MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence, Fall 2003
Recitation 10, November 4th & 5th 2003

Learning by perceptrons & neural nets Prof. Robert C. Berwick

Agenda

1. **Perceiving Perceptrons: single-layer neural networks – Who gets the blame?**
2. **How perceptrons learn; what they can learn; what they cannot learn (Example & demo)**
3. **Multilayer perceptrons – neural nets: the blame game - backpropagation**
4. **How Multilayer perceptrons learn: Example**

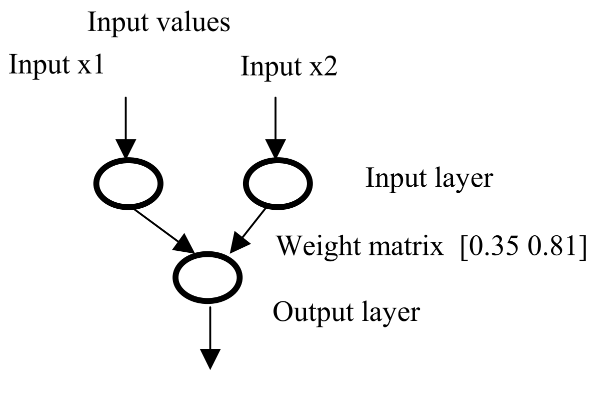
1. Neural Networks & Perceptrons

Recall that we are looking at the problem of learning as a kind of 'curve fitting': we are given some set of **example** (*x,y*) pairs, and our job is to find a function which is producing the relation between *x* and *y*. So, we looking for a function that gives a'best fit' in terms of the error between the observed *x* and the computed or predicted value *y*. Naturally, we would like this function to give good results on new values of *x* beyond those in the training set –the test set. To set the stage, we will look at functions computed by a network of a certain special, simple form, called perceptrons. They are one-layer neural nets where the input is **directly** connected to the output (y or predicted value) by a linear combination of weights. Our motivation is as follows: Both Perceptrons and Neural Nets work basically the same way.

1. Pick a random set of weight
2. For each (input, output) point in the training data, while the error > some epsilon value,
3. Compute the value **predicted** by the current network, the **forward value** or **output activation**
4. Compute the difference between the training data output and the forward value, i.e., the **error** the current network makes on this point
5. If there is no error, do nothing
6. If there is an error, compute how much to change the network's weights using hill-climbing method – ie, the **derivative** of the output with respect to the weights and update them; go to step 1.
7. If the error is < epsilon, stop

NOTE: The only difference between one-layer neural nets (perceptrons) and multi-layer neural nets is *how the error* in step 5 is *assigned to the weights* (how it is used to changed the weights). the weight correction is so easy, because the change in the output, y, with respect to x is just the derivative of *wx,* or simply the weight *w.*

Error driven learning means we start off by *guessing* an answer, and then *changing* this guess based on how far away they are from the correct answer (=the training data). The machine changes only if it is doing something wrong. That is the sense in which both sorts of machines are supervised, 'reward-punishment' or 'stimulus-response' learning methods.
For perceptrons is very simple to figure out how to change a guess in the right direction. This is because perceptron has its weights directly connected to the output,value, so we can 'blame' a weight directly if it is causing part of the error i.e, the derivative of *wx*, or just the weight *w*. For example, in the perceptron below, if we 'tweak' the weight on the left from 0.35 to 0.7, then the output value will go up by that amount, directly. Correspondingly, if the output value is 0.35 too low, we know that tweaking the left-hand weight by 0.35 will push it up to being correct.



As the figure shows, for perceptrons, the curve fitting or learning problem is specialized as follows. We imagine **T** to be some **target** pattern where we want the perceptron to output the number 1 if the pattern is in the input, and –1 otherwise. Note this means that our 'cost' or 'error function' is +1 (so actually a benefit) if we hit the target, and –1 otherwise (so want to maximize the benefit, same as minimizing the error function cost).

