```

## Optimizer

```
In [10]: def SGD(params, lr):
        for param in params:
```

```
param[:] = param - lr * param.grad
```

## Evaluation metric

```
In [11]: def evaluate_accuracy(data_iterator, net):
    numerator = 0.
    denominator = 0.
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1, 784))
        label = label.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        numerator += nd.sum(predictions == label)
        denominator += data.shape[0]
    return (numerator / denominator).asscalar()
```

## Execute the training loop

```
In [12]: epochs = 10
    learning_rate = .001
    smoothing_constant = .01

    for e in range(epochs):
        for i, (data, label) in enumerate(train_data):
            data = data.as_in_context(ctx).reshape((-1, 784))
            label = label.as_in_context(ctx)
            label_one_hot = nd.one_hot(label, 10)
            with autograd.record():
                output = net(data)
                loss = softmax_cross_entropy(output, label_one_hot)
            loss.backward()
            SGD(params, learning_rate)

            #####
            # Keep a moving average of the losses
            #####
            curr_loss = nd.mean(loss).asscalar()
            moving_loss = (curr_loss if ((i == 0) and (e == 0))
                           else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
                           * curr_loss)

            test_accuracy = evaluate_accuracy(test_data, net)
            train_accuracy = evaluate_accuracy(train_data, net)
            print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" %
                  (e, moving_loss, train_accuracy, test_accuracy))
```

```
Epoch 0. Loss: 0.455451145229, Train_acc 0.885817, Test_acc 0.8897
Epoch 1. Loss: 0.297250172348, Train_acc 0.921383, Test_acc 0.9205
Epoch 2. Loss: 0.202016335186, Train_acc 0.946467, Test_acc 0.9451
Epoch 3. Loss: 0.151867129294, Train_acc 0.960667, Test_acc 0.9584
Epoch 4. Loss: 0.113816030109, Train_acc 0.9688, Test_acc 0.9637
Epoch 5. Loss: 0.100374131216, Train_acc 0.97185, Test_acc 0.9658
Epoch 6. Loss: 0.0873043180085, Train_acc 0.9779, Test_acc 0.9713
Epoch 7. Loss: 0.0730908748383, Train_acc 0.98085, Test_acc 0.972
Epoch 8. Loss: 0.068088298137, Train_acc 0.984883, Test_acc 0.9735
Epoch 9. Loss: 0.0573755351742, Train_acc 0.986133, Test_acc 0.9731
```

## Conclusion

Nice! With just two hidden layers containing 256 hidden nodes, respectively, we can achieve over 95% accuracy on this task.

## Next

[Multilayer perceptrons with gluon](#)

For whinges or inquiries, [open an issue on GitHub](#).



## Multilayer perceptrons in `gluon`

Using `gluon`, we only need two additional lines of code to transform our logistic regression model into a multilayer perceptron.

```
In [1]: from __future__ import print_function
import mxnet as mx
import numpy as np
from mxnet import nd, autograd
from mxnet import gluon
```

We'll also want to set the compute context for our modeling. Feel free to go ahead and change this to `mx.gpu(0)` if you're running on an appropriately endowed machine.

```
In [2]: ctx = mx.cpu()
```

## The MNIST dataset

```
In [3]: batch_size = 64
num_inputs = 784
num_outputs = 10
def transform(data, label):
    return data.astype(np.float32)/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
                                                                    batch_size=batch_size, shuffle=True),
                                      transform=transform),
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
                                                                    batch_size=batch_size, shuffle=False),
                                      transform=transform),
```

## Define the model

*Here's the only real difference. We add two lines!*

```
In [4]: num_hidden = 256
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_outputs))
```

## Parameter initialization

---

```
In [5]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

## Softmax cross-entropy loss

```
In [6]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Optimizer

```
In [7]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .1})
```

## Evaluation metric

```
In [8]: def evaluate_accuracy(data_iterator, net):
    acc = mx.metric.Accuracy()
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1, 784))
        label = label.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

## Training loop

```
In [9]: epochs = 10
    smoothing_constant = .01

    for e in range(epochs):
        for i, (data, label) in enumerate(train_data):
            data = data.as_in_context(ctx).reshape((-1, 784))
            label = label.as_in_context(ctx)
            with autograd.record():
                output = net(data)
                loss = softmax_cross_entropy(output, label)
                loss.backward()
            trainer.step(data.shape[0])

            #####
            # Keep a moving average of the losses
            #####
            curr_loss = nd.mean(loss).asscalar()
            moving_loss = (curr_loss if ((i == 0) and (e == 0))
                           else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
                           * curr_loss)

            test_accuracy = evaluate_accuracy(test_data, net)
            train_accuracy = evaluate_accuracy(train_data, net)
            print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" %
                  (e, moving_loss, train_accuracy, test_accuracy))
```

```
Epoch 0. Loss: 0.208460539446, Train_acc 0.948683333333, Test_acc 0.9482
Epoch 1. Loss: 0.137320037022, Train_acc 0.958966666667, Test_acc 0.9551
Epoch 2. Loss: 0.0958231976158, Train_acc 0.956716666667, Test_acc 0.9492
Epoch 3. Loss: 0.0725868264617, Train_acc 0.98395, Test_acc 0.9754
Epoch 4. Loss: 0.0646171670057, Train_acc 0.9836, Test_acc 0.9735
Epoch 5. Loss: 0.0469602448996, Train_acc 0.987683333333, Test_acc 0.9766
Epoch 6. Loss: 0.0403166358583, Train_acc 0.99195, Test_acc 0.9783
```

Epoch 7. Loss: 0.034311452392, Train\_acc 0.991866666667, Test\_acc 0.977  
Epoch 8. Loss: 0.0319601120719, Train\_acc 0.994733333333, Test\_acc 0.9783  
Epoch 9. Loss: 0.0243036117522, Train\_acc 0.991466666667, Test\_acc 0.977

## Conclusion

We showed the much simpler way to define a multilayer perceptrons in `gluon`. Now let's take a look at how to build convolutional neural networks.

## Next

[Dropout regularization from scratch](#)

For whinges or inquiries, [open an issue on GitHub](#).

If you're reading the tutorials in sequence, then you might remember from Part 2 that machine learning models can be susceptible to overfitting. To recap: in machine learning, our goal is to discover general patterns. For example, we might want to learn an association between genetic markers and the development of dementia in adulthood. Our hope would be to uncover a pattern that could be applied successfully to assess risk for the entire population.

However, when we train models, we don't have access to the entire population (or current or potential humans). Instead, we can access only a small, finite sample. Even in a large hospital system, we might get hundreds of thousands of medical records. Given such a finite sample size, it's possible to uncover spurious associations that don't hold up for unseen data.

Let's consider an extreme pathological case. Imagine that you want to learn to predict which people will repay their loans. A lender hires you as a data scientist to investigate the case and gives you complete files on 100 applicants, of which 5 defaulted on their loans within 3 years. The files might include hundreds of features including income, occupation, credit score, length of employment etcetera. Imagine that they additionally give you video footage of their interview with a lending agent. That might seem like a lot of data!

Now suppose that after generating an enormous set of features, you discover that of the 5 applicants who defaults, all 5 were wearing blue shirts during their interviews, while only 40% of general population wore blue shirts. There's a good chance that any model you train would pick up on this signal and use it as an important part of its learned pattern.

Even if defaulters are no more likely to wear blue shirts, there's a 1% chance that we'll observe all five defaulters wearing blue shirts. And keeping the sample size low while we have hundreds or thousands of features, we may observe a large number of spurious correlations. Given trillions of training examples, these false associations might disappear. But we seldom have that luxury.

The phenomena of fitting our training distribution more closely than the real distribution is called *overfitting*, and the techniques used to combat overfitting are called regularization. In the previous chapter, we introduced one classical approach to regularize statistical models. We penalized the size (the  $\ell^2$  norm) of the weights, coercing them to take smaller values. In probabilistic terms we might say this imposes a Gaussian prior on the value of the weights. But

in more intuitive, functional terms, we can say this encourages the model to speed out its weights among many features and not to depend too much on a small number of potentially spurious associations.

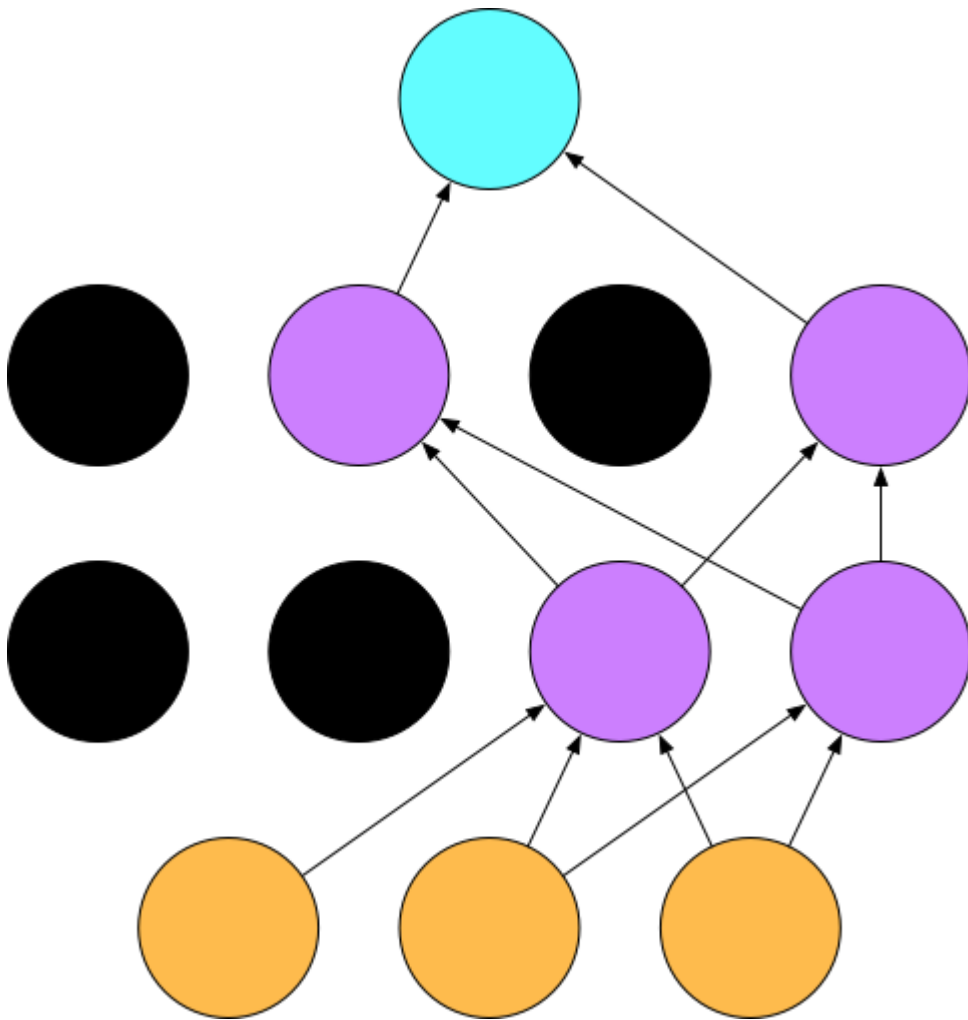
Given many more features than examples, linear models can overfit. But when there are many more examples than features, linear models can usually be counted on not to overfit. Unfortunately this propensity to generalize well comes at a cost. For every feature, a linear model has to assign it either positive or negative weight. Linear models can't take into account nuanced interactions between features. In more formal texts, you'll see this phenomena discussed as the bias-variance tradeoff. Linear models have high bias, (they can only represent a small class of functions), but low variance (they give similar results across different random samples of the data). **[point to more formal discussion of generalization when chapter exists]**

Deep neural networks, however, occupy the opposite end of the bias-variance spectrum. Neural networks are so flexible because they aren't confined to looking at each feature individually. Instead, they can learn complex interactions among groups of features. For example, they might infer that "Nigeria" and "Western Union" appearing together in an email indicates spam but that "Nigeria" without "Western Union" does not connote spam.

Even for a small number of features, deep neural networks are capable of overfitting. As one demonstration of the incredible flexibility of neural networks, researchers showed that [neural networks perfectly classify randomly labeled data](#). Let's think about what means. If the labels are assigned uniformly at random, and there are 10 classes, then no classifier can get better than 10% accuracy on holdout data. Yet even in these situations, when there is no true pattern to be learned, neural networks can perfectly fit the training labels.

In 2012, Professor Geoffrey Hinton and his students including Nitish Srivastava introduced a new idea for how to regularize neural network models. The intuition goes something like this. When a neural network overfits badly to training data, each layer depends too heavily on the exact configuration of features in the previous layer.

To prevent the neural network from depending too much on any exact activation pathway, Hinton and Srivastava proposed randomly *dropping out* (i.e. setting to 0) the hidden nodes in every layer with probability .5. Given a network with  $n$  nodes we are sampling uniformly at random from the  $2^n$  networks in which a subset of the nodes are turned off.



One intuition here is that because the nodes to drop out are chosen randomly on every pass, the representations in each layer can't depend on the exact values taken by nodes in the previous layer.

However, when it comes time to make predictions, we want to use the full representational power of our model. In other words, we don't want to drop out activations at test time. One principled way to justify the use of all nodes simultaneously, despite not training in this fashion, is that it's a form of model averaging. At each layer we average the representations of all of the  $2^n$  dropout networks. Because each node has a .5 probability of being on during training, its vote is scaled by .5 when we use all nodes at prediction time

```
In [1]: from __future__ import print_function
import mxnet as mx
import numpy as np
from mxnet import nd, autograd
import numpy as np
mx.random.seed(1)
ctx = mx.cpu()
```

Let's go ahead and grab our data.

[SWITCH TO CIFAR TO GET BETTER FEEL FOR REGULARIZATION]

```
In [2]: mnist = mx.test_utils.get_mnist()
batch_size = 64
def transform(data, label):
    return data.astype(np.float32)/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
transform=transform),
                                     batch_size, shuffle=False)
```

```
In [3]: W1 = nd.random_normal(shape=(784,256), ctx=ctx) *.01
b1 = nd.random_normal(shape=256, ctx=ctx) * .01

W2 = nd.random_normal(shape=(256,128), ctx=ctx) *.01
b2 = nd.random_normal(shape=128, ctx=ctx) * .01

W3 = nd.random_normal(shape=(128,10), ctx=ctx) *.01
b3 = nd.random_normal(shape=10, ctx=ctx) *.01

params = [W1, b1, W2, b2, W3, b3]
```

Again, let's allocate space for gradients.

```
In [4]: for param in params:
        param.attach_grad()
```

If we compose a multi-layer network but use only linear operations, then our entire network will still be a linear function. That's because  $\hat{y} = X \cdot W_1 \cdot W_2 = X \cdot W_4$  for  $W_4 = W_1 \cdot W_2 \cdot W_3$ . To give our model the capacity to capture nonlinear functions, we'll need to interleave our linear operations with activation functions. In this case, we'll use the rectified linear unit (ReLU):

```
In [5]: def relu(X):
        return nd.maximum(X, 0)
```

```
In [6]: def dropout(X, drop_probability):
        keep_probability = 1 - drop_probability
        mask = nd.random_uniform(0, 1.0, X.shape, ctx=X.context) < keep_probability
        #####
        # Avoid division by 0 when scaling
        #####
        if keep_probability > 0.0:
```

```

        scale = (1/keep_probability)
    else:
        scale = 0.0
    return mask * X * scale

```

```

In [7]: A = nd.arange(20).reshape((5,4))
        dropout(A, 0.0)

```

```

Out[7]:
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]]
<NDArray 5x4 @cpu(0)>

```

```

In [8]: dropout(A, 0.5)

```

```

Out[8]:
[[ 0.  0.  4.  6.]
 [ 8. 10. 12. 14.]
 [ 0. 18. 20. 22.]
 [24.  0.  0. 30.]
 [32.  0.  0. 38.]]
<NDArray 5x4 @cpu(0)>

```

```

In [9]: dropout(A, 1.0)

```

```

Out[9]:
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
<NDArray 5x4 @cpu(0)>

```

```

In [10]: def softmax(y_linear):
          exp = nd.exp(y_linear-nd.max(y_linear))
          partition = nd.nansum(exp, axis=0, exclude=True).reshape((-1,1))
          return exp / partition

```

```

In [11]: def softmax_cross_entropy(yhat_linear, y):
          return - nd.nansum(y * nd.log_softmax(yhat_linear), axis=0, exclude=True)

```

Now we're ready to define our model

```

In [12]: def net(X, drop_prob=0.0):
          #####
          # Compute the first hidden layer
          #####
          h1_linear = nd.dot(X, W1) + b1
          h1 = relu(h1_linear)

```



```

h1 = dropout(h1, drop_prob)

#####
# Compute the second hidden layer
#####
h2_linear = nd.dot(h1, W2) + b2
h2 = relu(h2_linear)
h2 = dropout(h2, drop_prob)

#####
# Compute the output layer.
# We will omit the softmax function here
# because it will be applied
# in the softmax_cross_entropy loss
#####
yhat_linear = nd.dot(h2, W3) + b3
return yhat_linear

```

```

In [13]: def SGD(params, lr):
          for param in params:
              param[:] = param - lr * param.grad

```

```

In [14]: def evaluate_accuracy(data_iterator, net):
          numerator = 0.
          denominator = 0.
          for i, (data, label) in enumerate(data_iterator):
              data = data.as_in_context(ctx).reshape((-1, 784))
              label = label.as_in_context(ctx)
              output = net(data)
              predictions = nd.argmax(output, axis=1)
              numerator += nd.sum(predictions == label)
              denominator += data.shape[0]
          return (numerator / denominator).asscalar()

```

```

In [15]: epochs = 10
          moving_loss = 0.
          learning_rate = .001

          for e in range(epochs):
              for i, (data, label) in enumerate(train_data):
                  data = data.as_in_context(ctx).reshape((-1, 784))
                  label = label.as_in_context(ctx)
                  label_one_hot = nd.one_hot(label, 10)
                  with autograd.record():
                      #####
                      # Drop out 50% of hidden activations on the forward pass
                      #####
                      output = net(data, drop_prob=.5)
                      loss = softmax_cross_entropy(output, label_one_hot)
                  loss.backward()
                  SGD(params, learning_rate)

                  #####
                  # Keep a moving average of the losses
                  #####
                  if i == 0:

```

```

        moving_loss = nd.mean(loss).asscalar()
    else:
        moving_loss = .99 * moving_loss + .01 * nd.mean(loss).asscalar()

    test_accuracy = evaluate_accuracy(test_data, net)
    train_accuracy = evaluate_accuracy(train_data, net)
    print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, moving_loss,
train_accuracy, test_accuracy))

```

```

Epoch 0. Loss: 0.737156236043, Train_acc 0.850967, Test_acc 0.8539
Epoch 1. Loss: 0.394209427167, Train_acc 0.92225, Test_acc 0.923
Epoch 2. Loss: 0.296510421107, Train_acc 0.946, Test_acc 0.9452
Epoch 3. Loss: 0.232048722573, Train_acc 0.9563, Test_acc 0.9531
Epoch 4. Loss: 0.205553142149, Train_acc 0.962967, Test_acc 0.9591
Epoch 5. Loss: 0.178349442085, Train_acc 0.969233, Test_acc 0.9641
Epoch 6. Loss: 0.175119599567, Train_acc 0.9735, Test_acc 0.967
Epoch 7. Loss: 0.157515936016, Train_acc 0.975067, Test_acc 0.9688
Epoch 8. Loss: 0.14164880119, Train_acc 0.977933, Test_acc 0.9705
Epoch 9. Loss: 0.129475182254, Train_acc 0.9798, Test_acc 0.9729

```

Nice. With just two hidden layers containing 256 and 128 hidden nodes, respectively, we can achieve over 95% accuracy on this task.

## Dropout regularization with gluon

For whinges or inquiries, [open an issue on GitHub](#).

## Dropout regularization with `gluon`

In [the previous chapter](#), we introduced Dropout regularization, implementing the algorithm from scratch. As a reminder, Dropout is a regularization technique that zeroes out some fraction of the nodes during training. Then at test time, we use all of the nodes, but scale down their values, essentially averaging the various dropped out nets. If you're approaching this chapter out of sequence, and aren't sure how Dropout works, it's best to take a look at the implementation by hand since `gluon` will manage the low-level details for us.

Dropout is a special kind of layer because it behaves differently when training and predicting. We've already seen how `gluon` can keep track of when to record vs not record the computation graph. Since this is a `gluon` implementation chapter, let's get into the thick of things by importing our dependencies and some toy data.

```
In [ ]: from __future__ import print_function
import mxnet as mx
import numpy as np
from mxnet import nd, autograd
from mxnet import gluon
ctx = mx.cpu()
```

## The MNIST dataset

```
In [ ]: batch_size = 64
num_inputs = 784
num_outputs = 10
def transform(data, label):
    return data.astype(np.float32)/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
                                                                transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
                                                                transform=transform),
                                     batch_size, shuffle=False)
```

## Define the model

Now we can add Dropout following each of our hidden layers.

```
In [ ]: num_hidden = 256
net = gluon.nn.Sequential()
with net.name_scope():
```

```
#####
# Adding first hidden layer
#####
net.add(gluon.nn.Dense(num_hidden, activation="relu"))
#####
# Adding dropout with rate .5 to the first hidden layer
#####
net.add(gluon.nn.Dropout(.5))

#####
# Adding first hidden layer
#####
net.add(gluon.nn.Dense(num_hidden, activation="relu"))
#####
# Adding dropout with rate .5 to the second hidden layer
#####
net.add(gluon.nn.Dropout(.5))

#####
# Adding the output layer
#####
net.add(gluon.nn.Dense(num_outputs))
```

## Parameter initialization

Now that we've got an MLP with dropout layers, let's register an initializer so we can play with some data.

```
In [ ]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

## Train mode and predict mode

Let's grab some data and pass it through the network. To see what effect dropout is having on our predictions, it's instructive to pass the same example through our net multiple times.

```
In [ ]: for x, _ in train_data:
        x = x.as_in_context(ctx)
        break
        print(net(x[0:1]))
        print(net(x[0:1]))
```

Note that we got the exact same answer on both forward passes through the net! That's because by default, `mxnet` assumes that we are in predict mode. We can explicitly invoke this scope by placing code within a `with autograd.predict_mode():` block.

```
In [ ]: with autograd.predict_mode():
        print(net(x[0:1]))
        print(net(x[0:1]))
```

Unless something's gone horribly wrong, you should see the same result as before. We can also run the code in *train mode*. This tells MXNet to run our Blocks as they would run during training.

```
In [ ]: with autograd.train_mode():
        print(net(x[0:1]))
        print(net(x[0:1]))
```

## Accessing `is_training()` status

You might wonder, how precisely do the Blocks determine whether they should run in train mode or predict mode? Basically, autograd maintains a Boolean state that can be accessed via `autograd.is_training()`. By default this value is `False` in the global scope. This way if someone just wants to make predictions and doesn't know anything about training models, everything will just work. When we enter a `train_mode()` block, we create a scope in which `is_training()` returns `True`.

```
In [ ]: with autograd.predict_mode():
        print(autograd.is_training())

        with autograd.train_mode():
            print(autograd.is_training())
```

## Integration with `autograd.record`

When we train neural network models, we nearly always enter `record()` blocks. The purpose of `record()` is to build the computational graph. And the purpose of `train` is to indicate that we are training our model. These two are highly correlated but should not be confused. For example, when we generate adversarial examples (a topic we'll investigate later) we may want to record, but for the model to behave as in predict mode. On the other hand, sometimes, even when we're not recording, we still want to evaluate the model's training behavior.

A problem then arises. Since `record()` and `train_mode()` are distinct, how do we avoid having to declare two scopes every time we train the model?

```
In [ ]: #####
        # Writing this every time could get cumbersome
        #####
        with autograd.record():
            with autograd.train_mode():
                yhat = net(x)
```

To make our lives a little easier, `record()` takes one argument, `train_mode`, which has a default value of `True`. So when we turn on autograd, this by default turns on `train_mode` (`with autograd.record()` is equivalent to `with autograd.record(train_mode=True):`). To change this default behavior (as when generating adversarial examples), we can optionally call `record` via (`with autograd.record(train_mode=False):`).

# Softmax cross-entropy loss

```
In [ ]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Optimizer

```
In [ ]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .1})
```

## Evaluation metric

```
In [ ]: def evaluate_accuracy(data_iterator, net):
    acc = mx.metric.Accuracy()
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1, 784))
        label = label.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

## Training loop

```
In [17]: epochs = 10
smoothing_constant = .01

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx).reshape((-1, 784))
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output, label)
            loss.backward()
        trainer.step(data.shape[0])

        #####
        # Keep a moving average of the losses
        #####
        curr_loss = nd.mean(loss).asscalar()
        moving_loss = (curr_loss if ((i == 0) and (e == 0))
                       else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
                       * curr_loss)

        test_accuracy = evaluate_accuracy(test_data, net)
        train_accuracy = evaluate_accuracy(train_data, net)
        print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" %
              (e, moving_loss, train_accuracy, test_accuracy))
```

Epoch 9. Loss: 0.121087726722, Train\_acc 0.986133333333, Test\_acc 0.9774

## Conclusion

Now let's take a look at how to build convolutional neural networks.

## Next

*Introduction to ``gluon.Block` and `gluon.nn.Sequential`* <../chapter03\_deep-neural-networks/plumbing.ipynb>`\_\_

For whinges or inquiries, [open an issue on GitHub](#).

## Plumbing: A look under the hood of `gluon`

In the previous tutorials, we taught you about linear regression and softmax regression. We explained how these models work in principle, showed you how to implement them from scratch, and presented a compact implementation using `gluon`. We explained *how to do things* in `gluon` but didn't really explain *how they work*. We relied on `nn.Sequential`, syntactically convenient shorthand for `nn.Block` but didn't peek under the hood. And while each notebook presented a working, trained model, we didn't show you how to inspect its parameters, save and load models, etc. In this chapter, we'll take a break from modeling to explore the gory details of `mxnet.gluon`.

### Load up the data

First, let's get the preliminaries out of the way.

```
In [1]: from __future__ import print_function
import mxnet as mx
import numpy as np
from mxnet import nd, autograd, gluon
from mxnet.gluon import nn, Block

#####
# Specify the context we'll be using
#####
ctx = mx.cpu()

#####
# Load up our dataset
#####
batch_size = 64
def transform(data, label):
    return data.astype(np.float32)/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
                                                                    batch_size, shuffle=True),
                                      transform=transform),
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
                                                                    batch_size, shuffle=False),
                                      transform=transform),
```

### Composing networks with `gluon.Block`

Now you might remember that up until now, we've defined neural networks (for, example a multilayer perceptron) like this:

```
In [2]: net1 = gluon.nn.Sequential()
```



```
with net1.name_scope():
    net1.add(gluon.nn.Dense(128, activation="relu"))
    net1.add(gluon.nn.Dense(64, activation="relu"))
    net1.add(gluon.nn.Dense(10))
```

This is a convenient shorthand that allows us to express a neural network compactly. When we want to build simple networks, this saves us a lot of time. But both (i) to understand how

`nn.Sequential` works, and (ii) to compose more complex architectures, you'll want to understand

`gluon.Block`.

Let's take a look at the same model would be expressed with `gluon.Block`.

```
In [3]: class MLP(Block):
        def __init__(self, **kwargs):
            super(MLP, self).__init__(**kwargs)
            with self.name_scope():
                self.dense0 = nn.Dense(128)
                self.dense1 = nn.Dense(64)
                self.dense2 = nn.Dense(10)

        def forward(self, x):
            x = nd.relu(self.dense0(x))
            x = nd.relu(self.dense1(x))
            return self.dense2(x)
```

Now that we've defined a class for MLPs, we can go ahead and instantiate one:

```
In [4]: net2 = MLP()
```

And initialize its parameters:

```
In [5]: net2.initialize(ctx=ctx)
```

At this point we can pass data through the network by calling it like a function, just as we have in the previous tutorials.

```
In [6]: for data, _ in train_data:
        data = data.as_in_context(ctx)
        break
        net2(data[0:1])
```

```
Out[6]: [[ 0.05750329  0.00230681 -0.012871   0.0038013 -0.04263662  0.03849379
           -0.04130694  0.03704495 -0.00853285  0.00490336]]
<NDArray 1x10 @cpu(0)>
```

## Calling `Block` as a function

Notice that `MLP` is a class and thus its instantiation, `net2`, is an object. If you're a casual Python user, you might be surprised to see that we can *call an object as a function*. This is a syntactic convenience owing to Python's `__call__` method. Basically, `gluon.Block.__call__(x)` is defined so that `net(data)` behaves identically to `net.forward(data)`. Since passing data through models is so fundamental and common, you'll be glad to save these 8 characters many times per day.

## So what is a `Block`?

In `gluon`, a `Block` is a generic component in a neural network. The entire network is a `Block`, each layer is a `Block`, and we can even have repeating sequences of layers that form an intermediate `Block`.

This might sound confusing, so let's make it simple. Each neural network has to do the following things: 1. Store parameters 2. Accept inputs 3. Produce outputs (the forward pass) 4. Take derivatives (the backward pass)

This can be said for the network as a whole, but it can also be said of each individual layer. A single fully-connected layer is parameterized by weight matrix and bias vector, produces outputs from inputs, and can given the derivative of some objective with respect to its outputs, can calculate the derivative with respect to its inputs.

Fortunately, MXNet can take derivatives automatically. So we only have to define the forward pass (`forward(self, x)`). Then, using `mxnet.autograd`, `gluon` can handle the backward pass. This is quite a powerful interface. For example we could define the forward pass for some component to take multiple inputs, and to combine them in arbitrary ways. We can even compose the `forward()` function such that it throws together a different architecture on the fly depending on some conditions that we could specify in Python. As long as the result is an NDArray, we're in the clear.

## What's the deal with `name_scope()`?

The next thing you might have noticed is that we added all of our layers inside a `with net1.name_scope():` block. This coerces `gluon` to give each parameter an appropriate name, indicating which model it belongs to, e.g. `sequential18_dense2_weight`. Keeping these names straight makes our lives much easier once we start writing more complex code where we might be working with multiple models and saving and loading the parameters of each. It helps us to make sure that we associate each weight with the right model.

## Demystifying `nn.Sequential`

So Sequential is basically a way of throwing together a Block on the fly. Let's revisit the `Sequential` version of our multilayer perceptron.

```
In [7]: net1 = gluon.nn.Sequential()
        with net1.name_scope():
            net1.add(gluon.nn.Dense(128, activation="relu"))
            net1.add(gluon.nn.Dense(64, activation="relu"))
            net1.add(gluon.nn.Dense(10))
```

In just 5 lines and 183 characters, we defined a multilayer perceptron with three fully-connected layers, each parametrized by weight matrix and bias term. We also specified the ReLU activation function for the hidden layers.

`Sequential` itself subclasses `Block` and maintains a list of `_children`. Then, every time we call `net1.add(...)` our net simply registers a new child. We can actually pass in an arbitrary `Block`, even layers that we write ourselves.

When we call `forward` on a `Sequential`, it executes the following code:

```
def forward(self, x):
    for block in self._children:
        x = block(x)
    return x
```

Basically, it calls each child on the output of the previous one, returning the final output at the end of the chain.

## Shape inference

One of the first things you might notice is that for each layer, we only specified the number of nodes output, we never specified how many input nodes! You might wonder, how does `gluon` know that the first weight matrix should be  $784 \times 128$  and not  $42 \times 128$ . In fact it doesn't. We can see this by accessing the network's parameters.

```
In [8]: print(net1.collect_params())

sequential1_ (
  Parameter sequential1_dense0_weight (shape=(128, 0), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense0_bias (shape=(128,), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense1_weight (shape=(64, 0), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense1_bias (shape=(64,), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense2_weight (shape=(10, 0), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense2_bias (shape=(10,), dtype=<class 'numpy.float32'>)
)
```

Take a look at the shapes of the weight matrices: (128,0), (64, 0), (10, 0). What does it mean to have zero dimension in a matrix? This is `gluon`'s way of marking that the shape of these matrices is not yet known. The shape will be inferred on the fly once the network is provided with some input.

So when we initialize our parameters, you might wonder, what precisely is happening?

```
In [9]: net1.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

In this situation, `gluon` is not actually initializing any parameters! Instead, it's making a note of which initializer to associate with each parameter, even though its shape is not yet known. The parameters are instantiated and the initializer is called once we provide the network with some input.

```
In [10]: net1(data)
print(net1.collect_params())

sequential1_ (
  Parameter sequential1_dense0_weight (shape=(128, 784), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense0_bias (shape=(128,), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense1_weight (shape=(64, 128), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense1_bias (shape=(64,), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense2_weight (shape=(10, 64), dtype=<class 'numpy.float32'>)
  Parameter sequential1_dense2_bias (shape=(10,), dtype=<class 'numpy.float32'>)
)
```

This shape inference can be extremely useful at times. For example, when working with convnets, it can be quite a pain to calculate the shape of various hidden layers. It depends on both the kernel size, the number of filters, the stride, and the precise padding scheme used which can vary in subtle ways from library to library.

## Specifying shape manually

If we want to specify the shape manually, that's always an option. We accomplish this by using the `in_units` argument when adding each layer.

```
In [11]: net2 = gluon.nn.Sequential()
with net2.name_scope():
    net2.add(gluon.nn.Dense(128, in_units=784, activation="relu"))
    net2.add(gluon.nn.Dense(64, in_units=128, activation="relu"))
    net2.add(gluon.nn.Dense(10, in_units=64))
```

Note that the parameters from this network can be initialized before we see any real data.

```
In [12]: net2.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
print(net2.collect_params())
```

```
sequential2_ (
  Parameter sequential2_dense0_weight (shape=(128, 784), dtype=<class 'numpy.float32'>)
  Parameter sequential2_dense0_bias (shape=(128,), dtype=<class 'numpy.float32'>)
  Parameter sequential2_dense1_weight (shape=(64, 128), dtype=<class 'numpy.float32'>)
  Parameter sequential2_dense1_bias (shape=(64,), dtype=<class 'numpy.float32'>)
  Parameter sequential2_dense2_weight (shape=(10, 64), dtype=<class 'numpy.float32'>)
  Parameter sequential2_dense2_bias (shape=(10,), dtype=<class 'numpy.float32'>)
)
```

## Next

*Writing custom layers with ``gluon.Block` <../chapter03\_deep-neural-networks/custom-layer.ipynb>`\_\_*

For whinges or inquiries, [open an issue on GitHub](#).

# Designing a custom layer with `gluon`

`nn.Sequential()``gluon``gluon``gluon``gluon`

```
In [1]: from __future__ import print_function
import mxnet as mx
import numpy as np
from mxnet import nd, autograd, gluon
from mxnet.gluon import nn, Block
mx.random.seed(1)

#####
# Specify the context we'll be using
#####
ctx = mx.cpu()

#####
# Load up our dataset
#####
batch_size = 64
def transform(data, label):
    return data.astype(np.float32)/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
```

```
transform=transform),
                                batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
transform=transform),
                                batch_size, shuffle=False)
```

## Defining a (toy) custom layer

gluon

Block

gluon

Block

```
In [2]: class CenteredLayer(Block):
        def __init__(self, **kwargs):
            super(CenteredLayer, self).__init__(**kwargs)

        def forward(self, x):
            return x - nd.mean(x)
```

```
In [3]: net = CenteredLayer()
        net(nd.array([1,2,3,4,5]))
```

```
Out[3]: [-2. -1.  0.  1.  2.]
        <NDArray 5 @cpu(0)>
```

nn.Sequential()

```
In [4]: net2 = nn.Sequential()
        net2.add(nn.Dense(128))
        net2.add(nn.Dense(10))
        net2.add(CenteredLayer())
```

```
In [5]: net2.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

```
In [6]: for data, _ in train_data:
        data = data.as_in_context(ctx)
        break
        output = net2(data[0:1])
        print(output)
```

```
[[ -0.10226583  0.10347994 -0.74226749  0.39843056  0.76840091  0.27723062
  0.01949821 -0.54039323  0.20809576 -0.39020956]]
<NDArray 1x10 @cpu(0)>
```

```
In [7]: nd.mean(output)
```

```
Out[7]: [-1.49011612e-08]
<NDArray 1 @cpu(0)>
```

2.68220894e-08

.000000027

## Custom layers with parameters

CenteredLayer

CenteredLayer

Block

## Parameters

Block

gluon

Block

Parameter

Parameter

NDArray

Parameter

Block



Block

```
In [8]: my_param = gluon.Parameter("exciting_parameter_yay", grad_req='write', shape=(5,5))
        print(my_param)
```

```
Parameter exciting_parameter_yay (shape=(5, 5), dtype=<class 'numpy.float32'>)
```

gluon

.attach\_grad()

.initialize()

.data()

```
In [9]: my_param.initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
        print(my_param.data())
```

```
[[ 0.50735062 -0.65750605 -0.56013602  0.46934015  0.1596154 ]
 [-0.65080845 -0.11559016  0.31085443 -0.49285054  0.57047993]
 [ 0.35613006  0.29938424  0.61431509  0.13020623  0.21408975]
 [-0.38888294  0.65209502 -0.08793807 -0.03835624  0.63372332]
 [-0.42945772 -0.36274379 -0.06317961 -0.58671117  0.2023437 ]]
<NDArray 5x5 @cpu(0)>
```

gluon.Trainer

```
In [10]: # my_param = gluon.Parameter("exciting_parameter_yay", grad_req='write', shape=(5,5))
        # my_param.initialize(mx.init.Xavier(magnitude=2.24), ctx=[mx.gpu(0), mx.gpu(1)])
        # print(my_param.data(mx.gpu(0)), my_param.data(mx.gpu(1)))
```

## Parameter dictionaries (introducing **ParameterDict**)

Parameters

Block

ParameterDict

ParameterDict

Block

```
In [11]: pd = gluon.ParameterDict(prefix="block1_")
```

ParameterDict

pd.get()

```
In [12]: pd.get("exciting_parameter_yay", grad_req='write', shape=(5,5))
```

```
Out[12]: Parameter block1_exciting_parameter_yay (shape=(5, 5), dtype=<class 'numpy.float32'>)
```

Block

Block

.keys()

```
In [13]: pd["block1_exciting_parameter_yay"]
```

```
Out[13]: Parameter block1_exciting_parameter_yay (shape=(5, 5), dtype=<class 'numpy.float32'>)
```

## Craft a bespoke fully-connected **gluon** layer

```
In [14]: def relu(X):  
         return nd.maximum(X, 0)
```

Block

```
In [15]: class MyDense(Block):  
         #####  
         # We add arguments to our constructor (__init__)  
         # to indicate the number of input units ('`in_units`')  
         # and output units ('`units`')  
         #####  
         def __init__(self, units, in_units=0, **kwargs):  
             super(MyDense, self).__init__(**kwargs)  
             with self.name_scope():  
                 self.units = units  
                 self._in_units = in_units  
                 #####  
                 # We add the required parameters to the ``Block``'s ParameterDict ,  
                 # indicating the desired shape  
                 #####  
                 self.weight = self.params.get(  
                     'weight', init=mx.init.Xavier(magnitude=2.24),  
                     shape=(in_units, units))  
                 self.bias = self.params.get('bias', shape=(units,))  
  
         #####  
         # Now we just have to write the forward pass.
```

```

# We could rely upon the FullyConnected primitive in NDArray,
# but it's better to get our hands dirty and write it out
# so you'll know how to compose arbitrary functions
#####
def forward(self, x):
    with x.context:
        linear = nd.dot(x, self.weight.data()) + self.bias.data()
        activation = relu(linear)
        return activation

```

```

In [16]: dense = MyDense(20, in_units=10)
         dense.collect_params().initialize(ctx=ctx)

```

```

In [17]: dense.params

```

```

Out[17]: mydense0_ (
  Parameter mydense0_weight (shape=(10, 20), dtype=<class 'numpy.float32'>)
  Parameter mydense0_bias (shape=(20,), dtype=<class 'numpy.float32'>)
)

```

```

In [18]: dense(nd.ones(shape=(2,10)))

```

```

Out[18]: [[ 0.         0.59868848  0.         1.08994353  0.         0.
  0.02280135  0.26122358  0.15244921  0.         0.         1.23705149
  0.535007     0.         0.         0.61897928  0.09488954  0.         0.
  0.46094614]
 [ 0.         0.59868848  0.         1.08994353  0.         0.
  0.02280135  0.26122358  0.15244921  0.         0.         1.23705149
  0.535007     0.         0.         0.61897928  0.09488954  0.         0.
  0.46094614]]
<NDArray 2x20 @cpu(0)>

```

## Using our layer to build an MLP

MyDense

nn.Sequential()

```

In [19]: net = gluon.nn.Sequential()
         with net.name_scope():
             net.add(MyDense(128, in_units=784))
             net.add(MyDense(64, in_units=128))
             net.add(MyDense(10, in_units=64))

```

# Initialize Parameters

```
In [20]: net.collect_params().initialize(ctx=ctx)
```

## Instantiate a loss

```
In [21]: loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Optimizer

```
In [22]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .1})
```

## Evaluation Metric

```
In [23]: metric = mx.metric.Accuracy()

def evaluate_accuracy(data_iterator, net):
    numerator = 0.
    denominator = 0.

    for i, (data, label) in enumerate(data_iterator):
        with autograd.record():
            data = data.as_in_context(ctx).reshape((-1, 784))
            label = label.as_in_context(ctx)
            label_one_hot = nd.one_hot(label, 10)
            output = net(data)

            metric.update([label], [output])
    return metric.get()[1]
```

## Training loop

```
In [24]: epochs = 2 # Low number for testing, set higher when you run!
moving_loss = 0.

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx).reshape((-1, 784))
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            cross_entropy = loss(output, label)
            cross_entropy.backward()
            trainer.step(data.shape[0])

    test_accuracy = evaluate_accuracy(test_data, net)
    train_accuracy = evaluate_accuracy(train_data, net)
    print("Epoch %s. Train_acc %s, Test_acc %s" % (e, train_accuracy, test_accuracy))
```

```
Epoch 0. Train_acc 0.750714285714, Test_acc 0.7489
Epoch 1. Train_acc 0.759571428571, Test_acc 0.75255
```

## Conclusion

## Next

In [25]:

---

# Serialization - saving, loading and checkpointing

gluon

gluon

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
from mxnet import gluon
ctx = mx.gpu()
```

## Saving and loading NDArrays

Pickle

ndarray.save

ndarray.load

```
In [2]: X = nd.ones((100, 100))
Y = nd.zeros((100, 100))
import os
os.makedirs('checkpoints', exist_ok=True)
filename = "checkpoints/test1.params"
nd.save(filename, [X, Y])
```

```
In [3]: A, B = nd.load(filename)
print(A)
```

```
print(B)
```

```
[[ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 ...,
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]]
<NDArray 100x100 @cpu(0)>

[[ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 ...,
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]]
<NDArray 100x100 @cpu(0)>
```

```
In [4]: mydict = {"X": X, "Y": Y}
        filename = "checkpoints/test2.params"
        nd.save(filename, mydict)
```

```
In [5]: C = nd.load(filename)
        print(C)
```

```
{'Y':
 [[ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 ...,
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]]
<NDArray 100x100 @cpu(0)>, 'X':
 [[ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 ...,
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]
 [ 1.  1.  1. ...,  1.  1.  1.]]
<NDArray 100x100 @cpu(0)>}
```

## Saving and loading the parameters of `gluon` models

`gluon`

Parameter

`gluon`

`.save_params()`

`.load_params()`

```
In [6]: num_hidden = 256
num_outputs = 1
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_outputs))
```

```
In [7]: net.collect_params().initialize(mx.init.Normal(sigma=1.), ctx=ctx)
net(nd.ones((1, 100), ctx=ctx))
```

```
Out[7]: [[ 381.35409546]]
<NDArray 1x1 @gpu(0)>
```

```
In [8]: filename = "checkpoints/testnet.params"
net.save_params(filename)
net2 = gluon.nn.Sequential()
with net2.name_scope():
    net2.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net2.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net2.add(gluon.nn.Dense(num_outputs))
net2.load_params(filename, ctx=ctx)
net2(nd.ones((1, 100), ctx=ctx))
```

```
Out[8]: [[ 381.35409546]]
<NDArray 1x1 @gpu(0)>
```

## Next





# Convolutional neural networks from scratch

Now let's take a look at *convolutional neural networks* (CNNs), the models people really use for classifying images.

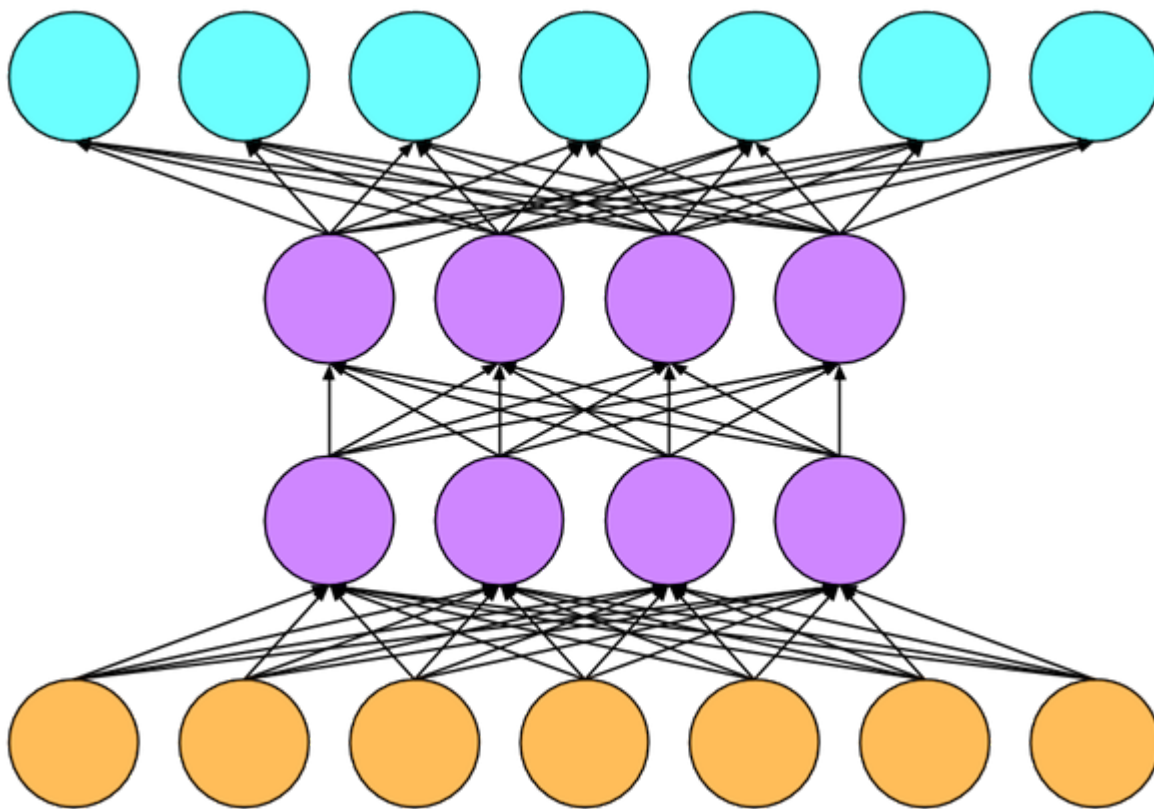
```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
import numpy as np
ctx = mx.gpu()
mx.random.seed(1)
```

## MNIST data (last one, we promise!)

```
In [2]: batch_size = 64
num_inputs = 784
num_outputs = 10
def transform(data, label):
    return nd.transpose(data.astype(np.float32), (2,0,1))/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
transform=transform),
                                     batch_size, shuffle=False)
```

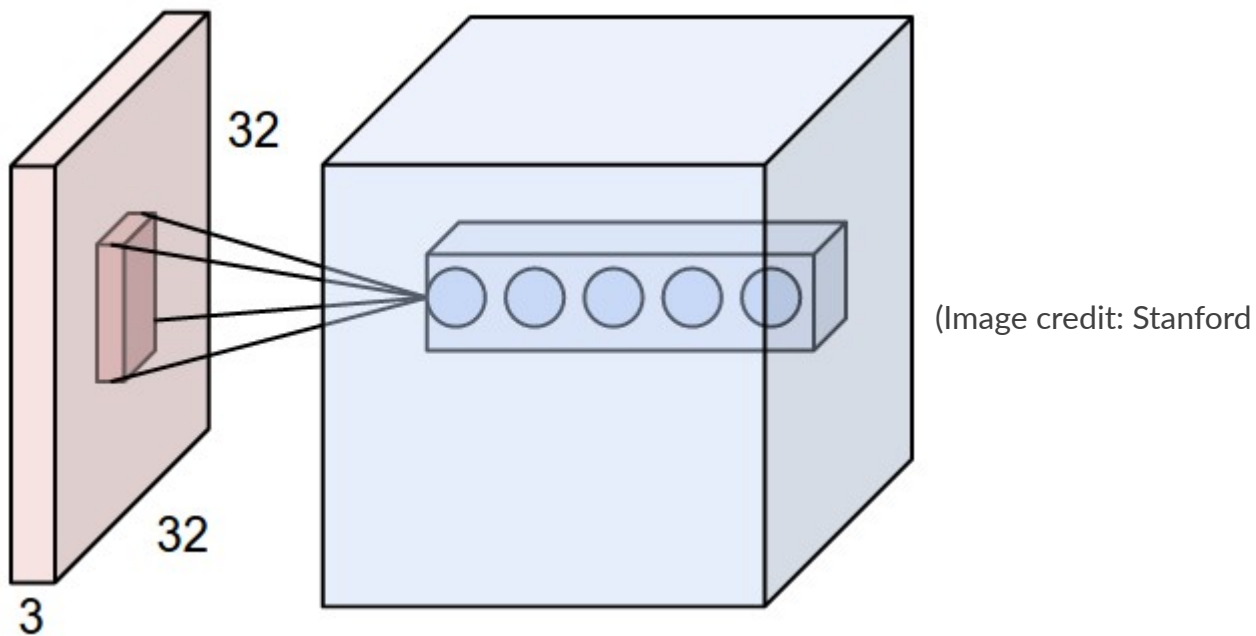
## Convolutional neural networks (CNNs)

In the [previous example](#), we connected the nodes of our neural networks in what seems like the simplest possible way. Every node in each layer was connected to every node in the subsequent layers.



This can require a lot of parameters! If our input were a 256x256 color image (still quite small for a photograph), and our network had 1,000 nodes in the first hidden layer, then our first weight matrix would require  $(256 \times 256 \times 3) \times 1000$  parameters. That's nearly 200 million. Moreover the hidden layer would ignore all the spatial structure in the input image even though we know the local structure represents a powerful source of prior knowledge.

Convolutional neural networks incorporate convolutional layers. These layers associate each of their nodes with a small window, called a *receptive field*, in the previous layer, instead of connecting to the full layer. This allows us to first learn local features via transformations that are applied in the same way for the top right corner as for the bottom left. Then we collect all this local information to predict global qualities of the image (like whether or not it depicts a dog).



cs231n <http://cs231n.github.io/assets/cnn/depthcol.jpeg>)

In short, there are two new concepts you need to grok here. First, we'll be introducing *convolutional* layers. Second, we'll be interleaving them with *pooling* layers.

## Parameters

Each node in convolutional layer is associated with a 3D block (height x width x channel) in the input tensor. Moreover, the convolutional layer itself has multiple output channels. So the layer is parameterized by a 4 dimensional weight tensor, commonly called a *convolutional kernel*.

The output tensor is produced by sliding the kernel across the input image skipping locations according to a pre-defined *stride* (but we'll just assume that to be 1 in this tutorial). Let's initialize some such kernels from scratch.

```
In [3]: #####
# Set the scale for weight initialization and choose
# the number of hidden units in the fully-connected layer
#####
weight_scale = .01
num_fc = 128

W1 = nd.random_normal(shape=(20, 1, 3,3), scale=weight_scale, ctx=ctx)
b1 = nd.random_normal(shape=20, scale=weight_scale, ctx=ctx)

W2 = nd.random_normal(shape=(50, 20, 5, 5), scale=weight_scale, ctx=ctx)
b2 = nd.random_normal(shape=50, scale=weight_scale, ctx=ctx)

W3 = nd.random_normal(shape=(800, num_fc), scale=weight_scale, ctx=ctx)
b3 = nd.random_normal(shape=128, scale=weight_scale, ctx=ctx)

W4 = nd.random_normal(shape=(num_fc, num_outputs), scale=weight_scale, ctx=ctx)
b4 = nd.random_normal(shape=10, scale=weight_scale, ctx=ctx)

params = [W1, b1, W2, b2, W3, b3, W4, b4]
```

And assign space for gradients

```
In [4]: for param in params:
        param.attach_grad()
```

## Convoluting with MXNet's NDArray

To write a convolution when using *raw MXNet*, we use the function `nd.Convolution()`. This function takes a few important arguments: inputs (`data`), a 4D weight matrix (`weight`), a bias (`bias`), the shape of the kernel (`kernel`), and a number of filters (`num_filter`).

```
In [5]: for data, _ in train_data:
        data = data.as_in_context(ctx)
        break
conv = nd.Convolution(data=data, weight=W1, bias=b1, kernel=(3,3), num_filter=20)
print(conv.shape)

(64, 20, 26, 26)
```

Note the shape. The number of examples (64) remains unchanged. The number of channels (also called *filters*) has increased to 20. And because the (3,3) kernel can only be applied in 26 different heights and widths (without the kernel busting over the image border), our output is 26,26. There are some weird padding tricks we can use when we want the input and output to have the same height and width dimensions, but we won't get into that now.

## Average pooling

The other new component of this model is the pooling layer. Pooling gives us a way to downsample in the spatial dimensions. Early convnets typically used average pooling, but max pooling tends to give better results.

```
In [6]: pool = nd.Pooling(data=conv, pool_type="max", kernel=(2,2), stride=(2,2))
        print(pool.shape)

(64, 20, 13, 13)
```

Note that the batch and channel components of the shape are unchanged but that the height and width have been downsampled from (26,26) to (13,13).

## Activation function

```
In [7]: def relu(X):
        return nd.maximum(X, nd.zeros_like(X))
```

# Softmax output

```
In [8]: def softmax(y_linear):
        exp = nd.exp(y_linear-nd.max(y_linear))
        partition = nd.sum(exp, axis=0, exclude=True).reshape((-1,1))
        return exp / partition
```

# Softmax cross-entropy loss

```
In [9]: def softmax_cross_entropy(yhat_linear, y):
        return - nd.nansum(y * nd.log_softmax(yhat_linear), axis=0, exclude=True)
```

# Define the model

Now we're ready to define our model

```
In [10]: def net(X, debug=False):
        #####
        # Define the computation of the first convolutional layer
        #####
        h1_conv = nd.Convolution(data=X, weight=W1, bias=b1, kernel=(3,3), num_filter=20)
        h1_activation = relu(h1_conv)
        h1 = nd.Pooling(data=h1_activation, pool_type="avg", kernel=(2,2), stride=(2,2))
        if debug:
            print("h1 shape: %s" % (np.array(h1.shape)))

        #####
        # Define the computation of the second convolutional layer
        #####
        h2_conv = nd.Convolution(data=h1, weight=W2, bias=b2, kernel=(5,5), num_filter=50)
        h2_activation = relu(h2_conv)
        h2 = nd.Pooling(data=h2_activation, pool_type="avg", kernel=(2,2), stride=(2,2))
        if debug:
            print("h2 shape: %s" % (np.array(h2.shape)))

        #####
        # Flattening h2 so that we can feed it into a fully-connected layer
        #####
        h2 = nd.flatten(h2)
        if debug:
            print("Flat h2 shape: %s" % (np.array(h2.shape)))

        #####
        # Define the computation of the third (fully-connected) layer
        #####
        h3_linear = nd.dot(h2, W3) + b3
        h3 = relu(h3_linear)
        if debug:
            print("h3 shape: %s" % (np.array(h3.shape)))

        #####
        # Define the computation of the output layer
        #####
        yhat_linear = nd.dot(h3, W4) + b4
        if debug:
            print("yhat_linear shape: %s" % (np.array(yhat_linear.shape)))

        return yhat_linear
```

# Test run

We can now print out the shapes of the activations at each layer by using the debug flag.

```
In [11]: output = net(data, debug=True)
```

```
h1 shape: [64 20 13 13]
h2 shape: [64 50 4 4]
Flat h2 shape: [ 64 800]
h3 shape: [ 64 128]
yhat_linear shape: [64 10]
```

## Optimizer

```
In [12]: def SGD(params, lr):
          for param in params:
              param[:] = param - lr * param.grad
```

## Evaluation metric

```
In [13]: def evaluate_accuracy(data_iterator, net):
          numerator = 0.
          denominator = 0.
          for i, (data, label) in enumerate(data_iterator):
              data = data.as_in_context(ctx)
              label = label.as_in_context(ctx)
              label_one_hot = nd.one_hot(label, 10)
              output = net(data)
              predictions = nd.argmax(output, axis=1)
              numerator += nd.sum(predictions == label)
              denominator += data.shape[0]
          return (numerator / denominator).asscalar()
```

## The training loop

```
In [14]: epochs = 1
          learning_rate = .01
          smoothing_constant = .01

          for e in range(epochs):
              for i, (data, label) in enumerate(train_data):
                  data = data.as_in_context(ctx)
                  label = label.as_in_context(ctx)
                  label_one_hot = nd.one_hot(label, num_outputs)
                  with autograd.record():
                      output = net(data)
                      loss = softmax_cross_entropy(output, label_one_hot)
                  loss.backward()
                  SGD(params, learning_rate)

                  #####
                  # Keep a moving average of the losses
                  #####
                  curr_loss = nd.mean(loss).asscalar()
                  moving_loss = (curr_loss if ((i == 0) and (e == 0))
                                else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
                                * curr_loss)
```

```
test_accuracy = evaluate_accuracy(test_data, net)
train_accuracy = evaluate_accuracy(train_data, net)
print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, moving_loss,
train_accuracy, test_accuracy))
```

Epoch 0. Loss: 0.140828552357, Train\_acc 0.9532, Test\_acc 0.9524

## Conclusion

Contained in this example are nearly all the important ideas you'll need to start attacking problems in computer vision. While state-of-the-art vision systems incorporate a few more bells and whistles, they're all built on this foundation. Believe it or not, if you knew just the content in this tutorial 5 years ago, you could probably have sold a startup to a Fortune 500 company for millions of dollars. Fortunately (or unfortunately?), the world has gotten marginally more sophisticated, so we'll have to come up with some more sophisticated tutorials to follow.

## Next

[Convolutional neural networks with gluon](#)

For whinges or inquiries, [open an issue on GitHub](#).



# Convolutional Neural Networks in `gluon`

Now let's see how succinctly we can express a convolutional neural network using `gluon`. You might be relieved to find out that this too requires hardly any more code than a logistic regression.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
from mxnet import gluon
import numpy as np
mx.random.seed(1)
```

## Set the context

```
In [2]: ctx = mx.gpu()
```

## Grab the MNIST dataset

```
In [3]: batch_size = 64
num_inputs = 784
num_outputs = 10
def transform(data, label):
    return nd.transpose(data.astype(np.float32), (2,0,1))/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
    transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
    transform=transform),
                                     batch_size, shuffle=False)
```

## Define a convolutional neural network

Again, a few lines here is all we need in order to change the model. Let's add a couple of convolutional layers using `gluon.nn`.

```
In [4]: num_fc = 512
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Conv2D(channels=20, kernel_size=5, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))
    net.add(gluon.nn.Conv2D(channels=50, kernel_size=5, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))
    # The Flatten layer collapses all axis, except the first one, into one axis.
    net.add(gluon.nn.Flatten())
```

```
net.add(gluon.nn.Dense(num_fc, activation="relu"))
net.add(gluon.nn.Dense(num_outputs))
```

## Parameter initialization

```
In [5]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

## Softmax cross-entropy Loss

```
In [6]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Optimizer

```
In [7]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .1})
```

## Write evaluation loop to calculate accuracy

```
In [8]: def evaluate_accuracy(data_iterator, net):
    acc = mx.metric.Accuracy()
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx)
        label = label.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

## Training Loop

```
In [9]: epochs = 1
    smoothing_constant = .01

    for e in range(epochs):
        for i, (data, label) in enumerate(train_data):
            data = data.as_in_context(ctx)
            label = label.as_in_context(ctx)
            with autograd.record():
                output = net(data)
                loss = softmax_cross_entropy(output, label)
            loss.backward()
            trainer.step(data.shape[0])

            #####
            # Keep a moving average of the losses
            #####
            curr_loss = nd.mean(loss).asscalar()
            moving_loss = (curr_loss if ((i == 0) and (e == 0))
                           else (1 - smoothing_constant) * moving_loss + smoothing_constant *
                               curr_loss)

            test_accuracy = evaluate_accuracy(test_data, net)
            train_accuracy = evaluate_accuracy(train_data, net)
```

```
print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, moving_loss,
train_accuracy, test_accuracy))
```

Epoch 0. Loss: 0.0823292345241, Train\_acc 0.974483333333, Test\_acc 0.9759

## Conclusion

You might notice that by using `gluon`, we get code that runs much faster whether on CPU or GPU. That's largely because `gluon` can call down to highly optimized layers that have been written in C++.

## Next

[Deep convolutional networks \(AlexNet\)](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Deep convolutional neural networks

In the previous chapters, you got a sense for how to classify images with convolutional neural network (CNNs). Specifically, we implemented a CNN with two convolutional layers interleaved with pooling layers, a singly fully-connected hidden layer, and a softmax output layer. That architecture loosely resembles a neural network affectionately named LeNet, in honor [Yann LeCun](#), an early pioneer of convolutional neural networks and the first to [reduced them to practice in 1989](#) by training them with gradient descent (i.e. backpropagation). At the time, this was fairly novel idea. A cadre of researchers interested in biologically-inspired learning models had taken to investigating artificial simulations of neurons as learning models. However, as remains true to this day, few researchers believed that real brains learn by gradient descent. The community of neural networks researchers had explored many other learning rules. LeCun demonstrated that CNNs trained by gradient descent, could get state-of-the-art results on the task of recognizing hand-written digits. These groundbreaking results put CNNs on the map.

However, in the intervening years, neural networks were superseded by numerous other methods. Neural networks were considered slow to train, and there wasn't wide consensus on whether it was possible to train very deep neural networks from a random initialization of the weights. Moreover, training networks with many channels, layers, and parameters required excessive computation relative to the resources available decades ago. While it was possible to train a LeNet for MNIST digit classification and get good scores, neural networks fell out of favor on larger, real-world datasets.

Instead researchers precomputed features based on a mixture of elbow grease, knowledge of optics, and black magic. A typical pattern was this: 1. Grab a cool dataset 2. Preprocess it with giant bag of predetermined feature functions 3. Dump the representations into a simple linear model to do the *machine learning part*.

This was the state of affairs in computer vision up until 2012, just before deep learning began to change the world of applied machine learning. One of us (Zack) entered graduate school in 2013. A friend in graduate school summarized the state of affairs thus:

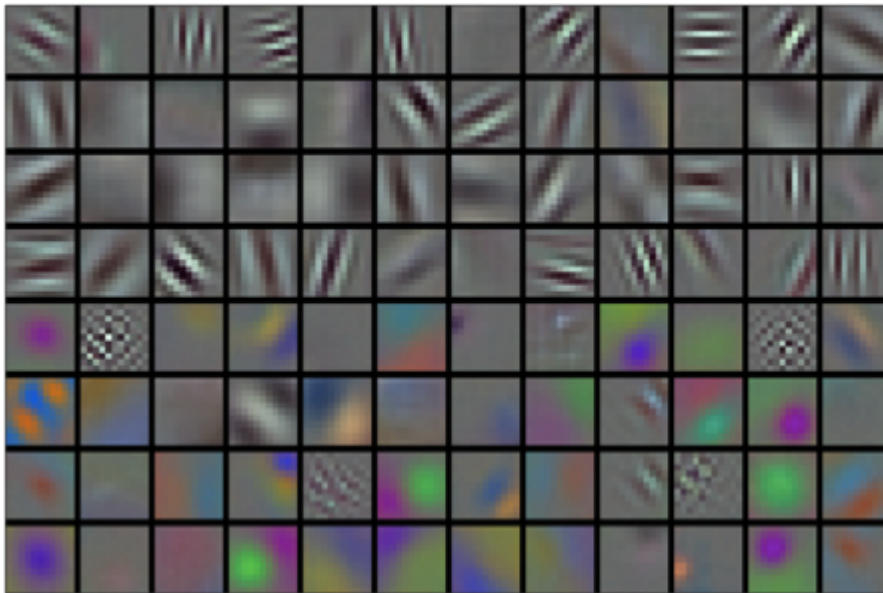
If you spoke to machine learning researchers, they believed that machine learning was both important and beautiful. Elegant theories proved the properties of various classifiers. The field of machine learning was thriving, rigorous and eminently useful. However, if you spoke to a computer vision researcher, you'd hear a very different story. The dirty truth of image recognition, they'd tell you, is that the really important aspects of the ML for CV pipeline were

data and features. A slightly cleaner dataset, or a slightly better hand-tuned feature mattered a lot to the final accuracy. However, the specific choice of classifier was little more than an afterthought. At the end of the day you could throw your features in a logistic regression model, a support vector machine, or any other classifier of choice, and they would all perform roughly the same.

## Learning the representations

Another way to cast the state of affairs is that the most important part of the pipeline was the representation. And up until 2012, this part was done mechanically, based on some hard-fought intuition. In fact, engineering a new set of feature functions, improving results, and writing up the method was a prominent genre of paper.

Another group of researchers had other plans. They believed that features themselves ought to be learned. Moreover they believed that to be reasonably complex, the features ought to be hierarchically composed. These researchers, including Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber believed that by jointly training many layers of a neural network, they might come to learn hierarchical representations of data. In the case of an image, the lowest layers might come to detect edges, colors, and textures.



Higher layers might build upon these representations to represent larger structures, like eyes, noses, blades of grass, and features. Yet higher layers might represent whole objects like people, airplanes, dogs, or frisbees. And ultimately, before the classification layer, the final hidden state might represent a compact representation of the image that summarized the contents in a space where data belonging to different categories would be linearly separable.

## Missing ingredient 1: data

Despite the sustained interest of a committed group of researchers in learning deep representations of visual data, for a long time these ambitions were frustrated. The failures to make progress owed to a few factors. First, while this wasn't yet known, supervised deep models with many representation require large amounts of labeled training data in order to outperform classical approaches. However, given the limited storage capacity of computers and the comparatively tighter research budgets in the 1990s and prior, most research relied on tiny datasets. For example, many credible research papers relied on a small set of corpora hosted by UCI, many of which contained hundreds or a few thousand images.

This changed in a big way when Fei-Fei Li presented the ImageNet database in 2009. The ImageNet dataset dwarfed all previous research datasets. It contained one million images: one thousand each from one thousand distinct classes.



This dataset pushed both computer vision and machine learning research into a new regime where the previous best methods would no longer dominate.

## **Missing ingredient 2: hardware**

Deep Learning has a voracious need for computation. This is one of the main reasons why in the 90s and early 2000s algorithms based on convex optimization were the preferred way of solving problems. After all, convex algorithms have fast rates of convergence, global minima, and efficient algorithms can be found.

The game changer was the availability of GPUs. They had long been tuned for graphics processing in computer games. In particular, they were optimized for high throughput 4x4 matrix-vector products, since these are needed for many computer graphics tasks. Fortunately, the math required for that is very similar to convolutional layers in deep networks. Furthermore, around that time, NVIDIA and ATI had begun optimizing GPUs for general compute operations, going as far as renaming them GPGPU (General Purpose GPUs).

To provide some intuition, consider the cores of a modern microprocessor. Each of the cores is quite powerful, running at a high clock frequency, it has quite advanced and large caches (up to several MB of L3). Each core is very good at executing a very wide range of code, with branch predictors, a deep pipeline and lots of other things that make it great at executing regular programs. This apparent strength, however, is also its Achilles' heel: general purpose cores are very expensive to build. They require lots of chip area, a sophisticated support structure (memory interfaces, caching logic between cores, high speed interconnects, etc.), and they're comparatively bad at any single task. Modern laptops have up to 4 cores, and even high end servers rarely exceed 64 cores, simply because it is not cost effective.

Compare that with GPUs. They consist of 100-1000 small processing elements (the details differ somewhat between NVIDIA, ATI, ARM and other chip vendors), often grouped into larger groups (NVIDIA calls them warps). While each core is relatively weak, running at sub-1GHz clock frequency, it is the total number of such cores that makes GPUs orders of magnitude faster than CPUs. For instance, NVIDIA's latest Volta generation offers up to 120 TFlops per chip for specialized instructions (and up to 24 TFlops for more general purpose ones), while floating point performance of CPUs has not exceeded 1 TFlop to date. The reason for why this is possible is actually quite simple: firstly, power consumption tends to grow *quadratically* with clock frequency. Hence, for the power budget of a CPU core that runs 4x faster (a typical number) you can use 16 GPU cores at 1/4 the speed, which yields  $16 \times 1/4 = 4x$  the performance. Furthermore GPU cores are much simpler (in fact, for a long time they weren't even *able* to execute general purpose code), which makes them more energy efficient. Lastly, many operations in deep learning require high memory bandwidth. Again, GPUs shine here with buses that are at least 10x as wide as many CPUs.

Back to 2012. A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep convolutional neural network that could run on GPU hardware. They realized that the computational bottlenecks in CNNs (convolutions and matrix multiplications) are all operations that could be parallelized in hardware. Using two NVIDIA GTX 580s with 3GB of memory (depicted below) they implemented fast convolutions. The code [cuda-convnet](#) was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.



## AlexNet

In 2012, using their cuda-convnet implementation on an eight-layer CNN, Khrizhevsky, Sutskever and Hinton won the ImageNet challenge on image recognition by a wide margin. Their model, [introduced in this paper](#), is very similar to the LeNet architecture from 1995.

In the rest of the chapter we're going to implement a similar model to the one designed by them. Due to memory constraints on the GPU they did some wacky things to make the model fit. For example, they designed a dual-stream architecture in which half of the nodes live on each GPU. The two streams, and thus the two GPUs only communicate at certain layers. This limits the amount of overhead for keeping the two GPUs in sync with each other. Fortunately, distributed deep learning has advanced a long way in the last few years, so we won't be needing those features (except for very unusual architectures). In later sections, we'll go into greater depth on how you can speed up your networks by training on many GPUs (in AWS you can get up to 16 on a single machine with 12GB each), and how you can train on many machine simultaneously. As usual, we'll start by importing the same dependencies as in the past gluon tutorials:

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
from mxnet import gluon
import numpy as np
mx.random.seed(1)
```



```
In [2]: ctx = mx.gpu()
```

## Load up a dataset

Now let's load up a dataset. This time we're going to use gluon's new `vision` package, and import the CIFAR dataset. Cifar is a much smaller color dataset, roughly the dimensions of ImageNet. It contains 50,000 training and 10,000 test images. The images belong in equal quantities to 10 categories. While this dataset is considerably smaller than the 1M image, 1k category, 256x256 ImageNet dataset, we'll use it here to demonstrate the model because we don't want to assume that you have a license for the ImageNet dataset or a machine that can store it comfortably. To give you some sense for the proportions of working with ImageNet data, we'll upsample the images to 224x224 (the size used in the original AlexNet).

```
In [3]: def transformer(data, label):
        data = mx.image.imresize(data, 224, 224)
        data = mx.nd.transpose(data, (2,0,1))
        data = data.astype(np.float32)
        return data, label
```

```
In [4]: batch_size = 64
        train_data = gluon.data.DataLoader(
            gluon.data.vision.CIFAR10('./data', train=True, transform=transformer),
            batch_size=batch_size, shuffle=True, last_batch='discard')

        test_data = gluon.data.DataLoader(
            gluon.data.vision.CIFAR10('./data', train=False, transform=transformer),
            batch_size=batch_size, shuffle=False, last_batch='discard')
```

Downloading ./data/cifar-10-binary.tar.gz from <https://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz>...

```
In [5]: for d, l in train_data:
        break
```

```
In [6]: print(d.shape, l.shape)
```

```
(64, 3, 224, 224) (64,)
```

```
In [7]: d.dtype
```

```
Out[7]: numpy.float32
```

## The AlexNet architecture

This model has some notable features. First, in contrast to the relatively tiny LeNet, AlexNet contains 8 layers of transformations, five convolutional layers followed by two fully connected hidden layers and an output layer.

