CMPT 882 Machine Learning, 2004-1

Week 5 Lecture Notes

Instructor: Dr. Oliver Schulte

Scribe: Sarah Brown and Angie Zhang

February 3rd and 5th

Contents

1. Artificial Neural Networks

1.1 Overview

This section presumes familiarity with some basic concepts of Artificial Neural Network (ANN), since we have covered some of the materials on the previous lectures. This starts from Gradient Descent and the Delta Rule and continues to Backpropagation. In addition some of the benefits of the various techniques as well as the drawbacks are discussed.

1.2 Gradient Descent and the Delta Rule

1.2.1 Overview of Gradient Descent and the Delta Rule

From the previous section, we know that the perceptron rule can find a successful weight vector when the training examples are linearly separable. It can fail to converge if the examples are not linearly separable. The delta rule is used to overcome this difficulty.

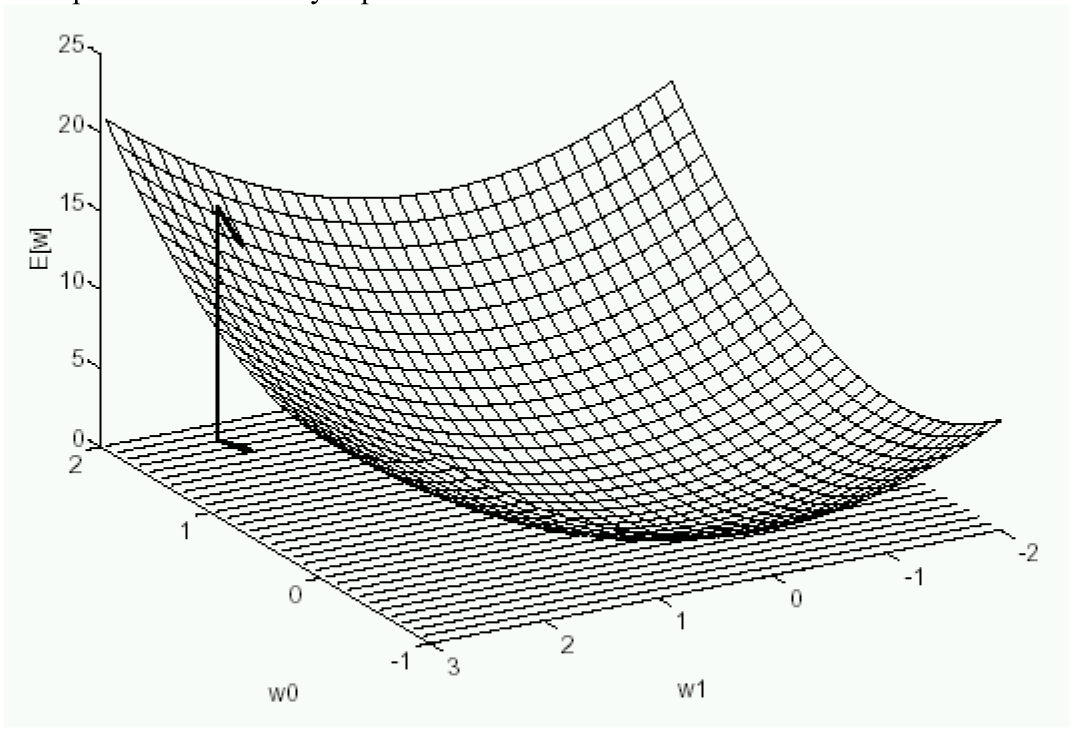


Figure 1.2.1

Figure 1.2.1 is the error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

Perceptron training rule guaranteed to converge to 0-error hypothesis after finite number of iterations if

- ❖ Training examples are linearly separable (i.e., no noise)
- ❖ Sufficiently small learning rate η

The advantages of Delta Rule over Perceptron Training Rule include:

- ❖ Guaranteed to always converge to a hypothesis with minimum squared error (with a small learning rate)
- ❖ Allows for noise in the data
- ❖ Allows for non-separable functions

The delta training rule is best understood as training an unthresholded perceptron, which is a linear unit with output o given as follows:

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

Training error of a hypothesis relative to the training examples is as follows:

$$E(w) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad (1.2.1)$$

- D : set of training examples,
- t_d : target output for training example d
- o_d: linear unit output for training example d, $o_d = o(\vec{w}^{(d)}) = \vec{w} \cdot \vec{x}^{(d)}$
- $E(\vec{w})$: half of squared different between target output t_d and linear unit output o_d, summed over all training examples

Let's consider a simpler linear unit
$o_{Linear}(\vec{x}) = w_0 + w_1 x_1 + ... + w_n x_n$ (No threshold!)

Learn w_i's that minimize squared error $E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$,

As $o_d = \vec{w} \cdot \vec{x}^{(d)}$ is linear in w_i, E[$\vec{w}$] is a quadratic formula, so the error surface has a single minimum. Therefore, Gradient Descent works well to locate the minimum.

