MULTILAYER NEURAL NETWORKS

Jeff Robble, Brian Renzenbrink, Doug Roberts

Multilayer Neural Networks

We learned in Chapter 5 that clever choices of nonlinear φ functions we can obtain arbitrary decision boundaries that lead to minimum error.

Identifying the appropriate nonlinear functions to be used can be difficult and incredibly expensive.

We need a way to *learn* the non-linearity at the same time as the linear discriminant.

Multilayer Neural Networks, in principle, do exactly this in order to provide the optimal solution to arbitrary classification problems.

Multilayer Neural Networks implement linear discriminants in a space where the inputs have been mapped non-linearly.

The form of the non-linearity can be learned from simple algorithms on training data.

Note that neural networks require continuous functions to allow for gradient descent

Multilayer Neural Networks

Training multilayer neural networks can involve a number of different algorithms, but the most popular is *the back propagation algorithm* or generalized delta rule.

Back propagation is a natural extension of the LMS algorithm.
The back propagation method is simple for models of arbitrary complexity.
This makes the method very flexible.

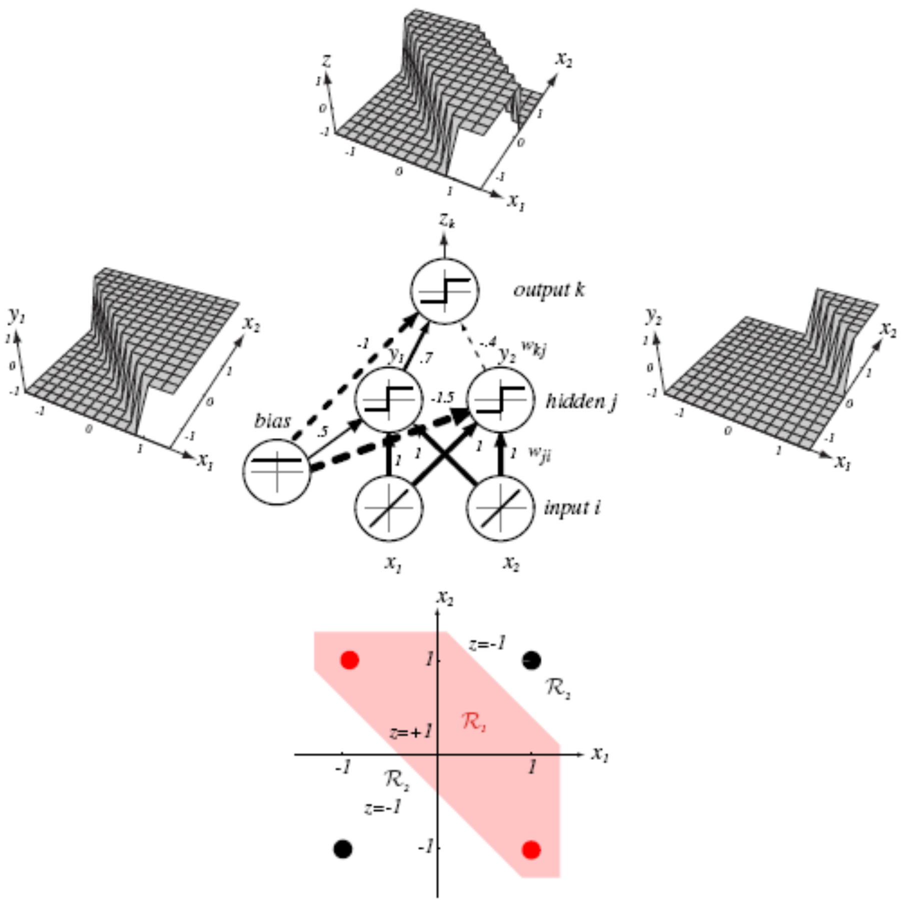
One of the largest difficulties with developing neural networks is regularization, or adjusting the complexity of the network.

Too many parameters = poor generalization

Too few parameters = poor learning

Example: Exclusive OR (XOR)

x – The feature vector
y – The vector of hidden layer outputs
k – The output of the network
w_{ji} – weight of connection between input unit i and hidden unit j
w_{kj} – weight of connection between input hidden unit j and output unit k
bias – A numerical bias used to make calculation easier



Example: Exclusive OR (XOR)

XOR is a Boolean function that is true for two variables if and only if one of the variables is true and the other is false.

This classification can not be solved with linear separation, but is very easy for a neural network to generate a non-linear solution to.

The hidden unit computing y_1 acts like a two-layer Perceptron.

It computes the boundary $x_1 + x_2 + bias_1 = 0$. If $x_1 + x_2 + bias_1 > 0$, then the hidden unit sets $y_1 = 1$, otherwise y_1 is set equal to –1. Analogous to the OR function.

The other hidden unit computes the boundary for $x_1 + x_2 + bias_2 = 0$, setting $y_2 = 1$ if $x_1 + x_2 + bias_2 > 0$. Analogous to the negation of the AND function.

The final output node emits a positive value if and only if both y_1 and y_2 equal 1.

Note that the symbols within the nodes graph the nodes activation function.

This is a 2-2-1 fully connected topology.

