6 Neural Networks (1. Perceptrons and 2. Multi layered Perceptrons)

6.1 Perceptrons

In the previous chapter, by a progressive simplification of single neuron models, we arrived at the McCulloch-Pitts neuron model which takes inputs from many neurons and produces a single output. If the net effect of the external inputs is greater than a threshold, the neuron goes into excited state (1), else it remains in its resting state (0).

Using this model, in 1943, its inventors Warren S. McCulloch, a neuroscientist, and Walter Pitts, a logician, set out to construct a model of brain function. Note that it was the time of World War-II. It was also a time when use of computing power was being tested for the first time on a large scale for war purposes – for calculating missile trajectories and breaking enemy codes. T he power of computing technology was just being realized by the world. Therefore it was natural to think of brain as a computer. Since the digital computer works on the basis of Boolean algebra, McCulloch and Pitts thought if it is possible for the brain also to use some form of Boolean algebra.

Since the MP neurons are binary units it seemed worthwhile to check if the basic logical operations can be performed by these neurons. McCulloch and Pitts quickly showed that the MP neuron can implement the basic logic gates AND, OR and NOT simply by proper choice of the weights.

OR Gate:

The truth table of an OR gate is:

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Note that the function below, which represents a MP neuron with two inputs, x_1 and x_2, implements an OR gate.

$y = g(x_1 + x_2 - b)$

where b = 0.5; g(.) is the step function; $x_1, x_2 \in \{0,1\}$. Actually any value of the bias term b, $0 < b < 1$, should work.

AND Gate:

The truth table of an AND gate is:

X1	X2	Y
0	0	0
0	1	0
1	0	0
1	1	1

Note that the function below implements an AND gate.
$y = g(x_1 + x_2 - b)$
where b = 1.5, g(.) is the step function; $x_1, x_2 \in \{0,1\}$. Actually any value of the bias term b, $1 < b < 2$, should work.

NOT Gate:

$y = g(-x + 0.5)$
The truth table of a NOT gate is:

X	Y
0	1
1	0

Note that the function below implements a NOT gate.
$y = g(-x + 0.5)$

More generally, in $y = g(-x + b)$, any value of b in, $0<b<1$, would give a NOT gate.

Thus it became clear that by connecting properly designed MP neurons in specific architectures, any complex Boolean circuit can be constructed. Thus we have a theory of how brain can perform logical operations. McCulloch and Pitts explained their ideas in a paper titled, "A logical calculus of the ideas immanent in nervous activity" which appeared in the Bulletin of Mathematical Biophysics 5:115-133.

Although the idea of considering neurons as logic gates and the brain itself as a large Boolean circuit is quite tempting, it does not satisfy other important requirements of a good theory of the brain. There are some crucial differences between the brain and a digital computer (Table 6.1.1).

Table 6.1.1: Difference between the brain and a digital Computer

Property	Computer	Brain
Shape	2d Sheets of inorganic matter	3d volume of organic matter
Power	Powered by DC mains	Powered by ATP
Signal	Digital	pulsed
Clock	Centralized clock	No centralized clock
Clock speed	Gigahertz	100s of Hz
Fault tolerance	Highly fault-sensitive	Very fault-tolerant
Performance	By programming	By learning

Thus there are some fundamental differences between the computer and the brain. The signals used in the two systems are very different. There is no centralized clock in the brain. Each neuron fires at its own frequency which further changes with time. A brain is very fault tolerant which can be seen by the manner in which a stroke patient recovers. Most importantly a computer has to be programmed whereas the brain can learn by a progressive trial-and-error process.

These considerations led to the feeling that something is wrong with the McCulloch-Pitts approach to the brain.

As an answer to the above need, Frank Rosenblatt developed the Perceptron in 1957. A Perceptron is essentially a network of MP neurons.

[perceptron figure here – n inputs and m outputs]
Thus a Perceptron maps an m-dimensional input vector, onto a n-dimensional output vector. A distinct feature of a Perceptron is that the weights are not pre-calculated as in a MP neuron but are adjusted by a iterative process called training. The general approach to training, not only of a Perceptron, but of a larger class of neural networks (feedforward networks which will be defined later) is depicted in the figure below.

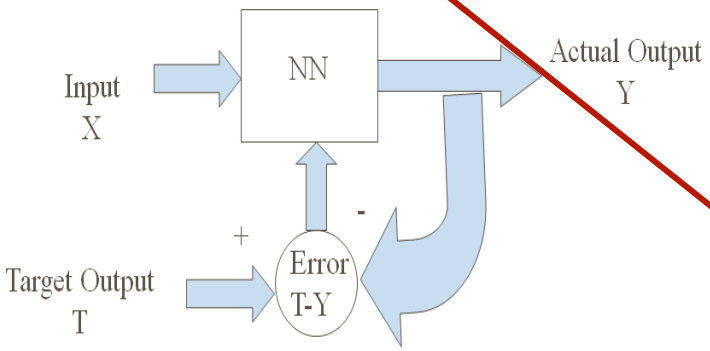


Figure 6.1.1: Training a neural network

The network is initialized with random weights.
When an input X is presented to a neural network (NN), it responds with an output vector Y. Since the weights are random, the network output is likely to be wrong and therefore different from a Desired or a Target output T. Error defined as E = T-Y, is used to adjust the weights in the Perceptron (or NN in general) in such a way that the next time when X is presented to the network, the response Y is likely to be closer to T than before. This iterative procedure is continued with a large number of patterns until the error is minimum on all patterns.

The mechanism by which the weights are adjusted as a function of the error is called the learning rule.
The learning rule can vary depending on the precise architectural details of the Neural Network (NN) used in the above scheme (Fig. 6.1.1).

Instead of directly taking up the task of deriving the learning rule for a Perceptron, let us begin with a very simple neuron model and derive the learning rule. In the process, we would introduce a few terms. The same procedure, with all its jargon, will be used to derive the learning rule for more complex architectures.