1.2.2 Derivation of the Gradient Descent Rule

To calculate the direction of steepest descent along the error surface, we have to compute the derivative of E with respect to each component of the vector $\vec{w}$. This vector derivative is called the gradient of E with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n} \right],$$

$\frac{\partial E}{\partial w_0}$ is the partial derivative of the error E with respect to a single weight component w_0 that is in $\vec{w}$. Notice that $\nabla E(\vec{w})$ is itself a vector. The gradient specifies the direction of steepest increase of E, the training rule for gradient descent is: $\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$, where $\Delta\vec{w} = -\eta\nabla E(\vec{w})$, η is the learning rate (a positive constant). We put a negative sign before the learning rate since we want to move the weight vector in the direction that decreases E. The component form of this training rule is: $w_i \leftarrow w_i + \Delta w_i$, where

$$\Delta w_i = -\eta\nabla E(w_i) = -\eta\frac{\partial E}{\partial w_i} \qquad (1.2.2a)$$

since we can write $\nabla E(\vec{w}) \equiv \frac{\partial E}{\partial w_i}$

The gradient can be obtained by differentiating E from Equation (1.2.1):

(1.2.2b)

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}(\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2)\\ &= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2\\ &= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)\\ &= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d)\\ \frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{i,d})\end{aligned}$$

$x_{i,d}$ denotes the single input component x_i for training example d.
After substituting Equation (1.2.2b) into (1.2.2a), we get the weight update rule for gradient descent:

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} = \eta\sum_{d\in D}(t_d - o_d)(x_{i,d})$$

The basis of the delta rule is to change weight in the opposite direction of gradient, which is the shortest and easiest way to decrease approximation error. For each step, the changing rate is decided by the learning rate η. If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. This can be likened to driving your car down an unfamiliar street looking for a house. If you drive too fast, η is too large, you could drive past the house you are looking for and have to turn around. Likewise, if you drive too slowly, η is too small, it might take you a very long time to find your destination.

One common modification to the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows. This can serve to prevent passing the minima and having to "come back down the hill".

Here is the complete Gradient-Descent algorithm for training a linear unit.

```
Gradient-Descent(training_examples, $\eta$)
        Each training example is a pair of the form <$\vec{x}$, t>, such that $\vec{x}$ is the vector of
        input values and t is the target output.  $\eta$ is the learning rate (e.g. 0.05).

                Initialize each $w_i$ to small random value (Typically $\in$ [-0.05, +0.05])
                Until termination condition is met, Do
                        Initialize each $\Delta w_i$ to 0
                        For each <$\vec{x}$, t> $\in$ D, Do
                                Compute o = o($\vec{x}$) using current $\vec{w}$
                                For each linear unit weight $w_i$, Do
                                        $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
                        For each linear unit weight $w_i$, Do
                                        $w_i \leftarrow w_i + \Delta w_i$
                Return $\vec{w}$
```

In summary, the algorithm is as follows:

Pick an initial random weight for each w_i in the weight vector

Apply the linear unit to all training examples and compute Δw_i for each weight according to $\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{i,d}$

Update each weight w_i by adding Δw_i

Repeat this process

1.2.3 Stochastic Approximation to Gradient Descent

The difficulties in applying gradient descent are:

- ❖ Converging to a local minimum can sometimes be quite slow and can require many thousands of gradient descent steps.
- ❖ No guarantee that the procedure will find the global minimum if there exists multiple local minima in the error surface.

One way to try to solve these problems is to use incremental gradient descent or stochastic gradient descent. Instead of updating the weights after summing over all training examples, stochastic gradient descent updates the weights incrementally, following the calculation of the error for each training example.

The training rule now becomes: $\Delta w_i = \eta(t-o)x_i$

where

t: target value

o: unit output

x_i: ith input for the training example

η: learning rate

The algorithm for stochastic gradient descent is nearly identical to standard gradient descent.

Here is the complete Stochastic Gradient-Descent algorithm for training a linear unit.

Stochastic-Gradient-Descent(training_examples, η)

Each training example is a pair of the form $<\vec{x}$, t> such that $\vec{x}$ is the vector of input values and t is the target output. η is the learning rate (e.g. 0.05).

```
Initialize each $w_i$ to small random values
Until termination condition is met, Do
    Initialize each $\Delta w_i$ to 0
    For each $<\vec{x}$, t> in traing_examples, Do
        Input the instance $\vec{x}$ to the unit and compute output o
        For each linear unit weight $w_i$, Do
            $w_i \leftarrow w_i + \eta(t-o)x_i$
```

The key differences between standard gradient descent and stochastic gradient descent are:

- In Standard gradient descent, the error is summed over all examples before updating weight. In Stochastic gradient descent, weights are updated upon examining each training example.
- Standard gradient descent requires more computation per weight update step for summing over multiple examples. Standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent since it uses the true gradient.
- In cases where there are multiple local minima with respect to $E(\vec{w})$,stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search.

1.3 Multilayer Networks and the Backpropagation Algorithm

1.3.1 Multilayer Networks

Perceptrons are great if we want single straight surface. If we have a nonlinear decision surface, we have to use multilayer network. For example, in Figure 1.3.1a, the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h_d". The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network.

