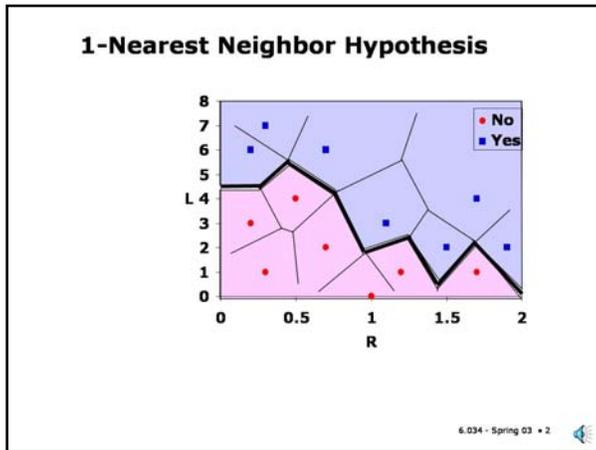
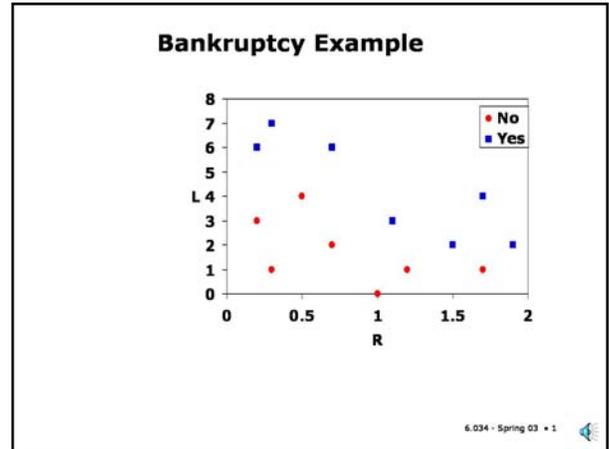


## 6.034 Notes: Section 7.1

### Slide 7.1.1

We have been using this simulated bankruptcy data set to illustrate the different learning algorithms that operate on continuous data. Recall that  $R$  is supposed to be the ratio of earnings to expenses while  $L$  is supposed to be the number of late payments on credit cards over the past year. We will continue using it in this section where we look at a new hypothesis class, **linear separators**.

One key observation is that each hypothesis class leads to a distinctive way of defining the **decision boundary** between the two classes. The decision boundary is where the class prediction changes from one class to another. Let's look at this in more detail.

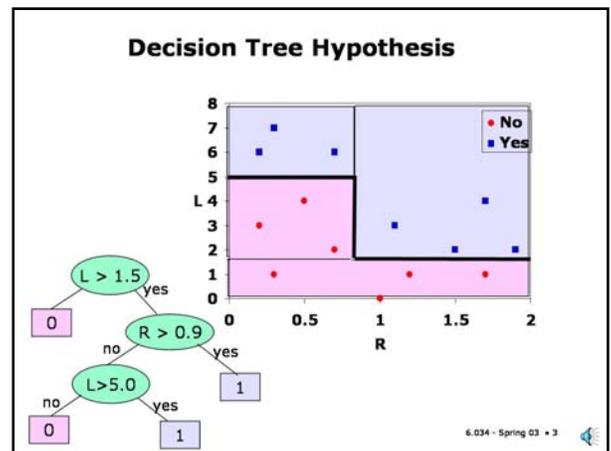


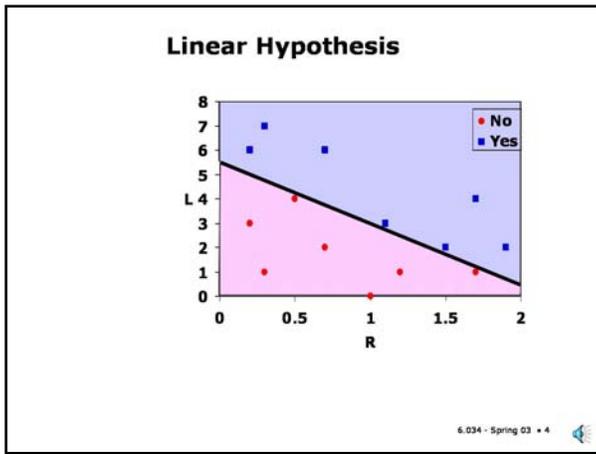
### Slide 7.1.2

We mentioned that a hypothesis for the 1-nearest neighbor algorithm can be understood in terms of a Voronoi partition of the feature space. The cells illustrated in this figure represent the feature space points that are closest to one of the training points. Any query in that cell will have that training point as its nearest neighbor and the prediction will be the class of that training point. The decision boundary will be the boundary between cells defined by points of different classes, as illustrated by the bold line shown here.

### Slide 7.1.3

Similarly, a decision tree also defines a decision boundary in the feature space. Note that although both 1-NN and decision trees agree on all the training points, they disagree on the precise decision boundary and so will classify some query points differently. This is the essential difference between different learning algorithms.





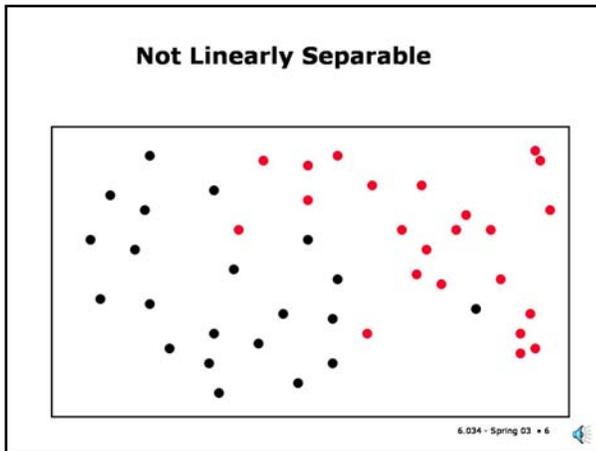
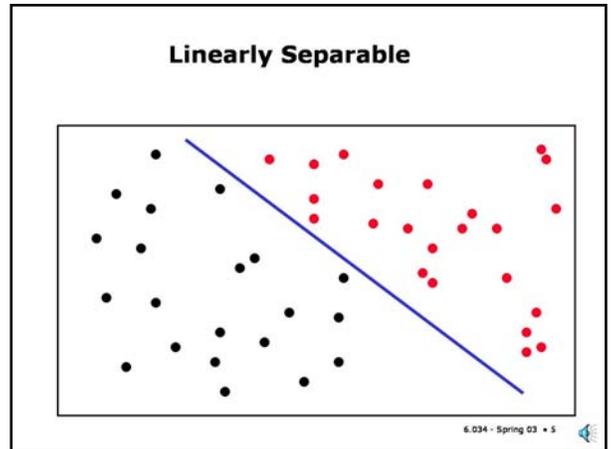
**Slide 7.1.4**

In this section we will be exploring **linear separators** which are characterized by a single linear decision boundary in the space. The bankruptcy data can be successfully separated in that manner. But, notice that in contrast to 1-NN and decision trees, there is no guarantee that a single linear separator will successfully classify any set of training data. The linear separator is a very simple hypothesis class, not nearly as powerful as either 1-NN or decision trees. However, as simple as this class is, in general, there will be many possible linear separators to choose from.

