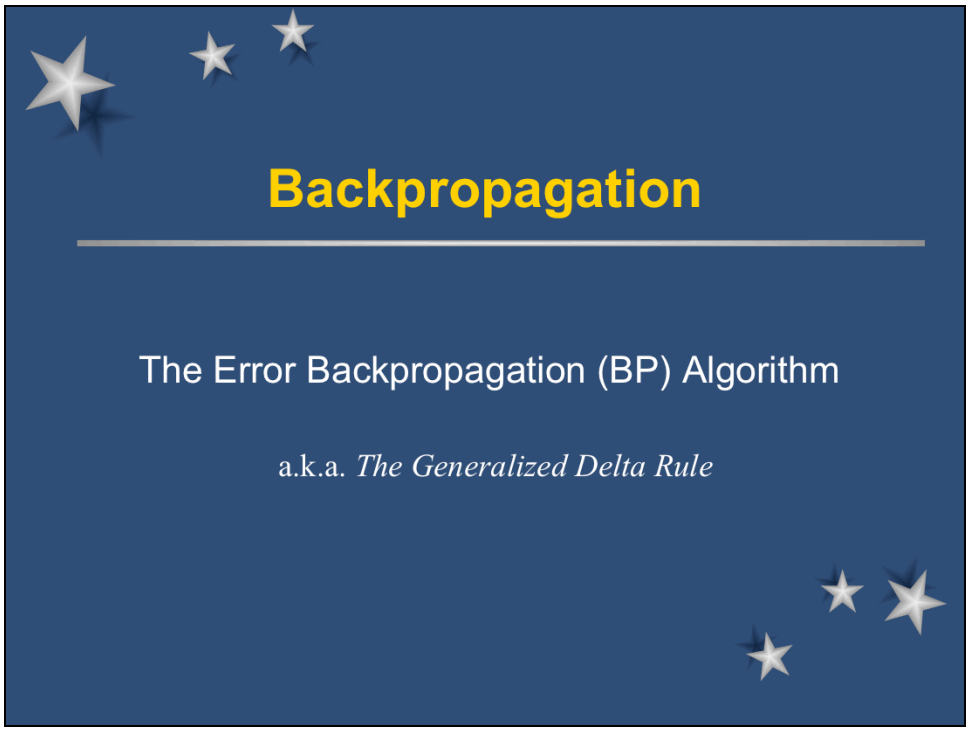
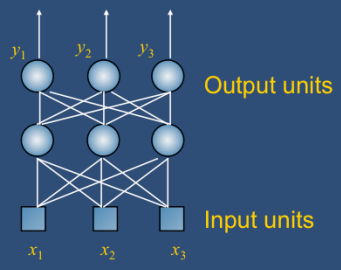

Backpropagation
The Error Backpropagation (BP) Algorithm
a.k.a. The Generalized Delta Rule

# Contents

- How to apply BP
- Theory
- Practical tips
- Historical notes

## Introduction to BP

- Popular learning technique for neural networks
- Used for training multi-layer perceptrons
- Learning based upon
  - Mean squared error
  - Gradient descent

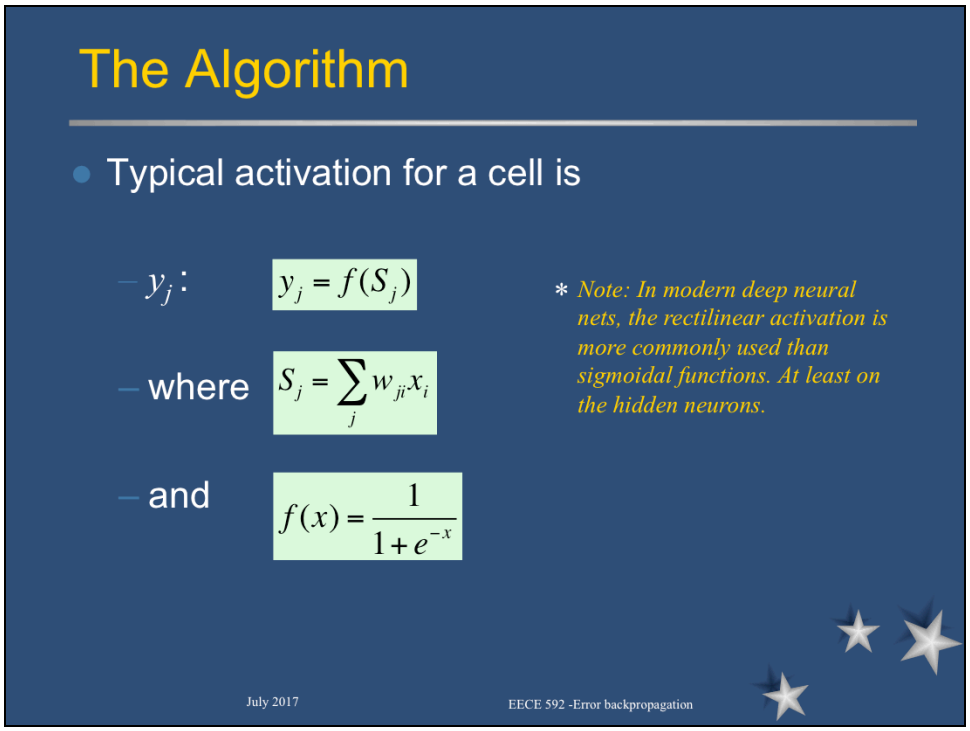





The error-backpropagation algorithm is one of the most important and widely used (and some would say wildly used) learning techniques for neural networks. First we will look at the algorithm itself and how it can be put to use. Then we'll look at the theory behind the algorithm and finally examples of backpropagation for some practical applications.

Backpropagation is used almost exclusively with feed forward, multi-layer perceptrons using continuous valued cells. Learning takes place based upon mean squared error and gradient descent. Backpropagation makes its easy to find the networks' error weight gradient for a given pattern.

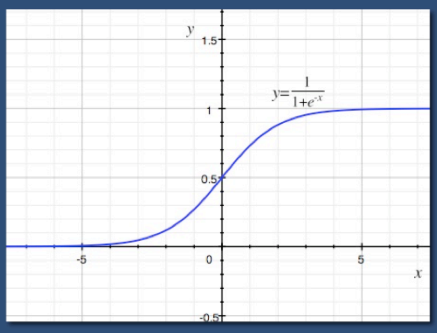
In Fausett, $x$ is used to label inputs, $y$ for output neurons and $z$ for hidden units.




The activation or threshold function acts upon the weighted sum of the neuron and is used to compute the output of the neuron. In backpropagation, the output is computed as shown above.

## Activation *f(x)* cont.

- The activation function is sigmoidal in shape



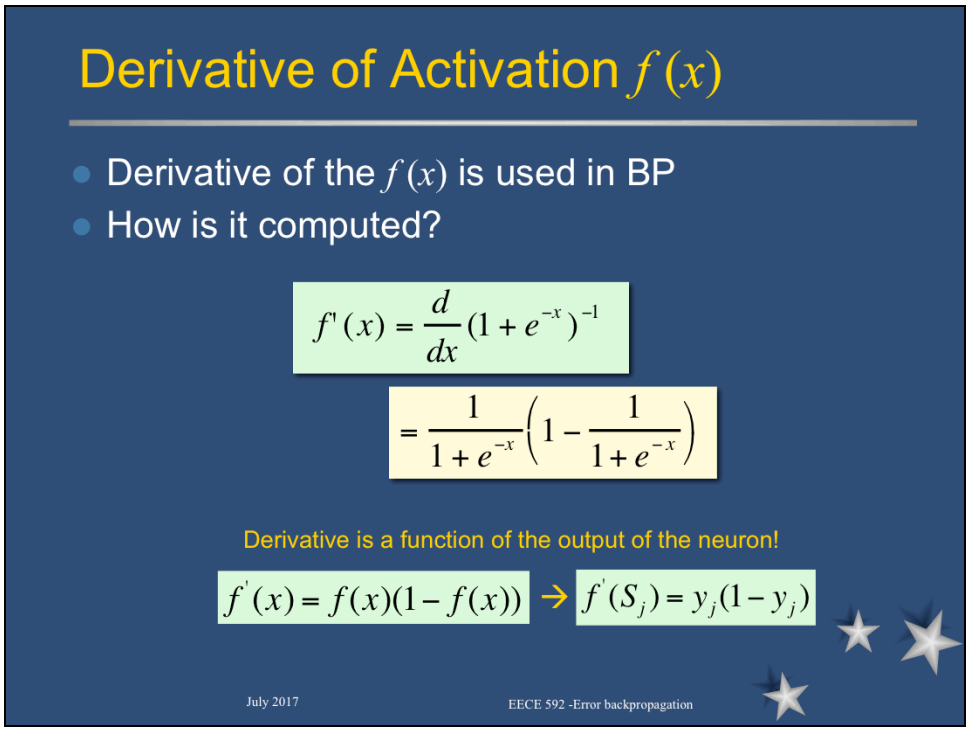



The activation function defined by $f(x) = \frac{1}{1+e^{-x}}$

Is *sigmoidal* in shape. It has asymptotes at 1.0 and 0.0. Also it returns a value of 0.5 when x=0.

The important point to note for the backpropagation algorithm is the function is continuous and has a derivative. This derivative has some interesting properties too! (See next slide)

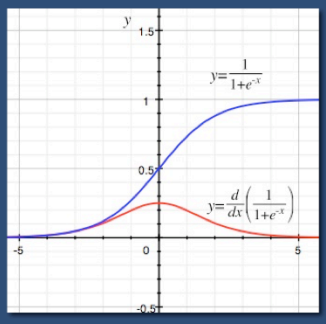
Note that actually, any sigmoidal activation function can be used. The asymptotes do not have to be at 1 or 0. Though attempting to stretch the range can lead to steep slopes which should be avoided.




Since backpropagation requires that we determine the derivative of the weight error, it is handy to find the derivative of the $f(x)$. It turns out that the derivative of the function can be expressed in terms of the function itself as shown above!