If we let y be the output value of the perceptron, x_1, x_2, ..., x_n be the input layer nodes; and w_1, w_2, ..., w_n the weights associated with the input nodes, so the output $y= w_1x_1+ w_2 x_2$ or generally [w] • [x] (the inner production of the weight matrix and the x vector). If the activation y is greater then a threshold θ, then we consider the output of the perceptron to be T=1 ('yes' or true); if otherwise, output is T=-1 ('no' or false); the 'true' y value for the threshold is y^*. So, in fact in this special case the input is in the form of a *pattern pair* (x,t), where t=+1 or t=-1 (is or is not 'a tank' or 'a circle').

Now we can write the perceptron learning algorithm more compactly as follows. We then give a simple geometric interpretation and run through an example and a demo.

Perceptron training algorithm

```
1. Select a pattern pair X,T from the training set
2. Calculate Y by applying W to X (W • X)
3. IF      Y >= θ and T = 1,    goto 1
   OR      Y <  θ and T = -1,   goto 1
   ELSE    IF    T = 1   $W_{(new)} = W_{(old)} + X * eta$, goto 1
           IF    T = -1  $W_{(new)} = W_{(old)} – X * eta$, goto 1
```

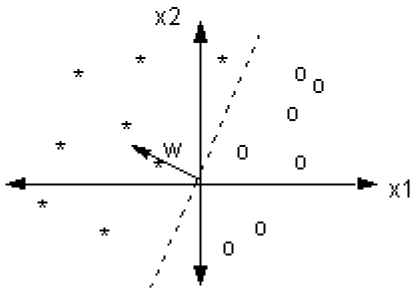
Note for the basic Perceptron Algorithm eta = 1 – i.e. the weight change is +/- X

What the algorithm does.
Calculate the output as the inner product of the weights and the Input
If the output is correct it takes no action
If it is incorrect it adjusts the weights so that the error between the required and computed result is decreased..
Where a positive result is required i.e. in set P it adds the input vector to the weights
Where a negative result is required i.e. in set N it subtracts the input vector from the weights

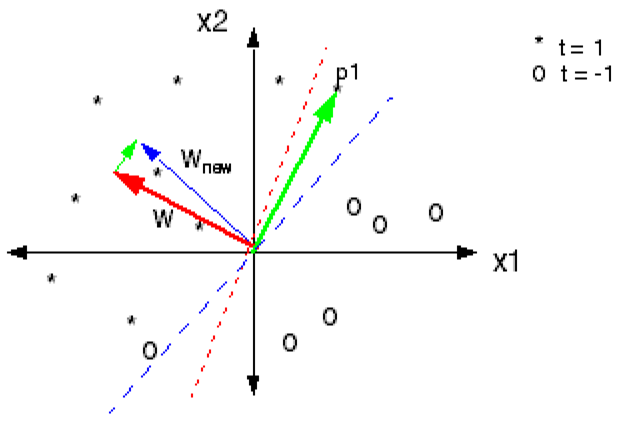
Weight change is only applied to weights for which there was positive input - we ignore zero inputs - these contributed nothing to the activation and therefore nothing to the error…

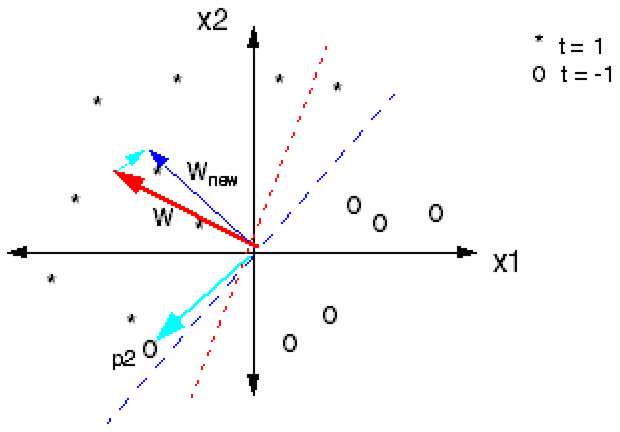
Geometric intuition/interpretation

We are really trying to find weights s.t. sign(W X) =T. We start with a *random* initial line, given the random weights. Now the weights (= a new line) must be chosen s.t. the projection of pattern X onto W has the same sign as the target T. But the boundary between the positive (above threshold t=1) and negative (below threshold, t=0) areas is just the line 0=[w•x] (plane, hyperplane). Thus, the way to picture the perceptron is as follows:



Now we can see how the weight update rule works to move our weights (line) to find the cut that divides the good guys from the bad guys. In the figure below, consider what happens when the training pattern p1 or p2 is chosen. Before updating the weights W, we note that both p1 and p2 are incorrectly classified by the current perceptron (the red dashed line is the true decision boundary). Suppose we choose p1 to update the weights as in picture below on the left. P1 has target value t=1. Then the first clause of the update rule is used, so that the weight is pushed a small amount in the direction of p1. Suppose we choose p2 to update the weights. P2 has target value t=-1 so the weight is pushed a small amount in the direction of -p2. In either case, the new boundary (blue dashed line) is better than before. (A remarkable result is that this algorithm is guaranteed to find a solution if one exists…but we don't have time to cover this – see Minsky & Papert, *Perceptrons*).





3. An example - Learning *AND*

Suppose we want to learn the AND predicate – a binary predicate, that has two inputs and outputs 1 if both inputs are 1, and 0 otherwise. Thus there are 8 possible inputs (and so two weights) and we know the target outputs, as described in the table below.

Input	Target
0 1	0
1 0	0
1 1	1
0 1	0
1 0	0
1 1	1
0 1	0
0 0	0

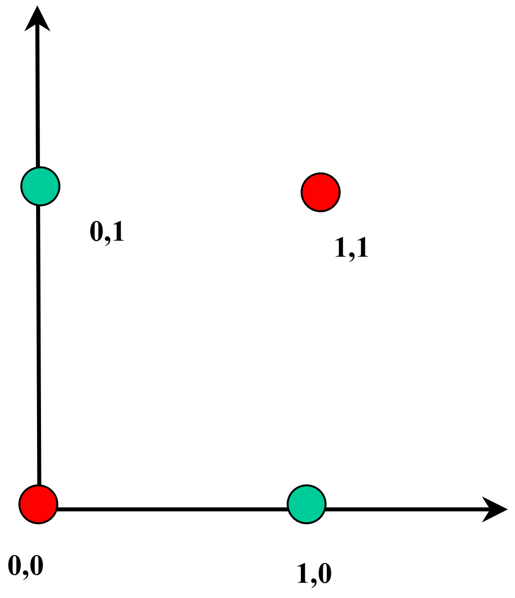
Now let us set up the table for perceptron learning. We will initialize the weights to –0.1 and 0.2. We have a threshold theta of 1. Given these weights, we can compute the activation via the forward computation of the network, and then see whether the perceptron with these weights works. Note that we can ignore input **00** during learning in this example as it contributes nothing to learning. We have left the computation of the 0,1 input to you. Also, we have placed in the last column whether that particular combination of target and weights requires any learning or not. The first one is done for you. The next to last column gives the **new** weights according to the perceptron learning algorithm. Thus, if the output value is already ok, we don't change this value. You fill in the rest of this column with 'OK' or 'change'. Now note **how we changed** the weights in row 3. The old weights were –0.1, +0.2. Since the activation was too **low,** following the algorithm we **increase** the weights by +1 each, to get 0.9, 1.2.

Input	Weights	Activation	θ	Output	Target	Weights	Learn?
0 1	-0.1 +0.2	0.2	1	0	0	-0.1 +0.2	ok
1 0	-0.1 +0.2	-0.1	1	0	0		
1 1	-0.1 +0.2	0.1	1	***0***	***1***	***+0.9 +1.2***	change
0 1	+0.9 +1.2	1.2	1				
1 0	+0.9 +0.2	0.9	1	0	0		
1 1	+0.9 +0.2	1.1	1	1	1		
0 1	+0.9 +0.2	0.2	1	0	0		
0 0	+0.9 +0.2	0	1	0	0		

This gives only **one** iteration, or 'epoch' of the algorithm. Now we have to go through it again! - with the new weights (which ones?), and recomputed. So you can see that even here, convergence might take a while. It's best to see it with a demo. (Demo).
What happens if we start with a different set of initial weights? Well, of course, convergence could be vastly different.

Note again that the weight correction is so easy, because the change in the output, y, with respect to x is just the derivative of *wx,* or simply the weight *w.*

An example: XOR
However, there are many functions perceptrons cannot compute. Since [**w** x] determines only lines, planes, hyperplanes, there are some surfaces that it cannot distinguish. A perceptron can only distinguish linearly separable objects, or linear combinations of these. A famous one that cannot be so distinguished is exclusive or. (XOR): a single line obviously cannot cut between the two sets of points, nor could a conjunction of them.