Feedforward Operation and Classification

Figure 6.1 is an example of a simple three layer neural network

The neural network consists of:

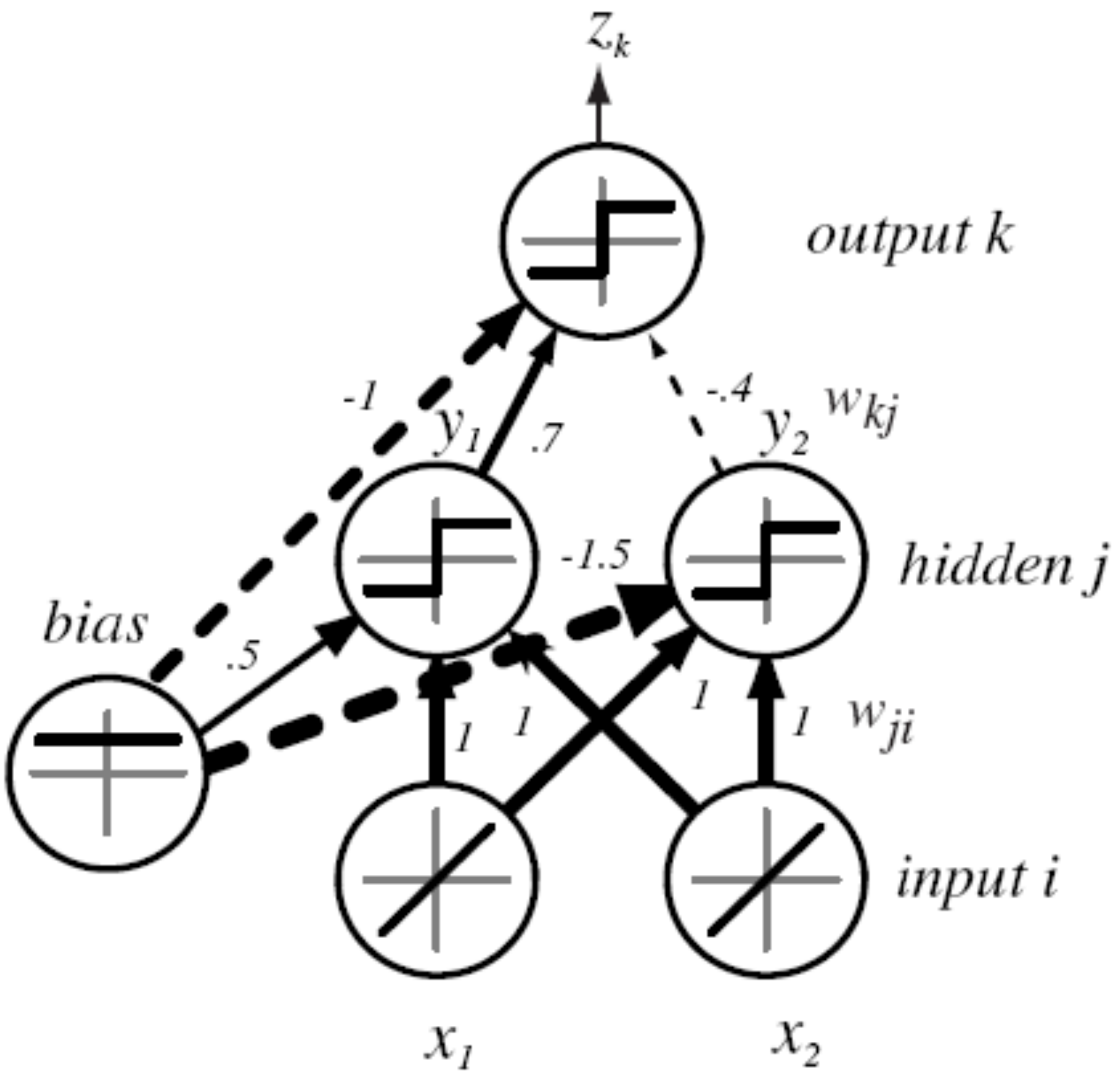
- An input layer
- A hidden layer
- An output layer

Each of the layers are interconnected by modifiable weights, which are represented by the links between layers

Each layer consists of a number of units (neurons) that loosely mimic the properties of biological neurons

The hidden layers are mappings from one space to another. The goal is to map to a space where the problem is linearly separable.

Feedforward Operation and Classification



Activation Function

The input layer takes a two dimensional vector as input.

The output of each input unit equals the corresponding component in the vector.

Each unit of the hidden layer computes the weighted sum of its inputs in order to form a a scalar net activation (net) which is the inner product of the inputs with the weights at the hidden layer.

Activation Function

$$net_j = \sum_{i=0}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv \mathbf{w}_j^t \mathbf{x} \qquad (1)$$

Where

- *i* indexes units in the input layer
- *j* indexes units in the hidden layer
- w_{ij} denotes the input-to-hidden layer weights at the hidden unit *j*

Activation Function

Each hidden unit emits an output that is a nonlinear function of its activation, f(net), that is:

$$y_j = f(net_j) \quad)2($$

One possible activation function is simply the sign function:

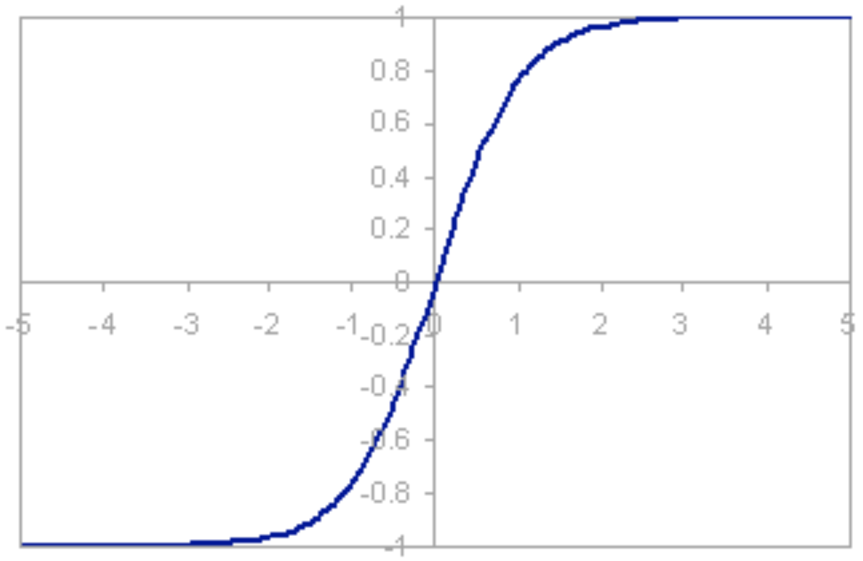
$$f(net) = Sgn(net) \equiv \begin{cases} 1 & if \quad net \geq= 0 \\ -1 & if \quad net \leq= 0 \end{cases} \qquad (3)$$

The activation function represents the *nonlinearity* of a unit.