## Derivative of Activation *f(x)*

- What does it look like?






The important thing to note here is that the derivative is practically zero for large negative or large positive values of *x*. This, as we will see later, has a significant implication for speed of learning in backpropagation.

## Aim of the Algorithm

- Aim of the BP algorithm is to minimize the total error:

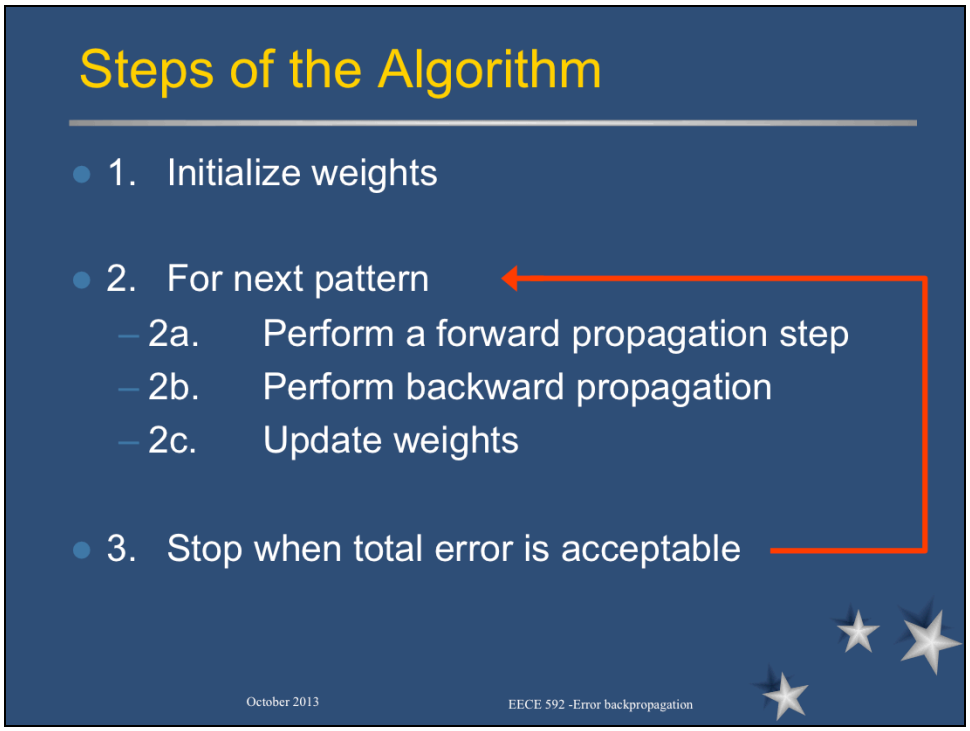
$$E = \sum_p E^p = \frac{1}{2}\sum_p (y^p - C^p)^2$$

  - Where
    - $y^p$ - actual output of pattern $p$
    - $C^p$ - required output for the pattern $p$
- Total error computed via forward propagation.
- Weights updated via backward propagation.

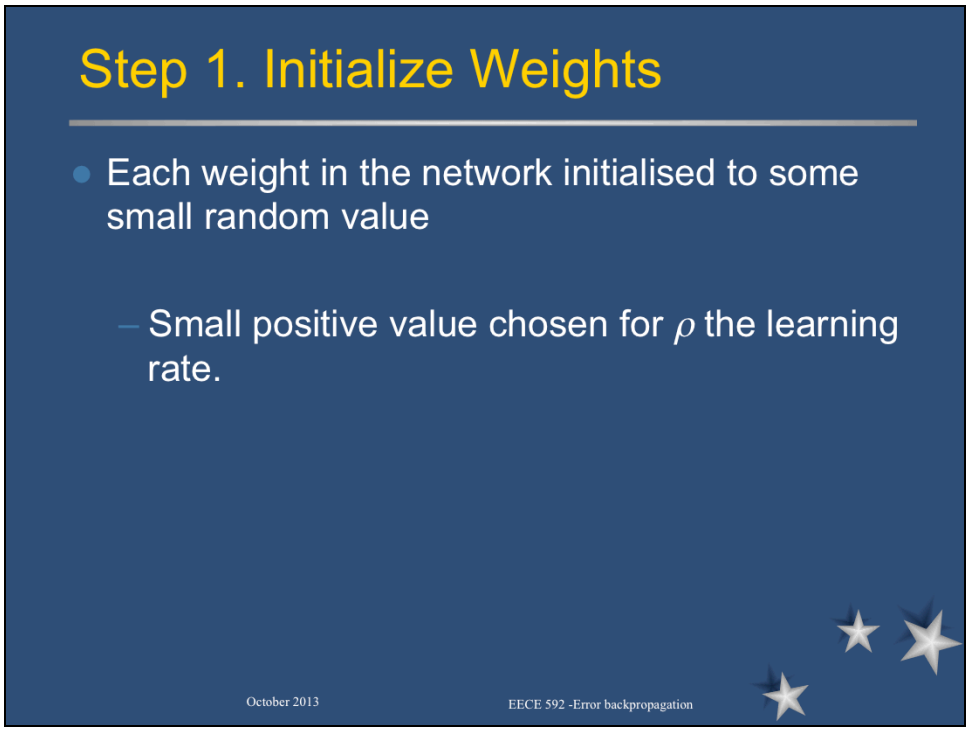
July 2017 EECE 592 -Error backpropagation

The total error is the sum of errors (squared difference between desired and actual outputs) generated by all patterns in the training set. The above equation describes the summation for single variable outputs. In the more general case, where the output if a vector of dimensionality $h$, the total error is defined as:

$$E = \sum_p E^p = \frac{1}{2}\sum_h \sum_p (y_h^p - C_h^p)^2$$




How "acceptable" the total error should be is dependent on how the network is to be used. For example, if the usage is such that the outputs represent binary values, e.g. to decide whether a request for a loan should be granted or not. Then it is valid to accept values of >0.5 as 1 and <0.5 as 0. If on the other hand the target values represent real numbers such as a speed or direction, then the total error must be "tighter". In this case you might find it useful to determine how the total error E, relates to the average error in the predicted out of the network. One way to do this is to convert E into an RMS (root mean square).

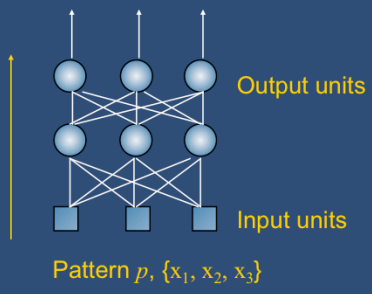



**Step 1**. Weight Initialisation

Chose a small positive value for $\rho$, the learning rate and initialize the weights ($w_{ji}$) of the network to small random values in the range [-0.1, 0.1] say. Motivation for weight initialisation is to scale them close to their final target values. This should keep training cycles short. If there is a large variation in magnitude between the final and initial weight values, the algorithm will take much longer to reach an acceptable total error level.

## Step 2. Process Training Sample

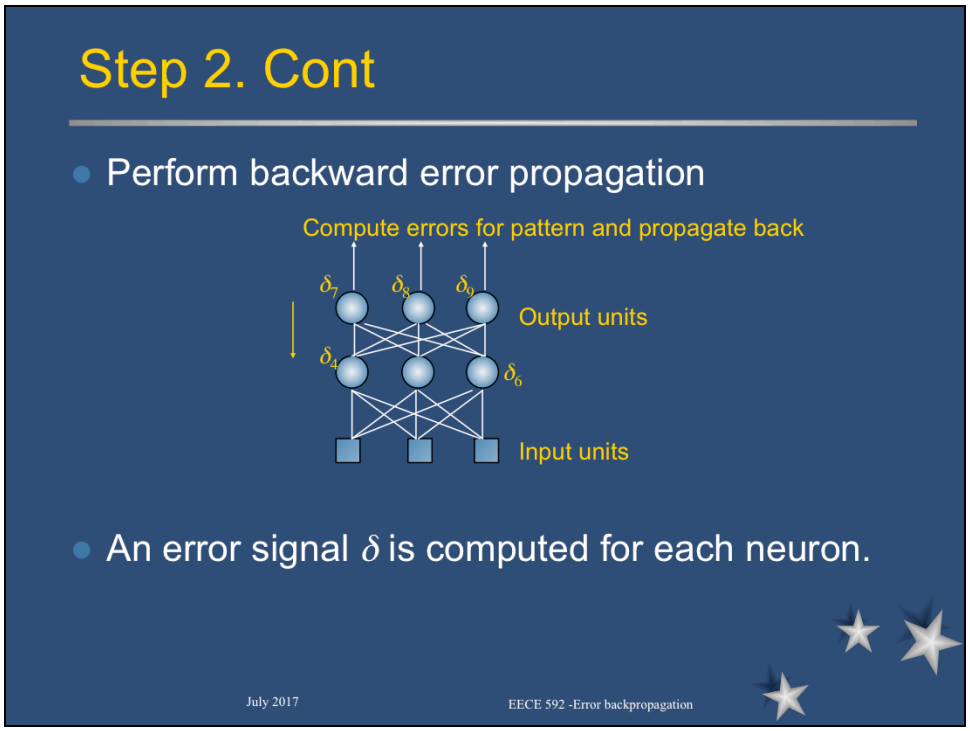
- For each training pattern
  - Perform forward propagation.






**Step 2a** - Forward propagation step

Make a bottom up pass through the net computing weighted sums $S_i$ and activations $y_i$ for all cells.




**Step 2b** - Backward error propagation step

Beginning with the outputs, evaluate errors and propagate them down through the weights as follows:

Compute

$$f'(S_i) = y_i(1 - y_i)$$

If $y_i$ is an output unit

$$\delta_i = (C_i - y_i) f'(S_i)$$

If $y_i$ is a hidden unit

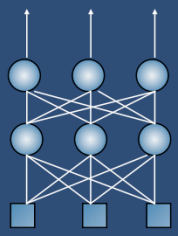
$$\delta_i = \sum_{h>i} w_{hi} \delta_h f'(S_i)$$

Note that for hidden layer units, the error signal, $\delta_i$, is the weighted sum of the errors at the units above.