The convolutional kernels in the first convolutional layer are reasonably large at  $11 \times 11$ , in the second they are  $5 \times 5$  and thereafter they are  $3 \times 3$ . Moreover, the first, second, and fifth convolutional layers are each followed by overlapping pooling operations with pool size  $3 \times 3$  and stride  $(2 \times 2)$ .

Following the convolutional layers, the original AlexNet had fully-connected layers with 4096 nodes each. Using `gluon.nn.Sequential()`, we can define the entire AlexNet architecture in just 14 lines of code. Besides the specific architectural choices and the data preparation, we can recycle all of the code we'd used for LeNet verbatim.

[right now relying on a different data pipeline (the new gluon.vision). Sync this with the other chapter soon and commit to one data pipeline.]

[add dropout once we are 100% final on API]

```
In [8]: alex_net = gluon.nn.Sequential()
        with alex_net.name_scope():
            # First convolutional layer
            alex_net.add(gluon.nn.Conv2D(channels=96, kernel_size=11, strides=(4,4),
            activation='relu'))
            alex_net.add(gluon.nn.MaxPool2D(pool_size=3, strides=2))
            # Second convolutional layer
            alex_net.add(gluon.nn.Conv2D(channels=192, kernel_size=5, activation='relu'))
            alex_net.add(gluon.nn.MaxPool2D(pool_size=3, strides=(2,2)))
            # Third convolutional layer
            alex_net.add(gluon.nn.Conv2D(channels=384, kernel_size=3, activation='relu'))
            # Fourth convolutional layer
            alex_net.add(gluon.nn.Conv2D(channels=384, kernel_size=3, activation='relu'))
            # Fifth convolutional layer
            alex_net.add(gluon.nn.Conv2D(channels=256, kernel_size=3, activation='relu'))
            alex_net.add(gluon.nn.MaxPool2D(pool_size=3, strides=2))
            # Flatten and apply fully connected layers
            alex_net.add(gluon.nn.Flatten())
            alex_net.add(gluon.nn.Dense(4096, activation="relu"))
            alex_net.add(gluon.nn.Dense(4096, activation="relu"))
            alex_net.add(gluon.nn.Dense(10))
```

## Initialize parameters

```
In [9]: alex_net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

## Optimizer

```
In [10]: trainer = gluon.Trainer(alex_net.collect_params(), 'sgd', {'learning_rate': .001})
```

## Softmax cross-entropy loss

```
In [11]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

# Evaluation loop

```
In [12]: def evaluate_accuracy(data_iterator, net):
    acc = mx.metric.Accuracy()
    for d, l in data_iterator:
        data = d.as_in_context(ctx)
        label = l.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

# Training loop

```
In [13]: #####
# Only one epoch so tests can run quickly, increase this variable to actually run
#####
epochs = 1
smoothing_constant = .01

for e in range(epochs):
    for i, (d, l) in enumerate(train_data):
        data = d.as_in_context(ctx)
        label = l.as_in_context(ctx)
        with autograd.record():
            output = alex_net(data)
            loss = softmax_cross_entropy(output, label)
        loss.backward()
        trainer.step(data.shape[0])

        #####
        # Keep a moving average of the losses
        #####
        curr_loss = nd.mean(loss).asscalar()
        moving_loss = (curr_loss if ((i == 0) and (e == 0))
                        else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
                        * curr_loss)

        test_accuracy = evaluate_accuracy(test_data, alex_net)
        train_accuracy = evaluate_accuracy(train_data, alex_net)
        print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, moving_loss,
                                                                    train_accuracy, test_accuracy))
```

Epoch 0. Loss: 1.83321327117, Train\_acc 0.359094910371, Test\_acc 0.347956730769

## Next

Very deep convolutional neural nets with repeating blocks

For whinges or inquiries, [open an issue on GitHub](#).

---

# Very deep networks with repeating elements

## VGG

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
from mxnet import gluon
import numpy as np
mx.random.seed(1)
```

```
In [2]: ctx = mx.gpu()
```

## Load up a dataset

```
In [3]: batch_size = 64

def transform(data, label):
    return nd.transpose(data.astype(np.float32), (2,0,1))/255, label.astype(np.float32)

train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
transform=transform),
                                     batch_size, shuffle=False)
```

## The VGG architecture

```
In [4]: from mxnet.gluon import nn

def vgg_block(num_convs, channels):
    out = nn.Sequential()
    for _ in range(num_convs):
        out.add(nn.Conv2D(channels=channels, kernel_size=3,
                           padding=1, activation='relu'))
    out.add(nn.MaxPool2D(pool_size=2, strides=2))
    return out

def vgg_stack(architecture):
    out = nn.Sequential()
    for (num_convs, channels) in architecture:
        out.add(vgg_block(num_convs, channels))
    return out

num_outputs = 10
architecture = ((1,64), (1,128), (2,256), (2,512))
net = nn.Sequential()
with net.name_scope():
    net.add(vgg_stack(architecture))
    net.add(nn.Flatten())
    net.add(nn.Dense(512, activation="relu"))
    net.add(nn.Dropout(.5))
    net.add(nn.Dense(512, activation="relu"))
    net.add(nn.Dropout(.5))
    net.add(nn.Dense(num_outputs))
```

## Initialize parameters

```
In [5]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

## Optimizer

```
In [6]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .05})
```

## Softmax cross-entropy loss

```
In [7]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Evaluation loop

```
In [8]: def evaluate_accuracy(data_iterator, net):
    acc = mx.metric.Accuracy()
    for d, l in data_iterator:
        data = d.as_in_context(ctx)
        label = l.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
```

```
acc.update(preds=predictions, labels=label)
return acc.get()[1]
```

## Training loop

```
In [9]: #####
# Only one epoch so tests can run quickly, increase this variable to actually run
#####
epochs = 1
smoothing_constant = .01

for e in range(epochs):
    for i, (d, l) in enumerate(train_data):
        data = d.as_in_context(ctx)
        label = l.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output, label)
        loss.backward()
        trainer.step(data.shape[0])

        #####
        # Keep a moving average of the losses
        #####
        curr_loss = nd.mean(loss).asscalar()
        moving_loss = (curr_loss if ((i == 0) and (e == 0))
                        else (1 - smoothing_constant) * moving_loss + smoothing_constant *
curr_loss)

        if i > 0 and i % 200 == 0:
            print('Batch %d. Loss: %f' % (i, moving_loss))

    test_accuracy = evaluate_accuracy(test_data, net)
    train_accuracy = evaluate_accuracy(train_data, net)
    print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, moving_loss,
train_accuracy, test_accuracy))
```

```
Batch 200. Loss: 2.299252
Batch 400. Loss: 2.272448
Batch 600. Loss: 1.222286
Batch 800. Loss: 0.454571
Epoch 0. Loss: 0.288249925115, Train_acc 0.939216666667, Test_acc 0.9427
```

## Next

## Batch Normalization from scratch

When you train a linear model, you update the weights in order to optimize some objective. And for the linear model, the distribution of the inputs stays the same throughout training. So all we have to worry about is how to map from these well-behaved inputs to some appropriate outputs. But if we focus on some layer in the middle of a deep neural network, for example the third, things look a bit different. After each training iteration, we update the weights in all the layers, including the first and the second. That means that over the course of training, as the weights for the first two layers are learned, the inputs to the third layer might look dramatically different than they did at the beginning. For starters, they might take values on a scale orders of magnitudes different from when we started training. And this shift in feature scale might have serious implications, say for the ideal learning rate at each time.

To explain, let us consider the Taylor's expansion for the objective function  $f$  with respect to the updated parameter  $\mathbf{w}$ , such as  $f(\mathbf{w} - \eta \nabla f(\mathbf{w}))$ . Coefficients of those higher-order terms with respect to the learning rate  $\eta$  may be so large in scale (usually due to many layers) that these terms cannot be ignored. However, the effect of common lower-order optimization algorithms, such as gradient descent, in iteratively reducing the objective function is based on an important assumption: all those higher-order terms with respect to the learning rate in the aforementioned Taylor's expansion are ignored.

Motivated by this sort of intuition, Sergey Ioffe and Christian Szegedy proposed [Batch Normalization](#), a technique that normalizes the mean and variance of each of the features at every level of representation during training. The technique involves normalization of the features across the examples in each mini-batch. While competing explanations for the technique's effect abound, its success is hard to deny. Empirically it appears to stabilize the gradient (less exploding or vanishing values) and batch-normalized models appear to overfit less. In fact, batch-normalized models seldom even use dropout. In this notebook, we'll explain how it works.

## Import dependencies and grab the MNIST dataset

We'll get going by importing the typical packages and grabbing the MNIST data.

```
In [1]: from __future__ import print_function
import mxnet as mx
import numpy as np
from mxnet import nd, autograd
```

```
mx.random.seed(1)
ctx = mx.gpu()
```

## The MNIST dataset

```
In [2]: batch_size = 64
num_inputs = 784
num_outputs = 10
def transform(data, label):
    return nd.transpose(data.astype(np.float32), (2,0,1))/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
transform=transform),
                                     batch_size, shuffle=False)
```

## Batch Normalization layer

The layer, unlike Dropout, is usually used **before** the activation layer (according to the authors' original paper), instead of after activation layer.

The basic idea is doing the normalization then applying a linear scale and shift to the mini-batch:

For input mini-batch  $B = \{x_1, \dots, x_m\}$ , we want to learn the parameter  $\gamma$  and  $\beta$ . The output of the layer is  $\{y_i = BN_{\gamma, \beta}(x_i)\}$ , where:

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i)$$

- formulas taken from Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." International Conference on Machine Learning. 2015.

With gluon, this is all actually implemented for us, but we'll do it this one time by ourselves, using the formulas from the original paper so you know how it works, and perhaps you can improve upon it!



Pay attention that, when it comes to (2D) CNN, we normalize `batch_size * height * width` over each channel. So that `gamma` and `beta` have the lengths the same as `channel_count`. In our implementation, we need to manually reshape `gamma` and `beta` so that they could (be automatically broadcast and) multiply the matrices in the desired way.

```
In [3]: def pure_batch_norm(X, gamma, beta, eps = 1e-5):
        if len(X.shape) not in (2, 4):
            raise ValueError('only supports dense or 2dconv')

        # dense
        if len(X.shape) == 2:
            # mini-batch mean
            mean = nd.mean(X, axis=0)
            # mini-batch variance
            variance = nd.mean((X - mean) ** 2, axis=0)
            # normalize
            X_hat = (X - mean) * 1.0 / nd.sqrt(variance + eps)
            # scale and shift
            out = gamma * X_hat + beta

        # 2d conv
        elif len(X.shape) == 4:
            # extract the dimensions
            N, C, H, W = X.shape
            # mini-batch mean
            mean = nd.mean(X, axis=(0, 2, 3))
            # mini-batch variance
            variance = nd.mean((X - mean.reshape((1, C, 1, 1))) ** 2, axis=(0, 2, 3))
            # normalize
            X_hat = (X - mean.reshape((1, C, 1, 1))) * 1.0 / nd.sqrt(variance.reshape((1, C, 1, 1)) + eps)
            # scale and shift
            out = gamma.reshape((1, C, 1, 1)) * X_hat + beta.reshape((1, C, 1, 1))

        return out
```

Let's do some sanity checks. We expect each **column** of the input matrix to be normalized.

```
In [4]: A = nd.array([1,7,5,4,6,10], ctx=ctx).reshape((3,2))
A
```

```
Out[4]:
[[ 1.  7.]
 [ 5.  4.]
 [ 6. 10.]]
<NDArray 3x2 @gpu(0)>
```

```
In [5]: pure_batch_norm(A,
        gamma = nd.array([1,1], ctx=ctx),
        beta=nd.array([0,0], ctx=ctx))
```

```
Out[5]:
[[-1.38872862  0.]
 [ 0.46290955 -1.22474384]
 [ 0.9258191  1.22474384]]
<NDArray 3x2 @gpu(0)>
```

```
In [6]: ga = nd.array([1,1], ctx=ctx)
        be = nd.array([0,0], ctx=ctx)

        B = nd.array([1,6,5,7,4,3,2,5,6,3,2,4,5,3,2,5,6], ctx=ctx).reshape((2,2,2,2))
        B
```

Out[6]:

```
[[[ 1.  6.]
   [ 5.  7.]]

  [[ 4.  3.]
   [ 2.  5.]]]

[[[ 6.  3.]
   [ 2.  4.]]

  [[ 5.  3.]
   [ 2.  5.]]]]
<NDArray 2x2x2x2 @gpu(0)>
```

In [7]: `pure_batch_norm(B, ga, be)`

Out[7]:

```
[[[[-1.63784397  0.88191599]
    [ 0.37796399  1.38586795]]

  [[ 0.30779248 -0.51298743]
    [-1.33376741  1.12857234]]]

[[[ 0.88191599 -0.62993997]
    [-1.13389194 -0.12598799]]

  [[ 1.12857234 -0.51298743]
    [-1.33376741  1.12857234]]]]
<NDArray 2x2x2x2 @gpu(0)>
```

Our tests seem to support that we've done everything correctly. Note that for batch normalization, implementing **backward** pass is a little bit tricky. Fortunately, you won't have to worry about that here, because the MXNet's `autograd` package can handle differentiation for us automatically.

Besides that, in the testing process, we want to use the mean and variance of the **complete dataset**, instead of those of **mini batches**. In the implementation, we use moving statistics as a trade off, because we don't want to or don't have the ability to compute the statistics of the complete dataset (in the second loop).

Then here comes another concern: we need to maintain the moving statistics **along with multiple runs of the BN**. It's an engineering issue rather than a deep/machine learning issue. On the one hand, the moving statistics are similar to `gamma` and `beta`; on the other hand, they are **not** updated by the gradient backwards. In this quick-and-dirty implementation, we use the global dictionary variables to store the statistics, in which each key is the name of the layer (`scope_name`), and the value is the statistics. (**Attention:** always be very careful if you have to use global variables!) Moreover, we have another parameter `is_training` to indicate whether we are doing training or testing.

Now we are ready to define our complete `batch_norm()`:

```

In [8]: def batch_norm(X,
                    gamma,
                    beta,
                    momentum = 0.9,
                    eps = 1e-5,
                    scope_name = '',
                    is_training = True,
                    debug = False):
    """compute the batch norm """
    global _BN_MOVING_MEANS, _BN_MOVING_VARS

    #####
    # the usual batch norm transformation
    #####

    if len(X.shape) not in (2, 4):
        raise ValueError('the input data shape should be one of:\n' +
                        'dense: (batch size, # of features)\n' +
                        '2d conv: (batch size, # of features, height, width)'
                        )

    # dense
    if len(X.shape) == 2:
        # mini-batch mean
        mean = nd.mean(X, axis=0)
        # mini-batch variance
        variance = nd.mean((X - mean) ** 2, axis=0)
        # normalize
        if is_training:
            # while training, we normalize the data using its mean and variance
            X_hat = (X - mean) * 1.0 / nd.sqrt(variance + eps)
        else:
            # while testing, we normalize the data using the pre-computed mean and
            variance
            X_hat = (X - _BN_MOVING_MEANS[scope_name]) * 1.0 /
            nd.sqrt(_BN_MOVING_VARS[scope_name] + eps)
        # scale and shift
        out = gamma * X_hat + beta

    # 2d conv
    elif len(X.shape) == 4:
        # extract the dimensions
        N, C, H, W = X.shape
        # mini-batch mean
        mean = nd.mean(X, axis=(0,2,3))
        # mini-batch variance
        variance = nd.mean((X - mean.reshape((1, C, 1, 1))) ** 2, axis=(0, 2, 3))
        # normalize
        X_hat = (X - mean.reshape((1, C, 1, 1))) * 1.0 / nd.sqrt(variance.reshape((1, C,
1, 1)) + eps)
        if is_training:
            # while training, we normalize the data using its mean and variance
            X_hat = (X - mean.reshape((1, C, 1, 1))) * 1.0 / nd.sqrt(variance.reshape((1,
C, 1, 1)) + eps)
        else:
            # while testing, we normalize the data using the pre-computed mean and
            variance
            X_hat = (X - _BN_MOVING_MEANS[scope_name].reshape((1, C, 1, 1))) * 1.0 \
            / nd.sqrt(_BN_MOVING_VARS[scope_name].reshape((1, C, 1, 1)) + eps)
        # scale and shift
        out = gamma.reshape((1, C, 1, 1)) * X_hat + beta.reshape((1, C, 1, 1))

    #####
    # to keep the moving statistics
    #####

    # init the attributes
    try: # to access them
        _BN_MOVING_MEANS, _BN_MOVING_VARS
    except: # error, create them
        _BN_MOVING_MEANS, _BN_MOVING_VARS = {}, {}

```

```

# store the moving statistics by their scope_names, inplace
if scope_name not in _BN_MOVING_MEANS:
    _BN_MOVING_MEANS[scope_name] = mean
else:
    _BN_MOVING_MEANS[scope_name] = _BN_MOVING_MEANS[scope_name] * momentum + mean *
(1.0 - momentum)
if scope_name not in _BN_MOVING_VARS:
    _BN_MOVING_VARS[scope_name] = variance
else:
    _BN_MOVING_VARS[scope_name] = _BN_MOVING_VARS[scope_name] * momentum + variance *
(1.0 - momentum)

#####
# debug info
#####
if debug:
    print('== info start ==')
    print('scope_name = {}'.format(scope_name))
    print('mean = {}'.format(mean))
    print('var = {}'.format(variance))
    print('_BN_MOVING_MEANS = {}'.format(_BN_MOVING_MEANS[scope_name]))
    print('_BN_MOVING_VARS = {}'.format(_BN_MOVING_VARS[scope_name]))
    print('output = {}'.format(out))
    print('== info end ==')

#####
# return
#####
return out

```

## Parameters and gradients

```

In [9]: #####
# Set the scale for weight initialization and choose
# the number of hidden units in the fully-connected layer
#####
weight_scale = .01
num_fc = 128

W1 = nd.random_normal(shape=(20, 1, 3,3), scale=weight_scale, ctx=ctx)
b1 = nd.random_normal(shape=20, scale=weight_scale, ctx=ctx)

gamma1 = nd.random_normal(shape=20, loc=1, scale=weight_scale, ctx=ctx)
beta1 = nd.random_normal(shape=20, scale=weight_scale, ctx=ctx)

W2 = nd.random_normal(shape=(50, 20, 5, 5), scale=weight_scale, ctx=ctx)
b2 = nd.random_normal(shape=50, scale=weight_scale, ctx=ctx)

gamma2 = nd.random_normal(shape=50, loc=1, scale=weight_scale, ctx=ctx)
beta2 = nd.random_normal(shape=50, scale=weight_scale, ctx=ctx)

W3 = nd.random_normal(shape=(800, num_fc), scale=weight_scale, ctx=ctx)
b3 = nd.random_normal(shape=num_fc, scale=weight_scale, ctx=ctx)

gamma3 = nd.random_normal(shape=num_fc, loc=1, scale=weight_scale, ctx=ctx)
beta3 = nd.random_normal(shape=num_fc, scale=weight_scale, ctx=ctx)

W4 = nd.random_normal(shape=(num_fc, num_outputs), scale=weight_scale, ctx=ctx)
b4 = nd.random_normal(shape=10, scale=weight_scale, ctx=ctx)

params = [W1, b1, gamma1, beta1, W2, b2, gamma2, beta2, W3, b3, gamma3, beta3, W4, b4]

```

```

In [10]: for param in params:
          param.attach_grad()

```

# Activation functions

```
In [11]: def relu(X):  
         return nd.maximum(X, 0)
```

## Softmax output

```
In [12]: def softmax(y_linear):  
         exp = nd.exp(y_linear-nd.max(y_linear))  
         partition = nd.nansum(exp, axis=0, exclude=True).reshape((-1,1))  
         return exp / partition
```

## The *softmax* cross-entropy loss function

```
In [13]: def softmax_cross_entropy(yhat_linear, y):  
         return - nd.nansum(y * nd.log_softmax(yhat_linear), axis=0, exclude=True)
```

## Define the model

We insert the BN layer right after each linear layer.

```
In [14]: def net(X, is_training = True, debug=False):  
         #####  
         # Define the computation of the first convolutional layer  
         #####  
         h1_conv = nd.Convolution(data=X, weight=W1, bias=b1, kernel=(3,3), num_filter=20)  
         h1_normed = batch_norm(h1_conv, gamma1, beta1, scope_name='bn1',  
is_training=is_training)  
         h1_activation = relu(h1_normed)  
         h1 = nd.Pooling(data=h1_activation, pool_type="avg", kernel=(2,2), stride=(2,2))  
         if debug:  
             print("h1 shape: %s" % (np.array(h1.shape)))  
  
         #####  
         # Define the computation of the second convolutional layer  
         #####  
         h2_conv = nd.Convolution(data=h1, weight=W2, bias=b2, kernel=(5,5), num_filter=50)  
         h2_normed = batch_norm(h2_conv, gamma2, beta2, scope_name='bn2',  
is_training=is_training)  
         h2_activation = relu(h2_normed)  
         h2 = nd.Pooling(data=h2_activation, pool_type="avg", kernel=(2,2), stride=(2,2))  
         if debug:  
             print("h2 shape: %s" % (np.array(h2.shape)))  
  
         #####  
         # Flattening h2 so that we can feed it into a fully-connected layer  
         #####  
         h2 = nd.flatten(h2)  
         if debug:  
             print("Flat h2 shape: %s" % (np.array(h2.shape)))  
  
         #####  
         # Define the computation of the third (fully-connected) layer  
         #####  
         h3_linear = nd.dot(h2, W3) + b3  
         h3_normed = batch_norm(h3_linear, gamma3, beta3, scope_name='bn3',  
is_training=is_training)
```

```

h3 = relu(h3_normed)
if debug:
    print("h3 shape: %s" % (np.array(h3.shape)))

#####
# Define the computation of the output layer
#####
yhat_linear = nd.dot(h3, W4) + b4
if debug:
    print("yhat_linear shape: %s" % (np.array(yhat_linear.shape)))

return yhat_linear

```

## Test run

Can data be passed into the `net()` ?

```

In [15]: for data, _ in train_data:
        data = data.as_in_context(ctx)
        break

```

```

In [16]: output = net(data, is_training=True, debug=True)

```

```

h1 shape: [64 20 13 13]
h2 shape: [64 50 4 4]
Flat h2 shape: [ 64 800]
h3 shape: [ 64 128]
yhat_linear shape: [64 10]

```

## Optimizer

```

In [17]: def SGD(params, lr):
        for param in params:
            param[:] = param - lr * param.grad

```

## Evaluation metric

```

In [18]: def evaluate_accuracy(data_iterator, net):
        numerator = 0.
        denominator = 0.
        for i, (data, label) in enumerate(data_iterator):
            data = data.as_in_context(ctx)
            label = label.as_in_context(ctx)
            label_one_hot = nd.one_hot(label, 10)
            output = net(data, is_training=False) # attention here!
            predictions = nd.argmax(output, axis=1)
            numerator += nd.sum(predictions == label)
            denominator += data.shape[0]
        return (numerator / denominator).asscalar()

```

## Execute the training loop

Note: you may want to use a gpu to run the code below. (And remember to set the

`ctx = mx.gpu()` accordingly in the very beginning of this article.)

```
In [19]: epochs = 1
moving_loss = 0.
learning_rate = .001

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx)
        label = label.as_in_context(ctx)
        label_one_hot = nd.one_hot(label, num_outputs)
        with autograd.record():
            # we are in training process,
            # so we normalize the data using batch mean and variance
            output = net(data, is_training=True)
            loss = softmax_cross_entropy(output, label_one_hot)
        loss.backward()
        SGD(params, learning_rate)

        #####
        # Keep a moving average of the losses
        #####
        if i == 0:
            moving_loss = nd.mean(loss).asscalar()
        else:
            moving_loss = .99 * moving_loss + .01 * nd.mean(loss).asscalar()

    test_accuracy = evaluate_accuracy(test_data, net)
    train_accuracy = evaluate_accuracy(train_data, net)
    print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, moving_loss,
train_accuracy, test_accuracy))
```

Epoch 0. Loss: 0.0563528287594, Train\_acc 0.989017, Test\_acc 0.9874

## Next

[Batch normalization with gluon](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Batch Normalization in `gluon`

In the preceding section, [we implemented batch normalization ourselves](#) using `NDArray` and `autograd`. As with most commonly used neural network layers, `Gluon` has batch normalization predefined, so this section is going to be straightforward.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
from mxnet import gluon
import numpy as np
mx.random.seed(1)
ctx = mx.cpu()
```

## The MNIST dataset

```
In [2]: batch_size = 64
num_inputs = 784
num_outputs = 10
def transform(data, label):
    return nd.transpose(data.astype(np.float32), (2,0,1))/255, label.astype(np.float32)
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
    transform=transform),
    batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
    transform=transform),
    batch_size, shuffle=False)
```

## Define a CNN with Batch Normalization

To add batchnormalization to a `gluon` model defined with `Sequential`, we only need to add a few lines. Specifically, we just insert `BatchNorm` layers before the applying the ReLU activations.

```
In [3]: num_fc = 512
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Conv2D(channels=20, kernel_size=5))
    net.add(gluon.nn.BatchNorm(axis=1, center=True, scale=True))
    net.add(gluon.nn.Activation(activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))

    net.add(gluon.nn.Conv2D(channels=50, kernel_size=5))
    net.add(gluon.nn.BatchNorm(axis=1, center=True, scale=True))
    net.add(gluon.nn.Activation(activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))

    # The Flatten Layer collapses all axis, except the first one, into one axis.
    net.add(gluon.nn.Flatten())
```



```

net.add(gluon.nn.Dense(num_fc))
net.add(gluon.nn.BatchNorm(axis=1, center=True, scale=True))
net.add(gluon.nn.Activation(activation='relu'))

net.add(gluon.nn.Dense(num_outputs))

```

## Parameter initialization

```
In [4]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

## Softmax cross-entropy Loss

```
In [5]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Optimizer

```
In [6]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .1})
```

## Write evaluation loop to calculate accuracy

```
In [7]: def evaluate_accuracy(data_iterator, net):
    acc = mx.metric.Accuracy()
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx)
        label = label.as_in_context(ctx)
        output = net(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    return acc.get()[1]
```

## Training Loop

```
In [8]: epochs = 1
smoothing_constant = .01

for e in range(epochs):
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx)
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output, label)
        loss.backward()
        trainer.step(data.shape[0])

        #####
        # Keep a moving average of the losses
        #####
        curr_loss = nd.mean(loss).asscalar()
        moving_loss = (curr_loss if ((i == 0) and (e == 0))
                       else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
                       * curr_loss)
```

```
test_accuracy = evaluate_accuracy(test_data, net)
train_accuracy = evaluate_accuracy(train_data, net)
print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" % (e, moving_loss,
train_accuracy, test_accuracy))
```

Epoch 0. Loss: 0.0453136331317, Train\_acc 0.976166666667, Test\_acc 0.973

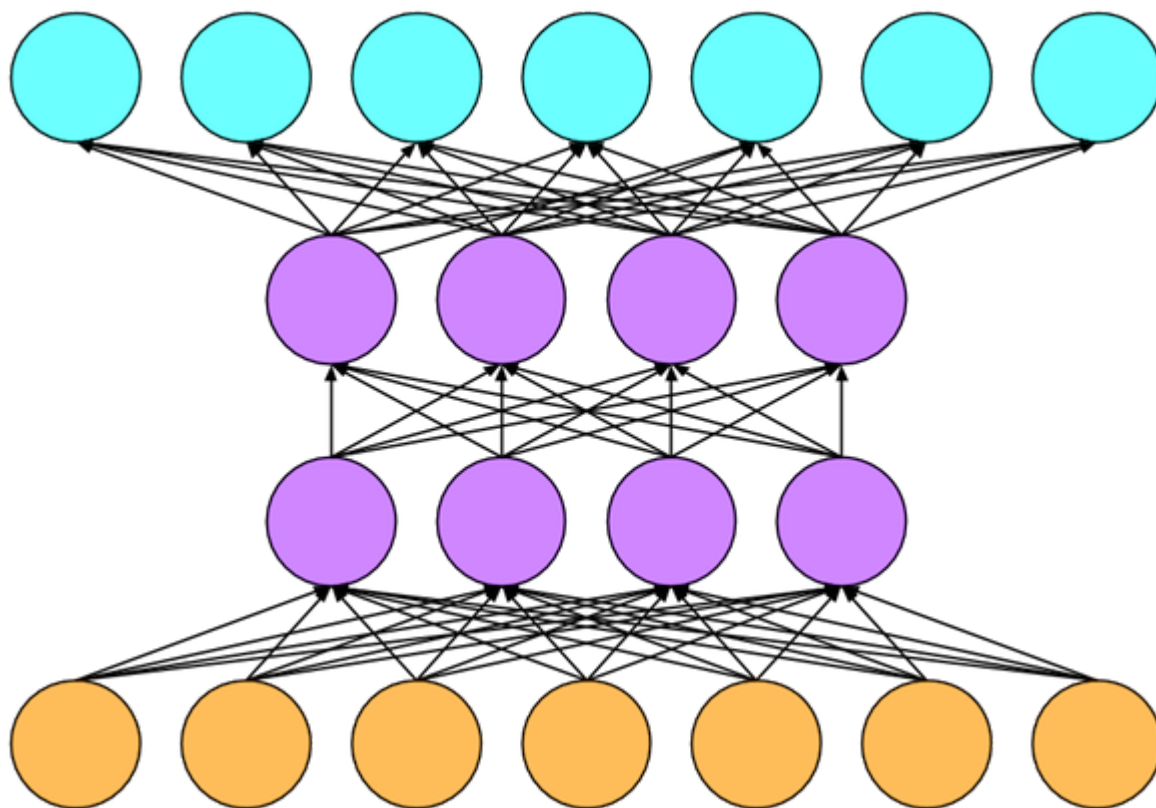
## Next

[Introduction to recurrent neural networks](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Recurrent Neural Networks (RNNs) for Language Modeling

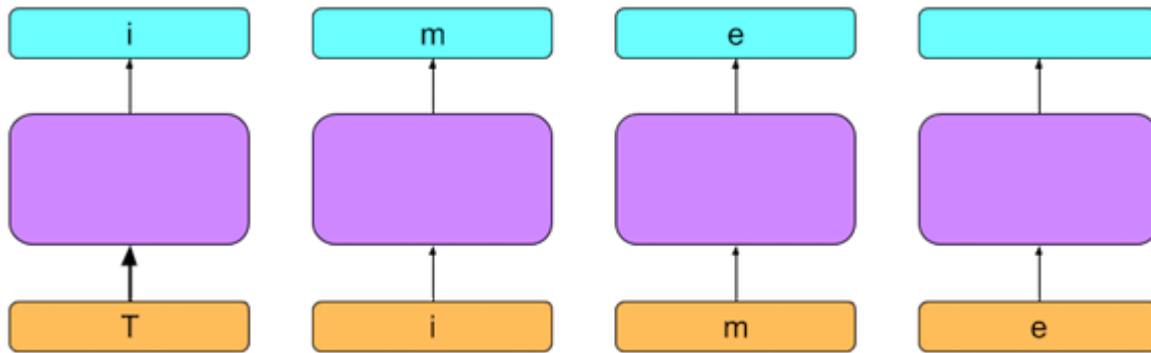
In previous tutorials, we worked with *feedforward* neural networks. They're called feedforward networks because each layer feeds into the next layer in a chain connecting the inputs to the outputs.



At each iteration  $t$ , we feed in a new example  $x_t$ , by setting the values of the input nodes (orange). We then *feed the activation forward* by successively calculating the activations of each higher layer in the network. Finally, we read the outputs from the topmost layer.

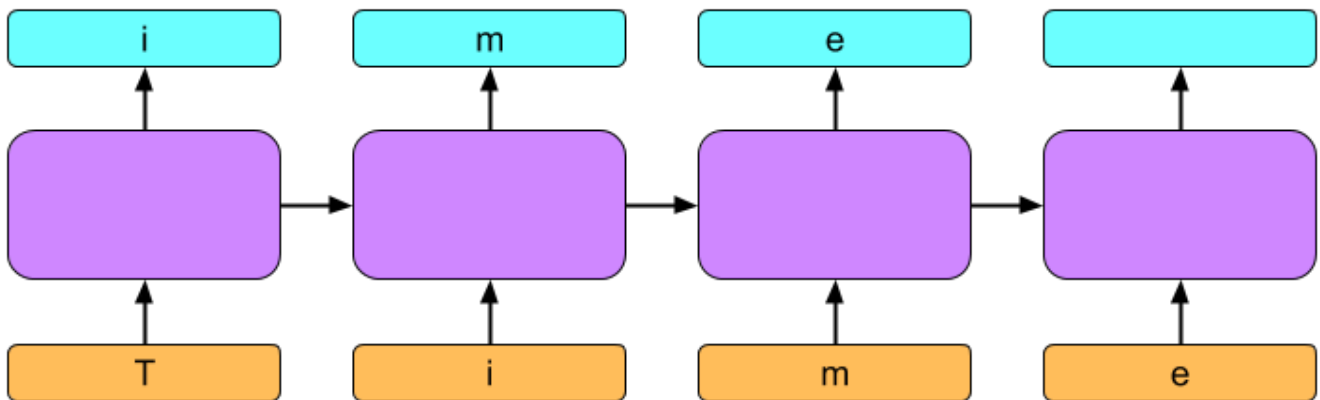
So when we feed the next example  $x_{t+1}$ , we overwrite all of the previous activations. If consecutive inputs to our network have no special relationship to each other (say, images uploaded by unrelated users), then this is perfectly acceptable behavior. But what if our inputs exhibit a sequential relationship?

Say for example that you want to predict the next character in a string of text. We might decide to feed each character into the neural network with the goal of predicting the succeeding character.

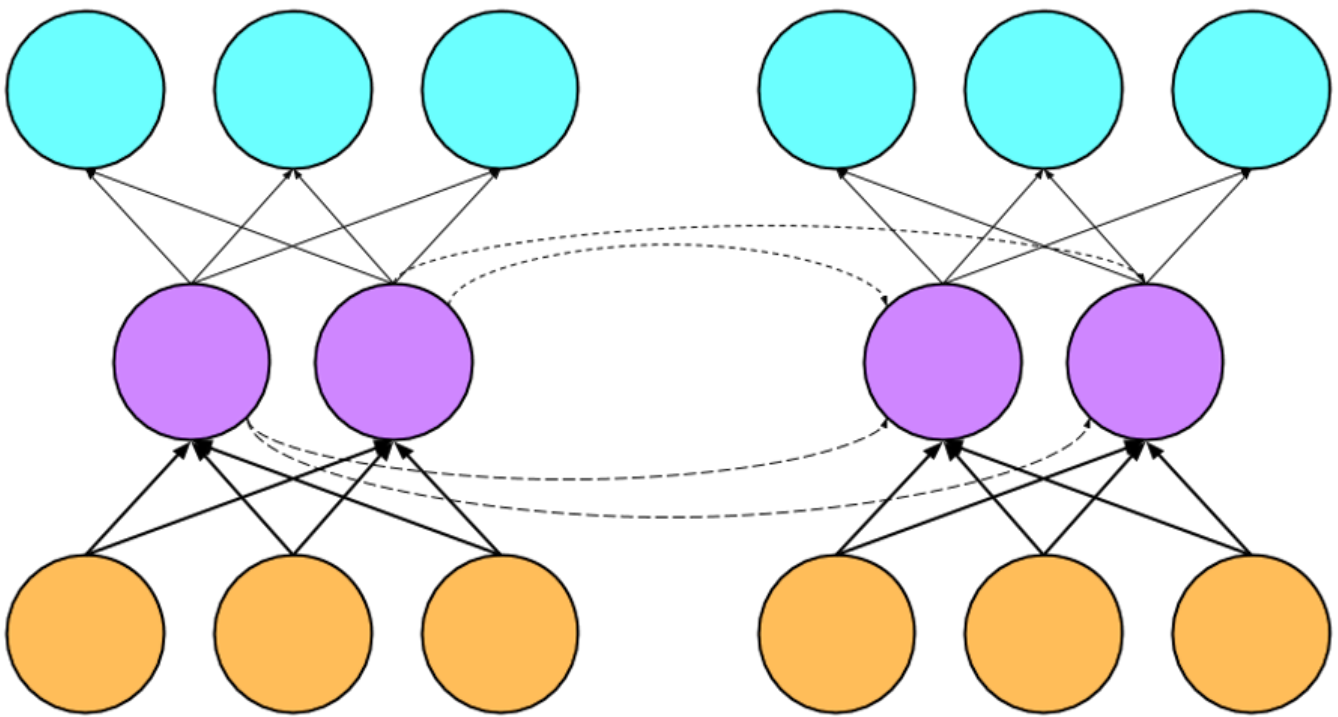


In the above example, the neural network forgets the previous context every time you feed a new input. How is the neural network supposed to know that “e” is followed by a space? It’s hard to see why that should be so probable if you didn’t know that the “e” was the final letter in the word “Time”.

Recurrent neural networks provide a slick way to incorporate sequential structure. At each time step  $t$ , each hidden layer  $h_t$  (typically) will receive input from both the current input  $x_t$  and from *that same hidden layer* at the previous time step  $h_{t-1}$



Now, when our net is trying to predict what comes after the “e” in time, it has access to its previous *beliefs*, and by extension, the entire history of inputs. Zooming back in to see how the nodes in a basic RNN are connected, you’ll see that each node in the hidden layer is connected to each node at the hidden layer at the next time step:



Even though the neural network contains loops (the hidden layer is connected to itself), because this connection spans a time step our network is still technically a feedforward network. Thus we can still train by backpropagation just as we normally would with an MLP. Typically the loss function will be an average of the losses at each time step.

In this tutorial, we're going to roll up our sleeves and write a simple RNN in MXNet using nothing but `mxnet.ndarray` and `mxnet.autograd`. In practice, unless you're trying to develop fundamentally new recurrent layers, you'll want to use the prebuilt layers that call down to extremely optimized primitives. You'll also want to rely on some pre-built batching code because batching sequences can be a pain. But we think in general, if you're going to work with this stuff, and have a modicum of self respect, you'll want to implement from scratch and understand how it works at a reasonably low level.

Let's go ahead and import our dependencies and specify our context. If you've been following along without a GPU until now, this might be where you'll want to get your hands on some faster hardware. GPU instances are available by the hour through Amazon Web Services. A single GPU via a [p2 instance](#) (NVIDIA K80s) or even an older g2 instance will be perfectly adequate for this tutorial.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
import numpy as np
mx.random.seed(1)
ctx = mx.gpu(0)
```

## Dataset: "The Time Machine"

Now mess with some data. I grabbed a copy of the `Time Machine`, mostly because it's available freely thanks to the good people at [Project Gutenberg](#) and a lot of people are tired of seeing RNNs generate Shakespeare. In case you prefer torturing Shakespeare to torturing H.G. Wells, I've also included Andrej Karpathy's `tinyshakespeare.txt` in the data folder. Let's get started by reading in the data.

```
In [2]: with open("../data/nlp/timemachine.txt") as f:
        time_machine = f.read()
```

And you'll probably want to get a taste for what the text looks like.

```
In [3]: print(time_machine[0:500])
```

Project Gutenberg's The Time Machine, by H. G. (Herbert George) Wells

This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at [www.gutenberg.net](http://www.gutenberg.net)

Title: The Time Machine

Author: H. G. (Herbert George) Wells

Release Date: October 2, 2004 [EBook #35]  
[Last updated: October 3, 2014]

Language: English

\*\*\* START OF THIS PR

## Tidying up

I went through and discovered that the last 38083 characters consist entirely of legalese from the Gutenberg gang. So let's chop that off lest our language model learn to generate such boring drivel.

```
In [4]: print(time_machine[-38075:-37500])
        time_machine = time_machine[:-38083]
```

End of Project Gutenberg's The Time Machine, by H. G. (Herbert George) Wells

\*\*\* END OF THIS PROJECT GUTENBERG EBOOK THE TIME MACHINE \*\*\*

\*\*\*\*\* This file should be named 35.txt or 35.zip \*\*\*\*\*

This and all associated files of various formats will be found in:  
<http://www.gutenberg.net/3/35/>

Updated editions will replace the previous one--the old editions will be renamed.

Creating the works from public domain print editions means that no

## Numerical representations of characters

When we create numerical representations of characters, we'll use one-hot representations. A one-hot is a vector that takes value 1 in the index corresponding to a character, and 0 elsewhere. Because this vector is as long as the vocab, let's get a definitive list of characters in this dataset so that our representation is not longer than necessary.

```
In [5]: character_list = list(set(time_machine))
vocab_size = len(character_list)
print(character_list)
print("Length of vocab: %s" % vocab_size)

['\n', '!', ' ', '#', '"', "'", ')', '(', '*', '-', ',', '.', '1', '0', '3', '2', '5',
'4', '9', '8', ';', ':', '?', 'A', 'C', 'B', 'E', 'D', 'G', 'F', 'I', 'H', 'K', 'J', 'M',
'L', 'O', 'N', 'Q', 'P', 'S', 'R', 'U', 'T', 'W', 'V', 'Y', 'X', '[', ']', '_', 'a', 'c',
'b', 'e', 'd', 'g', 'f', 'i', 'h', 'k', 'j', 'm', 'l', 'o', 'n', 'q', 'p', 's', 'r', 'u',
't', 'w', 'v', 'y', 'x', 'z']
Length of vocab: 77
```

We'll often want to access the index corresponding to each character quickly so let's store this as a dictionary.

```
In [6]: character_dict = {}
for e, char in enumerate(character_list):
    character_dict[char] = e
print(character_dict)

{'\n': 0, '!': 1, ' ': 2, '#': 3, '"': 4, "'": 5, ')': 6, '(': 7, '*': 8, '-': 9, ',': 10,
'.': 11, '1': 12, '0': 13, '3': 14, '2': 15, '5': 16, '4': 17, '9': 18, '8': 19, ';': 20,
':': 21, '?': 22, 'A': 23, 'C': 24, 'B': 25, 'E': 26, 'D': 27, 'G': 28, 'F': 29, 'I': 30,
'H': 31, 'K': 32, 'J': 33, 'M': 34, 'L': 35, 'O': 36, 'N': 37, 'Q': 38, 'P': 39, 'S': 40,
'R': 41, 'U': 42, 'T': 43, 'W': 44, 'V': 45, 'Y': 46, 'X': 47, '[': 48, ']': 49, '_': 50,
'a': 51, 'c': 52, 'b': 53, 'e': 54, 'd': 55, 'g': 56, 'f': 57, 'i': 58, 'h': 59, 'k': 60,
'j': 61, 'm': 62, 'l': 63, 'o': 64, 'n': 65, 'q': 66, 'p': 67, 's': 68, 'r': 69, 'u': 70,
't': 71, 'w': 72, 'v': 73, 'y': 74, 'x': 75, 'z': 76}
```

```
In [7]: time_numerical = [character_dict[char] for char in time_machine]
```

```
In [8]: #####
# Check that the length is right
#####
print(len(time_numerical))

#####
# Check that the format looks right
#####
print(time_numerical[:20])

#####
# Convert back to text
#####
print("".join([character_list[idx] for idx in time_numerical[:39]]))
```

```
179533
[39, 69, 64, 61, 54, 52, 71, 2, 28, 70, 71, 54, 65, 53, 54, 69, 56, 5, 68, 2]
Project Gutenberg's The Time Machine, b
```

# One-hot representations

We can use NDAarray's `one_hot()` operation to render a one-hot representation of each character. But frack it, since this is the from scratch tutorial, let's write this ourselves.

```
In [9]: def one_hots(numerical_list, vocab_size=vocab_size):
        result = nd.zeros((len(numerical_list), vocab_size), ctx=ctx)
        for i, idx in enumerate(numerical_list):
            result[i, idx] = 1.0
        return result
```

```
In [10]: print(one_hots(time_numerical[:2]))
```

```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.
  0.  0.  0.  0.  0.]]
<NDAarray 2x77 @gpu(0)>
```

That looks about right. Now let's write a function to convert our one-hots back to readable text.

```
In [11]: def textify(embedding):
        result = ""
        indices = nd.argmax(embedding, axis=1).asnumpy()
        for idx in indices:
            result += character_list[int(idx)]
        return result
```

```
In [12]: textify(one_hots(time_numerical[0:40]))
```

```
Out[12]: "Project Gutenberg's The Time Machine, by"
```

## Preparing the data for training

Great, it's not the most efficient implementation, but we know how it works. So we're already doing better than the majority of people with job titles in machine learning. Now, let's chop up our dataset into sequences that we could feed into our model.

You might think we could just feed in the entire dataset as one gigantic input and backpropagate across the entire sequence. When you try to backpropagate across thousands of steps a few things go wrong: (1) The time it takes to compute a single gradient update will be unreasonably long (2) The gradient across thousands of recurrent steps has a tendency to either blow up, causing NaN errors due to losing precision, or to vanish.



Thus we're going to look at feeding in our data in reasonably short sequences. Note that this home-brew version is pretty slow; if you're still running on a CPU, this is the right time to make dinner.

```
In [13]: seq_length = 64
         # -1 here so we have enough characters for labels later
         num_samples = (len(time_numerical) - 1) // seq_length
         dataset = one_hots(time_numerical[:seq_length*num_samples]).reshape((num_samples,
         seq_length, vocab_size))
         textify(dataset[0])
```

```
Out[13]: "Project Gutenberg's The Time Machine, by H. G. (Herbert George) "
```

Now that we've chopped our dataset into sequences of length `seq_length`, at every time step, our input is a single one-hot vector. This means that our computation of the hidden layer would consist of matrix-vector multiplications, which are not especially efficient on GPU. To take advantage of the available computing resources, we'll want to feed through a batch of sequences at the same time. The following code may look tricky but it's just some plumbing to make the data look like this.

```
In [14]: batch_size = 32
```

```
In [15]: print('# of sequences in dataset: ', len(dataset))
         num_batches = len(dataset) // batch_size
         print('# of batches: ', num_batches)
         train_data = dataset[:num_batches*batch_size].reshape((num_batches, batch_size,
         seq_length, vocab_size))
         # swap batch_size and seq_length axis to make later access easier
         train_data = nd.swapaxes(train_data, 1, 2)
         print('Shape of data set: ', train_data.shape)

# of sequences in dataset: 2805
# of batches: 87
Shape of data set: (87L, 64L, 32L, 77L)
```

Let's sanity check that everything went the way we hope. For each `data_row`, the second sequence should follow the first:

```
In [16]: for i in range(3):
         print("***Batch %s:***\n %s \n\n" % (i, textify(train_data[i, :, 0]) +
         textify(train_data[i, :, 1])))

***Batch 0:***
Project Gutenberg's The Time Machine, by H. G. (Herbert George) Wells

This eBook is for the use of anyone anywhere at no cost a

***Batch 1:***
, breadth, and thickness, can a cube have a
real existence.'

'There I object,' said Filby. 'Of course a solid body may exist. A
```

```

***Batch 2:***
mensions
particularly--why not another direction at right angles to the other
three?--and have even tried to construct a Four-Di

```

## Preparing our labels

Now let's repurpose the same batching code to create our label batches

```

In [17]: labels = one_hots(time_numerical[1:seq_length*num_samples+1])
train_label = labels.reshape((num_batches, batch_size, seq_length, vocab_size))
train_label = nd.swapaxes(train_label, 1, 2)
print(train_label.shape)

(87L, 64L, 32L, 77L)

```

## A final sanity check

Remember that our target at every time step is to predict the next character in the sequence. So our labels should look just like our inputs but offset by one character. Let's look at corresponding inputs and outputs to make sure everything lined up as expected.

```

In [18]: print(textify(train_data[0, :, 0]))
print(textify(train_label[0, :, 0]))

Project Gutenberg's The Time Machine, by H. G. (Herbert George)
roject Gutenberg's The Time Machine, by H. G. (Herbert George) W

```

## Recurrent neural networks

[Explain RNN updates]

Recall that the update for an ordinary hidden layer in a neural network with activation function  $\phi$  is given by

$$h = \phi(XW + b)$$

To make this a recurrent neural network, we're simply going to add a weight sum of the previous hidden state  $h_{t-1}$ :

$$h_t = \phi(X_t W_{xh} + h_{t-1} W_{hh} + b_h)$$

Then at every time set  $t$ , we'll calculate the output as:

$$\hat{y}_t = \text{softmax}(h_t W_{hy} + b_y)$$

## Allocate parameters

```
In [19]: num_inputs = 77
num_hidden = 256
num_outputs = 77

#####
# Weights connecting the inputs to the hidden Layer
#####
Wxh = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01

#####
# Recurrent weights connecting the hidden layer across time steps
#####
Whh = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01

#####
# Bias vector for hidden Layer
#####
bh = nd.random_normal(shape=num_hidden, ctx=ctx) * .01

#####
# Weights to the output nodes
#####
Why = nd.random_normal(shape=(num_hidden,num_outputs), ctx=ctx) * .01
by = nd.random_normal(shape=num_outputs, ctx=ctx) * .01

# NOTE: to keep notation consistent,
# we should really use capital letters
# for hidden layers and outputs,
# since we are doing batchwise computations]
```

## Attach the gradients

```
In [20]: params = [Wxh, Whh, bh, Why, by]

for param in params:
    param.attach_grad()
```

## Softmax Activation

```
In [21]: def softmax(y_linear, temperature=1.0):
lin = (y_linear-nd.max(y_linear)) / temperature
exp = nd.exp(lin)
partition =nd.sum(exp, axis=0, exclude=True).reshape((-1,1))
return exp / partition
```

```
In [22]: #####
# With a temperature of 1 (always 1 during training), we get back some set of
probabilities
#####
softmax(nd.array([[1, -1], [-1, 1]]), temperature=1.0)
```

```
Out[22]: [[ 0.88079703  0.11920292]
 [ 0.11920292  0.88079703]]
<NDArray 2x2 @cpu(0)>
```

```
In [23]: #####
# If we set a high temperature, we can get more entropic (*noisier*) probabilities
#####
softmax(nd.array([[1,-1],[-1,1]]), temperature=1000.0)
```

```
Out[23]: [[ 0.50049996  0.49949998]
 [ 0.49949998  0.50049996]]
<NDArray 2x2 @cpu(0)>
```

```
In [24]: #####
# Often we want to sample with low temperatures to produce sharp probabilities
#####
softmax(nd.array([[10,-10],[-10,10]]), temperature=.1)
```

```
Out[24]: [[ 1.  0.]
 [ 0.  1.]]
<NDArray 2x2 @cpu(0)>
```

## Define the model

```
In [25]: def simple_rnn(inputs, state, temperature=1.0):
          outputs = []
          h = state
          for X in inputs:
              h_linear = nd.dot(X, Wxh) + nd.dot(h, Whh) + bh
              h = nd.tanh(h_linear)
              yhat_linear = nd.dot(h, Why) + by
              yhat = softmax(yhat_linear, temperature=temperature)
              outputs.append(yhat)
          return (outputs, h)
```

## Cross-entropy loss function

At every time step our task is to predict the next character, given the string up to that point. This is the familiar multi-task classification that we introduced for handwritten digit classification. Accordingly, we'll rely on the same loss function, cross-entropy.

```
In [26]: # def cross_entropy(yhat, y):
#         return - nd.sum(y * nd.Log(yhat))

def cross_entropy(yhat, y):
    return - nd.mean(nd.sum(y * nd.log(yhat), axis=0, exclude=True))
```

```
In [27]: cross_entropy(nd.array([.2,.5,.3]), nd.array([1.,0,0]))
```

```
Out[27]: [ 0.53647929]
<NDArray 1 @cpu(0)>
```

## Averaging the loss over the sequence

Because the unfolded RNN has multiple outputs (one at every time step) we can calculate a loss at every time step. The weights corresponding to the net at time step  $t$  influence both the loss at time step  $t$  and the loss at time step  $t + 1$ . To combine our losses into a single global loss, we'll take the average of the losses at each time step.

```
In [28]: def average_ce_loss(outputs, labels):
         assert(len(outputs) == len(labels))
         total_loss = 0.
         for (output, label) in zip(outputs, labels):
             total_loss = total_loss + cross_entropy(output, label)
         return total_loss / len(outputs)
```

## Optimizer

```
In [29]: def SGD(params, lr):
         for param in params:
             param[:] = param - lr * param.grad
```

## Generating text by sampling

We have now defined a model that takes a sequence of real inputs from our training data and tries to predict the next character at every time step. You might wonder, what can we do with this model? Why should I care about predicting the next character in a sequence of text?

This capability is exciting because given such a model, we can now generate strings of plausible text. The generation procedure goes as follows. Say our string begins with the character "T". We can feed the letter "T" and get a conditional probability distribution over the next character  $P(x_2|x_1 = \text{"T"})$ . We can then sample from this distribution, e.g. producing an "i", and then assign  $x_2 = \text{"i"}$ , feeding this to the network at the next time step.

[Add a nice graphic to illustrate sampling]

```
In [30]: def sample(prefix, num_chars, temperature=1.0):
         #####
         # Initialize the string that we'll return to the supplied prefix
         #####
         string = prefix

         #####
         # Prepare the prefix as a sequence of one-hots for ingestion by RNN
         #####
         prefix_numerical = [character_dict[char] for char in prefix]
         input = one_hots(prefix_numerical)

         #####
         # Set the initial state of the hidden representation ($h_0$) to the zero vector
         #####
         sample_state = nd.zeros(shape=(1, num_hidden), ctx=ctx)

         #####
         # For num_chars iterations,
         #     1) feed in the current input
```

```

# 2) sample next character from from output distribution
# 3) add sampled character to the decoded string
# 4) prepare the sampled character as a one_hot (to be the next input)
#####
for i in range(num_chars):
    outputs, sample_state = simple_rnn(input, sample_state, temperature=temperature)
    choice = np.random.choice(77, p=outputs[-1][0].asnumpy())
    string += character_list[choice]
    input = one_hots([choice])
return string

```

```

In [ ]: epochs = 2000
moving_loss = 0.

learning_rate = .5

# state = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
for e in range(epochs):
    #####
    # Attenuate the learning rate by a factor of 2 every 100 epochs.
    #####
    if ((e+1) % 100 == 0):
        learning_rate = learning_rate / 2.0
    state = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
    for i in range(num_batches):
        data_one_hot = train_data[i]
        label_one_hot = train_label[i]
        with autograd.record():
            outputs, state = simple_rnn(data_one_hot, state)
            loss = average_ce_loss(outputs, label_one_hot)
            loss.backward()
        SGD(params, learning_rate)

    #####
    # Keep a moving average of the losses
    #####
    if (i == 0) and (e == 0):
        moving_loss = np.mean(loss.asnumpy()[0])
    else:
        moving_loss = .99 * moving_loss + .01 * np.mean(loss.asnumpy()[0])

    print("Epoch %s. Loss: %s" % (e, moving_loss))
    print(sample("The Time Ma", 1024, temperature=.1))
    print(sample("The Medical Man rose, came to the lamp,", 1024, temperature=.1))

```

## Conclusions

Once you start running this code, it will spit out a sample at the end of each epoch. I'll leave this output cell blank so you don't see megabytes of text, but here are some patterns that I observed when I ran this code.

The network seems to first work out patterns with no sequential relationship and then slowly incorporates longer and longer windows of context. After just 1 epoch, my RNN generated this:

e e ee e eee e ee e ee e ee e ee  
e e e e e e ee e aee e e ee e e ee ee  
e ee e e e e e ete e e e e ee n ee ee e e  
e e e e ee e e e e e e ee ee e e e e  
e e ee ee e e e e t e ee e ee e e ee e e  
e e e e e e e e e e e e ee ee ee a e e  
eee ee e e e e aee e e e e ee e  
e e e e e e e e e e ee e ee e e e e  
e e e e e e e e e e ee e e ee n e ee e e  
e e e e t ee ee ee ee et e e e ee e  
e e e e e e e e e e"

It's learned that spaces and "e"s (to my knowledge, there's no aesthetically pleasing way to spell the plural form of the letter "e") are the most common characters.

A little bit later on it spits out strings like:

[illegible]

At this point it's learned that after the space usually comes a nonspace character, and perhaps that "t" is the most common character to immediately follow a space, "h" to follow a "t" and "e" to follow "th". However it doesn't appear to be looking far enough back to realize that the word "the" should be very unlikely immediately after the word "the"...

By the 175th epoch, the model appears to be putting together a fairly large vocabulary although it puts words together in ways that might be charitably described as “creative”.

the little people had been as I store of the sungher had leartered along the realing of the stars of the little past and stared at the thing that I had the sun had to the stars of the sunghed a stirnt a moment the sun had come and fart as the stars of the sunghed a stirnt a moment the sun had to the was completely and of the little people had been as I stood and all amations of the staring and some of the really

In subsequent tutorials we'll explore sophisticated techniques for evaluating and improving language models. We'll also take a look at some related but more complicated problems like language translations and image captioning.

## Next

[LSTM recurrent neural networks from scratch](#)

For whinges or inquiries, [open an issue on GitHub](#).