How do we solve such a problem? Well, one way is to **add** extra hidden layers between inputs and outputs. But then we have to change our learning algorithm

What has to change to go to a more general, multi-layer system? Much remains the same. We'll still try to minimize our error, figure out the change in error with respect to change in weights, and use gradient descent to do this, but:

1. **The loss function E.** We need a different cost function or error function E. Instead of +/- 1, we need the sum of the distances between the true values *y** (or *t*) and the computed activation values. Suppose we have *p* examples The usual form here for E is then the following, the sum squared error:

$$E = \frac{1}{2}\sum_p (t_p - y_p)^2$$

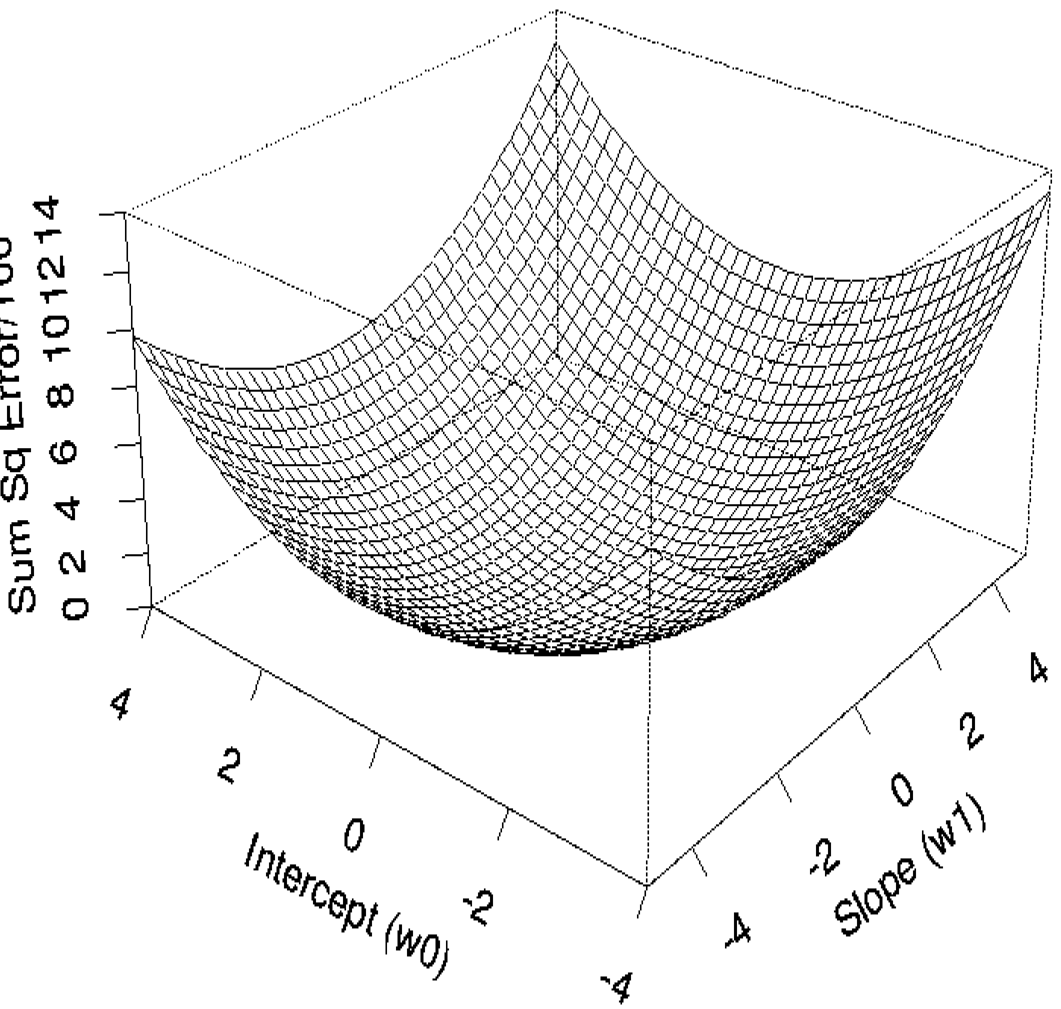
In other words, E is the sum over all points *i* in our data set of the squared difference between the **target** value t_i and the model's prediction y_i, calculated from the input value x_i . For a linear model, the sum-squared error is a quadratic function of the model parameters. See the figure below for what this looks like for a range of two weights.