Also, note that, once again, this decision boundary disagrees with that drawn by the previous algorithms. So, there will be some data sets where a linear separator is ideally suited to the data. For example, it turns out that if the data points are generated by two Gaussian distributions with different means but the same standard deviation, then the linear separator is optimal.

**Slide 7.1.5**

A data set that can be successfully split by a linear separator is called, not surprisingly, **linearly separable**.



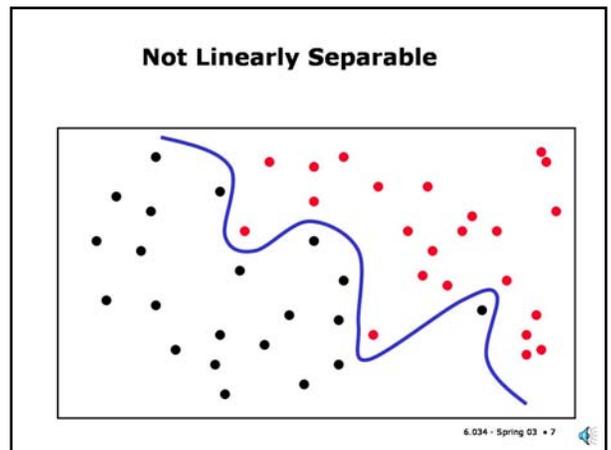
**Slide 7.1.6**

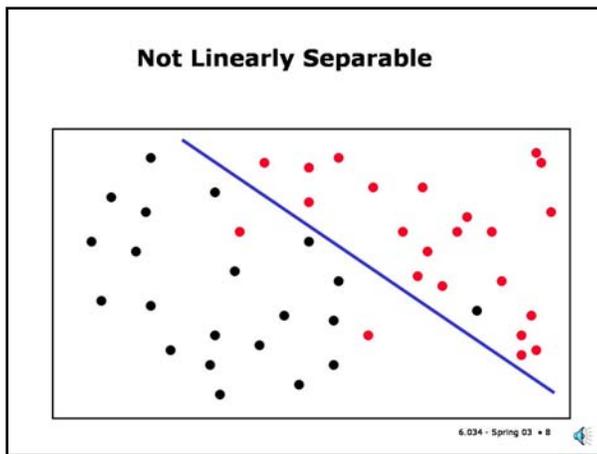
As we've mentioned, not all data sets are linearly separable. Here's one for example. Another classic non-linearly-separable data set is our old nemesis XOR.

It turns out, although it's not obvious, that the higher the dimensionality of the feature space, the more likely that a linear separator exists. This will turn out to be important later on, so let's just file that fact away.

**Slide 7.1.7**

When faced with a non-linearly-separable data set, we have two options. One is to use a more complex hypothesis class, such as shown here.



**Slide 7.1.8**

Or, keep the simple linear separator and accept some errors. This is the classic bias/variance tradeoff. Use a more complex hypothesis with greater variance or a simpler hypothesis with greater bias. Which is more appropriate depends on the underlying properties of the data, including the amount of noise. We can use our old friend cross-validation to make the choice if we don't have much understanding of the data.

**Slide 7.1.9**

So, let's look at the details of linear classifiers. First, we need to understand how to represent a particular hypothesis, that is, the equation of a linear separator. We will be illustrating everything in two dimensions but all the equations hold for an arbitrary number of dimensions.

The equation of a linear separator in an  $n$ -dimensional feature space is (surprise!) a linear equation which is determined by  $n+1$  values, the components of an  $n$ -dimensional coefficient vector  $\mathbf{w}$  and a scalar value  $b$ . These  $n+1$  values are what will be learned from the data. The  $\mathbf{x}$  will be some point in the feature space.

We will be using dot product notation for compactness and to highlight the geometric interpretation of this equation (more on this in a minute). Recall that the dot product is simply the sum of the componentwise products of the vector components, as shown here.

**Linear Hypothesis Class**

- Equation of a hyperplane in the feature space

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

$$\sum_{j=1}^n w_j x_j + b = 0$$

- $\mathbf{w}$ ,  $b$  are to be learned

6.034 - Spring 03 • 9

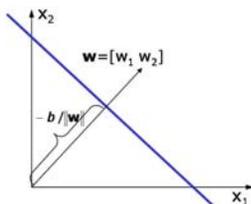
**Linear Hypothesis Class**

- Equation of a hyperplane in the feature space

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

$$\sum_{j=1}^n w_j x_j + b = 0$$

- $\mathbf{w}$ ,  $b$  are to be learned



6.034 - Spring 03 • 10

**Slide 7.1.10**

In two dimensions, we can see the geometric interpretation of  $\mathbf{w}$  and  $b$ . The vector  $\mathbf{w}$  is perpendicular to the linear separator; such a vector is known as the **normal** vector. Often we say "the vector normal to the surface". The scalar  $b$ , which we will call the **offset**, is proportional to the perpendicular distance from the origin to the linear separator. The constant of proportionality is the negative of the magnitude of the normal vector. We'll examine this in more detail soon.

By the way, the choice of the letter "w" is traditional and meant to suggest "weights", we'll see why when we look at neural nets. The choice of "b" is meant to suggest "bias" - which is the third different connotation of this word in machine learning (the bias of a hypothesis class, bias vs variance, bias of a separator). They are all fundamentally related; they all refer to a difference from a neutral value. To keep the confusion down to a dull roar, we won't call  $b$  a bias term but are telling you about this so you won't be surprised if you see it elsewhere.

**Slide 7.1.11**

Sometimes we will use the following trick to simplify the equations. We'll treat the offset as the  $0^{\text{th}}$  component of the weight vector  $\mathbf{w}$  and we'll augment the data vector  $\mathbf{x}$  with a  $0^{\text{th}}$  component that will always be equal to 1. Then we can write a linear equation as a dot product. When we do this, we will indicate it by using an overbar over the vectors.