The activation function is sometimes referred to as a *sigmoid* function, a *squashing* function, since its primary purpose is to limit the output of the neuron to some reasonable range like a range of -1 to +1, and thereby inject some degree of non-linearity into the network.

Activation Function

LeCun suggests the hyperbolic tangent function as a good activation function. *tanh* is completely symmetric:



Because *tanh* is asymptotic when producing outputs of ±1, the network should be trained for an intermediate value such as ±.8.

Also, the derivative of *tanh* is simply $1\text{-}tanh^2$, so if

$f(net) = tanh(net)$, then the derivative is simply $1\text{-}tanh^2(net)$.

When we discuss the update rule you will see why an activation function with an easy-to-compute derivative is desirable.

Activation Function

Each output unit similarly computes its net activation based on the hidden unit signals as:

$$net_k = \sum_{j=1}^{n_H} x_j w_{kj} + w_{k0} = \sum_{j=1}^{n_H} x_j w_{kj} \equiv \mathbf{w}_k^t \mathbf{y} \qquad (4)$$

$$net_j = \sum_{i=0}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv \mathbf{w}_j^t \mathbf{x} \qquad (1)$$

Where:

- *k* indexes units in the output layer
- n_h denotes the number of hidden units

Equation 4 is basically the same as Equation 1. The only difference is are the indexes.

Activation Function

An output unit computes the nonlinear function of its *net*

$$z_k = f(net_k) \quad (5)$$

The output z_k can be thought of as a function of the input feature vector **x**

If there are c output units, we think of the network as computing c discriminant functions $z_k = g_k(\mathbf{x})$

Inputs can be classified according to which discriminant function is determined to be the largest

General Feedforward Operation

Given a sufficient number of hidden units of a general type any function can be represented by the hidden layer

This also applies to:

- More inputs
- Other nonlinearities
- Any number of output units

Equations 1, 2, 4, and 5 can be combined to express the discriminant function $g_k(\boldsymbol{x})$:

$$g_k(\mathbf{x}) \equiv z_k = f\left(\sum_{j=1}^{n_H} w_{kj} f\left(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}\right) + w_{k0}\right) \qquad (7)$$

Expressive Power

Any continuous function from input to output can be implemented in a three layer network given

- A sufficient number of of hidden units nH
- Proper nonlinearities
- Weights

Kolmogorov proved that any continuous function $g(\mathbf{x})$ defined on the hypercube I^n $(I = [0,1]$ *and* $n \geq 2)$ can be represented in the form

$$g(\mathbf{x}) = \sum_{j=1}^{2n+1} \Xi_j \left(\sum_{i=1}^{d} \Psi_{ij}(x_i) \right) \qquad (8)$$

Expressive Power

Equation 8 can be expressed in neural network terminology as follows:

- Each of the $2n + 1$ hidden units takes as input a sum of d nonlinear functions, one for each input feature x_i
- Each hidden unit emits a nonlinear function Ξ of its total input
- The output unit emits the sum of the contributions of the hidden units

Expressive Power

Figure 6.2 represents a 2-4-1 network, with a bias.

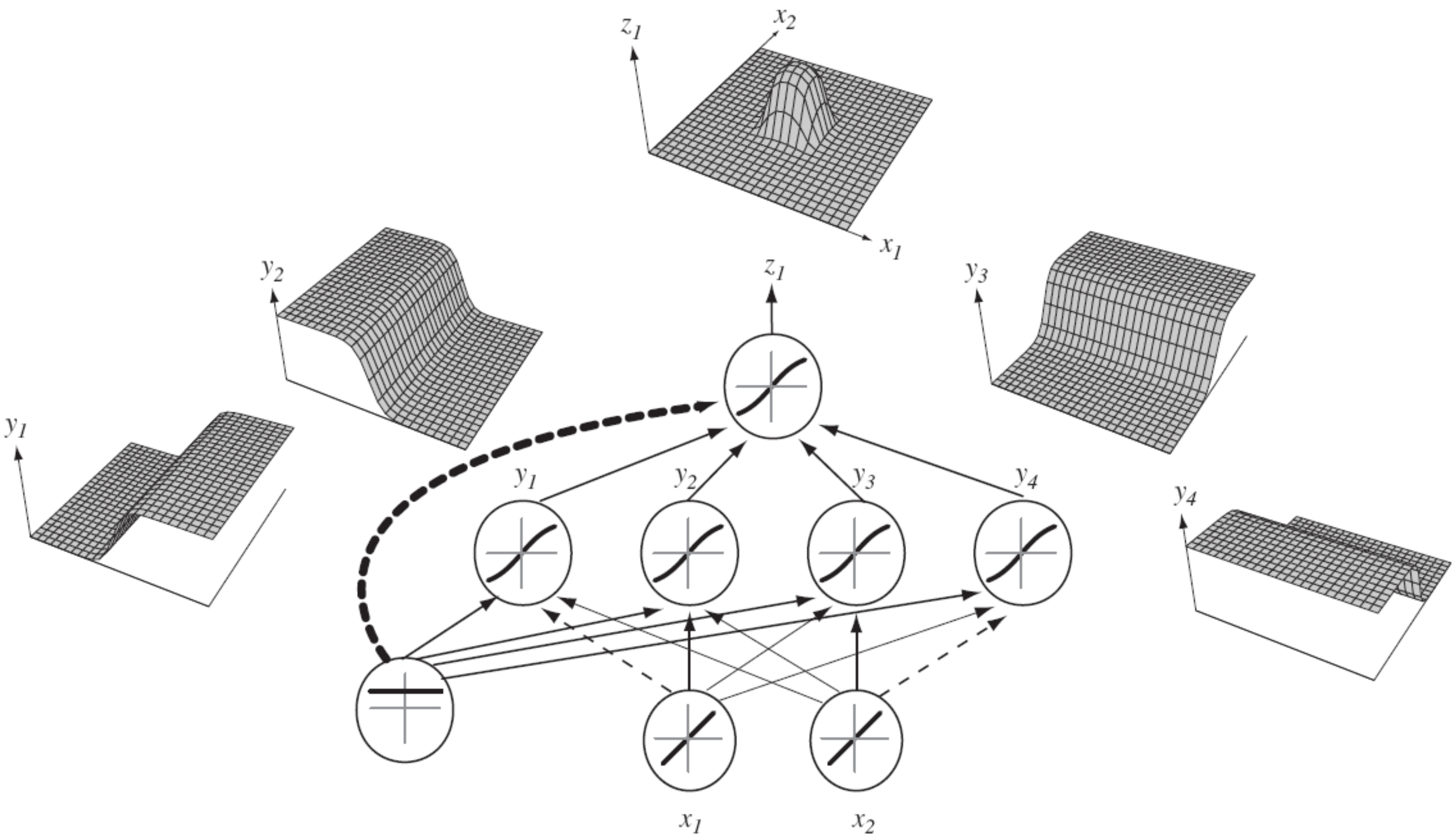
Each hidden output unit has a sigmoidal activation function *f(·)*