# Long short-term memory (LSTM) RNNs

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
import numpy as np
mx.random.seed(1)
ctx = mx.gpu(0)
```

## Dataset: “The Time Machine”

```
In [1]: with open("../data/nlp/timemachine.txt") as f:
        time_machine = f.read()
        time_machine = time_machine[:-38083]
```

## Numerical representations of characters

```
In [3]: character_list = list(set(time_machine))
vocab_size = len(character_list)
character_dict = {}
for e, char in enumerate(character_list):
    character_dict[char] = e
time_numerical = [character_dict[char] for char in time_machine]
```

## One-hot representations

```
In [4]: def one_hots(numerical_list, vocab_size=vocab_size):
        result = nd.zeros((len(numerical_list), vocab_size), ctx=ctx)
        for i, idx in enumerate(numerical_list):
            result[i, idx] = 1.0
        return result
```

```
In [5]: def textify(embedding):
        result = ""
        indices = nd.argmax(embedding, axis=1).asnumpy()
        for idx in indices:
            result += character_list[int(idx)]
        return result
```

## Preparing the data for training

```
In [6]: batch_size = 32
seq_length = 64
# -1 here so we have enough characters for labels later
num_samples = (len(time_numerical) - 1) // seq_length
dataset = one_hots(time_numerical[:seq_length*num_samples]).reshape((num_samples,
```

```
seq_length, vocab_size))
num_batches = len(dataset) // batch_size
train_data = dataset[:num_batches*batch_size].reshape((num_batches, batch_size,
seq_length, vocab_size))
# swap batch_size and seq_length axis to make later access easier
train_data = nd.swapaxes(train_data, 1, 2)
```

## Preparing our labels

```
In [7]: labels = one_hots(time_numerical[1:seq_length*num_samples+1])
train_label = labels.reshape((num_batches, batch_size, seq_length, vocab_size))
train_label = nd.swapaxes(train_label, 1, 2)
```

## Long short-term memory (LSTM) RNNs

An LSTM block has mechanisms to enable “memorizing” information for an extended number of time steps. We use the LSTM block with the following transformations that map inputs to outputs across blocks at consecutive layers and consecutive time steps:

$$g_t = \tanh(X_t W_{xg} + h_{t-1} W_{hg} + b_g),$$

$$i_t = \sigma(X_t W_{xi} + h_{t-1} W_{hi} + b_i),$$

$$f_t = \sigma(X_t W_{xf} + h_{t-1} W_{hf} + b_f),$$

$$o_t = \sigma(X_t W_{xo} + h_{t-1} W_{ho} + b_o),$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t,$$

$$h_t = o_t \odot \tanh(c_t),$$

where  $\odot$  is an element-wise multiplication operator, and for all  $\mathbf{x} = [x_1, x_2, \dots, x_k]^\top \in \mathbb{R}^k$  the two activation functions:

$$\sigma(\mathbf{x}) = \left[ \frac{1}{1 + \exp(-x_1)}, \dots, \frac{1}{1 + \exp(-x_k)} \right]^\top,$$

$$\tanh(\mathbf{x}) = \left[ \frac{1 - \exp(-2x_1)}{1 + \exp(-2x_1)}, \dots, \frac{1 - \exp(-2x_k)}{1 + \exp(-2x_k)} \right]^\top.$$

In the transformations above, the memory cell  $c_t$  stores the “long-term” memory in the vector form. In other words, the information accumulatively captured and encoded until time step  $t$  is stored in  $c_t$  and is only passed along the same layer over different time steps.

Given the inputs  $c_t$  and  $h_t$ , the input gate  $i_t$  and forget gate  $f_t$  will help the memory cell to decide how to overwrite or keep the memory information. The output gate  $o_t$  further lets the LSTM block decide how to retrieve the memory information to generate the current state  $h_t$ .

that is passed to both the next layer of the current time step and the next time step of the current layer. Such decisions are made using the hidden-layer parameters  $W$  and  $b$  with different subscripts: these parameters will be inferred during the training phase by `gluon`.

## Allocate parameters

```
In [8]: num_inputs = vocab_size
        num_hidden = 256
        num_outputs = vocab_size

        #####
        # Weights connecting the inputs to the hidden layer
        #####
        Wxg = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01
        Wxi = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01
        Wxf = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01
        Wxo = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01

        #####
        # Recurrent weights connecting the hidden layer across time steps
        #####
        Whg = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01
        Whi = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01
        Whf = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01
        Who = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01

        #####
        # Bias vector for hidden layer
        #####
        bg = nd.random_normal(shape=num_hidden, ctx=ctx) * .01
        bi = nd.random_normal(shape=num_hidden, ctx=ctx) * .01
        bf = nd.random_normal(shape=num_hidden, ctx=ctx) * .01
        bo = nd.random_normal(shape=num_hidden, ctx=ctx) * .01

        #####
        # Weights to the output nodes
        #####
        Why = nd.random_normal(shape=(num_hidden,num_outputs), ctx=ctx) * .01
        by = nd.random_normal(shape=num_outputs, ctx=ctx) * .01
```

## Attach the gradients

```
In [9]: params = [Wxg, Wxi, Wxf, Wxo, Whg, Whi, Whf, Who, bg, bi, bf, bo, Why, by]

        for param in params:
            param.attach_grad()
```

## Softmax Activation

```
In [10]: def softmax(y_linear, temperature=1.0):
        lin = (y_linear-nd.max(y_linear)) / temperature
        exp = nd.exp(lin)
        partition = nd.sum(exp, axis=0, exclude=True).reshape((-1,1))
        return exp / partition
```

# Define the model

```
In [11]: def lstm_rnn(inputs, h, c, temperature=1.0):
    outputs = []
    for X in inputs:
        g = nd.tanh(nd.dot(X, Wxg) + nd.dot(h, Whg) + bg)
        i = nd.sigmoid(nd.dot(X, Wxi) + nd.dot(h, Whi) + bi)
        f = nd.sigmoid(nd.dot(X, Wxf) + nd.dot(h, Whf) + bf)
        o = nd.sigmoid(nd.dot(X, Wxo) + nd.dot(h, Who) + bo)
        #####
        #
        #####
        c = f * c + i * g
        h = o * nd.tanh(c)
        #####
        #
        #####
        yhat_linear = nd.dot(h, Why) + by
        yhat = softmax(yhat_linear, temperature=temperature)
        outputs.append(yhat)
    return (outputs, h, c)
```

## Cross-entropy loss function

```
In [12]: def cross_entropy(yhat, y):
    return - nd.mean(nd.sum(y * nd.log(yhat), axis=0, exclude=True))
```

## Averaging the loss over the sequence

```
In [13]: def average_ce_loss(outputs, labels):
    assert(len(outputs) == len(labels))
    total_loss = 0.
    for (output, label) in zip(outputs, labels):
        total_loss = total_loss + cross_entropy(output, label)
    return total_loss / len(outputs)
```

## Optimizer

```
In [14]: def SGD(params, lr):
    for param in params:
        param[:] = param - lr * param.grad
```

## Generating text by sampling

```
In [15]: def sample(prefix, num_chars, temperature=1.0):
    #####
    # Initialize the string that we'll return to the supplied prefix
    #####
    string = prefix

    #####
    # Prepare the prefix as a sequence of one-hots for ingestion by RNN
    #####
    prefix_numerical = [character_dict[char] for char in prefix]
```

```

input = one_hots(prefix_numerical)

#####
# Set the initial state of the hidden representation ($h_0$) to the zero vector
#####
h = nd.zeros(shape=(1, num_hidden), ctx=ctx)
c = nd.zeros(shape=(1, num_hidden), ctx=ctx)

#####
# For num_chars iterations,
# 1) feed in the current input
# 2) sample next character from from output distribution
# 3) add sampled character to the decoded string
# 4) prepare the sampled character as a one_hot (to be the next input)
#####
for i in range(num_chars):
    outputs, h, c = lstm_rnn(input, h, c, temperature=temperature)
    choice = np.random.choice(vocab_size, p=outputs[-1][0].asnumpy())
    string += character_list[choice]
    input = one_hots([choice])
return string

```

```

In [ ]: epochs = 2000
        moving_loss = 0.

        learning_rate = 2.0

        # state = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
        for e in range(epochs):
            #####
            # Attenuate the Learning rate by a factor of 2 every 100 epochs.
            #####
            if ((e+1) % 100 == 0):
                learning_rate = learning_rate / 2.0
            h = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
            c = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
            for i in range(num_batches):
                data_one_hot = train_data[i]
                label_one_hot = train_label[i]
                with autograd.record():
                    outputs, h, c = lstm_rnn(data_one_hot, h, c)
                    loss = average_ce_loss(outputs, label_one_hot)
                    loss.backward()
                SGD(params, learning_rate)

            #####
            # Keep a moving average of the losses
            #####
            if (i == 0) and (e == 0):
                moving_loss = nd.mean(loss).asscalar()
            else:
                moving_loss = .99 * moving_loss + .01 * nd.mean(loss).asscalar()

            print("Epoch %s. Loss: %s" % (e, moving_loss))
            print(sample("The Time Ma", 1024, temperature=.1))
            print(sample("The Medical Man rose, came to the lamp,", 1024, temperature=.1))

```

## Conclusions

## Next

Gated recurrent units (GRU) RNNs from scratch

For whinges or inquiries, [open an issue on GitHub](#).



## Gated recurrent unit (GRU) RNNs

This chapter requires some exposition. The GRU updates are fully implemented and the code appears to work properly.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import nd, autograd
import numpy as np
mx.random.seed(1)
ctx = mx.gpu(0)
```

### Dataset: “The Time Machine”

```
In [1]: with open("../data/nlp/timemachine.txt") as f:
        time_machine = f.read()
        time_machine = time_machine[:-38083]
```

### Numerical representations of characters

```
In [3]: character_list = list(set(time_machine))
vocab_size = len(character_list)
character_dict = {}
for e, char in enumerate(character_list):
    character_dict[char] = e
time_numerical = [character_dict[char] for char in time_machine]
```

### One-hot representations

```
In [4]: def one_hots(numerical_list, vocab_size=vocab_size):
        result = nd.zeros((len(numerical_list), vocab_size), ctx=ctx)
        for i, idx in enumerate(numerical_list):
            result[i, idx] = 1.0
        return result
```

```
In [5]: def textify(embedding):
        result = ""
        indices = nd.argmax(embedding, axis=1).asnumpy()
        for idx in indices:
            result += character_list[int(idx)]
        return result
```

### Preparing the data for training

```
In [6]: batch_size = 32
        seq_length = 64
        # -1 here so we have enough characters for labels later
        num_samples = (len(time_numerical) - 1) // seq_length
        dataset = one_hots(time_numerical[:seq_length*num_samples]).reshape((num_samples,
        seq_length, vocab_size))
        num_batches = len(dataset) // batch_size
        train_data = dataset[:num_batches*batch_size].reshape((num_batches, batch_size,
        seq_length, vocab_size))
        # swap batch_size and seq_length axis to make later access easier
        train_data = nd.swapaxes(train_data, 1, 2)
```

## Preparing our labels

```
In [7]: labels = one_hots(time_numerical[1:seq_length*num_samples+1])
        train_label = labels.reshape((num_batches, batch_size, seq_length, vocab_size))
        train_label = nd.swapaxes(train_label, 1, 2)
```

## Gated recurrent units (GRU) RNNs

Similar to LSTM blocks, the GRU also has mechanisms to enable “memorizing” information for an extended number of time steps. However, it does so in a more expedient way:

- We no longer keep a separate memory cell  $c_t$ . Instead,  $h_{t-1}$  is added to a “new content” version of itself to give  $h_t$ .
- The “new content” version is given by  $g_t = \tanh(X_t W_{xh} + (r_t \odot h_{t-1}) W_{hh} + b_h)$ , and is analogous to  $g_t$  in the LSTM tutorial.
- Here, there is a reset gate  $r_t$  which moderates the impact of  $h_{t-1}$  on the “new content” version.
- The input gate  $i_t$  and forget gate  $f_t$  are replaced by a single update gate  $z_t$ , which weighs the old and new content via  $z_t$  and  $(1 - z_t)$  respectively.
- There is no output gate  $o_t$ ; the weighted sum is what becomes  $h_t$ .

We use the GRU block with the following transformations that map inputs to outputs across blocks at consecutive layers and consecutive time steps:

$$\begin{aligned}
 z_t &= \sigma(X_t W_{xz} + h_{t-1} W_{hz} + b_z), \\
 r_t &= \sigma(X_t W_{xr} + h_{t-1} W_{hr} + b_r), \\
 g_t &= \tanh(X_t W_{xh} + (r_t \odot h_{t-1}) W_{hh} + b_h), \\
 h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot g_t,
 \end{aligned}$$

where  $\sigma$  and  $\tanh$  are as before in the LSTM case.



Empirically, GRUs have similar performance to LSTMs, while requiring less parameters and forgoing an internal time state. Intuitively, GRUs have enough gates/state for long-term retention, but not too much, so that training and convergence remain fast and convex. See the work of Chung et al. [2014] (<https://arxiv.org/abs/1412.3555>).

## Allocate parameters

```
In [8]: num_inputs = vocab_size
        num_hidden = 256
        num_outputs = vocab_size

        #####
        # Weights connecting the inputs to the hidden layer
        #####
        Wxz = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01
        Wxr = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01
        Wxh = nd.random_normal(shape=(num_inputs,num_hidden), ctx=ctx) * .01

        #####
        # Recurrent weights connecting the hidden layer across time steps
        #####
        Whz = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01
        Whr = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01
        Whh = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx)* .01

        #####
        # Bias vector for hidden layer
        #####
        bz = nd.random_normal(shape=num_hidden, ctx=ctx) * .01
        br = nd.random_normal(shape=num_hidden, ctx=ctx) * .01
        bh = nd.random_normal(shape=num_hidden, ctx=ctx) * .01

        #####
        # Weights to the output nodes
        #####
        Why = nd.random_normal(shape=(num_hidden,num_outputs), ctx=ctx) * .01
        by = nd.random_normal(shape=num_outputs, ctx=ctx) * .01
```

## Attach the gradients

```
In [9]: params = [Wxz, Wxr, Wxh, Whz, Whr, Whh, bz, br, bh, Why, by]

        for param in params:
            param.attach_grad()
```

## Softmax Activation

```
In [10]: def softmax(y_linear, temperature=1.0):
        lin = (y_linear-nd.max(y_linear)) / temperature
        exp = nd.exp(lin)
        partition = nd.sum(exp, axis=0, exclude=True).reshape((-1,1))
        return exp / partition
```

## Define the model

```
In [11]: def gru_rnn(inputs, h, temperature=1.0):
          outputs = []
          for X in inputs:
              z = nd.sigmoid(nd.dot(X, Wxz) + nd.dot(h, Whz) + bz)
              r = nd.sigmoid(nd.dot(X, Wxr) + nd.dot(h, Whr) + br)
              g = nd.tanh(nd.dot(X, Wxh) + nd.dot(r * h, Whh) + bh)
              h = z * h + (1 - z) * g

              yhat_linear = nd.dot(h, Why) + by
              yhat = softmax(yhat_linear, temperature=temperature)
              outputs.append(yhat)
          return (outputs, h)
```

## Cross-entropy loss function

```
In [12]: def cross_entropy(yhat, y):
          return - nd.mean(nd.sum(y * nd.log(yhat), axis=0, exclude=True))
```

## Averaging the loss over the sequence

```
In [13]: def average_ce_loss(outputs, labels):
          assert(len(outputs) == len(labels))
          total_loss = nd.array([0.], ctx=ctx)
          for (output, label) in zip(outputs, labels):
              total_loss = total_loss + cross_entropy(output, label)
          return total_loss / len(outputs)
```

## Optimizer

```
In [14]: def SGD(params, lr):
          for param in params:
              param[:] = param - lr * param.grad
```

## Generating text by sampling

```
In [15]: def sample(prefix, num_chars, temperature=1.0):
          #####
          # Initialize the string that we'll return to the supplied prefix
          #####
          string = prefix

          #####
          # Prepare the prefix as a sequence of one-hots for ingestion by RNN
          #####
          prefix_numerical = [character_dict[char] for char in prefix]
          input = one_hots(prefix_numerical)

          #####
          # Set the initial state of the hidden representation ($h_0$) to the zero vector
          #####
          h = nd.zeros(shape=(1, num_hidden), ctx=ctx)
          c = nd.zeros(shape=(1, num_hidden), ctx=ctx)

          #####
          # For num_chars iterations,
          #     1) feed in the current input
          #     2) sample next character from output distribution
```

```

# 3) add sampled character to the decoded string
# 4) prepare the sampled character as a one_hot (to be the next input)
#####
for i in range(num_chars):
    outputs, h = gru_rnn(input, h, temperature=temperature)
    choice = np.random.choice(vocab_size, p=outputs[-1][0].asnumpy())
    string += character_list[choice]
    input = one_hots([choice])
return string

```

```

In [ ]: epochs = 2000
moving_loss = 0.

learning_rate = 2.0

# state = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
for e in range(epochs):
    #####
    # Attenuate the learning rate by a factor of 2 every 100 epochs.
    #####
    if ((e+1) % 100 == 0):
        learning_rate = learning_rate / 2.0
    h = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
    for i in range(num_batches):
        data_one_hot = train_data[i]
        label_one_hot = train_label[i]
        with autograd.record():
            outputs, h = gru_rnn(data_one_hot, h)
            loss = average_ce_loss(outputs, label_one_hot)
            loss.backward()
        SGD(params, learning_rate)

    #####
    # Keep a moving average of the losses
    #####
    if (i == 0) and (e == 0):
        moving_loss = nd.mean(loss).asscalar()
    else:
        moving_loss = .99 * moving_loss + .01 * nd.mean(loss).asscalar()

    print("Epoch %s. Loss: %s" % (e, moving_loss))
    print(sample("The Time Ma", 1024, temperature=.1))
    print(sample("The Medical Man rose, came to the lamp,", 1024, temperature=.1))

```

## Conclusions

[Placeholder]

## Next

Simple, LSTM, and GRU RNNs with gluon

For whinges or inquiries, [open an issue on GitHub](#).

# Recurrent Neural Networks with `gluon`

With `gluon`, now we can train the recurrent neural networks (RNNs) more neatly, such as the long short-term memory (LSTM) and the gated recurrent unit (GRU). To demonstrate the end-to-end RNN training and prediction pipeline, we take a classic problem in language modeling as a case study. Specifically, we will show how to predict the distribution of the next word given a sequence of previous words.

## Import packages

To begin with, we need to make the following necessary imports.

```
In [ ]: import math
import os
import time
import numpy as np
import mxnet as mx
from mxnet import gluon, autograd
from mxnet.gluon import nn, rnn
```

## Define classes for indexing words of the input document

In a language modeling problem, we define the following classes to facilitate the routine procedures for loading document data. In the following, the `Dictionary` class is for word indexing: words in the documents can be converted from the string format to the integer format.

In this example, we use consecutive integers to index words of the input document.

```
In [ ]: class Dictionary(object):
    def __init__(self):
        self.word2idx = {}
        self.idx2word = []

    def add_word(self, word):
        if word not in self.word2idx:
            self.idx2word.append(word)
            self.word2idx[word] = len(self.idx2word) - 1
        return self.word2idx[word]

    def __len__(self):
        return len(self.idx2word)
```

The `Dictionary` class is used by the `Corpus` class to index the words of the input document.

```
In [ ]: class Corpus(object):
    def __init__(self, path):
        self.dictionary = Dictionary()
        self.train = self.tokenize(path + 'train.txt')
        self.valid = self.tokenize(path + 'valid.txt')
        self.test = self.tokenize(path + 'test.txt')

    def tokenize(self, path):
        """Tokenizes a text file."""
        assert os.path.exists(path)
        # Add words to the dictionary
        with open(path, 'r') as f:
            tokens = 0
            for line in f:
                words = line.split() + ['<eos>']
                tokens += len(words)
                for word in words:
                    self.dictionary.add_word(word)

        # Tokenize file content
        with open(path, 'r') as f:
            ids = np.zeros((tokens,), dtype='int32')
            token = 0
            for line in f:
                words = line.split() + ['<eos>']
                for word in words:
                    ids[token] = self.dictionary.word2idx[word]
                    token += 1

        return mx.nd.array(ids, dtype='int32')
```

## Provide an exposition of different RNN models with `gluon`

Based on the `gluon.Block` class, we can make different RNN models available with the following single `RNNModel` class.

Users can select their preferred RNN model or compare different RNN models by configuring the argument of the constructor of `RNNModel`. We will show an example following the definition of the `RNNModel` class.

```
In [ ]: class RNNModel(gluon.Block):
    """A model with an encoder, recurrent layer, and a decoder."""

    def __init__(self, mode, vocab_size, num_embed, num_hidden,
                 num_layers, dropout=0.5, tie_weights=False, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        with self.name_scope():
            self.drop = nn.Dropout(dropout)
            self.encoder = nn.Embedding(vocab_size, num_embed,
                                       weight_initializer = mx.init.Uniform(0.1))

            if mode == 'rnn_relu':
                self.rnn = rnn.RNN(num_hidden, num_layers, activation='relu',
                                   dropout=dropout,
                                   input_size=num_embed)
            elif mode == 'rnn_tanh':
                self.rnn = rnn.RNN(num_hidden, num_layers, dropout=dropout,
                                   input_size=num_embed)
            elif mode == 'lstm':
                self.rnn = rnn.LSTM(num_hidden, num_layers, dropout=dropout,
                                   input_size=num_embed)
            elif mode == 'gru':
                self.rnn = rnn.GRU(num_hidden, num_layers, dropout=dropout,
```

```

        input_size=num_embed)
    else:
        raise ValueError("Invalid mode %s. Options are rnn_relu, "
                           "rnn_tanh, lstm, and gru"%mode)
    if tie_weights:
        self.decoder = nn.Dense(vocab_size, in_units = num_hidden,
                                params = self.encoder.params)
    else:
        self.decoder = nn.Dense(vocab_size, in_units = num_hidden)
    self.num_hidden = num_hidden

    def forward(self, inputs, hidden):
        emb = self.drop(self.encoder(inputs))
        output, hidden = self.rnn(emb, hidden)
        output = self.drop(output)
        decoded = self.decoder(output.reshape((-1, self.num_hidden)))
        return decoded, hidden

    def begin_state(self, *args, **kwargs):
        return self.rnn.begin_state(*args, **kwargs)

```

## Select an RNN model and configure parameters

For demonstration purposes, we provide an arbitrary selection of the parameter values. In practice, some parameters should be more fine tuned based on the validation data set.

For instance, to obtain a better performance, as reflected in a lower loss or perplexity, one can set `args_epochs` to a larger value.

In this demonstration, LSTM is the chosen type of RNN. For other RNN options, one can replace the `'lstm'` string to `'rnn_relu'`, `'rnn_tanh'`, or `'gru'` as provided by the aforementioned `gluon.Block` class.

```

In [1]: args_data = '../data/nlp/ptb.'
        args_model = 'rnn_relu'
        args_emsized = 100
        args_nhid = 100
        args_nlayers = 2
        args_lr = 1.0
        args_clip = 0.2
        args_epochs = 1
        args_batch_size = 32
        args_bptt = 5
        args_dropout = 0.2
        args_tied = True
        args_cuda = 'store_true'
        args_log_interval = 500
        args_save = 'model.param'

```

## Load data as batches

We load the document data by leveraging the aforementioned `Corpus` class.

To speed up the subsequent data flow in the RNN model, we pre-process the loaded data as batches. This procedure is defined in the following `batchify` function.

```
In [ ]: context = mx.cpu(0)
        corpus = Corpus(args_data)

        def batchify(data, batch_size):
            """Reshape data into (num_example, batch_size)"""
            nbatch = data.shape[0] // batch_size
            data = data[:nbatch * batch_size]
            data = data.reshape((batch_size, nbatch)).T
            return data

        train_data = batchify(corpus.train, args_batch_size).as_in_context(context)
        val_data = batchify(corpus.valid, args_batch_size).as_in_context(context)
        test_data = batchify(corpus.test, args_batch_size).as_in_context(context)
```

## Build the model

We go on to build the model, initialize model parameters, and configure the optimization algorithms for training the RNN model.

```
In [ ]: ntokens = len(corpus.dictionary)

        model = RNNModel(args_model, ntokens, args_emsize, args_nhid,
                          args_nlayers, args_dropout, args_tied)
        model.collect_params().initialize(mx.init.Xavier(), ctx=context)
        trainer = gluon.Trainer(model.collect_params(), 'sgd',
                                {'learning_rate': args_lr, 'momentum': 0, 'wd': 0})
        loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

## Train the model and evaluate on validation and testing data sets

Now we can define functions for training and evaluating the model. The following are two helper functions that will be used during model training and evaluation.

```
In [ ]: def get_batch(source, i):
        seq_len = min(args_bptt, source.shape[0] - 1 - i)
        data = source[i : i + seq_len]
        target = source[i + 1 : i + 1 + seq_len]
        return data, target.reshape((-1,))

        def detach(hidden):
            if isinstance(hidden, (tuple, list)):
                hidden = [i.detach() for i in hidden]
            else:
                hidden = hidden.detach()
            return hidden
```

The following is the function for model evaluation. It returns the loss of the model prediction. We will discuss the details of the loss measure shortly.

```
In [ ]: def eval(data_source):
        total_L = 0.0
        ntotal = 0
        hidden = model.begin_state(func = mx.nd.zeros, batch_size = args_batch_size,
```

```

ctx=context)
    for i in range(0, data_source.shape[0] - 1, args_bppt):
        data, target = get_batch(data_source, i)
        output, hidden = model(data, hidden)
        L = loss(output, target)
        total_L += mx.nd.sum(L).asscalar()
        ntotal += L.size
    return total_L / ntotal

```

Now we are ready to define the function for training the model. We can monitor the model performance on the training, validation, and testing data sets over iterations.

```

In [ ]: def train():
    best_val = float("Inf")
    for epoch in range(args_epochs):
        total_L = 0.0
        start_time = time.time()
        hidden = model.begin_state(func = mx.nd.zeros, batch_size = args_batch_size, ctx =
context)
        for ibatch, i in enumerate(range(0, train_data.shape[0] - 1, args_bppt)):
            data, target = get_batch(train_data, i)
            hidden = detach(hidden)
            with autograd.record():
                output, hidden = model(data, hidden)
                L = loss(output, target)
                L.backward()

            grads = [i.grad(context) for i in model.collect_params().values()]
            # Here gradient is for the whole batch.
            # So we multiply max_norm by batch_size and bppt size to balance it.
            gluon.utils.clip_global_norm(grads, args_clip * args_bppt * args_batch_size)

            trainer.step(args_batch_size)
            total_L += mx.nd.sum(L).asscalar()

            if ibatch % args_log_interval == 0 and ibatch > 0:
                cur_L = total_L / args_bppt / args_batch_size / args_log_interval
                print('[Epoch %d Batch %d] loss %.2f, perplexity %.2f' % (
                    epoch + 1, ibatch, cur_L, math.exp(cur_L)))
                total_L = 0.0

        val_L = eval(val_data)

        print('[Epoch %d] time cost %.2fs, validation loss %.2f, validation perplexity
%.2f' % (
            epoch + 1, time.time() - start_time, val_L, math.exp(val_L)))

        if val_L < best_val:
            best_val = val_L
            test_L = eval(test_data)
            model.save_params(args_save)
            print('test loss %.2f, test perplexity %.2f' % (test_L, math.exp(test_L)))
        else:
            args_lr = args_lr * 0.25
            trainer._init_optimizer('sgd',
                                   {'learning_rate': args_lr,
                                    'momentum': 0,
                                    'wd': 0})
            model.load_params(args_save, context)

```

Recall that the RNN model training is based on maximization likelihood of observations. For evaluation purposes, we have used the following two measures:



- Loss: the loss function is defined as the average negative log likelihood of the words under prediction:

$$\text{loss} = -\frac{1}{N} \sum_{i=1}^N \log p_{\text{predicted}_i},$$

where  $N$  is the number of predictions and  $p_{\text{predicted}_i}$  the likelihood of observing the next word in the  $i$ -th prediction.

- Perplexity: the average per-word perplexity is  $\exp(\text{loss})$ .

To orient the reader using concrete examples, let us illustrate the idea of the perplexity measure as follows.

- Consider a perfect scenario where the prediction model always predicts the likelihood of the next word correctly. In this case, for every  $i$  we have  $p_{\text{predicted}_i} = 1$ . As a result, the perplexity of a perfect prediction model is always 1.
- Consider a baseline scenario where the prediction model always predicts the likelihood of the next word randomly at uniform among the given word set  $W$ . In this case, for every  $i$  we have  $p_{\text{predicted}_i} = 1/|W|$ . As a result, the perplexity of a uniformly random prediction model is always  $|W|$ .

Therefore, a perplexity value is always between 1 and  $|W|$ . A model with a lower perplexity that is closer to 1 is generally more accurate in prediction.

Now we are ready to train the model and evaluate the model performance on validation and testing data sets.

```
In [ ]: train()  
         model.load_params(args_save, context)  
         test_L = eval(test_data)  
         print('Best test loss %.2f, test perplexity %.2f'%(test_L, math.exp(test_L)))
```

## Next

[Introduction to optimization](#)

For whinges or inquiries, [open an issue on GitHub](#).

```
In [ ]:
```

## Introduction

You might find it weird that we're sticking a chapter on optimization here. If you're following the tutorials in sequence, then you've probably already been optimizing over the parameters of ten or more machine learning models. You might consider yourself an old pro. In this chapter we'll supply some depth to complement your experience.

We need to think seriously about optimization matters for several reasons. First, we want optimizers to be fast. Optimizing complicated models with millions of parameters can take upsettingly long. You might have heard of researchers training deep learning models for many hours, days, or even weeks. They probably weren't exaggerating. Second, optimization is how we choose our parameters. So the performance (e.g. accuracy) of our models depends entirely on the quality of the optimizer.



## Challenges in optimization

The pre-defined loss function in the learning problem is called the objective function for optimization. Conventionally, optimization considers a minimization problem. Any maximization problem can be trivially converted to an equivalent minimization problem by flipping the sign for the objective function. Optimization is worth studying both because it's essential to learning. It's also worth studying because it's an area where progress is being made, and smart choices can

lead to superior performance. In other words, even fixing all the other modeling decisions, figuring out how to optimize the parameters is a formidable challenge. We'll briefly describe some of the issues that make optimization hard, especially for neural networks.

## Local minima

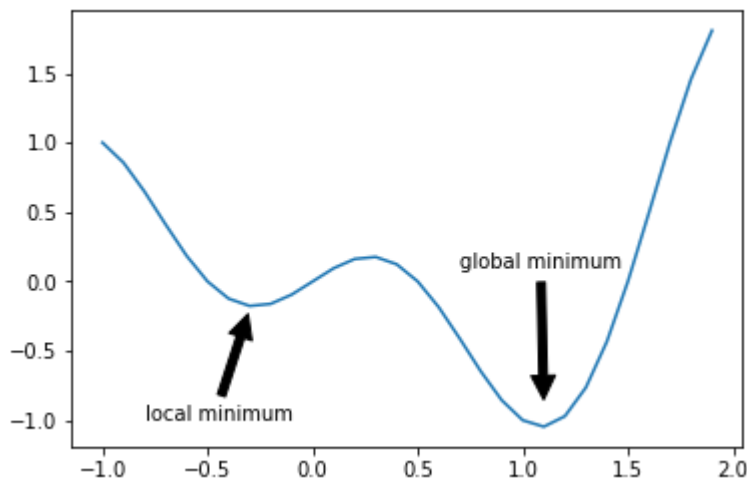
An objective function  $f(x)$  may have a local minimum  $x$ , where  $f(x)$  is smaller at  $x$  than at the neighboring points of  $x$ . If  $f(x)$  is the smallest value that can be obtained in the entire domain of  $x$ ,  $f(x)$  is a global minimum. The following figure demonstrates examples of local and global minima for the function:

$$f(x) = x \cdot \cos(\pi x), \quad -1.0 \leq x \leq 2.0.$$

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x * np.cos(np.pi * x)

x = np.arange(-1.0, 2.0, 0.1)
fig = plt.figure()
subplt = fig.add_subplot(111)
subplt.annotate('local minimum', xy=(-0.3, -0.2), xytext=(-0.8, -1.0),
               arrowprops=dict(facecolor='black', shrink=0.05))
subplt.annotate('global minimum', xy=(1.1, -0.9), xytext=(0.7, 0.1),
               arrowprops=dict(facecolor='black', shrink=0.05))
plt.plot(x, f(x))
plt.show()
```



## Analytic vs approximate solutions

Ideally, we'd find the optimal solution  $x^*$  that globally minimizes an objective function. For instance, the function  $f(x) = x^2$  has a global minimum solution at  $x^* = 0$ . We can obtain this solution analytically. Another way of saying this is that there exists a *closed-form* solution. This just means that we can analyze the equation for the function and produce an exact solution directly. Linear regression, for example, has an analytic solution. To refresh your memory, in linear regression we build a predictor of the form:

$$\hat{\mathbf{y}} = X\mathbf{w}$$

We ignored the intercept term  $b$  here but that can be handled by simply appending a column of all 1s to the design matrix  $X$ .

And we want to solve the following minimization problem

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - X\mathbf{w}\|_2^2$$

As a refresher, that's just the sum of the squared differences between our predictions and the ground truth answers.

$$\sum_{i=1}^n (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

Because we know that this function is quadratic, we know that it has a single critical point where the derivative of the loss with respect to the weights  $\mathbf{w}$  is equal to 0. Moreover, we know that the weights that minimize our loss constitute a critical point. So our solution corresponds to the one setting of the weights that gives derivative of 0. First, let's rewrite our loss function:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = (\mathbf{y} - X\mathbf{w})^T (\mathbf{y} - X\mathbf{w})$$

Now, setting the derivative of our loss to 0 gives the following equation:

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}} = -2(X)^T (\mathbf{y} - X\mathbf{w}) = 0$$

We can now simplify these equations to find the optimal setting of the parameters  $\mathbf{w}$ :

$$\begin{aligned} -2X^T \mathbf{y} + 2X^T X \mathbf{w} &= 0 \\ X^T X \mathbf{w} &= X^T \mathbf{y} \\ \mathbf{w} &= (X^T X)^{-1} X^T \mathbf{y} \end{aligned}$$

You might have noticed that we assumed that the matrix  $X^T X$  can be inverted. If you take this fact for granted, then it should be clear that we can recover the exact optimal value  $\mathbf{w}^*$  exactly. No matter what values the data  $X, \mathbf{y}$  takes we can produce an exact answer by computing just one matrix multiplication, one matrix inversion, and two matrix-vector products.

## Numerical optimization

However, in practice and for the most interesting models, we usually can't find such analytical solutions. Even for logistic regression, possibly the second simplest model considered in this book, we don't have any exact solution. When we don't have an analytic solution, we need to

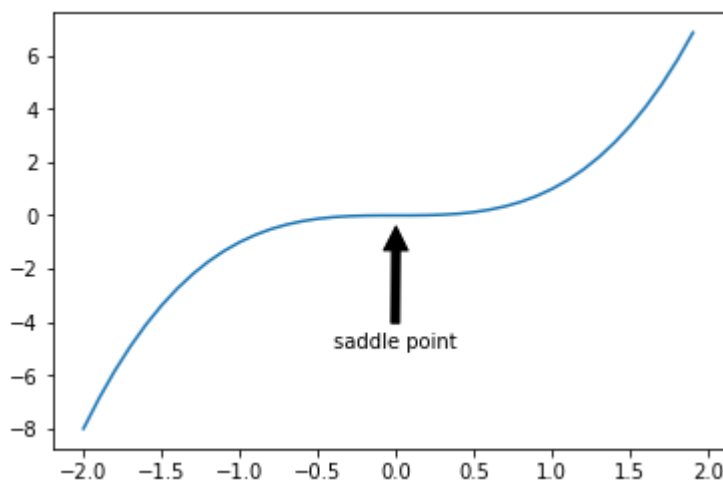
resort to a numerical solution. A numerical solution usually involves starting with some guess of the objective-minimizing setting of all the parameters, and successively improving the parameters iterative manner. The most popular optimization techniques of this variety are variants of gradient descent (GD). In the next notebook, we'll take a [deep dive into gradient descent and stochastic gradient descent \(SGD\)](#). Depending on the optimizer you use, iterative methods may take a long time to converge on a good answer.

For many problems, even if they don't have an analytic solution, they may have only one minima. An especially convenient class of functions are the *convex* functions. These are functions with a uniformly positive second derivative. They have no local minima and are especially well-suited to efficient optimization. Unfortunately, this is a book about neural networks. And neural networks are not in general convex. Moreover, they have abundant local minima. With numerical methods, it may not be possible to find the global minimizer of an objective function. For non-convex functions, a numerical method often halts around local minima that are not necessarily the global minima.

## Saddle points

Saddle points are another challenge for optimizers. Even though these points are not local minima, they are points where the gradient is equal to zero. For high dimensional models, saddle points are typically more numerous than local minima. We depict a saddle point example in one-dimensional space below.

```
In [2]: x = np.arange(-2.0, 2.0, 0.1)
fig = plt.figure()
subplt = fig.add_subplot(111)
subplt.annotate('saddle point', xy=(0, -0.2), xytext=(-0.4, -5.0),
               arrowprops=dict(facecolor='black', shrink=0.05))
plt.plot(x, x**3)
plt.show()
```



Many optimization algorithms, like Newton's method, are designed to be attracted to critical points, including minima and saddle points. Since saddle points are generally common in high-dimensional space, some optimization algorithms, such as the Newton's method, may fail to train deep learning models effectively as they may get stuck in saddle points. Another challenging scenarios for neural networks is that there may be large, flat regions in parameters space that correspond to bad values of the objective function.

## Challenges due to machine precision

Even for convex functions, where all minima are global minima, it may still be hard to find the precise optimal solutions. For one, the accuracy of any solution can be limited by the machine precision.

In computers, numbers are represented in a discrete manner. The accuracy of a floating-point system is characterized by a quantity called machine precision. For IEEE binary floating-point systems,

- single precision =  $2^{-24}$  (about 7 decimal digits of precision)
- double precision =  $2^{-53}$  (about 16 decimal digits of precision).

In fact, the precision of a solution to optimization can be worse than the machine precision. To demonstrate that, consider a function  $f : \mathbb{R} \rightarrow \mathbb{R}$ , its Taylor series expansion is

$$f(x + \epsilon) = f(x) + f'(x)\epsilon + \frac{f''(x)}{2}\epsilon^2 + \mathcal{O}(\epsilon^3)$$

where  $\epsilon$  is small. Denote the global optimum solution as  $x^*$  for minimizing  $f(x)$ . It usually holds that

$$f'(x^*) = 0 \quad \text{and} \quad f''(x^*) \neq 0.$$

Thus, for a small value  $\epsilon$ , we have

$$f(x^* + \epsilon) \approx f(x^*) + \mathcal{O}(\epsilon^2),$$

where the coefficient term of  $\mathcal{O}(\epsilon^2)$  is  $f''(x)/2$ . This means that a small change of order  $\epsilon$  in the optimum solution  $x^*$  will change the value of  $f(x^*)$  in the order of  $\epsilon^2$ . In other words, if there is an error in the function value, the precision of solution value is constrained by the order of the square root of that error. For example, if the machine precision is  $10^{-8}$ , the precision of the solution value is only in the order of  $10^{-4}$ , which is much worse than the machine precision.

## Optimality isn't everything

Although finding the precise global optimum solution to an objective function is hard, it is not always necessary for deep learning. To start with, we care about test set performance. So we may not even want to minimize the error on the training set to the lowest possible value. Moreover, finding a suboptimal minimum of a great model can still be better than finding the true global minimum of a lousy model.

Many algorithms have solid theoretical guarantees of convergence to global minima, but these guarantees often only hold for functions that are convex. In old times, most researchers tried to avoid non-convex optimizations due to the lack of guaranteed. Doing gradient descent without a theoretical guarantee of convergence was considered unprincipled. However, the practice is supported by a large body of empirical evidence. The state of the art models in computer vision, natural language processing, and speech recognition for example, all rely on applying numerical optimizer to non-convex objective functions. Machine learners now often have to choose between those methods that are beautiful and those that work. In the next sections we'll try to give you some more background on the field of optimisation and a deeper sense of the state of the art techniques for training neural networks.

## Next

[Gradient descent and stochastic gradient descent](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Optimization by gradient descent

In the previous tutorials, we decided *which direction* to move each parameter and *how much* to move each parameter by taking the gradient of the loss with respect to each parameter. We also scaled each gradient by some learning rate, although we never really explained where this number comes from. We then updated the parameters by performing a gradient step  $\theta_{t+1} \leftarrow \eta \nabla_{\theta} \mathcal{L}_t$ . Each update is called a *gradient step* and the process is called *gradient descent*.

The hope is that if we just take a whole lot of gradient steps, we'll wind up with an awesome model that gets very low loss on our training data, and that this performance might generalize to our hold-out data. But as a sharp reader, you might have any number of doubts. You might wonder, for instance:

- Why does gradient descent work?
- Why doesn't the gradient descent algorithm get stuck on the way to a low loss?
- How should we choose a learning rate?
- Do all the parameters need to share the same learning rate?
- Is there anything we can do to speed up the process?
- Why does the solution of gradient descent over training data generalize well to test data?

Some answers to these questions are known. For other questions, we have some answers but only for simple models like logistic regression that are easy to analyze. And for some of these questions, we know of best practices that seem to work even if they're not supported by any conclusive mathematical analysis. Optimization is a rich area of ongoing research. In this chapter, we'll address the parts that are most relevant for training neural networks. To begin, let's take a more formal look at gradient descent.

### Gradient descent in one dimension

To get going, consider a simple scenario in which we have one parameter to manipulate. Let's also assume that our objective associates every value of this parameter with a value. Formally, we can say that this objective function has the signature  $f : \mathbb{R} \rightarrow \mathbb{R}$ . It maps from one real number to another.

Note that the domain of  $f$  is in one-dimensional. According to its Taylor series expansion as shown in the [introduction chapter](#), we have



$$f(x + \epsilon) \approx f(x) + f'(x)\epsilon.$$

Substituting  $\epsilon$  with  $-\eta f'(x)$  where  $\eta$  is a constant, we have

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)^2.$$

If  $\eta$  is set as a small positive value, we obtain

$$f(x - \eta f'(x)) \leq f(x).$$

In other words, updating  $x$  as

$$x := x - \eta f'(x)$$

may reduce the value of  $f(x)$  if its current derivative value  $f'(x) \neq 0$ . Since the derivative  $f'(x)$  is a special case of gradient in one-dimensional domain, the above update of  $x$  is gradient descent in one-dimensional domain.

The positive scalar  $\eta$  is called the learning rate or step size. Note that a larger learning rate increases the chance of overshooting the global minimum and oscillating. However, if the learning rate is too small, the convergence can be very slow. In practice, a proper learning rate is usually selected with experiments.

## Gradient descent over multi-dimensional parameters

Consider the objective function  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  that takes any multi-dimensional vector  $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$  as its input. The gradient of  $f(\mathbf{x})$  with respect to  $\mathbf{x}$  is defined by the vector of partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[ \frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right]^\top.$$

To keep our notation compact we may use the notation  $\nabla f(\mathbf{x})$  and  $\nabla_{\mathbf{x}} f(\mathbf{x})$  interchangeably when there is no ambiguity about which parameters we are optimizing over. In plain English, each element  $\partial f(\mathbf{x}) / \partial x_i$  of the gradient indicates the rate of change for  $f$  at the point  $\mathbf{x}$  with respect to the input  $x_i$  only. To measure the rate of change of  $f$  in any direction that is represented by a unit vector  $\mathbf{u}$ , in multivariate calculus, we define the directional derivative of  $f$  at  $\mathbf{x}$  in the direction of  $\mathbf{u}$  as

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h},$$

which can be rewritten according to the chain rule as

$$D_{\mathbf{u}}f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}.$$

Since  $D_{\mathbf{u}}f(\mathbf{x})$  gives the rates of change of  $f$  at the point  $\mathbf{x}$  in all possible directions, to minimize  $f$ , we are interested in finding the direction where  $f$  can be reduced fastest. Thus, we can minimize the directional derivative  $D_{\mathbf{u}}f(\mathbf{x})$  with respect to  $\mathbf{u}$ . Since

$D_{\mathbf{u}}f(\mathbf{x}) = \|\nabla f(\mathbf{x})\| \cdot \|\mathbf{u}\| \cdot \cos(\theta) = \|\nabla f(\mathbf{x})\| \cdot \cos(\theta)$  where  $\theta$  is the angle between  $\nabla f(\mathbf{x})$  and  $\mathbf{u}$ , the minimum value of  $\cos(\theta)$  is -1 when  $\theta = \pi$ . Therefore,  $D_{\mathbf{u}}f(\mathbf{x})$  is minimized when  $\mathbf{u}$  is at the opposite direction of the gradient  $\nabla f(\mathbf{x})$ . Now we can iteratively reduce the value of  $f$  with the following gradient descent update:

$$\mathbf{x} := \mathbf{x} - \eta \nabla f(\mathbf{x}),$$

where the positive scalar  $\eta$  is called the learning rate or step size.

## Stochastic gradient descent

However, the gradient descent algorithm may be infeasible when the training data size is huge. Thus, a stochastic version of the algorithm is often used instead.

To motivate the use of stochastic optimization algorithms, note that when training deep learning models, we often consider the objective function as a sum of a finite number of functions:

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{x}),$$

where  $f_i(\mathbf{x})$  is a loss function based on the training data instance indexed by  $i$ . It is important to highlight that the per-iteration computational cost in gradient descent scales linearly with the training data set size  $n$ . Hence, when  $n$  is huge, the per-iteration computational cost of gradient descent is very high.

In view of this, stochastic gradient descent offers a lighter-weight solution. At each iteration, rather than computing the gradient  $\nabla f(\mathbf{x})$ , stochastic gradient descent randomly samples  $i$  at uniform and computes  $\nabla f_i(\mathbf{x})$  instead. The insight is, stochastic gradient descent uses  $\nabla f_i(\mathbf{x})$  as an unbiased estimator of  $\nabla f(\mathbf{x})$  since

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}).$$

In a generalized case, at each iteration a mini-batch  $\mathcal{B}$  that consists of indices for training data instances may be sampled at uniform with replacement. Similarly, we can use

$$\nabla f_{\mathcal{B}}(\mathbf{x}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla f_i(\mathbf{x})$$

to update  $\mathbf{x}$  as

$$\mathbf{x} := \mathbf{x} - \eta \nabla f_{\mathcal{B}}(\mathbf{x}),$$

where  $|\mathcal{B}|$  denotes the cardinality of the mini-batch and the positive scalar  $\eta$  is the learning rate or step size. Likewise, the mini-batch stochastic gradient  $\nabla f_{\mathcal{B}}(\mathbf{x})$  is an unbiased estimator for the gradient  $\nabla f(\mathbf{x})$ :

$$\mathbb{E}_{\mathcal{B}} \nabla f_{\mathcal{B}}(\mathbf{x}) = \nabla f(\mathbf{x}).$$

This generalized stochastic algorithm is also called mini-batch stochastic gradient descent and we simply refer to them as stochastic gradient descent (as generalized). The per-iteration computational cost is  $\mathcal{O}(|\mathcal{B}|)$ . Thus, when the mini-batch size is small, the computational cost at each iteration is light.

There are other practical reasons that may make stochastic gradient descent more appealing than gradient descent. If the training data set has many redundant data instances, stochastic gradients may be so close to the true gradient  $\nabla f(\mathbf{x})$  that a small number of iterations will find useful solutions to the optimization problem. In fact, when the training data set is large enough, stochastic gradient descent only requires a small number of iterations to find useful solutions such that the total computational cost is lower than that of gradient descent even for just one iteration. Besides, stochastic gradient descent can be considered as offering a regularization effect especially when the mini-batch size is small due to the randomness and noise in the mini-batch sampling. Moreover, certain hardware processes mini-batches of specific sizes more efficiently.

## Experiments

For demonstrating the aforementioned gradient-based optimization algorithms, we use the regression problem in the [linear regression chapter](#) as a case study.

First, we import related libraries, generate the synthetic data, and construct the model.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import autograd
from mxnet import gluon
import numpy as np

X = np.random.randn(10000, 2)
Y = 2 * X[:,0] - 3.4 * X[:,1] + 4.2 + .01 * np.random.normal(size=10000)

ctx = mx.cpu()
net = gluon.nn.Sequential()
```

```
net.add(gluon.nn.Dense(1))
net.collect_params().initialize()
loss = gluon.loss.L2Loss()
```

Then we specify the batch sizes and learning rates for stochastic gradient descent algorithms. Since the number of samples is 10,000, when the batch size is 10,000, the algorithm is essentially gradient descent.

```
In [2]: batch_sizes = [1, 10, 100, 1000, 10000]
        learning_rates = [0.1, 0.1, 0.5, 0.5, 1.0]
        epochs = 3
```

Now we are ready to train the models and observe the inferred parameter values after the model training.

```
In [3]: for batch_size, learning_rate in zip(batch_sizes, learning_rates):
        trainer = gluon.Trainer(net.collect_params(), 'sgd',
                                {'learning_rate': learning_rate})
        net.collect_params().initialize(mx.init.Xavier(magnitude=2.24),
                                         ctx=ctx, force_reinit=True)
        train_data = mx.io.NDArrayIter(X, Y, batch_size=batch_size, shuffle=True)

        for e in range(epochs):
            train_data.reset()
            for i, batch in enumerate(train_data):
                data = batch.data[0].as_in_context(ctx)
                label = batch.label[0].as_in_context(ctx).reshape((-1, 1))
                with autograd.record():
                    output = net(data)
                    mse = loss(output, label)
                mse.backward()
                trainer.step(data.shape[0])

        for para_name, para_value in net.collect_params().items():
            print("Batch size:", batch_size, para_name,
                  para_value.data().asnumpy()[0])
```

```
Batch size: 1 dense0_weight [ 2.00191855 -3.40352154]
Batch size: 1 dense0_bias 4.19731
Batch size: 10 dense0_weight [ 1.99962676 -3.40070438]
Batch size: 10 dense0_bias 4.20087
Batch size: 100 dense0_weight [ 1.99993229 -3.40007687]
Batch size: 100 dense0_bias 4.20093
Batch size: 1000 dense0_weight [ 2.0000422 -3.40001583]
Batch size: 1000 dense0_bias 4.20009
Batch size: 10000 dense0_weight [ 2.00000167 -3.40008521]
Batch size: 10000 dense0_bias 4.20016
```

As expected, all the investigated algorithms find the weight vector to be close to [2, -3.4] and the bias term to be close to 4.2 as specified in the synthetic data generation.

Although the above demonstration uses a fixed learning rate for stochastic gradient descent, in practice a decaying learning rate is often needed. This is because the noise in random sampling does not vanish throughout the iterations. For gradient descent and for some objective

functions, the true gradient tends to get smaller, approaching the zero vector. In these cases it may be ok to use a fixed learning rate because the steps will naturally get smaller as the gradient gets smaller.

## Next

[SGD with momentum](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Stochastic gradient descent with momentum

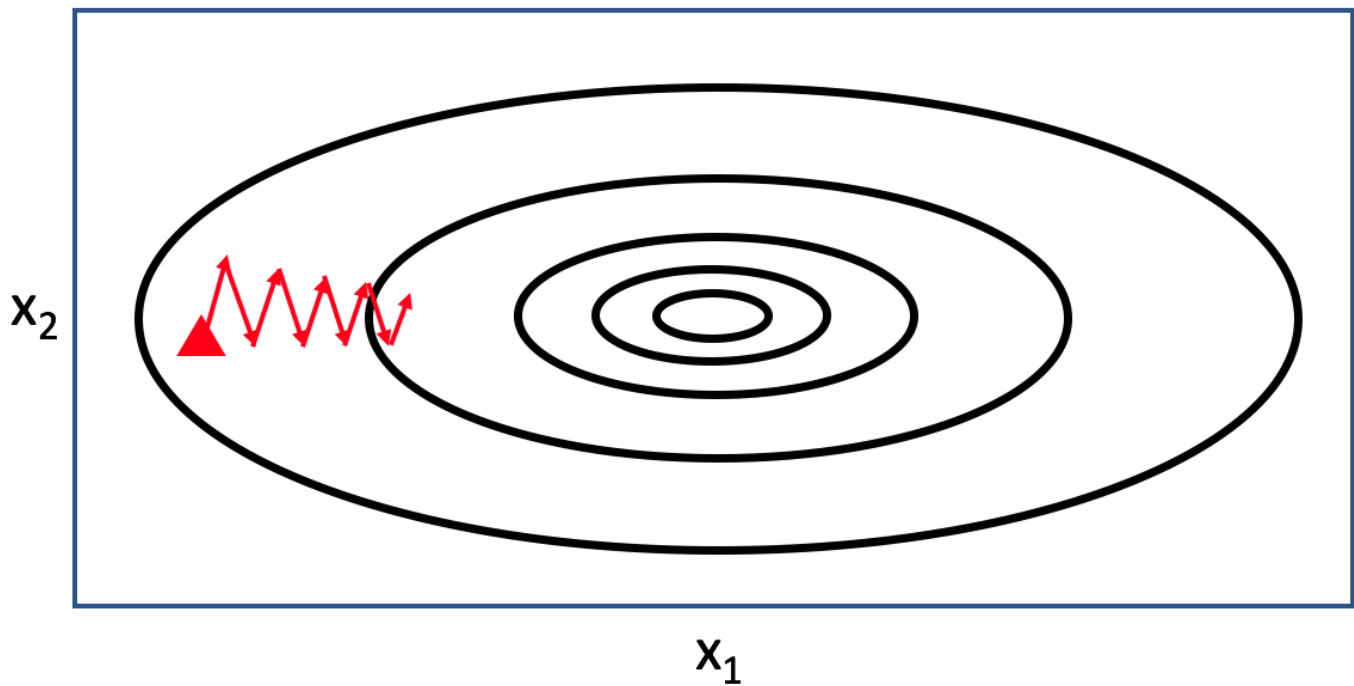
As discussed in the [previous chapter](#), at each iteration stochastic gradient descent (SGD) finds the direction where the objective function can be reduced fastest on a given example. Thus, gradient descent is also known as the method of steepest descent. Essentially, SGD is a myopic algorithm. It doesn't look very far into the past and it doesn't think much about the future. At each step, SGD just does whatever looks right just at that moment.

You might wonder, can we do something smarter? It turns out that we can. One class of methods use an idea called *momentum*. The idea of momentum-based optimizers is to remember the previous gradients from recent optimization steps and to use them to help to do a better job of choosing the direction to move next, acting less like a drunk student walking downhill and more like a rolling ball. In this chapter we'll motivate and explain SGD with momentum.

### Motivating example

In order to motivate the method, let's start by visualizing a simple quadratic objective function  $f: \mathbb{R}^2 \rightarrow \mathbb{R}$  taking a two-dimensional vector  $(\mathbf{x} = [x_1, x_2]^{\top})$  as the input. In the following figure, each contour line indicates points of equivalent value  $f(\mathbf{x})$ . The objective function is minimized in the center and the outer rings have progressively worse values.

The red triangle indicates the starting point for our stochastic gradient descent optimizer. The lines and arrows that follow indicate each step of SGD. You might wonder why the lines don't just point directly towards the center. That's because the gradient estimates in SGD are noisy, due to the small sample size. So the gradient steps are noisy even if they are correct on average (unbiased). As you can see, SGD wastes too much time swinging back and forth along the direction in parallel with the  $x_2$ -axis while advancing too slowly along the direction of the  $x_1$ -axis.



## Curvature and Hessian matrix

Even if we just did plain old gradient descent, we'd expect our function to bounce around quite a lot. That's because our gradient is changing as we move around in parameter space due to the curvature of the function.

We can reason about the curvature of objective function by considering their second derivative. The second derivative says how much the gradient changes as we move in parameter space. In one dimension, a second derivative of a function indicates how fast the first derivative changes when the input changes. Thus, it is often considered as a measure of the **curvature** of a function. *It is the rate of change of the rate of change.* If you've never done calculus before, that might sound rather *meta*, but you'll get over it.

Consider the objective function  $f: \mathbb{R}^d \rightarrow \mathbb{R}$  that takes a multi-dimensional vector  $\mathbf{x} = [x_1, x_2, \dots, x_d]^T$  as the input. Its **Hessian matrix**  $\mathbf{H} \in \mathbb{R}^{d \times d}$  collects its second derivatives. Each entry  $(i, j)$  says how much the gradient of the objective with respect to parameter  $(i)$  changes, with a small change in parameter  $(j)$ .

$$[\mathbf{H}]_{i,j} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}$$
for all  $(i, j = 1, \dots, d)$ . Since  $\mathbf{H}$  is a real symmetric matrix, by spectral theorem, it is orthogonally diagonalizable as

$$\mathbf{S}^T \mathbf{H} \mathbf{S} = \mathbf{\Lambda}$$
where  $\mathbf{S}$  is an orthonormal eigenbasis composed of eigenvectors of  $\mathbf{H}$  with corresponding eigenvalues in a diagonal matrix  $\mathbf{\Lambda}$ : the eigenvalue  $\mathbf{\Lambda}_{i,i}$  corresponds to the eigenvector in the  $i^{\text{th}}$  column of  $\mathbf{S}$

$(\mathbf{S})$ . The second derivative (curvature) of the objective function  $(f)$  in any direction  $(\mathbf{d})$  (unit vector) is a quadratic form  $(\mathbf{d}^T \mathbf{H} \mathbf{d})$ . Specifically, if the direction  $(\mathbf{d})$  is an eigenvector of  $(\mathbf{H})$ , the curvature of  $(f)$  in that direction is equal to the corresponding eigenvalue of  $(\mathbf{H})$ . Since the curvature of the objective function in any direction is a weighted average of all the eigenvalues of the Hessian matrix, the curvature is bounded by the minimum and maximum eigenvalues of the Hessian matrix  $(\mathbf{H})$ . The ratio of the maximum to the minimum eigenvalue is the **condition number** of the Hessian matrix  $(\mathbf{H})$ .

## Gradient descent in ill-conditioned problems

How does the condition number of the Hessian matrix of the objective function affect the performance of gradient descent? Let us revisit the problem in the motivating example.

Recall that gradient descent is a greedy approach that selects the steepest gradient at the current point as the direction of advancement. At the starting point, the search by gradient descent advances more aggressively in the direction of the  $(x_2)$ -axis than that of the  $(x_1)$ -axis.

In the plotted problem of the motivating example, the curvature in the direction of the  $(x_2)$ -axis is much larger than that of the  $(x_1)$ -axis. Thus, gradient descent tends to overshoot the bottom of the function that is projected to the plane in parallel with the  $(x_2)$ -axis. At the next iteration, if the gradient along the direction in parallel with the  $(x_2)$ -axis remains larger, the search continues to advance more aggressively along the direction in parallel with the  $(x_2)$ -axis and the overshooting continues to take place. As a result, gradient descent wastes too much time swinging back and forth in parallel with the  $(x_2)$ -axis due to overshooting while the advancement in the direction of the  $(x_1)$ -axis is too slow.

To generalize, the problem in the motivating example is an ill-conditioned problem. In an ill-conditioned problem, the condition number of the Hessian matrix of the objective function is large. In other words, the ratio of the largest curvature to the smallest is high.

## The momentum algorithm

The aforementioned ill-conditioned problems are challenging for gradient descent. By treating gradient descent as a special form of stochastic gradient descent, we can address the challenge with the following momentum algorithm for stochastic gradient descent.

$$\begin{aligned} \mathbf{v} &:= \gamma \mathbf{v} + \eta \nabla f_{\mathcal{B}}(\mathbf{x}), \\ \mathbf{x} &:= \mathbf{x} - \mathbf{v}, \end{aligned}$$



where  $\mathbf{v}$  is the current velocity and  $\gamma$  is the momentum parameter. The learning rate  $\eta$  and the stochastic gradient  $\nabla f_{\mathcal{B}}(\mathbf{x})$  with respect to the sampled mini-batch  $\mathcal{B}$  are both defined in the [previous chapter](#).

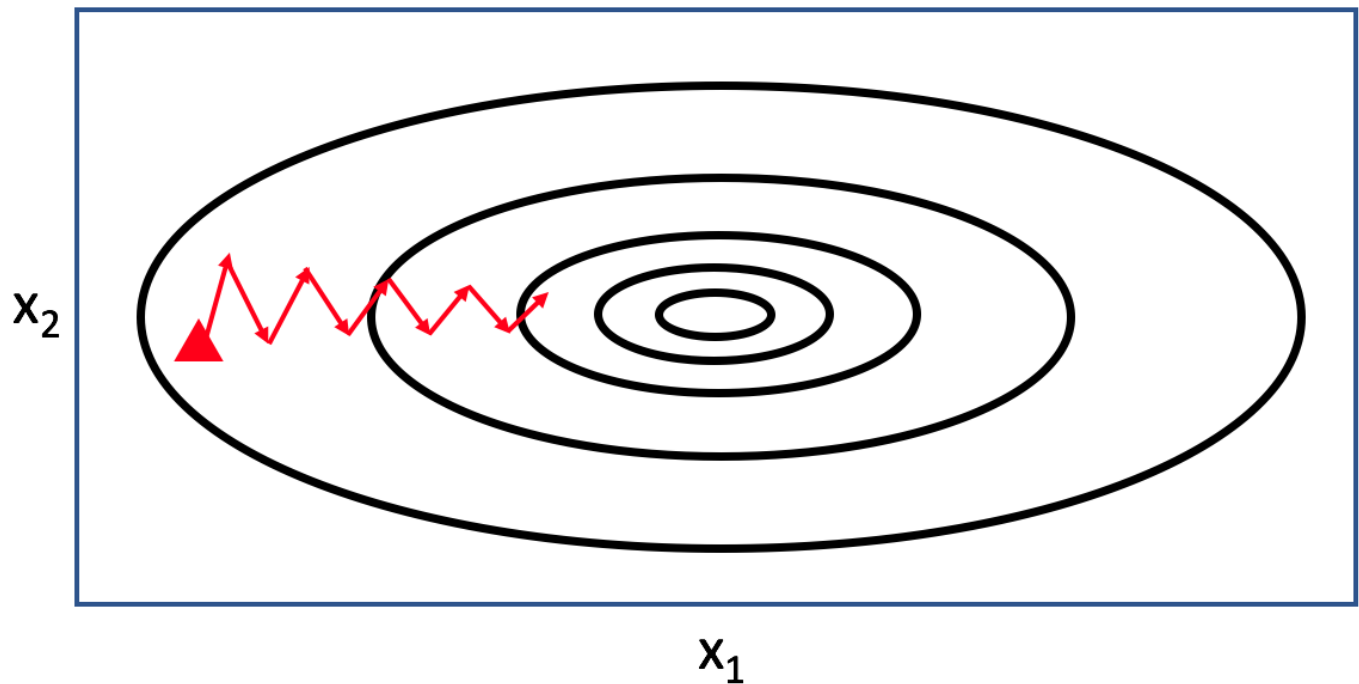
It is important to highlight that, the scale of advancement at each iteration now also depends on how aligned the directions of the past gradients are. This scale is the largest when all the past gradients are perfectly aligned to the same direction.

To better understand the momentum parameter  $\gamma$ , let us simplify the scenario by assuming the stochastic gradients  $\nabla f_{\mathcal{B}}(\mathbf{x})$  are the same as  $\mathbf{g}$  throughout the iterations. Since all the gradients are perfectly aligned to the same direction, the momentum algorithm accelerates the advancement along the same direction of  $\mathbf{g}$  as

$$\begin{aligned} \mathbf{v}_1 &:= \eta \mathbf{g}, \\ \mathbf{v}_2 &:= \gamma \mathbf{v}_1 + \eta \mathbf{g} = \eta \mathbf{g} (\gamma + 1), \\ \mathbf{v}_3 &:= \gamma \mathbf{v}_2 + \eta \mathbf{g} = \eta \mathbf{g} (\gamma^2 + \gamma + 1), \\ &\vdots \\ \mathbf{v}_\infty &:= \frac{\eta \mathbf{g}}{1 - \gamma}. \end{aligned}$$

Thus, if  $\gamma = 0.99$ , the final velocity is 100 times faster than that of the corresponding gradient descent where the gradient is  $\mathbf{g}$ .

Now with the momentum algorithm, a sample search path can be improved as illustrated in the following figure.



## Experiments

For demonstrating the momentum algorithm, we still use the regression problem in the [linear regression chapter](#) as a case study. Specifically, we investigate stochastic gradient descent with momentum.

As usual, we import related libraries, generate the synthetic data, and construct the model.

```
In [1]: from __future__ import print_function
import mxnet as mx
from mxnet import autograd
from mxnet import gluon
import numpy as np

X = np.random.randn(10000, 2)
Y = 2 * X[:,0] - 3.4 * X[:,1] + 4.2 + .01 * np.random.normal(size=10000)

ctx = mx.cpu()
net = gluon.nn.Sequential()
net.add(gluon.nn.Dense(1))
net.collect_params().initialize()
loss = gluon.loss.L2Loss()
```

Then we specify the batch sizes and learning rates for stochastic gradient descent algorithms with momentum. Specifically, the momentum parameter is set to 0.5.

```
In [2]: batch_sizes = [1, 10, 100, 1000]
learning_rates = [0.1, 0.1, 0.5, 0.5]
momentum_param = 0.5

epochs = 3
```

Now we are ready to train the models and observe the inferred parameter values after the model training.

```
In [3]: for batch_size, learning_rate in zip(batch_sizes, learning_rates):
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': learning_rate,
                             'momentum': momentum_param})
    net.collect_params().initialize(mx.init.Xavier(magnitude=2.24),
                                    ctx=ctx, force_reinit=True)
    train_data = mx.io.NDArrayIter(X, Y, batch_size=batch_size, shuffle=True)

    for e in range(epochs):
        train_data.reset()
        for i, batch in enumerate(train_data):
            data = batch.data[0].as_in_context(ctx)
            label = batch.label[0].as_in_context(ctx).reshape((-1, 1))
            with autograd.record():
                output = net(data)
                mse = loss(output, label)
            mse.backward()
            trainer.step(data.shape[0])

        for para_name, para_value in net.collect_params().items():
            print("Batch size:", batch_size, para_name,
                  para_value.data().asnumpy()[0])
```

```
Batch size: 1 dense0_weight [ 2.00212836 -3.39304566]
Batch size: 1 dense0_bias 4.20125
Batch size: 10 dense0_weight [ 2.00067949 -3.40136981]
```

```
Batch size: 10 dense0_bias 4.20053  
Batch size: 100 dense0_weight [ 2.00005388 -3.40086675]  
Batch size: 100 dense0_bias 4.19988  
Batch size: 1000 dense0_weight [ 2.00004411 -3.39969349]  
Batch size: 1000 dense0_bias 4.20037
```

As expected, all the investigated algorithms find the weight vector to be close to [2, -3.4] and the bias term to be close to 4.2 as specified in the synthetic data generation.

## Next

[Fast & flexible: combining imperative & symbolic nets with HybridBlocks](#)

For whinges or inquiries, [open an issue on GitHub](#).

# Fast, portable neural networks with Gluon HybridBlocks

The tutorials we saw so far adopt the *imperative*, or define-by-run, programming paradigm. It might not even occur to you to give a name to this style of programming because it's how we always write Python programs.

Take for example a prototypical program written below in pseudo-Python. We grab some input arrays, we compute upon them to produce some intermediate values, and finally we produce the result that we actually care about.

```
def our_function(A, B, C, D):  
    # Compute some intermediate values  
    E = basic_function1(A, B)  
    F = basic_function2(C, D)  
  
    # Finally, produce the thing you really care about  
    G = basic_function3(E, F)  
    return G  
  
# Load up some data  
W = some_stuff()  
X = some_stuff()  
Y = some_stuff()  
Z = some_stuff()  
  
result = our_function(W, X, Y, Z)
```

As you might expect when we compute `E`, we're actually performing some numerical operation, like multiplication, and returning an array that we assign to the variable `E`. Same for `F`. And if we want to do a similar computation many times by putting these lines in a function, each time our program *will have to step through these three lines of Python*.

The advantage of this approach is it's so natural that it might not even occur to some people that there is another way. But the disadvantage is that it's slow. That's because we are constantly engaging the Python execution environment (which is slow) even though our entire function performs the same three low-level operations in the same sequence every time. It's also holding on to all the intermediate values `D` and `E` until the function returns even though we can see that they're not needed. We might have made this program more efficient by re-using memory from either `E` or `F` to store the result `G`.

There actually is a different way to do things. It's called *symbolic* programming and most of the early deep learning libraries, including Theano and Tensorflow, embraced this approach exclusively. You might have also heard this approach referred to as *declarative* programming or *define-then-run* programming. These all mean the exact same thing. The approach consists of three basic steps:

- Define a computation workflow, like a pass through a neural network, using placeholder data
- Compile the program into a front-end language, e.g. Python, independent format
- Invoke the compiled function, feeding it real data

Revisiting our previous pseudo-Python example, a symbolic version of the same program might look something like this:

```
# Create some placeholders to stand in for real data that might be supplied to the compiled function.
A = placeholder()
B = placeholder()
C = placeholder()
D = placeholder()

# Compute some intermediate values
E = symbolic_function1(A, B)
F = symbolic_function2(C, D)

# Finally, produce the thing you really care about
G = symbolic_function3(E, F)

our_function = library.compile(inputs=[A, B, C, D], outputs=[G])

# Load up some data
W = some_stuff()
X = some_stuff()
Y = some_stuff()
Z = some_stuff()

result = our_function(W, X, Y, Z)
```

Here, when we run the line `E = symbolic_function1(A, B)`, *no numerical computation actually happens*. Instead, the symbolic library notes the way that `E` is related to `A` and `B` and records this information. We don't do actual computation, we just make *a roadmap* for how to go from inputs to outputs. Because we can draw all of the variables and operations (both inputs and intermediate values) as nodes, and the relationships between nodes with edges, we call the resulting roadmap a computational graph. In the symbolic approach, we first define the entire graph, and then compile it.

## Imperative Programs Tend to be More Flexible

When you're using an imperative-style library from Python, you are writing in Python. Nearly anything that would be intuitive to write in Python, you could accelerate by calling down in the appropriate places to the imperative deep learning library. On the other hand, when you write a symbolic program, you may not have access to all the familiar Python constructs, like iteration. It's also easy to debug an imperative program. For one, because all the intermediate values hang around, it's easy to introspect the program later. Imperative programs are also much easier to debug because we can just stick print statements in between operations.

In short, from a developer's standpoint, imperative programs are just better. They're a joy to work with. You don't have the tricky indirection of working with placeholders. You can do anything that you can do with native Python. And faster debugging, means you get to try out more ideas. But the catch is that imperative programs are *comparatively* slow.

## Symbolic Programs Tend to be More Efficient

The main reason is efficiency, both in terms of memory and speed. Let's revisit our toy example from before. Consider the following program:

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
...
```

Assume that each cell in the array occupies 8 bytes of memory. How much memory do we need to execute this program in the Python console? As an imperative program we need to allocate memory at each line. That leaves us allocating 4 arrays of size 10. So we'll need  $4 * 10 * 8 = 320$  bytes. On the other hand, if we built a computation graph, and knew in advance that we only needed d, we could reuse the memory originally allocated for intermediate values. For example, by performing computations in-place, we might recycle the bits allocated for b to store c. And we might recycle the bits allocated for c to store d. In the end we could cut our memory requirement in half, requiring just  $2 * 10 * 8 = 160$  bytes.