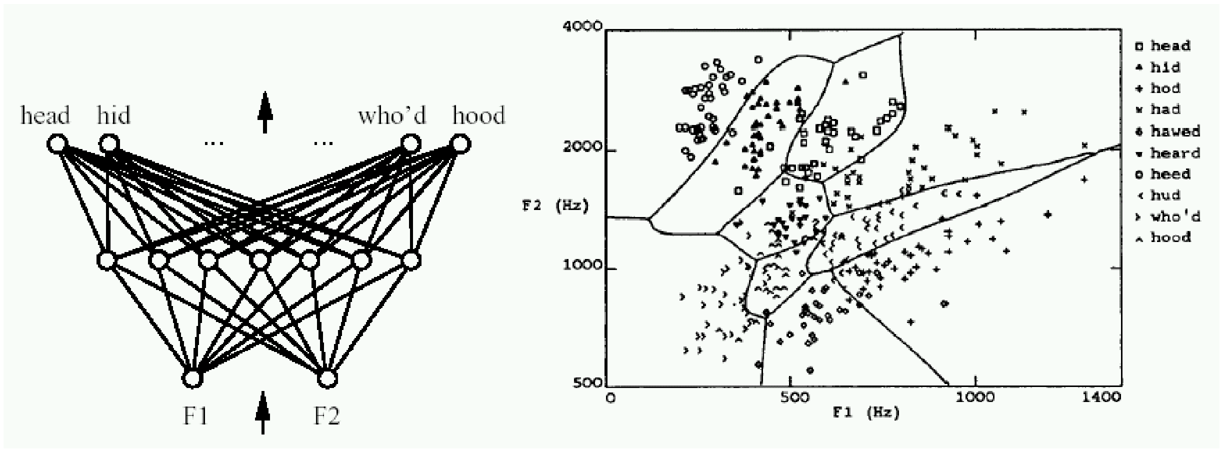


Figure 1.3.1a

Using multiple layers of linear units still only produces linear functions. One solution is the sigmoid unit, which is a unit similar to a perceptron, but based on a smooth, differentiable threshold function. The output of sigmoid unit is a nonlinear function of its inputs, but it is still differentiable.

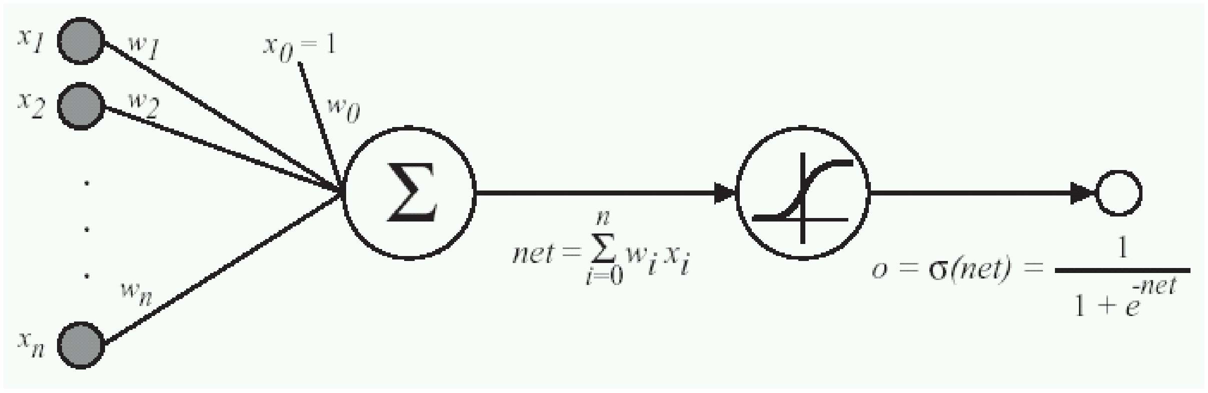

x_1
w_1
x_2
w_2
$x_0 = 1$
w_0
x_n
w_n
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Figure 1.3.1b

Figure 1.3.1b is the sigmoid threshold unit. The sigmoid unit computes the linear combination of its inputs then applies a threshold to the result, where the output is a continuous function of its input. The output o of the sigmoid unit is:

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

σ is the sigmoid function and its output ranges between 0 and 1. The sigmoid function has the nice property that its derivative is easy to compute: in particular, $\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$, other differentiable functions can be used in place of σ (such as tanh).

The sigmoid function has very important properties, including:

- It outputs real numbers between 0 and 1
- It maps a very large input domain to a small range of outputs
- It never loses information because it is a one to one function
- It increases monotonically

We can derive gradient decent rules to train:
- One sigmoid unit
- Multilayer networks of sigmoid units $\rightarrow$ Backpropagation

Error Gradient for a Sigmoid Unit:

How does the error of the output unit depend on a specific weight?

$$\frac{\partial E}{\partial w_i} = (o-t)(\frac{\partial o}{\partial w_i}) = (o-t)\frac{\partial \sigma(y)}{\partial y}\frac{\partial y}{\partial w_i}$$

where o is the output value, t is the target value, and $y = w_i x_i$.

Since we know that y = f(w) and o = g(y), f is the activation function and g is the sigmoid function. we can write the following equation:

$$\frac{\partial o}{\partial w_i} = \frac{\partial o}{\partial y}\frac{\partial y}{\partial w_i} = \frac{\partial \sigma(y)}{\partial y}\frac{\partial y}{\partial w_i}$$

The greater the number of training iterations, the lower the error will be for the training set. We can try to learn the architecture within the fixed weight.

The following is to show how we get $\frac{\partial E}{\partial w_i} = -\sum_{d \in D.} (t_d - o_d) o_d (1 - o_d) x_{i,d}$.

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d \in D.}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d.}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d.} 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d.}(t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\
&= -\sum_{d.}(t_d - o_d)\frac{\partial o_d}{\partial net_d}\frac{\partial net_d}{\partial w_i}
\end{aligned}$$

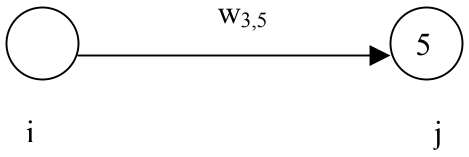
But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1-o_d), \text{ and } \frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D.} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Error Gradient for Network



Let $\delta_5 = \frac{\partial E}{\partial y_5}$, how does the error depend on y5?

$$\frac{\partial E(<\vec{x}, \vec{t}>)}{\partial \sigma(y_5)} = (o_5 - t_5)$$

$$\frac{\partial E}{\partial w_{3,5}} = \frac{\partial E}{\partial y_5} \cdot \frac{\partial y_5}{\partial w_{3,5}} = \delta_5 \cdot o_3$$

If we get an error at the output, we should propagate back until hit the input nodes, updating the weights. (Provided our termination condition as not been met)

If our training result can fit the data perfectly, it does not mean it generalize well. With Neural Network, we don't know hot to generalize it. If we have lots of data, which contain some noses, even the training result can fit the data very well, but it may not minimize errors on the whole distribution.