2. **Minimizing the loss.** Since our basic idea is to figure out how to make the error smaller and smaller by changing the weights, we will still want to know the change in the error with respect to the weights, i.e., the partial derivative of E with respect to W. This is the method of **gradient descent:**

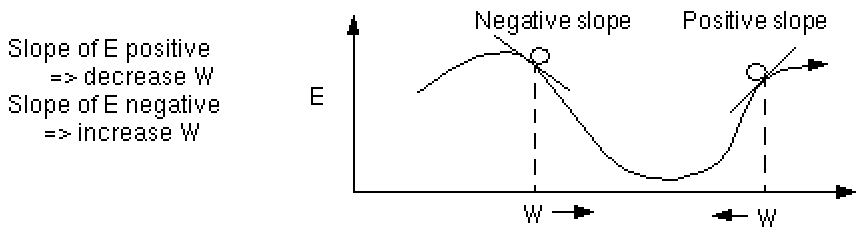
- Choose some (random) initial values for the model parameters.
- Calculate the gradient G of the error function with respect to each model parameter.
- Change the model parameters so that we move a short distance in the direction of the greatest rate of decrease of the error, i.e., in the direction of -G.
- Repeat steps 2 and 3 until G gets close to zero.

3. **Relating loss to weight change.** We need a way to relate the errors to changes we need to make in the weights. This is easy with perceptrons, but with multilayer nets, this is harder. Since there are no target activations for the hidden units, the perceptron learning rule does not extend to multilayer networks, The problem of how to train the hidden-unit weights is an acute problem of credit assignment. How can we determine the extent to which hidden-unit weights contribute to the error at the *output*, when there is not a

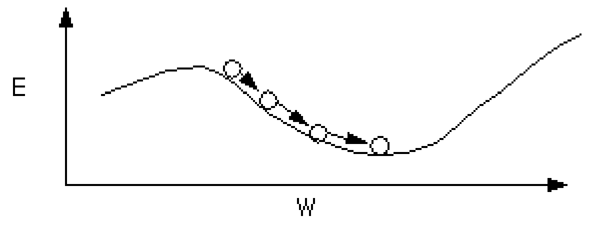
direct error signal for these units? The BackProp algorithm provides a solution to this credit assignment problem.



Before we plunge ahead and show how to compute the weight change in a multilayer network, how does gradient descent work? The gradient of E gives us the direction in which the loss function at the current settting of the *w* has the steepest **slope**. In order to decrease *E*, we take a small step in the opposite direction, -*G.* Here is a picture.



By repeating this over and over, we move "downhill" in *E* until we reach a minimum, where *G* = 0, so that no further progress is possible:



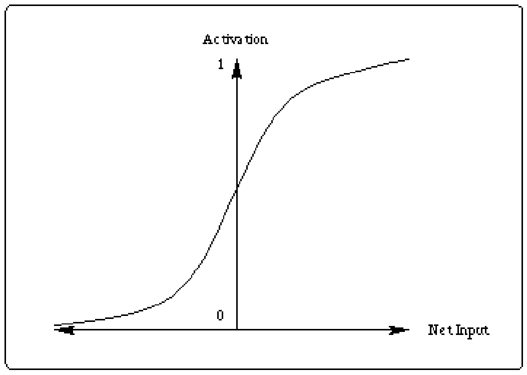
When a BackProp network is cycled, the activations of the input units are propagated forward to the output layer through the connecting weights. Like the perceptron, the net input to a unit is determined by the weighted sum of its inputs, where *net* is what we called the output *y* for a perceptron:

$$net_j = \sum_i w_{ji} a_i$$

where a_i is the input activation from unit i and w_{ji} is the weight connecting unit i to unit j. However, instead of calculating a binary output, the net input is added to the unit's bias θ and the resulting value is passed through a sigmoid function:

$$f(net_j) = 1/(1 + e^{-net_j + \theta_j}).$$

The bias term plays the same role as the threshold in the perceptron. But unlike binary output of the perceptron, the output of a sigmoid is a continuous real value between 0 and 1. The sigmoid function is sometimes called a "squashing" function because it maps its inputs onto a fixed rang



Learning in a backpropagation network is in two steps, the first as before, and the second as before except for the complication in the definition of the error function, and the way to change weights.