Symbolic programs can also perform another kind of optimization, called operation folding. Returning to our toy example, the multiplication and addition operations can be folded into one operation. If the computation runs on a GPU processor, one GPU kernel will be executed, instead of two. In fact, this is one way we hand-craft operations in optimized libraries, such as CXXNet and Caffe. Operation folding improves computation efficiency. Note, you can't perform operation folding in imperative programs, because the intermediate values might be referenced

in the future. Operation folding is possible in symbolic programs because we get the entire computation graph in advance, before actually doing any calculation, giving us a clear specification of which values will be needed and which will not.

## Getting the best of both worlds with MXNet Gluon's HybridBlocks

Most libraries deal with the imperative / symbolic design problem by simply choosing a side. Theano and those frameworks it inspired, like TensorFlow, run with the symbolic way. And because the first versions of MXNet optimized performance, they also went symbolic. Chainer and its descendants like PyTorch are fully imperative way. In designing MXNet Gluon, we asked the following question. Is it possible to get *all* of the benefits of imperative programming but to still exploit, whenever possible, the speed and memory efficiency of symbolic programming. In other words, a user should be able to use Gluon fully imperatively. And if they never want their lives to be more complicated then they can get on just fine imagining that the story ends there. But when a user needs production-level performance, it should be easy to compile the entire compute graph, or at least to compile large subsets of it.

MXNet accomplishes this through the use of HybridBlocks. Each `HybridBlock` can run fully imperatively defining their computation with real functions acting on real inputs. But they're also capable of running symbolically, acting on placeholders. Gluon hides most of this under the hood so you'll only need to know how it works when you want to write your own layers. Given a HybridBlock whose forward computation consists of going through other HybridBlocks, you can compile that section of the network by calling the HybridBlocks `.hybridize()` method.

**All of MXNet's predefined layers are HybridBlocks.** This means that any network consisting entirely of predefined MXNet layers can be compiled and run at much faster speeds by calling `.hybridize()`.

## HybridSequential

We already learned how to use `Sequential` to stack the layers. The regular `Sequential` can be built from regular Blocks and so it too has to be a regular Block. However, when you want to build a network using sequential and run it at crazy speeds, you can construct your network using `HybridSequential` instead. The functionality is the same `Sequential`:

```
In [1]: import mxnet as mx
        from mxnet.gluon import nn
        from mxnet import nd

        def get_net():
            # construct a MLP
            net = nn.HybridSequential()
            with net.name_scope():
                net.add(nn.Dense(256, activation="relu"))
                net.add(nn.Dense(128, activation="relu"))
                net.add(nn.Dense(2))
```

```
# initialize the parameters
net.collect_params().initialize()
return net

# forward
x = nd.random_normal(shape=(1, 512))
net = get_net()
print('=== net(x) ==={}'.format(net(x)))

=== net(x) ===
[[ 0.16526183 -0.14005636]]
<NDArray 1x2 @cpu(0)>
```

To compile and optimize the `HybridSequential`, we can then call its `hybridize` method. Only `HybridBlock`s, e.g. `HybridSequential`, can be compiled. But you can still call `hybridize` on normal `Block` and its `HybridBlock` children will be compiled instead. We will talk more about `HybridBlock`s later.

```
In [2]: net.hybridize()
print('=== net(x) ==={}'.format(net(x)))

=== net(x) ===
[[ 0.16526183 -0.14005636]]
<NDArray 1x2 @cpu(0)>
```

## Performance

To get a sense of the speedup from hybridizing, we can compare the performance before and after hybridizing by measuring in either case the time it takes to make 1000 forward passes through the network.

```
In [3]: from time import time
def bench(net, x):
    mx.nd.waitall()
    start = time()
    for i in range(1000):
        y = net(x)
    mx.nd.waitall()
    return time() - start

net = get_net()
print('Before hybridizing: %.4f sec'%(bench(net, x)))
net.hybridize()
print('After hybridizing: %.4f sec'%(bench(net, x)))

Before hybridizing: 0.4646 sec
After hybridizing: 0.2424 sec
```

As you can see, hybridizing gives a significant performance boost, almost 2x the speed.

## Get the symbolic program



Previously, we feed `net` with `NDArray` data `x`, and then `net(x)` returned the forward results. Now if we feed it with a `Symbol` placeholder, then the corresponding symbolic program will be returned.

```
In [4]: from mxnet import sym
x = sym.var('data')
print('=== input data holder ===')
print(x)

y = net(x)
print('\n=== the symbolic program of net===')
print(y)

y_json = y.tojson()
print('\n=== the according json definition===')
print(y_json)

=== input data holder ===
<Symbol data>

=== the symbolic program of net===
<Symbol hybridsequential1_dense2_fwd>

=== the according json definition===
{
  "nodes": [
    {
      "op": "null",
      "name": "data",
      "inputs": []
    },
    {
      "op": "null",
      "name": "hybridsequential1_dense0_weight",
      "attr": {
        "__dtype__": "0",
        "__lr_mult__": "1.0",
        "__shape__": "(256, 0)",
        "__wd_mult__": "1.0"
      },
      "inputs": []
    },
    {
      "op": "null",
      "name": "hybridsequential1_dense0_bias",
      "attr": {
        "__dtype__": "0",
        "__init__": "zeros",
        "__lr_mult__": "1.0",
        "__shape__": "(256,)",
        "__wd_mult__": "1.0"
      },
      "inputs": []
    },
    {
      "op": "FullyConnected",
      "name": "hybridsequential1_dense0_fwd",
      "attr": {"num_hidden": "256"},
      "inputs": [[0, 0, 0], [1, 0, 0], [2, 0, 0]]
    },
    {
      "op": "Activation",
      "name": "hybridsequential1_dense0_relu_fwd",
      "attr": {"act_type": "relu"},
      "inputs": [[3, 0, 0]]
    },
    {
      "op": "null",
```

```

    "name": "hybridsequential1_dense1_weight",
    "attr": {
      "__dtype__": "0",
      "__lr_mult__": "1.0",
      "__shape__": "(128, 0)",
      "__wd_mult__": "1.0"
    },
    "inputs": []
  },
  {
    "op": "null",
    "name": "hybridsequential1_dense1_bias",
    "attr": {
      "__dtype__": "0",
      "__init__": "zeros",
      "__lr_mult__": "1.0",
      "__shape__": "(128,)",
      "__wd_mult__": "1.0"
    },
    "inputs": []
  },
  {
    "op": "FullyConnected",
    "name": "hybridsequential1_dense1_fwd",
    "attr": {"num_hidden": "128"},
    "inputs": [[4, 0, 0], [5, 0, 0], [6, 0, 0]]
  },
  {
    "op": "Activation",
    "name": "hybridsequential1_dense1_relu_fwd",
    "attr": {"act_type": "relu"},
    "inputs": [[7, 0, 0]]
  },
  {
    "op": "null",
    "name": "hybridsequential1_dense2_weight",
    "attr": {
      "__dtype__": "0",
      "__lr_mult__": "1.0",
      "__shape__": "(2, 0)",
      "__wd_mult__": "1.0"
    },
    "inputs": []
  },
  {
    "op": "null",
    "name": "hybridsequential1_dense2_bias",
    "attr": {
      "__dtype__": "0",
      "__init__": "zeros",
      "__lr_mult__": "1.0",
      "__shape__": "(2,)",
      "__wd_mult__": "1.0"
    },
    "inputs": []
  },
  {
    "op": "FullyConnected",
    "name": "hybridsequential1_dense2_fwd",
    "attr": {"num_hidden": "2"},
    "inputs": [[8, 0, 0], [9, 0, 0], [10, 0, 0]]
  }
],
"arg_nodes": [0, 1, 2, 5, 6, 9, 10],
"node_row_ptr": [
  0,
  1,
  2,
  3,
  4,
  5,
  6,

```

```

7,
8,
9,
10,
11,
12
],
"heads": [[11, 0, 0]],
"attrs": {"mxnet_version": ["int", 1001]}
}

```

Now we can save both the program and parameters onto disk, so that it can be loaded later not only in Python, but in all other supported languages, such as C++, R, and Scala, as well.

```

In [5]: y.save('model.json')
        net.save_params('model.params')

```

## HybridBlock

Now let's dive deeper into how `hybridize` works. Remember that gluon networks are composed of Blocks each of which subclass `gluon.Block`. With normal Blocks, we just need to define a forward function that takes an input `x` and computes the result of the forward pass through the network. MXNet can figure out the backward pass for us automatically with autograd.

To define a `HybridBlock`, we instead have a `hybrid_forward` function:

```

In [6]: from mxnet import gluon

class Net(gluon.HybridBlock):
    def __init__(self, **kwargs):
        super(Net, self).__init__(**kwargs)
        with self.name_scope():
            self.fc1 = nn.Dense(256)
            self.fc2 = nn.Dense(128)
            self.fc3 = nn.Dense(2)

    def hybrid_forward(self, F, x):
        # F is a function space that depends on the type of x
        # If x's type is NDArray, then F will be mxnet.nd
        # If x's type is Symbol, then F will be mxnet.sym
        print('type(x): {}, F: {}'.format(
            type(x).__name__, F.__name__))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

```

The `hybrid_forward` function takes an additional input, `F`, which stands for a backend. This exploits one awesome feature of MXNet. MXNet has both a symbolic API (`mxnet.symbol`) and an imperative API (`mxnet.ndarray`). In this book, so far, we've only focused on the latter. Owing to fortuitous historical reasons, the imperative and symbolic interfaces both support roughly the same API. They have many of same functions (currently about 90% overlap) and when they do, they support the same arguments in the same order. When we define `hybrid_forward`, we pass in

`F`. When running in imperative mode, `hybrid_forward` is called with `F` as `mxnet.ndarray` and `x` as some ndarray input. When we compile with `hybridize`, `F` will be `mxnet.symbol` and `x` will be some placeholder or intermediate symbolic value. Once we call `hybridize`, the net is compiled, so we'll never need to call `hybrid_forward` again.

Let's demonstrate how this all works by feeding some data through the network twice. We'll do this for both a regular network and a hybridized net. You'll see that in the first case,

`hybrid_forward` is actually called twice.

```
In [7]: net = Net()
net.collect_params().initialize()
x = nd.random_normal(shape=(1, 512))
print('=== 1st forward ===')
y = net(x)
print('=== 2nd forward ===')
y = net(x)

=== 1st forward ===
type(x): NDArray, F: mxnet.ndarray
=== 2nd forward ===
type(x): NDArray, F: mxnet.ndarray
```

Now run it again after hybridizing.

```
In [8]: net.hybridize()
print('=== 1st forward ===')
y = net(x)
print('=== 2nd forward ===')
y = net(x)

=== 1st forward ===
type(x): Symbol, F: mxnet.symbol
=== 2nd forward ===
```

It differs from the previous execution in two aspects:

1. the input data type now is `Symbol` even when we fed an `NDArray` into `net`, because `glueon` implicitly constructed a symbolic data placeholder.
2. `hybrid_forward` is called once at the first time we run `net(x)`. It is because `glueon` will construct the symbolic program on the first forward, and then keep it for reuse later.

One main reason that the network is faster after hybridizing is because we don't need to repeatedly invoke the Python forward function, while keeping all computations within the highly efficient C++ backend engine.

But the potential drawback is the loss of flexibility to write the forward function. In other ways, inserting `print` for debugging or control logic such as `if` and `for` into the forward function is not possible now.

## Conclusion

Through `HybridSequential` and `HybridBlock`, we can convert an imperative program into a symbolic program by calling `hybridize`.

## Next

[Training MXNet models with multiple GPUs](#)

For whinges or inquiries, [open an issue on GitHub](#).



## gluon

Gluon makes it easy to implement data parallel training. In this notebook, we'll implement data parallel training for a convolutional neural network. If you'd like a finer grained view of the concepts, you might want to first read the previous notebook, [multi gpu from scratch](#) with `gluon`.

To get started, let's first define a simple convolutional neural network and loss function.

```
In [1]: from mxnet import gluon, gpu
net = gluon.nn.Sequential(prefix='cnn_')
with net.name_scope():
    net.add(gluon.nn.Conv2D(channels=20, kernel_size=3, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=(2,2), strides=(2,2)))
    net.add(gluon.nn.Conv2D(channels=50, kernel_size=5, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=(2,2), strides=(2,2)))
    net.add(gluon.nn.Flatten())
    net.add(gluon.nn.Dense(128, activation="relu"))
    net.add(gluon.nn.Dense(10))

loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

Gluon supports initialization of network parameters over multiple devices. We accomplish this by passing in an array of device contexts, instead of the single contexts we've used in earlier notebooks. When we pass in an array of contexts, the parameters are initialized to be identical across all of our devices.

```
In [2]: GPU_COUNT = 2 # increase if you have more
ctx = [gpu(i) for i in range(GPU_COUNT)]
net.collect_params().initialize(ctx=ctx)
```

Given a batch of input data, we can split it into parts (equal to the number of contexts) by calling `gluon.utils.split_and_load(batch, ctx)`. The `split_and_load` function doesn't just split the data, it also loads each part onto the appropriate device context.

So now when we call the forward pass on two separate parts, each one is computed on the appropriate corresponding device and using the version of the parameters stored there.

```
In [3]: from mxnet.test_utils import get_mnist
mnist = get_mnist()
```

```
batch = mnist['train_data'][0:GPU_COUNT*2, :]
data = gluon.utils.split_and_load(batch, ctx)
print(net(data[0]))
print(net(data[1]))
```

```
[[[-0.01017658  0.03012515  0.02999702  0.01175333 -0.01746453  0.00707828
   0.02404996  0.00616632 -0.02094562  0.0136827 ]
 [-0.01249129  0.0305641  0.02823936 -0.00159418 -0.00722831  0.00538148
   0.01476716  0.0225275  -0.02458289  0.0246105 ]]
<NDArray 2x10 @gpu(0)>

[[[-0.00349744  0.01896121  0.02959755  0.00261514  0.00015916 -0.00355723
   0.0040103  0.03075583 -0.00761715  0.00599077]
 [-0.00557119  0.02766508  0.02406837 -0.0007478  -0.00511122  0.00538528
   0.00292899  0.01488838 -0.00191687  0.01074106]]
<NDArray 2x10 @gpu(1)>
```

At any time, we can access the version of the parameters stored on each device. Recall from the first Chapter that our weights may not actually be initialized when we call `initialize` because the parameter shapes may not yet be known. In these cases, initialization is deferred pending shape inference.

```
In [4]: weight = net.collect_params()['cnn_conv0_weight']

for c in ctx:
    print('=== channel 0 of the first conv on {} ==={}'.format(
        c, weight.data(ctx=c)[0]))
```

```
=== channel 0 of the first conv on gpu(0) ===
[[[ 0.0068339  0.01299825  0.0301265 ]
 [ 0.04819721  0.01438687  0.05011239]
 [ 0.00628365  0.04861524 -0.01068833]]]
<NDArray 1x3x3 @gpu(0)>
=== channel 0 of the first conv on gpu(1) ===
[[[ 0.0068339  0.01299825  0.0301265 ]
 [ 0.04819721  0.01438687  0.05011239]
 [ 0.00628365  0.04861524 -0.01068833]]]
<NDArray 1x3x3 @gpu(1)>
```

Similarly, we can access the gradients on each of the GPUs. Because each GPU gets a different part of the batch (a different subset of examples), the gradients on each GPU vary.

```
In [5]: def forward_backward(net, data, label):
        with gluon.autograd.record():
            losses = [loss(net(X), Y) for X, Y in zip(data, label)]
            for l in losses:
                l.backward()

        label = gluon.utils.split_and_load(mnist['train_label'][0:4], ctx)
        forward_backward(net, data, label)
        for c in ctx:
            print('=== grad of channel 0 of the first conv2d on {} ==={}'.format(
                c, weight.grad(ctx=c)[0]))
```

```
=== grad of channel 0 of the first conv2d on gpu(0) ===
[[[-0.00481181  0.02549155  0.05066928]
 [ 0.01503928  0.04740803  0.0411102 ]
 [ 0.04527877  0.06305876  0.04087966]]]
<NDArray 1x3x3 @gpu(0)>
=== grad of channel 0 of the first conv2d on gpu(1) ===
[[[-0.01102538 -0.02251887 -0.02211753]
```



```
[-0.01587106 -0.03848277 -0.03960423]
[-0.03371562 -0.06092873 -0.064744   ]]]
<NDArray 1x3x3 @gpu(1)>
```

Now we can implement the remaining functions. Most of them are the same as [when we did everything by hand](#); one notable difference is that if a `gluon` trainer recognizes multi-devices, it will automatically aggregate the gradients and synchronize the parameters.

```
In [6]: from mxnet import nd
        from mxnet.io import NDArrayIter
        from time import time

        def train_batch(batch, ctx, net, trainer):
            # split the data batch and load them on GPUs
            data = gluon.utils.split_and_load(batch.data[0], ctx)
            label = gluon.utils.split_and_load(batch.label[0], ctx)
            # compute gradient
            forward_backward(net, data, label)
            # update parameters
            trainer.step(batch.data[0].shape[0])

        def valid_batch(batch, ctx, net):
            data = batch.data[0].as_in_context(ctx[0])
            pred = nd.argmax(net(data), axis=1)
            return nd.sum(pred == batch.label[0].as_in_context(ctx[0])).asscalar()

        def run(num_gpus, batch_size, lr):
            # the list of GPUs will be used
            ctx = [gpu(i) for i in range(num_gpus)]
            print('Running on {}'.format(ctx))

            # data iterator
            mnist = get_mnist()
            train_data = NDArrayIter(mnist["train_data"], mnist["train_label"], batch_size)
            valid_data = NDArrayIter(mnist["test_data"], mnist["test_label"], batch_size)
            print('Batch size is {}'.format(batch_size))

            net.collect_params().initialize(force_reinit=True, ctx=ctx)
            trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
            for epoch in range(5):
                # train
                start = time()
                train_data.reset()
                for batch in train_data:
                    train_batch(batch, ctx, net, trainer)
                nd.waitall() # wait until all computations are finished to benchmark the time
                print('Epoch %d, training time = %.1f sec'%(epoch, time()-start))

                # validating
                valid_data.reset()
                correct, num = 0.0, 0.0
                for batch in valid_data:
                    correct += valid_batch(batch, ctx, net)
                    num += batch.data[0].shape[0]
                print('validation accuracy = %.4f'%(correct/num))

            run(1, 64, .3)
            run(GPU_COUNT, 64*GPU_COUNT, .3*GPU_COUNT)
```

```
Running on [gpu(0)]
Batch size is 64
```

```
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_conv0_weight is already
```

```

initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_conv0_bias is already
initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_conv1_weight is already
initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_conv1_bias is already
initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_dense0_weight is already
initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_dense0_bias is already
initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_dense1_weight is already
initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)
/home/ubuntu/miniconda3/envs/gluon/lib/python3.6/site-
packages/mxnet/gluon/parameter.py:276: UserWarning: Parameter cnn_dense1_bias is already
initialized, ignoring. Set force_reinit=True to re-initialize.
"Set force_reinit=True to re-initialize."%self.name)

```

```

Epoch 0, training time = 10.3 sec
      validation accuracy = 0.9703
Epoch 1, training time = 10.1 sec
      validation accuracy = 0.9743
Epoch 2, training time = 10.1 sec
      validation accuracy = 0.9754
Epoch 3, training time = 10.1 sec
      validation accuracy = 0.9806
Epoch 4, training time = 10.1 sec
      validation accuracy = 0.1139
Running on [gpu(0), gpu(1)]
Batch size is 128
Epoch 0, training time = 8.4 sec
      validation accuracy = 0.1010
Epoch 1, training time = 8.3 sec
      validation accuracy = 0.1010
Epoch 2, training time = 8.3 sec
      validation accuracy = 0.1137
Epoch 3, training time = 8.3 sec
      validation accuracy = 0.1137
Epoch 4, training time = 8.3 sec
      validation accuracy = 0.1137

```

Both parameters and trainers in `gluon` support multi-devices. Moving from one device to multi-devices is straightforward.

### Distributed training with multiple machines

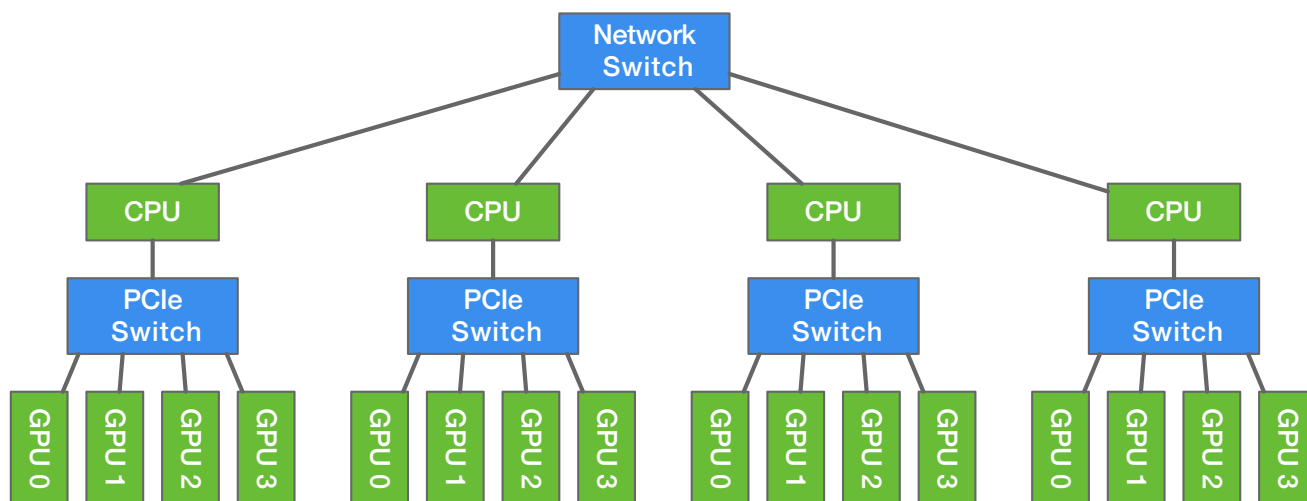
For whinges or inquiries, [open an issue on GitHub](#).



## Distributed training with multiple machines

In the previous two tutorials, we saw that using multiple GPUs within a machine can accelerate training. The speedup, however, is limited by the number of GPUs installed in that machine. And it's rare to find a single machine with more than 16 GPUs nowadays. For some truly large-scale applications, this speedup might still be insufficient. For example, it could still take many days to train a state-of-the-art CNN on millions of images.

In this tutorial, we'll discuss the key concepts you'll need in order to go from a program that does single-machine training to one that executes distributed training across multiple machines. We depict a typical distributed system in the following figure, where multiple machines are connected by network switches.



Note that the way we used `copyto` to copy data from one GPU to another in the [multiple-GPU tutorial](#) does not work when our GPUs are sitting on different machines. To make use of the available resources here we'll need a better abstraction.

## Key-value store

MXNet provides a key-value store to synchronize data among devices. The following code initializes an `ndarray` associated with the key “weight” on a key-value store.

```
In [1]: from mxnet import kv, nd
store = kv.create('local')
shape = (2, 3)
x = nd.random_uniform(shape=shape)
```

```
store.init('weight', x)
print('=== init "weight" ==={}'.format(x))
```

```
=== init "weight" ===
[[ 0.54881352  0.59284461  0.71518934]
 [ 0.84426576  0.60276335  0.85794562]]
<NDArray 2x3 @cpu(0)>
```

After initialization, we can pull the value to multiple devices.

```
In [2]: from mxnet import gpu
ctx = [gpu(0), gpu(1)]
y = [nd.zeros(shape, ctx=c) for c in ctx]
store.pull('weight', out=y)
print('=== pull "weight" to {} ===\n{}'.format(ctx, y))
```

```
=== pull "weight" to [gpu(0), gpu(1)] ===
[[ 0.54881352  0.59284461  0.71518934]
 [ 0.84426576  0.60276335  0.85794562]]
<NDArray 2x3 @gpu(0)>,
[[ 0.54881352  0.59284461  0.71518934]
 [ 0.84426576  0.60276335  0.85794562]]
<NDArray 2x3 @gpu(1)>
```

We can also push new data value into the store. It will first sum the data on the same key and then overwrite the current value.

```
In [3]: z = [nd.ones(shape, ctx=ctx[i])+i for i in range(len(ctx))]
store.push('weight', z)
print('=== push to "weight" ===\n{}'.format(z))
store.pull('weight', out=y)
print('=== pull "weight" ===\n{}'.format(y))
```

```
=== push to "weight" ===
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 2x3 @gpu(0)>,
[[ 2.  2.  2.]
 [ 2.  2.  2.]]
<NDArray 2x3 @gpu(1)>]
=== pull "weight" ===
[[ 3.  3.  3.]
 [ 3.  3.  3.]]
<NDArray 2x3 @gpu(0)>,
[[ 3.  3.  3.]
 [ 3.  3.  3.]]
<NDArray 2x3 @gpu(1)>]
```

With `push` and `pull` we can replace the `allreduce` function defined in [multiple-gpus-scratch](#) by

```
def allreduce(data, data_name, store):
    store.push(data_name, data)
    store.pull(data_name, out=data)
```

# Distributed key-value store

Not only can we synchronize data within a machine, with the key-value store we can facilitate inter-machine communication. To use it, one can create a distributed kvstore by using the following command: (Note: distributed key-value store requires `MXNet` to be compiled with the flag `USE_DIST_KVSTORE=1`, e.g. `make USE_DIST_KVSTORE=1`.)

```
store = kv.create('dist')
```

Now if we run the code from the previous section on two machines at the same time, then the store will aggregate the two ndarrays pushed from each machine, and after that, the pulled results will be:

```
[[ 6.  6.  6.]  
 [ 6.  6.  6.]]
```

In the distributed setting, `MXNet` launches three kinds of processes (each time, running `python myprog.py` will create a process). One is a *worker*, which runs the user program, such as the code in the previous section. The other two are the *server*, which maintains the data pushed into the store, and the *scheduler*, which monitors the aliveness of each node.

It's up to users which machines to run these processes on. But to simplify the process placement and launching, MXNet provides a tool located at [tools/launch.py](#).

Assume there are two machines, A and B. They are ssh-able, and their IPs are saved in a file named `hostfile`. Then we can start one worker in each machine through:

```
$ mxnet_path/tools/launch.py -H hostfile -n 2 python myprog.py
```

It will also start a server in each machine, and the scheduler on the same machine we are currently on.

## Using `kvstore` in `glueviz`

As mentioned in [our section on training with multiple GPUs from scratch](#), to implement data parallelism we just need to specify

- how to split data
- how to synchronize gradients and weights

We already see from [multiple-gpu-gluon](#) that a `gluon` trainer can automatically aggregate the gradients among different GPUs. What it really does is having a key-value store with type `local` within it. Therefore, to change to multi-machine training we only need to pass a distributed key-value store, for example,

```
store = kv.create('dist')
trainer = gluon.Trainer(..., kvstore=store)
```

To split the data, however, we cannot directly copy the previous approach. One commonly used solution is to split the whole dataset into  $k$  parts at the beginning, then let the  $i$ -th worker only read the  $i$ -th part of the data.

We can obtain the total number of workers by reading the attribute `num_workers` and the rank of the current worker from the attribute `rank`.

```
In [4]: print('total number of workers: %d'%(store.num_workers))
        print('my rank among workers: %d'%(store.rank))
```

```
total number of workers: 1
my rank among workers: 0
```

With this information, we can manually access the proper chunk of the input data. In addition, several data iterators provided by `MXNet` already support reading only part of the data. For example,

```
from mxnet.io import ImageRecordIter
data = ImageRecordIter(num_parts=store.num_workers, part_index=store.rank, ...)
```

For whinges or inquiries, [open an issue on GitHub](#).

# Object Detection Using Convolutional Neural Networks

So far, when we've talked about making predictions based on images, we were concerned only with classification. We asked questions like is this digit a "0", "1", ..., or "9?" or, does this picture depict a "cat" or a "dog"? Object detection is a more challenging task. Here our goal is not only to say *what* is in the image but also to recognize *where* it is in the image. As an example, consider the following image, which depicts two dogs and a cat together with their locations.



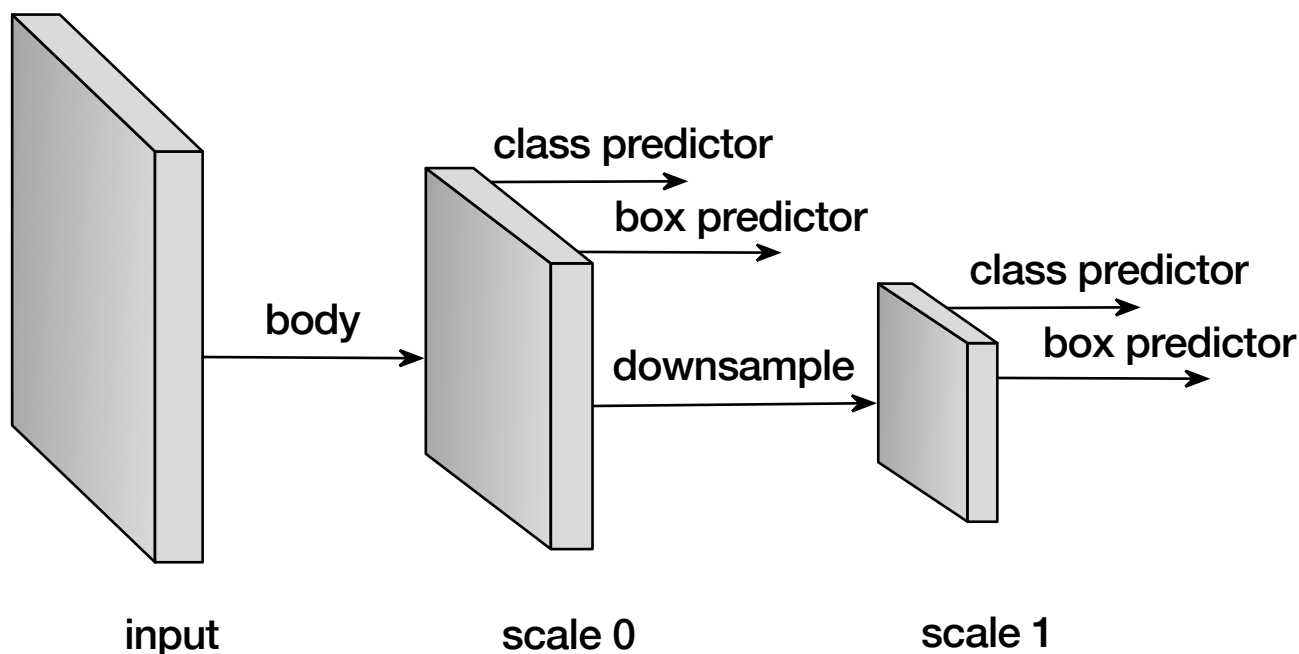
So object differs from image classification in a few ways. First, while a classifier outputs a single category per image, an object detector must be able to recognize multiple objects in a single image. Technically, this task is called *multiple object detection*, but most research in the area addresses the multiple object setting, so we'll abuse terminology just a little. Second, while classifiers need only to output probabilities over classes, object detectors must output both probabilities of class membership and also the coordinates that identify the location of the objects.

On this chapter we'll demonstrate the single shot multiple box object detector (SSD), a popular model for object detection that was first described in [this paper](#), and is straightforward to implement in MXNet Gluon.

## SSD: Single Shot MultiBox Detector

The SSD model predicts anchor boxes at multiple scales. The model architecture is illustrated in the following figure.





We first use a `body` network to extract the image features, which are used as the input to the first scale (scale 0). The class labels and the corresponding anchor boxes are predicted by `class_predictor` and `box_predictor`, respectively. We then downsample the representations to the next scale (scale 1). Again, at this new resolution, we predict both classes and anchor boxes. This downsampling and predicting routine can be repeated in multiple times to obtain results on multiple resolution scales. Let's walk through the components one by one in a bit more detail.

## Default anchor boxes

Since an anchor box can have arbitrary shape, we sample a set of anchor boxes as the candidate. In particular, for each pixel, we sample multiple boxes centered at this pixel but have various sizes and ratios. Assume the input size is  $w \times h$ , - for size  $s \in (0, 1]$ , the generated box shape will be  $ws \times hs$  - for ratio  $r > 0$ , the generated box shape will be  $w\sqrt{r} \times \frac{h}{\sqrt{r}}$

We can sample the boxes using the operator `MultiBoxPrior`. It accepts  $n$  sizes and  $m$  ratios to generate  $n+m-1$  boxes for each pixel. The first  $i$  boxes are generated from `sizes[i], ratios[0]` if  $i \leq n$  otherwise `sizes[0], ratios[i-n]`.

```
In [1]: import mxnet as mx
        from mxnet import nd
        from mxnet.contrib.ndarray import MultiBoxPrior

        n = 40
        # shape: batch x channel x height x weight
        x = nd.random_uniform(shape=(1, 3, n, n))

        y = MultiBoxPrior(x, sizes=[.5, .25, .1], ratios=[1, 2, .5])

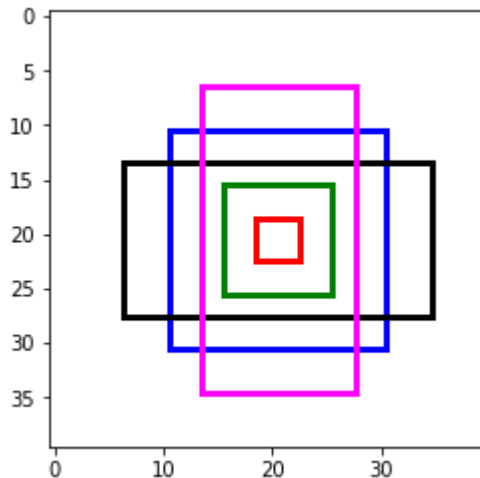
        # the first anchor box generated for pixel at (20,20)
        # its format is (x_min, y_min, x_max, y_max)
```

```
boxes = y.reshape((n, n, -1, 4))
print('The first anchor box at row 21, column 21:', boxes[20, 20, 0, :])
```

```
The first anchor box at row 21, column 21:
[ 0.26249999  0.26249999  0.76249999  0.76249999]
<NDArray 4 @cpu(0)>
```

We can visualize all anchor boxes generated for one pixel on a certain size feature map.

```
In [2]: import matplotlib.pyplot as plt
def box_to_rect(box, color, linewidth=3):
    """convert an anchor box to a matplotlib rectangle"""
    box = box.asnumpy()
    return plt.Rectangle(
        (box[0], box[1]), (box[2]-box[0]), (box[3]-box[1]),
        fill=False, edgecolor=color, linewidth=linewidth)
colors = ['blue', 'green', 'red', 'black', 'magenta']
plt.imshow(nd.ones((n, n, 3)).asnumpy())
anchors = boxes[20, 20, :, :]
for i in range(anchors.shape[0]):
    plt.gca().add_patch(box_to_rect(anchors[i,:]*n, colors[i]))
plt.show()
```



## Predict classes

For each anchor box, we want to predict the associated class label. We make this prediction by using a convolution layer. We choose a kernel of size  $3 \times 3$  with padding size  $(1, 1)$  so that the output will have the same width and height as the input. The confidence scores for the anchor box class labels are stored in channels. In particular, for the  $i$ -th anchor box:

- channel `i*(num_class+1)` store the scores for this box contains only background
- channel `i*(num_class+1)+j` store the scores for this box contains an object from the  $j$ -th class

```
In [3]: from mxnet.gluon import nn
def class_predictor(num_anchors, num_classes):
    """return a layer to predict classes"""
    return nn.Conv2D(num_anchors * (num_classes + 1), 3, padding=1)

cls_pred = class_predictor(5, 10)
```

```
cls_pred.initialize()
x = nd.zeros((2, 3, 20, 20))
print('Class prediction', cls_pred(x).shape)
```

Class prediction (2, 55, 20, 20)

## Predict anchor boxes

The goal is predict how to transfer the current anchor box to the correct box. That is, assume  $b$  is one of the sampled default box, while  $Y$  is the ground truth, then we want to predict the delta positions  $\Delta(Y, b)$ , which is a 4-length vector.

More specifically, the we define the delta vector as:  $[t_x, t_y, t_{width}, t_{height}]$ , where

- $t_x = (Y_x - b_x)/b_{width}$
- $t_y = (Y_y - b_y)/b_{height}$
- $t_{width} = (Y_{width} - b_{width})/b_{width}$
- $t_{height} = (Y_{height} - b_{height})/b_{height}$

Normalizing the deltas with box width/height tends to result in better convergence behavior.

Similar to classes, we use a convolution layer here. The only difference is that the output channel size is now `num_anchors * 4`, with the predicted delta positions for the  $i$ -th box stored from channel `i*4` to `i*4+3`.

```
In [4]: def box_predictor(num_anchors):
        """return a layer to predict delta locations"""
        return nn.Conv2D(num_anchors * 4, 3, padding=1)

box_pred = box_predictor(10)
box_pred.initialize()
x = nd.zeros((2, 3, 20, 20))
print('Box prediction', box_pred(x).shape)
```

Box prediction (2, 40, 20, 20)

## Down-sample features

Each time, we downsample the features by half. This can be achieved by a simple pooling layer with pooling size 2. We may also stack two convolution, batch normalization and ReLU blocks before the pooling layer to make the network deeper.

```
In [5]: def down_sample(num_filters):
        """stack two Conv-BatchNorm-Relu blocks and then a pooling layer
        to halve the feature size"""
        out = nn.HybridSequential()
        for _ in range(2):
            out.add(nn.Conv2D(num_filters, 3, strides=1, padding=1))
            out.add(nn.BatchNorm(in_channels=num_filters))
            out.add(nn.Activation('relu'))
```

```

        out.add(nn.MaxPool2D(2))
        return out

blk = down_sample(10)
blk.initialize()
x = nd.zeros((2, 3, 20, 20))
print('Before', x.shape, 'after', blk(x).shape)

```

Before (2, 3, 20, 20) after (2, 10, 10, 10)

## Manage predictions from multiple layers

A key property of SSD is that predictions are made at multiple layers with shrinking spatial size. Thus, we have to handle predictions from multiple feature layers. One idea is to concatenate them along convolutional channels, with each one predicting a corresponding value(class or box) for each default anchor. We give class predictor as an example, and box predictor follows the same rule.

```

In [6]: # a certain feature map with 20x20 spatial shape
feat1 = nd.zeros((2, 8, 20, 20))
print('Feature map 1', feat1.shape)
cls_pred1 = class_predictor(5, 10)
cls_pred1.initialize()
y1 = cls_pred1(feat1)
print('Class prediction for feature map 1', y1.shape)
# down-sample
ds = down_sample(16)
ds.initialize()
feat2 = ds(feat1)
print('Feature map 2', feat2.shape)
cls_pred2 = class_predictor(3, 10)
cls_pred2.initialize()
y2 = cls_pred2(feat2)
print('Class prediction for feature map 2', y2.shape)

```

Feature map 1 (2, 8, 20, 20)  
 Class prediction for feature map 1 (2, 55, 20, 20)  
 Feature map 2 (2, 16, 10, 10)  
 Class prediction for feature map 2 (2, 33, 10, 10)

```

In [7]: def flatten_prediction(pred):
        return nd.flatten(nd.transpose(pred, axes=(0, 2, 3, 1)))

def concat_predictions(preds):
    return nd.concat(*preds, dim=1)

flat_y1 = flatten_prediction(y1)
print('Flatten class prediction 1', flat_y1.shape)
flat_y2 = flatten_prediction(y2)
print('Flatten class prediction 2', flat_y2.shape)
print('Concat class predictions', concat_predictions([flat_y1, flat_y2]).shape)

```

Flatten class prediction 1 (2, 22000)  
 Flatten class prediction 2 (2, 3300)  
 Concat class predictions (2, 25300)

## Body network

The body network is used to extract features from the raw pixel inputs. Common choices follow the architectures of the state-of-the-art convolution neural networks for image classification. For demonstration purpose, we just stack several down sampling blocks to form the body network.

```
In [8]: from mxnet import gluon
def body():
    """return the body network"""
    out = nn.HybridSequential()
    for nfilters in [16, 32, 64]:
        out.add(down_sample(nfilters))
    return out

bnet = body()
bnet.initialize()
x = nd.zeros((2, 3, 256, 256))
print('Body network', [y.shape for y in bnet(x)])
```

Body network [(64, 32, 32), (64, 32, 32)]

## Create a toy SSD model

Now, let's create a toy SSD model that takes images of resolution  $256 \times 256$  as input.

```
In [9]: def toy_ssd_model(num_anchors, num_classes):
    """return SSD modules"""
    downsamples = nn.Sequential()
    class_preds = nn.Sequential()
    box_preds = nn.Sequential()

    downsamples.add(down_sample(128))
    downsamples.add(down_sample(128))
    downsamples.add(down_sample(128))

    for scale in range(5):
        class_preds.add(class_predictor(num_anchors, num_classes))
        box_preds.add(box_predictor(num_anchors))

    return body(), downsamples, class_preds, box_preds

print(toy_ssd_model(5, 2))
```

```
(HybridSequential(
  (0): HybridSequential(
    (0): Conv2D(16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=16)
    (2): Activation(relu)
    (3): Conv2D(16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=16)
    (5): Activation(relu)
    (6): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
  )
  (1): HybridSequential(
    (0): Conv2D(32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=32)
    (2): Activation(relu)
    (3): Conv2D(32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=32)
    (5): Activation(relu)
    (6): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
  )
  (2): HybridSequential(
    (0): Conv2D(64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```

        (1): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=64)
        (2): Activation(relu)
        (3): Conv2D(64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=64)
        (5): Activation(relu)
        (6): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
    )
), Sequential(
  (0): HybridSequential(
    (0): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=128)
    (2): Activation(relu)
    (3): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=128)
    (5): Activation(relu)
    (6): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
  )
  (1): HybridSequential(
    (0): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=128)
    (2): Activation(relu)
    (3): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=128)
    (5): Activation(relu)
    (6): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
  )
  (2): HybridSequential(
    (0): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=128)
    (2): Activation(relu)
    (3): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm(fix_gamma=False, axis=1, momentum=0.9, eps=1e-05, in_channels=128)
    (5): Activation(relu)
    (6): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
  )
), Sequential(
  (0): Conv2D(15, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): Conv2D(15, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): Conv2D(15, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): Conv2D(15, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): Conv2D(15, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
), Sequential(
  (0): Conv2D(20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): Conv2D(20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): Conv2D(20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): Conv2D(20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): Conv2D(20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
))

```

## Forward

Given an input and the model, we can run the forward pass.

```

In [10]: def toy_ssd_forward(x, body, downsamples, class_preds, box_preds, sizes, ratios):
          # extract feature with the body network
          x = body(x)

          # for each scale, add anchors, box and class predictions,
          # then compute the input to next scale
          default_anchors = []
          predicted_boxes = []
          predicted_classes = []

          for i in range(5):
              default_anchors.append(MultiBoxPrior(x, sizes=sizes[i], ratios=ratios[i]))
              predicted_boxes.append(flatten_prediction(box_preds[i](x)))
              predicted_classes.append(flatten_prediction(class_preds[i](x)))
              if i < 3:

```

```

        x = downsamples[i](x)
    elif i == 3:
        # simply use the pooling layer
        x = nd.Pooling(x, global_pool=True, pool_type='max', kernel=(4, 4))

    return default_anchors, predicted_classes, predicted_boxes

```

## Put all things together

```

In [11]: from mxnet import gluon
class ToySSD(gluon.Block):
    def __init__(self, num_classes, **kwargs):
        super(ToySSD, self).__init__(**kwargs)
        # anchor box sizes for 4 feature scales
        self.anchor_sizes = [[.2, .272], [.37, .447], [.54, .619], [.71, .79], [.88,
.961]]
        # anchor box ratios for 4 feature scales
        self.anchor_ratios = [[1, 2, .5]] * 5
        self.num_classes = num_classes

    with self.name_scope():
        self.body, self.downsamples, self.class_preds, self.box_preds =
toy_ssd_model(4, num_classes)

    def forward(self, x):
        default_anchors, predicted_classes, predicted_boxes = toy_ssd_forward(x,
self.body, self.downsamples,
        self.class_preds, self.box_preds, self.anchor_sizes, self.anchor_ratios)
        # we want to concatenate anchors, class predictions, box predictions from
different layers
        anchors = concat_predictions(default_anchors)
        box_preds = concat_predictions(predicted_boxes)
        class_preds = concat_predictions(predicted_classes)
        # it is better to have class predictions reshaped for softmax computation
        class_preds = nd.reshape(class_preds, shape=(0, -1, self.num_classes + 1))

        return anchors, class_preds, box_preds

```

## Outputs of ToySSD

```

In [12]: # instantiate a ToySSD network with 10 classes
net = ToySSD(2)
net.initialize()
x = nd.zeros((1, 3, 256, 256))
default_anchors, class_predictions, box_predictions = net(x)
print('Outputs:', 'anchors', default_anchors.shape, 'class prediction',
class_predictions.shape, 'box prediction', box_predictions.shape)

Outputs: anchors (1, 5444, 4) class prediction (1, 5444, 3) box prediction (1, 21776)

```

## Dataset

For demonstration purposes, we'll build a train our model to detect Pikachu in the wild. We generated a a synthetic toy dataset by rendering images from open-sourced 3D Pikachu models. The dataset consists of 1000 pikachus with random pose/scale/position in random background images. The exact locations are recorded as ground-truth for training and validation.



## Download dataset

```
In [13]: from mxnet.test_utils import download
import os.path as osp
def verified(file_path, sha1hash):
    import hashlib
    sha1 = hashlib.sha1()
    with open(file_path, 'rb') as f:
        while True:
            data = f.read(1048576)
            if not data:
                break
            sha1.update(data)
    matched = sha1.hexdigest() == sha1hash
    if not matched:
        print('Found hash mismatch in file {}, possibly due to incomplete
download.'.format(file_path))
    return matched

url_format = 'https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/pikachu/{}'
hashes = {'train.rec': 'e6bcb6ffba1ac04ff8a9b1115e650af56ee969c8',
          'train.idx': 'dcf7318b2602c06428b9988470c731621716c393',
          'val.rec': 'd6c33f799b4d058e82f2cb5bd9a976f69d72d520'}
for k, v in hashes.items():
    fname = 'pikachu_' + k
    target = osp.join('data', fname)
    url = url_format.format(k)
    if not osp.exists(target) or not verified(target, v):
        print('Downloading', target, url)
        download(url, fname=fname, dirname='data', overwrite=True)
```



## Load dataset

```
In [14]: import mxnet.image as image
data_shape = 256
batch_size = 32
def get_iterators(data_shape, batch_size):
    class_names = ['pikachu']
    num_class = len(class_names)
    train_iter = image.ImageDetIter(
        batch_size=batch_size,
        data_shape=(3, data_shape, data_shape),
        path_imgrec='./data/pikachu_train.rec',
        path_imgidx='./data/pikachu_train.idx',
        shuffle=True,
        mean=True,
        rand_crop=1,
        min_object_covered=0.95,
        max_attempts=200)
    val_iter = image.ImageDetIter(
        batch_size=batch_size,
        data_shape=(3, data_shape, data_shape),
        path_imgrec='./data/pikachu_val.rec',
        shuffle=False,
        mean=True)
    return train_iter, val_iter, class_names, num_class

train_data, test_data, class_names, num_class = get_iterators(data_shape, batch_size)
batch = train_data.next()
print(batch)
```

DataBatch: data shapes: [(32, 3, 256, 256)] label shapes: [(32, 1, 5)]

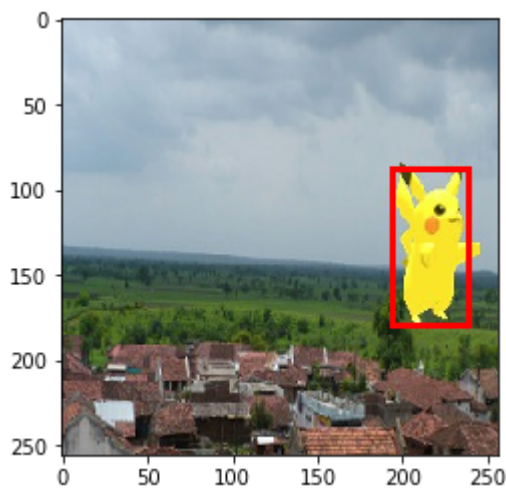
## Illustration

Let's display one image loaded by ImageDetIter.

```
In [15]: import numpy as np

img = batch.data[0][0].asnumpy() # grab the first image, convert to numpy array
img = img.transpose((1, 2, 0)) # we want channel to be the last dimension
img += np.array([123, 117, 104])
img = img.astype(np.uint8) # use uint8 (0-255)
# draw bounding boxes on image
for label in batch.label[0][0].asnumpy():
    if label[0] < 0:
        break
    print(label)
    xmin, ymin, xmax, ymax = [int(x * data_shape) for x in label[1:5]]
    rect = plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin, fill=False, edgecolor=(1,
0, 0), linewidth=3)
    plt.gca().add_patch(rect)
plt.imshow(img)
plt.show()
```

[ 0. 0.75724518 0.34316057 0.93332517 0.70017999]



## Train

### Losses

Network predictions will be penalized for incorrect class predictions and wrong box deltas.

```
In [16]: from mxnet.contrib.ndarray import MultiBoxTarget
def training_targets(default_anchors, class_predicts, labels):
    class_predicts = nd.transpose(class_predicts, axes=(0, 2, 1))
    z = MultiBoxTarget(*[default_anchors, labels, class_predicts])
    box_target = z[0] # box offset target for (x, y, width, height)
    box_mask = z[1] # mask is used to ignore box offsets we don't want to penalize, e.g.
                    # negative samples
    cls_target = z[2] # cls_target is an array of labels for all anchors boxes
    return box_target, box_mask, cls_target
```

Pre-defined losses are provided in `gluon.loss` package, however, we can define losses manually.

First, we need a Focal Loss for class predictions.

```
In [17]: class FocalLoss(gluon.loss.Loss):
    def __init__(self, axis=-1, alpha=0.25, gamma=2, batch_axis=0, **kwargs):
        super(FocalLoss, self).__init__(None, batch_axis, **kwargs)
        self._axis = axis
        self._alpha = alpha
        self._gamma = gamma

    def hybrid_forward(self, F, output, label):
        output = F.softmax(output)
        pt = F.pick(output, label, axis=self._axis, keepdims=True)
        loss = -self._alpha * ((1 - pt) ** self._gamma) * F.log(pt)
        return F.mean(loss, axis=self._batch_axis, exclude=True)

# cls_loss = gluon.loss.SoftmaxCrossEntropyLoss()
cls_loss = FocalLoss()
print(cls_loss)

FocalLoss(batch_axis=0, w=None)
```

Next, we need a SmoothL1Loss for box predictions.

```
In [18]: class SmoothL1Loss(gluon.loss.Loss):
def __init__(self, batch_axis=0, **kwargs):
    super(SmoothL1Loss, self).__init__(None, batch_axis, **kwargs)

def hybrid_forward(self, F, output, label, mask):
    loss = F.smooth_l1((output - label) * mask, scalar=1.0)
    return F.mean(loss, self._batch_axis, exclude=True)

box_loss = SmoothL1Loss()
print(box_loss)
```

```
SmoothL1Loss(batch_axis=0, w=None)
```

## Evaluation metrics

Here, we define two metrics that we'll use to evaluate our performance when training. You're already familiar with accuracy unless you've been naughty and skipped straight to object detection. We use the accuracy metric to assess the quality of the class predictions. Mean absolute error (MAE) is just the L1 distance, introduced in our [linear algebra chapter](#). We use this to determine how close the coordinates of the predicted bounding boxes are to the ground-truth coordinates. Because we are jointly solving both a classification problem and a regression problem, we need an appropriate metric for each task.

```
In [19]: cls_metric = mx.metric.Accuracy()
box_metric = mx.metric.MAE() # measure absolute difference between prediction and target
```

```
In [20]: ### Set context for training
ctx = mx.gpu() # it may takes too Long to train using CPU
try:
    _ = nd.zeros(1, ctx=ctx)
    # pad label for cuda implementation
    train_data.reshape(label_shape=(3, 5))
    train_data = test_data.sync_label_shape(train_data)
except mx.base.MXNetError as err:
    print('No GPU enabled, fall back to CPU, sit back and be patient...')
    ctx = mx.cpu()
```

## Initialize parameters

```
In [21]: net = ToySSD(num_class)
net.initialize(mx.init.Xavier(magnitude=2), ctx=ctx)
```

## Set up trainer

```
In [22]: net.collect_params().reset_ctx(ctx)
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1, 'wd': 5e-4})
```

## Start training

Optionally we load pretrained model for demonstration purpose. One can set

`from_scratch = True` to training from scratch, which may take more than 30 mins to finish using a single capable GPU.

```
In [23]: epochs = 150 # set larger to get better performance
log_interval = 20
from_scratch = False # set to True to train from scratch
if from_scratch:
    start_epoch = 0
else:
    start_epoch = 148
    pretrained = 'ssd_pretrained.params'
    sha1 = 'fbb7d872d76355fff1790d864c2238decdb452bc'
    url = 'https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/models/ssd_pikachu-
fbb7d872.params'
    if not osp.exists(pretrained) or not verified(pretrained, sha1):
        print('Downloading', pretrained, url)
        download(url, fname=pretrained, overwrite=True)
    net.load_params(pretrained, ctx)
```

```
In [24]: import time
from mxnet import autograd as ag
for epoch in range(start_epoch, epochs):
    # reset iterator and tick
    train_data.reset()
    cls_metric.reset()
    box_metric.reset()
    tic = time.time()
    # iterate through all batch
    for i, batch in enumerate(train_data):
        btic = time.time()
        # record gradients
        with ag.record():
            x = batch.data[0].as_in_context(ctx)
            y = batch.label[0].as_in_context(ctx)
            default_anchors, class_predictions, box_predictions = net(x)
            box_target, box_mask, cls_target = training_targets(default_anchors,
class_predictions, y)
            # losses
            loss1 = cls_loss(class_predictions, cls_target)
            loss2 = box_loss(box_predictions, box_target, box_mask)
            # sum all losses
            loss = loss1 + loss2
            # backpropagate
            loss.backward()
        # apply
        trainer.step(batch_size)
        # update metrics
        cls_metric.update([cls_target], [nd.transpose(class_predictions, (0, 2, 1))])
        box_metric.update([box_target], [box_predictions * box_mask])
        if (i + 1) % log_interval == 0:
            name1, val1 = cls_metric.get()
            name2, val2 = box_metric.get()
            print('[Epoch %d Batch %d] speed: %f samples/s, training: %s=%f, %s=%f'
                %(epoch, i, batch_size/(time.time()-btic), name1, val1, name2, val2))

    # end of epoch Logging
    name1, val1 = cls_metric.get()
    name2, val2 = box_metric.get()
    print('[Epoch %d] training: %s=%f, %s=%f'%(epoch, name1, val1, name2, val2))
    print('[Epoch %d] time cost: %f'%(epoch, time.time()-tic))

# we can save the trained parameters to disk
net.save_params('ssd_%d.params' % epochs)
```

```
[Epoch 148 Batch 19] speed: 109.217423 samples/s, training: accuracy=0.997539,
mae=0.001862
```

```
[Epoch 148] training: accuracy=0.997610, mae=0.001806
```

```
[Epoch 148] time cost: 17.762958
[Epoch 149 Batch 19] speed: 110.492729 samples/s, training: accuracy=0.997607,
mae=0.001824
[Epoch 149] training: accuracy=0.997692, mae=0.001789
[Epoch 149] time cost: 15.353258
```

## Test

Testing is similar to training, except that we don't need to compute gradients and training targets. Instead, we take the predictions from network output, and combine them to get the real detection output.

## Prepare the test data

```
In [25]: import numpy as np
import cv2
def preprocess(image):
    """Takes an image and apply preprocess"""
    # resize to data_shape
    image = cv2.resize(image, (data_shape, data_shape))
    # swap BGR to RGB
    image = image[:, :, (2, 1, 0)]
    # convert to float before subtracting mean
    image = image.astype(np.float32)
    # subtract mean
    image -= np.array([123, 117, 104])
    # organize as [batch-channel-height-width]
    image = np.transpose(image, (2, 0, 1))
    image = image[np.newaxis, :]
    # convert to ndarray
    image = nd.array(image)
    return image

image = cv2.imread('img/pikachu.jpg')
x = preprocess(image)
print('x', x.shape)

x (1, 3, 256, 256)
```

## Network inference

In a single line of code!

```
In [26]: # if pre-trained model is provided, we can load it
# net.load_params('ssd_%d.params' % epochs, ctx)
anchors, cls_preds, box_preds = net(x.as_in_context(ctx))
print('anchors', anchors)
print('class predictions', cls_preds)
print('box delta predictions', box_preds)

anchors
[[[-0.084375 -0.084375 0.115625 0.115625 ]
 [-0.12037501 -0.12037501 0.15162501 0.15162501]
 [-0.12579636 -0.05508568 0.15704636 0.08633568]
 ...,
 [ 0.01949999 0.01949999 0.98049998 0.98049998]
 [-0.12225395 0.18887302 1.12225389 0.81112695]
 [ 0.18887302 -0.12225395 0.81112695 1.12225389]]]
<NDArray 1x5444x4 @gpu(0)>
class predictions
```

```
[[[ 0.33754104 -1.64660323]
 [ 1.15297699 -1.77257478]
 [ 1.1535604 -0.98352218]
 ...,
 [-0.27562004 -1.29400492]
 [ 0.45524898 -0.88782215]
 [ 0.20327765 -0.94481993]]]
<NDArray 1x5444x2 @gpu(0)>
box delta predictions
[[-0.16735925 -0.13083346 -0.68860865 ..., -0.18972112 0.11822788
 -0.27067867]]
<NDArray 1x21776 @gpu(0)>
```

## Convert predictions to real object detection results

```
In [27]: from mxnet.contrib.ndarray import MultiBoxDetection
# convert predictions to probabilities using softmax
cls_probs = nd.SoftmaxActivation(nd.transpose(cls_preds, (0, 2, 1)), mode='channel')
# apply shifts to anchors boxes, non-maximum-suppression, etc...
output = MultiBoxDetection(*[cls_probs, box_preds, anchors], force_suppress=True,
clip=False)
print(output)
```

```
[[[ 0.          0.61178613  0.51807499  0.5042429   0.67325425  0.70118797]
 [-1.          0.59466797  0.52491206  0.50917625  0.66228026  0.70489514]
 [-1.          0.5731774   0.53843218  0.50217044  0.66522425  0.7118448 ]
 ...,
 [-1.         -1.         -1.         -1.         -1.         -1.         ]
 [-1.         -1.         -1.         -1.         -1.         -1.         ]
 [-1.         -1.         -1.         -1.         -1.         -1.         ]]]
<NDArray 1x5444x6 @gpu(0)>
```

Each row in the output corresponds to a detection box, as in format [class\_id, confidence, xmin, ymin, xmax, ymax].

Most of the detection results are -1, indicating that they either have very small confidence scores, or been suppressed through non-maximum-suppression.

## Display results

```
In [28]: def display(img, out, thresh=0.5):
import random
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (10,10)
pens = dict()
plt.clf()
plt.imshow(img)
for det in out:
    cid = int(det[0])
    if cid < 0:
        continue
    score = det[1]
    if score < thresh:
        continue
    if cid not in pens:
        pens[cid] = (random.random(), random.random(), random.random())
    scales = [img.shape[1], img.shape[0]] * 2
    xmin, ymin, xmax, ymax = [int(p * s) for p, s in zip(det[2:6].tolist(), scales)]
    rect = plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin, fill=False,
        edgecolor=pens[cid], linewidth=3)
```

```

plt.gca().add_patch(rect)
text = class_names[cid]
plt.gca().text(xmin, ymin-2, '{:s} {:.3f}'.format(text, score),
               bbox=dict(facecolor=pens[cid], alpha=0.5),
               fontsize=12, color='white')

plt.show()

display(image[:, :, (2, 1, 0)], output[0].asnumpy(), thresh=0.45)

```



## Conclusion

Detection is harder than classification, since we want not only class probabilities, but also localizations of different objects including potential small objects. Using sliding window together with a good classifier might be an option, however, we have shown that with a properly designed convolutional neural network, we can do single shot detection which is blazing fast and accurate!

For whinges or inquiries, [open an issue on GitHub](#).

## Transferring knowledge through finetuning

In previous chapters, we demonstrated how to train a neural network to recognize the categories corresponding to objects in images. We looked at toy datasets like hand-written digits, and thumbnail-sized pictures of animals. And we talked about the ImageNet dataset, the default academic benchmark, which contains 1M million images, 1000 each from 1000 separate classes.

The ImageNet dataset categorically changed what was possible in computer vision. It turns out some things are possible (these days, even easy) on gigantic datasets, that simply aren't with smaller datasets. In fact, we don't know of any technique that can comparably powerful model on a similar photograph dataset but containing only, say, 10k images.

And that's a problem. Because however impressive the results of CNNs on ImageNet may be, most people aren't interested in ImageNet itself. They're interested in their own problems. Recognize people based on pictures of their faces. Distinguish between photographs of 10 different types of corral on the ocean floor. Usually when individuals (and not Amazon, Google, or inter-institutional *big science* initiatives) are interested in solving a computer vision problem, they come to the table with modestly sized datasets. A few hundred examples may be common and a few thousand examples may be as much as you can reasonably ask for.

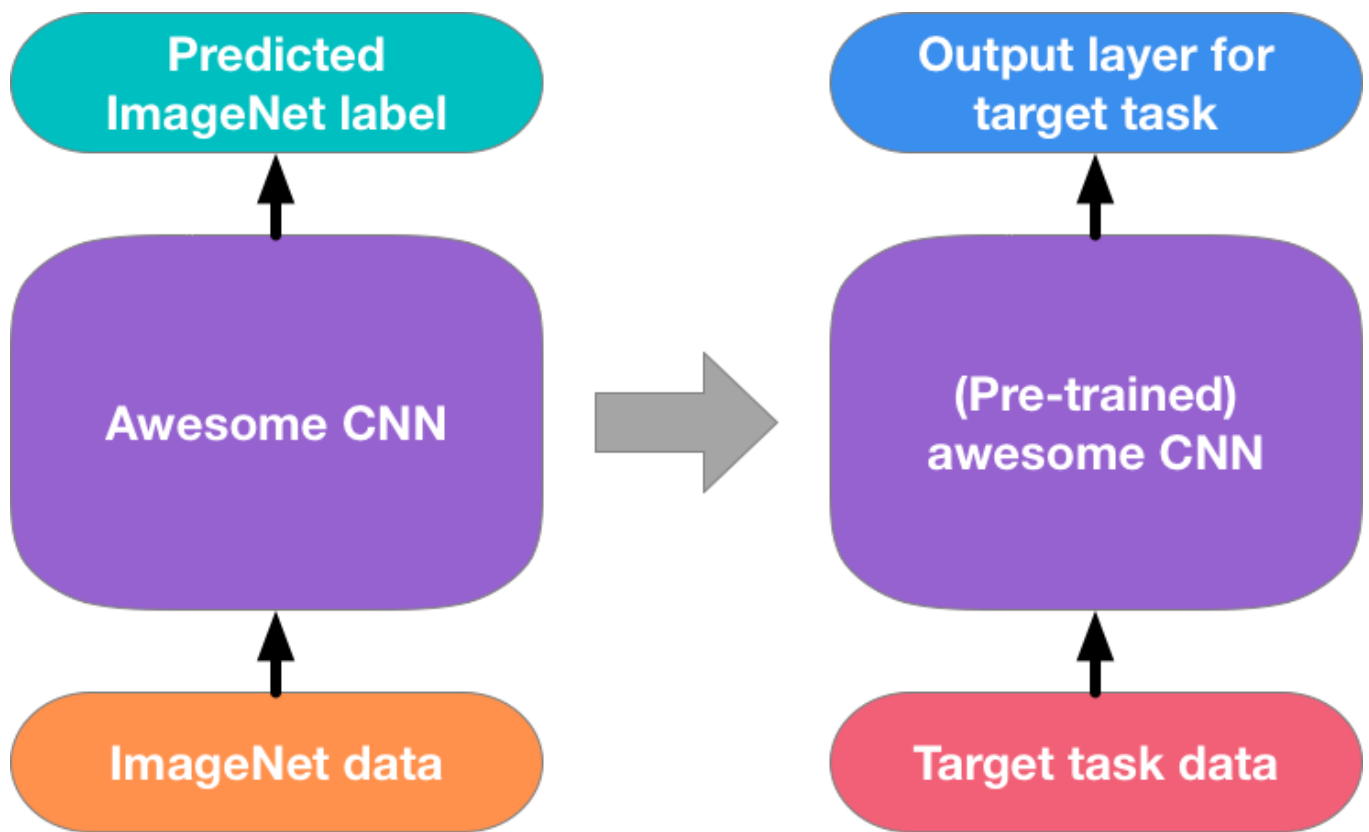
So one natural question emerges. Can we somehow use the powerful models trained on millions of examples for one dataset, and apply them to improve performance on a new problem with a much smaller dataset? This kind of problem (learning on source dataset, bringing knowledge to target dataset), is appropriately called *transfer learning*. Fortunately, we have some effective tools for solving this problem.

For deep neural networks, the most popular approach is called finetuning and the idea is both simple and effective:

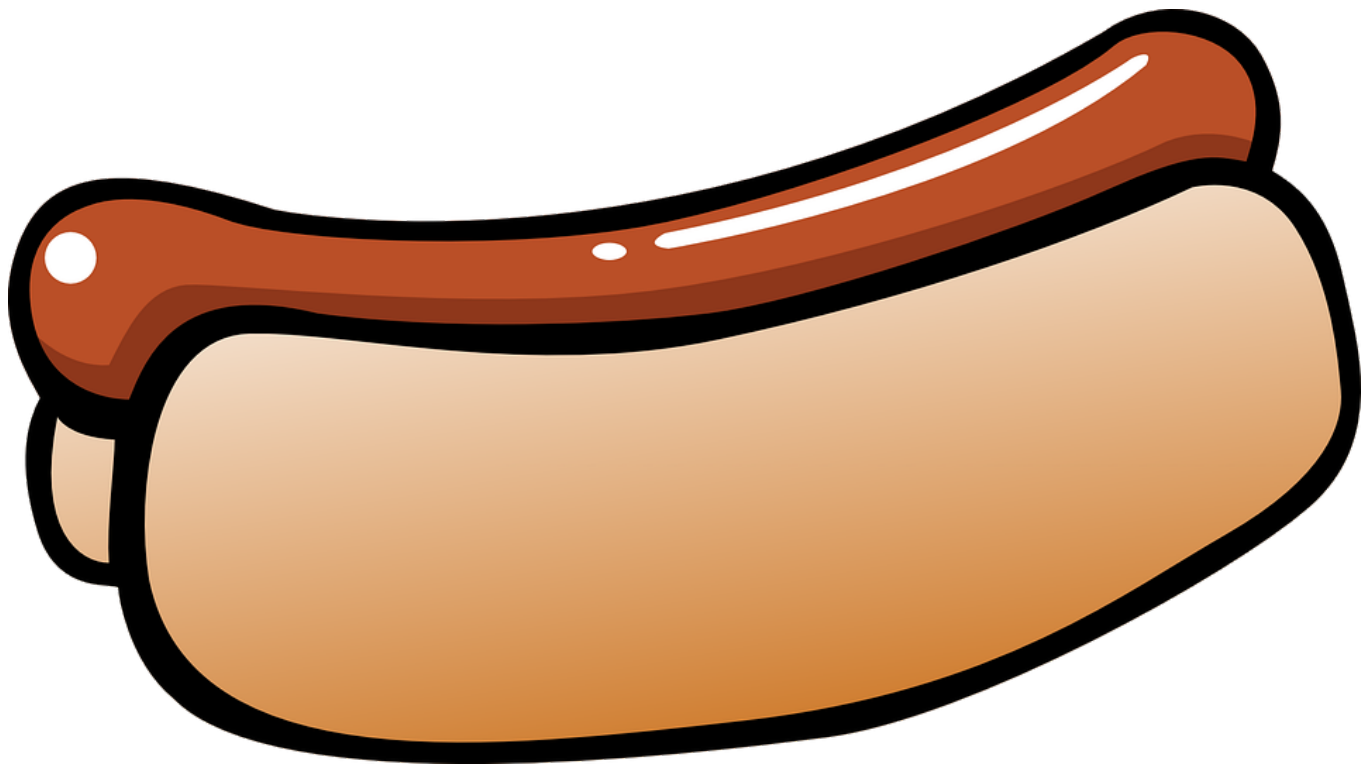
- Train a neural network on the source task  $S$ .
- Decapitate it, replacing it's output layer appropriate to target task  $T$ .
- Initialize the weights on the new output layer randomly, keeping all other (pretrained) weights the same.
- Begin training on the new dataset.

This might be clearer if we visualize the algorithm:





In this section, we'll demonstrate fine-tuning, using the popular and compact SqueezeNet architecture. Since we don't want to saddle you with the burden of downloading ImageNet, or of training on ImageNet from scratch, we'll pull the weights of the pretrained Squeeze net from the internet. Specifically, we'll be fine-tuning a squeezenet-1.1 that was pre-trained on imagenet-12. Finally, we'll fine-tune it to recognize **hotdogs**.



*hot dog*

We'll start with the obligatory ritual of importing a bunch of stuff that you'll need later.

