Neural Networks in a Nutshell

Rob Napora
JHU / FNAL / CDF

Feb. 10, 2001
Run II Advanced Analysis Mini-Workshop

The Plan

Introduction

The Basic Neuron (One Layer Networks)

Structure of Multilayer Feed Forward Networks

- Advantages in dividing parameter space (over basic neuron)
- How many layers do we need?

Training Multilayer Networks

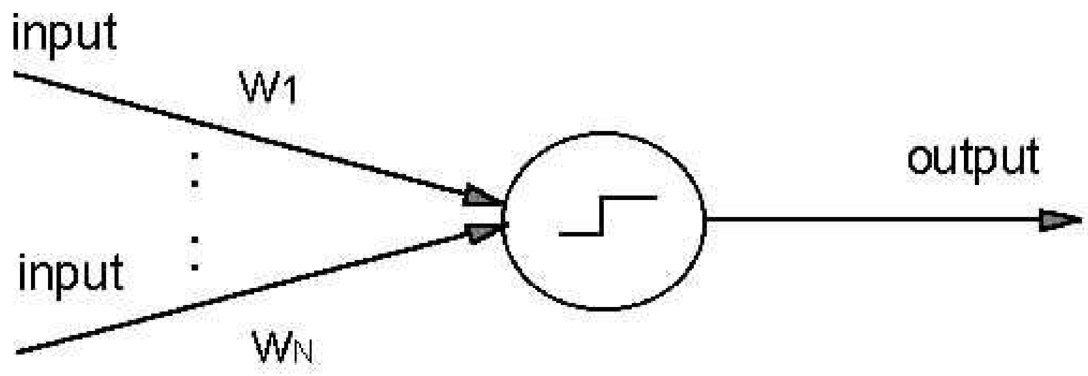
- Back-propogation Method
- Other options and tweaks

Other things to consider

- Over-training
- Too much freedom
- How we preprocess the data
- Output representation
- Importance of Testing

The Basic Neuron

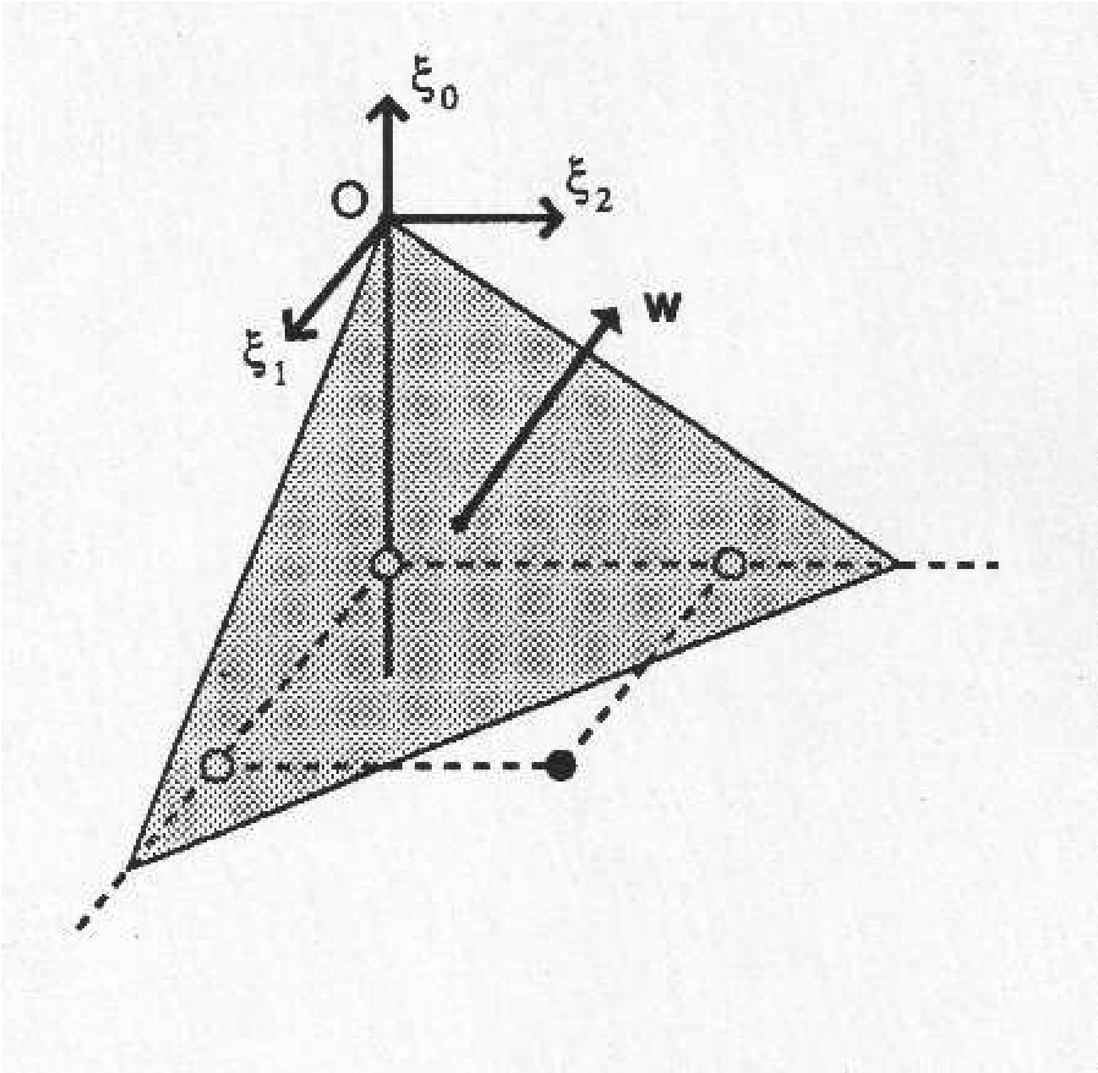
The **Neuron** (a.k.a. *unit*, a.k.a *node*, a.k.a. *simple perceptron*) is the basic building block for neural networks.