## Step 2. Cont.

- Update weights

$$w_{ji}^{*} = w_{ji} + \rho \delta_j x_i$$



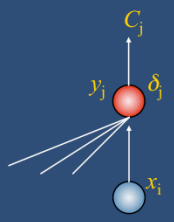


**Step 2c -** Weight update

The weights are updated typically immediately after the forward propagation and before the next pattern is presented. This is known as online updating and is a form of stochastic gradient descent. See later slides on batch versus online updating.

# Step2: compute δ

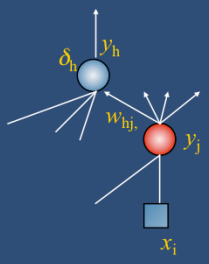
- If $y_j$ is an output unit.




$$\delta_j = y_j(1-y_j)(C_j - y_j)$$

# Step 2: compute δ

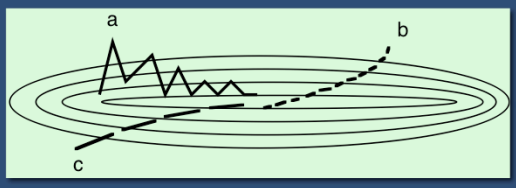
- If the unit is a hidden unit




$$\delta_j = y_j(1-y_j)\sum_h \delta_h w_{hj}$$

## Aside: Momentum

- Selection of learning rate can be difficult




  - a : learning with a large value for $\rho$
  - b : learning with a small value for $\rho$
  - c : learning using momentum and a large value for $\rho$



There is always a problem in deciding on the magnitude of $\rho$, too large and learning can oscillate, too small and convergence becomes very slow. The addition of a momentum term helps to smooth out the learning curve. The diagram is meant to show lines of equal error. You can see that in (a), taking large steps may cause you to oscillate in regions of higher error.

## Step 2. Momentum Term

- Weight update with momentum

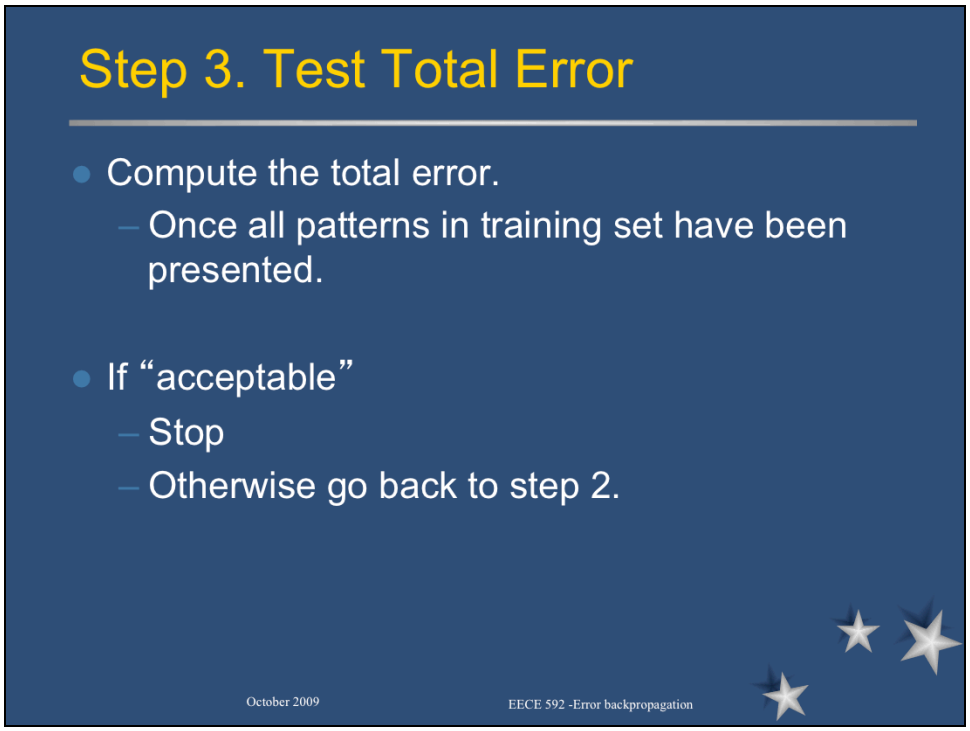
$$w_{ji}^{*} = w_{ji} + \alpha \Delta w_{ji} + \rho \delta_j x_i$$

  - Where
    - $\Delta w_{ji}$ is the previous weight change.
    - And $\alpha$ is the momentum term.



The addition of "momentum" to the weight update term is an easy way to reduce training times. As the name implies, the weight update is given some inertia to keep going in the direction of the previous update.

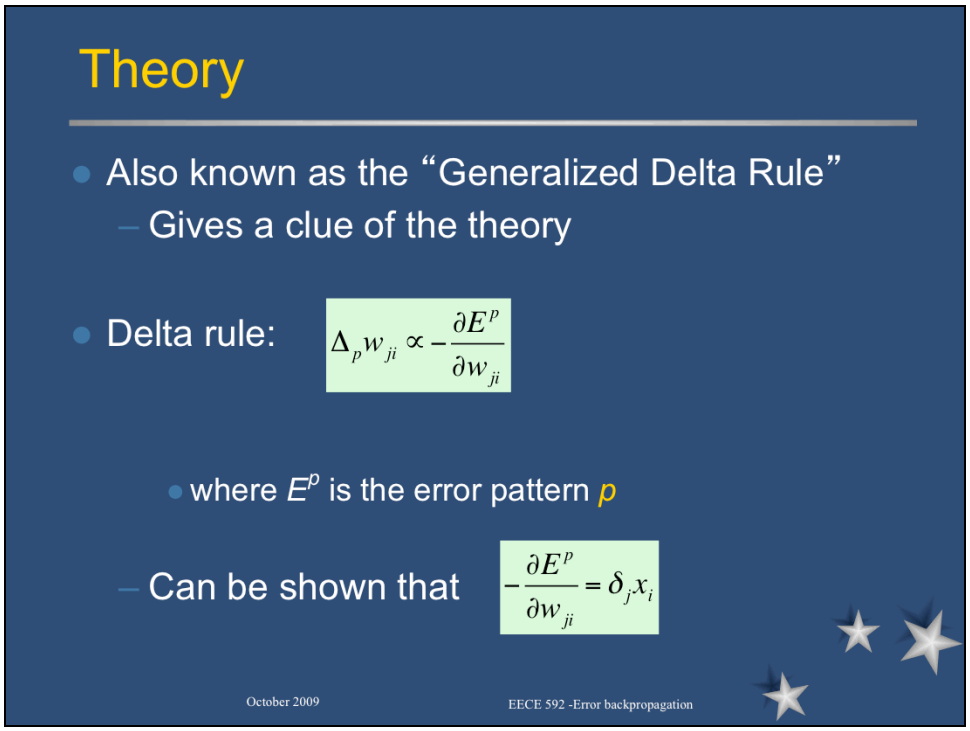
Momentum is easily implemented. The idea is that the weight change is pushed on by the previous weight change. Step 2c becomes as shown above. It does mean that your software implementation of backpropagation now needs to store both the current set of weights and a previous set of weights.




**3 -** Test total error

The total error is computed once all patterns in the training set have been presented. One presentation of a training set is known as an *epoch*. It's normal for many hundreds or thousands of epochs to pass before an acceptable total error is reached.

One question to ask yourself here, is what is an "acceptable" error?




Backpropagation is a generalization of the delta rule and is sometimes referred to as the "generalized delta rule".

We start with the delta rule as before: $\Delta_p w_{ji} \propto -\frac{\partial E^p}{\partial w_{ji}}$

where $E^p$ is the error pattern $p$

and $w_{ji}$ is the weight $i$th to neuron $j$, then using the chain rule,

$$\frac{\partial E^p}{\partial w_{ji}} = \frac{\partial E^p}{\partial S_j}\frac{\partial S_j}{\partial w_{ji}} \qquad \text{let} \qquad \frac{\partial E^p}{\partial S_j} = \delta_j$$

And

$$S_j = \sum_i w_{ji} x_i \qquad \text{so} \qquad \frac{\partial S_j}{\partial w_{ji}} = x_i$$

Leading to $-\frac{\partial E^p}{\partial w_{ji}} = \delta_j x_i$

The trick is to calculate the error signal $\delta_j$ for each unit $y_j$

## Theory - δ for output units

- Trick is to calculate the error signal, $\delta$ for each neuron:
  - Given

$$\delta_j = \frac{\partial E^p}{\partial S_j}$$

  - When the unit is an output neuron

$$\delta_j = (C_j - y_j) f'(S_j)$$



It turns out that there is a simple way in which $\delta_j$ can be computed and propagated back through the network.

$$\delta_j = \frac{\partial E^p}{\partial S_j}$$

Using the chain rule,

$$\delta_j = \frac{\partial E^p}{\partial y_j}\frac{\partial y_j}{\partial S_j}$$

If $y_j$ is an output unit

To calculate the second factor

$$y_j = f(S_j) \rightarrow \frac{\partial y_j}{\partial S_j} = f'(S_i)$$

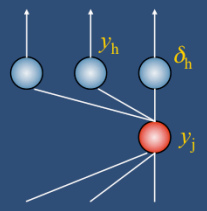
To calculate the first factor, $E^p = \frac{1}{2}\sum_j (y_j - C_j)^2 \rightarrow \frac{\partial E^p}{\partial y_j} = (y_j - C_j)$

Thus (folding in the –ve sign from the initial gradient descent starting point)

$$\delta_j = (C_j - y_j) f'(S_j)$$

## Theory - δ for hidden units

- If the unit is a hidden unit:




- then $\delta_j = f'(S_j)\sum_h \delta_h w_{hj}$

July 2017 EECE 592 -Error backpropagation

If $y_j$ is a hidden unit
Then using

$$\delta_j = f'(S_j)\sum_h \delta_h w_{hj}$$

we can define

$$\frac{\partial E^p}{\partial y_j}$$

in terms of the units of the layer above. Thus $\frac{\partial E^p}{\partial y_j} = \sum_h \frac{\partial E^p}{\partial S_h}\frac{\partial S_h}{\partial y_j}$

$$\frac{\partial E^p}{\partial y_j} = \sum_h \frac{\partial E^p}{\partial S_h}\frac{\partial}{\partial y_j}\sum_j w_{hj}x_j$$

and, as before,

$$\frac{\partial E^p}{\partial y_j} = \sum_h \frac{\partial E^p}{\partial S_h} w_{hj}$$

$\frac{\partial y_j}{\partial S_j} = f'(S_j)$ and $\frac{\partial E^p}{\partial S_h} = \delta_h$ *see previous slide*

so

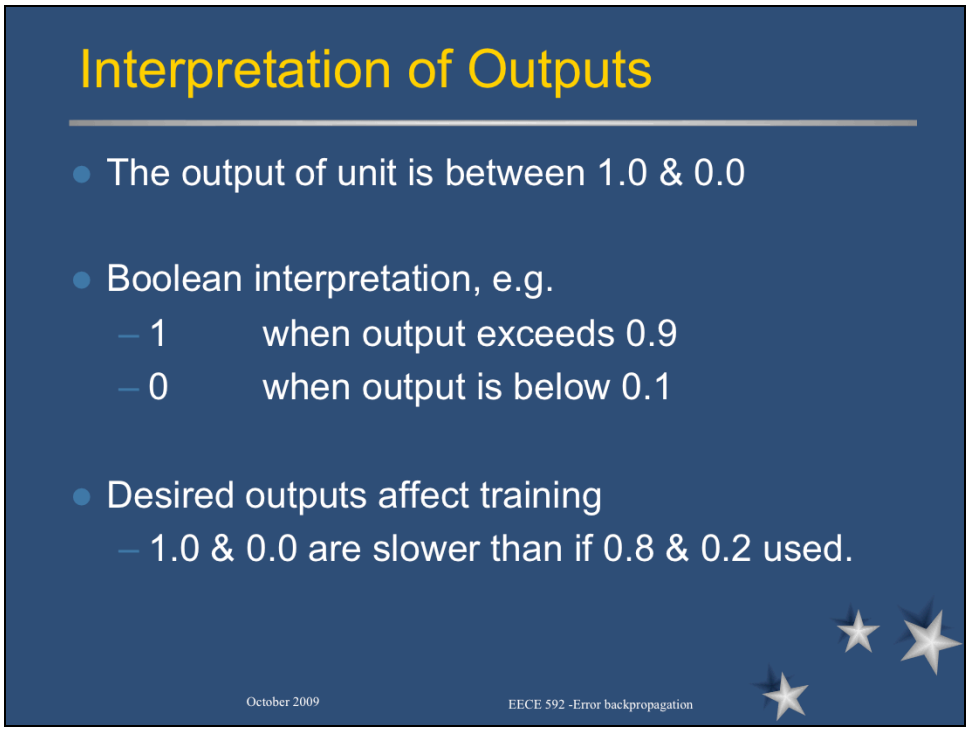
$$\delta_j = f'(S_j)\sum_h \delta_h w_{hj}$$

# Theory - Summary

- The weight change made in one learning step is

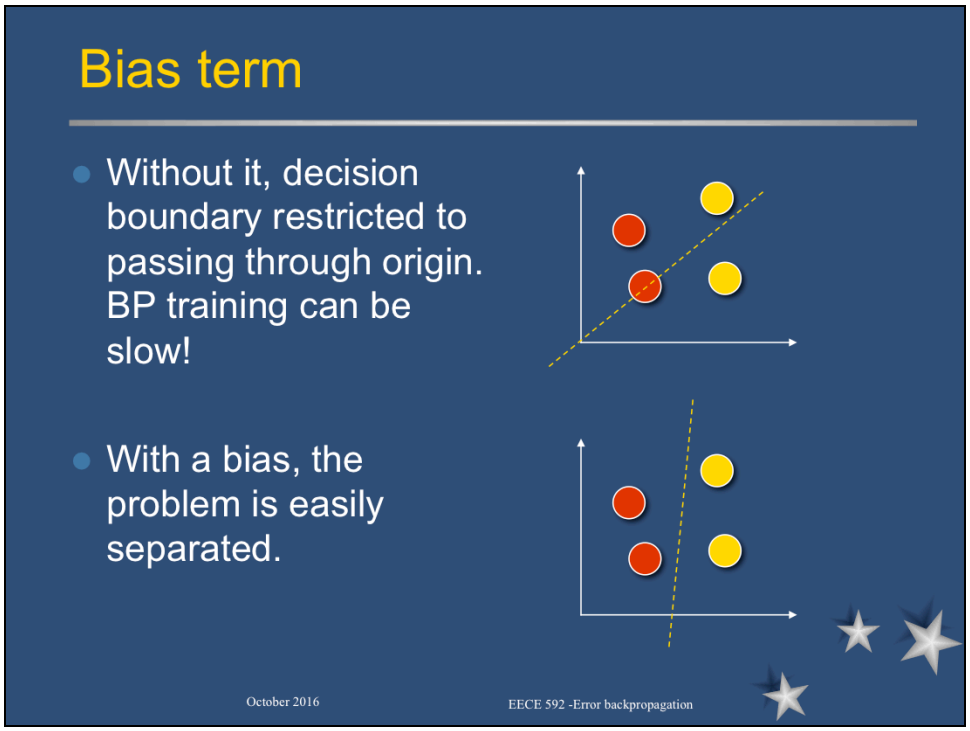
$$w_{ji}^{*} = w_{ji} + \rho \delta_j x_i$$

  - where $\delta_j$ is the error signal for neuron $j$
  - and $w_{ji}$ the $i$ th weight to neuron $j$




The *squashing* function guarantees that the output of a cell will be less than 1.0 and greater than 0.0. If the output is to be interpreted in a boolean sense, then arbitrary values must be assigned to 1 and 0. E.g. the output may be deemed to be a 1 if the output of the cell exceeds 0.9, or 0 if the output is below 0.1. Of course, the response could also be based on which side of 0.5, the output is.

To keep training fast, it might be better to use correct values $C^k$ as 0.2 and 0.8 instead of 0 and 1. This prevents from reaching flatter portions of the sigmoid where $f'(S_i)$ is close to zero and the weight change correspondingly smaller




When working with the BP algorithm, it is important to include a bias term. Without it, BP is restricted in the extent of the learning it can perform.

To explain its importance, consider a 2-dimensional categorisation problem as shown above. Without a bias term, backpropagation is restricted to finding a hyperplane that intersects the origin. In this case, one that correctly learns both categories of pattern cannot be found. Once a bias term is introduced, the problem is easily solved.

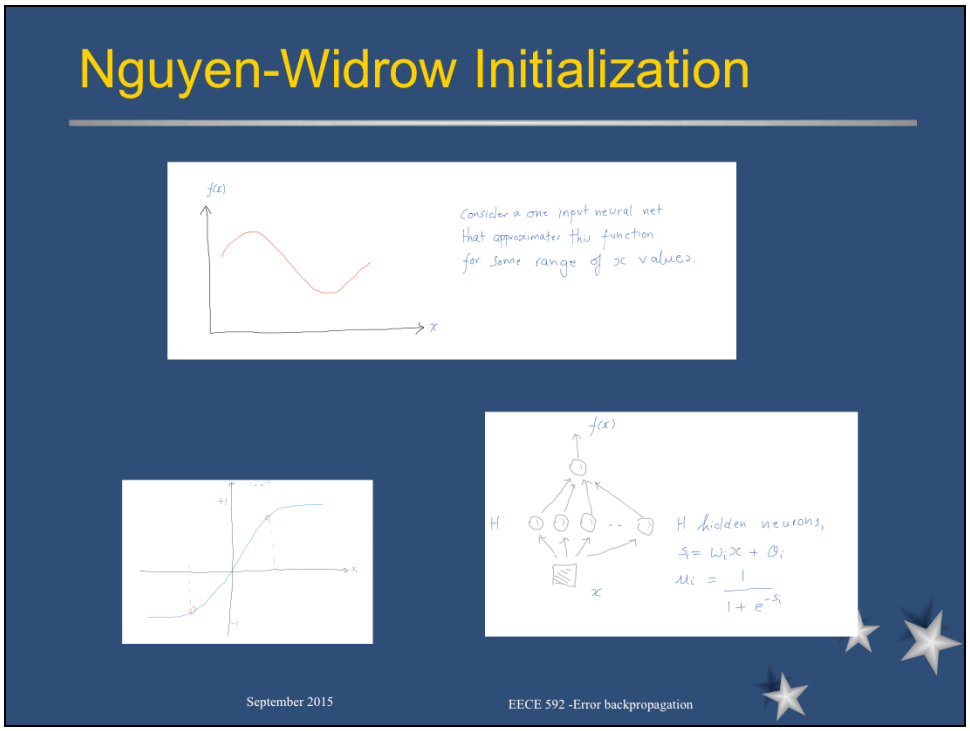
A bias term is simply a weight from an input that is always 1 and is usually built into the implementation of the algorithm. When training a single layer of neurons, we learned that you can add bias by simply changing your training patterns by adding an additional input of value 1, to each pattern. Unfortunately, this does not work for training in multiple layers. This is because the neurons in the other layers also need a bias.

## Initial Weights

- Weights should be small random values.
- Too large and $f'(Si)$ could be close to 0.



To break the initial symmetry, the weights of the net should be set to small random values before learning commences. If the values are too large, $f'(S_i)$ could be close to 0, slowing down learning. Gallant suggests if there are z inputs, the initial values should be in the range [-2/z, 2/z], but this is not critical. Note, you may also want to consider Nguyen-Widrow initialization procedure which is claimed to yield faster learning. (See page 297 of Fausett).




The general idea behind the Nguyen-Widrow initization procedure is simple and is typically described in terms of using a one input neural network to approximate a continuous function between some range of input values.