```
In [2]: %pylab inline
        pylab.rcParams['figure.figsize'] = (10, 6)
```

Populating the interactive namespace from numpy and matplotlib

## Settings

We'll set a few settings up here that you can configure later to manipulate the behavior of the algorithm. These are mostly familiar. Hybrid mode, uses the just in time compiler described in [our chapter on high performance training](#) to make the network much faster to train. Since we're not working with any crazy dynamic graphs that can't be compiled, there's not reason not to hybridize. The batch size, number of training epochs, weight decay, and learning rate should all be familiar by now. The positive class weight, says how much more we should upweight the importance of positive instances (photos of hot dogs) in the objective function. We use this to combat the extreme class imbalance (not surprisingly, most pictures do not depict hot dogs).

```
In [3]: # Demo mode uses the validation dataset for training, which is smaller and faster to
        train.
        demo = True
        log_interval = 100
        gpus = 0

        # Options are imperative or hybrid. Use hybrid for better performance.
        mode = 'hybrid'

        # training hyperparameters
        batch_size = 256
        if demo:
            epochs = 5
            learning_rate = 0.02
            wd = 0.002
        else:
            epochs = 40
            learning_rate = 0.05
            wd = 0.002

        # the class weight for hotdog class to help the imbalance problem.
        positive_class_weight = 5
```

```
In [4]: from __future__ import print_function
        import logging
        logging.basicConfig(level=logging.INFO)
        import os
        import time
        from collections import OrderedDict
        import skimage.io as io

        import mxnet as mx
        from mxnet.test_utils import download
        mx.random.seed(127)
```

## Dataset

Formally, hot dog recognition is a binary classification problem. We'll use 1 to represent the hotdog class, and 0 for the *not hotdog* class. Our hot dog dataset (the target dataset which we'll fine-tune the model to) contains 18,141 sample images, 2091 of which are hotdogs. Because the dataset is imbalanced (e.g. hotdog class is only 1% in mscoco dataset), sampling interesting negative samples can help to improve the performance of our algorithm. Thus, in the negative class in the our dataset, two thirds are images from food categories (e.g. pizza) other than hotdogs, and 30% are images from all other categories.

## Files

We prepare the dataset in the format of MXRecord using [im2rec](#) tool. As of the current draft, rec files are not yet explained in the book, but if you're reading after November or December 2017 and you still see this note, [open an issue on GitHub](#) and let us know to stop slacking off.

- not\_hotdog\_train.rec 641M (1882 positive, 10000 interesting negative, and 5000 random negative)
- not\_hotdog\_validation.rec 49M (209 positive, 700 interesting negative, and 350 random negative)

```
In [4]: dataset_files = {'train': ('not_hotdog_train-e6ef27b4.rec',  
                                '0aad7e1f16f5fb109b719a414a867bbee6ef27b4'),  
                        'validation': ('not_hotdog_validation-c0201740.rec',  
                                      '723ae5f8a433ed2e2bf729baec6b878ac0201740')}
```

To demo the model here, we're just going to use the smaller validation set. But if you're interested in training on the full set, set 'demo' to False in the settings at the beginning. Now we're ready to download and verify the dataset.

```
In [5]: if demo:  
        training_dataset, training_data_hash = dataset_files['validation']  
    else:  
        training_dataset, training_data_hash = dataset_files['train']  
  
    validation_dataset, validation_data_hash = dataset_files['validation']  
  
    def verified(file_path, sha1hash):  
        import hashlib  
        sha1 = hashlib.sha1()  
        with open(file_path, 'rb') as f:  
            while True:  
                data = f.read(1048576)  
                if not data:  
                    break  
                sha1.update(data)  
        matched = sha1.hexdigest() == sha1hash  
        if not matched:  
            logging.warn('Found hash mismatch in file {}, possibly due to incomplete  
download.'  
                        .format(file_path))  
        return matched  
  
    url_format = 'https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/{}'  
    if not os.path.exists(training_dataset) or not verified(training_dataset,
```

```

training_data_hash):
    logging.info('Downloading training dataset.')
    download(url_format.format(training_dataset),
              overwrite=True)
if not os.path.exists(validation_dataset) or not verified(validation_dataset,
validation_data_hash):
    logging.info('Downloading validation dataset.')
    download(url_format.format(validation_dataset),
              overwrite=True)

```

## Iterators

The record files can be read using [mx.io.ImageRecordIter](#)

```

In [6]: # Load dataset
train_iter = mx.io.ImageRecordIter(path_imgrec=training_dataset,
                                   min_img_size=256,
                                   data_shape=(3, 224, 224),
                                   rand_crop=True,
                                   shuffle=True,
                                   batch_size=batch_size,
                                   max_random_scale=1.5,
                                   min_random_scale=0.75,
                                   rand_mirror=True)
val_iter = mx.io.ImageRecordIter(path_imgrec=validation_dataset,
                                  min_img_size=256,
                                  data_shape=(3, 224, 224),
                                  batch_size=batch_size)

```

## Model

The model we are finetuning is [SqueezeNet](#). Gluon module offers squeezenet v1.0 and v1.1 that are pretrained on ImageNet. This is just a convolutional neural network, with an architecture chosen to have a small number of parameters and to require a minimal amount of computation. It's especially popular for folks that need to run CNNs on low-powered devices like cell phones and other internet-of-things devices.

## Pulling the pre-trained model

Fortunately, MXNet has a model zoo that gives us convenient access to a number of popular models, both their architectures and their pretrained parameters. Let's download SqueezeNet right now with just a few lines of code.

```

In [7]: from mxnet.gluon import nn
        from mxnet.gluon.model_zoo import vision as models

        # get pretrained squeezenet
        net = models.squeezenet1_1(pretrained=True, prefix='deep_dog_')
        # hot dog happens to be a class in imagenet.
        # we can reuse the weight for that class for better performance
        # here's the index for that class for later use
        imagenet_hotdog_index = 713

```

# DeepDog net

We can now use the feature extractor part from the pretrained squeezenet to build our own network. The model zoo, even handles the decaptiation for us. All we have to do is specify the number out of output classes in our new task, which we do via the keyword argument

```
classes=2.
```

```
In [8]: deep_dog_net = models.squeezenet1_1(prefix='deep_dog_', classes=2)
deep_dog_net.collect_params().initialize()
deep_dog_net.features = net.features
print(deep_dog_net)
```

```
SqueezeNet(
  (features): HybridSequential(
    (0): Conv2D(64, kernel_size=(3, 3), stride=(2, 2))
    (1): Activation(relu)
    (2): MaxPool2D(size=(3, 3), stride=(2, 2), padding=(0, 0), ceil_mode=True)
    (3): HybridSequential(
      (0): HybridSequential(
        (0): Conv2D(16, kernel_size=(1, 1), stride=(1, 1))
        (1): Activation(relu)
      )
      (1): HybridConcurrent(
        (0): HybridSequential(
          (0): Conv2D(64, kernel_size=(1, 1), stride=(1, 1))
          (1): Activation(relu)
        )
        (1): HybridSequential(
          (0): Conv2D(64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): Activation(relu)
        )
      )
    )
    (4): HybridSequential(
      (0): HybridSequential(
        (0): Conv2D(16, kernel_size=(1, 1), stride=(1, 1))
        (1): Activation(relu)
      )
      (1): HybridConcurrent(
        (0): HybridSequential(
          (0): Conv2D(64, kernel_size=(1, 1), stride=(1, 1))
          (1): Activation(relu)
        )
        (1): HybridSequential(
          (0): Conv2D(64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): Activation(relu)
        )
      )
    )
    (5): MaxPool2D(size=(3, 3), stride=(2, 2), padding=(0, 0), ceil_mode=True)
    (6): HybridSequential(
      (0): HybridSequential(
        (0): Conv2D(32, kernel_size=(1, 1), stride=(1, 1))
        (1): Activation(relu)
      )
      (1): HybridConcurrent(
        (0): HybridSequential(
          (0): Conv2D(128, kernel_size=(1, 1), stride=(1, 1))
          (1): Activation(relu)
        )
        (1): HybridSequential(
          (0): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): Activation(relu)
        )
      )
    )
  )
)
```

```

(7): HybridSequential(
  (0): HybridSequential(
    (0): Conv2D(32, kernel_size=(1, 1), stride=(1, 1))
    (1): Activation(relu)
  )
  (1): HybridConcurrent(
    (0): HybridSequential(
      (0): Conv2D(128, kernel_size=(1, 1), stride=(1, 1))
      (1): Activation(relu)
    )
    (1): HybridSequential(
      (0): Conv2D(128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): Activation(relu)
    )
  )
)
(8): MaxPool2D(size=(3, 3), stride=(2, 2), padding=(0, 0), ceil_mode=True)
(9): HybridSequential(
  (0): HybridSequential(
    (0): Conv2D(48, kernel_size=(1, 1), stride=(1, 1))
    (1): Activation(relu)
  )
  (1): HybridConcurrent(
    (0): HybridSequential(
      (0): Conv2D(192, kernel_size=(1, 1), stride=(1, 1))
      (1): Activation(relu)
    )
    (1): HybridSequential(
      (0): Conv2D(192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): Activation(relu)
    )
  )
)
(10): HybridSequential(
  (0): HybridSequential(
    (0): Conv2D(48, kernel_size=(1, 1), stride=(1, 1))
    (1): Activation(relu)
  )
  (1): HybridConcurrent(
    (0): HybridSequential(
      (0): Conv2D(192, kernel_size=(1, 1), stride=(1, 1))
      (1): Activation(relu)
    )
    (1): HybridSequential(
      (0): Conv2D(192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): Activation(relu)
    )
  )
)
(11): HybridSequential(
  (0): HybridSequential(
    (0): Conv2D(64, kernel_size=(1, 1), stride=(1, 1))
    (1): Activation(relu)
  )
  (1): HybridConcurrent(
    (0): HybridSequential(
      (0): Conv2D(256, kernel_size=(1, 1), stride=(1, 1))
      (1): Activation(relu)
    )
    (1): HybridSequential(
      (0): Conv2D(256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): Activation(relu)
    )
  )
)
(12): HybridSequential(
  (0): HybridSequential(
    (0): Conv2D(64, kernel_size=(1, 1), stride=(1, 1))
    (1): Activation(relu)
  )
  (1): HybridConcurrent(
    (0): HybridSequential(

```

```

        (0): Conv2D(256, kernel_size=(1,1), stride=(1, 1))
        (1): Activation(relu)
    )
    (1): HybridSequential(
        (0): Conv2D(256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): Activation(relu)
    )
)
)
)
(classifier): HybridSequential(
    (0): Dropout(p = 0.5)
    (1): Conv2D(2, kernel_size=(1, 1), stride=(1, 1))
    (2): Activation(relu)
    (3): AvgPool2D(size=(13, 13), stride=(13, 13), padding=(0, 0), ceil_mode=False)
    (4): Flatten
)
)

```

The network can already be used for prediction. However, since it hasn't been finetuned yet, the network performance could be bad.

```

In [9]: from skimage.color import rgba2rgb

def classify_hotdog(net, url):
    I = io.imread(url)
    if I.shape[2] == 4:
        I = rgba2rgb(I)
    image = mx.nd.array(I).astype(np.uint8)
    plt.subplot(1, 2, 1)
    plt.imshow(image.asnumpy())
    image = mx.image.resize_short(image, 256)
    image, _ = mx.image.center_crop(image, (224, 224))
    plt.subplot(1, 2, 2)
    plt.imshow(image.asnumpy())
    image = mx.image.color_normalize(image.astype(np.float32)/255,
                                     mean=mx.nd.array([0.485, 0.456, 0.406]),
                                     std=mx.nd.array([0.229, 0.224, 0.225]))
    image = mx.nd.transpose(image.astype('float32'), (2,1,0))
    image = mx.nd.expand_dims(image, axis=0)
    out = mx.nd.SoftmaxActivation(net(image))
    print('Probabilities are: '+str(out[0].asnumpy()))
    result = np.argmax(out.asnumpy())
    outstring = ['Not hotdog!', 'Hotdog!']
    print(outstring[result])

```

```

In [10]: classify_hotdog(deep_dog_net, '../img/real_hotdog.jpg')

```

```

Probabilities are: [ 0.84632009  0.15367992]
Not hotdog!

```

## Reuse class weights

As mentioned earlier, in addition to the feature extractor, we can reuse the class weights for hot dog from the pretrained model, since hot dog was already a class in the imagenet. To do that, we need to get the weight from the classifier layers of the pretrained model, find the right slice, and put it into our two-class classifier.

```
In [11]: # Let's examine the classifier and find the last conv layer
print(net.classifier)

HybridSequential(
  (0): Dropout(p = 0.5)
  (1): Conv2D(1000, kernel_size=(1, 1), stride=(1, 1))
  (2): Activation(relu)
  (3): AvgPool2D(size=(13, 13), stride=(13, 13), padding=(0, 0), ceil_mode=False)
  (4): Flatten
)
```

```
In [12]: # the last conv layer is the second layer
pretrained_conv_params = net.classifier[1].params

# weights can then be found from the above parameter dict
pretrained_weight_param = pretrained_conv_params.get('weight')
pretrained_bias_param = pretrained_conv_params.get('bias')

# next, we locate the right slice that we're interested in.
hotdog_w = mx.nd.split(pretrained_weight_param.data().as_in_context(mx.cpu()),
                        1000, axis=0)[imagenet_hotdog_index]
hotdog_b = mx.nd.split(pretrained_bias_param.data().as_in_context(mx.cpu()),
                        1000, axis=0)[imagenet_hotdog_index]

# our classifier is for two classes. here, we reuse the hotdog class weight,
# and randomly initialize the 'not hotdog' class.
new_classifier_w = mx.nd.concat(mx.nd.random_normal(shape=hotdog_w.shape, scale=0.02),
                                hotdog_w,
                                dim=0)
new_classifier_b = mx.nd.concat(mx.nd.random_normal(shape=hotdog_b.shape, scale=0.02),
                                hotdog_b,
                                dim=0)

# finally, we initialize the parameter buffers and set the values.
# since classifier is a HybridSequential/Sequential, the following
# takes the zero-indexed 1-st layer of the classifier
final_conv_layer_params = deep_dog_net.classifier[1].params
```



```
final_conv_layer_params.get('weight').set_data(new_classifier_w)
final_conv_layer_params.get('bias').set_data(new_classifier_b)
```

## Evaluation

Our task is a binary classification problem with imbalanced classes. So we'll monitor performance both using accuracy and F1 score, a metric favored in settings with extreme class imbalance.

[Note to authors: ensure that F1 score is explained earlier or explain it here in full]

```
In [13]: # return metrics string representation
def metric_str(names, accs):
    return ', '.join(['%s=%f'%(name, acc) for name, acc in zip(names, accs)])
metric = mx.metric.create(['acc', 'f1'])
```

The following snippet performs inferences on evaluation dataset, and updates the metrics. Once the evaluation data iterator is exhausted, it returns the values of each of the metrics.

```
In [14]: import mxnet.gluon as gluon
from mxnet.image import color_normalize

def evaluate(net, data_iter, ctx):
    data_iter.reset()
    for batch in data_iter:
        data = color_normalize(batch.data[0]/255,
                               mean=mx.nd.array([0.485, 0.456, 0.406]).reshape((1,3,1,1)),
                               std=mx.nd.array([0.229, 0.224, 0.225]).reshape((1,3,1,1)))
        data = gluon.utils.split_and_load(data, ctx_list=ctx, batch_axis=0)
        label = gluon.utils.split_and_load(batch.label[0], ctx_list=ctx, batch_axis=0)
        outputs = []
        for x in data:
            outputs.append(net(x))
        metric.update(label, outputs)
    out = metric.get()
    metric.reset()
    return out
```

## Training

We now can train the model just as we would any supervised model. In this example, we set up the training loop for multi-GPU use as described from first principles [here](#) and in the context of gluon [here](#).

```
In [15]: import mxnet.autograd as autograd

def train(net, train_iter, val_iter, epochs, ctx):
    if isinstance(ctx, mx.Context):
        ctx = [ctx]
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': learning_rate,
                                                         'wd': wd})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()

    best_f1 = 0
    val_names, val_accs = evaluate(net, val_iter, ctx)
    logging.info('[Initial] validation: %s'%(metric_str(val_names, val_accs)))
    for epoch in range(epochs):
```

```

tic = time.time()
train_iter.reset()
btic = time.time()
for i, batch in enumerate(train_iter):
    # the model zoo models expect normalized images
    data = color_normalize(batch.data[0]/255,
                           mean=mx.nd.array([0.485, 0.456,
0.406])).reshape((1,3,1,1)),
                           std=mx.nd.array([0.229, 0.224,
0.225])).reshape((1,3,1,1))
    data = gluon.utils.split_and_load(data, ctx_list=ctx, batch_axis=0)
    label = gluon.utils.split_and_load(batch.label[0], ctx_list=ctx, batch_axis=0)
    outputs = []
    Ls = []
    with autograd.record():
        for x, y in zip(data, label):
            z = net(x)
            # rescale the loss based on class to counter the imbalance problem
            L = loss(z, y) * (1+y*positive_class_weight)/positive_class_weight
            # store the loss and do backward after we have done forward
            # on all GPUs for better speed on multiple GPUs.
            Ls.append(L)
            outputs.append(z)
        for L in Ls:
            L.backward()
    trainer.step(batch.data[0].shape[0])
    metric.update(label, outputs)
    if log_interval and not (i+1)%log_interval:
        names, accs = metric.get()
        logging.info('[Epoch %d Batch %d] speed: %f samples/s, training: %s'%(
            epoch, i, batch_size/(time.time()-btic), metric_str(names,
accs)))
        btic = time.time()

    names, accs = metric.get()
    metric.reset()
    logging.info('[Epoch %d] training: %s'%(epoch, metric_str(names, accs)))
    logging.info('[Epoch %d] time cost: %f'%(epoch, time.time()-tic))
    val_names, val_accs = evaluate(net, val_iter, ctx)
    logging.info('[Epoch %d] validation: %s'%(epoch, metric_str(val_names, val_accs)))

    if val_accs[1] > best_f1:
        best_f1 = val_accs[1]
        logging.info('Best validation f1 found. Checkpointing...')
        net.save_params('deep-dog-%d.params'%(epoch))

if mode == 'hybrid':
    deep_dog_net.hybridize()
if epochs > 0:
    contexts = [mx.gpu(i) for i in range(gpus)] if gpus > 0 else [mx.cpu()]
    deep_dog_net.collect_params().reset_ctx(contexts)
    train(deep_dog_net, train_iter, val_iter, epochs, contexts)

```

```

INFO:root:[Initial] validation: accuracy=0.185938, f1=0.286732
INFO:root:[Epoch 0] training: accuracy=0.482031, f1=0.256851
INFO:root:[Epoch 0] time cost: 205.866237
INFO:root:[Epoch 0] validation: accuracy=0.612500, f1=0.360263
INFO:root:Best validation f1 found. Checkpointing...
INFO:root:[Epoch 1] training: accuracy=0.530469, f1=0.375600
INFO:root:[Epoch 1] time cost: 170.821589
INFO:root:[Epoch 1] validation: accuracy=0.611719, f1=0.418568
INFO:root:Best validation f1 found. Checkpointing...
INFO:root:[Epoch 2] training: accuracy=0.538281, f1=0.401126
INFO:root:[Epoch 2] time cost: 226.555910
INFO:root:[Epoch 2] validation: accuracy=0.643750, f1=0.446779
INFO:root:Best validation f1 found. Checkpointing...
INFO:root:[Epoch 3] training: accuracy=0.590625, f1=0.423331
INFO:root:[Epoch 3] time cost: 204.861021
INFO:root:[Epoch 3] validation: accuracy=0.735156, f1=0.496402
INFO:root:Best validation f1 found. Checkpointing...
INFO:root:[Epoch 4] training: accuracy=0.640625, f1=0.453261
INFO:root:[Epoch 4] time cost: 175.274520

```

```
INFO:root:[Epoch 4] validation: accuracy=0.810937, f1=0.556719
INFO:root:Best validation f1 found. Checkpointing...
```

## Try it out!

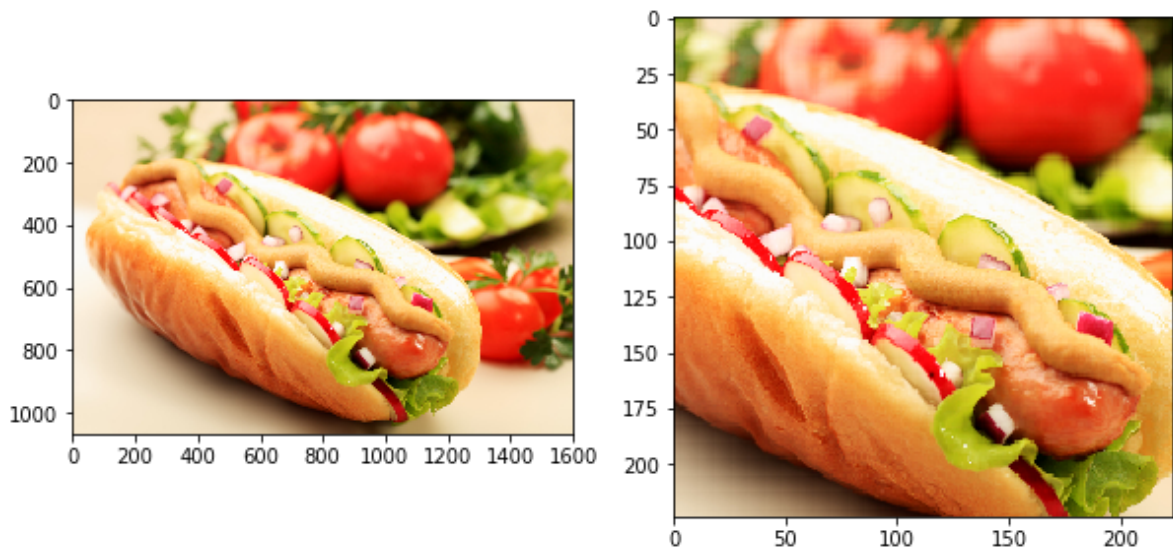
Once our model is trained, we can either use the `deep_dog_net` model in the notebook kernel, or load it from the best checkpoint.

```
In [16]: # Uncomment below line and replace the file name with the last checkpoint.
# deep_dog_net.load_params('deep-dog-4.params', mx.cpu())
#
# Alternatively, you can uncomment the following lines to get the model that we finetuned,
# with validation F1 score of 0.74.
download('https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/models/deep-dog-5a342a6f.params',
        overwrite=True)
deep_dog_net.load_params('deep-dog-5a342a6f.params', mx.cpu())
```

```
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/models/deep-dog-5a342a6f.params into deep-dog-5a342a6f.params successfully
```

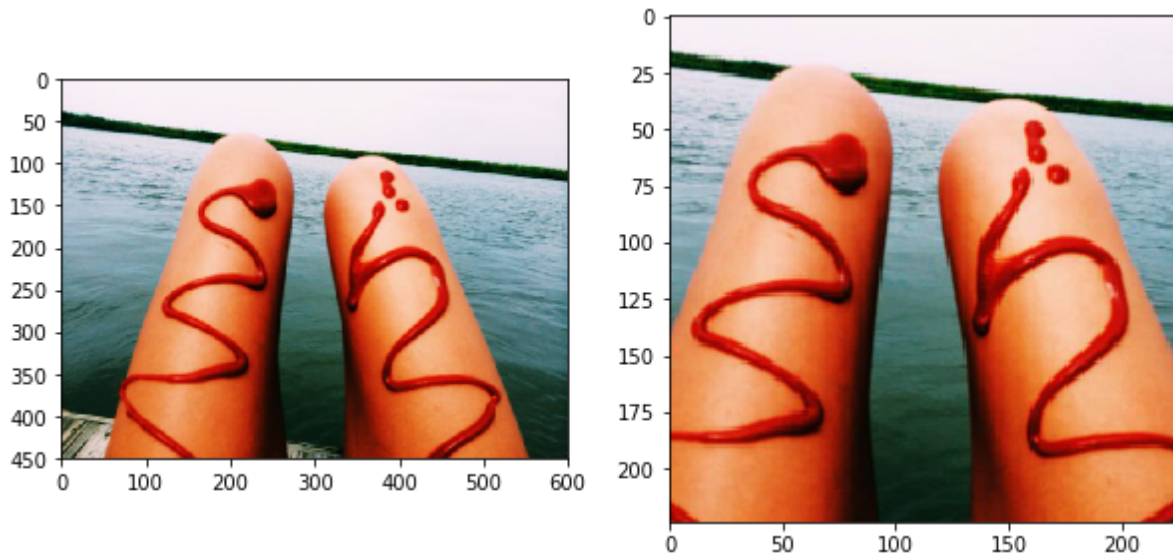
```
In [17]: classify_hotdog(deep_dog_net, '../img/real_hotdog.jpg')
```

```
Probabilities are: [ 0.19303364  0.80696636]
Hotdog!
```



```
In [18]: classify_hotdog(deep_dog_net, '../img/leg_hotdog.jpg')
```

```
Probabilities are: [ 0.92225069  0.07774931]
Not hotdog!
```



```
In [19]: classify_hotdog(deep_dog_net, '../img/dog_hotdog.jpg')
```

```
Probabilities are: [ 0.99648535  0.00351469]
Not hotdog!
```



## Conclusions

As you can see, given a pretrained model, we can get a great classifier, even for tasks where we simply don't have enough data to train from scratch. That's because the representations necessary to perform both tasks have a lot in common. Since they both address natural images, they both require recognizing textures, shapes, edges, etc. Whenever you have a small enough dataset that you fear impoverishing your model, try thinking about what larger datasets you might be able to pre-train your model on, so that you can just perform fine-tuning on the task at hand.

## Next

This section is still changing too fast to say for sure what will come next. Stay tuned!

For whinges or inquiries, [open an issue on GitHub](#).

# Visual Question Answering in gluon

This is a notebook for implementing visual question answering in gluon.

```
In [1]: from __future__ import print_function
import numpy as np
import mxnet as mx
import mxnet.ndarray as F
import mxnet.contrib.ndarray as C
import mxnet.gluon as gluon
from mxnet.gluon import nn
from mxnet import autograd
import bisect
from IPython.core.display import display, HTML
import logging
logging.basicConfig(level=logging.INFO)
import os
from mxnet.test_utils import download
import json
from IPython.display import HTML, display
```

## The VQA dataset

In the VQA dataset, for each sample, there is one image and one question. The label is the answer for the question regarding the image. You can download the VQA1.0 dataset from VQA website.



How many slices of pizza are there?

You need to preprocess the data:

1. Extract the samples from original json files.
2. Filter the samples giving top k answers(k can be 1000, 2000...). This will make the prediction easier.

## Pretrained Models

Usually people use pretrained models to extract features from the image and question.

:

VGG: A key aspect of VGG was to use many convolutional blocks with relatively narrow kernels, followed by a max-pooling step and to repeat this block multiple times.

Resnet: It is a residual learning framework to ease the training of networks that are substantially deep. It reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions.

:

Word2Vec: The word2vec tool takes a text corpus as input and produces the word vectors as output. It first constructs a vocabulary from the training text data and then learns vector representation of words. The model contains 300-dimensional vectors for 3 million words and phrases.

Glove: Similar to Word2Vec, it is a word embedding dataset. It contains 100/200/300-dimensional vectors for 2 million words.

skipthought: This is an encoder-decoder model that tries to reconstruct the surrounding sentences of an encoded passage. Sentences that share semantic and syntactic properties are thus mapped to similar vector representations. Different from the previous two model, this is a sentence based model.

GNMT encoder: We propose using the encoder of google neural machine translation system to extract the question features.

## Define the model

We define our model with gluon. `gluon.Block` is the basic building block of models. If any operator is not defined under gluon, you can use `mxnet.ndarray` operators to substitute.

```
In [2]: # Some parameters we are going to use
        batch_size = 64
        ctx = mx.cpu()
        compute_size = batch_size
        out_dim = 10000
        gpus = 1
```

In the `Net1` class, we will concatenate the image and question features and use multilayer perceptron(MLP) to predict the answer.

```
In [3]: class Net1(gluon.Block):
def __init__(self, **kwargs):
    super(Net1, self).__init__(**kwargs)
    with self.name_scope():
        # layers created in name_scope will inherit name space
        # from parent layer.
        self.bn = nn.BatchNorm()
        self.dropout = nn.Dropout(0.3)
        self.fc1 = nn.Dense(8192, activation="relu")
        self.fc2 = nn.Dense(1000)

def forward(self, x):
    x1 = F.L2Normalization(x[0])
    x2 = F.L2Normalization(x[1])
    z = F.concat(x1, x2, dim=1)
    z = self.fc1(z)
    z = self.bn(z)
    z = self.dropout(z)
    z = self.fc2(z)
    return z
```

In the `Net2` class, instead of linearly combine the image and text features, we use count sketch to estimate the outer product of the image and question features. It is also named as multimodal compact bilinear pooling(MCB).

This method was proposed in Multimodal Compact Bilinear Pooling for VQA. The key idea is:

$$\psi(x \otimes y, h, s) = \psi(x, h, s) \star \psi(y, h, s)$$

where  $\psi$  is the count sketch operator,  $x, y$  are the inputs,  $h, s$  are the hash tables,  $\otimes$  defines outer product and  $\star$  is the convolution operator. This can further be simplified by using FFT properties: convolution in time domain equals to elementwise product in frequency domain.

One improvement we made is adding ones vectors to each features before count sketch. The intuition is: given input vectors  $x, y$ , estimating outer product between  $[x, 1s]$  and  $[y, 1s]$  gives us information more than just  $x \otimes y$ . It also contains information of  $x$  and  $y$ .

```
In [4]: class Net2(gluon.Block):
def __init__(self, **kwargs):
    super(Net2, self).__init__(**kwargs)
    with self.name_scope():
        # layers created in name_scope will inherit name space
        # from parent layer.
        self.bn = nn.BatchNorm()
        self.dropout = nn.Dropout(0.3)
        self.fc1 = nn.Dense(8192, activation="relu")
        self.fc2 = nn.Dense(1000)

def forward(self, x):
    x1 = F.L2Normalization(x[0])
```



```

x2 = F.L2Normalization(x[1])
text_ones = F.ones((batch_size/gpus, 2048),ctx = ctx)
img_ones = F.ones((batch_size/gpus, 1024),ctx = ctx)
text_data = F.Concat(x1, text_ones,dim = 1)
image_data = F.Concat(x2,img_ones,dim = 1)
# Initialize hash tables
S1 = F.array(np.random.randint(0, 2, (1,3072))*2-1,ctx = ctx)
H1 = F.array(np.random.randint(0, out_dim,(1,3072)),ctx = ctx)
S2 = F.array(np.random.randint(0, 2, (1,3072))*2-1,ctx = ctx)
H2 = F.array(np.random.randint(0, out_dim,(1,3072)),ctx = ctx)
# Count sketch
cs1 = C.count_sketch( data = image_data, s=S1, h = H1 ,name='cs1',out_dim =
out_dim)
cs2 = C.count_sketch( data = text_data, s=S2, h = H2 ,name='cs2',out_dim =
out_dim)
fft1 = C.fft(data = cs1, name='fft1', compute_size = compute_size)
fft2 = C.fft(data = cs2, name='fft2', compute_size = compute_size)
c = fft1 * fft2
ifft1 = C.ifft(data = c, name='ifft1', compute_size = compute_size)
# MLP
z = self.fc1(ifft1)
z = self.bn(z)
z = self.dropout(z)
z = self.fc2(z)
return z

```

## Data Iterator

The inputs of the data iterator are extracted image and question features. At each step, the data iterator will return a data batch list: question data batch and image data batch.

We need to separate the data batches by the length of the input data because the input questions are in different lengths. The *buckets* parameter defines the max length you want to keep in the data iterator. Here since we already used pretrained model to extract the question feature, the question length is fixed as the output of the pretrained model.

The *layout* parameter defines the layout of the data iterator output. “N” specify where is the data batch dimension is.

*reset()* function is called after every epoch. *next()* function is call after each batch.

```

In [5]: class VQAttrainIter(mx.io.DataIter):
    def __init__(self, img, sentences, answer, batch_size, buckets=None, invalid_label=-1,
        text_name='text', img_name = 'image', label_name='softmax_label',
        dtype='float32', layout='NTC'):
        super(VQAttrainIter, self).__init__()
        if not buckets:
            buckets = [i for i, j in enumerate(np.bincount([len(s) for s in sentences]))
                if j >= batch_size]
            buckets.sort()

            ndiscard = 0
            self.data = [[] for _ in buckets]
            for i in range(len(sentences)):
                buck = bisect.bisect_left(buckets, len(sentences[i]))
                if buck == len(buckets):

```

```

        ndiscard += 1
        continue
    buff = np.full((buckets[buck],), invalid_label, dtype=dtype)
    buff[:len(sentences[i])] = sentences[i]
    self.data[buck].append(buff)

self.data = [np.asarray(i, dtype=dtype) for i in self.data]
self.answer = answer
self.img = img
print("WARNING: discarded %d sentences longer than the largest bucket."%ndiscard)

self.batch_size = batch_size
self.buckets = buckets
self.text_name = text_name
self.img_name = img_name
self.label_name = label_name
self.dtype = dtype
self.invalid_label = invalid_label
self.nd_text = []
self.nd_img = []
self.ndlabel = []
self.major_axis = layout.find('N')
self.default_bucket_key = max(buckets)

if self.major_axis == 0:
    self.provide_data = [(text_name, (batch_size, self.default_bucket_key)),
                        (img_name, (batch_size, self.default_bucket_key))]
    self.provide_label = [(label_name, (batch_size, self.default_bucket_key))]
elif self.major_axis == 1:
    self.provide_data = [(text_name, (self.default_bucket_key, batch_size)),
                        (img_name, (self.default_bucket_key, batch_size))]
    self.provide_label = [(label_name, (self.default_bucket_key, batch_size))]
else:
    raise ValueError("Invalid layout %s: Must by NT (batch major) or TN (time
major)")

self.idx = []
for i, buck in enumerate(self.data):
    self.idx.extend([(i, j) for j in range(0, len(buck) - batch_size + 1,
batch_size)])
self.curr_idx = 0

self.reset()

def reset(self):
    self.curr_idx = 0
    self.nd_text = []
    self.nd_img = []
    self.ndlabel = []
    for buck in self.data:
        label = np.empty_like(buck.shape[0])
        label = self.answer
        self.nd_text.append(mx.ndarray.array(buck, dtype=self.dtype))
        self.nd_img.append(mx.ndarray.array(self.img, dtype=self.dtype))
        self.ndlabel.append(mx.ndarray.array(label, dtype=self.dtype))

def next(self):
    if self.curr_idx == len(self.idx):
        raise StopIteration
    i, j = self.idx[self.curr_idx]
    self.curr_idx += 1

    if self.major_axis == 1:
        img = self.nd_img[i][j:j + self.batch_size].T
        text = self.nd_text[i][j:j + self.batch_size].T
        label = self.ndlabel[i][j:j+self.batch_size]
    else:
        img = self.nd_img[i][j:j + self.batch_size]
        text = self.nd_text[i][j:j + self.batch_size]
        label = self.ndlabel[i][j:j+self.batch_size]

    data = [text, img]

```

```

        return mx.io.DataBatch(data, [label],
                                bucket_key=self.buckets[i],
                                provide_data=[(self.text_name, text.shape), (self.img_name,
img.shape)],
                                provide_label=[(self.label_name, label.shape)])

```

## Load the data

Here we will use subset of VQA dataset in this tutorial. We extract the image feature from ResNet-152, text feature from GNMT encoder. In first two model, we have 21537 training samples and 1044 validation samples in this tutorial. Image feature is a 2048-dim vector. Question feature is a 1048-dim vector.

```

In [6]: # Download the dataset
dataset_files = {'train': ('train_question.npz', 'train_img.npz', 'train_ans.npz'),
                  'validation': ('val_question.npz', 'val_img.npz', 'val_ans.npz'),
                  'test':
('test_question_id.npz', 'test_question.npz', 'test_img_id.npz', 'test_img.npz', 'atoi.json', 't

train_q, train_i, train_a = dataset_files['train']
val_q, val_i, val_a = dataset_files['validation']

url_format = 'https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-
notebook/{}'
if not os.path.exists(train_q):
    logging.info('Downloading training dataset.')
    download(url_format.format(train_q), overwrite=True)
    download(url_format.format(train_i), overwrite=True)
    download(url_format.format(train_a), overwrite=True)
if not os.path.exists(val_q):
    logging.info('Downloading validation dataset.')
    download(url_format.format(val_q), overwrite=True)
    download(url_format.format(val_i), overwrite=True)
    download(url_format.format(val_a), overwrite=True)

```

```

INFO:root:Downloading training dataset.
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-
notebook/train_question.npz into train_question.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-
notebook/train_img.npz into train_img.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-
notebook/train_ans.npz into train_ans.npz successfully
INFO:root:Downloading validation dataset.
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-
notebook/val_question.npz into val_question.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-
notebook/val_img.npz into val_img.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-
notebook/val_ans.npz into val_ans.npz successfully

```

```

In [7]: layout = 'NT'
bucket = [1024]

train_question = np.load(train_q)['x']
val_question = np.load(val_q)['x']
train_ans = np.load(train_a)['x']
val_ans = np.load(val_a)['x']
train_img = np.load(train_i)['x']
val_img = np.load(val_i)['x']

print("Total training sample:", train_ans.shape[0])
print("Total validation sample:", val_ans.shape[0])

```

```
data_train = VQATrainIter(train_img, train_question, train_ans, batch_size, buckets =
bucket,layout=layout)
data_eva = VQATrainIter(val_img, val_question, val_ans, batch_size, buckets =
bucket,layout=layout)
```

Total training sample: 21537

Total validation sample: 1044

WARNING: discarded 0 sentences longer than the largest bucket.

WARNING: discarded 0 sentences longer than the largest bucket.

## Initialize the Parameters

```
In [8]: net = Net1()
        #net = Net2()
        net.collect_params().initialize(mx.init.Xavier(), ctx=ctx)
```

## Loss and Evaluation Metrics

```
In [9]: loss = gluon.loss.SoftmaxCrossEntropyLoss()

metric = mx.metric.Accuracy()

def evaluate_accuracy(data_iterator, net):
    numerator = 0.
    denominator = 0.

    data_iterator.reset()
    for i, batch in enumerate(data_iterator):
        with autograd.record():
            data1 = batch.data[0].as_in_context(ctx)
            data2 = batch.data[1].as_in_context(ctx)
            data = [data1, data2]
            label = batch.label[0].as_in_context(ctx)
            output = net(data)

            metric.update([label], [output])
    return metric.get()[1]
```

## Optimizer

```
In [10]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.01})
```

## Training loop

```
In [11]: epochs = 10
        moving_loss = 0.
        best_eva = 0
        for e in range(epochs):
            data_train.reset()
            for i, batch in enumerate(data_train):
                data1 = batch.data[0].as_in_context(ctx)
                data2 = batch.data[1].as_in_context(ctx)
                data = [data1, data2]
                label = batch.label[0].as_in_context(ctx)
```

```

with autograd.record():
    output = net(data)
    cross_entropy = loss(output, label)
    cross_entropy.backward()
trainer.step(data[0].shape[0])

#####
# Keep a moving average of the losses
#####
if i == 0:
    moving_loss = np.mean(cross_entropy.asnumpy()[0])
else:
    moving_loss = .99 * moving_loss + .01 * np.mean(cross_entropy.asnumpy()[0])
# if i % 200 == 0:
#     print("Epoch %s, batch %s. Moving avg of Loss: %s" % (e, i, moving_loss))
eva_accuracy = evaluate_accuracy(data_eva, net)
train_accuracy = evaluate_accuracy(data_train, net)
print("Epoch %s. Loss: %s, Train_acc %s, Eval_acc %s" % (e, moving_loss,
train_accuracy, eva_accuracy))
if eva_accuracy > best_eva:
    best_eva = eva_accuracy
    logging.info('Best validation acc found. Checkpointing...')
    net.save_params('vqa-mlp-%d.params'%(e))

```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 0. Loss: 3.07848375872, Train_acc 0.439319957386, Eval_acc 0.3525390625
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 1. Loss: 2.08781239439, Train_acc 0.478870738636, Eval_acc 0.436820652174
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 2. Loss: 1.63500481371, Train_acc 0.515536221591, Eval_acc 0.476584201389
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 3. Loss: 1.45585072303, Train_acc 0.549283114347, Eval_acc 0.513701026119
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 4. Loss: 1.17097555747, Train_acc 0.579172585227, Eval_acc 0.547500438904
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 5. Loss: 1.0625076159, Train_acc 0.606460108902, Eval_acc 0.577517947635
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 6. Loss: 0.832051645247, Train_acc 0.629863788555, Eval_acc 0.60488868656
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 7. Loss: 0.749606922723, Train_acc 0.650507146662, Eval_acc 0.62833921371
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 8. Loss: 0.680526961879, Train_acc 0.668269610164, Eval_acc 0.649105093573
```

```
INFO:root:Best validation acc found. Checkpointing...
```

```
Epoch 9. Loss: 0.53362678042, Train_acc 0.683984375, Eval_acc 0.666923484611
```

## Try it out!

Currently we have test data for the first two models we mentioned above. After the training loop over Net1 or Net2, we can try it on test data. Here we have 10 test samples.

```
In [12]: test = True
if test:
    test_q_id, test_q, test_i_id, test_i, atoi, text = dataset_files['test']

if test and not os.path.exists(test_q):
    logging.info('Downloading test dataset.')
    download(url_format.format(test_q_id), overwrite=True)
    download(url_format.format(test_q), overwrite=True)
    download(url_format.format(test_i_id), overwrite=True)
    download(url_format.format(test_i), overwrite=True)
    download(url_format.format(atoi), overwrite=True)
    download(url_format.format(text), overwrite=True)

if test:
    test_question = np.load("test_question.npz")['x']
    test_img = np.load("test_img.npz")['x']
    test_question_id = np.load("test_question_id.npz")['x']
    test_img_id = np.load("test_img_id.npz")['x']
    #atoi = np.load("atoi.json")['x']
```

```
INFO:root:Downloading test dataset.
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-notebook/test_question_id.npz into test_question_id.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-notebook/test_question.npz into test_question.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-notebook/test_img_id.npz into test_img_id.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-notebook/test_img.npz into test_img.npz successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-notebook/atoi.json into atoi.json successfully
INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-notebook/test_question_txt.json into test_question_txt.json successfully
```

We pass the test data iterator to the trained model.

```
In [13]: data_test = VQATrainIter(test_img, test_question, np.zeros((test_img.shape[0],1)), 10,
    buckets = bucket, layout=layout)
for i, batch in enumerate(data_test):
    with autograd.record():
        data1 = batch.data[0].as_in_context(ctx)
        data2 = batch.data[1].as_in_context(ctx)
        data = [data1, data2]
        #label = batch.label[0].as_in_context(ctx)
        #label_one_hot = nd.one_hot(label, 10)
        output = net(data)
output = np.argmax(output.asnumpy(), axis = 1)
```

```
WARNING: discarded 0 sentences longer than the largest bucket.
```

```
In [17]: idx = np.random.randint(10)
print(idx)
question = json.load(open(text))
print("Question:", question[idx])
```

```
6
Question: Is there a boat in the water?
```

```
In [18]: image_name = 'COCO_test2015_' + str(int(test_img_id[idx])).zfill(12) + '.jpg'
if not os.path.exists(image_name):
    logging.info('Downloading training dataset.')
    download(url_format.format('test_images/' + image_name), overwrite=True)

from IPython.display import Image
Image(filename=image_name)
```

```
INFO:root:Downloading training dataset.
```

INFO:root:downloaded https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/VQA-notebook/test\_images/COCO\_test2015\_000000419358.jpg into COCO\_test2015\_000000419358.jpg successfully

Out[18]:



```
In [19]: dataset = json.load(open('atoi.json'))
ans = dataset['ix_to_ans'][str(output[idx]+1)]
print("Answer:", ans)
```

Answer: yes

For whinges or inquiries, [open an issue on GitHub](#).



# Tree LSTM modeling for semantic relatedness

Just five years ago, many of the most successful models for doing supervised learning with text ignored word order altogether. Some of the most successful models represented documents or sentences with the order-invariant *bag-of-words* representation. Anyone thinking hard should probably have realized that these models couldn't dominate forever. That's because we all know that word order actually does matter. Bag-of-words models, which ignored word order, left some information on the table.

The recurrent neural networks that [we introduced in chapter 5](#) model word order, by passing over the sequence of words in order, updating the models representation of the sentence after each word. And, with LSTM recurrent cells and training on GPUs, even the straightforward LSTM far outpaces classical approaches, on a number of tasks, including language modeling, named entity recognition and more.

But while those models are impressive, they still may be leaving some knowledge on the table. To begin with, we know a priori that sentence have a grammatical structure. And we already have some tools that are very good at recovering parse trees that reflect grammatical structure of the sentences. While it may be possible for an LSTM to learn this information implicitly, it's often a good idea to build known information into the structure of a neural network. Take for example convolutional neural networks. They build in the prior knowledge that low level feature should be translation-invariant. It's possible to come up with a fully connected net that does the same thing, but it would require many more nodes and would be much more susceptible to overfitting. In this case, we would like to build the grammatical tree structure of the sentences into the architecture of an LSTM recurrent neural network. This tutorial walks through *tree LSTMs*, an approach that does precisely that. The models here are based on the [tree-structured LSTM](#) by Kai Sheng Tai, Richard Socher, and Chris Manning. Our implementation borrows from [this Pytorch example](#).

## Sentences involving Compositional Knowledge

This tutorial walks through training a child-sum Tree LSTM model for analyzing semantic relatedness of sentence pairs given their dependency parse trees.

## Preliminaries

Before getting going, you'll probably want to note a couple preliminary details:





```

        self.hc2h_weight = self.params.get('hc2h_weight', shape=(hidden_size,
hidden_size),
                                           init=hc2h_weight_initializer)
        self.i2h_bias = self.params.get('i2h_bias', shape=(4*hidden_size,),
                                           init=i2h_bias_initializer)
        self.hs2h_bias = self.params.get('hs2h_bias', shape=(3*hidden_size,),
                                           init=hs2h_bias_initializer)
        self.hc2h_bias = self.params.get('hc2h_bias', shape=(hidden_size,),
                                           init=hc2h_bias_initializer)

def forward(self, F, inputs, tree):
    children_outputs = [self.forward(F, inputs, child)
                        for child in tree.children]
    if children_outputs:
        _, children_states = zip(*children_outputs) # unzip
    else:
        children_states = None

    with inputs.context as ctx:
        return self.node_forward(F, F.expand_dims(inputs[tree.idx], axis=0),
children_states,
                                self.i2h_weight.data(ctx),
                                self.hs2h_weight.data(ctx),
                                self.hc2h_weight.data(ctx),
                                self.i2h_bias.data(ctx),
                                self.hs2h_bias.data(ctx),
                                self.hc2h_bias.data(ctx))

def node_forward(self, F, inputs, children_states,
                 i2h_weight, hs2h_weight, hc2h_weight,
                 i2h_bias, hs2h_bias, hc2h_bias):
    # comment notation:
    # N for batch size
    # C for hidden state dimensions
    # K for number of children.

    # FC for i, f, u, o gates (N, 4*C), from input to hidden
    i2h = F.FullyConnected(data=inputs, weight=i2h_weight, bias=i2h_bias,
                           num_hidden=self._hidden_size*4)
    i2h_slices = F.split(i2h, num_outputs=4) # (N, C)*4
    i2h_iuo = F.concat(*[i2h_slices[i] for i in [0, 2, 3]], dim=1) # (N, C*3)

    if children_states:
        # sum of children states, (N, C)
        hs = F.add_n(*[state[0] for state in children_states])
        # concatenation of children hidden states, (N, K, C)
        hc = F.concat(*[F.expand_dims(state[0], axis=1) for state in children_states],
dim=1)

        # concatenation of children cell states, (N, K, C)
        cs = F.concat(*[F.expand_dims(state[1], axis=1) for state in children_states],
dim=1)

        # calculate activation for forget gate. addition in f_act is done with
broadcast
        i2h_f_slice = i2h_slices[1]
        f_act = i2h_f_slice + hc2h_bias + F.dot(hc, hc2h_weight) # (N, K, C)
        forget_gates = F.Activation(f_act, act_type='sigmoid') # (N, K, C)
    else:
        # for leaf nodes, summation of children hidden states are zeros.
        hs = F.zeros_like(i2h_slices[0])

    # FC for i, u, o gates, from summation of children states to hidden state
    hs2h_iuo = F.FullyConnected(data=hs, weight=hs2h_weight, bias=hs2h_bias,
                               num_hidden=self._hidden_size*3)
    i2h_iuo = i2h_iuo + hs2h_iuo

    iuo_act_slices = F.SliceChannel(i2h_iuo, num_outputs=3) # (N, C)*3
    i_act, u_act, o_act = iuo_act_slices[0], iuo_act_slices[1], iuo_act_slices[2] #
(N, C) each

    # calculate gate outputs
    in_gate = F.Activation(i_act, act_type='sigmoid')

```

```

in_transform = F.Activation(u_act, act_type='tanh')
out_gate = F.Activation(o_act, act_type='sigmoid')

# calculate cell state and hidden state
next_c = in_gate * in_transform
if children_states:
    next_c = F.sum(forget_gates * cs, axis=1) + next_c
next_h = out_gate * F.Activation(next_c, act_type='tanh')

return next_h, [next_h, next_c]

```

## Similarity regression module

```

In [5]: # module for distance-angle similarity
class Similarity(nn.Block):
    def __init__(self, sim_hidden_size, rnn_hidden_size, num_classes):
        super(Similarity, self).__init__()
        with self.name_scope():
            self.wh = nn.Dense(sim_hidden_size, in_units=2*rnn_hidden_size)
            self.wp = nn.Dense(num_classes, in_units=sim_hidden_size)

    def forward(self, F, lvec, rvec):
        # lvec and rvec will be tree_lstm cell states at roots
        mult_dist = F.broadcast_mul(lvec, rvec)
        abs_dist = F.abs(F.add(lvec, -rvec))
        vec_dist = F.concat(*[mult_dist, abs_dist], dim=1)
        out = F.log_softmax(self.wp(F.sigmoid(self.wh(vec_dist))))
        return out

```

## Final model

```

In [6]: # putting the whole model together
class SimilarityTreeLSTM(nn.Block):
    def __init__(self, sim_hidden_size, rnn_hidden_size, embed_in_size, embed_dim,
num_classes):
        super(SimilarityTreeLSTM, self).__init__()
        with self.name_scope():
            self.embed = nn.Embedding(embed_in_size, embed_dim)
            self.childsumtreelstm = ChildSumLSTMCell(rnn_hidden_size,
input_size=embed_dim)
            self.similarity = Similarity(sim_hidden_size, rnn_hidden_size, num_classes)

    def forward(self, F, l_inputs, r_inputs, l_tree, r_tree):
        l_inputs = self.embed(l_inputs)
        r_inputs = self.embed(r_inputs)
        # get cell states at roots
        lstate = self.childsumtreelstm(F, l_inputs, l_tree)[1][1]
        rstate = self.childsumtreelstm(F, r_inputs, r_tree)[1][1]
        output = self.similarity(F, lstate, rstate)
        return output

```

## Dataset classes

### Vocab

```

In [7]: import os
import logging
logging.basicConfig(level=logging.INFO)
import numpy as np
import random

```

```

from tqdm import tqdm

import mxnet as mx

# class for vocabulary and the word embeddings
class Vocab(object):
    # constants for special tokens: padding, unknown, and beginning/end of sentence.
    PAD, UNK, BOS, EOS = 0, 1, 2, 3
    PAD_WORD, UNK_WORD, BOS_WORD, EOS_WORD = '<blank>', '<unk>', '<s>', '</s>'
    def __init__(self, filepaths=[], embedpath=None, include_unseen=False, lower=False):
        self.idx2tok = []
        self.tok2idx = {}
        self.lower = lower
        self.include_unseen = include_unseen

        self.add(Vocab.PAD_WORD)
        self.add(Vocab.UNK_WORD)
        self.add(Vocab.BOS_WORD)
        self.add(Vocab.EOS_WORD)

        self.embed = None

        for filename in filepaths:
            logging.info('loading %s'%filename)
            with open(filename, 'r') as f:
                self.load_file(f)
        if embedpath is not None:
            logging.info('loading %s'%embedpath)
            with open(embedpath, 'r') as f:
                self.load_embedding(f, reset=set([Vocab.PAD_WORD, Vocab.UNK_WORD,
Vocab.BOS_WORD,
                                                    Vocab.EOS_WORD]))

    @property
    def size(self):
        return len(self.idx2tok)

    def get_index(self, key):
        return self.tok2idx.get(key.lower() if self.lower else key,
                                Vocab.UNK)

    def get_token(self, idx):
        if idx < self.size:
            return self.idx2tok[idx]
        else:
            return Vocab.UNK_WORD

    def add(self, token):
        token = token.lower() if self.lower else token
        if token in self.tok2idx:
            idx = self.tok2idx[token]
        else:
            idx = len(self.idx2tok)
            self.idx2tok.append(token)
            self.tok2idx[token] = idx
        return idx

    def to_indices(self, tokens, add_bos=False, add_eos=False):
        vec = [BOS] if add_bos else []
        vec += [self.get_index(token) for token in tokens]
        if add_eos:
            vec.append(EOS)
        return vec

    def to_tokens(self, indices, stop):
        tokens = []
        for i in indices:
            tokens += [self.get_token(i)]
            if i == stop:
                break
        return tokens

```

```

def load_file(self, f):
    for line in f:
        tokens = line.rstrip('\n').split()
        for token in tokens:
            self.add(token)

def load_embedding(self, f, reset=[]):
    vectors = {}
    for line in tqdm(f.readlines(), desc='Loading embeddings'):
        tokens = line.rstrip('\n').split(' ')
        word = tokens[0].lower() if self.lower else tokens[0]
        if self.include_unseen:
            self.add(word)
        if word in self.tok2idx:
            vectors[word] = [float(x) for x in tokens[1:]]
    dim = len(vectors.values()[0])
    def to_vector(tok):
        if tok in vectors and tok not in reset:
            return vectors[tok]
        elif tok not in vectors:
            return np.random.normal(-0.05, 0.05, size=dim)
        else:
            return [0.0]*dim
    self.embed = mx.nd.array([vectors[tok] if tok in vectors and tok not in reset
                              else [0.0]*dim for tok in self.idx2tok])

```

## Data iterator

```

In [8]: # Iterator class for SICK dataset
class SICKDataIter(object):
    def __init__(self, path, vocab, num_classes, shuffle=True):
        super(SICKDataIter, self).__init__()
        self.vocab = vocab
        self.num_classes = num_classes
        self.l_sentences = []
        self.r_sentences = []
        self.l_trees = []
        self.r_trees = []
        self.labels = []
        self.size = 0
        self.shuffle = shuffle
        self.reset()

    def reset(self):
        if self.shuffle:
            mask = list(range(self.size))
            random.shuffle(mask)
            self.l_sentences = [self.l_sentences[i] for i in mask]
            self.r_sentences = [self.r_sentences[i] for i in mask]
            self.l_trees = [self.l_trees[i] for i in mask]
            self.r_trees = [self.r_trees[i] for i in mask]
            self.labels = [self.labels[i] for i in mask]
        self.index = 0

    def next(self):
        out = self[self.index]
        self.index += 1
        return out

    def set_context(self, context):
        self.l_sentences = [a.as_in_context(context) for a in self.l_sentences]
        self.r_sentences = [a.as_in_context(context) for a in self.r_sentences]

    def __len__(self):
        return self.size

    def __getitem__(self, index):
        l_tree = self.l_trees[index]
        r_tree = self.r_trees[index]

```

```

l_sent = self.l_sentences[index]
r_sent = self.r_sentences[index]
label = self.labels[index]
return (l_tree, l_sent, r_tree, r_sent, label)

```

## Training with autograd

```

In [9]: import argparse, pickle, math, os, random
import logging
logging.basicConfig(level=logging.INFO)
import numpy as np

import mxnet as mx
from mxnet import gluon
from mxnet.gluon import nn
from mxnet import autograd as ag

# training settings and hyper-parameters
use_gpu = False
optimizer = 'AdaGrad'
seed = 123
batch_size = 25
training_batches_per_epoch = 10
learning_rate = 0.01
weight_decay = 0.0001
epochs = 1
rnn_hidden_size, sim_hidden_size, num_classes = 150, 50, 5

# initialization
context = [mx.gpu(0) if use_gpu else mx.cpu()]

# seeding
mx.random.seed(seed)
np.random.seed(seed)
random.seed(seed)

# read dataset
def verified(file_path, sha1hash):
    import hashlib
    sha1 = hashlib.sha1()
    with open(file_path, 'rb') as f:
        while True:
            data = f.read(1048576)
            if not data:
                break
            sha1.update(data)
        matched = sha1.hexdigest() == sha1hash
    if not matched:
        logging.warn('Found hash mismatch in file {}, possibly due to incomplete
download.'.format(file_path))
    return matched

data_file_name = 'tree_lstm_dataset-3d85a6c4.cPickle'
data_file_hash = '3d85a6c44a335a33edc060028f91395ab0dcf601'
if not os.path.exists(data_file_name) or not verified(data_file_name, data_file_hash):
    from mxnet.test_utils import download
    download('https://apache-mxnet.s3-
accelerate.amazonaws.com/gluon/dataset/%s'%data_file_name,
            overwrite=True)

with open('tree_lstm_dataset-3d85a6c4.cPickle', 'rb') as f:
    train_iter, dev_iter, test_iter, vocab = pickle.load(f)

logging.info('==> SICK vocabulary size : %d ' % vocab.size)
logging.info('==> Size of train data : %d ' % len(train_iter))
logging.info('==> Size of dev data : %d ' % len(dev_iter))
logging.info('==> Size of test data : %d ' % len(test_iter))

```

```

# get network
net = SimilarityTreeLSTM(sim_hidden_size, rnn_hidden_size, vocab.size,
vocab.embed.shape[1], num_classes)

# use pearson correlation and mean-square error for evaluation
metric = mx.metric.create(['pearsonr', 'mse'])

# the prediction from the network is Log-probability vector of each score class
# so use the following function to convert scalar score to the vector
# e.g 4.5 -> [0, 0, 0, 0.5, 0.5]
def to_target(x):
    target = np.zeros((1, num_classes))
    ceil = int(math.ceil(x))
    floor = int(math.floor(x))
    if ceil==floor:
        target[0][floor-1] = 1
    else:
        target[0][floor-1] = ceil - x
        target[0][ceil-1] = x - floor
    return mx.nd.array(target)

# and use the following to convert log-probability vector to score
def to_score(x):
    levels = mx.nd.arange(1, 6, ctx=x.context)
    return [mx.nd.sum(levels*mx.nd.exp(x), axis=1).reshape((-1,1))]

# when evaluating in validation mode, check and see if pearson-r is improved
# if so, checkpoint and run evaluation on test dataset
def test(ctx, data_iter, best, mode='validation', num_iter=-1):
    data_iter.reset()
    samples = len(data_iter)
    data_iter.set_context(ctx[0])
    preds = []
    labels = [mx.nd.array(data_iter.labels, ctx=ctx[0]).reshape((-1,1))]
    for _ in tqdm(range(samples), desc='Testing in {} mode'.format(mode)):
        l_tree, l_sent, r_tree, r_sent, label = data_iter.next()
        z = net(mx.nd, l_sent, r_sent, l_tree, r_tree)
        preds.append(z)

    preds = to_score(mx.nd.concat(*preds, dim=0))
    metric.update(preds, labels)
    names, values = metric.get()
    metric.reset()
    for name, acc in zip(names, values):
        logging.info(mode+' acc: %s=%f'%(name, acc))
        if name == 'pearsonr':
            test_r = acc
    if mode == 'validation' and num_iter >= 0:
        if test_r >= best:
            best = test_r
            logging.info('New optimum found: {}'.format(best))
    return best

def train(epoch, ctx, train_data, dev_data):
    # initialization with context
    if isinstance(ctx, mx.Context):
        ctx = [ctx]
    net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx[0])
    net.embed.weight.set_data(vocab.embed.as_in_context(ctx[0]))
    train_data.set_context(ctx[0])
    dev_data.set_context(ctx[0])

    # set up trainer for optimizing the network.
    trainer = gluon.Trainer(net.collect_params(), optimizer, {'learning_rate':
learning_rate, 'wd': weight_decay})

    best_r = -1
    Loss = gluon.loss.KLDivLoss()
    for i in range(epoch):
        train_data.reset()

```

```

num_samples = min(len(train_data), training_batches_per_epoch*batch_size)
# collect predictions and labels for evaluation metrics
preds = []
labels = [mx.nd.array(train_data.labels[:num_samples],
ctx=ctx[0]).reshape((-1,1))]
for j in tqdm(range(num_samples), desc='Training epoch {}'.format(i)):
    # get next batch
    l_tree, l_sent, r_tree, r_sent, label = train_data.next()
    # use autograd to record the forward calculation
    with ag.record():
        # forward calculation. the output is Log probability
        z = net(mx.nd, l_sent, r_sent, l_tree, r_tree)
        # calculate loss
        loss = Loss(z, to_target(label).as_in_context(ctx[0]))
        # backward calculation for gradients.
        loss.backward()
        preds.append(z)
    # update weight after every batch_size samples
    if (j+1) % batch_size == 0:
        trainer.step(batch_size)

# translate Log-probability to scores, and evaluate
preds = to_score(mx.nd.concat(*preds, dim=0))
metric.update(preds, labels)
names, values = metric.get()
metric.reset()
for name, acc in zip(names, values):
    logging.info('training acc at epoch %d: %s=%f'%(i, name, acc))
best_r = test(ctx, dev_data, best_r, num_iter=i)

train(epochs, context, train_iter, dev_iter)

```

```

INFO:root:==> SICK vocabulary size : 2412
INFO:root:==> Size of train data : 4500
INFO:root:==> Size of dev data : 500
INFO:root:==> Size of test data : 4927
Training epoch 0: 100%|██████████| 250/250 [00:11<00:00, 21.48it/s]
INFO:root:training acc at epoch 0: pearsonr=0.096197
INFO:root:training acc at epoch 0: mse=1.138699
Testing in validation mode: 100%|██████████| 500/500 [00:09<00:00, 51.57it/s]
INFO:root:validation acc: pearsonr=0.490352
INFO:root:validation acc: mse=1.237509
INFO:root:New optimum found: 0.49035187610029013.

```

## Conclusion


- Gluon offers great tools for modeling in an imperative way.



# Introduction to recommender systems

[Early, early draft]

This chapter introduces recommender systems (commonly called RecSys), tools that recommend *items* to *users*. Many of the most popular uses of recommender systems involve suggesting products to customers. Amazon, for example, uses recommender systems to choose which retail products to display. Recommender systems aren't limited to physical products. For example, the algorithms that Pandora and Spotify use to curate playlists are recommender systems. Personalized suggestions on news websites are recommender systems. And as of this writing, several carousels on the home page for Amazon's Prime Videos's contain personalized TV and Movie recommendations.