The hidden output units are paired in opposition and

thus produce a “bump” at the output unit.

Given a sufficiently large number of hidden units, any continuous function from input to output can be approximated arbitrarily well by such a network.

Expressive Power



Backpropagation Learning Rule

"Backpropagation of Errors" – during training an error must be propagated from the output layer back to the hidden layer in order learn *input-to-hidden weights*

<u>Credit Assignment Problem</u> – there is no explicit teacher to tell us what the hidden unit's output should be

Backpropagation is used for the *supervised learning* of networks.

Modes of Operation

Networks have 2 primary modes of operation:

- Feed-forward Operation – present a pattern to the input units and pass signals through the network to yield outputs from the output units (ex. XOR network)

- Supervised Learning – present an input pattern and change the network parameters to bring the actual outputs closer to desired *target values*

3-Layer Neural Network Notation

d – dimensions of input pattern **x**

x_i – signal emitted by input unit i

(ex. pixel value for an input image)

n_H – number of hidden units

f() – nonlinear activation function

w_{ji} – weight of connection between input unit i and hidden unit j

net_j – inner product of input signals with weights w_{ij} at the hidden unit j

y_j – signal emitted by hidden unit j,

$y_j = f(net_j)$

w_{kj} – weight of connection between input hidden unit j and output unit k

net_k – inner product of hidden signals with weights w_{kj} at the output unit k

z_k – signal emitted by output unit k

(one for each classifier), $z_k = f(net_k)$

t – target vector that output signals **z** are compared with to find differences between actual and desired values

c – number of classes (size of **t** and **z**)

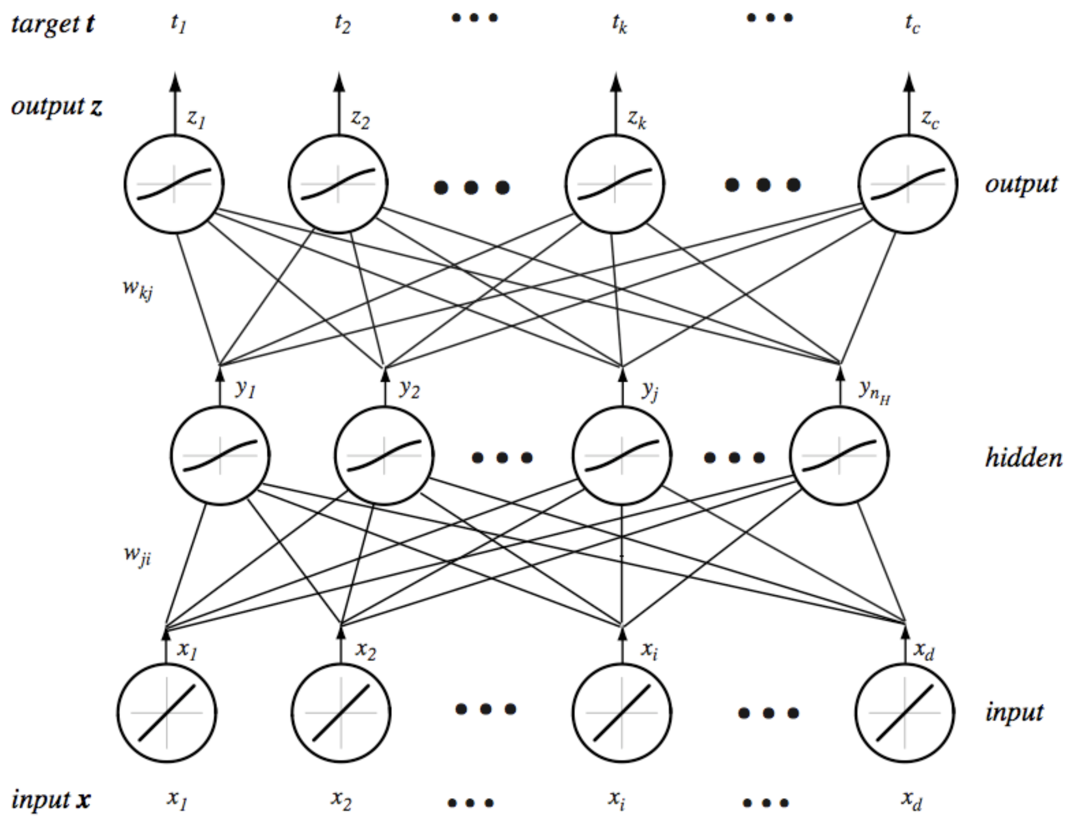


FIGURE 6.4. A d-n_H-c fully connected three-layer network

Training Error

Using **t** and **z** we can determine the training error of a given pattern using the following criterion function:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{z}\|^2 \quad (9)$$

Which gives us half the sum of the squared difference between the desired output t_k and the actual output z_k over all outputs. This is based on the vector of current weights in the network, **w**.