**Linear Hypothesis Class**

- Equation of a hyperplane in the feature space

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

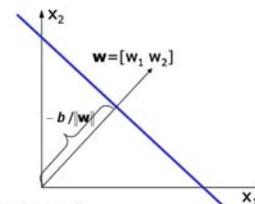
$$\sum_{j=1}^n w_j x_j + b = 0$$

- $\mathbf{w}$ ,  $b$  are to be learned

- A useful trick: let  $x_0=1$  and  $w_0=b$

$$\overline{\mathbf{w}} \cdot \overline{\mathbf{x}} = 0$$

$$\sum_{j=0}^n w_j x_j = 0$$



6.034 - Spring 03 • 11

### Hyperplane: Geometry

6.034 - Spring 03 • 12

**Slide 7.1.12**

First a word on terminology: the equations we will be writing apply to linear separators in  $n$  dimensions. In two dimensions, such a linear separator is referred to as a "line". In three dimensions, it is called a "plane". These are familiar words. What do we call it in higher dimensions? The usual terminology is **hyperplane**. I know that sounds like some type of fast aircraft, but that's the accepted name.

Let's look at the geometry of a hyperplane a bit more closely. We saw earlier that the offset  $b$  in the linear separator equation is proportional to the perpendicular distance from the origin to the linear separator and that the constant of proportionality is the magnitude of the  $w$  vector (negated). Basically, we can multiply both sides of the equation by any number without affecting the equality. So, there are an infinite set of equations all of which represent the same separator.

If we divide the equation through by the magnitude of  $w$  we end up with the situation shown in the figure. The normal vector is now unit length (denoted by the hat on the  $w$ ) and the offset  $b$  is now equal to the perpendicular distance from the origin (negated).

**Slide 7.1.13**

It's crucial to understand that the quantity  $w\text{-hat} \cdot x + b$  is the perpendicular distance of point  $x$  to the linear separator.

If you recall, the geometric interpretation of a dot product  $a \cdot b$  is that it is a number which is the magnitude of  $a$  times the magnitude of  $b$  times the cosine of the angle between the vectors. If one of the vectors, say  $a$ , has unit magnitude then what we have is precisely the magnitude of the projection of the  $b$  vector onto the direction defined by  $a$ . Thus  $w\text{-hat} \cdot x$  is the distance from  $x$  to the origin measured perpendicular to the hyperplane.

Looking at the right triangle defined by the  $w\text{-hat}$  and the  $x$  vector, both emanating from the origin, we see that the projection of  $x$  onto  $w\text{-hat}$  is the length of the base of the triangle, where  $x$  is the hypotenuse and the base angle is theta.

Now, if we subtract out the perpendicular distance to the origin we get the distance of  $x$  from the hyperplane (rather than from the origin). Note that when theta is 90 degrees (that is,  $w$  and  $x$  are perpendicular), the cosine is equal to 0 and the distance is precisely  $b$  as we expect.

### Hyperplane: Geometry

$\hat{w} \cdot x + b$

signed perpendicular distance of point  $x$  to hyperplane.

recall:  $a \cdot b = \|a\| \|b\| \cos \theta$

6.034 - Spring 03 • 13

### Hyperplane: Geometry

$\hat{w} \cdot x + b$

signed perpendicular distance of point  $x$  to hyperplane.

recall:  $a \cdot b = \|a\| \|b\| \cos \theta$

6.034 - Spring 03 • 14

**Slide 7.1.14**

This distance measure from the hyperplane is **signed**. It is zero for points on the hyperplane, it is positive for points in the side of the space towards which the normal vector points, and negative for points on the other side. Notice that if you multiply the normal vector  $w$  and the offset  $b$  by  $-1$ , you get an equation for the same hyperplane but you switch which side of the hyperplane has positive distances.

**Slide 7.1.15**

We can now exploit the sign of this distance to define a linear classifier, one whose decision boundary is a hyperplane. Instead of using 0 and 1 as the class labels (which was an arbitrary choice anyway) we use the sign of the distance, either  $+1$  or  $-1$  as the labels (that is the values of the  $y_i$ ).

### Linear Classifier

$h(x) = \text{sign}(w \cdot x + b) \equiv \text{sign}(\bar{w} \cdot \bar{x})$

outputs  $+1$  or  $-1$

6.034 - Spring 03 • 15

### Linear Classifier

**Margin:**  
 $\gamma_i = y^i (\mathbf{w} \cdot \mathbf{x}^i + b) = y^i \mathbf{w} \cdot \mathbf{x}^i$   
 proportional to perpendicular distance of point  $\mathbf{x}^i$  to hyperplane.

$\gamma_i > 0$  : point is **correctly** classified (sign of distance =  $y^i$ )

$\gamma_i < 0$  : point is **incorrectly** classified (sign of distance  $\neq y^i$ )

6.034 - Spring 03 • 16

**Slide 7.1.16**

A variant of the signed distance of a training point to a hyperplane is the **margin** of the point. The margin ( $\gamma$ ) is the product of the actual signed distance for the point and the desired sign of the distance,  $y_i$ . If they agree (the point is correctly classified), then the margin is positive; if they disagree (the classification is in error), then the margin is negative.

**6.034 Notes: Section 7.2****Slide 7.2.1**

So far we've talked about how to represent a linear hypothesis but not how to find one. In this slide is the perceptron algorithm, developed by Rosenblatt in the mid 50's. This is not exactly the original form of the algorithm but it is equivalent and it will help us later to see it in this form.

This is a greedy, "mistake driven" algorithm not unlike the Boolean function learning algorithms we saw earlier. We will be using the extended form of the weight and data-point vectors in this algorithm. The extended weight vector is what we are trying to learn.

The first step is to start with an initial value of the weight vector, usually all zeros. Then we repeat the inner loop until all the points are correctly classified using the current weight vector. The inner loop is to consider each point. If the point's margin is positive then it is correctly classified and we do nothing. Otherwise, if it is negative or zero, we have a mistake and we want to change the weights so as to increase the margin (so that it ultimately becomes positive).

The trick is how to change the weights. It turns out that using a value proportional to  $y\mathbf{x}$  is the right thing. We'll see why, formally, later. For now, let's convince ourselves that it makes sense.