6.1.1 Case 1: Linear Neuron model: $y = \mathbf{w}^T \mathbf{x}$

Procedure to find the weights:

1) Noniterative, 2) Iterative

Output Error:

$$E = \frac{1}{2}\sum_n (y(p) - w^T x(p))^2$$

$$\nabla_w E = \sum_p (-x(p))(y(p) - w^T x(p)) = 0$$

1) Noniterative:

Pseudoinverse:

Let,

$$X = \begin{bmatrix} x(1) \\ : \\ x(N) \end{bmatrix} = \begin{bmatrix} x_1(1) & & x_m(1) \\ & & \\ x_1(N) & & x_m(N) \end{bmatrix}_{NXm}$$

Y = X W.

$$w = \begin{bmatrix} w_1 & \dots & w_m \end{bmatrix}^T$$

$$y = \begin{bmatrix} y(1) & \dots & y(N) \end{bmatrix}^T$$

$$d = \begin{bmatrix} d(1) & \dots & d(N) \end{bmatrix}^T$$

$E = (1/2)\,(d-XW)^T(d-XW) = (1/2)d^Td - (X^Td)^Tw + (1/2)\,w^T(X^TX)w$ (6.1.1.1)

$W = (X^TX)^{-1}X^Td$

$R_x = (X^TX)$	correlation matrix
$r_{xd} = X^Td$	cross-correlation matrix
$(X^TX)^{-1}$	pseudo-inverse

2) Iterative:

a) Steepest Descent

$$w(t+1) = w(t) + \eta \sum_{p} (x(p))(d(p) - w^T x(p))$$

$$w(t+1) = w(t) + \eta(r_{xd} - R_x\, w(t)) \qquad (6.1.1.2)$$

In the method of steepest descent, all the training data is used at the same time (packed into R_x and r_{xd}) to update every single weight. This can involve a large memory requirement and can be computationally expensive.

b) Least Mean Square Rule

$$w(t+1) = w(t) + (x(p))(d(p) - w^T x(p)) \qquad (6.1.1.3)$$

Also called the Delta Rule, Widrow-Hoff Rule.

Note that the key difference between the steepest descent rule above (eqn. (6.1.1.2)) and the delta rule (eqn. (6.1.1.3)) is the absence of summation over all training patterns in the latter.

In this case the weight vector does not smoothly converge on the final solution. Instead, it performs a random walk around the final solution and converges only in a least square sense.

Issues:

1. Convergence

1a. Shape of Error function, E (Single minimum for quadratic Error function)

Note that the dominant term in the error function of eqn. (6.1.1.1) is a quadratic form associated with the correlation matrix, R_x.

Since R_x is a positive definite matrix the error function always has a unique minimum. It also has real, positive eigenvalues (λ_i).

1b. Effect of eigenvalues of correlation matrix

Condition number: $\lambda_{max}/\lambda_{min}$

where λ_{max} is the largest eigenvalue and λ_{min} is the smallest eigenvalue of the correlation matrix Rx.

Slows down the descent over the error function if the condition number is too large

2. The need to choose η.

Large η ➔ oscillations, instability

Small η ➔ slow convergence

Bounds over the learning rate, η:

$0 < \eta < 2/(\eta_{max})$

Proof:

$R_x = (X^T X)$ correlation matrix

$r_{xd} = X^T d$ cross-correlation matrix

$(X^T X)^{-1}$ pseudo-inverse

$E = (1/2)(d-XW)^T(d-XW) = (1/2)d^T d - (X^T d)^T w + (1/2)\, w^T (X^T X) w$
$= (1/2)\, d^T d - r^T_{xd}\, w + (1/2)\, w^T (R_x) w$

Final value of w, $w^* = R_x^{-1} r_{xd}$

$E = Emin + (1/2)(w-w^*)^T R_x (w-w^*)$

$Grad(E) = R_x(w-w^*)$

Delta $w = -\eta\, grad(E)$
$= -\eta\, w(t+1) = w(t) - \eta R_x(w(t)-w^*)$
$w(t+1)-w^* = (I-\eta R_x)(w(t)-w^*)$
Let, $v(t) = (w(t)-w^*)$

$$v(t+1) = (I-\eta R_x)\, v(t) \qquad (6.1.1.4)$$

Let $v' = Qv$, where Q is an orthogonal matrix that diagonalizes R_x.

$$v'(t+1) = (I-\eta D)\, v'(t) \qquad (6.1.1.5)$$

$$D = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}$$

If we consider the individual components, v_i, of eqn. (6.1.1.5) above,

$$v_i(t+1) = (1-\eta\lambda_i) v_i(t) \qquad (6.1.1.6)$$

The condition for stability of the last equation, is

$|1-\eta\lambda_i| < 1$. Let us consider the two possible cases of this inequality.

a) $1-\eta\lambda_i < 1 \rightarrow \eta > 0$ which is trivial.

b) $-(1-\eta\lambda_i) < 1 \rightarrow \eta < 2/\lambda_i$ for all i.

Therefore,

$$\eta < 2/\lambda_{max}$$

3. The need to reduce η with time:

There are obvious tradeoffs between use of a large vs. small η. Large η speeds up learning but can be unstable. Small η is stable but results in slower learning.
Therefore, it is desirable to begin with a large η and reduce it with time.
Learning rate variation schedules:

a) $\eta = c/n$; b) $\eta = \eta_o/(1+(n/\tau))$ (6.1.1.7)

6.1.2 Case 2: Perceptron: MP Neuron which has Sigmoid nonlinearity

With the hard-limiting or threshold nonlinearity the neuron acts as a classifier.

Final solution is not unique.
Convergence only if linearly separable.

$$y = g(\sum_{i=1}^{n} w_i x_i - b)$$

Hardlimiter characteristics:

$$g(v) = 1, v \geq 0$$
$$= 0, v < 0$$

For a Perceptron with a single output neuron, the regions corresponding to the 2 classes are separated by a hyperplane given by:

$$\sum_{i=1}^{n} w_i x_i - b = 0$$

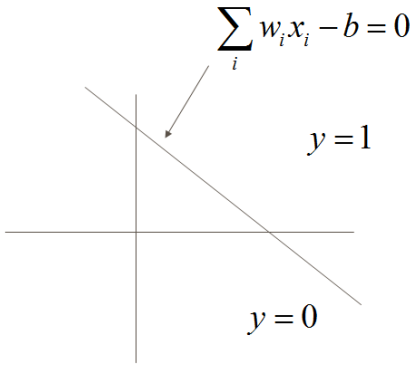


Figure 6.1.2.1: Classification by a perceptron

In other words, a Perceptron classifies input patterns by dividing the input space into two semi-infinite regions using a hyperplane.

6.1.3 Perceptron Learning Rule:

It is also called the LMS Rule or Delta Rule or Widrow-Hoff Rule

The steps involved in Perceptron learning are as follows:

1. Initialization of weights: Set the initial values of the weights to 0. **w**(0) = 0.
2. Present the p'th training pattern, x, and calculate the network output, y.
3. Using the desired output, d, for the input pattern x, adjust the weights by a small amount using the following learning rule:

$$
\begin{aligned}
&w(t+1) = w(t) + \eta[d(t) - y(t)]x(t) \\
&where, \\
&d(n) = +1, x(t) \in C1 \\
&d(n) = -1, x(t) \in C2
\end{aligned}
\qquad (6.1.3.1)
$$

4. Go back to step 2 and continue until the network output error, e = d-y, is 0 for all patterns.

The above training process will converge after N_{max} iterations where,

$$
\begin{aligned}
&\alpha = \min_{x(n) \in C1} w_0^T x(n) \\
&\beta = \max_{x(k) \in C1} \left\| x(k) \right\|^2 \\
&N_{\max} = \beta \left\| w_0 \right\|^2 / \alpha
\end{aligned}
\qquad (6.1.3.2)
$$

See (Haykin 1999, Chapter 3, Section 3.9) for proof of convergence.