Given *H* hidden neurons, once trained we would like each neuron to approximate, in a piecewise fashion, a section of the curve. This can be done if the linear portion of the output of each neuron maps to some range of the input domain. Using this idea, Nguyen-Widrow extend it to apply to any number of input neurons.

# Nguyen-Widrow Initialization

1. Initialize weights as usual e.g.

$$w_{ji} \in [-0.5, +0.5]$$

2. For each hidden neuron $j$, set weights $w_{ji}$:

$$w_{ji}^{*} = \frac{\beta w_{ji}}{\|w_j\|}$$

where

$$\beta = 0.7\sqrt[d]{p}$$

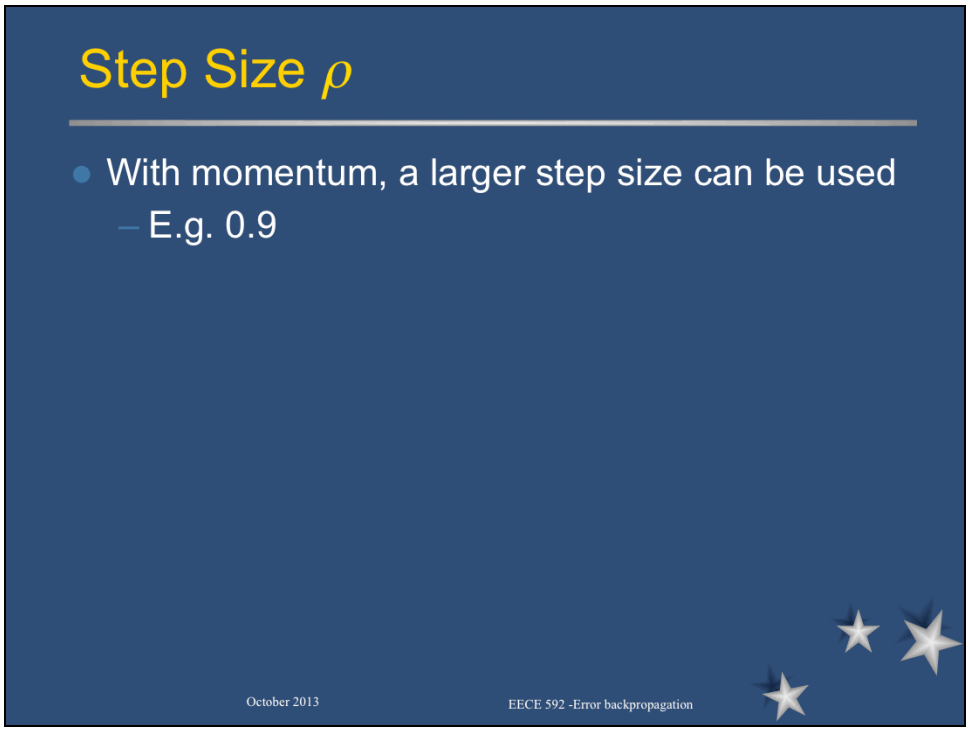
$d$ = # inputs

$p$ = # hidden



From page 297 Fausett.

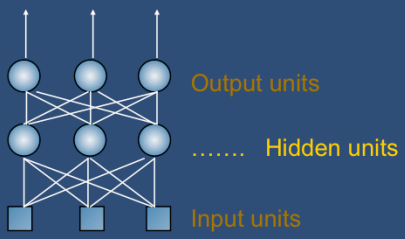
Note that Fausett suggests that only the weights of the hidden neurons are adjusted. However, other authors tend to apply Nguyen-Widrow initialization to all layers.




As mentioned before, small step sizes will invariably reduce the speed of convergence and too large could lead to oscillations as learning jumps around a minimum. The use of a momentum term allows higher values of step size to be used (e.g. 0.9).

## Generalization vs Number of Weights

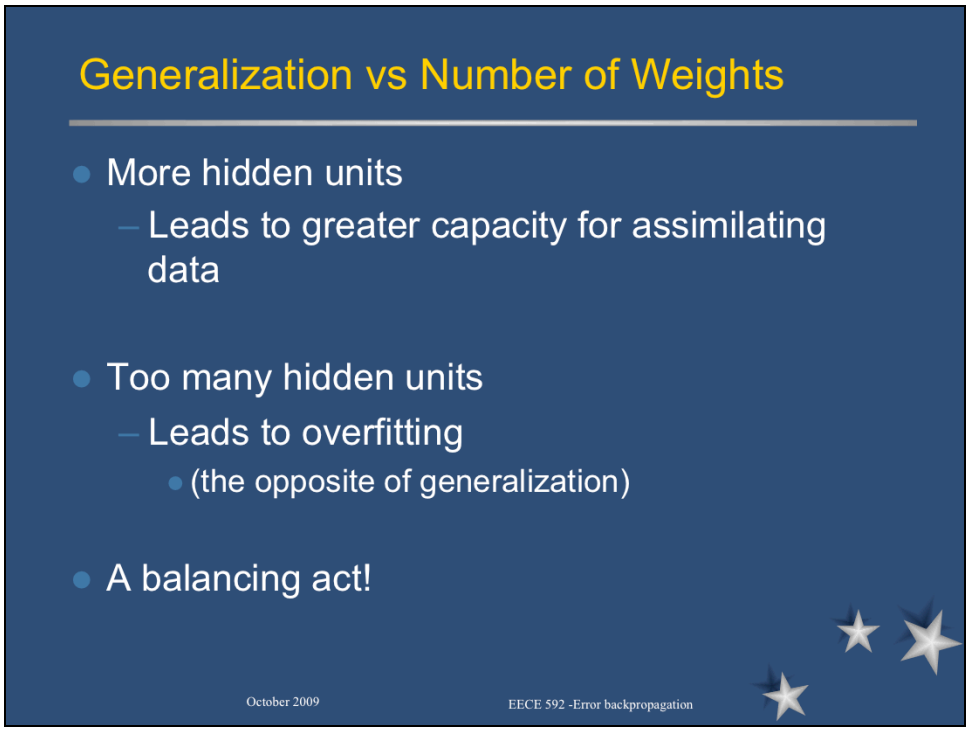
- How many units to use in the hidden layer?




- Selecting the optimum number is somewhat of a *black art* !



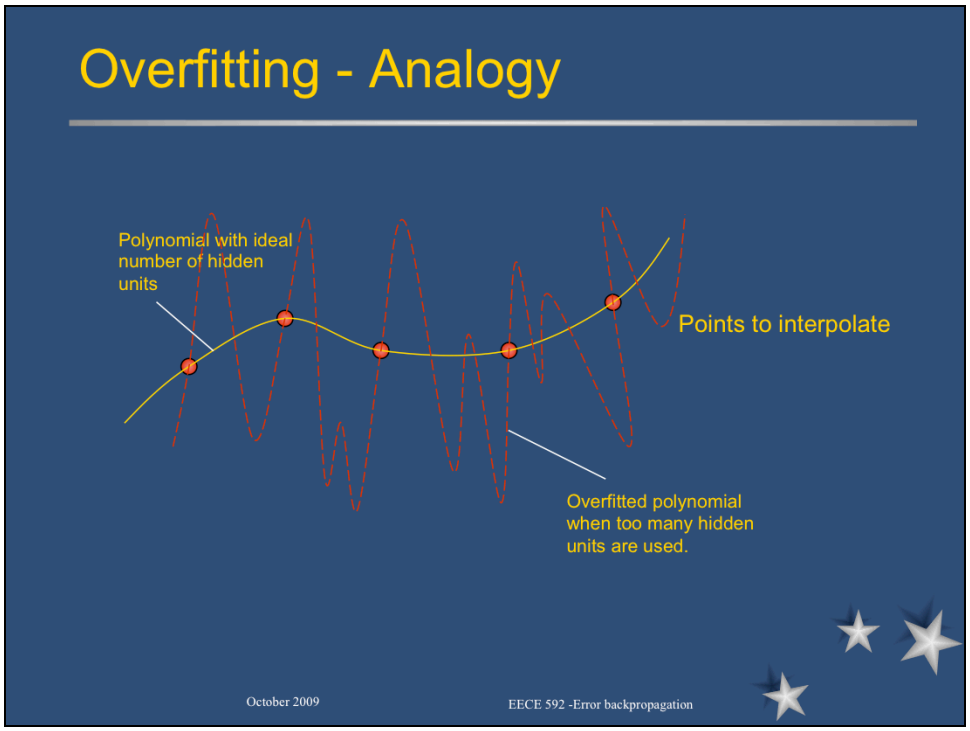
The number of output units and the number of input units is completely defined by the problem itself and the representation of the input/output data. However the number of hidden units is arbitrary. The hidden layer may contain as few or as many hidden units as desired.




The number of hidden units must be sufficient to accommodate learning of the problem, I.e. all the patterns in the training set.

However, the best generalization which is typically desirable, is achieved when the fewest possible units are used.