Gradient descent is a very general method; it cannot only apply to Neural Network, but also can apply to other methods, such as decision tree.

1.3.2 The Backpropagation Algorithm

The backpropagation algorithm learns the weights for a multilayer network given a fixed network of units and interconnections by employing gradient descent to minimize the squared error between the network output and target values for those outputs.

Because these networks have multiple output units, E must be redefined to sum errors over all network output units:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D.} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

outputs: set of output units in the network
t_{kd}: target output values associated with the kth output unit and training example d
o_{kd}; output values associated with the kth output unit and training example d

In multilayer networks, there can be multiple local minima and backpropagation is not guaranteed to converge to a global minima.

Backpropagation Algorithm

```
Backpropagation(training_examples, $\eta$, $n_{in}$, $n_{out}$, $n_{hidden}$)
    Training example: $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is vector of network inputs, $\vec{t}$ is vector of
    target output values
    $\eta$: learning rate (e.g. 0.05)
    $n_{in}$: number of network inputs
    $n_{hidden}$: number of units in the hidden layer
    $n_{out}$: number of output units
    $x_{ji}$: input from unit i to unit j
    $w_{ji}$: weight from unit i to unit j

•   Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output
    units
•   Initialize all network weights to a small random number (e.g. between -.05 and
    .05)
•   Until termination condition is met, Do:
        For each $\langle \vec{x}, \vec{t} \rangle$ in training examples, Do
            Propagate the input forward through the network:
            1. Input the instance $\vec{x}$ to the network and compute the output $o_u$
               of every unit u in the network
            Propagate the errors backward through the network:
            2. For each network output unit k, calculate its error term $\delta_k$
                   $\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$

            3. For each hidden unit h, calculate its error term $\delta_h$

                   $\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{kh}\delta_k$
            4. Update each network weight $w_{ji}$

                       $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$

                   where
                       $\Delta w_{ji} = \eta \bullet \delta_j \bullet x_{ji}$
```

This algorithm applies to layered networks containing two layers of sigmoid units with units at each layer connected to all units from the previous layer. This is the incremental gradient descent version of backpropagation. It iterates over the training examples repeatedly, calculating the error and updating the weights often thousands of times until the network performs well.

The gradient descent weight-update rule, $\Delta w_{ji} = \eta \bullet \delta_j \bullet x_{ji}$, is similar to the delta training rule except that the error (t-o) in the delta rule is replaced by the error term

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

Calculating the error term for hidden units is similar. However, since training examples only provide target output for the network, the error term for hidden units is calculated by summing the error terms δ_k for each output influenced by hidden unit h. Each of these error terms is multiplied by the weight from the hidden unit h to output unit k which characterizes the degree to which the hidden unit h is "responsible for" the error in k. This is essentially the credit assignment problem.

$Credit(wi) = \partial f \frac{\partial E}{\partial w_i}$ How much of the error depends on an individual weight, w_i?

Halting Backpropagation Options

- Stop after a certain number of iterations. If the number of iterations is too low, the error may not be reduced sufficiently. Too much iteration can lead to overfitting the training data.
- Error on training examples falls below some threshold
- Error on a separate validation set of examples meets some criteria

1.3.2.1 Adding Momentum

The addition of momentum is a variation on the standard backpropagation algorithm, which has several positive properties:

- It can prevent the algorithm from becoming trapped in local minima
- It prevent the algorithm from stalling on portions of the error surface with no gradient
- In regions where the gradient is not changing, it can speed up convergence by increasing the step size

The alteration of the algorithm makes the weight updates partially dependent on the update that occurred in the previous iteration. The weight update rule then becomes:

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

$\Delta w_{ji}(n)$: weight update performed during the nth iteration through the main loop of the algorithm
α: $0 \leq \alpha \leq 1$, momentum
$\Delta w_{ji}(n-1)$: the weight update that occurred during the (n-1)th iteration

1.3.2.2 Learning in Arbitrary Acyclic Networks

The backpropagation algorithm can be easily extended to feedforward networks of an arbitrary depth. The change in the weight update rule is such that when computing δ

values, the δ_r value for a unit r in a layer m is computed from the δ at the next deeper layer m+1 according to

$$\delta_r = o_r(1-o_r) \sum_{s \in layer\_m+1} w_{sr}\delta_s$$

This step may be repeated for any number of hidden layers in the network. Furthermore, it can be generalized to any acyclic network regardless of how the network units are arranged. The general rule for calculating δ for any internal (non-output) unit is

$$\delta_r = o_r(1-o_r) \sum_{s \in Downstream(r)} w_{sr}\delta_s$$

1.3.2.3 Comments and examples on Backpropagation:

- Gradient descent over entire network weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global, error minimum. (However, it does tend to perform well in practice by running multiple times)
- Often include weight momentum α in $\Delta w_{ji}(n) = \eta\delta_j x_{ji} + \alpha\Delta w_{ji}(n-1)$
- Minimizes error over training examples
- Training can take thousands of iterations $\rightarrow$ slow.
- Using network after training is very fast.

Let's work out exercise 4.7 to show how Backpropagation algorithm works.