Looks very similar to the the Minimum Squared Error criterion function:

$$J_s(\mathbf{a}) = \sum_{i=1}^{n}(\mathbf{a}^t\mathbf{y}_i - b_i)^2 = \|\mathbf{Ya} - \mathbf{b}\|^2 \quad (44)$$

Y– n-by- $\acute{d}$ matrix of x space feature points to y space feature points in $\acute{d}$ dimensions

a – $\acute{d}$ dimensional weight vector

b – margin vector

Update Rule: Hidden Units to Output Units

Weights are initialized with random values and changed in a direction that will reduce the error:

$$\Delta \mathbf{w} = -\eta \frac{\partial J}{\partial \mathbf{w}} \quad (10) \qquad \Delta w_{pq} = -\eta \boxed{\frac{\partial J}{\partial w_{pq}}} \quad (11)$$

η – learning rate that controls the relative size of the change in weights

The weight vector is updated per iteration m, as follows:

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta \mathbf{w}(m) \quad (12)$$

Let's evaluate Δ**w** for a 3-layer network for the output weights w_{kj}:

$$\boxed{net_k = \sum_{j=0}^{n_H} y_j w_{kj}} \quad (4)$$

$$\boxed{\frac{\partial J}{\partial w_{kj}}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -\boxed{\delta_k} \overset{y_j}{\boxed{\frac{\partial net_k}{\partial w_{kj}}}} \quad (13)$$

Sensitivity of unit k describes how the overall error changes with respect to the unit's net activation:

$$\boxed{\delta_k = -\frac{\partial J}{\partial net_k}} \quad (14)$$

Update Rule: Hidden Units to Output Units

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 \quad (9)$$

$$z_k = f(net_k) \quad (5)$$

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k) \quad (15)$$

After plugging it all in we have the weight update / learning rule:

$$\Delta w_{kj} = -\eta\left(\frac{\partial J}{\partial w_{kj}}\right) = -\eta(-\delta_k y_j) = \eta\delta_k y_j = \eta(t_k - z_k) f'(net_k) y_j \quad (17)$$

If we get the correct outputs, $t_k = z_k$, then $\Delta w_{kj} = 0$ as desired

Iterative form:

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta\mathbf{w}(m) = \mathbf{w}(m) + \eta(m)(t_k - z_k) f'(net_k) y_j$$

Looks a lot like the Least Mean Squared Rule when $f'(net_k)=1$:

$$\mathbf{a}(k+1) = \mathbf{a}(k) + \Delta\mathbf{a}(k) = \mathbf{a}(k) + \eta(k)\,(b(k) - \mathbf{a}^{\mathbf{t}}(k) y^k) y^k \quad (61)$$

Update Rule: Input Units to Hidden Units

The update rule for a connection weight between hidden unit j and output unit k :

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j$$

Working backwards from outputs to inputs, we need to find the update rule for a connection weight between input unit i and hidden unit j:

$$\Delta w_{ji} = \eta \delta_j x_i = \eta \left(\left[\sum_{k=1}^{c} w_{kj} \delta_k \right] f'(net_j) \right) x_j \quad (21)$$

Intuitive reasoning

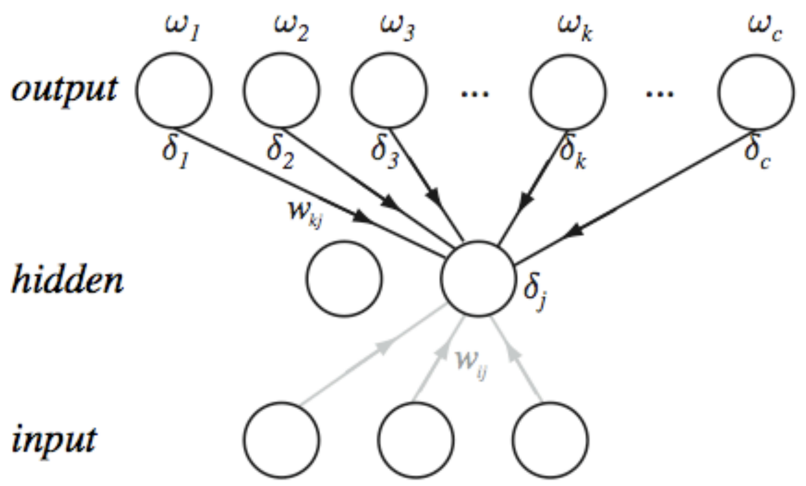


FIGURE 6.5. The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\boxed{\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k.}$ The output unit sensitivities are thus propagated "back" to the hidden units.

Backpropagation Analysis

What if we set the all the initial weights, w_{kj}, to 0 in the update rule?

$$\Delta w_{ji} = \eta \delta_j x_i = \eta \left(\left[\sum_{k=1}^{c} w_{kj} \delta_k \right] f'(net_j) \right) x_j \quad (21)$$

Clearly, Δw_{ji} = 0 and the weights would never change.

It would be bad of all network weights were ever equal to 0.

This is why we start the network with <u>random</u> weights.

The update rules discussed only apply to a 3-layer network where all of the connections were between a layer and it's preceeding layer.

The concepts can be generalized to apply to a network with n hidden layers, to include bias units, to handle a different learning rate for each unit, etc.

It's more difficult to apply these concepts to <u>recurrent networks</u> where there are connections from higher layers back to lower layers/

Training Protocols

Training consists of presenting to the network the collection of patters whose category we know, collectively known as the training set. Then we go about finding the output of the network and adjusting the weights to make the next output more like the desired target values.