**Perceptron Algorithm**  
Rosenblatt, 1956

- Pick initial weight vector (including  $b$ ), e.g.  $[0 \dots 0]$
- Repeat until all points correctly classified
  - Repeat for each point
    - Calculate margin (  $y^i \mathbf{w} \cdot \mathbf{x}^i$  ) for point  $i$
    - If margin  $> 0$ , point is correctly classified
    - Else change weights to increase margin; change in weight proportional to  $y^i \mathbf{x}^i$

6.034 - Spring 03 • 1

**Perceptron Algorithm**  
Rosenblatt, 1956

- Pick initial weight vector (including  $b$ ), e.g.  $[0 \dots 0]$
- Repeat until all points correctly classified
  - Repeat for each point
    - Calculate margin (  $y^i \mathbf{w} \cdot \mathbf{x}^i$  ) for point  $i$
    - If margin  $> 0$ , point is correctly classified
    - Else change weights to increase margin; change in weight proportional to  $y^i \mathbf{x}^i$
- Note that, if  $y^i = 1$ 
  - if  $x_j^i > 0$  then  $w_j$  increased (increases margin)
  - if  $x_j^i < 0$  then  $w_j$  decreased (increases margin)
- And, similarly for  $y^i = -1$

6.034 - Spring 03 • 2

**Slide 7.2.2**

Consider the case in which  $y$  is positive; the negative case is analogous. If the  $j$ th component of  $\mathbf{x}$  is positive then we will increase the corresponding component of  $\mathbf{w}$ . Note that the resulting effect on the margin is positive. If the  $j$ th component of  $\mathbf{x}$  is negative then we will decrease the corresponding component of  $\mathbf{w}$ , and the resulting effect on the margin is also positive.

**Slide 7.2.3**

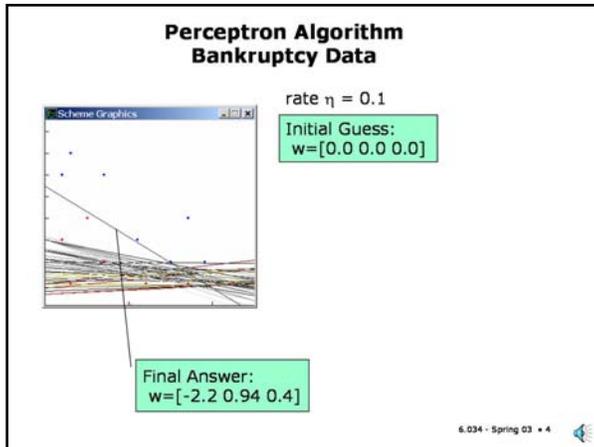
So, each change of  $\mathbf{w}$  increases the margin on a particular point. However, the changes for the different points interfere with each other, that is, different points might change the weights in opposing directions. So, it will not be the case that one pass through the points will produce a correct weight vector. In general, we will have to go around multiple times.

The remarkable fact is that the algorithm is guaranteed to terminate with the weights for a separating hyperplane as long as the data is linearly separable. The proof of this fact is beyond our scope.

Notice that if the data is not separable, then this algorithm is an infinite loop. It turns out that it is a good idea to keep track of the best separator you've seen so far (the one that makes the fewest mistakes) and after you get tired of going around the loop, return that one. This algorithm even has a name (the **pocket** algorithm: see, it keeps the best answer in its pocket...).

### Perceptron Algorithm Rosenblatt, 1956

- Pick initial weight vector (including  $b$ ), e.g.  $[0 \dots 0]$
- Repeat until all points correctly classified
  - Repeat for each point
    - Calculate margin ( $y^i \mathbf{w} \mathbf{x}^i$ ) for point  $i$
    - If margin  $> 0$ , point is correctly classified
    - Else change weights to increase margin; change in weight proportional to  $y^i \mathbf{x}^i$
- Note that, if  $y^i = 1$ 
  - if  $x_j^i > 0$  then  $w_j$  increased (increases margin)
  - if  $x_j^i < 0$  then  $w_j$  decreased (increases margin)
- And, similarly for  $y^i = -1$
- Guaranteed to find separating hyperplane if one exists
- Otherwise, data are not linearly separable, loops forever

**Slide 7.2.4**

This shows a trace of the perceptron algorithm on the bankruptcy data. Here it took 49 iterations through the data (the outer loop) for the algorithm to stop. The hypothesis at the end of each loop is shown here. Recall that the first element of the weight vector is actually the offset. So, the normal vector to the separating hyperplane is  $[0.94 \ 0.4]$  and the offset is  $-2.2$  (recall that is proportional to the negative perpendicular distance from origin to the line).

Note that the units in the horizontal and vertical directions in this graph are not equal (the tick marks along the axes indicate unit distances). We did this since the range of the data on each axis is so different.

One usually picks some small "rate" constant to scale the change to  $\mathbf{w}$ . It turns out that for this algorithm the value of the rate constant does not matter. We have used 0.1 in our examples, but 1 also works well.

**Slide 7.2.5**

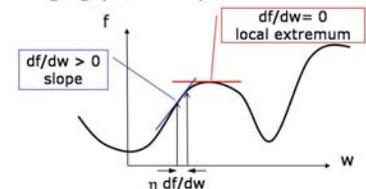
Let's revisit the issue of why we picked  $y\mathbf{x}$  to increment  $\mathbf{w}$  in the perceptron algorithm. It might have seemed arbitrary but it's actually an instance of a general strategy called **gradient ascent** for finding the input(s) that maximize a function's output (or gradient descent when we are minimizing).

The strategy in one input dimension is shown here. We guess an initial value of the input. We calculate the slope of the function at that input value and we take a step that is proportional to the slope. Note that the sign of the slope will tell us whether an increase of the input variable will increase or decrease the value of the output. The magnitude of the slope will tell us how fast the function is changing at that input value. The slope is basically a linear approximation of the function which is valid "near" the chosen input value. Since the approximation is only valid locally, we want to take a small step (determined by the rate constant  $\eta$ ) and repeat.

We want to stop when the output change is zero (or very small). This should correspond to a point where the slope is zero, which should be a local extremum of the function. This strategy will not guarantee finding the global maximal value, only a local one.

### Gradient Ascent

- Why pick  $y^i \mathbf{x}^i$  as increment to weights?
- To maximize scalar function of one variable  $f(\mathbf{w})$ 
  - Pick initial  $\mathbf{w}$
  - Change  $\mathbf{w}$  to  $\mathbf{w} + \eta \frac{df}{d\mathbf{w}}$  ( $\eta > 0$ , small)
  - until  $f$  stops changing ( $\frac{df}{d\mathbf{w}} \approx 0$ )



### Gradient Ascent/Descent

- To maximize  $f(\mathbf{w})$   $\nabla_{\mathbf{w}} f = \left[ \frac{\partial f}{\partial w_1}, \dots, \frac{\partial f}{\partial w_n} \right]$ 
  - Pick initial  $\mathbf{w}$
  - Change  $\mathbf{w}$  to  $\mathbf{w} + \eta \nabla_{\mathbf{w}} f$  ( $\eta > 0$ , small)
  - until  $f$  stops changing ( $\nabla_{\mathbf{w}} f \approx 0$ )
- Finds local maximum; global maximum if function is globally convex.

**Slide 7.2.6**

The generalization of this strategy to multiple input variables is based on the generalization of the notion of slope, which is the **gradient** of the function. The gradient is the vector of first (partial) derivatives of the function with respect to each of the input variables. The gradient vector points in the direction of steepest increase of the function output. So, we take a small step in that direction, recompute the gradient and repeat until the output stops changing. Once again, this will only find us a local maximum of the function, in general. However, if the function is globally convex, then it will find the global optimum.