Assume learning rate $\eta = 0.3$, momentum $\alpha = 0.9$, five weights $w_{ca} = w_{cb} = w_{c0} = w_{dc} = w_{d0} = 0.1$, the threshold input $x_0 = 1$, incremental weight updates, and the following training examples:

a	b	d
1	0	1
0	1	0

After the first training iteration of the BACKPROPAGATION algorithm:

a = 1, b = 0, d = 1

$o_c = \sigma(w_{c0} + w_{ca}*a + w_{cb}*b) = \sigma(0.1 + 0.1*1 + 0.1*0) = \sigma(0.2) = 1/(1+e^{-0.2}) = 0.54983$

$o_d = \sigma(w_{d0} + w_{dc}*c) = \sigma(0.1 + 0.1*0.54983) = \sigma(0.154983) = 1/(1+e^{-0.154983}) = 0.53866$

$\delta_d = o_d*(1-o_d)*(d-o_d) = 0.53866 * (1 - 0.53866) * (1 - 0.53866) = 0.11464$

$\delta_c = o_c*(1- o_c)*(w_{dc}*\delta_d) = 0.54983 * (1 - 0.54983) * (0.1 * 0.11464) = 0.00283$

$\Delta w_{ca}(1) = \eta*\delta_c*a + \alpha*\Delta w_{ca}(1-1) = 0.3*0.00283*1 + 0.9*0 = 0.00085$

$\Delta w_{cb}(1) = \eta*\delta_c*b + \alpha*\Delta w_{cb}(1-1) = 0.3*0.00283*0 + 0.9*0 = 0$

$\Delta w_{c0}(1) = \eta*\delta_c*x_0 + \alpha*\Delta w_{c0}(1-1) = 0.3*0.00283*1 + 0.9*0 = 0.00085$

$\Delta w_{dc}(1) = \eta*\delta_d*c + \alpha*\Delta w_{dc}(1-1) = 0.3*0.11464*0.54983 + 0.9*0 = 0.01891$

$\Delta w_{d0}(1) = \eta*\delta_d*x_0 + \alpha*\Delta w_{d0}(1-1) = 0.3*0.11464*1 + 0.9*0 = 0.03439$

Update each network weight:

$w_{ca} = w_{ca}+\Delta w_{ca} = 0.1 + 0.00085 = 0.10085$

$w_{cb} = w_{cb}+\Delta w_{cb} = 0.1 + 0 = 0.1$

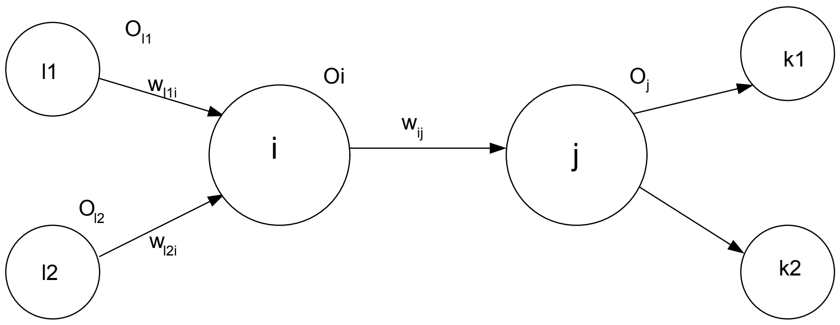
$w_{c0} = w_{c0}+\Delta w_{c0} = 0.1 + 0.00085 = 0.10085$

$w_{dc} = w_{dc} + \Delta w_{dc} = 0.1 + 0.01891 = 0.11891$
$w_{d0} = w_{d0} + \Delta w_{d0} = 0.1 + 0.03439 = 0.13439$

After the second training iteration of the BACKPROPAGATION algorithm:
$a = 0, b = 1, d = 0$
$o_c = \sigma(w_{c0} + w_{ca}*a + w_{cb}*b) = \sigma(0.10085 + 0.10085*0 + 0.1*1) = \sigma(0.20085)$
$= 1/(1+e^{-0.20085}) = 0.55004$
$o_d = \sigma(w_{d0} + w_{dc}*c) = \sigma(0.13439 + 0.11891*0.55004) = \sigma(0.19979) = 1/(1+e^{-0.19979}) =$
0.54978
$\delta_d = o_d*(1-o_d)*(d-o_d) = 0.54978 * (1 - 0.54978) * (1 - 0.54978) = -0.13608$
$\delta_c = o_c*(1- o_c)*(w_{dc}*\delta_d) = 0.55004 * (1 - 0.55004) * (0.11891 * (-0.13608)) = -0.004$
$\Delta w_{ca}(2) = \eta*\delta_c*a + \alpha*\Delta w_{ca}(2-1) = 0.3*(-0.004)*0 + 0.9*0.00085*1 = 0.00076$
$\Delta w_{cb}(2) = \eta*\delta_c*b + \alpha*\Delta w_{cb}(2-1) = 0.3*(-0.004)*1 + 0.9*0*1 = -0.0012$
$\Delta w_{c0}(2) = \eta*\delta_c*x_0 + \alpha*\Delta w_{c0}(2-1) = 0.3*(-0.004)*1 + 0.9*0.00085*1 = -0.00043$
$\Delta w_{dc}(2) = \eta*\delta_d*c + \alpha*\Delta w_{dc}(2-1) = 0.3*(-0.13608)*0.55004 + 0.9*0.01891 = -0.00543$
$\Delta w_{d0}(2) = \eta*\delta_d*x_0 + \alpha*\Delta w_{d0}(2-1) = 0.3*(-0.13608)*0 + 0.9*0.03439*1 = -0.00987$