Range of η:

$0 < \eta \leq 1$

- Averaging of past inputs leads to stable weight dynamics, which requires small η
- Fast adaptation requires large η

Learning rule can also be derived from an error function:

$$E = \frac{1}{2} \sum_p [(d_p - y_p)^2 \qquad (6.1.3.3)$$

where E denotes the squared error over all patterns.

The learning rule may be derived by performing gradient descent over the error function.

Gradient of Error:

$\Delta w = -\eta \nabla_w E$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\frac{\partial E}{\partial w_i} = -[d - y]\frac{\partial y}{\partial w_i} = -[d - y]g'x_i \qquad (6.1.3.4)$$

The last term in the above equation has g', which is zero everywhere except at the origin if g is a hardlimiting nonlinearity. But if we take a smoother version of g(), which saturate at +1 and -1, like the tanh() function, the learning rule becomes,

$$\Delta w_i = \eta[d - y]g'x_i \qquad (6.1.3.5)$$

Since g' > 0 always for tanh() function, we can absorb it into h, considering it as a quantity that varies with x. We then have,

$$\Delta w_i = \eta'[d - y]x_i \qquad (6.1.3.6)$$

Which is identical to the Perceptron learning rule given in eqn. (6.1.3.1) above.

6.1.3.1 Features of Perceptron:

1) The Perceptrons can only classify linearly separable classes.

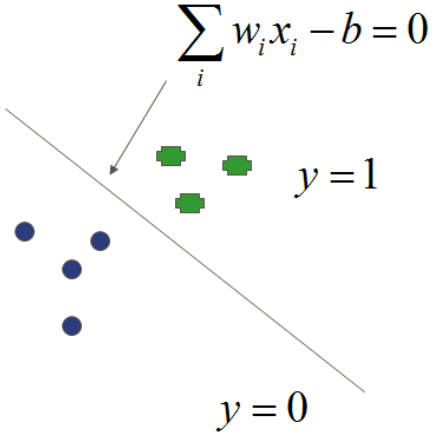


Figure 6.1.3.1.1: Classification of linear separable classes by a perceptron

2. When the training data is linearly separable, there can be an infinite number of solutions.

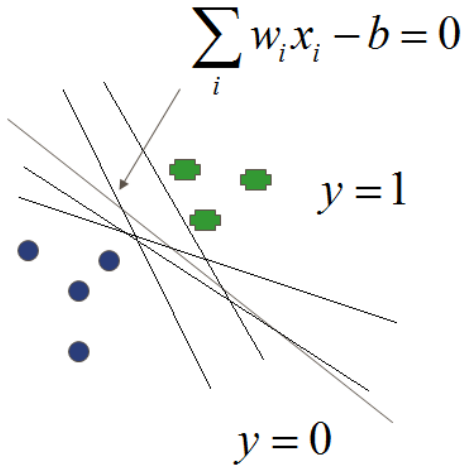
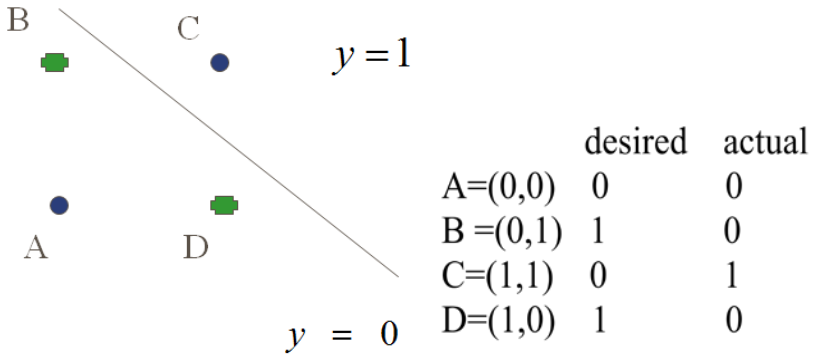


Figure 6.1.3.1.2: Solutions to classify linear separable classes by a perceptron

6.1.3.2 Critique of Perceptrons:

- Perceptrons cannot even solve simple problems like Xor problem (Fig. 6.1.3.2.1)
- Linear model: Can only discriminate linearly separable classes
- Even the multi-layered versions may be afflicted by these weaknesses
  (Minsky & Papert, 1969)



	desired	actual
A=(0,0)	0	0
B =(0,1)	1	0
C=(1,1)	0	1
D=(1,0)	1	0

Figure 6.1.3.2.1: Inability to solve Xor by a perceptron

Exercises:

1) Perform gradient descent on,
   $E = x^2 + 10\,y^2$, with h ranging from 0.01 to 0.2. What is the value of h at which instability occurs?

2) Generate the training set using the formula $f(x,y) = 2x + 3y + n$, where x and y are uniformly distributed over the interval [-1,1], and n is a Gaussian random variable with 0 mean and variance 0.5.
   a) Train a 2-input, 1-output linear neuron model using the data set generated above. Find weights w1 and w2 (no bias) using all the 3 methods: i) pseudoinversion, ii) steepest descent and iii) LMS rule. Compare the 3 solutions. Comment on the variation of $w=(w_1,w_2)$ with time in case of the LMS rule.
3) Character recognition of digits (0-9) in 7-segment representation.
4) Represent digits (0 to 4) in a 10X10 array. Train a 100-input, 5 output perceptron to classify these 5 classes of characters. Generate several examples of each character by slightly shifting the character within the array, adding noise, randomly flipping some bits etc. Separate the data generated into training and test portions. Train the perceptron on the training data, freeze the weights and test it on the test data. Evaluate performance.

6.2 The Multi-layered Perceptron

Improvements over Perceptron:

1) Smooth nonlinearity - sigmoid
2) 1 or more hidden layers

6.2.1 **Adding a hidden layer**:

The perceptron, which has no hidden layers, can classify only linearly separable patterns. The MLP, with at least 1 hidden layer can classify *any* linearly non-separable classes also. An MLP can approximate any continuous multivariate function to any degree of accuracy, provided there are sufficiently many hidden neurons (Cybenko, 1988; Hornik et al, 1989). A more precise formulation is given below.
A serious limitation disappears suddenly by adding a single hidden layer.

It can easily be shown that the XOR problem which was not solvable by a Perceptron can be solved by a MLP with a single hidden layer containing two neurons.