A sigmoid function can be differentiated to get the slope dy/dz = y(1-y).

Using this slope to adjust the weights gives:

$\Delta w = -rx(y - y^*)y(1 - y)$ Change in weight = - Learning Rate * Input * Error * Slope

Step 1, Forward activation: Each target pattern $\mathbf{t_p}$ is presented to the network and its inputs propagated forward to the output.

Step 2, Change weights. A method called gradient descent is used to minimize the total error on the patterns in the training set. In gradient descent, weights are changed in proportion to the negative of an error derivative with respect to each weight:

$$\Delta w_{ji} = -\varepsilon\, [\delta E / \delta w_{ji}]$$

Weights move in the direction of steepest descent on the error surface defined by the total error (summed across patterns):

$$E = 1/2 \sum_p \sum_j (t_{pj} - o_{pj})^2$$

where $\mathbf{o_{pj}}$ is the activation of output unit $\mathbf{u_j}$ in response to pattern **p** and $\mathbf{t_{pj}}$ is the true target output value for unit $\mathbf{u_j}$ The figure below (like the one earlier) illustrates the concept of gradient descent using a single weight. After the error on each pattern is computed, each weight is adjusted in proportion to the calculated error gradient backpropagated from the outputs to the inputs. The changes in the weights reduce the overall error in the network.

The backpropagation learning algorithm can be summarized as follows. During learning, the weight on each connection are changed by an amount that is proportional to an error signal **d**. Using gradient descent on the error E_p, the weight change for the weight connecting unit $\mathbf{u}_i$ to $\mathbf{u}_j$ is given by

$$\Delta_p w_{ji} = \varepsilon d_{pj} a_{pj}$$

where ε is the learning rate. When $\mathbf{u}_j$ is an output unit, then the error signal $\mathbf{d}_{pj}$ is given by case 1 (the base case). Otherwise $\mathbf{d}_{pj}$ is determined recursively according to case 2. We describe these cases in detail below.

Case 1. Sigmoid learning for an output node:

A sigmoid function can be differentiated to get the slope dy/dz = y(1-y).
Using this slope to adjust the weights gives:

$\Delta w = -rx(y - y^*)y(1-y)$ Change in weight = - Learning Rate * Input * Error * Slope

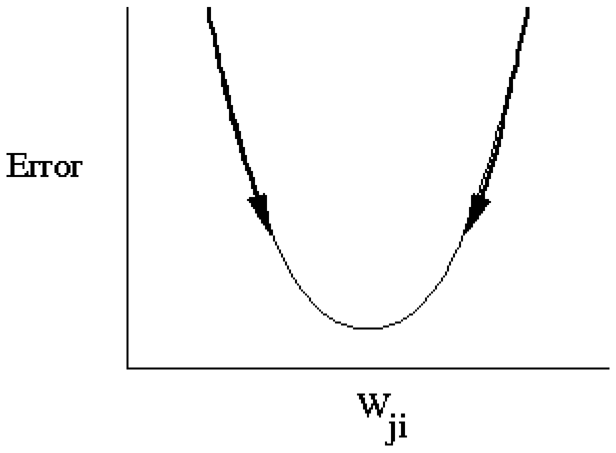
Case 2. Sigmoid learning for a hidden (non-output) node:

The difference for hidden nodes is that error is defined in terms of the error of the later nodes:

$\Delta w_{ij} = -rx\delta_j$ Change in weight = - Learning Rate * Input * Delta

For output nodes, delta is the slope times the error.
For hidden nodes, delta is the slope times the sum of the weighted deltas of the <u>next</u> layer of nodes.

$$\delta_j = \begin{cases} y_j(1-y_j)(y_j - y_j^*) & \text{If node i is an output unit} \\ y_j(1-y_j)\sum_k \delta_k w_{jk} & \text{If node i is not an output} \end{cases}$$



Since the Error *does not* connect directly to the weights, we have to break down the connection into two parts: (1) the link from the Error E to changes in the net input; and (2) changes in the net input relative to changes in the weights. That will tell us how changes in weights affect Error, which is what we need. This is what the BackProp algorithm does.