Update each network weight:
$w_{ca} = w_{ca}+\Delta w_{ca} = 0.10085 + 0.00076 = 0.10161$
$w_{cb} = w_{cb}+\Delta w_{cb} = 0.1 + (-0.0012) = 0.0988$
$w_{c0} = w_{c0}+\Delta w_{c0} = 0.10085 + (-0.00043) = 0.10042$
$w_{dc} = w_{dc} +\Delta w_{dc} = 0.11891 + (-0.00543) = 0.11348$
$w_{d0} = w_{d0} +\Delta w_{d0} = 0.13439 + (-0.00987) = 0.12452$

1.3.3 Derivation of the Backpropagation Rule



Given a simple network such as the one above, it simply a lot of calculus to derive the weight update rule for the backpropagation algorithm.

Given that:
a_i: net activation of node i such that $a_i = \sum_{l_i \in parents\_of\_i} w_{li} \cdot o_l$
o_i: output of node i
$\frac{\partial E}{\partial w_{ij}}$: derivative of the error with respect to a weight

Based on the chain rule, we get:

$$\frac{\partial E}{\partial w_{ij}} = \overset{\textcircled{2}}{\frac{\partial E}{\partial a_j}} \cdot \overset{\textcircled{1}}{\frac{\partial a_j}{\partial w_{ij}}}$$

1. How does the activation of a_j depend on the weight from i to j?
   The activation of a_j is the sum of the weights of the parents of j multiplied by their individual outputs.
   $a_j = w_{ij} \bullet o_i + \ldots.$

   So

   $$\textcircled{1} \quad \frac{\partial a_j}{\partial w_{ij}} = o_i$$

2. Just as w_{ij} can only influence the network through the activation of a_j, the activation of a_j can only influence the network through the output o_j. By using the chain rule again, we get:

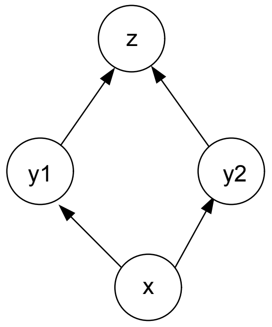
$$\textcircled{2} \quad \frac{\partial E}{\partial a_j} = \overset{\textcircled{4}}{\frac{\partial E}{\partial o_j}} \cdot \overset{\textcircled{3}}{\frac{\partial o_j}{\partial a_j}}$$

3. How does the output of j, o_j, depend on the activation of j?

   $o_j = \sigma(a_j)$ where σ is the sigmoid function

$$\textcircled{3} \quad \frac{\partial o_j}{\partial a_j} = o_j(1 - o_j)$$

4. How does the error in the network depend on the output of j?



Given a structure like the graph shown on the left, we could ask how much does the output of x, mediated through y1 and y2, affect the error at the output z?

The normal chain rule is:

$$\begin{tikzcd} x \arrow[r] & v \arrow[r] & z \end{tikzcd}$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x}$$ This is the chain rule for the above problem.

Using the same principle, we can calculate how the error in the network depends on the output of j:

$$\textcircled{4}\ \frac{\partial E}{\partial o_j} = \sum_{k_i \in children\_of\_j} \overset{\textcircled{6}}{\frac{\partial E}{\partial a_k}} \cdot \overset{\textcircled{5}}{\frac{\partial a_k}{\partial o_j}}$$

5. How does the activation of k depend on the output of j?
   Once again, the activation of k will be the sum of the weights of the parents of k to k multiplied by their outputs.

$$a_k = o_j \bullet w_{jk} + \ldots..$$

So, $\frac{\partial a_k}{\partial o_j} = w_{jk} \quad \textcircled{5}$

6. How does the error depend on the activation of k?

If k is hidden, we have already computed;
If k is output, we use the following equation:

$$\frac{\partial E}{\partial a_k} = (t_k - o_k)\frac{\partial o_k}{\partial a_k} = -(o_k - t_k)o_k(1 - o_k)$$

By making several substitutions, we can arrive at the training rule for backpropagation.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial a_j}{\partial w_{ij}} \cdot \frac{\partial E}{\partial a_j}$$

$$= \overset{\textcircled{1}}{o_i} \cdot \overset{\textcircled{2}}{\frac{\partial o_j}{\partial a_j}} \cdot \frac{\partial E}{\partial o_j}$$

$$= o_i \cdot \overset{\textcircled{3}}{o_j(1 - o_j)} \cdot \frac{\partial E}{\partial o_j}$$

$$= o_i \cdot o_j(1 - o_j) \cdot \overset{\textcircled{4}}{\sum_{k_i \in children\_of\_j} \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial o_j}}$$

$$= o_i \cdot o_j(1 - o_j) \cdot \sum_{k_i \in children\_of\_j} \overset{\textcircled{5}}{w_{jk}} \cdot \frac{\partial E}{\partial a_k}$$

Therefore, the update rule:

1. calculate $\frac{\partial E}{\partial a_j}$
2. set $\Delta w_{ij} = -\frac{\partial E}{\partial w_{ij}} \times \eta$

1.3.4 Efficiency

- ❖ For training the network, numbers of iterations are very important. If it is too small, the errors will be high. If it is too large, overfitting may occur, and cause errors go high.

- ❖ Learning: Intractable in general. Training can take thousands of iterations, which will be very slow. Learning net with single hidden unit is NP-hard.

- ❖ In practice, backpropagation is very useful.

- ❖ It will be very fast to classify examples using network after training.