XOR Example:

Neuron 1:

$$V_1 = \sigma(x_1 + x_2 - 1.5)$$

$$V_2 = \sigma(x_1 + x_2 - 0.5)$$

$$y = \sigma(V_1 - V_2 - 0.5)$$

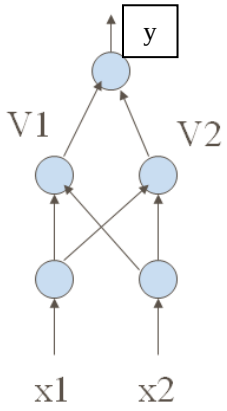


Figure 6.2.1.1: MLP for solving Xor

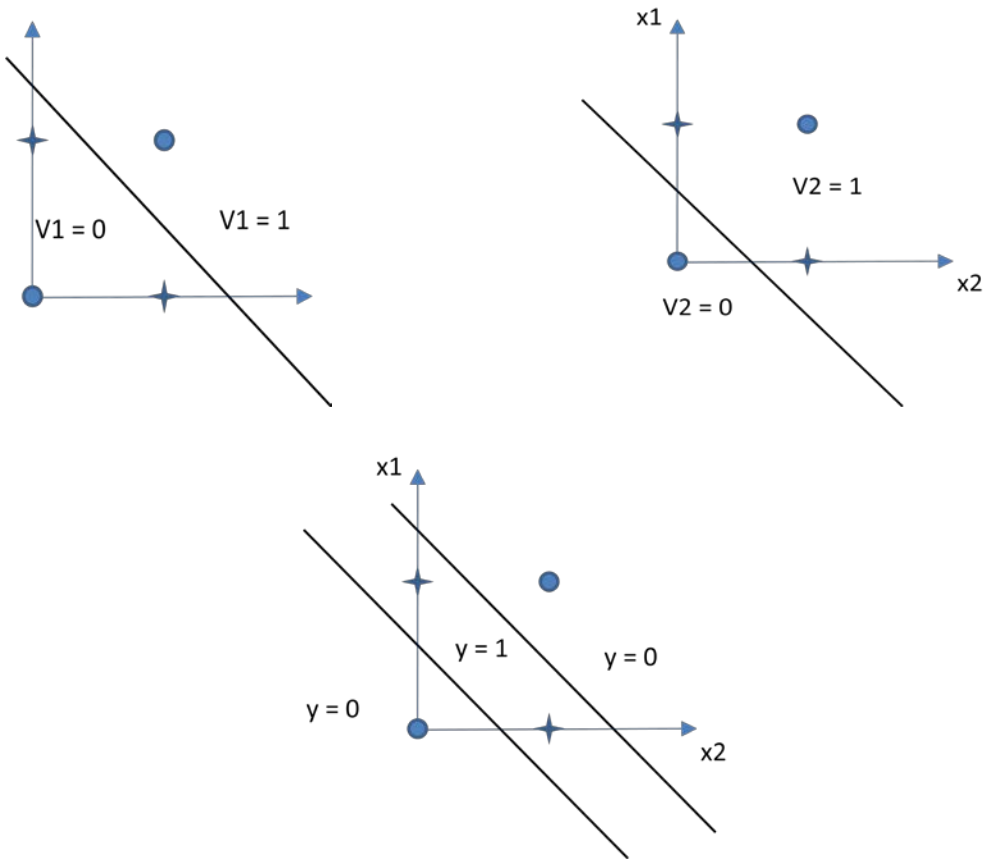


Figure 6.2.1.2: Plots showing the classification by V1,V2 and y(output of MLP)

6.2.2 Training the hidden layer:

Not obvious how to train the hidden layer parameters.

The error term is meaningful only to the weights connected to the output layer. How to adjust hidden layer connections so as to reduce output error? – *credit assignment* problem.

Any connection can be adapted by taking a full partial derivative over the error function, but then to update a single weight in the first stage we need information about distant neurons/connections close to the output layer (locality rule is violated). In a large network with many layers, this implies that information is exchanged over distant elements of the network though they are not directly connected. Such an algorithm may be mathematically valid, but is biologically unrealistic.

6.2.2.1 The Backpropagation Algorithm:

History:

1. Fist described by Paul Werbos (1974) in his PhD thesis at MIT.
2. Rediscovered by Rumelhart, McClelland and Williams (1986)
3. Also discovered by Parker (1985) and LeCun (1985) in the same year.

As in Perceptron, this training algorithm involves 2 passes:

The forward pass – outputs of various layers are computed

The backward pass – weight corrections are computed

Consider a simple 3-layer network with a single neuron in each layer.

Total output error over all patterns: $E = \sum_p E_p$ (6.2.2.1.1)

Squared Output error for the p'th pattern: $E_p = \frac{1}{2}\sum_i e_i^2$ (6.2.2.1.2)

Output error for the p'th pattern: $e_i = d - y_i$ (6.2.2.1.3)

Network output: $y_i = g(h_i^s)$ (6.2.2.1.4)

Net input to the output layer: $h_i^s = \sum_j w_{ij}^s V_j - \theta_i^s$ (6.2.2.1.5)

Output of the hidden layer: $V_j^f = g(h_j^f)$ (6.2.2.1.6)

Net input of the hidden layer: $h_j^f = \sum_k w_{jk}^f x_k - \theta_j^f$ (6.2.2.1.7)

Update rule for the weights using gradient descent:

$$\Delta w_{ij}^s = -\eta \frac{\partial E_p}{\partial w_{ij}^s}; \qquad \Delta \theta_i^s = -\eta \frac{\partial E_p}{\partial \theta_i^s} \quad (6.2.2.1.8)$$

$$\Delta w_{jk}^f = -\eta \frac{\partial E_p}{\partial w_{jk}^f}; \qquad \Delta \theta_j^f = -\eta \frac{\partial E_p}{\partial \theta_j^f} \quad (6.2.2.1.9)$$

Updating w_{ij}^s **:**

$$\frac{\partial E_p}{\partial w_{ij}^s} = \frac{\partial E_p}{\partial e_i}\frac{\partial e_i}{\partial y_i}\frac{\partial y_i}{\partial h_i^s}\frac{\partial h_i^s}{\partial w_{ij}^s}$$

$$= e(-1)g'(h_i^s)V_j \quad (6.2.2.1.10)$$

The delta at the output layer, δ_i^s, is defined as,

$$\delta_i^s = e\ g'(h_i^s) \tag{6.2.2.1.11}$$

Therefore,

$$\Delta w_{ij}^s = -\eta \frac{\partial E_p}{\partial w_{ij}^s} = \eta \delta_i^s V_j \tag{6.2.2.1.12}$$