One more aside. If there is just **one** layer, then the combination of the weights is again just a linear combination as in perceptrons – the only difference is that we have a smooth output function, and a different error function E. But the update rule is easy to write, because taking the derivatives of linear functions is easy (mu is the 'learning rate').

$$W(new) = W(old) - \mu \frac{\partial E}{\partial W}$$

$$E = \frac{1}{2} \sum_{i=1}^{n} (t_i - y_i)^2$$

$$\frac{\partial E}{\partial W} = \sum_{i=1}^{n} (t_i - y_i)\, x_i$$

But with more than one layer, we cannot get away with this simple a method.
In order to derive the BackProp learning rule, we use the chain rule to rewrite the error gradient for each pattern as the product of **two** partial derivatives. The first partial derivative reflects the change in error as a function of the net input; the second partial derivative reflects the effect of a weight change on a change in the net input.

Thus, the error gradient becomes

$$E_p / \delta w_{ji} = [\delta E_p / \delta net_{pj}]\ [\delta net_{pj} / \delta w_{ji}]$$

Since we know what the equation for the net input to a unit is, we can calculate the second partial derivative directly:

$$\delta net_{pj} / \delta w_{ji} = \delta(\Sigma_k w_{jk} o_{pk}) / \delta w_{ji} = o_{pi}.$$

We will call the negative of the first partial derivative the <u>error signal</u>:

$$\mathbf{d_{pj}} = - \delta E_p / \delta net_{pj}.$$

Thus, the appropriate change in the weight w_{ij} with respect to the error $\mathbf{E_p}$ can be written as

$$\Delta_p w_{ji} = \varepsilon \mathbf{d}_{pj} o_{pj}$$

In this equation, the parameter ε is called the learning rate.

The next (and final) task in the derivation of the BackProp learning rule is to determine what $\mathbf{d}_{pj}$ should be for each unit it the network. It turns out that there is a simple recursive computation of the **d** terms which can be calculated by backpropagating the error from the outputs to the inputs. However, to compute $\mathbf{d}_{pj}$ we need to again apply the chain rule. The error derivative

$$\mathbf{d_{pj}}$$

can be rewritten as the product of two partial derivatives:

$$\mathbf{d}_{pj} = - (\delta E_p / \delta o_{pj}) (\delta o_{pj} / \delta net_{pj})$$

Consider the calculation of the second factor first. Since

$$o_{pj} = f(net_{pj}),$$
$$\delta o_{pj} / \delta net_{pj} = f'(net_{pj}).$$

The derivative of the sigmoid function has an elegant derivation, yielding:

$$f'(net_{pj}) = f(net_{pj})(1 - f(net_{pj}))$$

To calculate the first partial derivative there are two cases to consider.

Case 1: Suppose that $\mathbf{u_j}$ is an output unit of the network, then it follows directly from the definition of $\mathbf{E_p}$ that

$$\delta E_p / \delta o_{pj} = -2(t_{pj} - o_{pj})$$

If we substitute this back into the equation for $\mathbf{d}_{pj}$ we obtain

$$\mathbf{d}_{pj} = 2(t_{pj} - o_{pj}) f'(net_{pj})$$

Case 2: Suppose that $\mathbf{u_j}$ is not an output unit of the network, then we again use the chain rule to write

$$E_p / \delta o_{pj} = \Sigma_k [\delta E_p / \delta net_{pk}] [\delta net_{pk} / \delta o_{pj}]$$

$$= \Sigma_k [\delta E_p / \delta net_{pk}] [\delta(\Sigma_i w_{ki} o_{pi}) / \delta o_{pj}]$$

$$= \Sigma_k [\delta E_p / \delta net_{pk}] w_{kj}$$

$$= -\Sigma_k d_{pk} w_{kj}$$

What's next? What can these beasts learn?