**Slide 7.2.7**

In general, the choice of the rate constant (eta), which determines the step size, is fairly critical. Unfortunately, no single value is appropriate for all functions. If one chooses a very conservative small rate, it can take a long time to find a minimum, if one takes too big steps there is no guarantee that the algorithm will even converge to a minimum; it can oscillate as shown in the figure here where the sign of the slope changes and causes a back-and-forth search.

In more sophisticated search algorithms one does a search along the specified direction looking for a value of the step size that guarantees an increase in the function value.

### Gradient Ascent/Descent

- To maximize  $f(\mathbf{w})$   $\nabla_{\mathbf{w}} f = \left[ \frac{\partial f}{\partial w_1}, \dots, \frac{\partial f}{\partial w_n} \right]$ 
  - Pick initial  $\mathbf{w}$
  - Change  $\mathbf{w}$  to  $\mathbf{w} + \eta \nabla_{\mathbf{w}} f$  ( $\eta > 0$ , small)
  - until  $f$  stops changing ( $\nabla_{\mathbf{w}} f \approx 0$ )
- Finds local maximum; global maximum if function is globally convex
- Rate ( $\eta$ ) has to be chosen carefully.
  - Too small – slow convergence
  - Too big – oscillation

### Perceptron Training via Gradient Descent

- Maximize sum of margins of misclassified points

$$f(\mathbf{w}) = \sum_{i \text{ misclassified}} y^i \mathbf{w} \mathbf{x}^i$$

$$\nabla_{\mathbf{w}} f = \sum_{i \text{ misclassified}} y^i \mathbf{x}^i$$

6.034 - Spring 03 • 8

**Slide 7.2.8**

Now we can see that our choice of increment in the perceptron algorithm is related to the gradient of the sum of the margins for the misclassified points.

**Slide 7.2.9**

If we actually want to maximize this sum via gradient descent we should sum all the corrections for every misclassified point using a single  $\mathbf{w}$  vector and then apply that correction to get a new weight vector. We can then repeat the process until convergence. This is normally called an **off-line** algorithm in that it assumes access to all the input points.

What we actually did was a bit different, we modified  $\mathbf{w}$  based on each point as we went through the inner loop. This is called an **on-line** algorithm because, in principle, if the points were arriving over a communication link, we would make our update to the weights based on each arrival and we could discard the points after using them, counting on more arriving later.

Another way of thinking about the relationship of these algorithms is that the on-line version is using a (randomized) approximation to the gradient at each point. It is randomized in the sense that rather than taking a step based on the true gradient, we take a step based on an estimate of the gradient based on a randomly drawn example point. In fact, the on-line version is sometimes called "stochastic (randomized) gradient ascent" for this reason. In some cases, this randomness is good because it can get us out of shallow local minima.

### Perceptron Training via Gradient Descent

- Maximize sum of margins of misclassified points

$$f(\mathbf{w}) = \sum_{i \text{ misclassified}} y^i \mathbf{w} \mathbf{x}^i$$

$$\nabla_{\mathbf{w}} f = \sum_{i \text{ misclassified}} y^i \mathbf{x}^i$$

- Off-line training: Compute gradient as sum over all training points.
- On-line training: Approximate gradient by one of the terms in the sum:  $y^i \mathbf{x}^i$

6.034 - Spring 03 • 9

### Perceptron Algorithm Bankruptcy Data

rate  $\eta = 0.1$

$w_0$	$w_1$	$w_2$
-1.0	1.00	1.0
-1.3	0.71	0.6
-1.4	0.66	0.5
-1.4	0.66	0.8
-1.5	0.61	0.7
-1.6	0.56	0.6
-1.6	0.65	0.6
-1.6	0.74	0.6
-1.6	0.83	0.6
-1.7	0.81	0.3

Initial Guess:  
 $w = [-1.0 \ 1.0 \ 1.0]$

Final Answer:  
 $w = [-1.7 \ 0.81 \ 0.3]$

6.034 - Spring 03 • 10

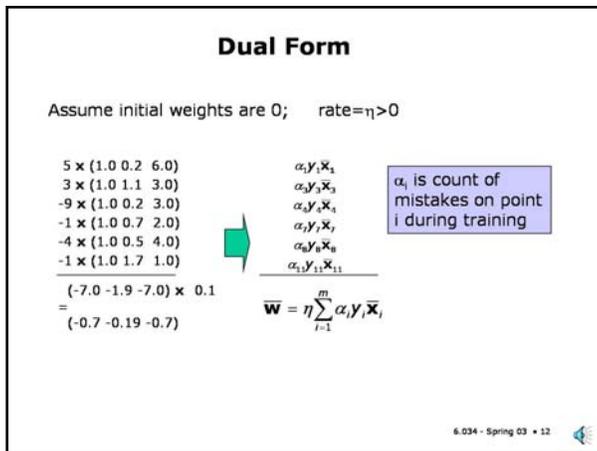
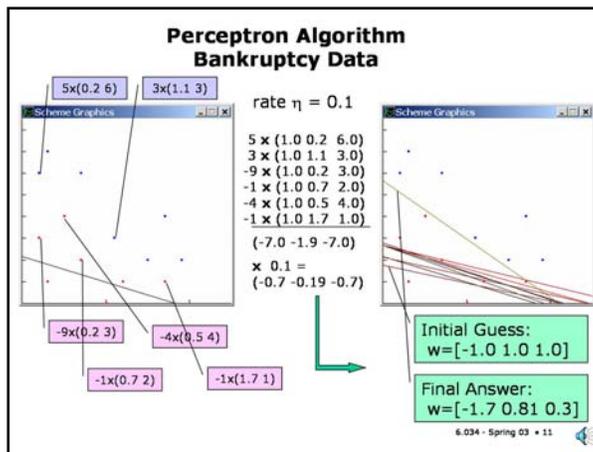
**Slide 7.2.10**

Here's another look at the perceptron algorithm on the bankruptcy data with a different initial starting guess of the weights. You can see the different separator hypotheses that it goes through. Note that it converges to a different set of weights from our previous example. However, recall that one can scale these weights and get the same separator. In fact these numbers are approximately 0.8 of the ones we got before, but only approximately; this is a slightly different separator.

The perceptron algorithm can be described as a gradient ascent algorithm, but its error criterion is slightly unusual in that there are many separators that all have zero error.

**Slide 7.2.11**

Recall that the perceptron algorithm starts with an initial guess for the weights and then adds in scaled versions of the misclassified training points to get the final weights. In this particular set of 10 iterations, the points indicated on the left are misclassified some number of times each. For example, the leftmost negative point is misclassified in each iteration except the last one. If we sum up the coordinates of each of these points, scaled by how many times each is misclassified and by the rate constant we get the total change in the weight vector.