- ❖ Advantages of Backpropagation
  - o Robust to noise in the data
  - o Handles continuous value
  - o Very general
- ❖ Very general in terms of "patterns"

- ❖ Disadvantages of Backpropagation
  - o No Human Interpretation
  - o Don't know how it generates

1.4 Convergence and Local Minima

Backpropagation is only guaranteed to converge to a local, and not a global, minima. However, since each weight in a network essentially corresponds to a different dimension in the error space, a local minimum with respect to one weight may not be a local minimum with respect to other weights. This can provide an "escape route" from becoming trapped in local minima.

If the weights are initialized to values close to zero, the sigmoid threshold function is approximately linear and so they produce linear outputs. As the weights grow, though, the network is able to represent more complex functions that are not linear in nature. It is the hope that by the time the weights are able to approximate the desired function that they will be close enough to the global minimum that even becoming stuck in a local minima will be acceptable.

Common heuristic methods to reduce the problem of local minima are:
- Add a momentum term to the weight-update rule

- Use stochastic gradient descent rather than true gradient descent
- Train multiple networks using the same training data but initialize the networks with different random weights. If the different networks lead to different local minima, choose the network that performs best on a validation set of data or all networks can be kept and treated as a committee whose output is the (possibly weighted) average of individual network outputs.

1.5 Representational Power of Feedforward Networks

Functions that can be represented:

- ❖ Boolean Functions: every Boolean function can be represented by some network with exactly 2 layers of units, though the number of hidden units can grow up to exponentially with the number of inputs.
- ❖ Continuous Functions: Every bounded continuous function can be approximated by a network with 2 layers of units such that the hidden layer contains sigmoid units while the output layer consists of linear units. The number of hidden units will depend on the function to be approximated.
- ❖ Arbitrary Functions: Any function can be approximated with a network of 3 layers of units. The output uses linear units and the hidden layers use sigmoid units. The number of units at each layer is not known.

1.6 Hypothesis Space Search and Inductive Bias

In backpropagation, every possible assignment of network weights represents a distinct hypothesis. This space is continuous, unlike the discrete representations such as decision trees. The hypothesis space is then an n-dimensional Euclidean space of the n network weights.

The general inductive bias in backpropagation can be described as "smooth interpolation between data points." In other words, given two positive data points with no negative examples between them, backpropagation tends to label the points in between as positive as well.

1.7 Hidden Layer Representations

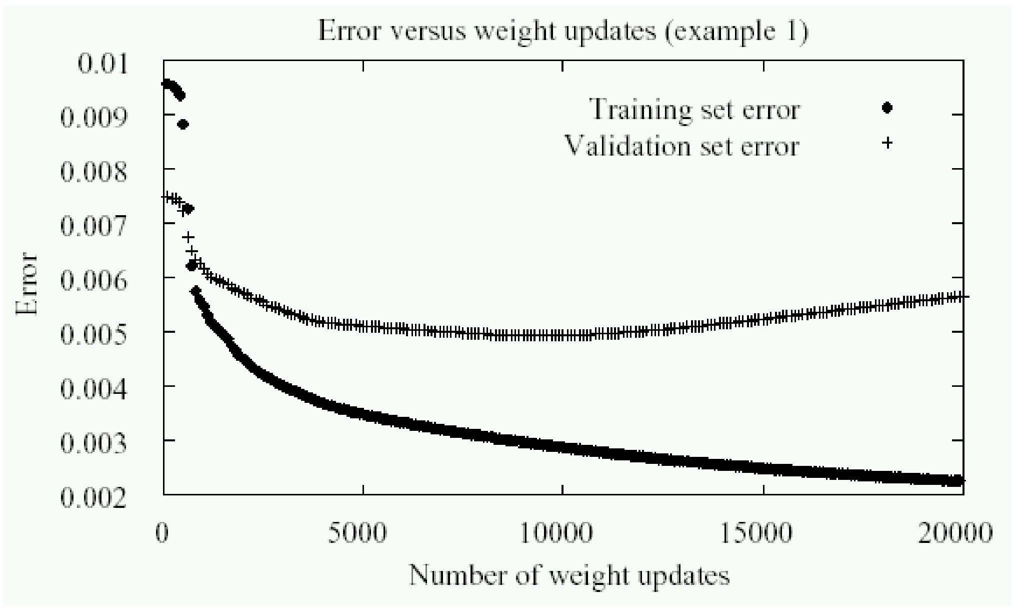
Because training examples only provide input and target output weights, the network is free to update the internal weights in order to minimize the error. This result in the network capturing properties that is not explicit in the input representation, which is a key feature to artificial neural network learning. This flexibility allows these networks to create new features not explicitly introduced by the designer, though these features must still be computable as sigmoid function units.

ANNs are used in general to approximate functions. However not every artificial neural network can approximate every function. There are limitations due to the architecture chosen, the number of layers and the hidden units per layer. For example, a network with just one hidden unit cannot learn the identity function. The network needs at least three hidden units to learn this function since there are 8 inputs and 8 outputs, $2^3 = 8$. We need 3 hidden units to encode every input as binary numbers (0, 1). For example, in the text, the hidden value (.89 .04 .08) are mapping to the binary number (1 0 0), which represents the input 10000000 (please refer to Dr. Schulte's answer key for detailed information of question 4.9).

How many hidden units are required? Intuition tells us, the more hidden units, the more local minimums will be introduced to the error surface. Additionally, there's a tradeoff between the generalization ability and approximation accuracy of an ANN. If we have too many hidden units, overfitting can occur.