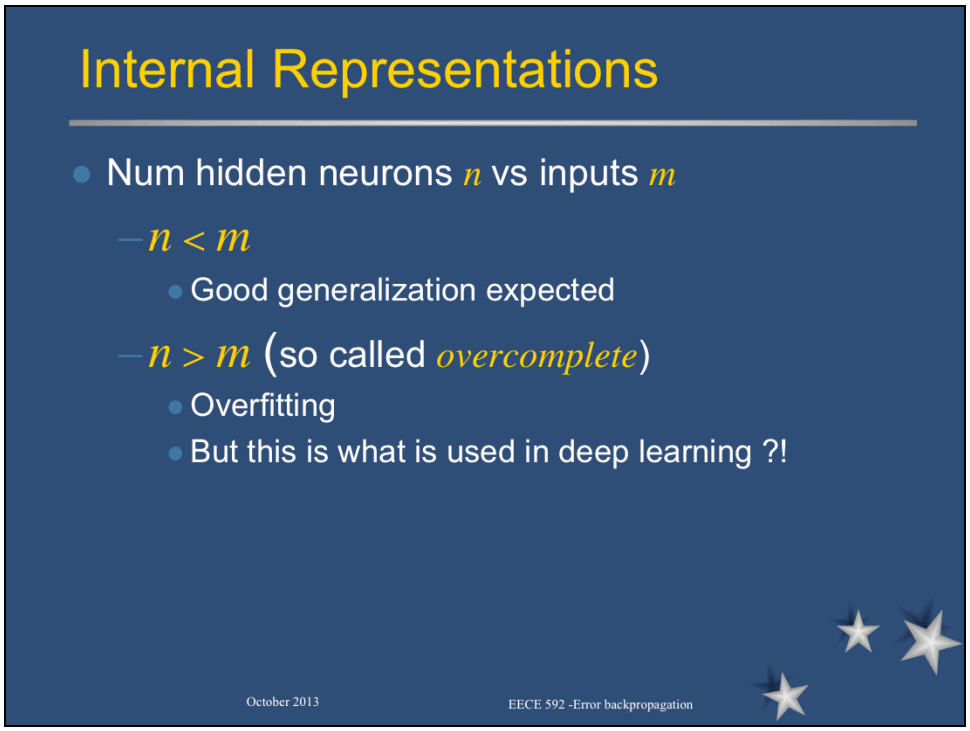
Selection of the actual amount of units is a balancing act! The network should be forced to learn with a limited, but just enough, capacity. This should lead to the network, learning the "general" concept rather than say "parrot-fashion".




As an analogy, consider using a neural net to interpolate a set of points.

When just the right amount of hidden units are used, the "general concept" is learned.

When too many hidden units are used, overfitting occurs and the result is a high-order polynomial that obviously has no bearing on the set of points being interpolated.




Recently, learning in deep neural networks uses backpropagation in which the number of hidden units in a layer may be large compared to the number of inputs. Does this not lead to overfitting?

In Deep Learning, backpropagation is used in an autoencoder configuration, for which useful internal representations result when used with large numbers of neurons. To prevent overfitting, techniques such as denoising and dropout are used. However, it may be that these only work for autoencoders and not the more general application of BP.

## Weight Decay

- Mechanism for minimizing # hidden units
  - Network will eliminate 'non-useful' connection during training.
  - Also called 'regularization'
- Weight update term includes:

$$w_{ji}^{*} = (1-\varepsilon)w_{ji}$$

  - Weights 'decay' unless continuously reinforced

July 2017 EECE 592 -Error backpropagation

Weight decay is a mechanism for dynamically keeping the number of hidden neurons to a minimum.

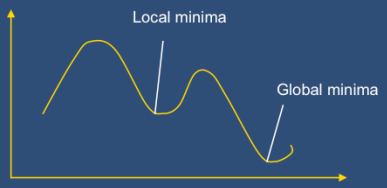
During training, each weight is decayed according to the following step:

$$w_{ji}^{*} = (1-\varepsilon)w_{ji}$$

Where ε is some small parameter between 0 and 1 and determines the magnitude of the decay. Those weights that are not continuously reinforced, will gradually decay to 0. When a weight is 0, it is effectively equivalent to disconnecting that input.

## Local Minima

- Theoretical deficiency of gradient descent.
- Generally difficult to identify a local minima.

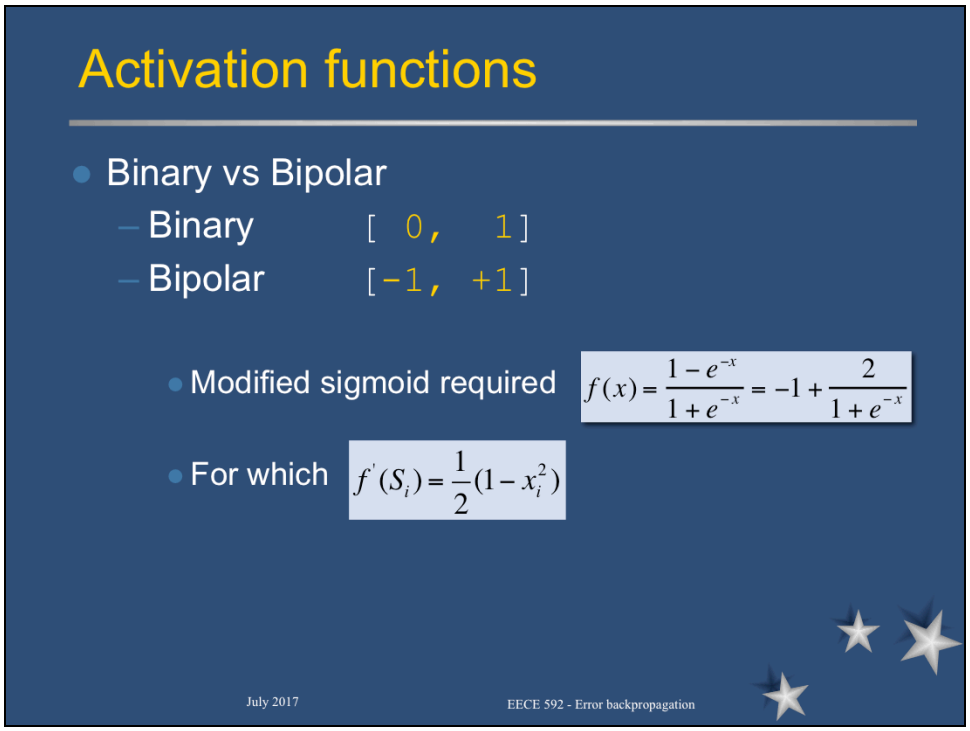



- In practice, infrequently encountered
  - Especially for problems where number of variables is high.



The aim of learning is to reach the lowest possible error minimum for the pattern being trained. This is called the global minima. However a known problem associated with gradient descent type learning is that of local minima. The diagram illustrates the situation. Generally it is difficult to know whether a minima is local or global!

In practice, local minima are not necessarily a problem, especially for problems where dimensionality of the input is high. This is because to be a real minima, the point must be a minima in all dimensions simultaneously. For example, a gutter is not a minima since it curves up only in cross-section and not along its length.




[0,1] vs [-1, +1] activations

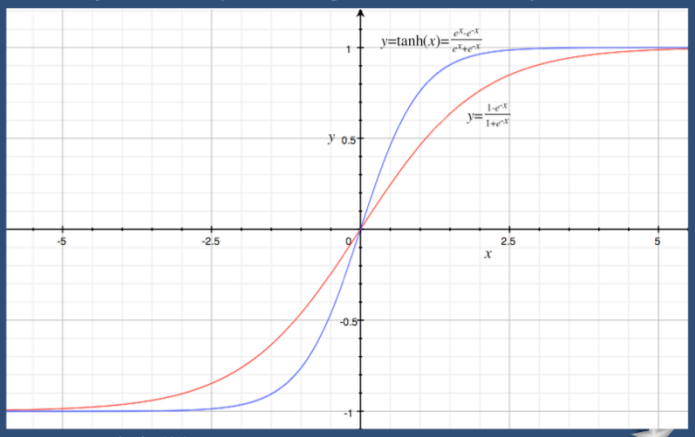
Earlier it was mentioned that [-1, +1] activation is preferred over the traditional [0,1] values. Here's why. Consider the update: $w_{ji}^{*} = w_{ji} + \rho\delta_j x_i$

Using [0,1], with a 0 input $x_i = 0$
thus the weight change is 0 and no learning takes place.
Using [-1, +1], with a -1 input $x_i$ =-1 and the weight change is non-zero.

Convergence will obviously be faster and this result can be verified experimentally. To use these [-1, +1] activations requires a simple modification to the sigmoid squashing function as shown above. In fact it is easy to change the activation function to cover any required range of values, not just -1 to +1.

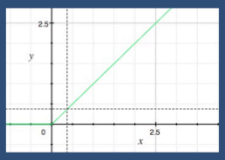
# Activation functions cont.

- *Tanh* activations claimed to exhibit faster convergence.
  - C. M. Bishop, *Neural Networks for Pattern Recognition*. 1995 Oxford Univeristy Press

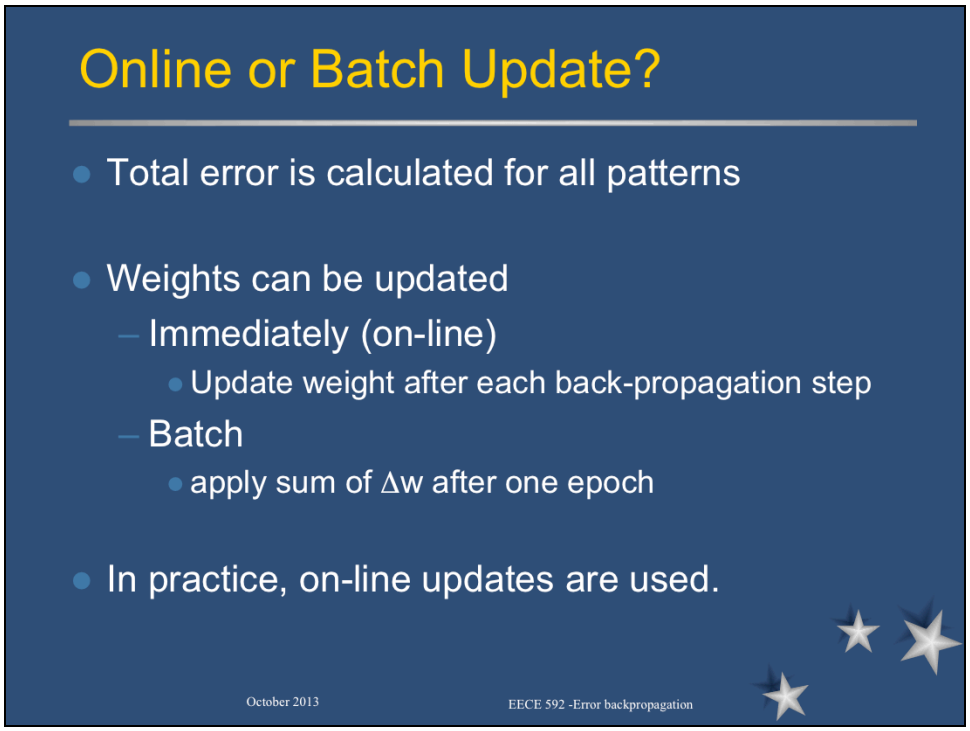


# Activation functions cont.

- In Deep Learning
  - rectilinear threshold function
  - shown to converge faster




$$y = \max(0, x)$$




Remember the total error calculated is for all patterns in the training set. There are two alternatives to updating the weights, on-line updating or batch updating.