It takes the weighted sum of the inputs and compares them to a threshold value. If they are greater than the threshold the neuron 'fires' by giving an output of 1, if they are less than one the neuron does not fire and gives an output of 0.

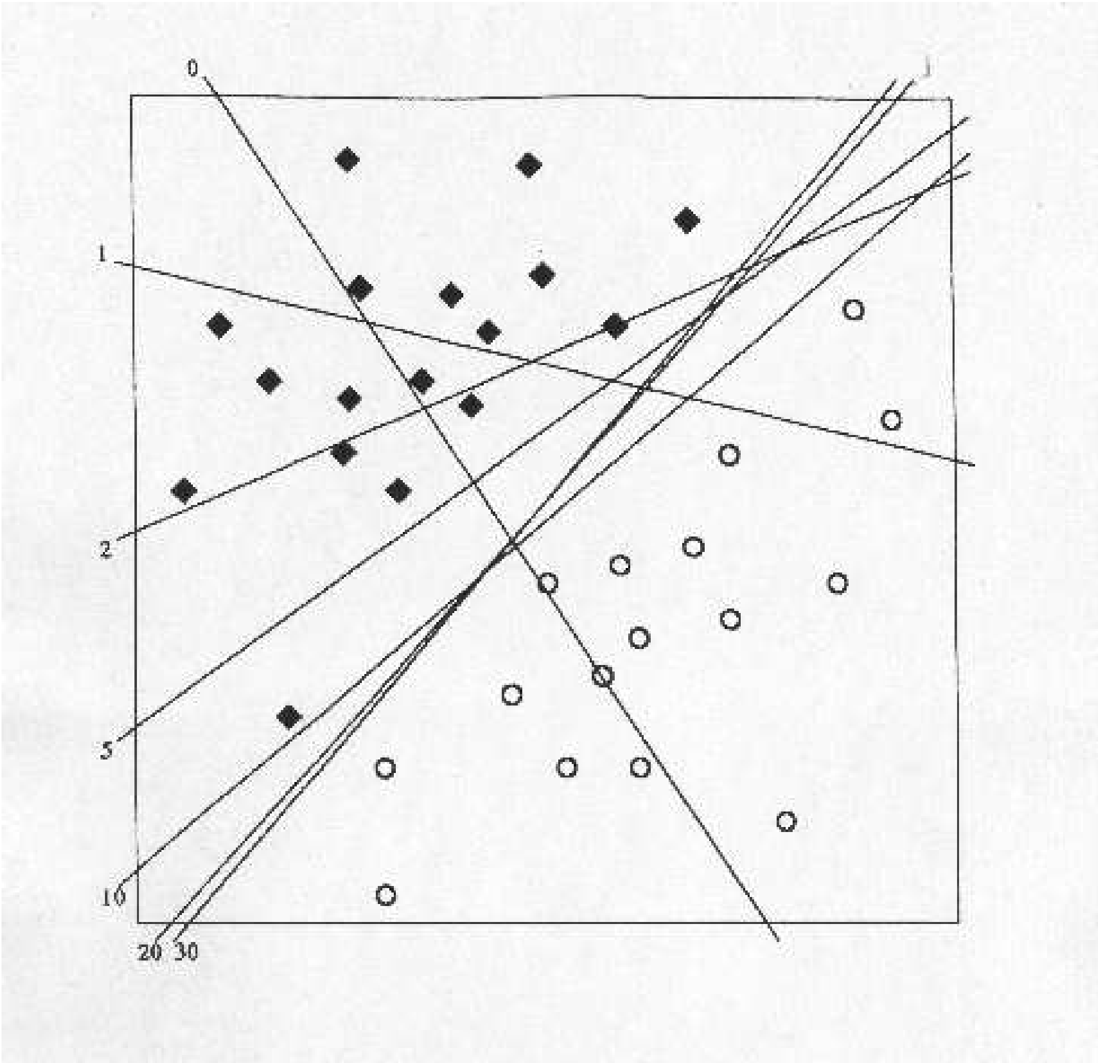