Now that we waded through all of the details of the backpropagation learning equations, let us consider how we should choose the initial weights for our network. Suppose that all of the weights start out at equal values. If the solution to the problem requires that the network learn unequal weights, then having equal weights to start with will prevent the network from learning. To see why this is the case, recall that the error backpropagated through the network is proportional to the value of the weights. If all the weights are the same, then the backpropaged errors will be the same, and consequently all of the weights will be updated by the same amount. To avoid this symmetry problem, the initial weights to the network should be unequal.

Another issue to consider is the magnitude of the initial weights. In most programs, all of the network weights and biases are initialized to small random values. What advantage is there in randomized the weights to **small** values? Reconsider the derivative of the sigmoid activation function. The derivative of the sigmoid is

$f'(net_{pj}) = f(net_{pj})(1 - f(net_{pj}))$

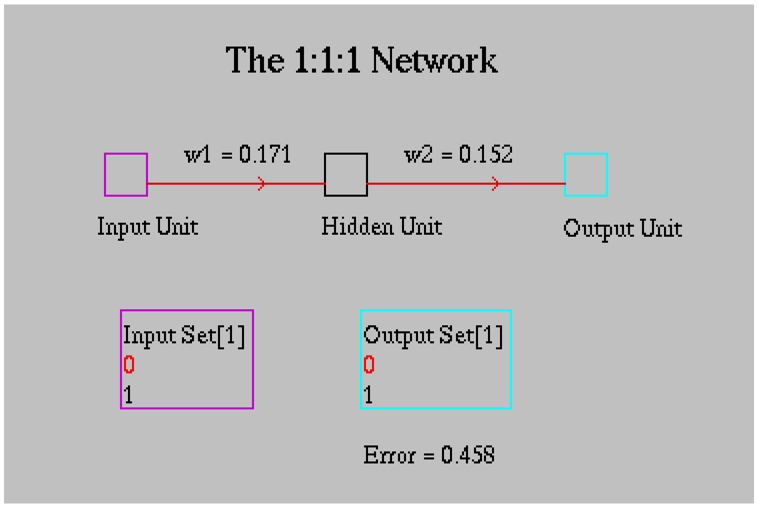
Since the weights updates in the BackProp algorithm are proportional to this derivative, it is important to consider how the net input affects its value. The derivative is a maximum when $f(net_{pj})$ is equal to 0.5 and aproaches its minimum as $f(net_{pj})$ approaches 0 or 1. Thus, weights will be changed most for unit activations closest to 0.5 (the steepest portion of the sigmoid). Once a unit's activation becomes close to 0 or 1, it is committed to being on or off, and its associated weights will change very little. It is therefore important to select small initial weights so that all of the units are uncommitted (having activations that are all close to 0.5 - the point of maximal weight change)

Local Minima

Since Backprop uses a gradient-descent procedure, a BackProp network follows the contour of an error surface with weight updates moving it in the direction of steepest descent. For simple two-layer networks (without a hidden layer), the error surface is bowl shaped and using gradient-descent to minimize error is not a problem; the network will always find an errorless solution (at the bottom of the bowl). Such errorless solutions are called **global minima**. However, when an extra hidden layer is added to solve more difficult problems, the possibility arises for complex error surfaces which contain many minima. Since some minima are deeper than others, it is possible that gradient descent will not find a global minima. Instead, the network may fall into **local minima** which represent suboptimal solutions.

Obviously, we would like to avoid local minima when training a BackProp network. In some case this may be difficult to do. However, in practice it is important to try to assess how frequently and under what conditions local minima occur, and to examine possible strategies for avoiding them. As a general rule of thumb, the more hidden units you have in a network the less likely you are to encounter a local minimum during training. Although additional hidden units increase the complexity of the error surface, the extra dimensionalilty increases the number of possible escape routes.

The concept of a local minimum can be demonstrated with a very simple network consisting of one input unit, one hidden unit, and one output unit below. The task of this network, called the 1:1:1 network by McClelland and Rumelhart (1988), is simply to copy the value on the input unit to the output unit.


The 1:1:1 Network
w1 = 0.171
w2 = 0.152
Input Unit
Hidden Unit
Output Unit
Input Set[1]
0
1
Output Set[1]
0
1
Error = 0.458