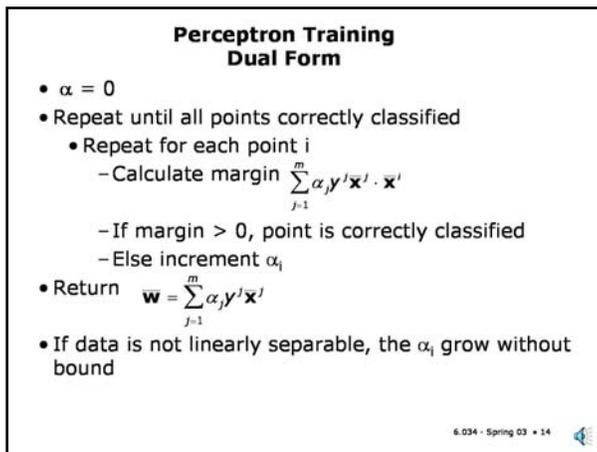
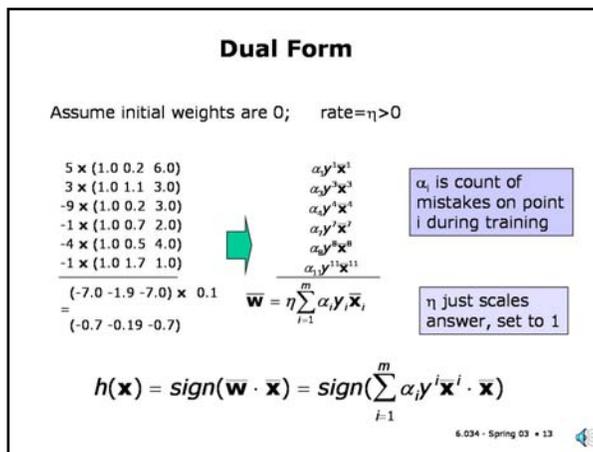


**Slide 7.2.12**

This analysis leads us to a somewhat different view of the perceptron algorithm, usually called the **dual form** of the algorithm. Call the count of how many times point  $i$  is misclassified,  $\alpha_i$ . Then, assuming the weight vector is initialized to 0s, we can write the final weight vector in terms of these counts and the input data (as well as the rate constant).

**Slide 7.2.13**

Since the rate constant does not change the separator we can simply assume that it is 1 and ignore it. Now, we can substitute this form of the weights in the classifier and we get the classifier at the bottom of the slide, which has the interesting property that the data points only appear in dot-products with other data points. This will turn out to be extremely important later; file this one away.



**Slide 7.2.14**

We can now restate the perceptron algorithm in this interesting way. The separator is described as a weighted sum of the input points, with  $\alpha_i$  the weight for point  $i$ . Initially, set all of the alphas to zero, so the separator has all zero's as coefficients.

Then, for each point, compute its margin with respect to the current separator. If the margin is positive, the point is classified correctly, so do nothing. If the margin is negative, add that point into the weights of the separator. We can do that simply by incrementing the associated alpha.

Finally, when all of the points are classified correctly, we return the weighted sum of the inputs as the coefficients for the separator. Note that if the data is not linearly separable, then the algorithm will loop forever, the alphas growing without bound.

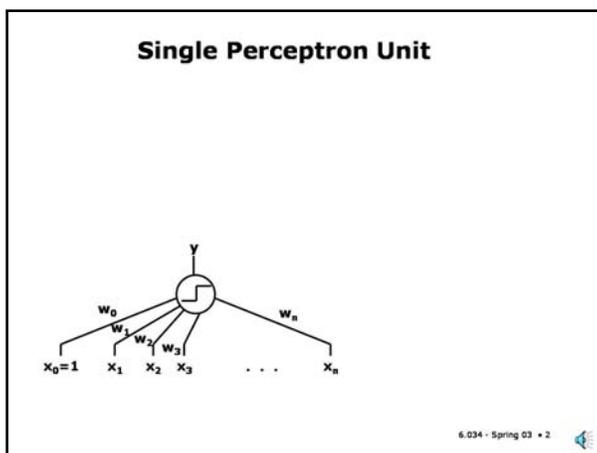
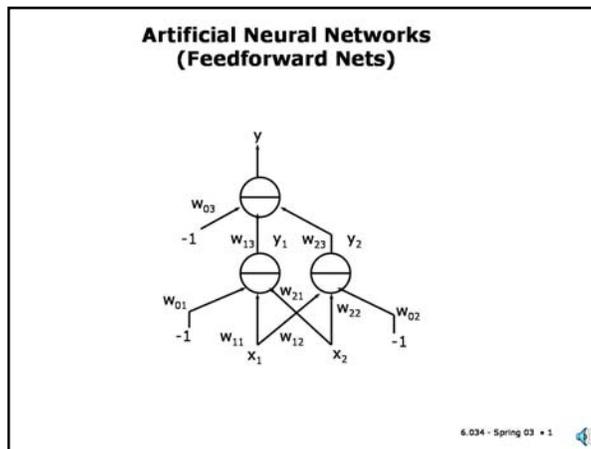
You should convince yourself that this dual form is equivalent to the original. Once again, you may be wondering...so what? I'll say again; file this away. It has surprising consequences.

## 6.034 Notes: Section 7.3

### Slide 7.3.1

We will now turn our attention to artificial neural nets, sometimes also called "feedforward nets".

The basic idea in neural nets is to define interconnected networks of simple units (let's call them "artificial neurons") in which each connection has a weight. Weight  $w_{ij}$  is the weight of the  $i^{\text{th}}$  input into unit  $j$ . The networks have some inputs where the feature values are placed and they compute one or more output values. The learning takes place by adjusting the weights in the network so that the desired output is produced whenever a sample in the input data set is presented.



### Slide 7.3.2

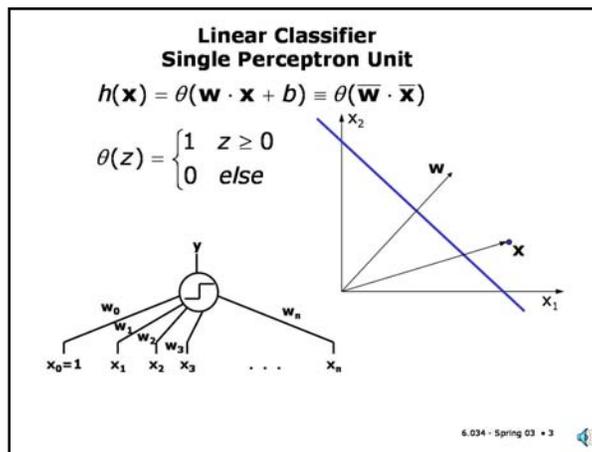
We start by looking at a simpler kind of "neural-like" unit called a **perceptron**. This is where the perceptron algorithm that we saw earlier came from. Perceptrons antedate the modern neural nets. Examining them can help us understand how the more general units work.