$$Output = \Theta(\sum_i w_i x_i - b)$$

A single neuron of this type effectively defines a cut in the parameter space. The weights determine the orientation of the plane and the threshold determines the offset from the origin.

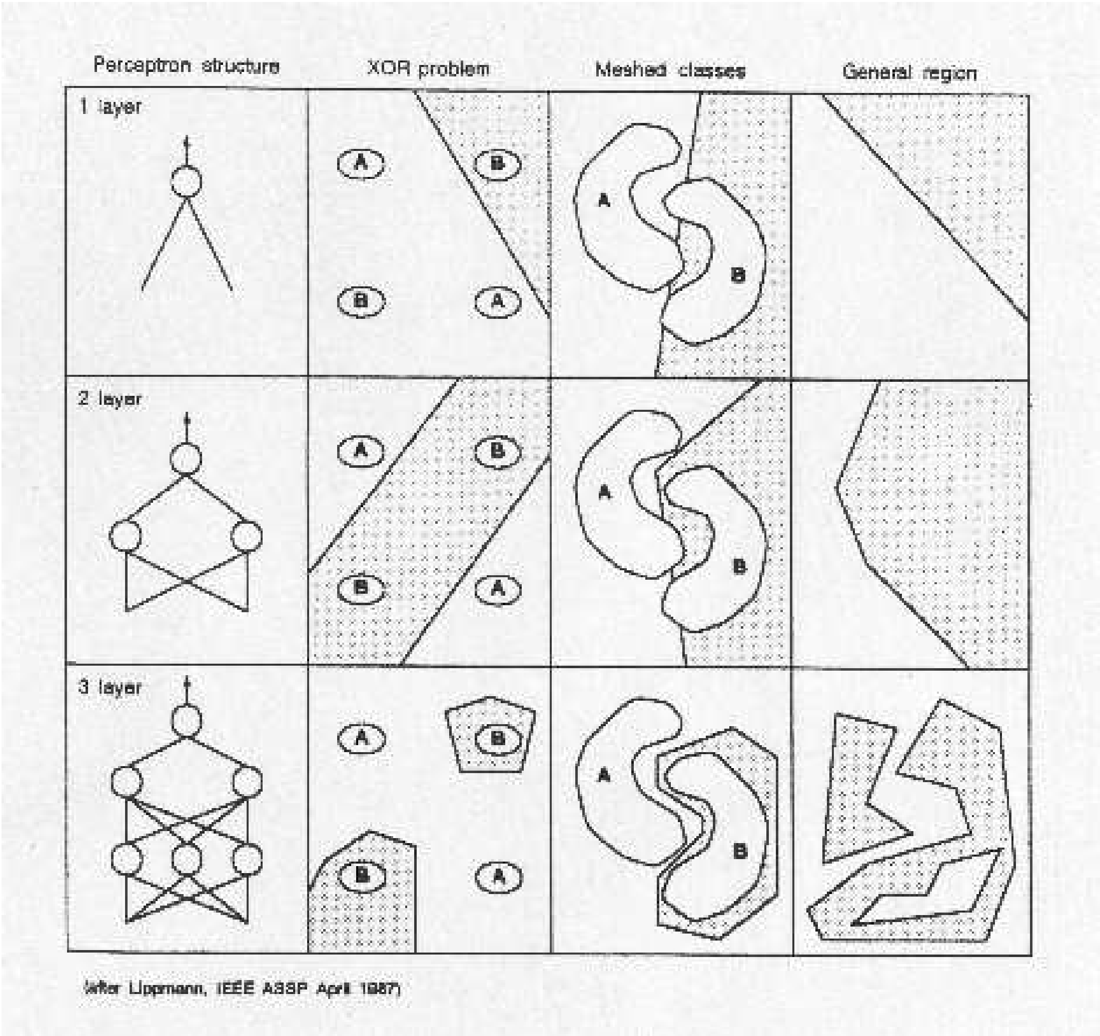


(Reference [2], page 95)