I (Zack) have honestly no idea why Amazon wants me to watch Bubble Guppies. It's possible that Bubble Guppies is a masterpiece, and the recommender systems knows that my life will change upon watching it. It's also possible that the recommender made a mistake. For example, it might have extrapolated incorrectly from my affinity for the anime Death Note, thinking that I would similarly love any animated series. And, since I've never rated a nickelodean series (either positively or negatively), the system may have no knowledge to the contrary. It's also possible that this series is a new addition to the catalogue, and thus they need to recommend the item to many users in order to develop a sense of *who* likes Bubble Guppies. This problem, of sorting out how to handle a new item, is called the *cold-start* problem.

A recommender system doesn't have to use any sophisticated machine learning techniques. And it doesn't even have to be personalized. One reasonable baseline for most applications is to suggest the most popular items to everyone. But we have to be careful. Depending on how we define popularity, we might create a feedback loop. The most popular items get recommended which makes them even more popular, which makes them even more frequently recommended, etc.

For services with diverse users, however, personalization can be essential. Diapers are among the most popular items on Amazon, but we probably shouldn't recommend diapers to adolescents. We also probably *should not* recommend anything associated with Justin Bieber to a user who *isn't* an adolescent. Moreover, we might want to personalize, not only to the user, but to the context. For example, just after I bought a Pixel phone, I was in the market for a phone case. But I have no interest in buying a phone case one year later.

## Many ways to pose the problem

While it might seem obvious, that personalization is a good strategy, it's not immediately obvious how best to articulate recommendation as a machine learning problem.

Discuss: \* Rating prediction \* Passive feedback (view/notview) \* Content-based recommendation

## Amazon review dataset

- introduce dataset

```
In [5]: import mxnet
import mxnet.ndarray as nd
import urllib
import gzip
```

```
In [10]: with
gzip.open(urllib.request.urlopen("http://snap.stanford.edu/data/amazon/productGraph/categor
as f:
    data = [eval(l) for l in f]
```

```
In [11]: data[0]
```

```
Out[11]: {'asin': '616719923X',
'helpful': [0, 0],
'overall': 4.0,
'reviewText': 'Just another flavor of Kit Kat but the taste is unique and a bit
different. The only thing that is bothersome is the price. I thought it was a bit
expensive...',
'reviewTime': '06 1, 2013',
'reviewerID': 'A1VEELTKS8NLZB',
'reviewerName': 'Amazon Customer',
'summary': 'Good Taste',
'unixReviewTime': 1370044800}
```

## [Do some dataset exploration]

- Look at the average rating
- Look at the number of unique users and items
- Plot a histogram of the number of ratings/reviews corresponding to each user
- "" for items

```
In [17]: users = [d['reviewerID'] for d in data]
```

```
In [18]: items = [d['asin'] for d in data]
```

```
In [14]: ratings = [d['overall'] for d in data]
```

## Models

- Just the average
- Offset plus user and item biases
- Latent factor model / matrix factorization

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

# Linear Dynamical Systems with MXNet

In this notebook we will look at how to implement filtering in general linear dynamical systems (aka Kalman filtering) using MXNet.

First, a short mathematical description of the problem.

A general (Gaussian) linear dynamical system is specified by two equations.

- The first, called the transition equation,

$$h_t = Ah_{t-1} + \epsilon_t \quad \epsilon_t \sim \mathcal{N}(0, \Sigma_h)$$

describes how the hidden (also called “latent”) state  $h_t \in \mathbb{R}^H$  evolves with time. In a LDS this involves applying a linear transformation  $A \in \mathbb{R}^{H \times H}$  to the previous hidden state  $h_{t-1}$ , followed by adding zero-mean Gaussian noise.

- The second, the observation equation or emission model,

$$v_t = Bh_t + \nu_t \quad \nu_t \sim \mathcal{N}(0, \Sigma_v)$$

describes how the latent state  $h_t$  relates to the observations (“visibles”)  $v_t \in \mathbb{R}^D$ . In particular,  $v_t$  is a linear transformation of the hidden state,  $Bh_t$ , to which Gaussian noise is added.

Finally, we need to specify the initial state, usually by placing a Gaussian prior on  $h_0$ ,

$$h_0 \sim \mathcal{N}(\mu_0, \Sigma_0)$$

The LDS is thus fully specified by the system parameters  $A \in \mathbb{R}^{H \times H}$ ,  $B \in \mathbb{R}^{D \times H}$ ,  $\Sigma_h \in \mathcal{S}_+^H$ ,  $\Sigma_v \in \mathcal{S}_+^D$ ,  $\mu_0 \in \mathbb{R}^H$ ,  $\Sigma_0 \in \mathcal{S}_+^H$ .  $\mathcal{S}_+$  denotes the space of positive definite (PD) matrices.

Given such a LDS specification, and a sequence of observations  $v_0, v_1, \dots, v_T$ , one is typically interested in one of the following

1. (Log-)Likelihood computation, i.e. computing the probability of the data under the model,  $P(v_0, v_1, \dots, v_T)$
2. Filtering, i.e. computing the mean and covariance of  $P(h_t | v_0, v_1, \dots, v_t)$
3. Smoothing, i.e. computing the mean and covariance of  $P(h_t | v_0, v_1, \dots, v_T)$

4. Parameter learning: find the system parameters that best describe the data, e.g. by maximizing likelihood

In this notebook we will focus on the filtering problem, and will also see how to compute the log-likelihood as a byproduct. For details on other problems, See e.g. [Barber, 2012](#), Chapter 24.

## Filtering

We want to find the “filtered” distributions  $p(h_t|v_{0:t})$  where  $v_{0:t}$  denotes  $\{v_0, \dots, v_t\}$ . Due to the closure properties of Gaussian distributions, each of these distributions is also Gaussian  $p(h_t|v_{0:t}) = \mathcal{N}(h_t|f_t, F_t)$ . The filtering procedure proceeds sequentially, by expressing  $f_t$  and  $F_t$  in terms of  $f_{t-1}$  and  $F_{t-1}$ . We initialize  $f_0$  and  $F_0$  to be 0.

## Prerequisite

To derive the formulas for filtering, here is all you need [see Bishop 2008, Appendix B]

- Conditional Gaussian equations

$$\begin{aligned} p(x) &= \mathcal{N}(\mu, \Lambda^{-1}) \\ p(y|x) &= \mathcal{N}(y|Ax + b, L^{-1}) \end{aligned}$$

The marginal distribution of  $y$  and the conditional distribution of  $x$  given  $y$  are

$$\begin{aligned} p(y) &= \mathcal{N}(y|A\mu + b, L^{-1} + A\Lambda^{-1}A^T) \quad (1) \\ p(x|y) &= \mathcal{N}(x|\Sigma [A^T L(y - b) + \Lambda\mu], \Sigma), \quad \Sigma = (\Lambda + A^T L A)^{-1} \quad (2) \end{aligned}$$

- Matrix Inversion Lemma (aka [Woodbury matrix identity](#))

$$(A + BD^{-1}C)^{-1} = A^{-1} - A^{-1}B(D + CA^{-1}B)^{-1}CA^{-1} \quad (3)$$

## Derivation

Now we are ready to derive the filtering equations, by Bayes Theorem

$$\begin{aligned} p(h_t|v_{0:t}) &= p(h_t|v_t, v_{0:t-1}) \\ &\propto p(v_t|h_t, v_{0:t-1})p(h_t|v_{0:t-1}) \\ &= p(v_t|h_t)p(h_t|v_{0:t-1}) \quad \text{by Markov property} \end{aligned}$$

The derivation boils down to calculate the two terms on the right hand side (you can think that the first is  $p(y|x)$  and the second is  $p(x)$  as in the conditional Gaussian equations) and use (2) above to get the desired formula.

The first term is directly given by the observation equation, i.e.,  $p(v_t|h_t) = \mathcal{N}(Bh_t, \Sigma_v)$ , and the second term can be calculated as follows

$$\begin{aligned}
p(h_t|v_{0:t-1}) &= \int p(h_t|h_{t-1}, v_{0:t-1})p(h_{t-1}|v_{0:t-1})dh_{t-1} \\
&= \int p(h_t|h_{t-1})p(h_{t-1}|v_{0:t-1})dh_{t-1} \quad \text{by Markov property} \\
&= \int \mathcal{N}(h_t|Ah_{t-1}, \Sigma_h)\mathcal{N}(h_{t-1}|f_{t-1}, F_{t-1})dh_{t-1} \\
&= \mathcal{N}(Af_{t-1}, AF_{t-1}A^T + \Sigma_h) \quad \text{using the marginalization equation (1)} \\
&= \mathcal{N}(\mu_f, \Sigma_{hh})
\end{aligned}$$

First, we calculate the covariance matrix  $F_t$ ,

$$F_t = (\Sigma_{hh}^{-1} - B^T \Sigma_v^{-1} B)^{-1} = \Sigma_{hh} - \Sigma_{hh} B^T (\Sigma_v + B \Sigma_{hh} B^T)^{-1} B \Sigma_{hh} = (I - KB) \Sigma_{hh},$$

where we have used the matrix inversion lemma and define the **Kalman gain matrix** as

$$K = \Sigma_{hh} B^T (\Sigma_v + B \Sigma_{hh} B^T)^{-1}.$$

Notice that for numerical stability, the covariance matrix is normally calculated using so-called “Joseph’s symmetrized update,”

$$F_t = (I - K_t B) \Sigma_{hh} (I - K_t B)^T + K_t \Sigma_v K_t^T,$$

which consists of summing two PD matrices.

Finally, after some algebraic manipulation, we have the mean

$$f_t = \mu_h + K(v - B\mu_h).$$

## LDS Foward Pass

To summarize, the iterative algorithm proceeds as follows

$$\begin{aligned}
\mu_h &= Af_{t-1} & \mu_v &= B\mu_h \\
\Sigma_{hh} &= AF_{t-1}A^T + \Sigma_h & \Sigma_{vv} &= B\Sigma_{hh}B^T + \Sigma_v \\
K_t &= \Sigma_{hh}B^T \Sigma_{vv}^{-1} \\
f_t &= \mu_h + K(v - \mu_v) & F_t &= (I - K_t B) \Sigma_{hh} (I - K_t B)^T + K_t \Sigma_v K_t^T
\end{aligned}$$

As we can see, each step in the recursive filtering procedure involves a few matrix-matrix and matrix-vector multiplications, as well as some matrix and vector additions and subtractions. These are standard operators available in most deep learning frameworks (including MXNet, where matrix multiplication is available through `mx.sym.dot()`). However, the update also involves the term  $\Sigma_{vv}^{-1}$ , the inverse of a  $D$ -by- $D$  symmetric, positive semidefinite matrix (due to it being a covariance matrix). If the output dimensionality  $D$  is 1, this is simply the scalar  $1/\Sigma_{vv}$ , but in the general case we need to compute the inverse (or, preferably, solve the corresponding linear system directly).

Luckily, operators for doing exactly that have recently been added to MXNet. In particular, we have

- `mx.nd.linalg_gemm2` (Matrix-matrix product, more flexible than `dot`)
- `mx.nd.linalg_potrf` (Cholesky decomposition  $A = LL^T$  for symmetric, PD matrix  $A$ )
- `mx.nd.linalg_trsm` (solve system of equations involving triangular matrices)
- `mx.nd.linalg_sumlogdiag` (compute the sum of the log of the diagonal elements of a matrix)
- `mx.nd.linalg_potri` (compute  $A^{-1}$  from a previously computed Cholesky factor  $L$  of  $A$  (i.e.  $LL^T = A$ )).

## Computing the likelihood

The terms  $\mu_v$  and  $\Sigma_{vv}$  computed during the filtering update correspond to the mean and covariance of the predictive distribution  $P(v_t | v_0, v_1, \dots, v_{t-1})$ , which allows us to compute the likelihood by decomposing it into telescoping conditional distributions,

$$P(v_0, v_1, \dots, v_t) = P(v_0)P(v_1 | v_0)P(v_2 | v_0, v_1) \cdots P(v_t | v_0, v_1, \dots, v_{t-1})$$

and then using  $P(v_t | v_0, v_1, \dots, v_{t-1}) = \mathcal{N}(v_t | \mu_v, \Sigma_{vv})$  with parameters obtained during filtering to compute each term.

```
In [1]: import mxnet as mx

from mxnet.ndarray import linalg_gemm as gemm
from mxnet.ndarray import linalg_gemm2 as gemm2
from mxnet.ndarray import linalg_potrf as potrf
from mxnet.ndarray import linalg_trsm as trsm
from mxnet.ndarray import linalg_sumlogdiag as sumlogdiag

import mxnet.ndarray as nd
```

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams["figure.figsize"] = (10, 5)
```

## Generating Synthetic Dataset

This example is adapted from Example 24.3 in (Barber, 2017). The two-dimensional latent vector  $h_t$  is rotated at each iteration and then is projected to produce a scalar observation. More precisely, we have

$$h_{t+1} = Ah_t + \epsilon_h, \quad A = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}, \epsilon_h \sim \mathcal{N}(0, \alpha^2 \cdot \mathbb{I}_2)$$

$$v_{t+1} = [1, 0] \cdot h_{t+1} + \epsilon_v, \quad \epsilon_v \sim \mathcal{N}(0, \sigma^2).$$

```
In [3]: alpha = 0.5
        sigma = 0.5
        theta = np.pi / 6
        T = 50
```

```
In [4]: A = nd.array([[np.cos(theta), -np.sin(theta)],
                      [np.sin(theta), np.cos(theta)]])
        B = nd.array([[1, 0]])

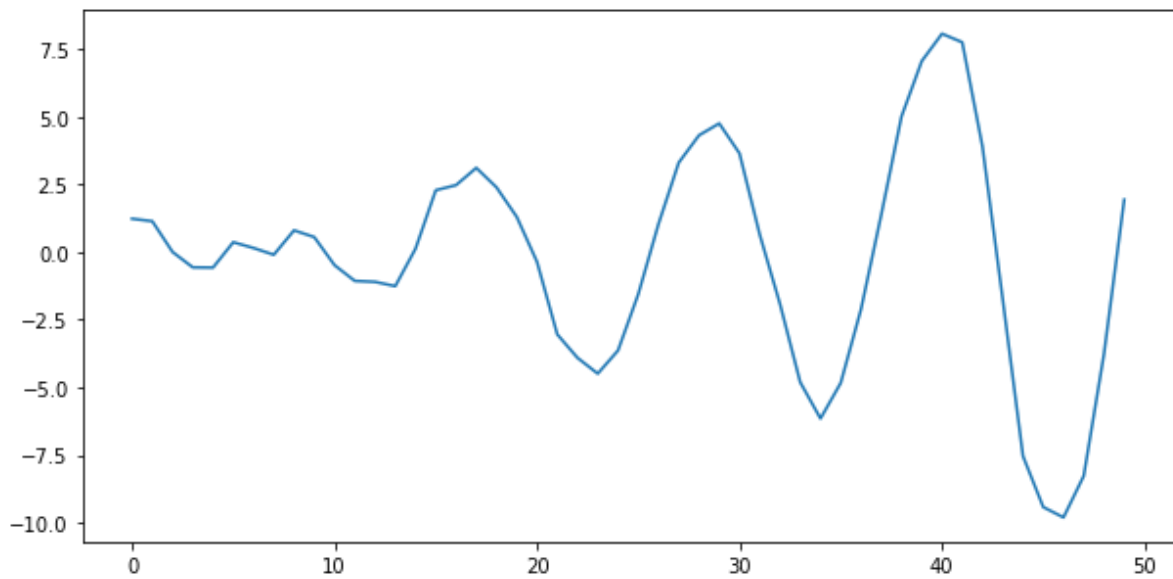
        S_h = nd.array(np.square(alpha) * np.eye(2))
        S_v = nd.array(np.square(sigma) * np.eye(1))

        v = []
        # initial state h_0
        h = np.array([1, 0])
        for t in range(T):
            # h_t = Bh_{t-1} + \epsilon_h
            h = np.random.multivariate_normal(A.asnumpy().dot(h), S_h.asnumpy())

            # v_t = Ah_t + \epsilon_v
            vv = np.random.normal(B.asnumpy().dot(h), S_v.asnumpy())

            v.append(vv)
        v = nd.array(np.array(v).reshape((T,1)))
```

```
In [5]: plt.plot(v.asnumpy());
```



## LDS Forward Function (Filtering)

```
In [6]: def LDS_forward(v, A, B, S_h, S_v):
```



```

H = A.shape[0] # dim of latent state
D = B.shape[0] # dim of observation
T = v.shape[0] # num of observations

f_0 = nd.zeros((H,1))
F_0 = nd.zeros((H,H))

eye_h = nd.array(np.eye(H))

F_t = None
f_t = None
F_seq = []
f_seq = []
log_p_seq = []

for t in range(T):

    if t == 0:
        # At the first time step, use the prior
        mu_h = f_0
        S_hh = F_0
    else:
        # Otherwise compute using update eqns.
        mu_h = gemm2(A, f_t)
        S_hh = gemm2(A, gemm2(F_t, A, transpose_b=1)) + S_h

    # direct transcription of the update equations above
    mu_v = gemm2(B, mu_h)
    S_hh_x_B_t = gemm2(S_hh, B, transpose_b=1)
    S_vv = gemm2(B, S_hh_x_B_t) + S_v
    S_vh = gemm2(B, S_hh)

    # use potrf to compute the Cholesky decomposition S_vv = LL^T
    S_vv_chol = potrf(S_vv)

    # K = S_hh X with X = B^T S_vv^{-1}
    # We have X = B^T S_vv^{-1} => X S_vv = B^T => X LL^T = B^T
    # We can thus obtain X by solving two linear systems involving L
    K = trsm(S_vv_chol, trsm(S_vv_chol, S_hh_x_B_t, rightside=1, transpose=1),
rightsided=1)

    delta = v[t] - mu_v
    f_t = mu_h + gemm2(K, delta)

    ImKB = eye_h - gemm2(K, B)
    F_t = (gemm2(ImKB, gemm2(S_hh, ImKB, transpose_b=True))
          + gemm2(K, gemm2(S_v, K, transpose_b=True), name="Ft"))

    # save filtered covariance and mean
    F_seq.append(F_t)
    f_seq.append(f_t)

    # compute the likelihood using mu_v and L (LL^T = S_vv)
    Z = trsm(S_vv_chol, trsm(S_vv_chol, delta), transpose=1)
    log_p = (-0.5 * (mx.nd.reshape(gemm2(delta, Z, transpose_a=True), shape=(0,)),
name="reshaped")
          + D*np.log(2.0 * np.pi)) - sumlogdiag(S_vv_chol))
    log_p_seq.append(log_p)

return f_seq, F_seq, log_p_seq

```

In [7]: f\_seq, F\_seq, \_ = LDS\_forward(v, A, B, S\_h, S\_v)

## Calculate the filtered mean and variance

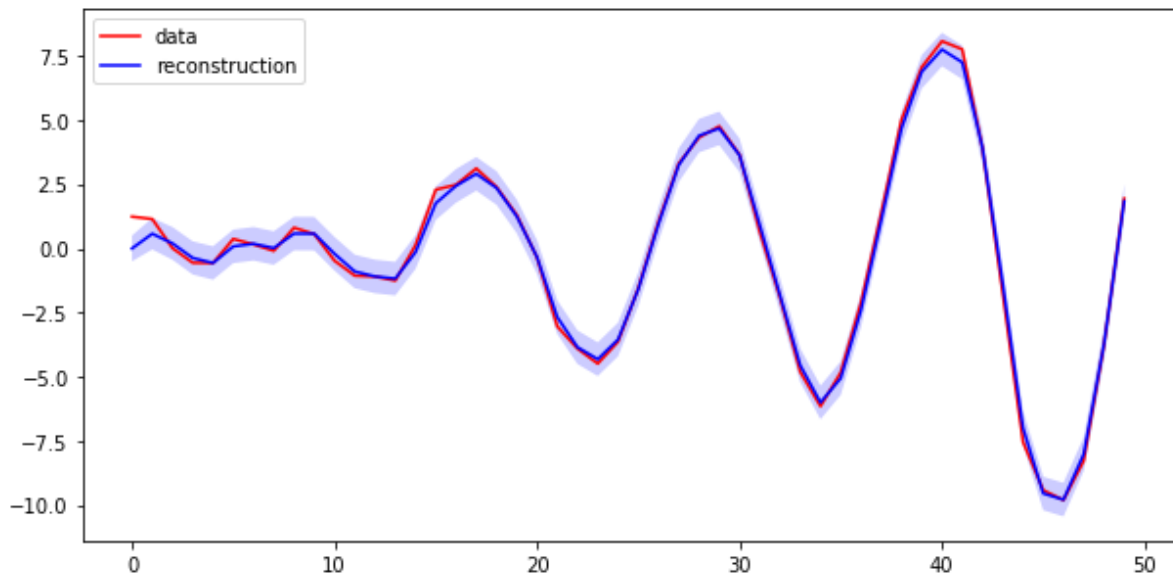
Given  $p(h_t|v_{0:t}) = \mathcal{N}(\mu_t, \Sigma_t)$ , we can compute the distribution of the reconstructed observations

$$p(\widehat{v}_t) = \mathcal{N}(B\mu_t, B\Sigma_t B^T + \sigma^2).$$

```
In [8]: from funtools import reduce
B_np = B.asnumpy()
h_states = reduce(lambda x, y: np.hstack((x,y)), [ff.asnumpy() for ff in f_seq])
v_filtered_mean = B.asnumpy().dot(h_states).reshape((T,))
```

```
In [9]: v_filtered_var = np.sqrt(
    np.array([B_np.dot(ff.asnumpy()).dot(B_np.T) + np.square(sigma) for ff in
F_seq])).reshape((T,))
```

```
In [10]: plt.plot(v.asnumpy(), color="r")
plt.plot(v_filtered_mean, color="b")
x = np.arange(T)
plt.fill_between(x, v_filtered_mean-v_filtered_var,
                v_filtered_mean+v_filtered_var,
                facecolor="blue", alpha=0.2)
plt.legend(["data", "reconstruction"];
```



In the next notebook, we will use Kalman filtering as a subroutine in more complex models. In particular, we will show how to do time series forecasting with innovative state space models (ISSMs).

# Exponential Smoothing and Innovation State Space Model (ISSM)

In this notebook we will illustrate the implementation of filtering in innovation state space model (ISSM, for short) using MXNet. Let us first briefly review the basic concepts.

Time series forecasting is a central problem occurring in many applications from optimal inventory management, staff scheduling to topology planning. Given a sequence of measurements  $z_1, \dots, z_T$  observed over time, the problem here is to predict future values of the time series  $z_{T+1}, \dots, z_{T+\tau}$ , where  $\tau$  is referred as the time horizon.

Exponential smoothing (ETS, which stands for *Error, Trend, and Seasonality*) is a family of very successful forecasting methods which are based on the key property that forecasts are weighted combinations of past observations ([Hyndman et. al, 2008](#)).

For example, in simple exponential smoothing, the forecast  $\hat{z}_{T+1}$  for time step  $T + 1$  is written as ([Hyndman, Athanasopoulos, 2012](#))

$$\hat{z}_{T+1} = \hat{z}_T + \alpha(z_T - \hat{z}_T) = \alpha \cdot z_T + (1 - \alpha) \cdot \hat{z}_T,$$

In words, the next step forecast is a convex combination of the most recent observation and forecast. Expanding the above equation, it is clear that the forecast is given by the exponentially weighted average of past observations,

$$\hat{z}_{T+1} = \alpha z_T + \alpha(1 - \alpha)z_{T-1} + \alpha(1 - \alpha)^2 z_{T-2} + \dots.$$

Here  $\alpha > 0$  is a smoothing parameter that controls the weight given to each observation. Note that the recent observations are given more weight than the older observations. In fact the weight given to the past observation decreases exponentially as it gets older and hence the name **exponential smoothing**.

General exponential smoothing methods consider the extensions of simple ETS to include time series patterns such as (linear) trend, various periodic seasonal effects. All ETS methods fall under the category of forecasting methods as the predictions are point forecasts (a single value is predicted for each future time step). On the other hand a statistical model describes the underlying data generation process and has an advantage that it can produce an entire

probability distribution for each of the future time steps. Innovation state space model (ISSM) is an example of such models with considerable flexibility in representing commonly occurring time series patterns and underlie the exponential smoothing methods.

The idea behind ISSMs is to maintain a latent state vector  $l_t$  with recent information about level, trend, and seasonality factors. The state vector  $l_t$  evolves over time adding small *innovation* (i.e., the Gaussian noise) at each time step. The observations are then a linear combination of the components of the current state.

Mathematically, ISSM is specified by two equations

- The state transition equation is given by

$$l_t = F_t l_{t-1} + g_t \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, 1).$$

Note that the innovation strength is controlled by  $g_t$ , i.e.,  $g_t \epsilon_t \sim \mathcal{N}(0, g_t^2)$ .

- The observation equation is given by

$$z_t = a_t^\top l_{t-1} + b_t + \nu_t, \quad \nu_t \sim \mathcal{N}(0, \sigma_t^2)$$

Note that here we allow for an additional term  $b_t$  which can model any deterministic component (exogenous variables).

This describes a fairly generic model allowing the user to encode specific time series patterns using the coefficients  $F$ ,  $a_t$  and thus are problem dependent. The innovation vector  $g_t$  comes in terms of parameters to be learned (the innovation strengths). Moreover, the initial state  $l_0$  has to be specified. We do so by specifying a Gaussian prior distribution  $P(l_0)$ , whose parameters (means, standard deviation) are learned from data as well.

The parameters of the ISSM are typically learned using the maximum likelihood principle. This requires the computation of the log-likelihood of the given observations i.e., computing the probability of the data under the model,  $P(z_1, \dots, z_T)$ . Fortunately, in the previous notebook, we have learned how to compute the log-likelihood as a byproduct of LDS filtering problem.

## Filtering

We remark that ISSM is a special case of linear dynamical system except that the coefficients are allowed to change over time. The filtering equations for ISSM can readily be obtained from the general derivation described in LDS.

Note the change in the notation in the following equations for filtered mean ( $\mu_t$ ) and filtered variance ( $S_t$ ) because of the conflict of notation for the ISSM coefficient  $F$ . Also note that the deterministic part  $b_t$  needs to be subtracted from the observations  $[z_t]$ .

$$\mu_h = F_t \mu_{t-1} \quad \mu_v = a_t^\top \mu_h$$

$$\Sigma_{hh} = F_t S_{t-1} F_t^\top + g_t g_t^\top \quad \sigma_v^2 = a_t^\top \Sigma_{hh} a_t + \sigma_t^2$$

$$K_t = \frac{1}{\sigma_v^2} \Sigma_{hh} a_t$$

$$\mu_t = \mu_h + K(z_t - b_t - \mu_v) \quad S_t = (I - K_t a_t^\top) \Sigma_{hh} (I - K_t a_t^\top)^\top + \sigma_t^2 K_t K_t^\top$$

```
In [1]: import mxnet as mx
from mxnet.ndarray import linalg_gemm2 as gemm2
import mxnet.ndarray as nd
```

## ISSM Filtering Function

```
In [2]: def ISSM_filter(z, b, F, a, g, sigma, m_prior, S_prior):

    H = F.shape[0] # dim of latent state
    T = z.shape[0] # num of observations

    eye_h = nd.array(np.eye(H))

    mu_seq = []
    S_seq = []
    log_p_seq = []

    for t in range(T):

        if t == 0:
            # At the first time step, use the prior
            mu_h = m_prior
            S_hh = S_prior
        else:
            # Otherwise compute using update eqns.
            F_t = F[:, :, t]
            g_t = g[:, t].reshape((H,1))

            mu_h = gemm2(F_t, mu_t)
            S_hh = gemm2(F_t, gemm2(S_t, F_t, transpose_b=1)) + \
                    gemm2(g_t, g_t, transpose_b=1)

            a_t = a[:, t].reshape((H,1))
            mu_v = gemm2(mu_h, a_t, transpose_a=1)

            # Compute the Kalman gain (vector)
            S_hh_x_a_t = gemm2(S_hh, a_t)

            sigma_t = sigma[t]
            S_vv = gemm2(a_t, S_hh_x_a_t, transpose_a=1) + nd.square(sigma_t)
            kalman_gain = nd.broadcast_div(S_hh_x_a_t, S_vv)

            # Compute the error (delta)
            delta = z[t] - b[t] - mu_v

            # Filtered estimates
            mu_t = mu_h + gemm2(kalman_gain, delta)
```

```

    # Joseph's symmetrized update for covariance:
    ImKa = nd.broadcast_sub(eye_h, gemm2(kalman_gain, a_t, transpose_b=1))
    S_t = gemm2(gemm2(ImKa, S_hh), ImKa, transpose_b=1) + \
        nd.broadcast_mul(gemm2(kalman_gain, kalman_gain, transpose_b=1),
nd.square(sigma_t))

    # Likelihood term
    log_p = (-0.5 * (delta * delta / S_vv
                    + np.log(2.0 * np.pi)
                    + nd.log(S_vv))
            )

    mu_seq.append(mu_t)
    S_seq.append(S_t)
    log_p_seq.append(log_p)

return mu_seq, S_seq, log_p_seq

```

## Data

We will use the [beer shipment dataset](#) to illustrate two specific instances of ISSM models.

```

In [3]: import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (12, 5)
import seaborn as sns

```

```

In [4]: df = pd.read_csv("./data/fourweekly-totals-of-beer-shipme.csv", header=0)

```

```

In [5]: df.set_index("Week")

# get the time series
ts = df.values[:,1]

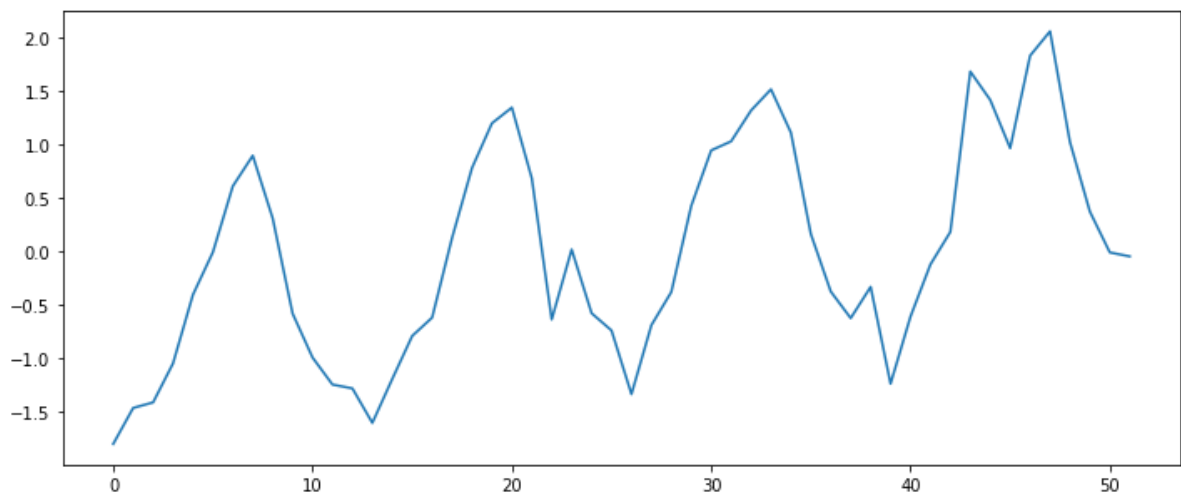
# Let us normalize the time series
ts = np.array((ts - np.mean(ts)) / np.std(ts), dtype=np.double)

```

```

In [6]: plt.plot(ts);

```



## Level ISSM

The simplest possible ISSM maintains a level component only. Abusing the notation and let  $l_t$  denote *level*, the level ISSM can be written as

$$l_t = \delta l_{t-1} + \alpha \epsilon_t.$$

Or in ISSM terminology,

$$a_t = [\delta], \quad F_t = [\delta], \quad g_t = [\alpha], \quad \alpha > 0.$$

The level  $l_t \in \mathbb{R}$  evolves over time by adding a random innovation  $\alpha \epsilon_t \sim \mathcal{N}(0, \alpha^2)$  to the previous level, so that  $\alpha$  specifies the amount of level drift over time. At time  $t$ , the previous level  $l_{t-1}$  is used in the prediction  $z_t$  and then the level is updated. The damping factor  $\delta \in (0, 1]$  allows the “damping” of the level. The initial state prior  $P(l_0)$  is given by  $l_0 \sim N(\mu_0, \sigma_0^2)$ . For Level-ISSM, we learn the parameters  $\alpha > 0, \mu_0, \sigma_0 > 0$ .

Here we will fix the parameters for the illustration of filtering. Learning of the parameters will be discussed in another notebook.

```
In [7]: latent_dim = 1
        T          = len(ts)

        # Set the coefficients of the ISSM
        delta      = 1.0
        F          = delta * nd.ones((1, 1, T))
        a          = delta * nd.ones((1, T))

        # Set the parameters of the ISSM
        alpha      = 0.5
        g          = alpha * nd.ones((1, T))

        m_prior    = nd.zeros((latent_dim, 1))
        S_prior    = nd.zeros((latent_dim, latent_dim))
        sigma      = 0.5 * nd.ones((T, 1))
        b          = nd.zeros((T, 1))
        z          = nd.array(ts).reshape((T, 1))
```

```
In [8]: mu_seq, S_seq, _ = ISSM_filter(z, b, F, a, g, sigma, m_prior, S_prior)
```

## Calculate the filtered mean and variance of observations

Given  $p(l_{t-1} | z_{1:t}) = \mathcal{N}(\mu_t, S_t)$ , we can compute the distribution of the reconstructed observations

$$p(\widehat{z}_t) = \mathcal{N}(a_t^T \mu_t, a_t^T S_t a_t + \sigma_t^2).$$

```
In [9]: from functools import reduce

        def reconstruct(mu_seq, S_seq):
            a_np = a.asnumpy()
            T = len(mu_seq)
            sigma_np = sigma.asnumpy()

            v_filtered_mean = np.array([a_np[:, t].dot(mu_t.asnumpy())
```

```

        for t, mu_t in enumerate(mu_seq)]
        ).reshape(T, )

    v_filtered_std = np.sqrt(np.array([a_np[:, t].dot(S_t.asnumpy()).dot(a_np[:, t]) +
        np.square(sigma_np[t])
        for t, S_t in enumerate(S_seq)]).reshape((T,)))

    return v_filtered_mean, v_filtered_std

```

```
In [10]: reconst_mean, reconst_std = reconstruct(mu_seq, S_seq)
```

## Forecast

One advantage of the ISSM model is that one can obtain the complete probability distribution for each of the future time steps:

$$p(\widehat{z_{T+t}}) = \mathcal{N}(a_{T+t}^T \mu_{T+t}, a_{T+t}^T S_{T+t} a_{T+t} + \sigma_{T+t}^2), \quad t > 0$$

$$p(l_{T+t}) = \mathcal{N}(F \mu_{T+t-1}, F S_{T+t-1} F^T + g_{T+t} g_{T+t}^T)$$

```
In [11]: def forecast(mu_last_state, S_last_state, F, a, g, sigma, horizon):

    forecasts_mean = []
    forecasts_std = []

    mu_last_state = mu_last_state.asnumpy()
    S_last_state = S_last_state.asnumpy()
    F = F.asnumpy()
    a = a.asnumpy()
    g = g.asnumpy()
    sigma = sigma.asnumpy()

    for t in range(horizon):
        a_t = a[:, t]
        forecast_mean = a_t.dot(mu_last_state)[0]
        forecast_std = a_t.dot(S_last_state).dot(a_t) + np.square(sigma[t])[0]

        forecasts_mean.append(forecast_mean)
        forecasts_std.append(forecast_std)

        mu_last_state = F[:, :, t].dot(mu_last_state)
        S_last_state = F[:, :, t].dot(S_last_state).dot(F[:, :, t].T)

    return np.array(forecasts_mean), np.array(forecasts_std)

```

```
In [12]: # Let us use the same coefficients (constant over time) for the future as well
forecasts_mean, forecasts_std = forecast(mu_seq[-1],
                                         S_seq[-1],
                                         F, a, g, sigma, horizon=13)

```

## Plot the reconstruction as well as the forecasts

```
In [13]: def plot_reconstruction_forecasts(v_filtered_mean, v_filtered_std, forecasts_mean,
forecasts_std):

    plt.plot(ts, color="r")
    plt.plot(v_filtered_mean, color="b")
    T = len(v_filtered_mean)
    x = np.arange(T)
    plt.fill_between(x, v_filtered_mean-v_filtered_std,
                    v_filtered_mean+v_filtered_std,

```



```

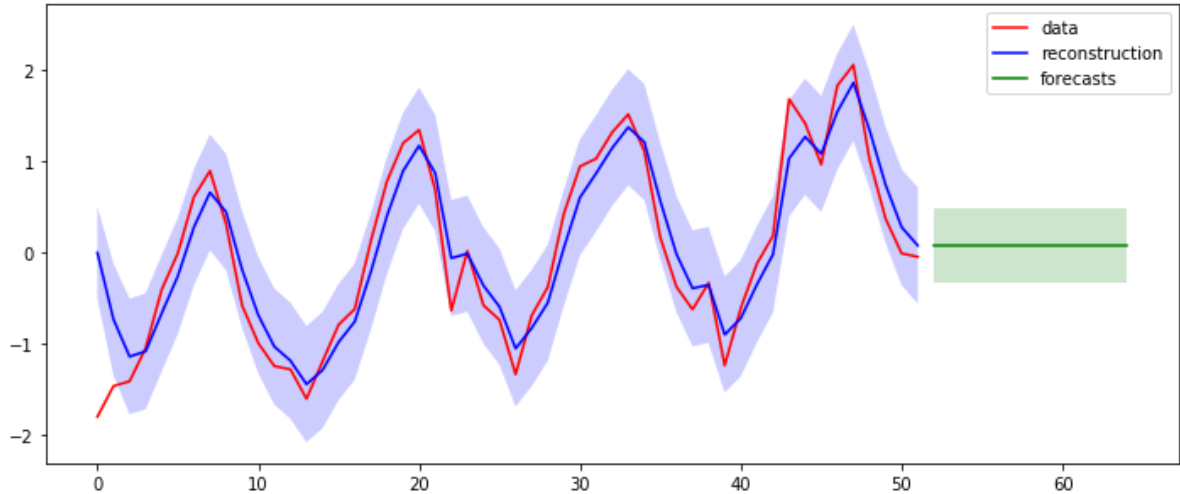
        facecolor="blue", alpha=0.2)

plt.plot(np.arange(T, T+len(forecasts_mean)), forecasts_mean, color="g")
plt.fill_between(np.arange(T, T+len(forecasts_mean)), forecasts_mean-forecasts_std,
                forecasts_mean+forecasts_std,
                facecolor="green", alpha=0.2)

plt.legend(["data", "reconstruction", "forecasts"]);

```

In [14]: `plot_reconstruction_forecasts(reconst_mean, reconst_std, forecasts_mean, forecasts_std)`



## Level Trend ISSM

We can model a piecewise linear random process by using a two-dimensional latent state  $l_t \in \mathbb{R}^2$ , where one dimension represents the level (again with a slight abusing of notation,  $l$ ) and the other represents the trend (slope)  $b$ .

$$\begin{aligned}
 l_t &= \delta l_{t-1} + \gamma b_{t-1} + \alpha \cdot \epsilon_t \\
 b_t &= \gamma b_{t-1} + \beta \cdot \epsilon_t
 \end{aligned}$$

In ISSM framework, such a (Damped) LevelTrend-ISSM is given by

$$a_t = \begin{bmatrix} \delta \\ \gamma \end{bmatrix}, \quad F_t = \begin{bmatrix} \delta & \gamma \\ 0 & \gamma \end{bmatrix}, \quad g_t = \begin{bmatrix} \alpha \\ \beta \end{bmatrix},$$

where  $\alpha > 0, \beta > 0$  and the damping factors  $\delta, \gamma \in (0, 1]$ . Both the level and slope components evolve over time by adding innovations  $\alpha\epsilon_t$  and  $\beta\epsilon_t$  respectively, so that  $\beta > 0$  is the innovation strength for the slope. The level at time  $t$  is the sum of level at  $t - 1$  and slope at  $t - 1$  (linear prediction) modulo the damping factors for level  $\delta$  and growth  $\gamma$ .

```

In [15]: latent_dim = 2
         T           = len(ts)

         # Set the coefficients of the ISSM
         damp_fact = 1.0
         damp_growth = 1.0

         # Set the parameters of the ISSM

```

```

alpha      = 0.5
beta       = 0.1
g_t        = nd.array([alpha, beta])
g          = nd.repeat(g_t, T).reshape((latent_dim, T))

# F and a are constant over time
F_t = nd.reshape(nd.array([damp_fact, damp_growth, 0, damp_growth]), (latent_dim,
latent_dim))
a_t = nd.array([damp_fact, damp_growth])
F   = nd.repeat(F_t, T).reshape((latent_dim, latent_dim, T))
a   = nd.repeat(a_t, T).reshape((latent_dim, T))

m_prior    = nd.zeros((latent_dim, 1))
S_prior    = nd.zeros((latent_dim, latent_dim))
sigma      = 0.5 * nd.ones((T, 1))
b          = nd.zeros((T, 1))
z          = nd.array(ts).reshape((T, 1))

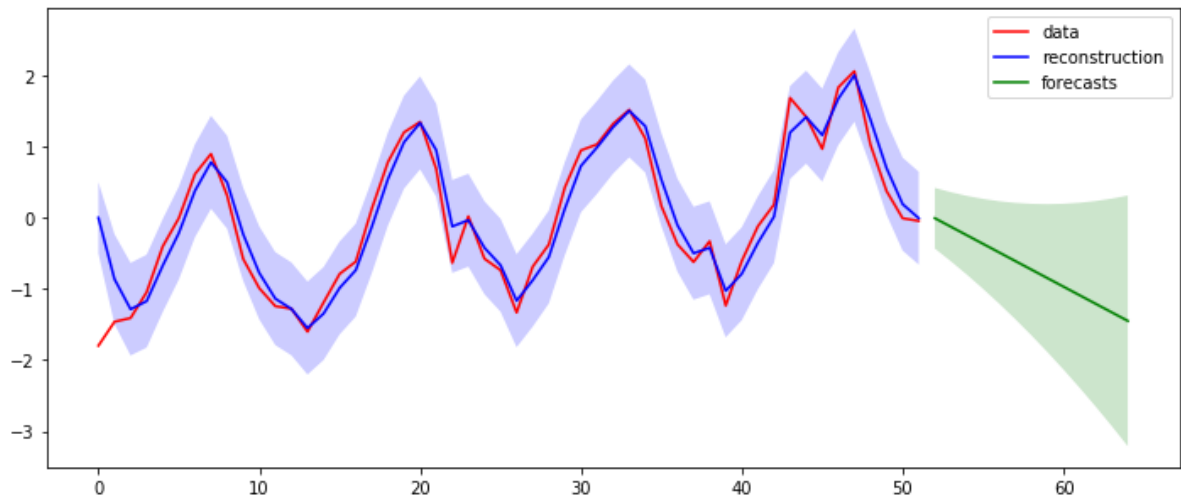
```

```
In [16]: mu_seq, S_seq, _ = ISSM_filter(z, b, F, a, g, sigma, m_prior, S_prior)
```

```
In [17]: # Let us use the same coefficients (constant over time) for the future as well
forecasts_mean, forecasts_std = forecast(mu_seq[-1],
S_seq[-1],
F, a, g, sigma, horizon=13)
```

## Plot the reconstruction as well as the forecasts

```
In [18]: reconst_mean, reconst_std = reconstruct(mu_seq, S_seq)
plot_reconstruction_forecasts(reconst_mean, reconst_std, forecasts_mean, forecasts_std)
```



# Generative Adversarial Networks

Throughout most of this book, we've talked about how to make predictions. In some form or another, we used deep neural networks learned mappings from data points to labels. This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs. Classifiers and regressors are both examples of discriminative learning. And neural networks trained by backpropagation have upended everything we thought we knew about discriminative learning on large complicated datasets. Classification accuracies on high-res images has gone from useless to human-level (with some caveats) in just 5-6 years. We'll spare you another spiel about all the other discriminative tasks where deep neural networks do astoundingly well.

But there's more to machine learning than just solving discriminative tasks. For example, given a large dataset, without any labels, we might want to learn a model that concisely captures the characteristics of this data. Given such a model, we could sample synthetic data points that resemble the distribution of the training data. For example, given a large corpus of photographs of faces, we might want to be able to generate a *new* photorealistic image that looks like it might plausibly have come from the same dataset. This kind of learning is called *generative modeling*.

Until recently, we had no method that could synthesize novel photorealistic images. But the success of deep neural networks for discriminative learning opened up new possibilities. One big trend over the last three years has been the application of discriminative deep nets to overcome challenges in problems that we don't generally think of as supervised learning problems. The recurrent neural network language models are one example of using a discriminative network (trained to predict the next character) that once trained can act as a generative model.

In 2014, a young researcher named Ian Goodfellow introduced [Generative Adversarial Networks \(GANs\)](#) a clever new way to leverage the power of discriminative models to get good generative models. GANs made quite a splash so it's quite likely you've seen the images before. For instance, using a GAN you can create fake images of bedrooms, as done by [Radford et al. in 2015](#) and depicted below.

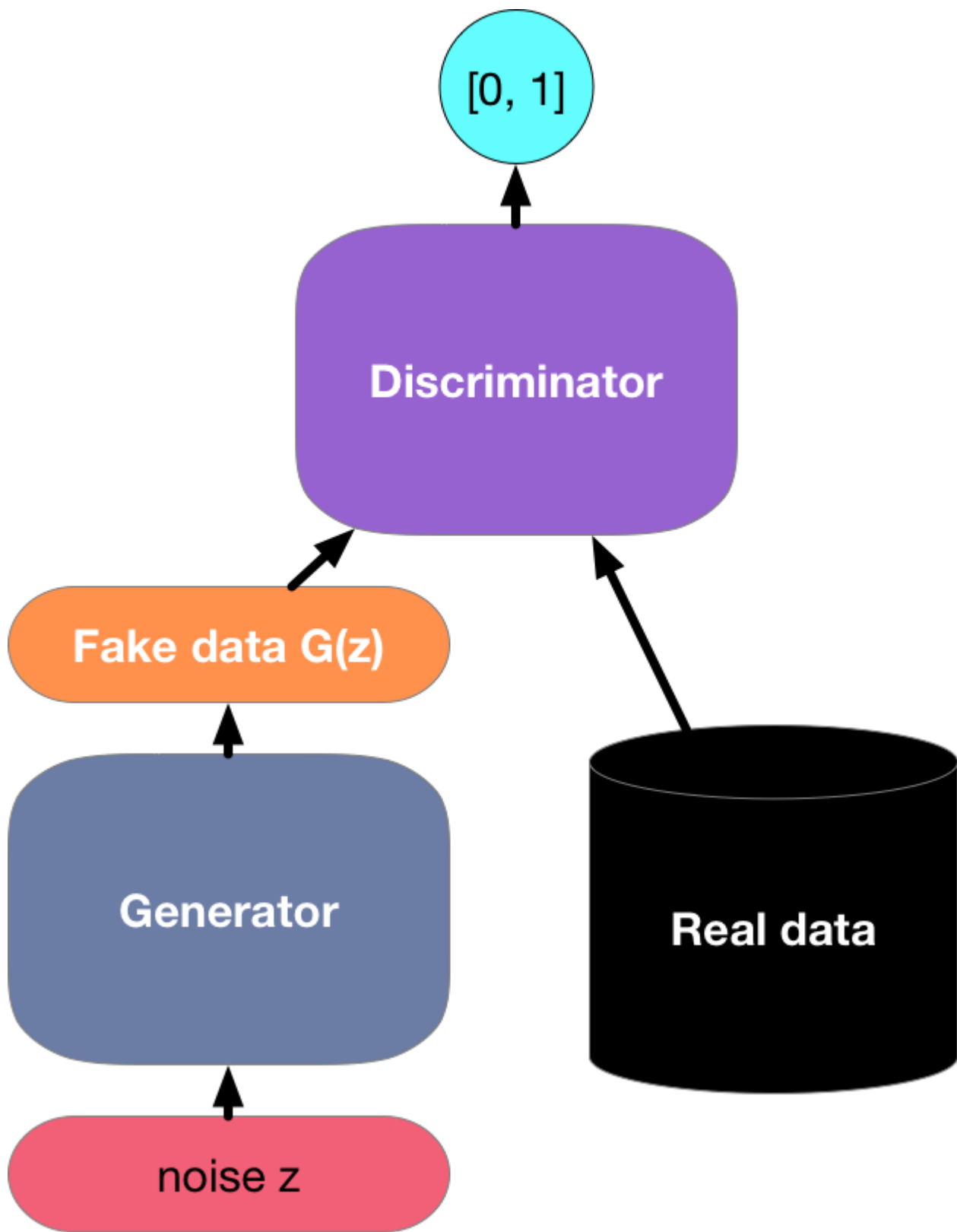


At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data. In statistics, this is called a two-sample test - a test to answer the question whether datasets  $X = \{x_1, \dots, x_n\}$  and  $X' = \{x'_1, \dots, x'_n\}$  were drawn from the same distribution. The main difference between most statistics papers and GANs is that the latter use this idea in a constructive way. In other words, rather than just training a model to say 'hey, these two datasets don't look like they came from the same distribution', they use the two-sample test to provide training signal to a generative model. This allows us to improve the data generator until it generates something that resembles the real data. At the very least, it needs to fool the classifier. And if our classifier is a state of the art deep neural network.

As you can see, there are two pieces to GANs - first off, we need a device (say, a deep network but it really could be anything, such as a game rendering engine) that might potentially be able to generate data that looks just like the real thing. If we are dealing with images, this needs to generate images. If we're dealing with speech, it needs to generate audio sequences, and so on. We call this the *generator network*. The second component is the *discriminator network*. It attempts to distinguish fake and real data from each other. Both networks are in competition with each other. The generator network attempts to fool the discriminator network. At that point, the discriminator network adapts to the new fake data. This information, in turn is used to improve the generator network, and so on.

**Generator** \* Draw some parameter  $z$  from a source of randomness, e.g. a normal distribution  $z \sim \mathcal{N}(0, 1)$ . \* Apply a function  $f$  such that we get  $x' = G(u, w)$  \* Compute the gradient with respect to  $w$  to minimize  $\log p(y = \text{fake} | x')$

**Discriminator** \* Improve the accuracy of a binary classifier  $f$ , i.e. maximize  $\log p(y = \text{fake} | x')$  and  $\log p(y = \text{true} | x)$  for fake and real data respectively.



In short, there are two optimization problems running simultaneously, and the optimization terminates if a stalemate has been reached. There are lots of further tricks and details on how to modify this basic setting. For instance, we could try solving this problem in the presence of side information. This leads to cGAN, i.e. conditional Generative Adversarial Networks. We can change the way how we detect whether real and fake data look the same. This leads to wGAN (Wasserstein GAN), kernel-inspired GANs and lots of other settings, or we could change how closely we look at the objects. E.g. fake images might look real at the texture level but not so at the larger level, or vice versa.

Many of the applications are in the context of images. Since this takes too much time to solve in a Jupyter notebook on a laptop, we're going to content ourselves with fitting a much simpler distribution. We will illustrate what happens if we use GANs to build the world's most inefficient estimator of parameters for a Gaussian. Let's get started.

```
In [1]: from __future__ import print_function
import matplotlib as mpl
from matplotlib import pyplot as plt
import mxnet as mx
from mxnet import gluon, autograd, nd
from mxnet.gluon import nn
import numpy as np

ctx = mx.cpu()
```

## Generate some 'real' data

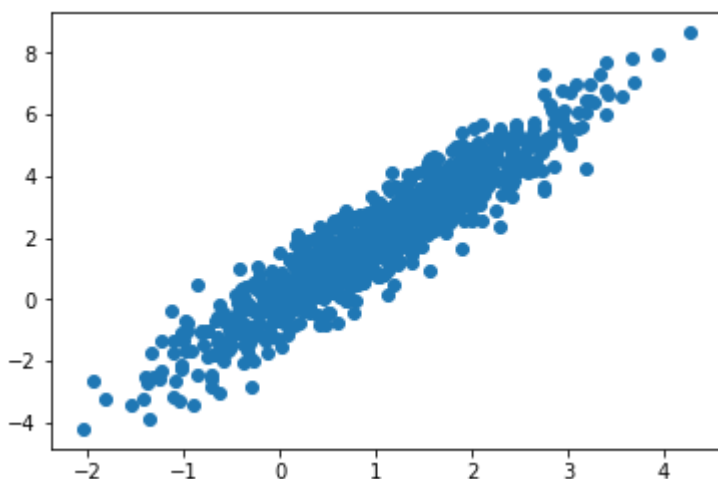
Since this is going to be the world's lamest example, we simply generate data drawn from a Gaussian. And let's also set a context where we'll do most of the computation.

```
In [2]: X = nd.random_normal(shape=(1000, 2))
A = nd.array([[1, 2], [-0.1, 0.5]])
b = nd.array([1, 2])
X = nd.dot(X,A) + b
Y = nd.ones(shape=(1000,1))

# and stick them into an iterator
batch_size = 4
train_data = mx.io.NDArrayIter(X, Y, batch_size, shuffle=True)
```

Let's see what we got. This should be a Gaussian shifted in some rather arbitrary way with mean  $b$  and covariance matrix  $A^T A$ .

```
In [3]: plt.scatter(X[:, 0].asnumpy(), X[:, 1].asnumpy())
plt.show()
print("The covariance matrix is")
print(nd.dot(A, A.T))
```



The covariance matrix is

```
[[ 5.          0.89999998]
 [ 0.89999998  0.25999999]]
<NDArray 2x2 @cpu(0)>
```

## Defining the networks

Next we need to define how to fake data. Our generator network will be the simplest network possible - a single layer linear model. This is since we'll be driving that linear network with a Gaussian data generator. Hence, it literally only needs to learn the parameters to fake things perfectly. For the discriminator we will be a bit more discriminating: we will use an MLP with 3 layers to make things a bit more interesting.

The cool thing here is that we have *two* different networks, each of them with their own gradients, optimizers, losses, etc. that we can optimize as we please.

```
In [4]: # build the generator
netG = nn.Sequential()
with netG.name_scope():
    netG.add(nn.Dense(2))

# build the discriminator (with 5 and 3 hidden units respectively)
netD = nn.Sequential()
with netD.name_scope():
    netD.add(nn.Dense(5, activation='tanh'))
    netD.add(nn.Dense(3, activation='tanh'))
    netD.add(nn.Dense(2))

# Loss
loss = gluon.loss.SoftmaxCrossEntropyLoss()

# initialize the generator and the discriminator
netG.initialize(mx.init.Normal(0.02), ctx=ctx)
netD.initialize(mx.init.Normal(0.02), ctx=ctx)

# trainer for the generator and the discriminator
trainerG = gluon.Trainer(netG.collect_params(), 'adam', {'learning_rate': 0.01})
trainerD = gluon.Trainer(netD.collect_params(), 'adam', {'learning_rate': 0.05})
```

## Setting up the training loop

We are going to iterate over the data a few times. To make life simpler we need a few variables

```
In [5]: real_label = mx.nd.ones((batch_size,), ctx=ctx)
fake_label = mx.nd.zeros((batch_size,), ctx=ctx)
metric = mx.metric.Accuracy()

# set up logging
from datetime import datetime
import os
import time
```

## Training loop



```

In [6]: stamp = datetime.now().strftime('%Y_%m_%d-%H_%M')
for epoch in range(10):
    tic = time.time()
    train_data.reset()
    for i, batch in enumerate(train_data):
        #####
        # (1) Update D network: maximize Log(D(x)) + Log(1 - D(G(z)))
        #####
        # train with real_t
        data = batch.data[0].as_in_context(ctx)
        noise = nd.random_normal(shape=(batch_size, 2), ctx=ctx)

        with autograd.record():
            real_output = netD(data)
            errD_real = loss(real_output, real_label)

            fake = netG(noise)
            fake_output = netD(fake.detach())
            errD_fake = loss(fake_output, fake_label)
            errD = errD_real + errD_fake
            errD.backward()

        trainerD.step(batch_size)
        metric.update([real_label,], [real_output,])
        metric.update([fake_label,], [fake_output,])

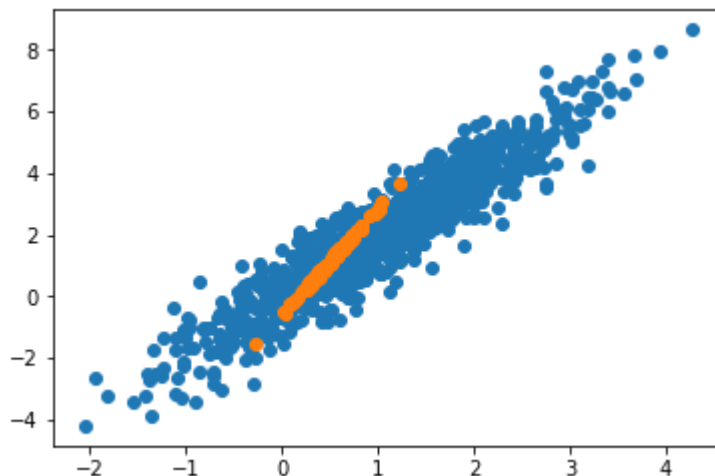
        #####
        # (2) Update G network: maximize Log(D(G(z)))
        #####
        with autograd.record():
            output = netD(fake)
            errG = loss(output, real_label)
            errG.backward()

        trainerG.step(batch_size)

    name, acc = metric.get()
    metric.reset()
    print('\nbinary training acc at epoch %d: %s=%f' % (epoch, name, acc))
    print('time: %f' % (time.time() - tic))
    noise = nd.random_normal(shape=(100, 2), ctx=ctx)
    fake = netG(noise)
    plt.scatter(X[:, 0].asnumpy(), X[:, 1].asnumpy())
    plt.scatter(fake[:, 0].asnumpy(), fake[:, 1].asnumpy())
    plt.show()

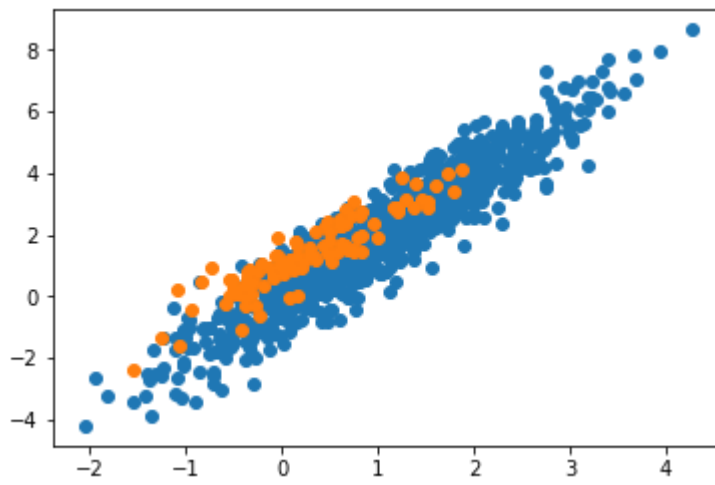
```

binary training acc at epoch 0: accuracy=0.764500  
time: 5.838877

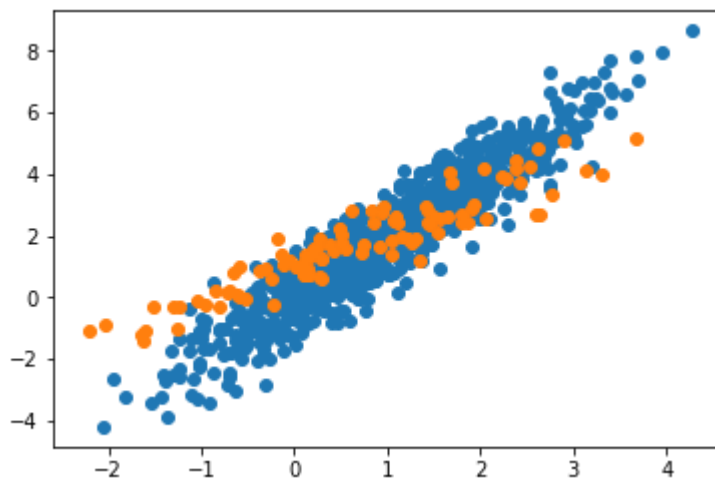


binary training acc at epoch 1: accuracy=0.639000  
time: 6.052228

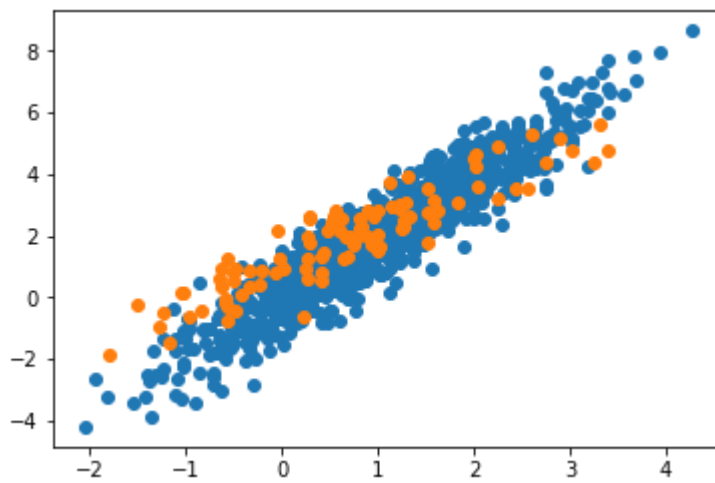




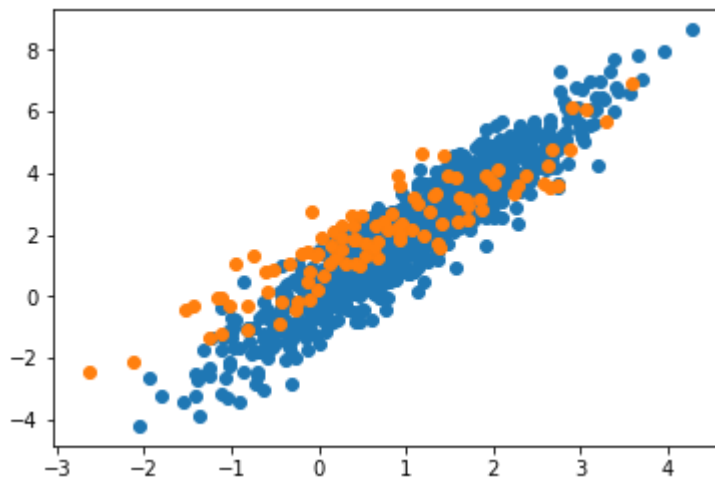
binary training acc at epoch 2: accuracy=0.551000  
time: 5.773329



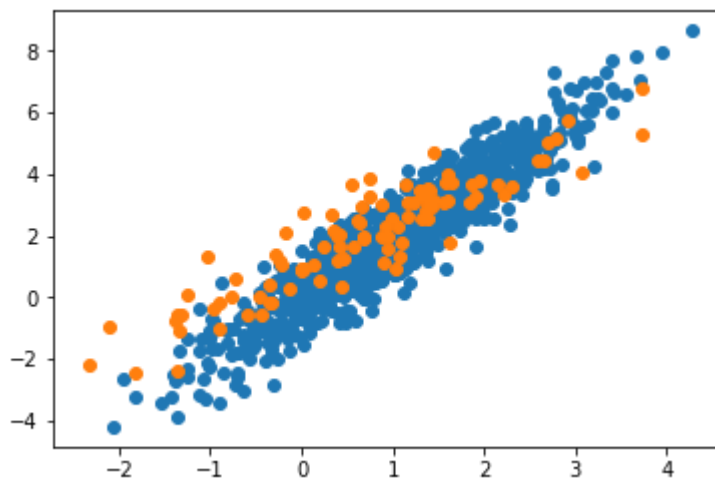
binary training acc at epoch 3: accuracy=0.522000  
time: 5.613472



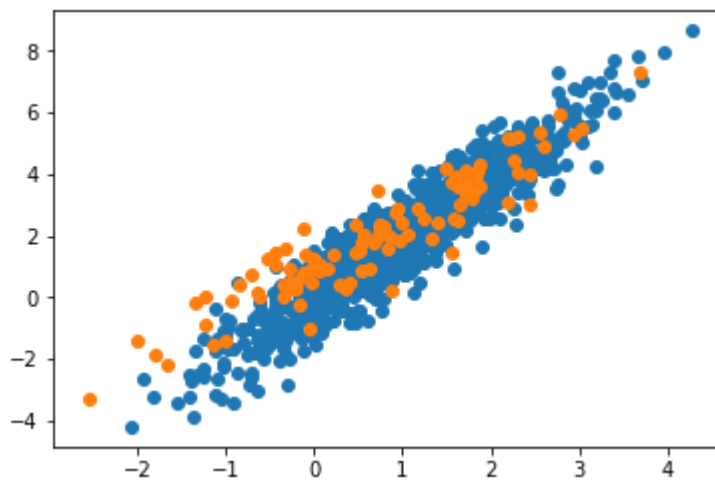
binary training acc at epoch 4: accuracy=0.498000  
time: 6.069607



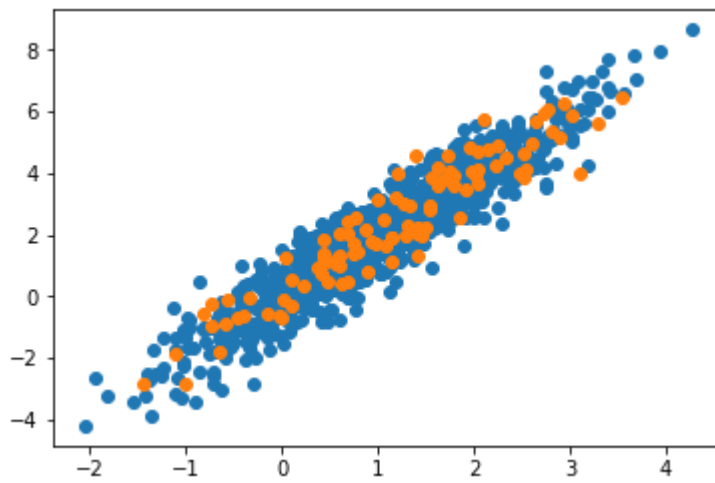
binary training acc at epoch 5: accuracy=0.496500  
time: 5.800509



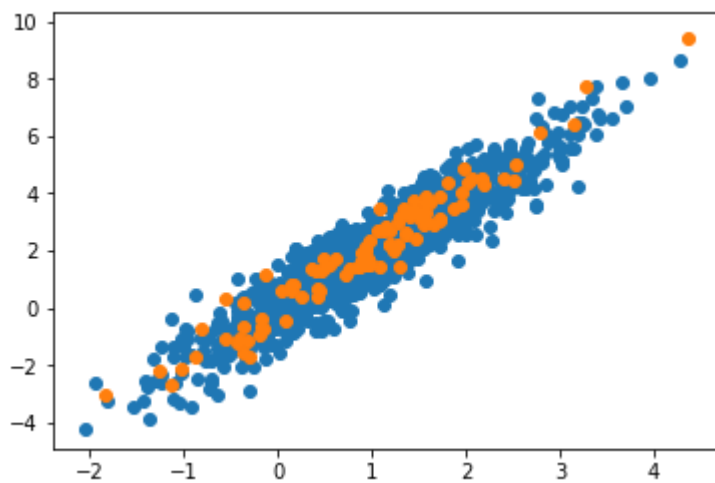
binary training acc at epoch 6: accuracy=0.498500  
time: 5.982538



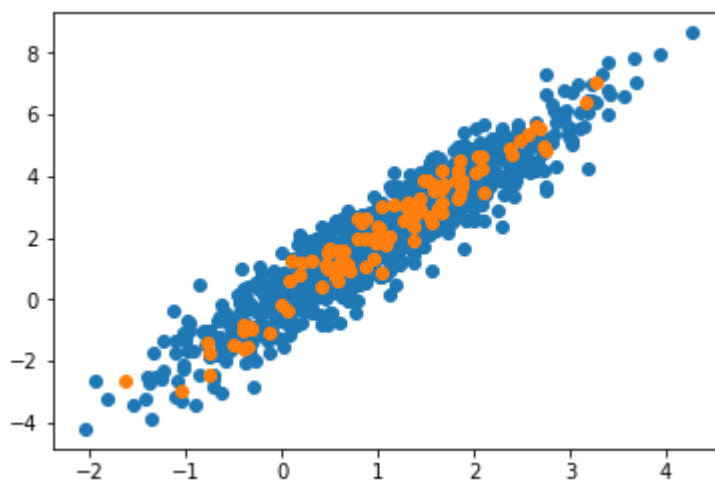
binary training acc at epoch 7: accuracy=0.515500  
time: 6.017519



```
binary training acc at epoch 8: accuracy=0.500000  
time: 6.143714
```



```
binary training acc at epoch 9: accuracy=0.499000  
time: 6.123487
```

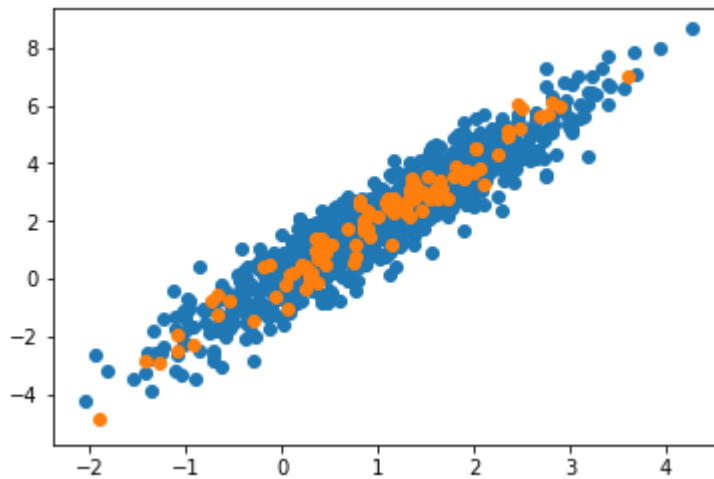


## Checking the outcome

Let's now generate some fake data and check whether it looks real.