By similar arguments, it can be easily be shown that the update rule for the threshold term is,

$$\Delta \theta_i^s = -\eta \delta_i^s \tag{6.2.2.1.13}$$

Updating w_{jk}^f :

$$\Delta w_{jk}^f = -\eta \frac{\partial E_p}{\partial w_{jk}^f}$$

$$\frac{\partial E_p}{\partial w_{jk}^f} = \sum_i \frac{\partial E_p}{\partial e_i} \frac{\partial e_i}{\partial y_i} \frac{\partial y_i}{\partial h_i^s} \frac{\partial h_i^s}{\partial V_j} \frac{\partial V_j}{\partial w_{jk}^f}$$

$$= \sum_i e_i (-1) g'(h_i^s) w_{ij}^s \frac{\partial V_j}{\partial w_{jk}^f}$$

$$= \sum_i e_i (-1) g'(h_i^s) w_{ij}^s g'(h_j^f) x_k$$

$$= \sum_i \delta_i^s w_{ij}^s g'(h_j^f) x_k \tag{6.2.2.1.14}$$

Define an error term at the hidden layer as,

$$\delta_j^f = \sum_i \delta_i^s w_{ij}^s g'(h_j^f) \tag{6.2.2.1.15}$$

Therefore,

$$\Delta w_{jk}^f = \eta \delta_j^f x_k \tag{6.2.2.1.16}$$

Similarly the update rule for the threshold term is,

$$\Delta\theta_j^f = -\eta\delta_j^f \qquad (6.2.2.1.17)$$

weight correction = (learning rate) * (local δ from 'top') * (activation from 'bottom')

General formulation of the Backpropagation Algorithm:
Notation:

Input (at k'th input neuron)	-	x_k
Actual Output (at i'th output neuron)	-	y_i
Target output (at i'th output neuron)	-	d_i
Hidden neuron activation	-	V_j^l (of j'th neuron in l'th layer)
Layer number	-	l=0 (input layer) to L (output layer)
Net input	-	h_{jl} (for j'th neuron in l'th layer)
g(h)	-	sigmoid nonlinearity = 1/(1+exp(-βh))

Steps:

1. Initialize weights with small random values
2. (Loop over training data)
3. Choose a pattern and apply it to the input layer

$$V_k^o = x_k^p \quad for\ all\ k.$$

4. Propagate the signal forwards thro' the network using:

$$V_j^l = g(h_j^l) = g(\sum_k w_{jk}^l V_k^{l-1} - b_j^l).$$

for each j, k and and 'l' until final outputs V_i^L have all been calculated.

5. Compute errors, δ's, for the output layer.

$$\delta_i^L = g'(h_i^L)[d_i(p) - V_i^L]$$

6. Compute δ's for preceding layers by backropagation of error:

$$\delta_i^{l-1} = g'(h_i^{l-1})[\sum_j w_{ji}^l \delta_j^l]$$

For l = L, L-1,…1

7. Update weights using the following:

$$\Delta w_{ij}^{l} = \eta \delta_i^{l} V_j^{l-1};$$
$$\Delta \theta_i^{l} = -\eta \delta_i^{l}$$

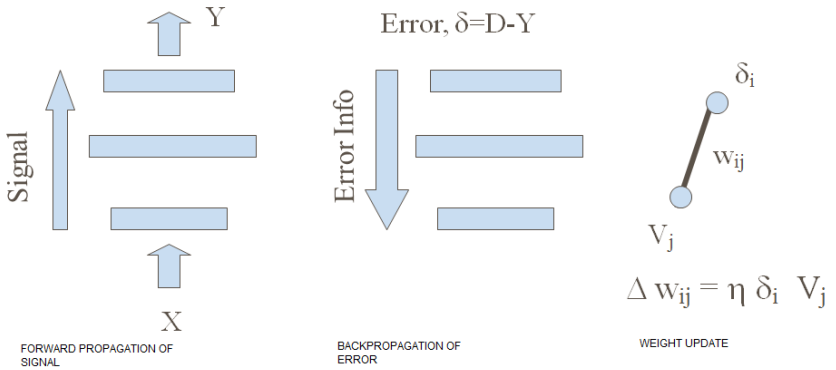


Figure 6.2.2.1: Training using Back propagation algorithm

Traning:

Randomly initialize weights.
Train network using backprop eqns.
Stop training when error is sufficiently low and freeze the weights.

Testing

Start using the network.

Merits of MLP trained by BP:

a) A general solution to a large class of problems.
b) With sufficient number of hidden layer nodes, MLP can approximate arbitrary target functions.
c) Backprop applies for arbitrary number of layers, partial connectivity (no loops).
d) Training is local both in time and space – parallel implementation made easy.
e) Hidden units act as “feature detectors.”
f) Good when no model is available

Problems with MLP trained by BP:

a) Blackbox approach
b) Limits of generalization not clear
c) Hard to incorporate prior knowledge of the model into the network
d) slow training
e) local minima

6.2.3 Architecture of MLP:

If there is no nonlinearity then an MLP can be reduced to a linear neuron.

1. Universal Approximator:

Theorem:
Let g(.) be a nonconstant, bounded, and monotone-increasing continuous function. Let Im denote the m-dimensional hypercube $[0,1]^m$. The space of continuous functions on Im is denoted by $C(I_m)$. Then given any function f in $C(I_m)$ and $\varepsilon > 0$, there exist an integer M and sets of real constants a_i, b_i and w_{ij}, where i = 1, …,n, and j = 1,…,m such that we may define:

$$F(x_1,...,x_m) = \sum_{i=1}^{n} a_i g(\sum_{j=1}^{m} w_{ij} x_j + b_i)$$

As an approximate realization of the function f(.); that is,

$$| F(x_1,\ldots, x_m) - f(x_1,\ldots,x_m)| < \varepsilon$$

for all $x_1,\ldots, x_m$ that lie in the input space.

For the above theorem to be valid, the sigmoid function g(.) has to satisfy some conditions. It must be: 1) non-constant, 2) bounded, 3) monotone-increasing and 4) continuous.

All the four transfer functions described in the section on Perceptrons satisfy conditions #1,2 and 3. But the hardlimiting nonlinearities are not continuous. Therefore, the logistic function or the tanh function are suitable for use as sigmoids in MLPs.

2. In general more layers/nodes greater network complexity

2.1 Although 3 hidden layers with full connectivity are enough to learn any function often more hidden layers and/or special architectures are used.
More hidden layers and/or hidden nodes:

3-layer network:
arbitrary continuous function over a finite domain

4-layer network
Neurons in a 3-layer architecture tend to interact globally.
In a complex situation it is hard to improve the approximation at one point without worsening it at another.
So in a 4-layer architecture:
1st hidden layer nodes are combined to construct locally sensitive neurons in the second hidden layer.
Discontinuous functions:
learns discontinuous (inverse function of continuous function) functions also (Sontag, 1992)

For hard-limiting threshold functions:

1st hidden layer: semi-infinite regions separated by a hyper-plane
2nd hidden layer: convex regions
3rd hidden layer: non-convex regions also

6.2.4 Training MLP:

1. Initialization: is VERY important.

g'(.) appears on the right side of all weight update rules (Refer sections 6.1.1, 6.1.2, 6.2.1). Note that g'(.) is high at the origin and falls on both sides. Therefore most learning happens when the net input (h) to the neurons is close to 0. Hence it is desirable to make initial weights small. A general rule for initialization of input weights for a given neuron is:

Mean(w(0)) = 0.