Training a Basic Neuron



Evolution of the classification line from an initial, random orientation into one that successfully classifies the two classes. (Reference [1], page 53)



(Reference [1], page 87)

How Many Layers Do We Need?

If we are trying to approximate a particular set of functions $F_i[x_k]$ we need *at most two* hidden layers, with arbitrary accuracy being obtainable.

It has also been proven that *only one* hidden layer is needed to approximate any continuous function.

Sound too good to be true?

It is. These theorems only tell us that a solution exists.

- They tell us nothing about the number of nodes needed in these layers.
- They do not guarantee that training is always possible.

Training a Multilayer Network

While we have now gained some utility, we have also lost the ability to train. With multiple layers the hard limiting threshold functions remove information that is needed to learn. This is called the *credit assignment problem*.

To fix this we change our step functions to continuous and differentiable functions. This allows us to construct an **error** (a.k.a. *cost*, a.k.a. *energy*) function which measures the system's performance error as a differentiable function of the weights. We can then use an optimization technique to minimize the error.

The Squashing Functions

Sigmoid functions are generally used for these. Either [0,1] or [-1, +1] ranges can be used, with

$$g(h) = \frac{1}{1 + exp(-2\beta h)}$$

and

$$g(h) = tanh(\beta h)$$

respectively. These functions have the "nice" properties that:

- They are well understood
- They saturate at both extremes
- The derivatives of these functions can be expressed in terms of themselves [$g'(h) = 2\beta g(1-g)$ and $g'(h) = \beta(1-g^2)$, respectively] which saves on computational time.

A sample error function

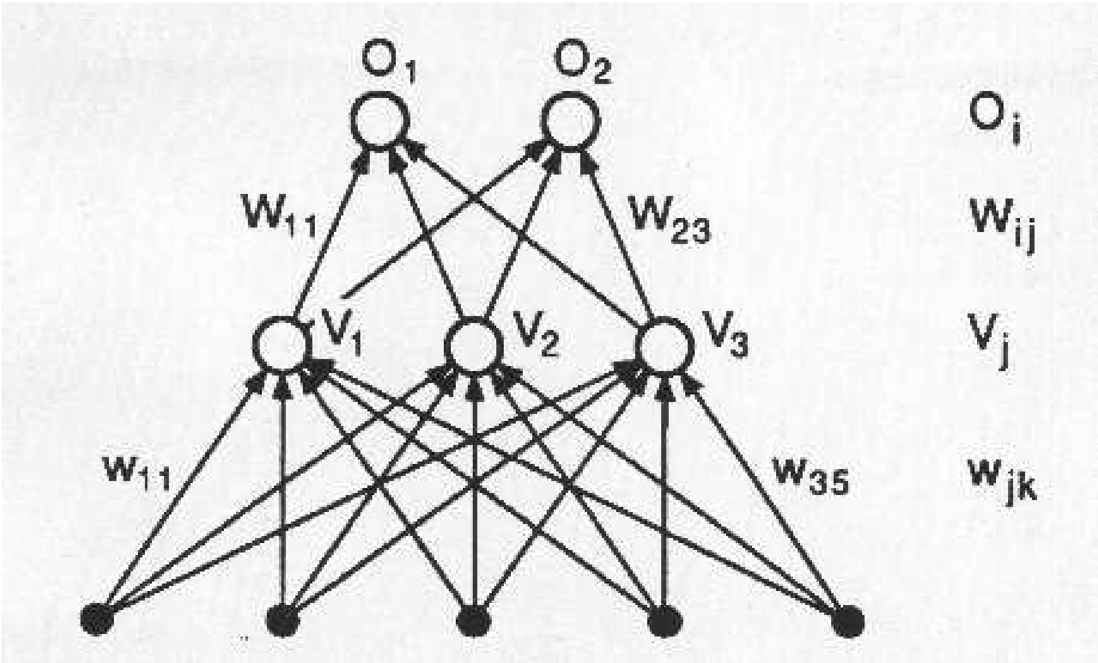
As an example we will define our error function as:

$$E\{w\} = \frac{1}{2}\sum_{\mu,i}[\xi_i^\mu - O_i^\mu]^2$$

Where: ξ is the expected output
O is the actual output
μ is the index over all input patterns
i is the index over all output nodes

If we have a single hidden layer this function written in terms of the weights becomes:

$$E\{W,w\} = \frac{1}{2}\sum_{\mu,i}[\xi_i^\mu - g(\sum_j W_{ij} g(\sum_k w_{jk} x_k^\mu))]^2$$



Now we can use a standard gradient descent method to update the weights.