```
In [7]: noise = mx.nd.random_normal(shape=(100, 2), ctx=ctx)
fake = netG(noise)

plt.scatter(X[:, 0].asnumpy(), X[:, 1].asnumpy())
plt.scatter(fake[:, 0].asnumpy(), fake[:, 1].asnumpy())
plt.show()
```



## Conclusion

A word of caution here - to get this to converge properly, we needed to adjust the learning rates *very carefully*. And for Gaussians, the result is rather mediocre - a simple mean and covariance estimator would have worked *much better*. However, whenever we don't have a really good idea of what the distribution should be, this is a very good way of faking it to the best of our abilities. Note that a lot depends on the power of the discriminating network. If it is weak, the fake can be very different from the truth. E.g. in our case it had trouble picking up anything along the axis of reduced variance. In summary, this isn't exactly easy to set and forget. One nice resource for dirty practioner's knowledge is [Soumith Chintala's handy list of tricks](#) for how to babysit GANs.

For whinges or inquiries, [open an issue on GitHub](#).

# Deep Convolutional Generative Adversarial Networks

In [our introduction to generative adversarial networks \(GANs\)](#), we introduced the basic ideas behind how GANs work. We showed that they can draw samples from some simple, easy-to-sample distribution, like a uniform or normal distribution, and transform them into samples that appear to match the distribution of some data set. And while our example of matching a 2D Gaussian distribution got the point across, it's not especially exciting.

In this notebook, we'll demonstrate how you can use GANs to generate photorealistic images. We'll be basing our models on the deep convolutional GANs introduced in [this paper](#). We'll borrow the convolutional architecture that have proven so successful for discriminative computer vision problems and show how via GANs, they can be leveraged to generate photorealistic images.

In this tutorial, concentrate on the [LWF Face Dataset](#), which contains roughly 13000 images of faces. By the end of the tutorial, you'll know how to generate photo-realistic images of your own, given any dataset of images. First, we'll the the preliminaries out of the way.

```
In [1]: from __future__ import print_function
import os
import matplotlib as mpl
import tarfile
import matplotlib.image as mpimg
from matplotlib import pyplot as plt

import mxnet as mx
from mxnet import gluon
from mxnet import ndarray as nd
from mxnet.gluon import nn, utils
from mxnet import autograd
import numpy as np
```

## Set training parameters

```
In [2]: epochs = 2 # Set low by default for tests, set higher when you actually run this code.
batch_size = 64
latent_z_size = 100

use_gpu = True
ctx = mx.gpu() if use_gpu else mx.cpu()

lr = 0.0002
beta1 = 0.5
```

# Download and preprocess the LFW Face Dataset

```
In [3]: lfw_url = 'http://vis-www.cs.umass.edu/lfw/lfw-deepfunneled.tgz'
data_path = 'lfw_dataset'
if not os.path.exists(data_path):
    os.makedirs(data_path)
    data_file = utils.download(lfw_url)
    with tarfile.open(data_file) as tar:
        tar.extractall(path=data_path)
```

Downloading lfw-deepfunneled.tgz from http://vis-www.cs.umass.edu/lfw/lfw-deepfunneled.tgz...

First, we resize images to size  $64 \times 64$ . Then, we normalize all pixel values to the  $[-1, 1]$  range.

```
In [4]: target_wd = 64
target_ht = 64
img_list = []

def transform(data, target_wd, target_ht):
    # resize to target_wd * target_ht
    data = mx.image.imresize(data, target_wd, target_ht)
    # transpose from (target_wd, target_ht, 3)
    # to (3, target_wd, target_ht)
    data = nd.transpose(data, (2,0,1))
    # normalize to [-1, 1]
    data = data.astype(np.float32)/127.5 - 1
    # if image is greyscale, repeat 3 times to get RGB image.
    if data.shape[0] == 1:
        data = nd.tile(data, (3, 1, 1))
    return data.reshape((1,) + data.shape)

for path, _, fnames in os.walk(data_path):
    for fname in fnames:
        if not fname.endswith('.jpg'):
            continue
        img = os.path.join(path, fname)
        img_arr = mx.image.imread(img)
        img_arr = transform(img_arr, target_wd, target_ht)
        img_list.append(img_arr)
train_data = mx.io.NDArrayIter(data=nd.concatenate(img_list), batch_size=batch_size)
```

Visualize 4 images:

```
In [5]: def visualize(img_arr):
plt.imshow(((img_arr.asnumpy().transpose(1, 2, 0) + 1.0) * 127.5).astype(np.uint8))
plt.axis('off')

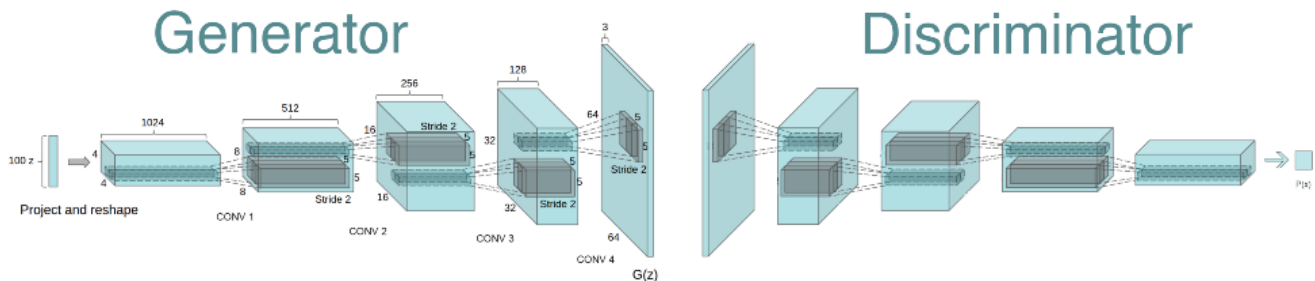
for i in range(4):
    plt.subplot(1,4,i+1)
    visualize(img_list[i + 10][0])
plt.show()
```



# Defining the networks

The core to the DCGAN architecture uses a standard CNN architecture on the discriminative model. For the generator, convolutions are replaced with upconvolutions, so the representation at each layer of the generator is actually successively larger, as it maps from a low-dimensional latent vector onto a high-dimensional image.

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batch normalization in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.



```
In [6]: # build the generator
nc = 3
ngf = 64
netG = nn.Sequential()
with netG.name_scope():
    # input is Z, going into a convolution
    netG.add(nn.Conv2DTranspose(ngf * 8, 4, 1, 0, use_bias=False))
    netG.add(nn.BatchNorm())
    netG.add(nn.Activation('relu'))
    # state size. (ngf*8) x 4 x 4
    netG.add(nn.Conv2DTranspose(ngf * 4, 4, 2, 1, use_bias=False))
    netG.add(nn.BatchNorm())
    netG.add(nn.Activation('relu'))
    # state size. (ngf*8) x 8 x 8
    netG.add(nn.Conv2DTranspose(ngf * 2, 4, 2, 1, use_bias=False))
    netG.add(nn.BatchNorm())
    netG.add(nn.Activation('relu'))
    # state size. (ngf*8) x 16 x 16
    netG.add(nn.Conv2DTranspose(ngf, 4, 2, 1, use_bias=False))
    netG.add(nn.BatchNorm())
    netG.add(nn.Activation('relu'))
    # state size. (ngf*8) x 32 x 32
    netG.add(nn.Conv2DTranspose(nc, 4, 2, 1, use_bias=False))
    netG.add(nn.Activation('tanh'))
    # state size. (nc) x 64 x 64

# build the discriminator
ndf = 64
netD = nn.Sequential()
with netD.name_scope():
    # input is (nc) x 64 x 64
    netD.add(nn.Conv2D(ndf, 4, 2, 1, use_bias=False))
    netD.add(nn.LeakyReLU(0.2))
    # state size. (ndf) x 32 x 32
```

```

netD.add(nn.Conv2D(ndf * 2, 4, 2, 1, use_bias=False))
netD.add(nn.BatchNorm())
netD.add(nn.LeakyReLU(0.2))
# state size. (ndf) x 16 x 16
netD.add(nn.Conv2D(ndf * 4, 4, 2, 1, use_bias=False))
netD.add(nn.BatchNorm())
netD.add(nn.LeakyReLU(0.2))
# state size. (ndf) x 8 x 8
netD.add(nn.Conv2D(ndf * 8, 4, 2, 1, use_bias=False))
netD.add(nn.BatchNorm())
netD.add(nn.LeakyReLU(0.2))
# state size. (ndf) x 4 x 4
netD.add(nn.Conv2D(1, 4, 1, 0, use_bias=False))

```

## Setup Loss Function and Optimizer

We use binary cross-entropy as our loss function and use the Adam optimizer. We initialize the network's parameters by sampling from a normal distribution.

```

In [7]: # Loss
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()

# initialize the generator and the discriminator
netG.initialize(mx.init.Normal(0.02), ctx=ctx)
netD.initialize(mx.init.Normal(0.02), ctx=ctx)

# trainer for the generator and the discriminator
trainerG = gluon.Trainer(netG.collect_params(), 'adam', {'learning_rate': lr, 'beta1':
beta1})
trainerD = gluon.Trainer(netD.collect_params(), 'adam', {'learning_rate': lr, 'beta1':
beta1})

```

## Training Loop

We recommend that you use a GPU for training this model. After a few epochs, we can see human-face-like images are generated.

```

In [8]: from datetime import datetime
import time
import logging

real_label = nd.ones((batch_size,), ctx=ctx)
fake_label = nd.zeros((batch_size,), ctx=ctx)

def facc(label, pred):
    pred = pred.ravel()
    label = label.ravel()
    return ((pred > 0.5) == label).mean()
metric = mx.metric.CustomMetric(facc)

stamp = datetime.now().strftime('%Y_%m_%d-%H_%M')
logging.basicConfig(level=logging.DEBUG)

for epoch in range(epochs):
    tic = time.time()
    btic = time.time()
    train_data.reset()
    iter = 0
    for batch in train_data:
        #####

```



```

# (1) Update D network: maximize Log(D(x)) + Log(1 - D(G(z)))
#####
data = batch.data[0].as_in_context(ctx)
latent_z = mx.nd.random_normal(0, 1, shape=(batch_size, latent_z_size, 1, 1),
ctx=ctx)

with autograd.record():
    # train with real image
    output = netD(data).reshape((-1, 1))
    errD_real = loss(output, real_label)
    metric.update([real_label,], [output,])

    # train with fake image
    fake = netG(latent_z)
    output = netD(fake).reshape((-1, 1))
    errD_fake = loss(output, fake_label)
    errD = errD_real + errD_fake
    errD.backward()
    metric.update([fake_label,], [output,])

trainerD.step(batch.data[0].shape[0])

#####
# (2) Update G network: maximize Log(D(G(z)))
#####
with autograd.record():
    fake = netG(latent_z)
    output = netD(fake).reshape((-1, 1))
    errG = loss(output, real_label)
    errG.backward()

trainerG.step(batch.data[0].shape[0])

# Print Log information every ten batches
if iter % 10 == 0:
    name, acc = metric.get()
    logging.info('speed: {} samples/s'.format(batch_size / (time.time() - btic)))
    logging.info('discriminator loss = %f, generator loss = %f, binary training
acc = %f at iter %d epoch %d'
                %(nd.mean(errD).asscalar(),
                  nd.mean(errG).asscalar(), acc, iter, epoch))
    iter = iter + 1
    btic = time.time()

name, acc = metric.get()
metric.reset()
# logging.info('\nbinary training acc at epoch %d: %s=%f' % (epoch, name, acc))
# logging.info('time: %f' % (time.time() - tic))

# Visualize one generated image for each epoch
# fake_img = fake[0]
# visualize(fake_img)
# plt.show()

```

```

INFO:root:speed: 7.755799027099384 samples/s
INFO:root:discriminator loss = 1.267250, generator loss = 3.865826, binary training acc =
0.593750 at iter 0 epoch 0
INFO:root:speed: 588.3748969822371 samples/s
INFO:root:discriminator loss = 0.158070, generator loss = 7.865579, binary training acc =
0.885653 at iter 10 epoch 0
INFO:root:speed: 585.7634125158751 samples/s
INFO:root:discriminator loss = 0.062522, generator loss = 9.711842, binary training acc =
0.920387 at iter 20 epoch 0
INFO:root:speed: 580.0184872678558 samples/s
INFO:root:discriminator loss = 0.113141, generator loss = 7.781098, binary training acc =
0.930444 at iter 30 epoch 0
INFO:root:speed: 583.7342663729535 samples/s
INFO:root:discriminator loss = 0.059121, generator loss = 10.791117, binary training acc =
0.920922 at iter 40 epoch 0
INFO:root:speed: 579.9194960292427 samples/s
INFO:root:discriminator loss = 0.965627, generator loss = 22.836861, binary training acc =
0.923866 at iter 50 epoch 0

```

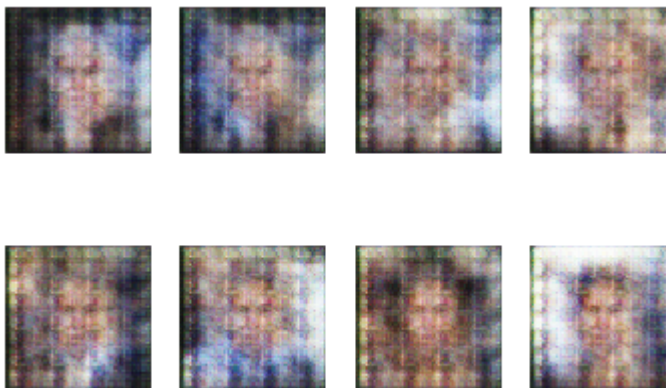
INFO:root:speed: 581.6119964379888 samples/s  
INFO:root:discriminator loss = 3.174960, generator loss = 27.712910, binary training acc = 0.914703 at iter 60 epoch 0  
INFO:root:speed: 577.151467198734 samples/s  
INFO:root:discriminator loss = 5.759238, generator loss = 26.630047, binary training acc = 0.901629 at iter 70 epoch 0  
INFO:root:speed: 577.1105201263284 samples/s  
INFO:root:discriminator loss = 0.161419, generator loss = 8.691389, binary training acc = 0.894097 at iter 80 epoch 0  
INFO:root:speed: 586.9802412336328 samples/s  
INFO:root:discriminator loss = 1.483496, generator loss = 15.815531, binary training acc = 0.888650 at iter 90 epoch 0  
INFO:root:speed: 573.1734239812782 samples/s  
INFO:root:discriminator loss = 3.764262, generator loss = 13.882145, binary training acc = 0.873376 at iter 100 epoch 0  
INFO:root:speed: 578.515962043676 samples/s  
INFO:root:discriminator loss = 1.166709, generator loss = 6.694098, binary training acc = 0.862190 at iter 110 epoch 0  
INFO:root:speed: 582.7154687917198 samples/s  
INFO:root:discriminator loss = 1.872752, generator loss = 4.427429, binary training acc = 0.852144 at iter 120 epoch 0  
INFO:root:speed: 567.5912191450042 samples/s  
INFO:root:discriminator loss = 0.367282, generator loss = 3.143092, binary training acc = 0.850549 at iter 130 epoch 0  
INFO:root:speed: 580.2128088187615 samples/s  
INFO:root:discriminator loss = 1.140358, generator loss = 8.651748, binary training acc = 0.848349 at iter 140 epoch 0  
INFO:root:speed: 577.9815862680757 samples/s  
INFO:root:discriminator loss = 0.495512, generator loss = 5.053850, binary training acc = 0.847630 at iter 150 epoch 0  
INFO:root:speed: 574.0019501513933 samples/s  
INFO:root:discriminator loss = 0.444338, generator loss = 3.439436, binary training acc = 0.844818 at iter 160 epoch 0  
INFO:root:speed: 571.9412026650133 samples/s  
INFO:root:discriminator loss = 0.399045, generator loss = 5.902631, binary training acc = 0.847542 at iter 170 epoch 0  
INFO:root:speed: 573.7676120612081 samples/s  
INFO:root:discriminator loss = 0.467865, generator loss = 4.489837, binary training acc = 0.845520 at iter 180 epoch 0  
INFO:root:speed: 562.1119887424929 samples/s  
INFO:root:discriminator loss = 0.630585, generator loss = 5.973484, binary training acc = 0.846818 at iter 190 epoch 0  
INFO:root:speed: 576.4809157190564 samples/s  
INFO:root:discriminator loss = 0.627957, generator loss = 5.089905, binary training acc = 0.843206 at iter 200 epoch 0  
INFO:root:speed: 580.216571165489 samples/s  
INFO:root:discriminator loss = 0.542177, generator loss = 4.040678, binary training acc = 0.906250 at iter 0 epoch 1  
INFO:root:speed: 580.091185896397 samples/s  
INFO:root:discriminator loss = 0.815529, generator loss = 7.728797, binary training acc = 0.909091 at iter 10 epoch 1  
INFO:root:speed: 576.0256816869916 samples/s  
INFO:root:discriminator loss = 0.633265, generator loss = 3.232196, binary training acc = 0.861979 at iter 20 epoch 1  
INFO:root:speed: 569.4611750479969 samples/s  
INFO:root:discriminator loss = 0.575909, generator loss = 4.796301, binary training acc = 0.849042 at iter 30 epoch 1  
INFO:root:speed: 554.097813006368 samples/s  
INFO:root:discriminator loss = 0.447131, generator loss = 5.489185, binary training acc = 0.856898 at iter 40 epoch 1  
INFO:root:speed: 570.8538411645242 samples/s  
INFO:root:discriminator loss = 1.440910, generator loss = 8.547214, binary training acc = 0.852788 at iter 50 epoch 1  
INFO:root:speed: 570.9121179445625 samples/s  
INFO:root:discriminator loss = 1.095329, generator loss = 4.820041, binary training acc = 0.850282 at iter 60 epoch 1  
INFO:root:speed: 578.9326874758721 samples/s  
INFO:root:discriminator loss = 0.777688, generator loss = 6.919479, binary training acc = 0.849142 at iter 70 epoch 1  
INFO:root:speed: 583.4411875937317 samples/s  
INFO:root:discriminator loss = 0.679454, generator loss = 5.001040, binary training acc = 0.858893 at iter 80 epoch 1

```
INFO:root:speed: 584.0937211690776 samples/s
INFO:root:discriminator loss = 0.295851, generator loss = 5.098488, binary training acc = 0.858431 at iter 90 epoch 1
INFO:root:speed: 565.923142440316 samples/s
INFO:root:discriminator loss = 0.551316, generator loss = 4.998213, binary training acc = 0.864016 at iter 100 epoch 1
INFO:root:speed: 580.1852197669191 samples/s
INFO:root:discriminator loss = 0.421467, generator loss = 5.157113, binary training acc = 0.867047 at iter 110 epoch 1
INFO:root:speed: 580.7186547451903 samples/s
INFO:root:discriminator loss = 0.769515, generator loss = 10.623252, binary training acc = 0.872482 at iter 120 epoch 1
INFO:root:speed: 578.0351038883015 samples/s
INFO:root:discriminator loss = 1.008357, generator loss = 2.453021, binary training acc = 0.862059 at iter 130 epoch 1
INFO:root:speed: 574.0375470193125 samples/s
INFO:root:discriminator loss = 0.833088, generator loss = 0.940359, binary training acc = 0.862810 at iter 140 epoch 1
INFO:root:speed: 584.0771994107779 samples/s
INFO:root:discriminator loss = 0.662278, generator loss = 4.458453, binary training acc = 0.855960 at iter 150 epoch 1
INFO:root:speed: 569.7331182613125 samples/s
INFO:root:discriminator loss = 0.584824, generator loss = 2.921254, binary training acc = 0.857046 at iter 160 epoch 1
INFO:root:speed: 578.4935208232315 samples/s
INFO:root:discriminator loss = 1.130421, generator loss = 8.173367, binary training acc = 0.859147 at iter 170 epoch 1
INFO:root:speed: 570.2680925734728 samples/s
INFO:root:discriminator loss = 0.167190, generator loss = 5.951529, binary training acc = 0.861576 at iter 180 epoch 1
INFO:root:speed: 562.6634023643834 samples/s
INFO:root:discriminator loss = 0.130641, generator loss = 5.944596, binary training acc = 0.864938 at iter 190 epoch 1
INFO:root:speed: 576.3435275122596 samples/s
INFO:root:discriminator loss = 0.271539, generator loss = 5.291772, binary training acc = 0.865050 at iter 200 epoch 1
```

## Results

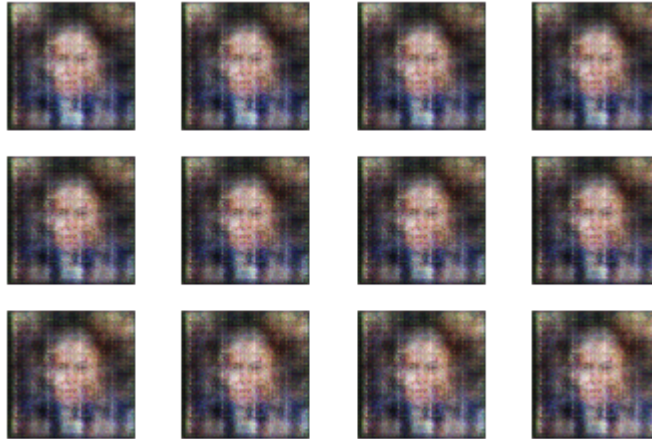
Given a trained generator, we can generate some images of faces.

```
In [9]: num_image = 8
        for i in range(num_image):
            latent_z = mx.nd.random_normal(0, 1, shape=(1, latent_z_size, 1, 1), ctx=ctx)
            img = netG(latent_z)
            plt.subplot(2,4,i+1)
            visualize(img[0])
        plt.show()
```



We can also interpolate along the manifold between images by interpolating linearly between points in the latent space and visualizing the corresponding images. We can see that small changes in the latent space results in smooth changes in generated images.

```
In [10]: num_image = 12
latent_z = mx.nd.random_normal(0, 1, shape=(1, latent_z_size, 1, 1), ctx=ctx)
step = 0.05
for i in range(num_image):
    img = netG(latent_z)
    plt.subplot(3,4,i+1)
    visualize(img[0])
    latent_z += 0.05
plt.show()
```



For whinges or inquiries, [open an issue on GitHub](#).

# Pixel to Pixel Generative Adversarial Networks

[Pixel to Pixel Generative Adversarial Networks](#) applies [Conditional Generative Adversarial Networks](#) as a general-purpose solution to image-to-image translation problems. These networks not only learn the mapping from input image to output image, but also learn a loss function to train this mapping.

With pixel2pixel GAN, it is possible to train different type of image translation tasks with small datasets. In this tutorial, we will train on three image translation tasks: facades with 400 images from [CMP Facades dataset](#), cityscapes with 2975 images from [Cityscapes training set](#) and maps with 1096 training images scraped from Google Maps.

For harder problems such as [edges2shoes](#) and [edges2handbags](#), it may be important to train on far larger datasets, which takes significantly more time. You can try them with [Multiple GPUs](#).

```
In [1]: from __future__ import print_function
import os
import matplotlib as mpl
import tarfile
import matplotlib.image as mpimg
from matplotlib import pyplot as plt

import mxnet as mx
from mxnet import gluon
from mxnet import ndarray as nd
from mxnet.gluon import nn, utils
from mxnet.gluon.nn import Dense, Activation, Conv2D, Conv2DTranspose, \
    BatchNorm, LeakyReLU, Flatten, HybridSequential, HybridBlock, Dropout
from mxnet import autograd
import numpy as np
```

## Set Training parameters

```
In [2]: epochs = 100
batch_size = 10

use_gpu = True
ctx = mx.gpu() if use_gpu else mx.cpu()

lr = 0.0002
beta1 = 0.5
lambda1 = 100

pool_size = 50
```

## Download and Preprocess Dataset

We first train on facades dataset. We need to crop images to input images and output images. Notice that pixel2pixel GAN is capable to train these tasks bidirectional. You can set `is-reversed=True` to switch input and output image patterns.

```
In [3]: dataset = 'facades'
```

We first resize images to size 512 \* 256. Then normalize image pixel values to be between -1 and 1.

```
In [4]: img_wd = 256
img_ht = 256
train_img_path = '%s/train' % (dataset)
val_img_path = '%s/val' % (dataset)

def download_data(dataset):
    if not os.path.exists(dataset):
        url =
        'https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets/%s.tar.gz' %
        (dataset)
        os.mkdir(dataset)
        data_file = utils.download(url)
        with tarfile.open(data_file) as tar:
            tar.extractall(path='.')
        os.remove(data_file)

def load_data(path, batch_size, is_reversed=False):
    img_in_list = []
    img_out_list = []
    for path, _, fnames in os.walk(path):
        for fname in fnames:
            if not fname.endswith('.jpg'):
                continue
            img = os.path.join(path, fname)
            img_arr = mx.image.imread(img).astype(np.float32)/127.5 - 1
            img_arr = mx.image.imresize(img_arr, img_wd * 2, img_ht)
            # Crop input and output images
            img_arr_in, img_arr_out = [mx.image.fixed_crop(img_arr, 0, 0, img_wd, img_ht),
                                      mx.image.fixed_crop(img_arr, img_wd, 0, img_wd,
img_ht)]
            img_arr_in, img_arr_out = [nd.transpose(img_arr_in, (2,0,1)),
                                      nd.transpose(img_arr_out, (2,0,1))]
            img_arr_in, img_arr_out = [img_arr_in.reshape((1,) + img_arr_in.shape),
                                      img_arr_out.reshape((1,) + img_arr_out.shape)]
            img_in_list.append(img_arr_out if is_reversed else img_arr_in)
            img_out_list.append(img_arr_in if is_reversed else img_arr_out)

    return mx.io.NDArrayIter(data=[nd.concat(*img_in_list, dim=0),
nd.concat(*img_out_list, dim=0)],
                            batch_size=batch_size)

download_data(dataset)
train_data = load_data(train_img_path, batch_size, is_reversed=True)
val_data = load_data(val_img_path, batch_size, is_reversed=True)
```

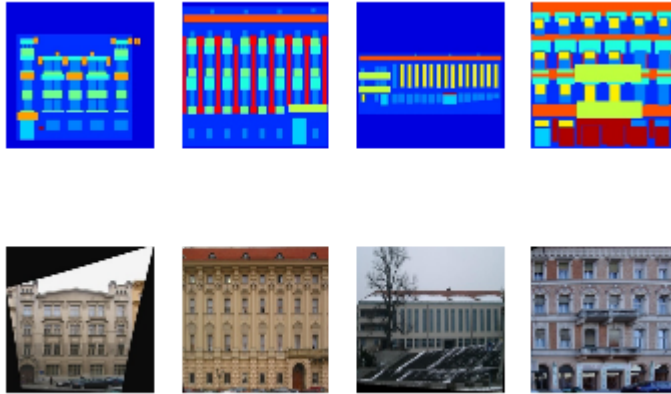
Visualize 4 images:

```
In [5]: def visualize(img_arr):
plt.imshow(((img_arr.asnumpy().transpose(1, 2, 0) + 1.0) * 127.5).astype(np.uint8))
plt.axis('off')

def preview_train_data():
```

```
img_in_list, img_out_list = train_data.next().data
for i in range(4):
    plt.subplot(2,4,i+1)
    visualize(img_in_list[i])
    plt.subplot(2,4,i+5)
    visualize(img_out_list[i])
plt.show()

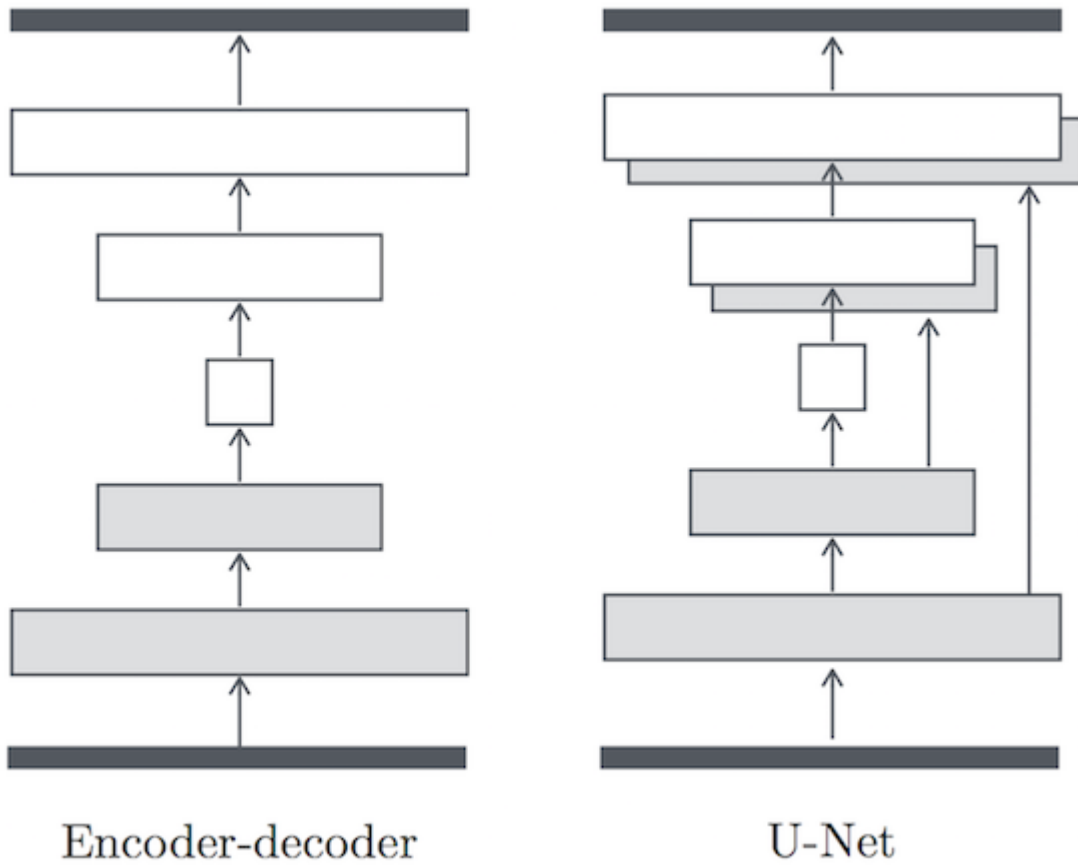
preview_train_data()
```



## Defining the networks

Both generator and discriminator use modules of the form convolution-BatchNorm-ReLu.

The key for generator is U-net architecture adding skip connections which shuttle low-level information shared between input and output images across net.



PatchGAN – that only penalizes structure at the scale of patches is applied as discriminator architecture. This discriminator tries to classify if each  $N \times N$  patch in an image is real or fake. We run this discriminator convolutionally across the image, averaging all responses to provide the ultimate output of netD.

```
In [6]: # Define Unet generator skip block
class UnetSkipUnit(HybridBlock):
    def __init__(self, inner_channels, outer_channels, inner_block=None, innermost=False,
                 outermost=False,
                 use_dropout=False, use_bias=False):
        super(UnetSkipUnit, self).__init__()

        with self.name_scope():
            self.outermost = outermost
            en_conv = Conv2D(channels=inner_channels, kernel_size=4, strides=2, padding=1,
                             in_channels=outer_channels, use_bias=use_bias)
            en_relu = LeakyReLU(alpha=0.2)
            en_norm = BatchNorm(momentum=0.1, in_channels=inner_channels)
            de_relu = Activation(activation='relu')
            de_norm = BatchNorm(momentum=0.1, in_channels=outer_channels)

            if innermost:
                de_conv = Conv2DTranspose(channels=outer_channels, kernel_size=4,
                                           strides=2, padding=1,
                                           in_channels=inner_channels, use_bias=use_bias)
            encoder = [en_relu, en_conv]
            decoder = [de_relu, de_norm, de_conv]
            model = encoder + decoder
```



```

        elif outermost:
            de_conv = Conv2DTranspose(channels=outer_channels, kernel_size=4,
strides=2, padding=1,
                                     in_channels=inner_channels * 2)
            encoder = [en_conv]
            decoder = [de_relu, de_conv, Activation(activation='tanh')]
            model = encoder + [inner_block] + decoder
        else:
            de_conv = Conv2DTranspose(channels=outer_channels, kernel_size=4,
strides=2, padding=1,
                                     in_channels=inner_channels * 2,
use_bias=use_bias)
            encoder = [en_relu, en_conv, en_norm]
            decoder = [de_relu, de_conv, de_norm]
            model = encoder + [inner_block] + decoder
        if use_dropout:
            model += [Dropout(rate=0.5)]

        self.model = HybridSequential()
        with self.model.name_scope():
            for block in model:
                self.model.add(block)

def hybrid_forward(self, F, x):
    if self.outermost:
        return self.model(x)
    else:
        return F.concat(self.model(x), x, dim=1)

# Define Unet generator
class UnetGenerator(HybridBlock):
    def __init__(self, in_channels, num_downs, ngf=64, use_dropout=True):
        super(UnetGenerator, self).__init__()

        #Build unet generator structure
        unet = UnetSkipUnit(ngf * 8, ngf * 8, innermost=True)
        for _ in range(num_downs - 5):
            unet = UnetSkipUnit(ngf * 8, ngf * 8, unet, use_dropout=use_dropout)
        unet = UnetSkipUnit(ngf * 8, ngf * 4, unet)
        unet = UnetSkipUnit(ngf * 4, ngf * 2, unet)
        unet = UnetSkipUnit(ngf * 2, ngf * 1, unet)
        unet = UnetSkipUnit(ngf, in_channels, unet, outermost=True)

        with self.name_scope():
            self.model = unet

    def hybrid_forward(self, F, x):
        return self.model(x)

# Define the PatchGAN discriminator
class Discriminator(HybridBlock):
    def __init__(self, in_channels, ndf=64, n_layers=3, use_sigmoid=False,
use_bias=False):
        super(Discriminator, self).__init__()

        with self.name_scope():
            self.model = HybridSequential()
            kernel_size = 4
            padding = int(np.ceil((kernel_size - 1)/2))
            self.model.add(Conv2D(channels=ndf, kernel_size=kernel_size, strides=2,
padding=padding, in_channels=in_channels))
            self.model.add(LeakyReLU(alpha=0.2))

            nf_mult = 1
            for n in range(1, n_layers):
                nf_mult_prev = nf_mult
                nf_mult = min(2 ** n, 8)
                self.model.add(Conv2D(channels=ndf * nf_mult, kernel_size=kernel_size,
strides=2,
padding=padding, in_channels=ndf * nf_mult_prev,
use_bias=use_bias))
                self.model.add(BatchNorm(momentum=0.1, in_channels=ndf * nf_mult))

```

```

        self.model.add(LeakyReLU(alpha=0.2))

        nf_mult_prev = nf_mult
        nf_mult = min(2 ** n_layers, 8)
        self.model.add(Conv2D(channels=ndf * nf_mult, kernel_size=kernel_size,
strides=1,
                                padding=padding, in_channels=ndf * nf_mult_prev,
                                use_bias=use_bias))
        self.model.add(BatchNorm(momentum=0.1, in_channels=ndf * nf_mult))
        self.model.add(LeakyReLU(alpha=0.2))
        self.model.add(Conv2D(channels=1, kernel_size=kernel_size, strides=1,
                                padding=padding, in_channels=ndf * nf_mult))

        if use_sigmoid:
            self.model.add(Activation(activation='sigmoid'))

def hybrid_forward(self, F, x):
    out = self.model(x)
    #print(out)
    return out

```

## Construct networks, Initialize parameters, Setup Loss Function and Optimizer

We use binary cross entropy and L1 loss as loss functions. L1 loss can be used to capture low frequencies in images.

```

In [7]: def param_init(param):
        if param.name.find('conv') != -1:
            if param.name.find('weight') != -1:
                param.initialize(init=mx.init.Normal(0.02), ctx=ctx)
            else:
                param.initialize(init=mx.init.Zero(), ctx=ctx)
        elif param.name.find('batchnorm') != -1:
            param.initialize(init=mx.init.Zero(), ctx=ctx)
            # Initialize gamma from normal distribution with mean 1 and std 0.02
            if param.name.find('gamma') != -1:
                param.set_data(nd.random_normal(1, 0.02, param.data().shape))

def network_init(net):
    for param in net.collect_params().values():
        param_init(param)

def set_network():
    # Pixel2pixel networks
    netG = UnetGenerator(in_channels=3, num_downs=8)
    netD = Discriminator(in_channels=6)

    # Initialize parameters
    network_init(netG)
    network_init(netD)

    # trainer for the generator and the discriminator
    trainerG = gluon.Trainer(netG.collect_params(), 'adam', {'learning_rate': lr, 'beta1':
beta1})
    trainerD = gluon.Trainer(netD.collect_params(), 'adam', {'learning_rate': lr, 'beta1':
beta1})

    return netG, netD, trainerG, trainerD

# Loss
GAN_loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
L1_loss = gluon.loss.L1Loss()

netG, netD, trainerG, trainerD = set_network()

```

# Image pool for discriminator

We use history image pool to help discriminator memorize history errors instead of just comparing current real input and fake output.

```
In [8]: class ImagePool():
    def __init__(self, pool_size):
        self.pool_size = pool_size
        if self.pool_size > 0:
            self.num_imgs = 0
            self.images = []

    def query(self, images):
        if self.pool_size == 0:
            return images
        ret_imgs = []
        for i in range(images.shape[0]):
            image = nd.expand_dims(images[i], axis=0)
            if self.num_imgs < self.pool_size:
                self.num_imgs = self.num_imgs + 1
                self.images.append(image)
                ret_imgs.append(image)
            else:
                p = nd.random_uniform(0, 1, shape=(1,)).asscalar()
                if p > 0.5:
                    random_id = nd.random_uniform(0, self.pool_size - 1, shape=(1,)).astype(np.uint8).asscalar()
                    tmp = self.images[random_id].copy()
                    self.images[random_id] = image
                    ret_imgs.append(tmp)
                else:
                    ret_imgs.append(image)
        ret_imgs = nd.concat(*ret_imgs, dim=0)
        return ret_imgs
```

## Training Loop

We recommend to use gpu to boost training. After a few epochs, we can see images similar to building structure are generated.

```
In [9]: from datetime import datetime
import time
import logging

def facc(label, pred):
    pred = pred.ravel()
    label = label.ravel()
    return ((pred > 0.5) == label).mean()

def train():
    image_pool = ImagePool(pool_size)
    metric = mx.metric.CustomMetric(facc)

    stamp = datetime.now().strftime('%Y_%m_%d-%H_%M')
    logging.basicConfig(level=logging.DEBUG)

    for epoch in range(epochs):
        tic = time.time()
        btic = time.time()
        train_data.reset()
        iter = 0
        for batch in train_data:
```

```

#####
# (1) Update D network: maximize  $\log(D(x, y)) + \log(1 - D(x, G(x, z)))$ 
#####
real_in = batch.data[0].as_in_context(ctx)
real_out = batch.data[1].as_in_context(ctx)

fake_out = netG(real_in)
fake_concat = image_pool.query(nd.concat(real_in, fake_out, dim=1))
with autograd.record():
    # Train with fake image
    # Use image pooling to utilize history images
    output = netD(fake_concat)
    fake_label = nd.zeros(output.shape, ctx=ctx)
    errD_fake = GAN_loss(output, fake_label)
    metric.update([fake_label,], [output,])

    # Train with real image
    real_concat = nd.concat(real_in, real_out, dim=1)
    output = netD(real_concat)
    real_label = nd.ones(output.shape, ctx=ctx)
    errD_real = GAN_loss(output, real_label)
    errD = (errD_real + errD_fake) * 0.5
    errD.backward()
    metric.update([real_label,], [output,])

trainerD.step(batch.data[0].shape[0])

#####
# (2) Update G network: maximize  $\log(D(x, G(x, z))) - \lambda * L1(y, G(x, z))$ 
#####
with autograd.record():
    fake_out = netG(real_in)
    fake_concat = nd.concat(real_in, fake_out, dim=1)
    output = netD(fake_concat)
    real_label = nd.ones(output.shape, ctx=ctx)
    errG = GAN_loss(output, real_label) + L1_loss(real_out, fake_out) *
lambda1
    errG.backward()

trainerG.step(batch.data[0].shape[0])

# Print Log information every ten batches
if iter % 10 == 0:
    name, acc = metric.get()
    logging.info('speed: {} samples/s'.format(batch_size / (time.time() -
btic)))
    logging.info('discriminator loss = %f, generator loss = %f, binary
training acc = %f at iter %d epoch %d'
                %(nd.mean(errD).asscalar(),
                  nd.mean(errG).asscalar(), acc, iter, epoch))
    iter = iter + 1
    btic = time.time()

name, acc = metric.get()
metric.reset()
logging.info('\nbinary training acc at epoch %d: %s=%f' % (epoch, name, acc))
logging.info('time: %f' % (time.time() - tic))

# Visualize one generated image for each epoch
fake_img = fake_out[0]
visualize(fake_img)
plt.show()

train()

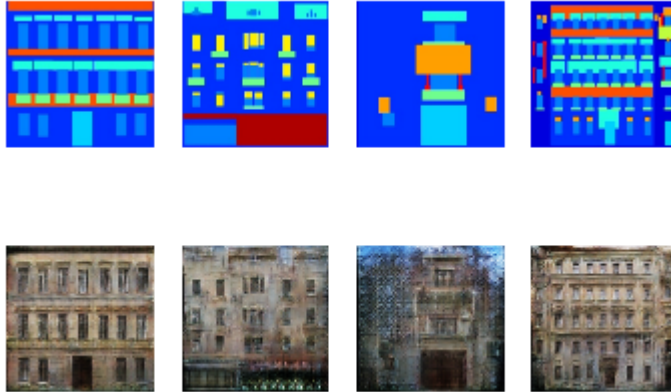
```

## Results

Generate images with generator.

```
In [10]: def print_result():
num_image = 4
img_in_list, img_out_list = val_data.next().data
for i in range(num_image):
    img_in = nd.expand_dims(img_in_list[i], axis=0)
    plt.subplot(2,4,i+1)
    visualize(img_in[0])
    img_out = netG(img_in.as_in_context(ctx))
    plt.subplot(2,4,i+5)
    visualize(img_out[0])
plt.show()

print_result()
```



## Other dataset experiments

Run experiments on cityscapes and maps datasets

```
In [11]: datasets = ['cityscapes', 'maps']
is_reversed = False
batch_size = 64

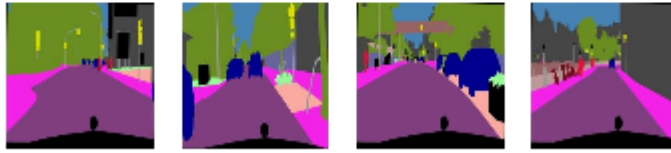
for dataset in datasets:
    train_img_path = '%s/train' % (dataset)
    val_img_path = '%s/val' % (dataset)
    download_data(dataset)
    train_data = load_data(train_img_path, batch_size, is_reversed=is_reversed)
    val_data = load_data(val_img_path, batch_size, is_reversed=is_reversed)

    print("Preview %s training data:" % (dataset))
    preview_train_data()

    netG, netD, trainerG, trainerD = set_network()
    train()

    print("Training result for %s" % (dataset))
    print_result()
```

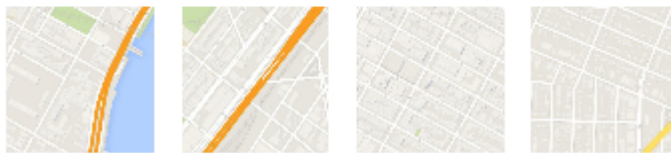
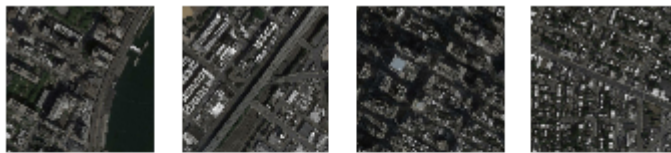
Preview cityscapes training data:



Training result for cityscapes



Preview maps training data:



Training result for maps



## Citation

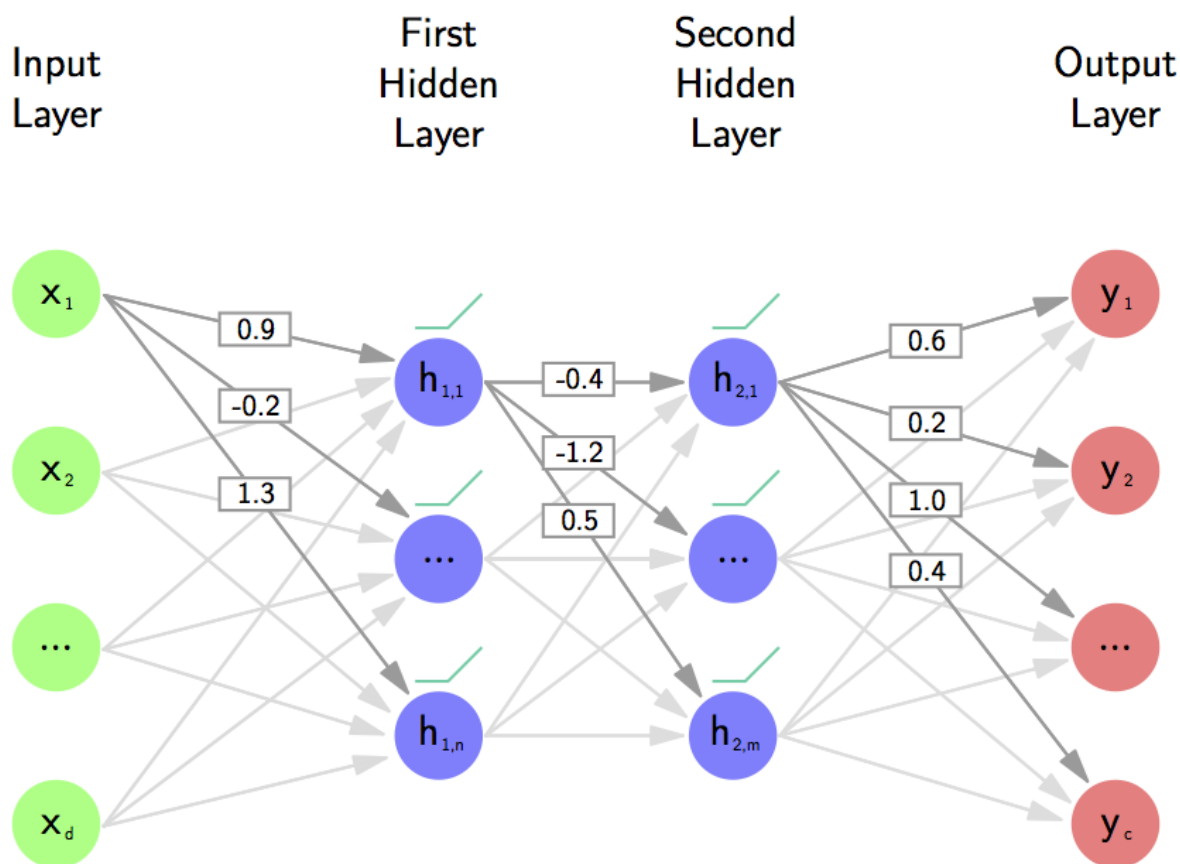
CMP Facades dataset: @INPROCEEDINGS{ Tylecek13, author = {Radim Tyle{:raw-latex:``\v c`}ek, Radim {:raw-latex:``\v S`}a}, title = {Spatial Pattern Templates for Recognition of Objects with Regular Structure}, booktitle = {Proc. GCPR}, year = {2013}, address = {Saarbrücken, Germany}, }

Cityscapes training set: @inproceedings{Cordts2016Cityscapes, title={The Cityscapes Dataset for Semantic Urban Scene Understanding}, author={Cordts, Marius and Omran, Mohamed and Ramos, Sebastian and Rehfeld, Timo and Enzweiler, Markus and Benenson, Rodrigo and Franke, Uwe and Roth, Stefan and Schiele, Bernt}, booktitle={Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)}, year={2016} }

## Bayes by Backprop from scratch (NN, classification)

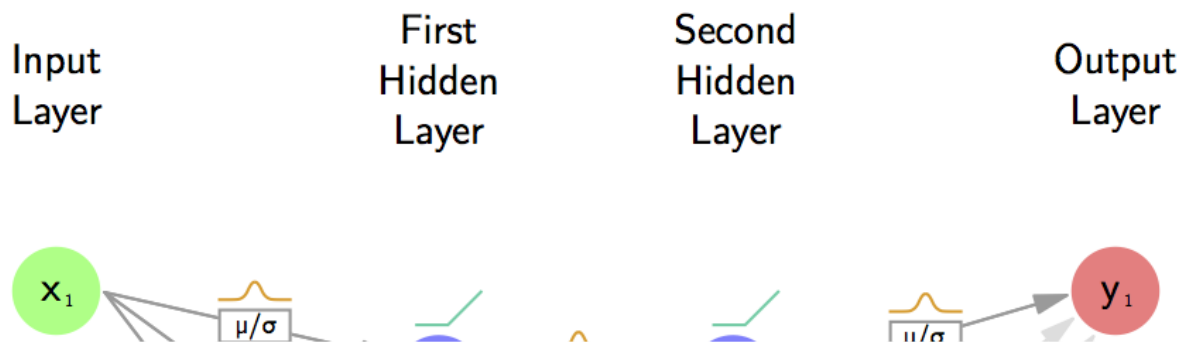
We have already learned how to implement deep neural networks and how to use them for classification and regression tasks. In order to fight overfitting, we further introduced a concept called *dropout*, which randomly turns off a certain percentage of the weights during training.

Recall the classic architecture of a MLP (shown below, without bias terms). So far, when training a neural network, our goal was to find an optimal point estimate for the weights.



While networks trained using this approach usually perform well in regions with lots of data, they fail to express uncertainty in regions with little or no data, leading to overconfident decisions. This drawback motivates the application of Bayesian learning to neural networks, introducing probability distributions over the weights. These distributions can be of various nature in theory. To make our lives easier and to have an intuitive understanding of the distribution at each weight, we will use a Gaussian distribution.





Unfortunately though, exact Bayesian inference on the parameters of a neural network is intractable. One promising way of addressing this problem is presented by the “Bayes by Backprop” algorithm (introduced by the “[Weight Uncertainty in Neural Networks](#)” paper) which derives a variational approximation to the true posterior. This algorithm does not only make networks more “honest” with respect to their overall uncertainty, but also automatically leads to regularization, thereby eliminating the need of using dropout in this model.

While we will try to explain the most important concepts of this algorithm in this notebook, we also encourage the reader to consult the paper for deeper insights.

Let’s start implementing this idea and evaluate its performance on the MNIST classification problem. We start off with the usual set of imports.

```
In [1]: from __future__ import print_function
import collections
import mxnet as mx
import numpy as np
from mxnet import nd, autograd
from matplotlib import pyplot as plt
```

For easy tuning and experimentation, we define a dictionary holding the hyper-parameters of our model.

```
In [2]: config = {  
    "num_hidden_layers": 2,  
    "num_hidden_units": 400,  
    "batch_size": 128,  
    "epochs": 10,  
    "learning_rate": 0.001,  
    "num_samples": 1,  
    "pi": 0.25,  
    "sigma_p": 1.0,  
    "sigma_p1": 0.75,  
    "sigma_p2": 0.1,  
}
```

Also, we specify the device context for MXNet.

```
In [3]: ctx = mx.cpu()
```

## Load MNIST data

We will again train and evaluate the algorithm on the MNIST data set and therefore load the data set as follows:

```
In [4]: def transform(data, label):  
    return data.astype(np.float32)/126.0, label.astype(np.float32)  
  
mnist = mx.test_utils.get_mnist()  
num_inputs = 784  
num_outputs = 10  
batch_size = config['batch_size']  
  
train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,  
    transform=transform),  
    batch_size, shuffle=True)  
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,  
    transform=transform),  
    batch_size, shuffle=False)  
  
num_train = sum([batch_size for i in train_data])  
num_batches = num_train / batch_size
```

In order to reproduce and compare the results from the paper, we preprocess the pixels by dividing by 126.

## Model definition

### Activation function

As with lots of past examples, we will again use the ReLU as our activation function for the hidden units of our neural network.

```
In [5]: def relu(X):  
    return nd.maximum(X, nd.zeros_like(X))
```

# Neural net modeling

As our model we are using a straightforward MLP and we are wiring up our network just as we are used to.

```
In [6]: num_layers = config['num_hidden_layers']

# define function for evaluating MLP
def net(X, layer_params):
    layer_input = X
    for i in range(len(layer_params) // 2 - 2):
        h_linear = nd.dot(layer_input, layer_params[2*i]) + layer_params[2*i + 1]
        layer_input = relu(h_linear)
    # Last layer without ReLU
    output = nd.dot(layer_input, layer_params[-2]) + layer_params[-1]
    return output

# define network weight shapes
layer_param_shapes = []
num_hidden = config['num_hidden_units']
for i in range(num_layers + 1):
    if i == 0: # input layer
        W_shape = (num_inputs, num_hidden)
        b_shape = (num_hidden,)
    elif i == num_layers: # last layer
        W_shape = (num_hidden, num_outputs)
        b_shape = (num_outputs,)
    else: # hidden layers
        W_shape = (num_hidden, num_hidden)
        b_shape = (num_hidden,)
    layer_param_shapes.extend([W_shape, b_shape])
```

## Build objective/loss

As we briefly mentioned at the beginning of the notebook, we will use variational inference in order to make the prediction of the posterior tractable. While we can not model the posterior  $P(\mathbf{w} \mid \mathcal{D})$  directly, we try to find the parameters  $\theta$  of a distribution on the weights  $q(\mathbf{w} \mid \theta)$  (commonly referred to as the *variational posterior*) that minimizes the KL divergence with the true posterior. Formally this can be expressed as:

$$\begin{aligned}\theta^* &= \arg \min_{\theta} \text{KL}[q(\mathbf{w} \mid \theta) \parallel P(\mathbf{w} \mid \mathcal{D})] \\ &= \arg \min_{\theta} \int q(\mathbf{w} \mid \theta) \log \frac{q(\mathbf{w} \mid \theta)}{P(\mathbf{w})P(\mathcal{D} \mid \mathbf{w})} d\mathbf{w} \\ &= \arg \min_{\theta} \text{KL}[q(\mathbf{w} \mid \theta) \parallel P(\mathbf{w})] - \mathbb{E}_{q(\mathbf{w} \mid \theta)}[\log P(\mathcal{D} \mid \mathbf{w})]\end{aligned}$$

The resulting loss function, commonly referred to as either *variational free energy* or *expected lower bound (ELBO)*, has to be minimized and is then given as follows:

$$\mathcal{F}(\mathcal{D}, \theta) = \text{KL}[q(\mathbf{w} \mid \theta) \parallel P(\mathbf{w})] - \mathbb{E}_{q(\mathbf{w} \mid \theta)}[\log P(\mathcal{D} \mid \mathbf{w})]$$

As one can easily see, the cost function tries to balance the complexity of the data  $P(\mathcal{D} \mid \mathbf{w})$  and the simplicity of the prior  $P(\mathbf{w})$ .

We can approximate this exact cost through a Monte Carlo sampling procedure as follows

$$\mathcal{F}(\mathcal{D}, \theta) \approx \sum_{i=1}^n \log q(\mathbf{w}^{(i)} | \theta) - \log P(\mathbf{w}^{(i)}) - \log P(\mathcal{D} | \mathbf{w}^{(i)})$$

where  $\mathbf{w}^{(i)}$  corresponds to the  $i$ -th Monte Carlo sample from the variational posterior. While writing this notebook, we noticed that even taking just one sample leads to good results and we will therefore stick to just sampling once throughout the notebook.

Since we will be working with mini-batches, the exact loss form we will be using looks as follows:

$$\begin{aligned} \mathcal{F}(\mathcal{D}_i, \theta) &= \frac{1}{M} \text{KL}[\log q(\mathbf{w} | \theta) || \log P(\mathbf{w})] - \mathbb{E}_{q(\mathbf{w} | \theta)}[\log P(\mathcal{D}_i | \mathbf{w})] \\ &\approx \frac{1}{M} (\log q(\mathbf{w} | \theta) - \log P(\mathbf{w})) - \log P(\mathcal{D}_i | \mathbf{w}) \end{aligned}$$

where  $M$  corresponds to the number of batches.

Let's now look at each of these single terms individually.

## Likelihood

As with lots of past examples, we will again use the softmax to define our likelihood  $P(\mathcal{D}_i | \mathbf{w})$ . Revisit the [MLP from scratch notebook](#) for a detailed motivation of this function.

```
In [7]: def log_softmax_likelihood(yhat_linear, y):  
        return nd.nansum(y * nd.log_softmax(yhat_linear), axis=0, exclude=True)
```

## Prior

Since we are introducing a Bayesian treatment for the network, we need to define a Prior over the weights.

### Gaussian prior

A popular and simple prior is the Gaussian distribution. The prior over the entire weight vector simply corresponds to the product of the individual Gaussians:

$$P(\mathbf{w}) = \prod_i \mathcal{N}(\mathbf{w}_i | 0, \sigma_p^2)$$

We can define the Gaussian distribution and our Gaussian prior as seen below. Note that we are ultimately interested in the log-prior  $\log P(\mathbf{w})$  and therefore compute the sum of the log-Gaussians.

$$\log P(\mathbf{w}) = \sum_i \log \mathcal{N}(\mathbf{w}_i \mid 0, \sigma_p^2)$$

```
In [8]: LOG2PI = np.log(2.0 * np.pi)

def log_gaussian(x, mu, sigma):
    return -0.5 * LOG2PI - nd.log(sigma) - (x - mu) ** 2 / (2 * sigma ** 2)

def gaussian_prior(x):
    sigma_p = nd.array([config['sigma_p']], ctx=ctx)

    return nd.sum(log_gaussian(x, 0., sigma_p))
```

## Scale mixture prior

Instead of a single Gaussian, the paper also suggests the usage of a scale mixture prior for  $P(\mathbf{w})$  as an alternative:

$$P(\mathbf{w}) = \prod_i \left( \pi \mathcal{N}(\mathbf{w}_i \mid 0, \sigma_1^2) + (1 - \pi) \mathcal{N}(\mathbf{w}_i \mid 0, \sigma_2^2) \right)$$

where  $\pi \in [0, 1]$ ,  $\sigma_1 > \sigma_2$  and  $\sigma_2 \ll 1$ . Again we are interested in the log-prior  $\log P(\mathbf{w})$ , which can be expressed as follows:

$$\log P(\mathbf{w}) = \sum_i \log \left( \pi \mathcal{N}(\mathbf{w}_i \mid 0, \sigma_1^2) + (1 - \pi) \mathcal{N}(\mathbf{w}_i \mid 0, \sigma_2^2) \right)$$

```
In [9]: def gaussian(x, mu, sigma):
    scaling = 1.0 / nd.sqrt(2.0 * np.pi * (sigma ** 2))
    bell = nd.exp(-(x - mu) ** 2 / (2.0 * sigma ** 2))

    return scaling * bell

def scale_mixture_prior(x):
    sigma_p1 = nd.array([config['sigma_p1']], ctx=ctx)
    sigma_p2 = nd.array([config['sigma_p2']], ctx=ctx)
    pi = config['pi']

    first_gaussian = pi * gaussian(x, 0., sigma_p1)
    second_gaussian = (1 - pi) * gaussian(x, 0., sigma_p2)

    return nd.log(first_gaussian + second_gaussian)
```

## Variational Posterior

The last missing piece in the equation is the variational posterior. Again, we choose a Gaussian distribution for this purpose. The variational posterior on the weights is centered on the mean vector  $\mu$  and has variance  $\sigma^2$ :

$$q(\mathbf{w} \mid \theta) = \prod_i \mathcal{N}(\mathbf{w}_i \mid \mu, \sigma^2)$$

The log-posterior  $\log q(\mathbf{w} \mid \theta)$  is given by:

$$\log q(\mathbf{w} \mid \theta) = \sum_i \log \mathcal{N}(\mathbf{w}_i \mid \mu, \sigma^2)$$

## Combined Loss

After introducing the data likelihood, the prior, and the variational posterior, we are now able to build our combined loss function:  $\mathcal{F}(\mathcal{D}_i, \theta) = \frac{1}{M}(\log q(\mathbf{w} \mid \theta) - \log P(\mathbf{w})) - \log P(\mathcal{D}_i \mid \mathbf{w})$

```
In [10]: def combined_loss(output, label_one_hot, params, mus, sigmas, log_prior, log_likelihood):  
    # Calculate data likelihood  
    log_likelihood_sum = nd.sum(log_likelihood(output, label_one_hot))  
  
    # Calculate prior  
    log_prior_sum = sum([nd.sum(log_prior(param)) for param in params])  
  
    # Calculate variational posterior  
    log_var_posterior_sum = sum([nd.sum(log_gaussian(params[i], mus[i], sigmas[i])) for i  
in range(len(params))])  
  
    # Calculate total loss  
    return 1.0 / num_batches * (log_var_posterior_sum - log_prior_sum) -  
    log_likelihood_sum
```

## Optimizer

We use vanilla stochastic gradient descent to optimize the variational parameters. Note that this implements the updates described in the paper, as the gradient contribution due to the reparametrization trick is automatically included by taking the gradients of the combined loss function with respect to the variational parameters.

```
In [11]: def SGD(params, lr):  
    for param in params:  
        param[:] = param - lr * param.grad
```

## Evaluation metric

In order to being able to assess our model performance we define a helper function which evaluates our accuracy on an ongoing basis.

```
In [12]: def evaluate_accuracy(data_iterator, net, layer_params):
    numerator = 0.
    denominator = 0.
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1, 784))
        label = label.as_in_context(ctx)
        output = net(data, layer_params)
        predictions = nd.argmax(output, axis=1)
        numerator += nd.sum(predictions == label)
        denominator += data.shape[0]
    return (numerator / denominator).asscalar()
```

## Parameter initialization

We are using a Gaussian distribution for each individual weight as our variational posterior, which means that we need to store two parameters, mean and variance, for each weight. For the variance we need to ensure that it is non-negative, which we will do by using the softplus function to express  $\sigma$  in terms of an unconstrained parameter  $\rho$ . While gradient descent will be performed on  $\theta = (\mu, \rho)$ , the distribution for each individual weight is represented as  $w_i \sim \mathcal{N}(w_i \mid \mu_i, \sigma_i)$  with  $\sigma_i = \text{softplus}(\rho_i)$ .

We initialize  $\mu$  with a Gaussian around 0 (just as we would initialize standard weights of a neural network). It is important to initialize  $\rho$  (and hence  $\sigma$ ) to a small value, otherwise learning might not work properly.

```
In [13]: weight_scale = .1
    rho_offset = -3

    # initialize variational parameters; mean and variance for each weight
    mus = []
    rhos = []

    for shape in layer_param_shapes:
        mu = nd.random_normal(shape=shape, ctx=ctx, scale=weight_scale)
        rho = rho_offset + nd.zeros(shape=shape, ctx=ctx)
        mus.append(mu)
        rhos.append(rho)

    variational_params = mus + rhos
```

Since these are the parameters we wish to do gradient descent on, we need to allocate space for storing the gradients.