$std(w(0)) = \frac{1}{\sqrt{m}}$ where m is the number of inputs going into a neuron.

2. Batch mode and Sequential mode:

Epoch: presentation of all training patterns is called an epoch.

Batch mode:

Updating network weights once every epoch is called batch mode update.

- memory intensive
- greater chance of getting stuck in local minima

Sequential mode:

Updating the network weights after every presentation of a data point is sequential mode of update.

- lesser memory requirement
- The random order of presentation of input patterns acts as a noise source lesser chance of local minima

Rate of learning:

We have already seen the tradeoffs involved in choice of a learning rate.

Small learning rate → approximate original continuous domain equations more closely but slows down learning.
Large learning rate η → poorer approximation of original equations. Error may not decrease monotonically and may even oscillate. But learning is faster.

A good thumb rule for choosing eta 'η':
η = 1/m
Where 'm' is the number of inputs to a neuron. This rule assumes that there are different η s for different neurons.

3. Important tip relating learning rate and error surface:
   Rough error surface → slow down, low eta
   Smooth (flat) error surface → speed up, high eta

   i) Momentum:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

Action of momentum:

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)}$$

a) If |a| <1, the above time-series is convergent.
b) If the sign of the gradient remains the same over consecutive iterations the weighted sum delta w_{ji} grows exponentially i.e., accelerate when the terrain is clear.
c) If the gradient changes sign in consecutive iterations, delta w_{ji} shrinks in magnitude i.e., slow down when the terrain is rough.

   ii) Separate eta for each weight:

      a) Separate η for each weight
      b) Every eta varies with time
      c) If delta(w) changes sign several time in the past few iters, decrease η
      d) If delta(w) doesn't change sign in the past few iters, increase η

Stopping Criteria: when do we stop training?

a) Error < a minimum.
b) Rate of change in error averaged over an epoch < a minimum.
c) Magnitude of gradient ||g(w)|| < a minimum.
d) When performance over a test set has peaked.

Premature Saturation:

All the weight modification activity happens only when |h| is within certain limits.
g'(h) ≈ 0, or delta(w) = 0, for large |h|.
NN gets stuck in a shallow local minimum.

Solutions:

1) - Keep a copy of weights
   - Retract to pre-saturation state
   - Perturb weights, decrease η and proceed

2) - Reduce sigmoid gain (lambda) initially
   e) Increase lambda gradually as error is minimized
   f) lambda → 0 means the NN is a linear model in the operating range.
   g) So there is only one minimum

3) g'(h) ← g'(h) + ε
   Quick prop
   Derivative is never 0.
   Network doesn't get stuck, but never settles either.
   Again ε →0 as the network approaches a minimum.

6.2.5 Testing/generalization:

Idea of overfitting or overtraining:

Using too many hidden nodes, may cause overtraining. The network might just learn noise and generalize poorly.

Example of polynomial interpolation:

Consider a data set generated from a quadratic function with noise added. A linear fit is likely to give a large error. Best fit is obtained with a quadratic function. Fit 10^{th} degree might give a low error but is likely to learn the variations due to noise also. Such a fit is likely to do poorly on a test data set. This is called overfitting or poor generalization.

This happens because there are many ways of generalizing from a given training data set.

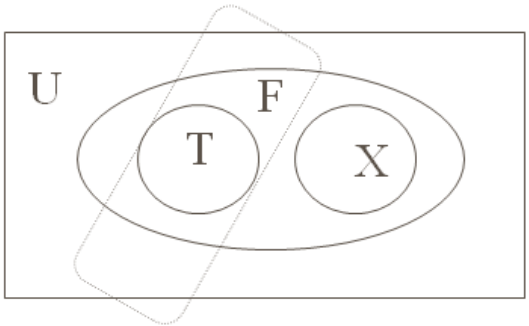


Figure 6.2.5.1: Existence of a multitude of ways of generalization

The above Venn diagram illustrates the possibility of generalizing in multiple ways from a given training data set. U is the universe of all possible input-output patterns. F (the ellipse) represents

the set of I/O pairs that define the function to be learnt by the mlp. T (circle) denotes the training data set which is a subset of F. X denotes the test data set. The dotted rectangle denotes the actual function learnt by the NN, which is consistent with the training set T, but is completely non-overlapping with the test set X, and very different from the unknown function F.

A simple calculation from (Hertz et al 1991).
Boolean function
N-inputs, 1-output

2^N patterns and 2^{2^N} rules totally.

Assume,
p – training patterns say, represent the rule T.
Then there are (2^N – p) test patterns and there are $2^{(2^N - p)}$ rules, R, consistent with rule T.

N = 30, p = 1000 patterns.
You have 2^{10^9} generalizations exist for the same training set T.

6.2.6 Applications of MLP

Three applications of MLPs that simulate aspects of sensory, motor or cognitive functions are described.

1. Nettalk
2. Past tense learning
3. Autonomous Land Vehicle in a Neural Network (ALVINN)

6.2.6.1 NetTalk: A neural network that can read text aloud (Sejnowski and Rosenberg 1986)

Nettalk is a system that can read English text aloud and can pronounce letters accurately based on context (Sejnowski and Rosenberg 1986). It uses a three layer MLP.

Background:

English is not a phonetic language.

Char → Sound mapping is not unique.

The same character is pronounced differently depending on the context.

Examples:

The character ‘c’ is pronounced as /k/ in “cat” and as /s/ in “façade.”
The letters “-ave” form a long vowel in “gave” and “brave” but not in “have.”

Similarly, the letters “-ea-“ are pronounced as /ii/ (long vowel) in “read” (present tense) (pronounced as “reed”) and as /i/ (short vowel) as in “read” (past tense) (pronounced as “red”).

6.2.6.1.1 Network Architecture

Input representation:

26 alphabets and three punctuation marks (comma, fullstop and blank space) are supported. Input characters are not presented one at a time to the network. In order to incorporate the context, each character is presented along with a context which consists of three additional characters on either side of the character. Thus text is presented as windows of 7 symbols. Therefore the number of neurons to the input layer are:

$(26+3)\times 7$ input neurons

Hidden layer : 80 hidden neurons

Output representation :

26 output neurons encoding phonemes.