1.8 Generalization, Overfitting, and Stopping Criterion

One of the stopping conditions for backpropagation is to continue to run until the error falls below some threshold. However, this has the tendency to overfit the training examples. This occurs because the network tunes the weights to fit the training data explicitly and not the general distribution of the examples. This tends to occur with larger iterations because at first the weights are random, but as the network runs, the complexity of the learned decision surface increases in response to weights being changed. This results in a learned function that can include noise. Another problem is simply that, if the training data contains a trend or tendency that doesn't exist in the actual data, the ANN is likely to learn that trend if it is given a sufficiently long training period.



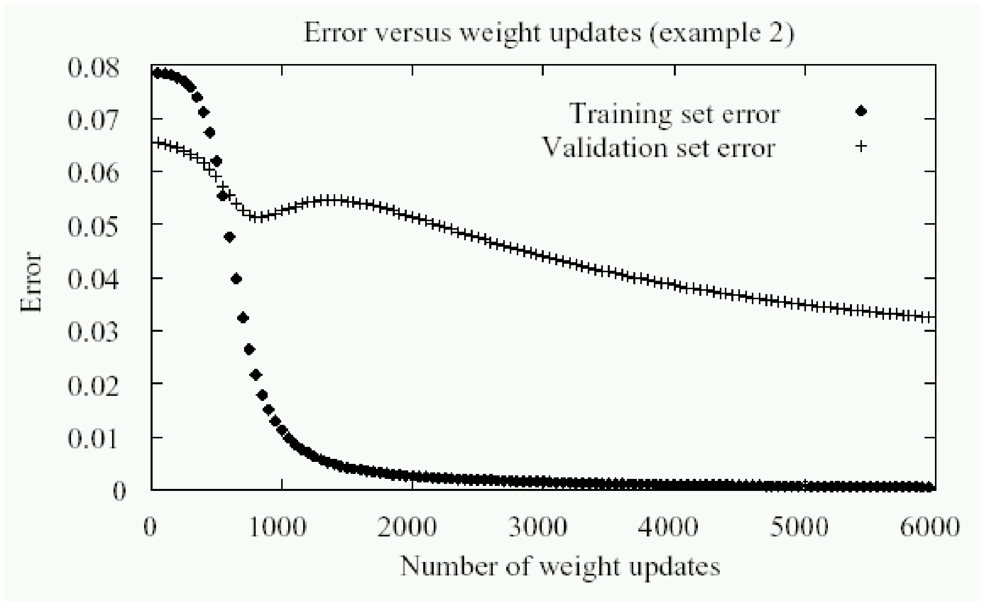


Figure 1.8

Figure 1.8 illustrates a typical situation that can arise in ANN learning. In both learning cases, error E over the training examples decreases monotonically, as gradient descent minimizes this measure of error. Error over the separate "validation" set of examples typically decreases at first, then may later increase due to overfitting the training examples. The network most likely to generalize correctly to unseen data is the network with the lowest error over the validation set. Notice in the second plot, one must be careful to not stop training too soon when the validation set error begins to increase. Note that the validation error eventually starts to increase, while the training set error continues to decrease. This is a clear indication of overfitting, because it illustrates that the behavior of the ANN is matched too closely to the training data.

Overfitting tends to occur during later iterations, but not during earlier iterations. The reason is as following: Consider that network weights are initialized to small random values with weights of nearly identical value, only very smooth decision surface are describable. As training proceeds, some weights begin to grow in order to reduce the error over the training data, and the complexity of the learned decision surface increases. Thus, the effective complexity of the hypotheses that can be reached by Backpropagation increases with the number of weight-tuning iterations. Given enough weight-tuning iterations, Backpropagation will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training example.

There are several techniques used to avoid overfitting. The first technique is called weight decay; where the weights are decreased by a small factor at every iteration. This process keeps weight values small and avoids complex decision surfaces. The second technique consists on using a validation set. In this case, the validation set serves as "unseen data". While network weights are updated according to the training data, the best set of weights with respect to error is kept separately. Then as the network trains, when the current error is far greater than the “best so far”, the training stops and the best set of weights is accepted. Here the trick is to stop training when error goes up on the validation set. This works well when there is enough data for a training and validation set.

If the available data set is not large, a k-fold cross-validation approach can be used. In this case, cross-validation is performed k different times, each time using a different partitioning of the data set, which is then averaged. One variation of this approach is to partition the data set into k disjoint subsets of size m/k where m is the size of the data set. Each run then uses a different subset as the validation set and unions the remaining sets for training. The cross-validation approach is used to determine the number of iterations i that produce the best performance on the validation set. The average $\bar{i}$ is then computed and a final run of the backpropagation algorithm is run on the entire data set for $\bar{i}$ iterations, with no validation set.

1.9 Definitions

Supervised learning: all learning algorithms where the known targets are used to adjust the network.
Linearly separable function: A function where if plotted in a n-dimensional plane, the negative and positive examples of the function can be totally separated using a straight plane across the space.
Perceptrons: a network with no units that are output or input, where the direction of data flow is in only one direction.
Artificial Neural Networks: (ANNs) these networks allow for learning using highly parallel series of simple units and are suited for data that is noisy and vector based.
Learning rate: a value greater than 0 but less than 1, this is used so that the weights on the links do not change to quickly, or the ANN might never converge onto the optimal solution.
Multi-layer Feed Forward Networks: a network with at least one unit that is not output or input, where the direction of data flow is in only one direction.
Backpropagation: a learning algorithm for multi-layered feed forward networks that uses the sigmoid function

References:

1. Dr. Oliver Schulte’s notes
2. Text book
3. Tom M. Mitchell’s online slides