The three most useful training protocols are:

- Stochastic
- On-Line
- Batch

In the stochastic and batch training methods, we will usually make several passes through the training data. For on-line training, we will use each pattern once and only once.

We use the term *epoch* to describe the overall number of training set passes. The number of epochs is an indication of the relative amount of learning.

Stochastic Training

In stochastic training patterns are chosen at random from the training set and network weights are updated for each pattern presentation.

Stochastic Back propagation Algorithm

```
1.  begin initialize $n_H$ , w, criterion $\theta$, $\eta$, $m \leftarrow 0$
2.      do m $\leftarrow$ m + 1
3.          $\mathbf{x}^m \leftarrow$ randomly chosen pattern
4.          $w_{ji} \leftarrow w_{ji} + \eta \delta_j x_i$ ; $w_{kj} \leftarrow w_{kj} + \eta \delta_k y_j$
5.      until $\|\nabla J(\mathbf{w})\| < \theta$
6.    return w
7.  end
```

On-Line Training

In on-line training, each pattern is presented once and only once.
There is no use of memory for storing the patterns.

Useful for live training with human testers.

On-line Back propagation Algorithm

```
1.  begin initialize $n_H$, $\mathbf{w}$, criterion $\theta$, $\eta$, $m \leftarrow 0$
2.      do $m \leftarrow m+1$
3.        $\mathbf{x}^m \leftarrow$ select a new, unique pattern to be used
4.        $w_{ji} \leftarrow w_{ji} + \eta \delta_j x_i$   $w_{kj} \leftarrow w_{kj} + \eta \delta_k y_j$
5.      until $\|\nabla J(\mathbf{w})\| < \theta$
6.    return $\mathbf{w}$
7.  end
```

Batch Training

In batch training, the entire training set is presented first and their corresponding weight updates are summed; from there the actual weights in the network are updated. This process is iterated until the stopping criteria is met.

The total training error over *n* individual patterns can be written as: $\sum_{p=1}^{n} J_p$

Batch Back propagation Algorithm

```
1.  begin initialize $n_H$ , w, criterion $\theta$ , $\eta$ , $m \leftarrow 0$
2.      do r $\leftarrow$ m+1 (increment epoch)
3.          m $\leftarrow$ 0; $\Delta w_{ji} \leftarrow 0; \Delta w_{kj} \leftarrow 0;$
4.          do m $\leftarrow$ m+1
5.              $x^m$ = select pattern
6.              $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta\delta_j x_i; \Delta w_{kj} \leftarrow \Delta w_{kj} + \eta\delta_{kj} y_{ji};$
7.          until m=n
8.              $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}; w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$
9.      until $\|\nabla J(\mathbf{w})\| < \theta$
10.   return w
11. end
```

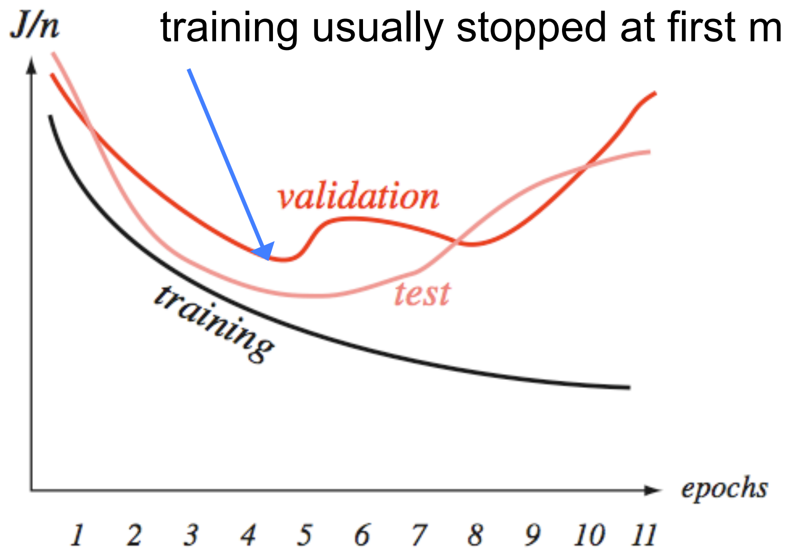
Learning Curves

training set – network hones weights using update rule

test set – estimate generalization error due to weight training; performance of fielded network.

validation set – represent novel patterns not yet classified

training usually stopped at first minimum of validation set



low expressive power (few network weights)