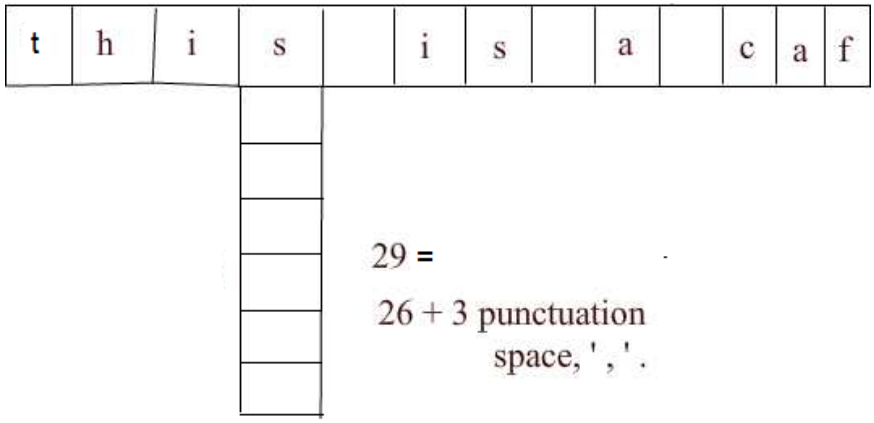


Figure 6.2.6.1.1: Input Representation

Network architecture: Ref: (fig. 6.8 in Hetz et al 1991)

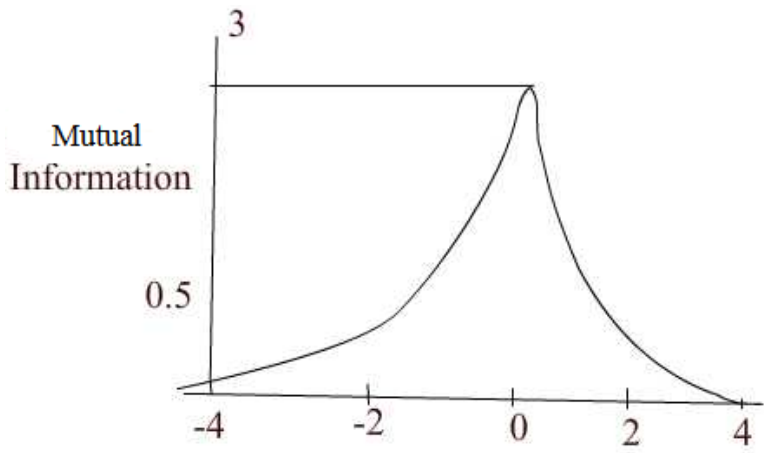


Figure 6.2.6.1.2: Mutuality in the window

A context of window of 7 characters is chosen since mutual information from a central character falls off rapidly in both directions as shown in the above figure. Beyond 3 neighbors mutual information falls to less than 0.1.

6.2.6.1.2 Training data

- Phonetic transcription from informal continuous speech of a child
- 20,000 words from a pocket dictionary

Eg: phone

$\Rightarrow$ f – o n –

A set of 1000 words were chosen form this dictionary; these are selected from the Brown corpus of the most frequent words in English.

6.2.6.1.3 Learning curve

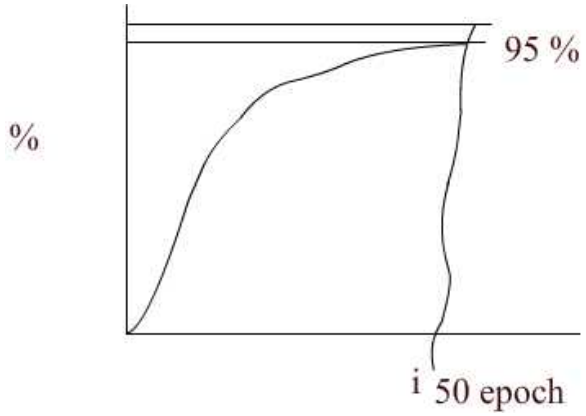


Figure 6.2.6.1.3: Performance curve

Found in human skill learning

Performance reaching about 95% correct after training the network with about 50,000 words.

The presence of a hidden layer is found to be crucial to achieve this high level of performance. When a two-layer network (perceptron) was used for the same problem, performance quickly rose to 82% and saturated at that level.

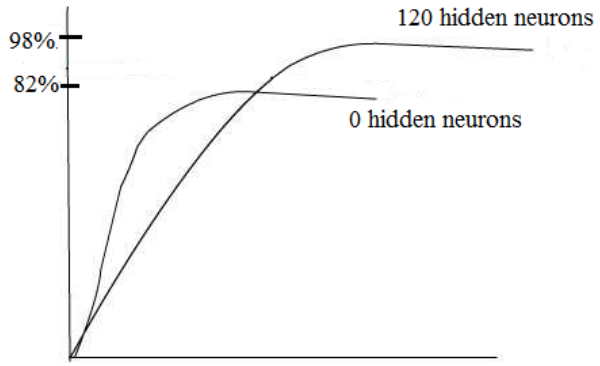


Figure 6.2.6.1.4: Performance curve on alteration of the hidden neurons

6.2.6.1.4 Stages of learning

1. One of the first features learnt by the network is distinction between vowels and consonants
2. Next the network learnt to pause at word boundaries. Output resembled pseudo words with pauses between them.
3. After 10 passes text was understandable

Error patterns

- Errors were meaningful

  e.g thesis

  these

- The network rarely confused between vowels & consonants
- Some errors actually were due to errors in annotation.

6.2.6.1.5 Test Performance

439 words from the regular speech. These words were not present in the training set.

Performance was at78%. Thus the network demonstrated that it can generalize from one set of words to another.

Damage testing

In order to test the robustness of the network in face of damage, random noise, n, uniformly distributed between -0.5 and 0.5 was added to each weight in the trained network.

$$\omega_i \leftarrow \omega_i \pm \upsilon$$

$$\upsilon \in [-0.5, 0.5]$$

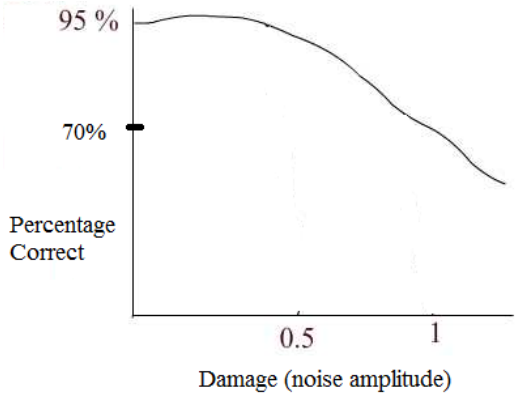


Figure 6.2.6.1.5: Performance on damage

The network was retrained after damage. The damaged network showed 67% performance. Therefore to compare the learning rate of the damaged network with the original network, the original learning curve was considered from a performance level of 67%. Figure 6.2.6.1.6 below shows that although the damaged network begins with the same performance level as the original network, it relearns the task much faster than the original network.