For the hidden-to-output connections the update to the weights is

$$\Delta W_{ij} = -\frac{\partial E}{\partial W_{ij}}$$

$$= \sum_\mu [\xi_i^\mu - O_i^\mu] g'(\sum_j W_{ij} g(\sum_k w_{jk} x_k^\mu)) V_j^\mu$$

$$= \sum_\mu \delta_i^\mu V_j^\mu$$

Where

$$\delta_i^\mu \equiv g'(\sum_j W_{ij} g(\sum_k w_{jk} x_k^\mu))[\xi_i^\mu - O_i^\mu]$$

For the input-to-hidden connections the update to the weights is

$$\Delta w_{jk} = -\frac{\partial E}{\partial w_{ij}}$$

$$= \sum_{\mu,i} [\xi_i^\mu - O_i^\mu] g'(\sum_j W_{ij} g(\sum_k w_{jk} x_k^\mu)) \times W_{ij} g'(\sum_k w_{jk} x_k^\mu) x_k^\mu$$

$$\Delta w_{jk} = \sum_{\mu,i} \delta_i^\mu W_{ij} g'(\sum_k w_{jk} x_k^\mu) x_k^\mu$$

$$\Delta w_{jk} = \sum_\mu \delta_j^\mu x_k^\mu$$

Where

$$\delta_j^\mu \equiv g'(\sum_k w_{jk} x_k^\mu) \sum_i \delta_i^\mu W_{ij}$$

In general the back-propogation update rule always has the form

$$\Delta w_{jk} = \sum_{patterns} \delta_{output} * V_{input}$$

where *output* and *input* refer only to that specific layer

- The update rule is *local* allowing truly parallel computation.
- If we had n connections then calculating the n derivatives directly would take of order n^2 operations. Back-propogation takes only of order n operations to do the same.

Other Options and Tweaks

- We can add a gain term, η, to our weight adjustments.
- We can add a momentum term, α, to our weight adjustments.

i.e.,

$$\Delta w \to \Delta w(t)' = \eta \Delta w(t) + \alpha \Delta w'(t-1)$$

- We can use alternative Error Functions (explicitly written in terms of purity and efficiency?)
- We can use methods other than Gradient Descent to find the minima in the Error Function
- We can explicitly add noise to our system
- We can *snip* dead weights during training

Things To Watch Out For

- **Over Training**
  You may think that by going through as many training cycles (epochs) as possible you are improving your network performance. It is possible to "over-fit" your data, however, so that additional training actually degrades your network performance.
- **Too Much Freedom**
  More is not always better when deciding how many nodes/layers/inputs to use. If you give your network too much leeway it may find correlations *specific to your training data*, and therefore not applicable to the data you want your network to analyze.

Both of these pitfalls stress the necessity to **test** our network with data independent of our training set.

Preprocessing

We should take care in how we preprocess our data

Consider the two example predicates

- N is odd
- N has an odd number of prime factors

If we wanted a neural network to select on one of these our input representation would clearly make a significant impact on its speed and accuracy.

A **Silly Theorem**:
"Learning will always succeed, given the right preprocessor."

References

1. *Neural Computing an Introduction*, R. Beale and T. Jackson, Institute of Physics Publishing, Bristol and Philedalphia, 1990.
2. *Introduction to the Theory of Neural Computation*, John Hertz, Anders Krogh, and Richard G. Palmer, Perseus Books, Cambridge, Massachusetts, 1991.
3. *Neural Network Tutorial for High Energy Physicists*, Bruce Denby, Fermilab-Conf 90/94, May 1990.
4. *Tutorial on Neural Network Applications in high Energy Physics: A 1992 Perspective*, Bruce Denby, Fermilab-Conf 92/121-E, Jan 1992.
5. *Neural Networks for Triggering*, B. Denby, M. Campbell, F. Bedeschi, N. Chriss, C. Bowers, F. Nesti, Fermilab-Conf 90/20, Jan 1990.