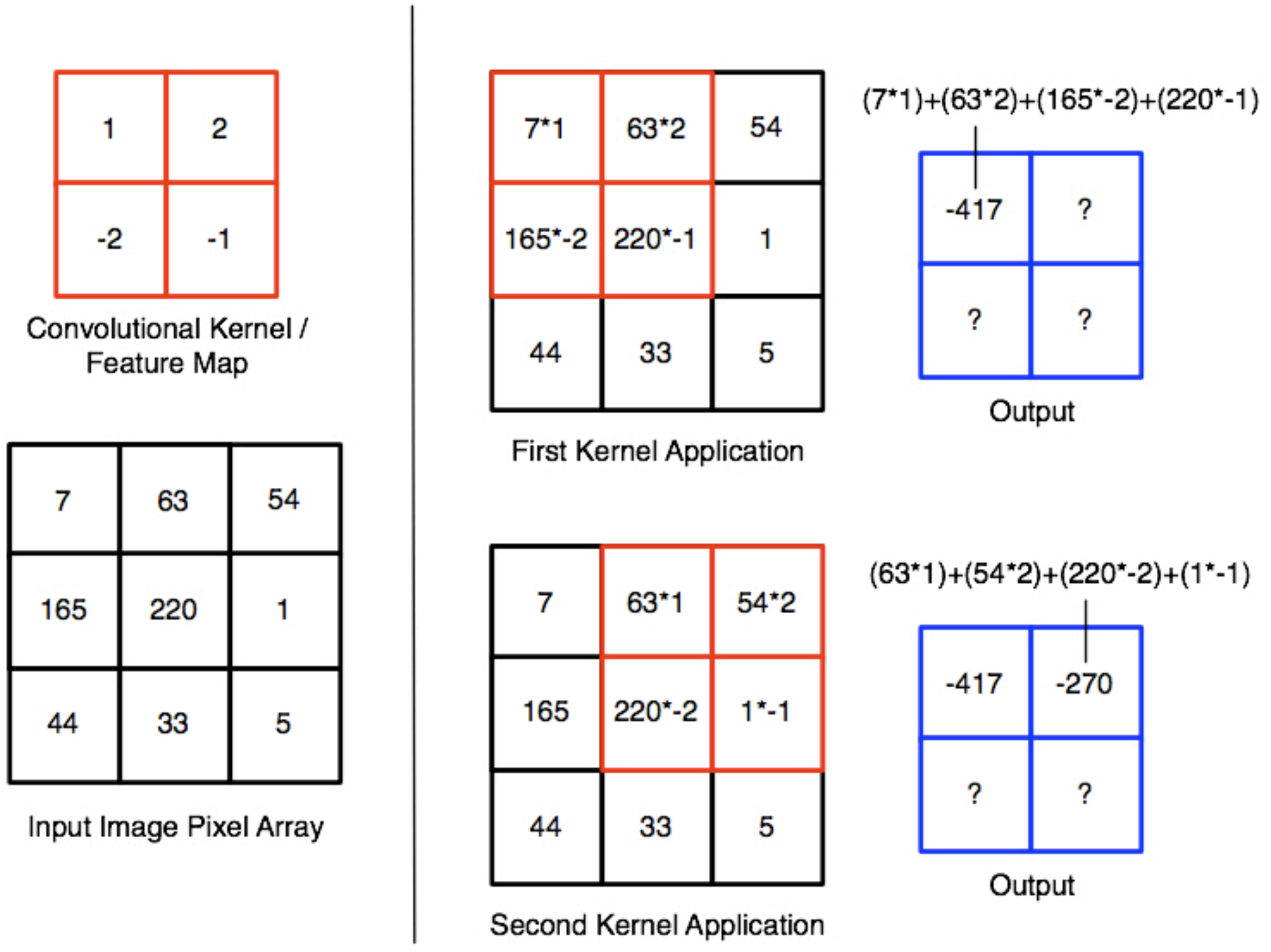
+ high Bayes error

= high asymptotic training error

FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^{n} J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set.

Convolutional Neural Network

Convolutional kernel application example:



Convolutional Neural Network

Each pixel in the sample MNIST image becomes an input node.

Convolutional kernel is applied to the input image in the second layer:

1	2
-2	-1

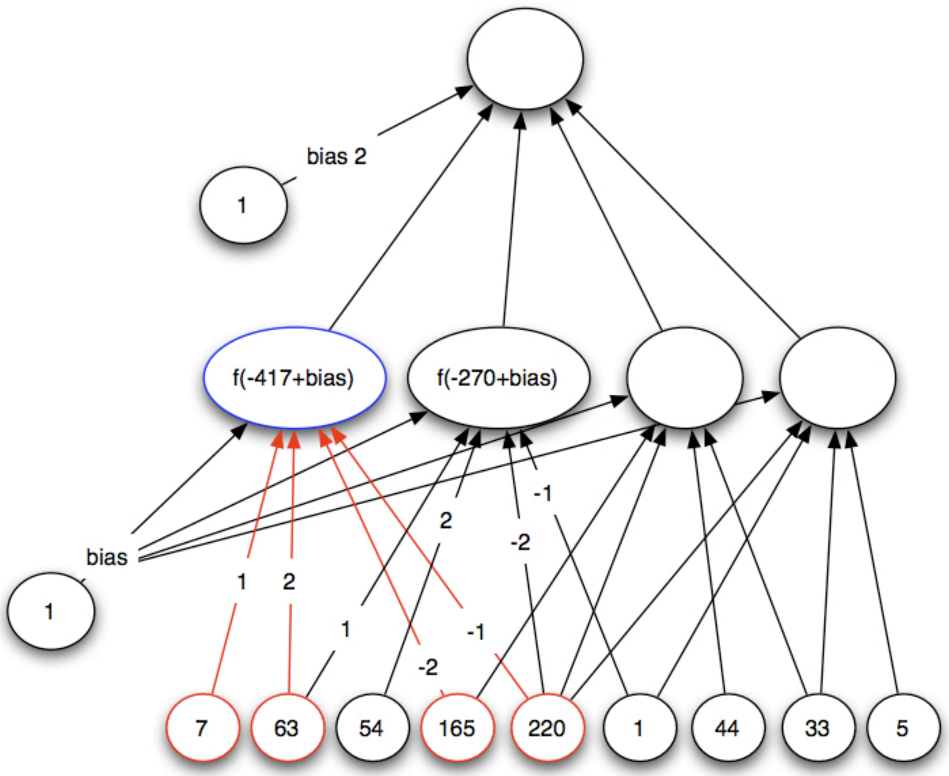
Convolutional Kernel / Feature Map

7	63	54
165	220	1
44	33	5

Input Image Pixel Array

-417	-270
?	?

Output



Convolutional Neural Network Applied to Handwritten Digit Recognition

Simard recommends a convolutional neural network consisting of 5 layers.