### Slide 7.3.3

A perceptron unit basically compares a weighted combination of its inputs against a threshold value and then outputs a 1 if the weighted inputs exceed the threshold. We use our trick here of treating the (arbitrary) threshold as if it were a weight ( $w_0$ ) on a constant input ( $x_0$ ) whose value is -1 (note the sign is different from what we saw in our previous treatment but the idea is the same). In this way, we can write the basic rule of operation as computing the weighted sum of all the inputs and comparing to 0.

The key observation is that the decision boundary for a single perceptron unit is a hyperplane in the feature space. That is, it is a linear equation that divides the space into two half-spaces. We can easily see this in two dimensions. The equation that tells us when the perceptron's total input goes to zero is the equation of a line whose normal is the weight vector  $[w_1 \ w_2]$ . On one side of this line, the value of the weighted input is negative and so the perceptron's output is 0, on the other side of the line the weighted input is positive and the output is 1.

We have seen that there's a simple gradient-descent algorithm for finding such a linear separator if one exists.



### Beyond Linear Separability

6.034 - Spring 03 • 4

Slide 7.3.4

Since a single perceptron unit can only define a single linear boundary, it is limited to solving linearly separable problems. A problem like that illustrated by the values of the XOR boolean function cannot be solved by a single perceptron unit.

Slide 7.3.5

We have already seen in our treatment of SVMs how the "kernel trick" can be used to generalize a perceptron-like classifier to produce arbitrary boundaries, basically by mapping into a high-dimensional space of non-linear mappings of the input features.

### Beyond Linear Separability

6.034 - Spring 03 • 5

### Beyond Linear Separability

6.034 - Spring 03 • 6

Slide 7.3.6

We will now explore a different approach (although later we will also introduce non-linear mappings). What about if we consider more than one linear separator and combine their outputs; can we get a more powerful classifier?

Slide 7.3.7

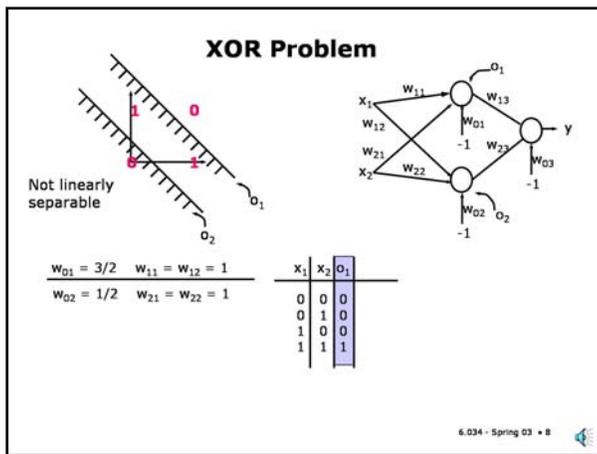
The answer is yes. Since a single perceptron unit is so limited, a network of these units will be less limited. In fact, the introduction of "hidden" (not connected directly to the output) units into these networks make them much more powerful: they are no longer limited to linearly separable problems.

What these networks do is basically use the earlier layers (closer to the input) to transform the problem into more tractable problems for the latter layers.

### Multi-Layer Perceptron

- More powerful than single layer.
- Lower layers transform the input problem into more tractable (linearly separable) problems for subsequent layers.

6.034 - Spring 03 • 7



Slide 7.3.8

To see how having hidden units can help, let us see how a two-layer perceptron network can solve the XOR problem that a single unit failed to solve.

We see that each hidden unit defines its own "decision boundary" and the output from each of these units is fed to the output unit, which returns a solution to the whole problem. Let's look in detail at each of these boundaries and its effect.

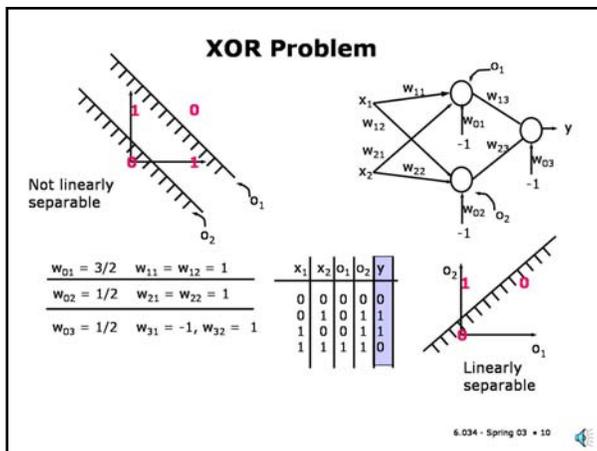
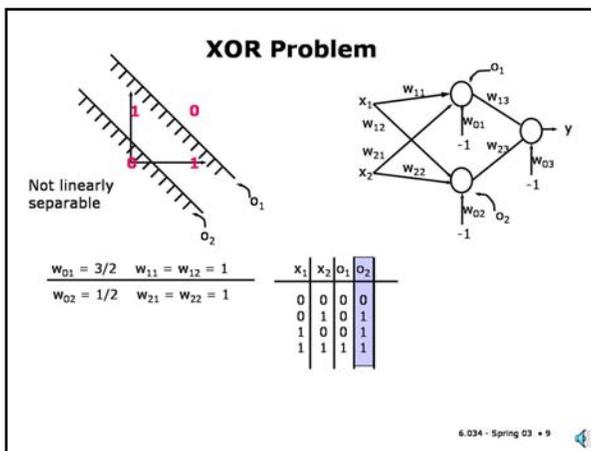
Note that each of the weights in the first layer, except for the offsets, has been set to 1. So, we know that the decision boundaries are going to have normal vectors equal to  $[1 \ 1]$ , that is, pointing up and to the right, as shown in the diagram. The values of the offsets show that the hidden unit labeled  $o_1$  has a larger offset (that is, distance from the origin) and the hidden unit labeled  $o_2$  has a smaller offset. The actual distances from the line to the origin are obtained by dividing the offsets by  $\sqrt{2}$ , the magnitude of the normal vectors.

If we focus on the first decision boundary we see only one of the training points (the one with feature values  $(1,1)$ ) is in the half space that the normal points into. This is the only point with a positive distance and thus a one output from the perceptron unit. The other points have negative

distance and produce a zero output. This is shown in the shaded column in the table.