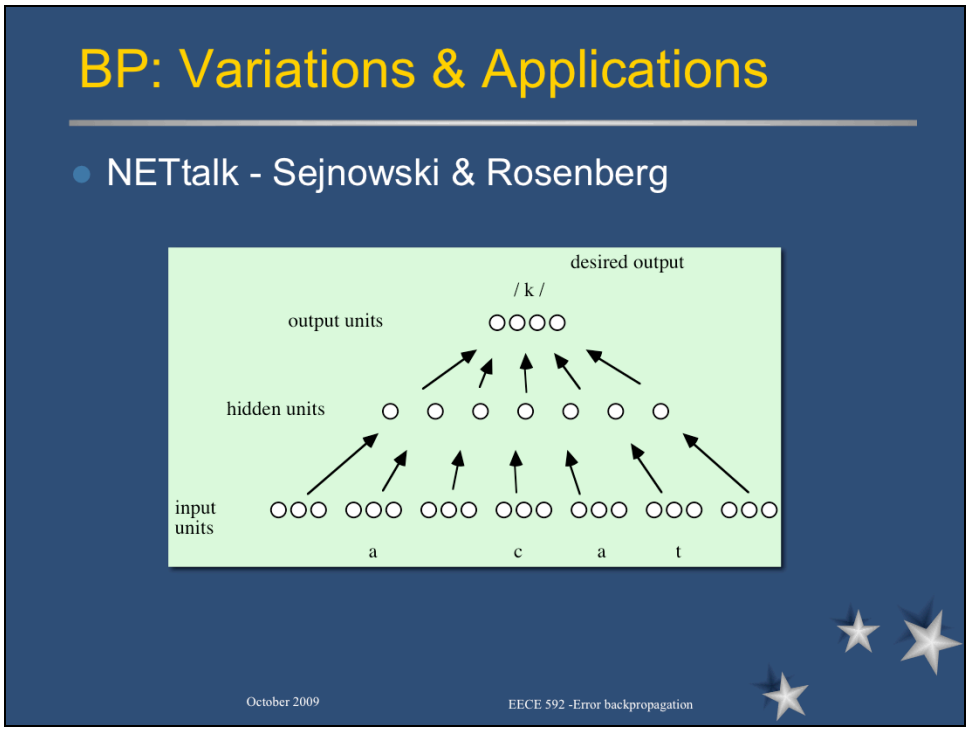
On-line updating (stochastic gradient descent)

> A training example is presented and the errors computed for this pattern. The errors are used immediately to update the weights before the next pattern is presented.

Batch updating

> As each training pattern is presented, the weight changes based on the errors are summed. Once all patterns in the training set have been presented, the sum of the weight changes is used to update the weights before another pass through the training set.

Note that gradient descent is defined in terms of batch updates. In practice, the best (fastest) approach is to update after presentation of each example, i.e. on-line updates.

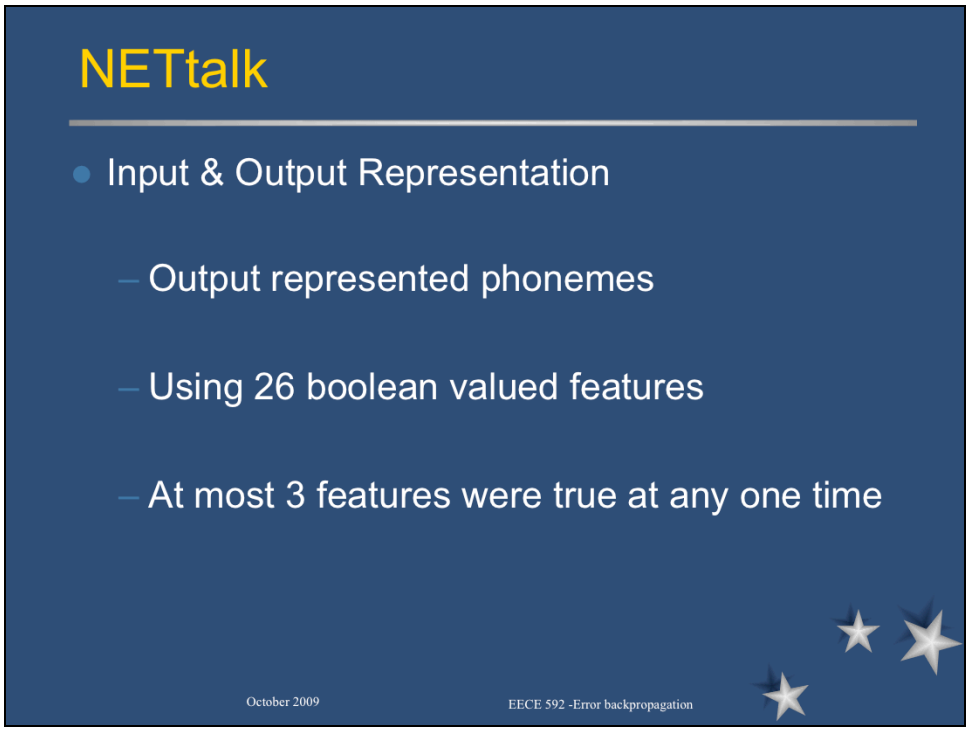



NETtalk is a now well known early application of backpropagation that helped revive interest in neural networks.

The goal of the project was to develop a network that could learn to pronounce English text.

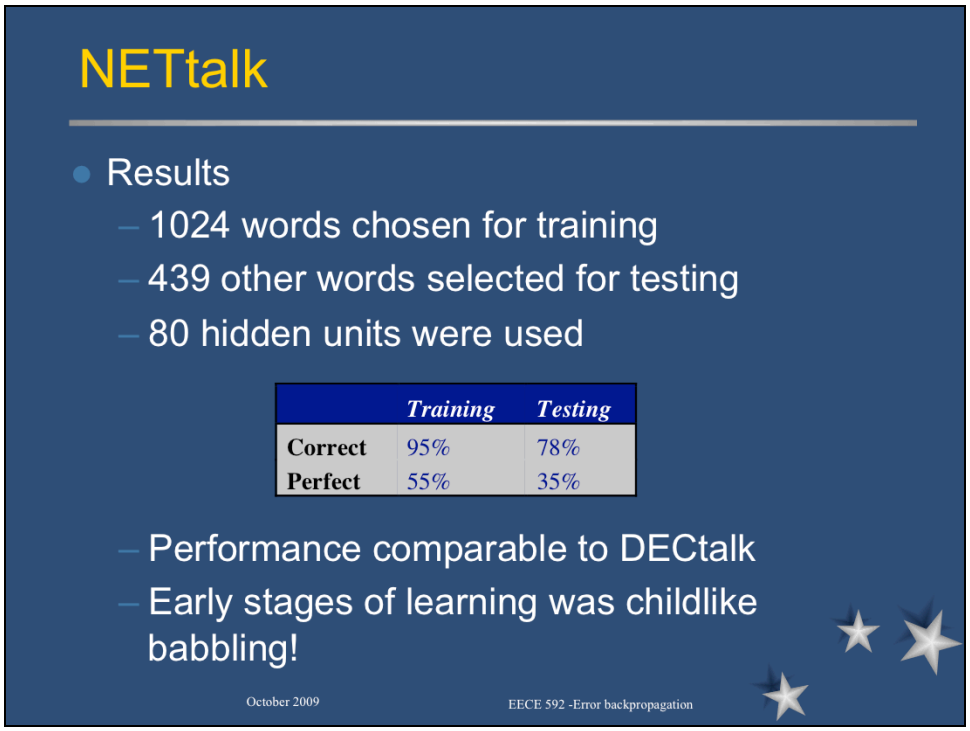
Input & output Representation

> The input to the net consisted of ASCII text and the output was a phoneme description. The output of the net was connected to a device for actually producing the voice for a given phoneme.
>
> To capture contextual data, the input was a seven character moving window, 3 characters before and 3 after the character to be spoken. Each character was represented by 29 boolean inputs (26 for each letter of the alphabet, 2 for punctuations and 1 for a space).




The output was represented using 26 features, e.g. *low*, *voiced*, *affricative*, *nasal*, etc. and again, each output was boolean indicating either the presence or absence of a feature.

At most, 3 features were true and the rest false at any one time.




1024 words were chosen for training and 439 other words selected for testing. 80 hidden units were used.