Input layer consists of one neuron per pixel in a 29x29 padded MNIST sample image

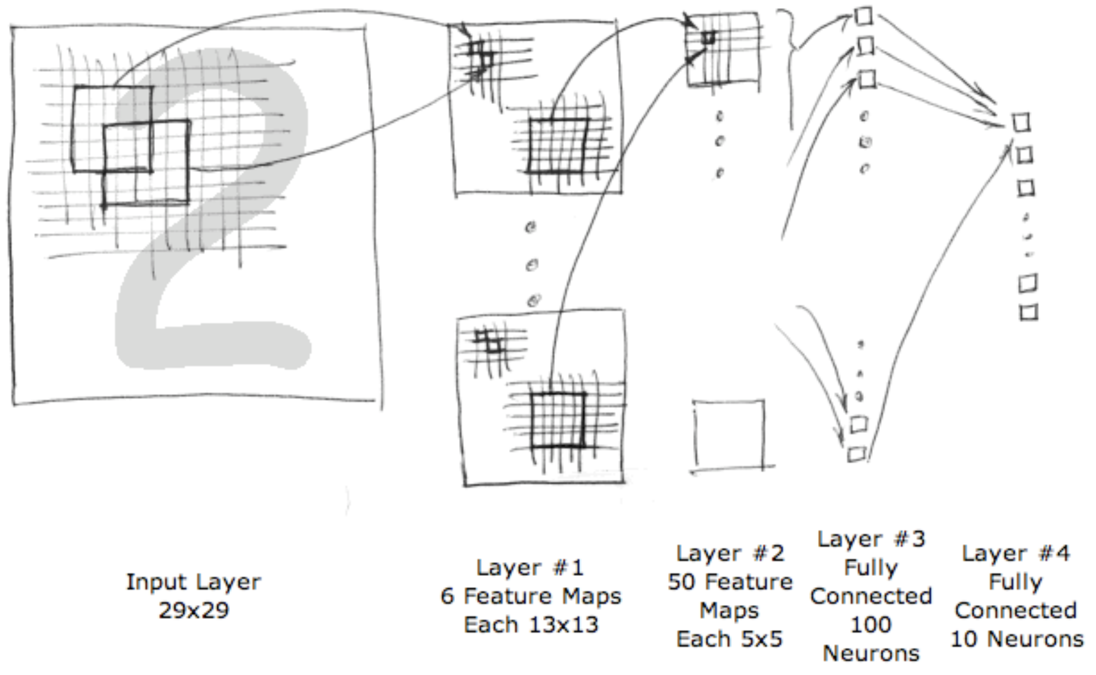
First layer applies 6 feature maps to the input layer. Each map is a randomly distributed 5x5 kernel.

Second layer applies 50 feature maps to all 6 previous maps. Each map is a randomly distributed 5x5 kernel.

First and second layers referred to as trainable feature extractor.

Third and fourth layers are fully connected and compose a universal classifier with 100 hidden units.

Third and fourth layers referred to as a trainable feature classifier.



http://www.codeproject.com/KB/library/NeuralNetRecognition/IllustrationNeuralNet.gif

Structural Adaptability

One of the biggest issues with neural networks is the selection of a suitable structure for the network. This is especially true in unknown environments.

Lee Tsu-Chang developed a method for altering the structure of the neural network during the learning process.

If, during training, the error has stabilized but is larger than our target error, we will generate a new hidden layer neuron.

A neuron can be annihilated when it is no longer a functioning element of the network. This occurs if the neuron is a redundant element, has a constant output, or is entirely dependent on another neuron.

These criteria can be checked by monitoring the input weight vectors of neurons in the same layer. If two of them are linear dependent, then the represent the same hyper plane in the data space spanned by their receptive field and are totally dependent on each other. If the output is a constant value, then the output entropy is approximately zero.

This methodology is incredibly useful for finding the optimal design for a neural network without requiring extensive domain knowledge.

Structural Adaptability

In the graph to the right, the letters A, B, and C represent the times when structural adaptability caused a hidden layer neuron to be removed. Note that these occurred at performance plateaus and did in fact improve the error rate.

The error rate was calculated for an OCR network over each upper-case English letter.

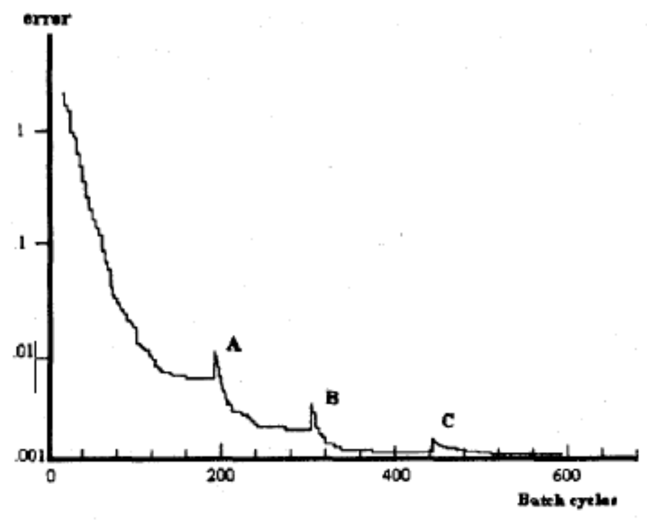


Figure 2. Error convergence; structural adaptability.

References

Duda, R., Hart, P., Stork, D. Pattern Classification, 2nd ed. John Wiley & Sons, 2001.

Y. LeCun, L. Bottou, G.B. Orr, and K.-R. Muller. Efficient Backprop. Neural Networks: Tricks of the Trade, Springer Lecture Notes in Computer Sciences, No. 1524, pp. 5-50, 1998.

P. Simard, D. Steinkraus, and J. Platt. Best Practice for Convolutional Neural Networks Applied to Visual Document Analysis. International Conference on Document Analysis and Recognition, ICDAR 2003, IEEE Computer Society, Vol. 2, pp. 958-962, 2003.

L. Tsu-Chang, *Structure Level Adaptation for Artificial Neural Networks, Kluwer Academic* Publishers, Boston, 1991.

Ramirez, J.M.; Gomez-Gil, P.; Baez-Lopez, D., "On structural adaptability of neural networks in character recognition," *Signal Processing, 1996., 3rd International Conference on* , vol.2, no., pp.1481-1483 vol.2, 14-18 Oct 1996