```
In [14]: for param in variational_params:
    param.attach_grad()
```

## Main training loop

The main training loop is pretty similar to the one we used in the MLP example. The only adaptation we need to make is to add the weight sampling which is performed during each optimization step. Generating a set of weights, which will subsequently be used in the neural network and the loss function, is a 3-step process:

1. Sample  $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}^d)$

```
In [15]: def sample_epsilon(param_shapes):
          epsilons = [nd.random_normal(shape=shape, loc=0., scale=1.0, ctx=ctx) for shape in
                      param_shapes]
          return epsilons
```

2. Transform  $\rho$  to a positive vector via the softplus function:

$$\sigma = \text{softplus}(\rho) = \log(1 + \exp(\rho))$$

```
In [16]: def softplus(x):
          return nd.log(1. + nd.exp(x))

          def transform_rhos(rhos):
              return [softplus(rho) for rho in rhos]
```

3. Compute  $\mathbf{w}$ :  $\mathbf{w} = \mu + \sigma \circ \epsilon$ , where the  $\circ$  operator represents the element-wise multiplication. This is the “reparametrization trick” for separating the randomness from the parameters of  $q$ .

```
In [17]: def transform_gaussian_samples(mus, sigmas, epsilons):
          samples = []
          for j in range(len(mus)):
              samples.append(mus[j] + sigmas[j] * epsilons[j])
          return samples
```

## Complete loop

The complete training loop is given below.

```
In [18]: epochs = config['epochs']
          learning_rate = config['learning_rate']
          smoothing_constant = .01
          train_acc = []
          test_acc = []

          for e in range(epochs):
              for i, (data, label) in enumerate(train_data):
                  data = data.as_in_context(ctx).reshape((-1, 784))
                  label = label.as_in_context(ctx)
                  label_one_hot = nd.one_hot(label, 10)

                  with autograd.record():
                      # sample epsilons from standard normal
                      epsilons = sample_epsilon(layer_param_shapes)

                      # compute softplus for variance
```



```

        sigmas = transform_rhos(rhos)

        # obtain a sample from  $q(w|\theta)$  by transforming the epsilons
        layer_params = transform_gaussian_samples(mus, sigmas, epsilons)

        # forward-propagate the batch
        output = net(data, layer_params)

        # calculate the loss
        loss = combined_loss(output, label_one_hot, layer_params, mus, sigmas,
                              gaussian_prior, log_softmax_likelihood)

        # backpropagate for gradient calculation
        loss.backward()

        # apply stochastic gradient descent to variational parameters
        SGD(variational_params, learning_rate)

        # calculate moving loss for monitoring convergence
        curr_loss = nd.mean(loss).asscalar()
        moving_loss = (curr_loss if ((i == 0) and (e == 0))
                       else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
                       * curr_loss)

        test_accuracy = evaluate_accuracy(test_data, net, mus)
        train_accuracy = evaluate_accuracy(train_data, net, mus)
        train_acc.append(np.asscalar(train_accuracy))
        test_acc.append(np.asscalar(test_accuracy))
        print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" %
              (e, moving_loss, train_accuracy, test_accuracy))

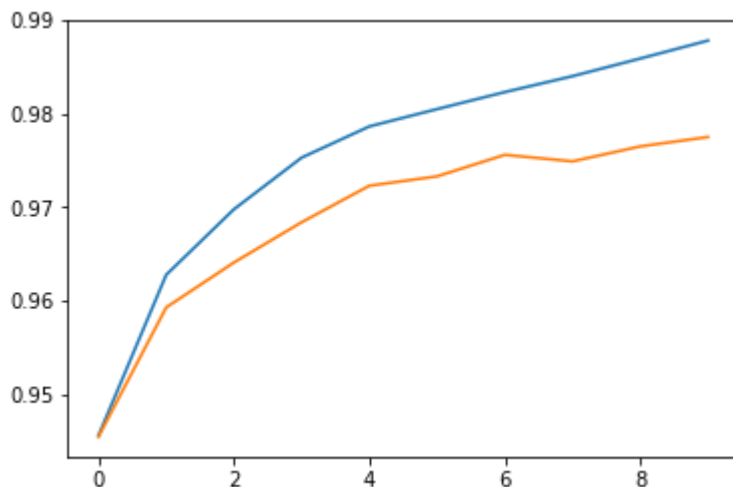
plt.plot(train_acc)
plt.plot(test_acc)
plt.show()

```

```

Epoch 0. Loss: 2626.47417991, Train_acc 0.945617, Test_acc 0.9455
Epoch 1. Loss: 2606.28165139, Train_acc 0.962783, Test_acc 0.9593
Epoch 2. Loss: 2600.2452303, Train_acc 0.969783, Test_acc 0.9641
Epoch 3. Loss: 2595.75639899, Train_acc 0.9753, Test_acc 0.9684
Epoch 4. Loss: 2592.98582057, Train_acc 0.978633, Test_acc 0.9723
Epoch 5. Loss: 2590.05895182, Train_acc 0.980483, Test_acc 0.9733
Epoch 6. Loss: 2588.57918775, Train_acc 0.9823, Test_acc 0.9756
Epoch 7. Loss: 2586.00932367, Train_acc 0.984, Test_acc 0.9749
Epoch 8. Loss: 2585.4614887, Train_acc 0.985883, Test_acc 0.9765
Epoch 9. Loss: 2582.92995846, Train_acc 0.9878, Test_acc 0.9775

```

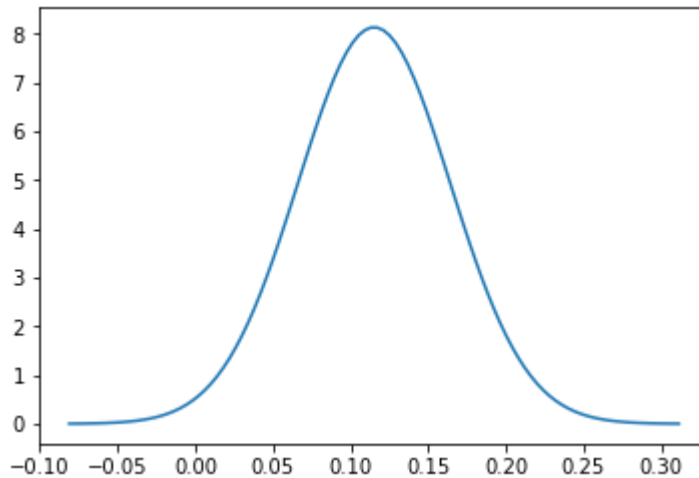


For demonstration purposes, we can now take a look at one particular weight by plotting its distribution.

```
In [19]: def show_weight_dist(mean, variance):
    sigma = nd.sqrt(variance)
    x = np.linspace(mean.asscalar() - 4*sigma.asscalar(), mean.asscalar() +
4*sigma.asscalar(), 100)
    plt.plot(x, gaussian(nd.array(x, ctx=ctx), mean, sigma).asnumpy())
    plt.show()

mu = mus[0][0][0]
var = softplus(rhos[0][0][0]) ** 2

show_weight_dist(mu, var)
```



Great! We have obtained a fully functional Bayesian neural network. However, the number of weights now is twice as high as for traditional neural networks. As we will see in the final section of this notebook, we are able to drastically reduce the number of weights our network uses for prediction with *weight pruning*.

## Weight pruning

To measure the degree of redundancy present in the trained network and to reduce the model's parameter count, we now want to examine the effect of setting some of the weights to 0 and evaluate the test accuracy afterwards. We can achieve this by ordering the weights according to their signal-to-noise-ratio,  $\frac{|\mu_i|}{\sigma_i}$ , and setting a certain percentage of the weights with the lowest ratios to 0.

We can calculate the signal-to-noise-ratio as follows:

```
In [20]: def signal_to_noise_ratio(mus, sigmas):
    sign_to_noise = []
    for j in range(len(mus)):
        sign_to_noise.extend([nd.abs(mus[j]) / sigmas[j]])
    return sign_to_noise
```

We further introduce a few helper methods which turn our list of weights into a single vector containing all weights. This will make our subsequent actions easier.

```
In [21]: def vectorize_matrices_in_vector(vec):
    for i in range(0, (num_layers + 1) * 2, 2):
        if i == 0:
            vec[i] = nd.reshape(vec[i], num_inputs * num_hidden)
        elif i == num_layers * 2:
            vec[i] = nd.reshape(vec[i], num_hidden * num_outputs)
        else:
            vec[i] = nd.reshape(vec[i], num_hidden * num_hidden)

    return vec

def concat_vectors_in_vector(vec):
    concat_vec = vec[0]
    for i in range(1, len(vec)):
        concat_vec = nd.concat(concat_vec, vec[i], dim=0)

    return concat_vec

def transform_vector_structure(vec):
    vec = vectorize_matrices_in_vector(vec)
    vec = concat_vectors_in_vector(vec)

    return vec
```

In addition, we also have a helper method which transforms the pruned weight vector back to the original layered structure.

```
In [22]: from functools import reduce
import operator

def prod(iterable):
    return reduce(operator.mul, iterable, 1)

def restore_weight_structure(vec):
    pruned_weights = []

    index = 0

    for shape in layer_param_shapes:
        incr = prod(shape)
        pruned_weights.extend([nd.reshape(vec[index : index + incr], shape)])
        index += incr

    return pruned_weights
```

The actual pruning of the vector happens in the following function. Note that this function accepts an ordered list of percentages to evaluate the performance at different pruning rates. In this setting, pruning at each iteration means extracting the index of the lowest signal-to-noise-ratio weight and setting the weight at this index to 0.

```
In [23]: def prune_weights(sign_to_noise_vec, prediction_vector, percentages):
    pruning_indices = nd.argsort(sign_to_noise_vec, axis=0)

    for percentage in percentages:
        prediction_vector = mus_copy_vec.copy()
        pruning_indices_percent = pruning_indices[0:int(len(pruning_indices)*percentage)]
        for pr_ind in pruning_indices_percent:
            prediction_vector[int(pr_ind.asscalar())] = 0
        pruned_weights = restore_weight_structure(prediction_vector)
        test_accuracy = evaluate_accuracy(test_data, net, pruned_weights)
        print("s --> s" % (percentage, test_accuracy))
```

Putting the above functions together:

```
In [24]: sign_to_noise = signal_to_noise_ratio(mus, sigmas)
         sign_to_noise_vec = transform_vector_structure(sign_to_noise)

         mus_copy = mus.copy()
         mus_copy_vec = transform_vector_structure(mus_copy)

         prune_weights(sign_to_noise_vec, mus_copy_vec, [0.1, 0.25, 0.5, 0.75, 0.95, 0.99, 1.0])

0.1 --> 0.9777
0.25 --> 0.9779
0.5 --> 0.9756
0.75 --> 0.9602
0.95 --> 0.7259
0.99 --> 0.3753
1.0 --> 0.098
```

Depending on the number of units used in the original network and the number of training epochs, the highest achievable pruning percentages (without significantly reducing the predictive performance) can vary. The paper, for example, reports almost no change in the test accuracy when pruning 95% of the weights in a 2x1200 unit Bayesian neural network, which creates a significantly sparser network, leading to faster predictions and reduced memory requirements.

## Conclusion

We have taken a look at an efficient Bayesian treatment for neural networks using variational inference via the “Bayes by Backprop” algorithm (introduced by the “[Weight Uncertainty in Neural Networks](#)” paper). We have implemented a stochastic version of the variational lower bound and optimized it in order to find an approximation to the posterior distribution over the weights of a MLP network on the MNIST data set. As a result, we achieve regularization on the network’s parameters and can quantify our uncertainty about the weights accurately. Finally, we saw that it is possible to significantly reduce the number of weights in the neural network after training while still keeping a high accuracy on the test set.

We also note that, given this model implementation, we were able to reproduce the paper’s results on the MNIST data set, achieving a comparable test accuracy for all documented instances of the MNIST classification problem.

For whinges or inquiries, [open an issue on GitHub](#).

## Bayes by Backprop with `gluon` (NN, classification)

After discussing [Bayes by Backprop from scratch](#) in a previous notebook, we can now look at the corresponding implementation as `gluon` components.

We start off with the usual set of imports.

```
In [1]: from __future__ import print_function
import collections
import mxnet as mx
import numpy as np
from mxnet import nd, autograd
from matplotlib import pyplot as plt
from mxnet import gluon
```

For easy tuning and experimentation, we define a dictionary holding the hyper-parameters of our model.

```
In [2]: config = {
    "num_hidden_layers": 2,
    "num_hidden_units": 400,
    "batch_size": 128,
    "epochs": 10,
    "learning_rate": 0.001,
    "num_samples": 1,
    "pi": 0.25,
    "sigma_p": 1.0,
    "sigma_p1": 0.75,
    "sigma_p2": 0.01,
}
```

Also, we specify the device context for MXNet.

```
In [3]: ctx = mx.cpu()
```

## Load MNIST data

We will again train and evaluate the algorithm on the MNIST data set and therefore load the data set as follows:

```
In [4]: def transform(data, label):
    return data.astype(np.float32)/126.0, label.astype(np.float32)

mnist = mx.test_utils.get_mnist()
```

```

num_inputs = 784
num_outputs = 10
batch_size = config['batch_size']

train_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=True,
transform=transform),
                                     batch_size, shuffle=True)
test_data = mx.gluon.data.DataLoader(mx.gluon.data.vision.MNIST(train=False,
transform=transform),
                                     batch_size, shuffle=False)

num_train = sum([batch_size for i in train_data])
num_batches = num_train / batch_size

```

In order to reproduce and compare the results from the paper, we preprocess the pixels by dividing by 126.

## Model definition

### Neural net modeling

As our model we are using a straightforward MLP and we are wiring up our network just as we are used to in `gluon`. Note that we are not using any special layers during the definition of our network, as we believe that Bayes by Backprop should be thought of as a training method, rather than a special architecture.

```

In [5]: num_layers = config['num_hidden_layers']
        num_hidden = config['num_hidden_units']

        net = gluon.nn.Sequential()
        with net.name_scope():
            for i in range(num_layers):
                net.add(gluon.nn.Dense(num_hidden, activation="relu"))
            net.add(gluon.nn.Dense(num_outputs))

```

## Build objective/loss

Again, we define our loss function as described in [Bayes by Backprop from scratch](#). Note that we are bundling all of this functionality as part of a `gluon.loss.Loss` subclass, where the loss computation is performed in the `hybrid_forward` function.

```

In [6]: class BBBLoss(gluon.loss.Loss):
        def __init__(self, log_prior="gaussian", log_likelihood="softmax_cross_entropy",
                     sigma_p1=1.0, sigma_p2=0.1, pi=0.5, weight=None, batch_axis=0, **kwargs):
            super(BBBLoss, self).__init__(weight, batch_axis, **kwargs)
            self.log_prior = log_prior
            self.log_likelihood = log_likelihood
            self.sigma_p1 = sigma_p1
            self.sigma_p2 = sigma_p2
            self.pi = pi

        def log_softmax_likelihood(self, yhat_linear, y):
            return nd.nansum(y * nd.log_softmax(yhat_linear), axis=0, exclude=True)

```

```

def log_gaussian(self, x, mu, sigma):
    return -0.5 * np.log(2.0 * np.pi) - nd.log(sigma) - (x - mu) ** 2 / (2 * sigma **
2)

def gaussian_prior(self, x):
    sigma_p = nd.array([self.sigma_p1], ctx=ctx)
    return nd.sum(self.log_gaussian(x, 0., sigma_p))

def gaussian(self, x, mu, sigma):
    scaling = 1.0 / nd.sqrt(2.0 * np.pi * (sigma ** 2))
    bell = nd.exp(-(x - mu) ** 2 / (2.0 * sigma ** 2))

    return scaling * bell

def scale_mixture_prior(self, x):
    sigma_p1 = nd.array([self.sigma_p1], ctx=ctx)
    sigma_p2 = nd.array([self.sigma_p2], ctx=ctx)
    pi = self.pi

    first_gaussian = pi * self.gaussian(x, 0., sigma_p1)
    second_gaussian = (1 - pi) * self.gaussian(x, 0., sigma_p2)

    return nd.log(first_gaussian + second_gaussian)

def hybrid_forward(self, F, output, label, params, mus, sigmas, sample_weight=None):
    log_likelihood_sum = nd.sum(self.log_softmax_likelihood(output, label))
    prior = None
    if self.log_prior == "gaussian":
        prior = self.gaussian_prior
    elif self.log_prior == "scale_mixture":
        prior = self.scale_mixture_prior
    log_prior_sum = sum([nd.sum(prior(param)) for param in params])
    log_var_posterior_sum = sum([nd.sum(self.log_gaussian(params[i], mus[i],
sigmas[i])) for i in range(len(params))])
    return 1.0 / num_batches * (log_var_posterior_sum - log_prior_sum) -
log_likelihood_sum

bbb_loss = BBBLoss(log_prior="scale_mixture", sigma_p1=config['sigma_p1'],
sigma_p2=config['sigma_p2'])

```

## Parameter initialization

First, we need to initialize all the network's parameters, which are only point estimates of the weights at this point. We will soon see, how we can still train the network in a Bayesian fashion, without interfering with the network's architecture.

```
In [7]: net.collect_params().initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
```

Then we have to forward-propagate a single data set entry once to set up all network parameters (weights and biases) with the desired initializer specified above.

```
In [8]: for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(ctx).reshape((-1, 784))
        net(data)
        break
```

```
In [9]: weight_scale = .1
        rho_offset = -3

        # initialize variational parameters; mean and variance for each weight
        mus = []
```

```

rhos = []

shapes = list(map(lambda x: x.shape, net.collect_params().values()))

for shape in shapes:
    mu = gluon.Parameter('mu', shape=shape, init=mx.init.Normal(weight_scale))
    rho = gluon.Parameter('rho', shape=shape, init=mx.init.Constant(rho_offset))
    mu.initialize(ctx=ctx)
    rho.initialize(ctx=ctx)
    mus.append(mu)
    rhos.append(rho)

variational_params = mus + rhos

raw_mus = list(map(lambda x: x.data(ctx), mus))
raw_rhos = list(map(lambda x: x.data(ctx), rhos))

```

## Optimizer

Now, we still have to choose the optimizer we wish to use for training. This time, we are using the `adam` optimizer.

```

In [10]: trainer = gluon.Trainer(variational_params, 'adam', {'learning_rate':
    config['learning_rate']})

```

## Main training loop

### Sampling

Recall the 3-step process for the variational parameters:

1. Sample  $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}^d)$

```

In [11]: def sample_epsilon(param_shapes):
    epsilons = [nd.random_normal(shape=shape, loc=0., scale=1.0, ctx=ctx) for shape in
    param_shapes]
    return epsilons

```

2. Transform  $\rho$  to a positive vector via the softplus function:

$$\sigma = \text{softplus}(\rho) = \log(1 + \exp(\rho))$$

```

In [12]: def softplus(x):
    return nd.log(1. + nd.exp(x))

def transform_rhos(rhos):
    return [softplus(rho) for rho in rhos]

```

3. Compute  $\mathbf{w}$ :  $\mathbf{w} = \mu + \sigma \circ \epsilon$ , where the  $\circ$  operator represents the element-wise multiplication. This is the “reparametrization trick” for separating the randomness from the parameters of  $q$ .



```
In [13]: def transform_gaussian_samples(mus, sigmas, epsilons):
    samples = []
    for j in range(len(mus)):
        samples.append(mus[j] + sigmas[j] * epsilons[j])
    return samples
```

Putting these three steps together we get:

```
In [14]: def generate_weight_sample(layer_param_shapes, mus, rhos):
    # sample epsilons from standard normal
    epsilons = sample_epsilons(layer_param_shapes)

    # compute softplus for variance
    sigmas = transform_rhos(rhos)

    # obtain a sample from q(w|theta) by transforming the epsilons
    layer_params = transform_gaussian_samples(mus, sigmas, epsilons)

    return layer_params, sigmas
```

## Evaluation metric

In order to being able to assess our model performance we define a helper function which evaluates our accuracy on an ongoing basis.

```
In [15]: def evaluate_accuracy(data_iterator, net, layer_params):
    numerator = 0.
    denominator = 0.
    for i, (data, label) in enumerate(data_iterator):
        data = data.as_in_context(ctx).reshape((-1, 784))
        label = label.as_in_context(ctx)

        for l_param, param in zip(layer_params, net.collect_params().values()):
            param._data[list(param._data.keys())[0]] = l_param

        output = net(data)
        predictions = nd.argmax(output, axis=1)
        numerator += nd.sum(predictions == label)
        denominator += data.shape[0]
    return (numerator / denominator).asscalar()
```

## Complete loop

The complete training loop is given below.

```
In [16]: epochs = config['epochs']
    learning_rate = config['learning_rate']
    smoothing_constant = .01
    train_acc = []
    test_acc = []

    for e in range(epochs):
        for i, (data, label) in enumerate(train_data):
            data = data.as_in_context(ctx).reshape((-1, 784))
            label = label.as_in_context(ctx)
            label_one_hot = nd.one_hot(label, 10)

            with autograd.record():
```

```

# generate sample
layer_params, sigmas = generate_weight_sample(shapes, raw_mus, raw_rhos)

# overwrite network parameters with sampled parameters
for sample, param in zip(layer_params, net.collect_params().values()):
    param._data[list(param._data.keys())[0]] = sample

# forward-propagate the batch
output = net(data)

# calculate the loss
loss = bbb_loss(output, label_one_hot, layer_params, raw_mus, sigmas)

# backpropagate for gradient calculation
loss.backward()

trainer.step(data.shape[0])

# calculate moving loss for monitoring convergence
curr_loss = nd.mean(loss).asscalar()
moving_loss = (curr_loss if ((i == 0) and (e == 0))
               else (1 - smoothing_constant) * moving_loss + (smoothing_constant)
               * curr_loss)

test_accuracy = evaluate_accuracy(test_data, net, raw_mus)
train_accuracy = evaluate_accuracy(train_data, net, raw_mus)
train_acc.append(np.asscalar(train_accuracy))
test_acc.append(np.asscalar(test_accuracy))
print("Epoch %s. Loss: %s, Train_acc %s, Test_acc %s" %
      (e, moving_loss, train_accuracy, test_accuracy))

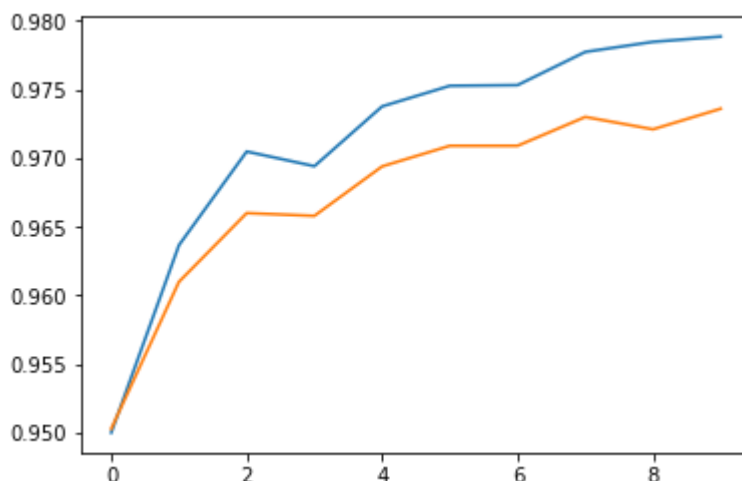
plt.plot(train_acc)
plt.plot(test_acc)
plt.show()

```

```

Epoch 0. Loss: 2121.26601683, Train_acc 0.950017, Test_acc 0.9503
Epoch 1. Loss: 1918.8522369, Train_acc 0.963667, Test_acc 0.961
Epoch 2. Loss: 1813.43826684, Train_acc 0.970483, Test_acc 0.966
Epoch 3. Loss: 1740.46931458, Train_acc 0.969417, Test_acc 0.9658
Epoch 4. Loss: 1681.04620544, Train_acc 0.973767, Test_acc 0.9694
Epoch 5. Loss: 1625.9179831, Train_acc 0.975267, Test_acc 0.9709
Epoch 6. Loss: 1568.97912286, Train_acc 0.975317, Test_acc 0.9709
Epoch 7. Loss: 1509.50606071, Train_acc 0.977733, Test_acc 0.973
Epoch 8. Loss: 1449.39600539, Train_acc 0.978467, Test_acc 0.9721
Epoch 9. Loss: 1390.66561781, Train_acc 0.97885, Test_acc 0.9736

```



For demonstration purposes, we can now take a look at one particular weight by plotting its distribution.

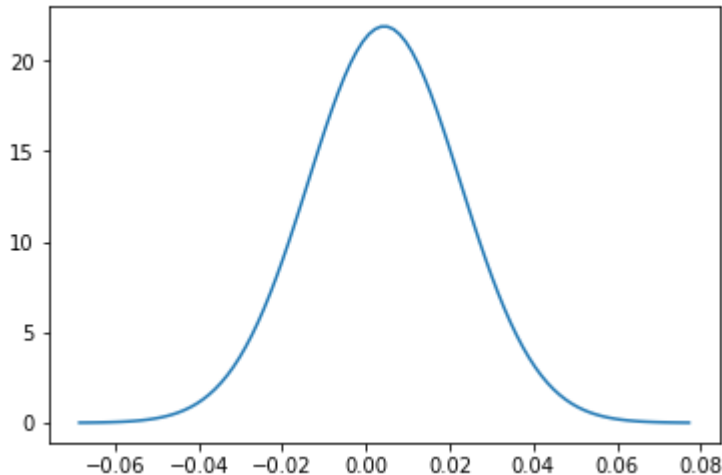
```
In [17]: def gaussian(x, mu, sigma):
scaling = 1.0 / nd.sqrt(2.0 * np.pi * (sigma ** 2))
bell = nd.exp(- (x - mu) ** 2 / (2.0 * sigma ** 2))

return scaling * bell

def show_weight_dist(mean, variance):
sigma = nd.sqrt(variance)
x = np.linspace(mean.asscalar() - 4*sigma.asscalar(), mean.asscalar() +
4*sigma.asscalar(), 100)
plt.plot(x, gaussian(nd.array(x, ctx=ctx), mean, sigma).asnumpy())
plt.show()

mu = raw_mus[0][0][0]
var = softplus(raw_rhos[0][0][0]) ** 2

show_weight_dist(mu, var)
```



## Weight pruning

To measure the degree of redundancy present in the trained network and to reduce the model's parameter count, we now want to examine the effect of setting some of the weights to 0 and evaluate the test accuracy afterwards. We can achieve this by ordering the weights according to their signal-to-noise-ratio,  $\frac{|\mu_i|}{\sigma_i}$ , and setting a certain percentage of the weights with the lowest ratios to 0.

We can calculate the signal-to-noise-ratio as follows:

```
In [18]: def signal_to_noise_ratio(mus, sigmas):
sign_to_noise = []
for j in range(len(mus)):
sign_to_noise.extend([nd.abs(mus[j]) / sigmas[j]])
return sign_to_noise
```

We further introduce a few helper methods which turn our list of weights into a single vector containing all weights. This will make our subsequent actions easier.

```
In [19]: def vectorize_matrices_in_vector(vec):
for i in range(0, (num_layers + 1) * 2, 2):
```

```

        if i == 0:
            vec[i] = nd.reshape(vec[i], num_inputs * num_hidden)
        elif i == num_layers * 2:
            vec[i] = nd.reshape(vec[i], num_hidden * num_outputs)
        else:
            vec[i] = nd.reshape(vec[i], num_hidden * num_hidden)

    return vec

def concat_vectors_in_vector(vec):
    concat_vec = vec[0]
    for i in range(1, len(vec)):
        concat_vec = nd.concat(concat_vec, vec[i], dim=0)

    return concat_vec

def transform_vector_structure(vec):
    vec = vectorize_matrices_in_vector(vec)
    vec = concat_vectors_in_vector(vec)

    return vec

```

In addition, we also have a helper method which transforms the pruned weight vector back to the original layered structure.

```

In [20]: from functools import reduce
import operator

def prod(iterable):
    return reduce(operator.mul, iterable, 1)

def restore_weight_structure(vec):
    pruned_weights = []

    index = 0

    for shape in shapes:
        incr = prod(shape)
        pruned_weights.extend([nd.reshape(vec[index : index + incr], shape)])
        index += incr

    return pruned_weights

```

The actual pruning of the vector happens in the following function. Note that this function accepts an ordered list of percentages to evaluate the performance at different pruning rates. In this setting, pruning at each iteration means extracting the index of the lowest signal-to-noise-ratio weight and setting the weight at this index to 0.

```

In [21]: def prune_weights(sign_to_noise_vec, prediction_vector, percentages):
    pruning_indices = nd.argsort(sign_to_noise_vec, axis=0)

    for percentage in percentages:
        prediction_vector = mus_copy_vec.copy()
        pruning_indices_percent = pruning_indices[0:int(len(pruning_indices)*percentage)]
        for pr_ind in pruning_indices_percent:
            prediction_vector[int(pr_ind.asscalar())] = 0
        pruned_weights = restore_weight_structure(prediction_vector)
        test_accuracy = evaluate_accuracy(test_data, net, pruned_weights)
        print("%s --> %s" % (percentage, test_accuracy))

```

Putting the above function together:

```
In [22]: sign_to_noise = signal_to_noise_ratio(raw_mus, sigmas)
         sign_to_noise_vec = transform_vector_structure(sign_to_noise)

         mus_copy = raw_mus.copy()
         mus_copy_vec = transform_vector_structure(mus_copy)

         prune_weights(sign_to_noise_vec, mus_copy_vec, [0.1, 0.25, 0.5, 0.75, 0.95, 0.98, 1.0])

0.1 --> 0.9737
0.25 --> 0.9737
0.5 --> 0.9748
0.75 --> 0.9754
0.95 --> 0.9697
0.98 --> 0.9549
1.0 --> 0.098
```

Depending on the number of units used in the original network, the highest achievable pruning percentages (without significantly reducing the predictive performance) can vary. The paper, for example, reports almost no change in the test accuracy when pruning 95% of the weights in a 1200 unit Bayesian neural network, which creates a significantly sparser network, leading to faster predictions and reduced memory requirements.

## Conclusion

We have taken a look at an efficient Bayesian treatment for neural networks using variational inference via the “Bayes by Backprop” algorithm (introduced by the “[Weight Uncertainty in Neural Networks](#)” paper). We have implemented a stochastic version of the variational lower bound and optimized it in order to find an approximation to the posterior distribution over the weights of a MLP network on the MNIST data set. As a result, we achieve regularization on the network’s parameters and can quantify our uncertainty about the weights accurately. Finally, we saw that it is possible to significantly reduce the number of weights in the neural network after training while still keeping a high accuracy on the test set.

We also note that, given this model implementation, we were able to reproduce the paper’s results on the MNIST data set, achieving a comparable test accuracy for all documented instances of the MNIST classification problem.

For whinges or inquiries, [open an issue on GitHub](#).

# Kaggle house price prediction with `gluon` and k-fold cross-validation

Updates: `Share your score and method here <<https://discuss.mxnet.io/t/kaggle-exercise-1-house-price-prediction-with-gluon/51>>` \_\_.

How have you been doing so far on the journey of `Deep Learning---the Straight Dope`?

It's time to get your hands dirty.

Let's get started.

## Introduction

In this tutorial, we introduce how to use `gluon` for competition on [Kaggle](#). Specifically, we will take the [house price prediction problem](#) as a case study.

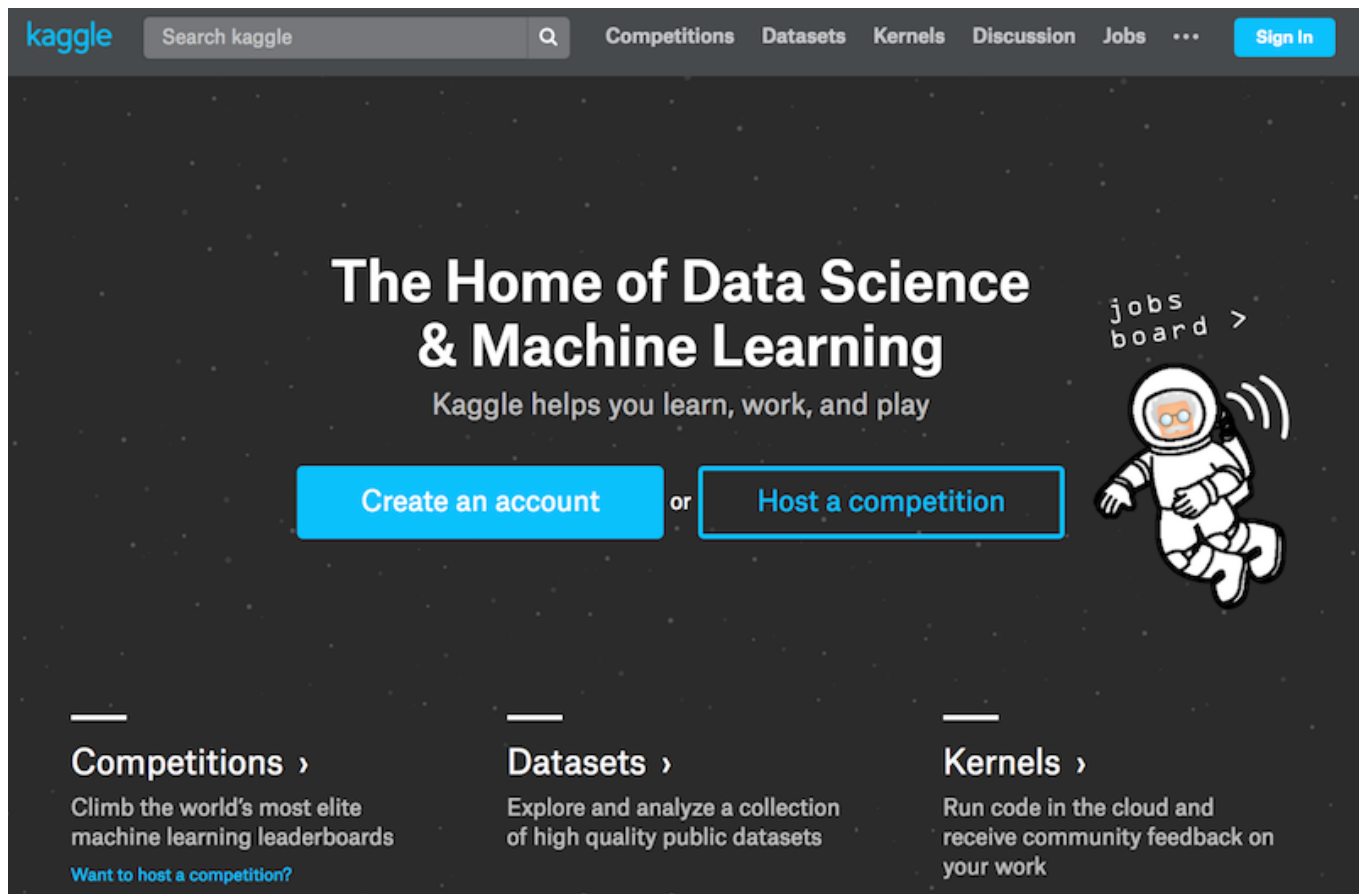
We will provide a basic end-to-end pipeline for completing a common Kaggle challenge. We will demonstrate how to use `pandas` to pre-process the real-world data sets, such as:

- Handling categorical data
- Handling missing values
- Standardizing numerical features

Note that this tutorial only provides a very basic pipeline. We expect you to tweak the following code, such as re-designing models and fine-tuning parameters, to obtain a desirable score on Kaggle.

## House Price Prediction Problem on Kaggle

[Kaggle](#) is a popular platform for people who love machine learning. To submit the results, please register a [Kaggle](#) account. Please note that, **Kaggle limits the number of daily submissions to 10.**



We take the [house price prediction problem](#) as an example to demonstrate how to complete a Kaggle competition with [Gluon](#). Please learn details of the problem by clicking the [link to the house price prediction problem](#) before starting.



## House Prices: Advanced Regression Techniques

Predict sales prices and practice feature engineering, RFs, and gradient boosting  
1,698 teams · 2 years to go

[Overview](#) [Data](#) [Kernels](#) [Discussion](#) [Leaderboard](#) [Rules](#)

### Overview

#### Description

#### Evaluation

#### Frequently Asked Questions

#### Tutorials

### Start here if...

You have some experience with R or Python and machine learning basics. This is a perfect competition for data science students who have completed an online course in machine learning and are looking to expand their skill set before trying a featured competition.

### Competition Description

## Load the data set

There are separate training and testing data sets for this competition. Both data sets describe features of every house, such as type of the street, year of building, and basement conditions. Such features can be numeric, categorical, or even missing (`na`). Only the training data set has the sale price of each house, which shall be predicted based on features of the testing data set.

The data sets can be downloaded via the [link to problem](#). Specifically, you can directly access the [training data set](#) and the [testing data set](#) after logging in Kaggle.

We load the data via `pandas`. Please make sure that it is installed (`pip install pandas`).

```
In [1]: import pandas as pd
import numpy as np

train = pd.read_csv("../data/kaggle/house_pred_train.csv")
test = pd.read_csv("../data/kaggle/house_pred_test.csv")
all_X = pd.concat((train.loc[:, 'MSSubClass': 'SaleCondition'],
                  test.loc[:, 'MSSubClass': 'SaleCondition']))
```

We can take a look at a few rows of the training data set.



```
In [2]: train.head()
```

```
Out[2]:
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl

5 rows × 81 columns

Here is the shapes of the data sets.

```
In [3]: train.shape
```

```
Out[3]: (1460, 81)
```

```
In [4]: test.shape
```

```
Out[4]: (1459, 80)
```

## Pre-processing data

We can use `pandas` to standardize the numerical features:

$$x_i = \frac{x_i - \mathbb{E}x_i}{\text{std}(x_i)}$$

```
In [5]: numeric_feats = all_X.dtypes[all_X.dtypes != "object"].index
all_X[numeric_feats] = all_X[numeric_feats].apply(
    lambda x: (x - x.mean()) / (x.std()))
```

Let us transform categorical values to numerical ones.

```
In [6]: all_X = pd.get_dummies(all_X, dummy_na=True)
```

We can approximate the missing values by the mean values of the current feature.

```
In [7]: all_X = all_X.fillna(all_X.mean())
```

Let us convert the formats of the data sets.

```
In [8]: num_train = train.shape[0]
X_train = all_X[:num_train].as_matrix()
```

```
X_test = all_X[num_train:].as_matrix()
y_train = train.SalePrice.as_matrix()
```

## Loading data in `NDArray`

To facilitate the interactions with `Gluon`, we need to load data in the `NDArray` format.

```
In [9]: from mxnet import ndarray as nd
        from mxnet import autograd
        from mxnet import gluon

        X_train = nd.array(X_train)
        y_train = nd.array(y_train)
        y_train.reshape((num_train, 1))

        X_test = nd.array(X_test)
```

We define the loss function to be the squared loss.

```
In [10]: square_loss = gluon.loss.L2Loss()
```

Below defines the root mean square loss between the logarithm of the predicted values and the true values used in the competition.

```
In [11]: def get_rmse_log(net, X_train, y_train):
        num_train = X_train.shape[0]
        clipped_preds = nd.clip(net(X_train), 1, float('inf'))
        return np.sqrt(2 * nd.sum(square_loss(
            nd.log(clipped_preds), nd.log(y_train))).asscalar() / num_train)
```

## Define the model

We define a **basic** linear regression model here. This may be modified to achieve better results on Kaggle.

```
In [12]: def get_net():
        net = gluon.nn.Sequential()
        with net.name_scope():
            net.add(gluon.nn.Dense(1))
        net.initialize()
        return net
```

We define the training function.

```
In [13]: %matplotlib inline
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 120
import matplotlib.pyplot as plt

def train(net, X_train, y_train, X_test, y_test, epochs,
```

```

        verbose_epoch, learning_rate, weight_decay):
train_loss = []
if X_test is not None:
    test_loss = []
batch_size = 100
dataset_train = gluon.data.ArrayDataset(X_train, y_train)
data_iter_train = gluon.data.DataLoader(
    dataset_train, batch_size, shuffle=True)
trainer = gluon.Trainer(net.collect_params(), 'adam',
                        {'learning_rate': learning_rate,
                         'wd': weight_decay})
net.collect_params().initialize(force_reinit=True)
for epoch in range(epochs):
    for data, label in data_iter_train:
        with autograd.record():
            output = net(data)
            loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)

    cur_train_loss = get_rmse_log(net, X_train, y_train)
    if epoch > verbose_epoch:
        print("Epoch %d, train loss: %f" % (epoch, cur_train_loss))
    train_loss.append(cur_train_loss)
    if X_test is not None:
        cur_test_loss = get_rmse_log(net, X_test, y_test)
        test_loss.append(cur_test_loss)
plt.plot(train_loss)
plt.legend(['train'])
if X_test is not None:
    plt.plot(test_loss)
    plt.legend(['train', 'test'])
plt.show()
if X_test is not None:
    return cur_train_loss, cur_test_loss
else:
    return cur_train_loss

```

## K-Fold Cross-Validation

We described the [overfitting problem](#), where we cannot rely on the training loss to infer the testing loss. In fact, when we fine-tune the parameters, we typically rely on  $k$ -fold cross-validation.

In  $k$ -fold cross-validation, we divide the training data sets into  $k$  subsets, where one set is used for the validation and the remaining  $k - 1$  subsets are used for training.

We care about the average training loss and average testing loss of the  $k$  experimental results. Hence, we can define the  $k$ -fold cross-validation as follows.

```

In [14]: def k_fold_cross_valid(k, epochs, verbose_epoch, X_train, y_train,
                                learning_rate, weight_decay):
    assert k > 1
    fold_size = X_train.shape[0] // k
    train_loss_sum = 0.0
    test_loss_sum = 0.0
    for test_i in range(k):
        X_val_test = X_train[test_i * fold_size: (test_i + 1) * fold_size, :]
        y_val_test = y_train[test_i * fold_size: (test_i + 1) * fold_size]

        val_train_defined = False

```

```

for i in range(k):
    if i != test_i:
        X_cur_fold = X_train[i * fold_size: (i + 1) * fold_size, :]
        y_cur_fold = y_train[i * fold_size: (i + 1) * fold_size]
        if not val_train_defined:
            X_val_train = X_cur_fold
            y_val_train = y_cur_fold
            val_train_defined = True
        else:
            X_val_train = nd.concat(X_val_train, X_cur_fold, dim=0)
            y_val_train = nd.concat(y_val_train, y_cur_fold, dim=0)
    net = get_net()
    train_loss, test_loss = train(
        net, X_val_train, y_val_train, X_val_test, y_val_test,
        epochs, verbose_epoch, learning_rate, weight_decay)
    train_loss_sum += train_loss
    print("Test loss: %f" % test_loss)
    test_loss_sum += test_loss
return train_loss_sum / k, test_loss_sum / k

```

## Train and cross-validate the model

The following parameters can be fine-tuned.

```

In [15]: k = 5
         epochs = 100
         verbose_epoch = 95
         learning_rate = 5
         weight_decay = 0.0

```

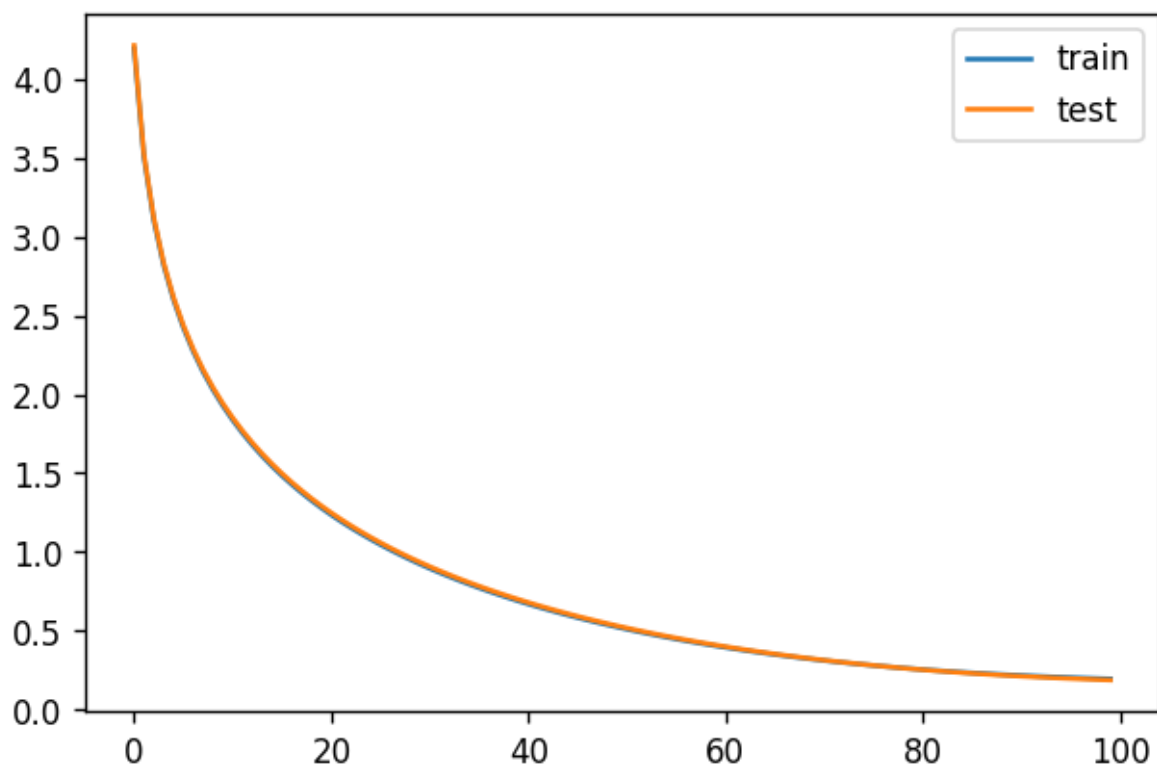
Given the above parameters, we can train and cross-validate our model.

```

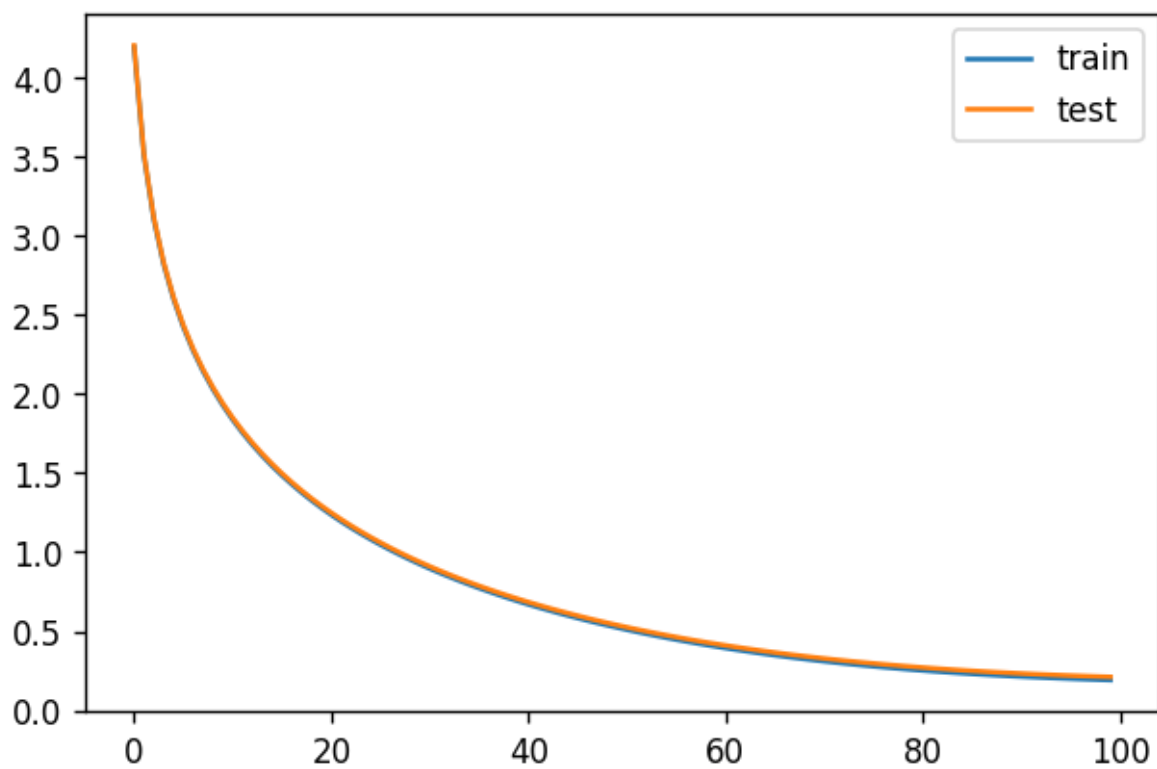
In [16]: train_loss, test_loss = k_fold_cross_valid(k, epochs, verbose_epoch, X_train,
         y_train, learning_rate, weight_decay)
         print("%d-fold validation: Avg train loss: %f, Avg test loss: %f" %
         (k, train_loss, test_loss))

Epoch 96, train loss: 0.201713
Epoch 97, train loss: 0.199597
Epoch 98, train loss: 0.197594
Epoch 99, train loss: 0.195709

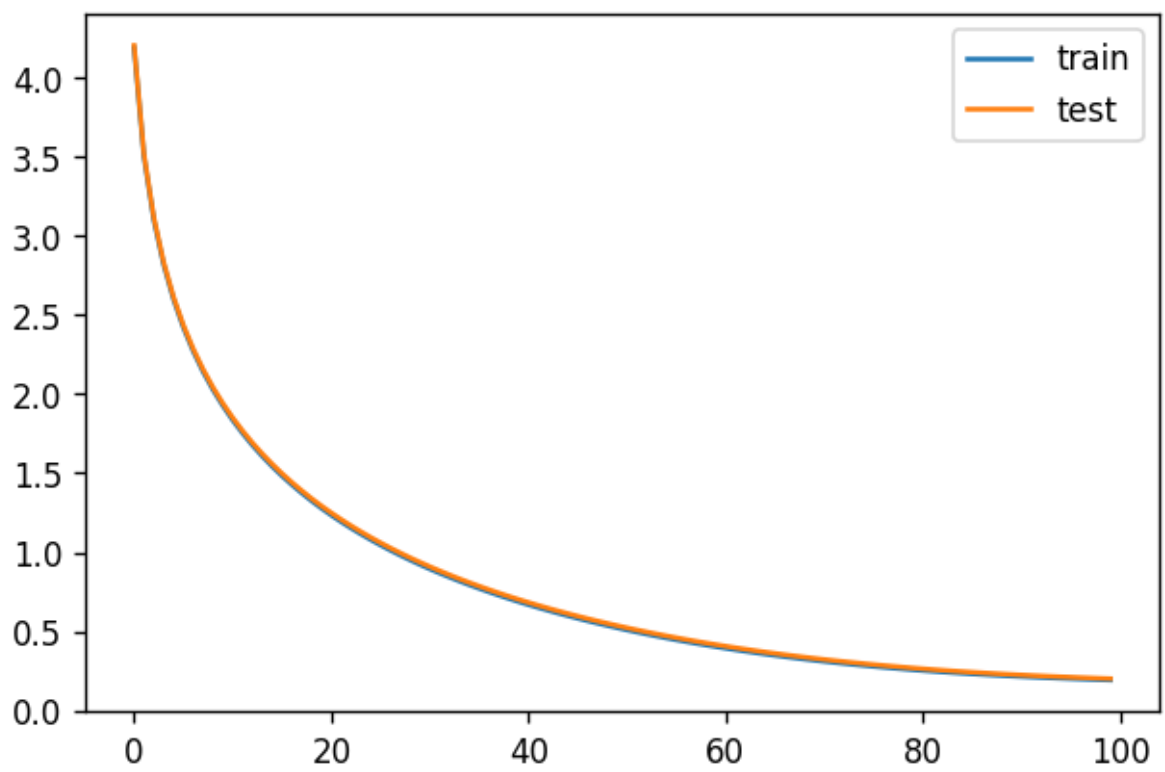
```



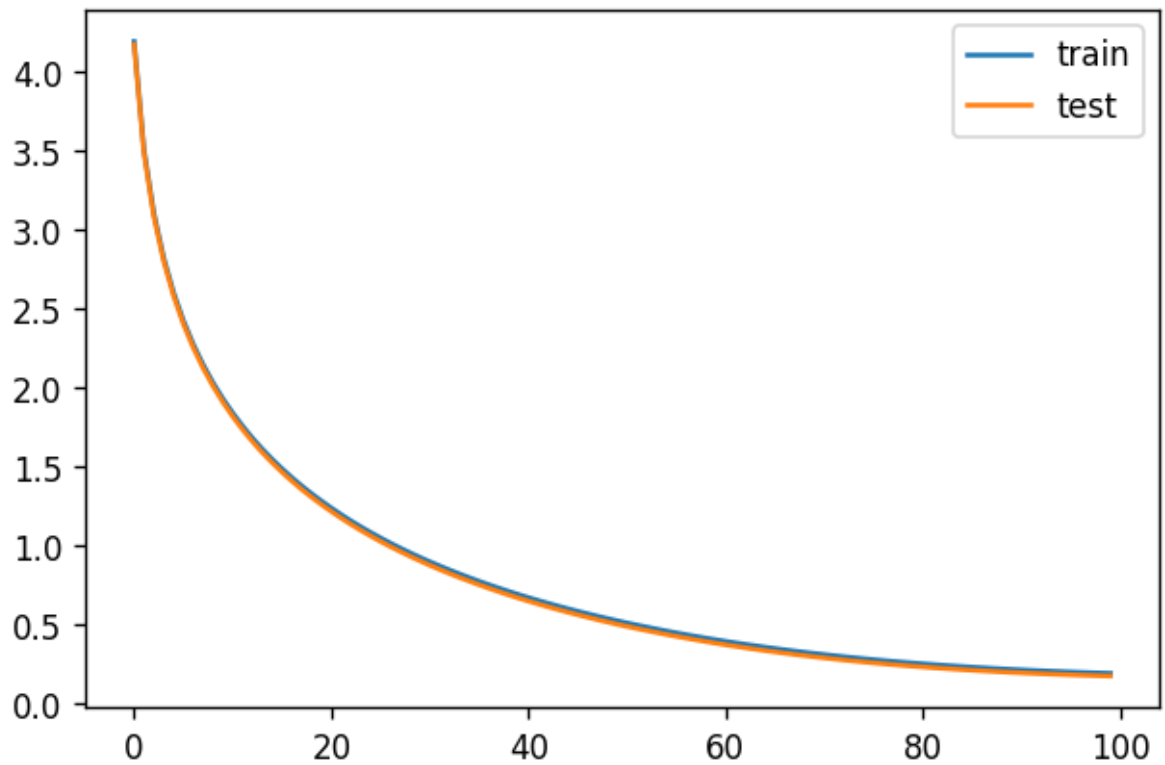
Test loss: 0.188469  
Epoch 96, train loss: 0.198065  
Epoch 97, train loss: 0.195917  
Epoch 98, train loss: 0.193840  
Epoch 99, train loss: 0.191888



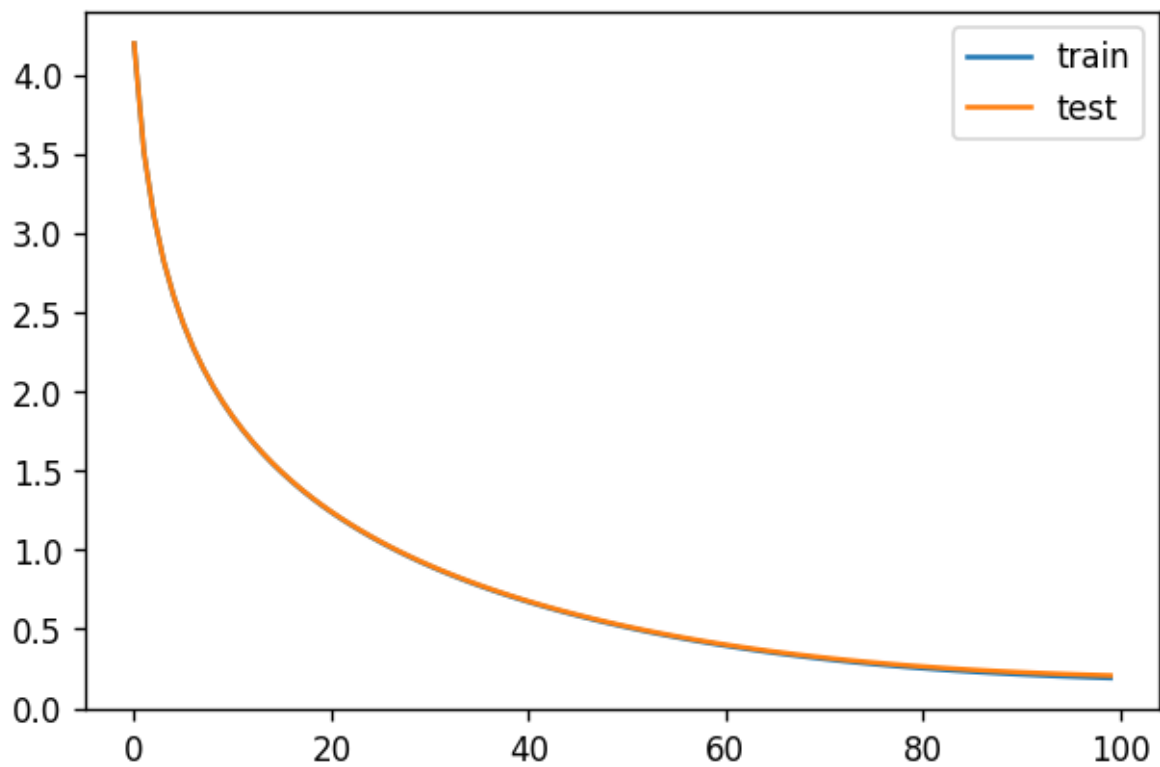
Test loss: 0.211287  
Epoch 96, train loss: 0.199244  
Epoch 97, train loss: 0.197074  
Epoch 98, train loss: 0.194988  
Epoch 99, train loss: 0.193050



Test loss: 0.201556  
Epoch 96, train loss: 0.201637  
Epoch 97, train loss: 0.199430  
Epoch 98, train loss: 0.197397  
Epoch 99, train loss: 0.195466



Test loss: 0.178427  
Epoch 96, train loss: 0.196650  
Epoch 97, train loss: 0.194443  
Epoch 98, train loss: 0.192334  
Epoch 99, train loss: 0.190332



Test loss: 0.206326  
 5-fold validation: Avg train loss: 0.193289, Avg test loss: 0.197213

After fine-tuning, even though the training loss can be very low, but the validation loss for the  $k$ -fold cross validation can be very high. Thus, when the training loss is very low, we need to observe whether the validation loss is reduced at the same time and watch out for overfitting. We often rely on  $k$ -fold cross-validation to fine-tune parameters.

## Make predictions and submit results on Kaggle

Let us define the prediction function.

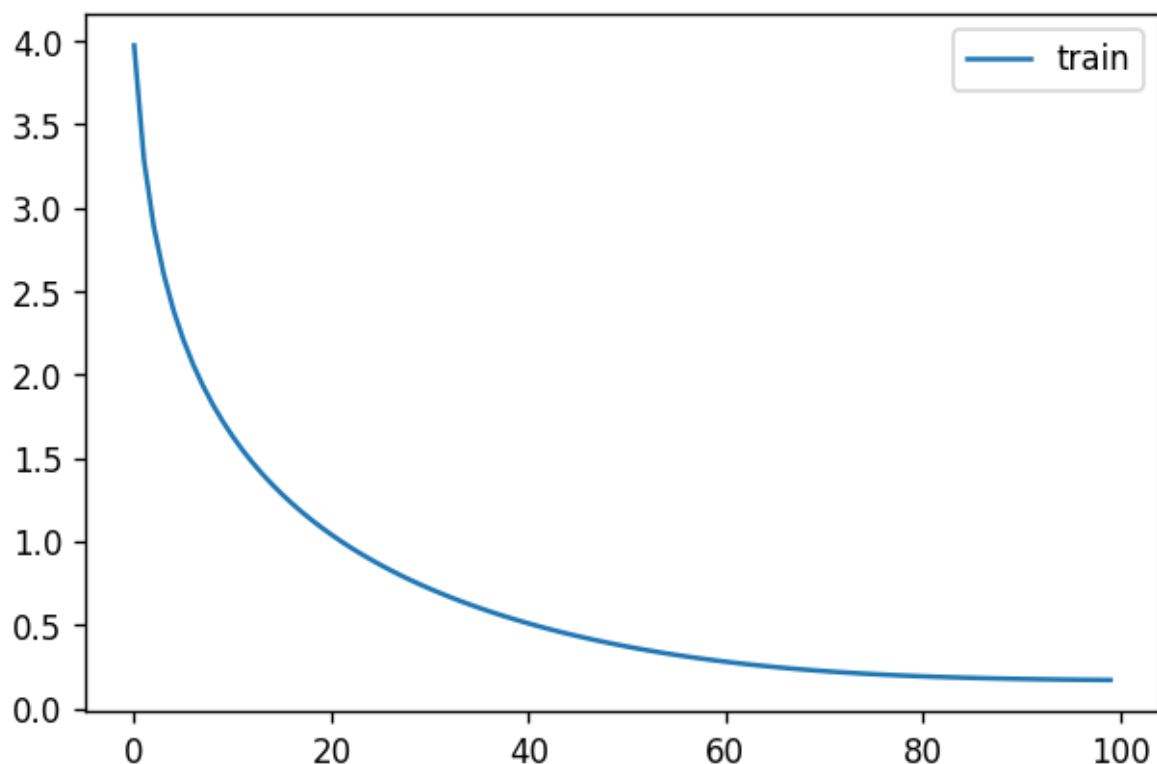
```
In [17]: def learn(epochs, verbose_epoch, X_train, y_train, test, learning_rate,
            weight_decay):
            net = get_net()
            train(net, X_train, y_train, None, None, epochs, verbose_epoch,
                  learning_rate, weight_decay)
            preds = net(X_test).asnumpy()
            test['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
            submission = pd.concat([test['Id'], test['SalePrice']], axis=1)
            submission.to_csv('submission.csv', index=False)
```

After fine-tuning parameters, we can predict and submit results on Kaggle.


```
In [18]: learn(epochs, verbose_epoch, X_train, y_train, test, learning_rate,
            weight_decay)
```

Epoch 96, train loss: 0.171151  
 Epoch 97, train loss: 0.170580

Epoch 98, train loss: 0.170049  
Epoch 99, train loss: 0.169546



After executing the above code, a `submission.csv` file will be generated. It is in the required format by Kaggle. Now, we can submit our predicted sales prices of houses based on the testing data set and compare them with the true prices on Kaggle. You need to log in Kaggle, open the [link to the house prediction problem](#), and click the `Submit Predictions` button.



### House Prices: Advanced Regression Techniques


Predict sales prices and practice feature engineering, RFs, and gradient boosting  
1,698 teams · 2 years to go

[Overview](#) [Data](#) [Kernels](#) [Discussion](#) [Leaderboard](#) [Rules](#) [Team](#) [My Submissions](#) [Submit Predictions](#)

You may click the `Upload Submission File` button to select your results. Then click `Make Submission` at the bottom to view your results.



**Step 1**  
Upload submission file

  
Upload Submission File


File Format

Your submission should be in CSV format.  
You can upload this in a zip/gz/rar/7z archive, if you prefer.

Number of Predictions

We expect the solution file to have 1459 prediction rows. This file should have a header row. Please see sample submission file on the [data page](#).

**Step 2**  
Describe submission

**B** */* | 🔗 🗨️ </> 🖼️ | ☰ ☷ H | ↺ ↻ |  Styling with Markdown supported

Briefly describe your submission.

Make Submission

Just a kind reminder again, **Kaggle limits the number of daily submissions to 10.**

## Exercise (**Share your score and method here**):

- What loss can you obtain on Kaggle by using this tutorial?
- By re-designing and fine-tuning the model and  $k$ -fold cross-validation, can you beat [the 0.14765 baseline](#) achieved by Random Forest regressor (a powerful model) on Kaggle? You may start by getting some ideas after reading a few previous chapters, such as
- [Multilayer perceptrons in gluon](#)
- [Overfitting and regularization \(with gluon\)](#)
- [Dropout regularization with gluon](#)

For whinges or inquiries, [open an issue on GitHub](#).

## Run these tutorials

Each tutorial is made from a Jupyter notebook, which is editable and runnable. Assume `python` is already installed, then in addition, both `jupyter` and a recent version of `mxnet` are required. The following commands install them through `pip`:

```
# optional: update pip to the newest version
sudo pip install --upgrade pip
# install jupyter
pip install jupyter --user
# install the nightly built mxnet
pip install mxnet --pre --user
```

The default `MXNet` package only supports CPU while some tutorials may need GPUs. If GPU is available and either CUDA 7.5 or 8.0 is installed, then we can install the GPU-supported package

```
pip install mxnet-cu75 --pre --user # for CUDA 7.5
pip install mxnet-cu80 --pre --user # for CUDA 8.0
```

Now we are ready to obtain the source codes and run them

```
git clone https://github.com/zackchase/mxnet-the-straight-dope/
cd mxnet-the-straight-dope
jupyter notebook
```

The last command starts the jupyter notebook, and now you can edit and run these tutorials now.

# How to contribute

For whinges and inquiries, please open [an issue at github](#).

To contribute codes, please follow the following guidelines:

1. Check the [roadmap](#) before creating a new tutorial.
2. Only cover a single new concept on a tutorial, and explain it in detail. Do not assume readers will know it before.
3. Make both words and codes as simple as possible. Each tutorial should take no more than 20 minutes to read
4. Do not submit large files, such as dataset or images, to the repo. You can upload them to a different repo and cross reference it. For example

- Insert an image:

```

```

- Download a dataset if not exists in local:

```
mx.test_utils.download('https://raw.githubusercontent.com/dmlc/web-data/master/mxnet/ptb/ptb.train.txt')
```

5. Resize the images to proper sizes. Large size images look fine in notebook, but they may be ugly in the HTML or PDF format.
6. Either restart and evaluate all code blocks or clean all outputs before submitting
  - For the former, you can click `Kernel -> Restart & Run All` in the Jupyter notebook menu.
  - For the latter, use `Kernel -> Restart & Clear Output`. Then our Jenkins server will evaluate this notebook when building the documents. It is recommended because it can be used as a unit test. But only do it if this notebook is fast to run (e.g. less than 5 minutes) and does not require GPU.
7. (Update, this feature is not available for Jupyter now.) If you want to reference a function or class, use [sphinx domains](#). For example
  - function: `:func:`mxnet.ndarray.zeros`` to `mxnet.ndarray.zeros()`

- class `:class:`mxnet.gluon.Parameter`` to `mxnet.gluon.Parameter`
- also works for numpy: `:func:`numpy.zeros`` to `numpy.zeros()`

8. You can build the documents locally to preview the changes. Assume `conda` is available, then following commands create an environment with all requirements installed:

```
# assume at the root directory of this project
conda env create -f environment.yml
source activate gluon_docs
```

Now you are able to build the HTMLs:

```
make html
```

If latex is installed, you can also build the PDF version:

```
make latex
make -C _build/latex
```