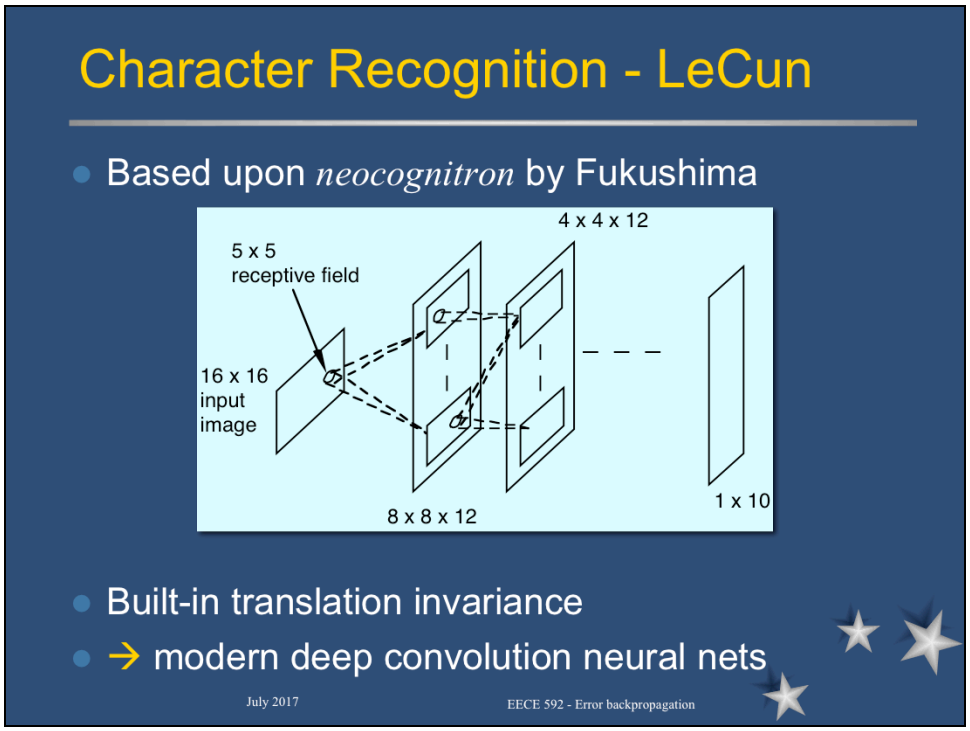
When examined, units in the hidden layer were found to develop according to a distributed representation rather than a local representation, giving the net the benefit of resistance to damage and noise.

An example was considered *perfect* if all output cells had the desired activation and *correct* if the output vector was closer in angle to the desired vector than to any other vector.

The audible speech generated was reasonably coherent and its performance comparable to DECtalk, an earlier, rule based, text-to-speech system.

It's interesting to note that learning was very *childlike* with the initial babbling turning gradually to garbled words and then to understandable words.

NETtalk was a milestone because it proved connectionist approaches were a feasible alternative to conventional programming. DECtalk had taken in the region of a couple of years to develop, mainly because of the need to find and "tune" rules capable of adequately describing the text-to-speech process. NETtalk on the other hand was built in as little as a few months over summer and required no knowledge of speech.



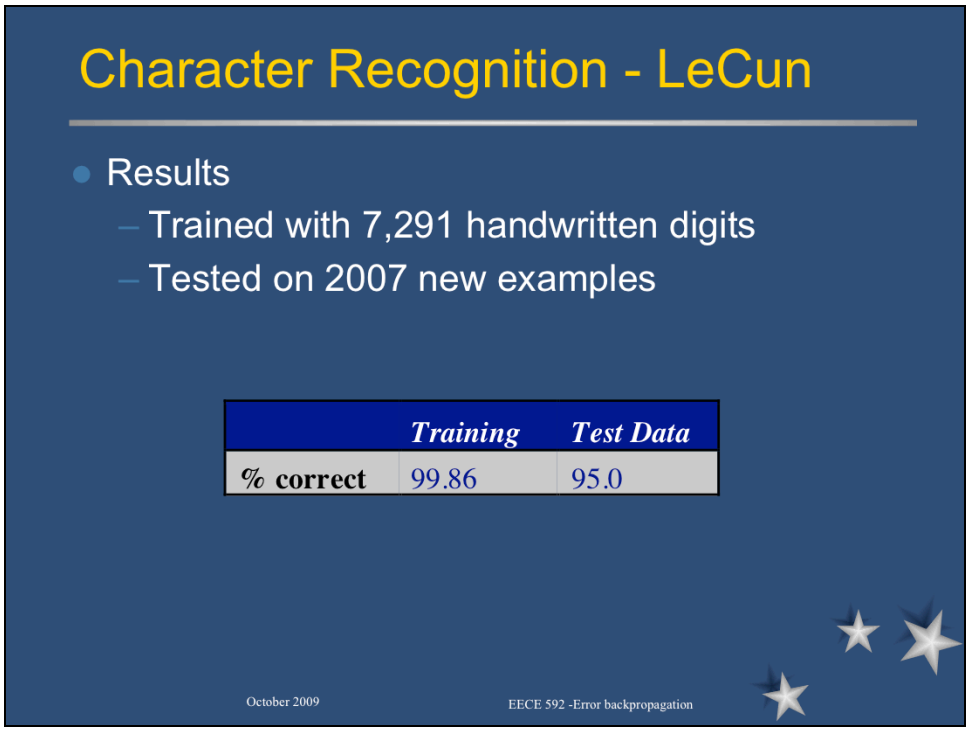

LeCun *et al*. applied a backpropagation model to handwritten character recognition. The model, based upon the *neocognitron* by Fukushima, used a hierarchically structured multi-layer network.

The input was a 16 x 16 gray scale image containing examples of handwritten digits. The outputs were 10 boolean units, used to represent the digit appearing on the input.

The idea was that each layer contained a number of grids of cells. Each cell in the grid takes its input from a small *receptive field* from either the image or the cells in the previous layer. The receptive fields were made to overlap.

All cells in a layer or plane, shared the same weights, The idea here is that each plane is meant to recognize the same pattern, be it from any location in the input. This was an attempt to manually bestow the net with built in *translation invariance*.

Each successive layer is geared towards responding to increasingly complex patterns. The nice thing about the approach, is that all cells in the structure are the same, and all undergo training according to the backpropagation algorithm. However, rather than using a fully connected net and hoping that the required behaviour develops, connectivity within the net has been strategically restricted, thus forcing the net to develop in a particular way.




Results

The net was trained with 7,291 handwritten digits and tested on 2007 new examples. The approach it seems provided for good generalization. In comparison, a normal fully connected net with 40 units was trained to 98.4% which upon testing was found to generalize only 91.9% of the time.

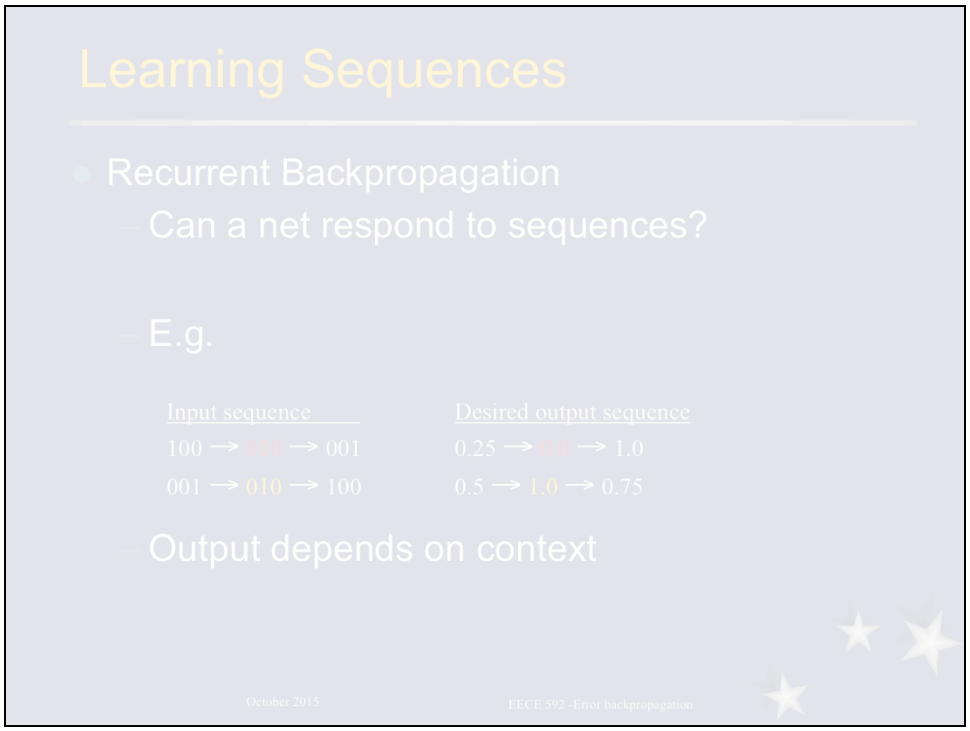
Overfitting of Data

As the number of hidden units in a backpropagation net increase, the net seems to *overfit* the data. That is, it in effect learns the patterns by rote, slowly giving up it ability to generalize.

With fewer hidden units, the net does not have the capacity to store each pattern and is forced to learn a more general description of the input - thus enhancing generalization.

However, some complex problems such as this cannot be learnt well, while at the same time retaining the ability for good generalization.

The approach used by Le Cun appears to prevent overfitting of data by limiting the connectivity within the net.




Given an input pattern and a network trained using backpropagation, the net will produce the same output each time, for that input.

What if instead, we want the net to respond to a particular *sequence* appearing through time at the input?

For example, consider a network with three (boolean) inputs and one continuous output. The required behaviour is shown above.

Following an input of 100, the pattern 010 should generate 0.0, but following an input of 001, the input 010 should generate a 1.0. Clearly a standard backpropagation net could not handle this.

In this case, the output depends on the *context*.

## Learning Sequences

- Recurrent backpropagation






The diagram shows a modified multi-layer perceptron using recurrent units (marked $r_n$). The network is *recurrent* because the connections are not feed forward only; $r_1$ and $r_2$ not only provide inputs to $u_4$ and $u_5$, but the outputs of $u_4$ and $u_5$ are fed back in to $r_1$ and $r_2$.

How to use recurrent backpropagation

A forward pass through the net is based upon the fact that the recurrent units inputs reflect their corresponding regular unit outputs in the previous "tick" of the clock. This provides a way of preserving contextual information for the current pattern on the inputs. Training is a little more complicated and is performed in batch for each pattern sequence. I.e. the weights changes are accumulated for each pattern in the sequence before updating takes place. Errors are accumulated for the last "tick" in the sequence working back to the first. The weight changes are computed using the normal delta rule for backpropagation. In some cases, when the input pattern is small, it may be possible to simplify the sequence by joining it together into one large input, much the same way as it was done for NETtalk. That is not practical if a single pattern in the sequence is already large. This might be the case in areas such as stock market prediction where large amounts of data are often needed for a single prediction to be made. One set of recurrent units provides a temporal context extending back to the previous input pattern. An addition of a second set of recurrent units, attached to the first, would increase the context to the