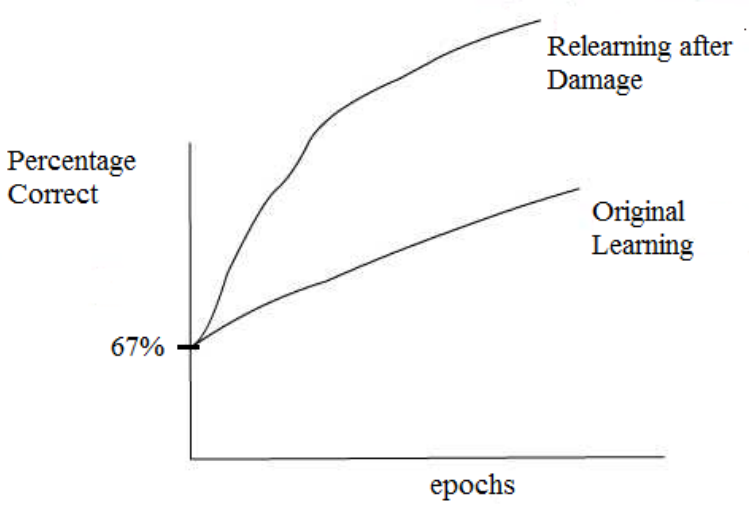


Figure 6.2.6.1.6: Relearning after damage

A careful analysis of the performance of the network specific letter-to-phoneme correspondences showed that different letter-to-phoneme correspondences were learnt at different rates.

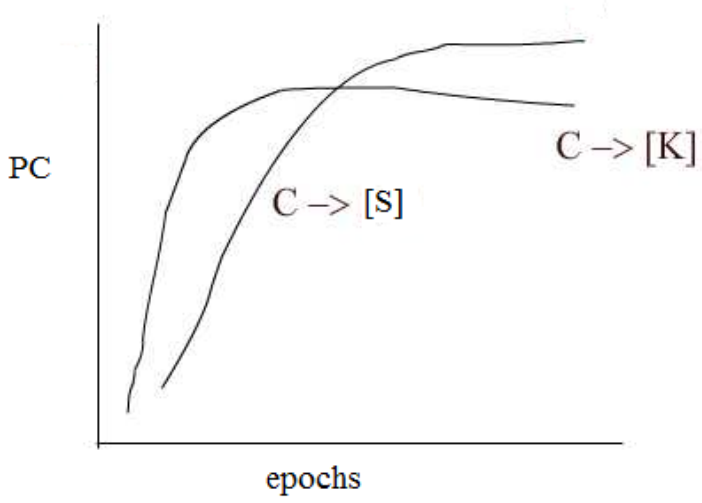


Figure 6.2.6.1.7: Rates of learning for different letter to phenome represenatation

Soft ‘c’ (as in ‘nice’) takes longer to learn than hard ‘c’ (as in ‘cat’). This is probably because hard c occurs about twice as often as soft c in the corpus.

Interestingly children show a similar difficulty in learning.

Ref: Sejnowski, TJ Rosenberg, CR (1986). NETtalk: A parallel network that learns to read aloud, Tech. Rep. No. JHU/EECS-86/Q1

6.2.6.2 Past tense learning (Rumelhart & McClelland, 1986)

Three stages of past tense learning is seen in children.

Stage 1:- Only a small number of words correctly used correctly in past tense

- High frequency words, majority are irregular
- Children tend to get the past tense quickly.
- Typical examples: Came, got, gave, looked, needed, took, went.

Stage 2:-

- Children use much larger number of words
- Many verbs are used with correct past tense forms
  Majority are regular
  Eg. Wiped, pulled

- Children now incorrectly apply regular past tense endings for words which they used in stage 1

  Eg: come $\Rightarrow$ comed or camed

Stage 3:- Regular or irregular forms exist.

- Children have required the use of correct irregular form of past tense
- But continue to apply the regular form to new words they learn.

6.2.6.2.1 Training

Words

10- high frequency verbs (8 irregular + 2 regular)
Came, get, give, look, take, go, have, live, feel

410- medium frequency verbs,
334 regular, 76 irregular

86-low frequency words,
72 regular, 14 irregular

Stage 1:-

10 epochs of high frequency verbs
Enough to produce good performance

Stage 2:-

410 medium frequency verbs were added to 10 verbs (trained for 190 more epochs after phase 2). In this stage errors suddenly start creeping in. Most of the errors are due to regularization of irregular verbs (see fig. 8 below).

Stage 3:-

86 low frequency verbs are tested without training. Beyond state 2, the performance over regulars and irregulars is nearly the same, each touching 100%.

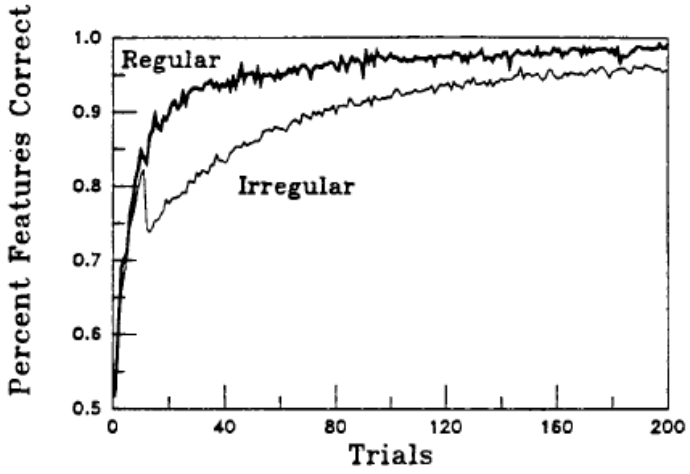


Figure 6.2.6.2.1: Performance curve on past tense learning

Ref: Rumelhart and McClelland, On learning the past tenses of English verbs, Parallel Distributed Processing, Vol. II, 1986

6.2.6.3 Autonomous Land Vehicle in a Neural Network (ALVINN)

6.2.6.3.1 Training:-

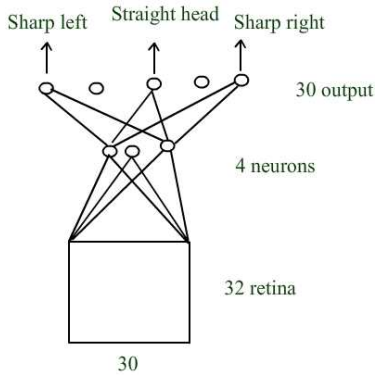


Figure 6.2.6.3.1: Network Architecture - ALVINN

$\frac{1}{4}$ *to* $\frac{1}{2}$ mile of training road which takes about 2 minutes to cover.

50 real images

Transformed 15 times to create 750 images.

6.2.6.3.2 Generalization:

Generalized to new points of the road

Different weather & lightning conditions

6.2.6.3.3 Specialization

Separate networks are trained on:

- Single-lane dirt & paired roads
- Two-lane suburban roads/city, streets
- multilane divided (90 mph) highways

6.2.6.3.4 Hidden layer Analysis

Hidden layer

Unlined paved roads → “edge detectors” trapezoidal shaped road regions

Lined highways → detectors for line markings

Dirt roads → ‘rut’ detectors.

Reference:

Dean Pomerleau, "ALVINN: An Autonomous Land Vehicle In a Neural Network," ***Advances in Neural Information Processing Systems 1,*** **1989.**