Slide 7.3.9

Looking at the second decision boundary we see that three of the training points (except for the one with feature values  $(0,0)$ ) are in the half space that the normal points into. These points have a positive distance and thus a one output from the perceptron unit. The other point has negative distance and produces a zero output. This is shown in the shaded column in the table.



Slide 7.3.10

On the lower right, we see that the problem has been mapped into a linearly separable problem in the space of the outputs of the hidden units. We can now easily find a linear separator, for example, the one shown here. This mapping is where the power of the multi-layer perceptron comes from.

Slide 7.3.11

It turns out that a three-layer perceptron (with sufficiently many units) can separate any data set. In fact, even a two-layer perceptron (with lots of units) can separate almost any data set that one would see in practice.

However, the presence of the discontinuous threshold in the operation means that there is no simple local search for a good set of weights; one is forced into trying possibilities in a combinatorial way.

The limitations of the single-layer perceptron and the lack of a good learning algorithm for multi-layer perceptrons essentially killed the field of statistical machine learning for quite a few years. The stake through the heart was a slim book entitled "Perceptrons" by Marvin Minsky and Seymour Papert of MIT.

**Multi-Layer Perceptron Learning**

- Any set of training points can be separated by a three-layer perceptron network.
- "Almost any" set of points separable by two-layer perceptron network.
- But, no efficient learning rule is known.

6.034 - Spring 03 • 11

### Multi-Layer Perceptron Learning

- Any set of training points can be separated by a three-layer perceptron network.
- "Almost any" set of points separable by two-layer perceptron network.
- But, no efficient learning rule is known.
- Could we use gradient ascent/descent?
- We would need smoothness: small change in weights produces small change in output.
- Threshold function is not smooth.

6.034 - Spring 03 • 12

### Slide 7.3.12

A natural question to ask is whether we could use gradient descent to train a multi-layer perceptron. The answer is that we can't as long as the output is discontinuous with respect to changes in the inputs and the weights. In a perceptron unit it doesn't matter how far a point is from the decision boundary, you will still get a 0 or a 1. We need a smooth output (as a function of changes in the network weights) if we're to do gradient descent.

### Slide 7.3.13

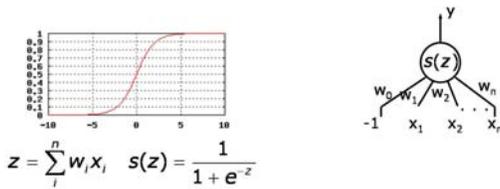
Eventually people realized that if one "softened" the thresholds, one could get information as to whether a change in the weights was helping or hurting and define a local improvement procedure that way.

### Multi-Layer Perceptron Learning

- Any set of training points can be separated by a three-layer perceptron network.
- "Almost any" set of points separable by two-layer perceptron network.
- But, no efficient learning rule is known.
- Could we use gradient ascent/descent?
- We would need smoothness: small change in weights produces small change in output.
- Threshold function is not smooth.
- Use a smooth threshold function!

6.034 - Spring 03 • 13

### Sigmoid Unit



6.034 - Spring 03 • 14

### Slide 7.3.14

The classic "soft threshold" that is used in neural nets is referred to as a "sigmoid" (meaning S-like) and is shown here. The variable  $z$  is the "total input" or "activation" of a neuron, that is, the weighted sum of all of its inputs.

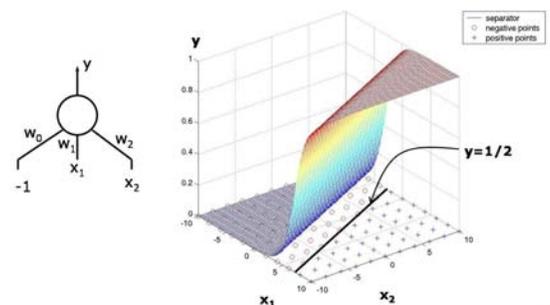
Note that when the input ( $z$ ) is 0, the sigmoid's value is 1/2. The sigmoid is applied to the weighted inputs (including the threshold value as before). There are actually many different types of sigmoids that can be (and are) used in neural networks. The sigmoid shown here is actually called the **logistic** function.

### Slide 7.3.15

We can think of a sigmoid unit as a "soft" perceptron. The line where the perceptron switches from a 0 output to a 1, is now the line along which the output of the sigmoid unit is 1/2. On one side of this line, the output tends to 0, on the other it tends to 1.

So, this "logistic perceptron" is still a linear separator in the input space. In fact, there's a well known technique in statistics, called **logistic regression** which uses this type of model to fit the probabilities of boolean-valued outputs, which are not properly handled by a linear regression. Note that since the output of the logistic function is between 0 and 1, the output can be interpreted as a probability.

### Sigmoid Unit



6.034 - Spring 03 • 15

### Training

$y(\mathbf{x}, \mathbf{w})$   
 $\mathbf{w}$  is a vector of weights  
 $\mathbf{x}$  is a vector of inputs

$$y = s(w_{13}s(w_{11}x_1 + w_{21}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02})) - w_{03}$$

6.034 - Spring 03 - 16

Slide 7.3.16

The key property of the sigmoid is that it is differentiable. This means that we can use gradient-based methods of minimization for training. Let's see what that means.

The output of a multi-layer net of sigmoid units is a function of two vectors, the inputs ( $x$ ) and the weights ( $w$ ). An example of what that function looks like for a simple net is shown along the bottom, where  $s()$  is whatever output function we are using, for example, the logistic function we saw in the last slide.

The output of this function ( $y$ ) varies smoothly with changes in the input and, importantly, with changes in the weights. In fact, the weights and inputs both play similar roles in the function.

Slide 7.3.17

Given a dataset of training points, each of which specifies the net inputs and the desired outputs, we can write an expression for the **training error**, usually defined as the sum of the squared differences between the actual output (given the weights) and the desired output. The goal of training is to find a weight vector that minimizes the training error.

We could also use the mean squared error (MSE), which simply divides the sum of the squared errors by the number of training points instead of just 2. Since the number of training points is a constant, the value of the minimum is not affected.

### Training

$y(\mathbf{x}, \mathbf{w})$   
 $\mathbf{w}$  is a vector of weights  
 $\mathbf{x}$  is a vector of inputs  
 $y^i$  is desired output:

Error over the training set for a given weight vector:

$$E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$$

Our goal is to find weight vector that minimizes error

$$y = s(w_{13}s(w_{11}x_1 + w_{21}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02})) - w_{03}$$

6.034 - Spring 03 - 17

### Gradient Descent

$$E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$$

Error on training set

$$\nabla_{\mathbf{w}} E = \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i) \nabla_{\mathbf{w}} y(\mathbf{x}^i, \mathbf{w})$$

Gradient of Error

$$\nabla_{\mathbf{w}} y = \left[ \frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

Gradient Descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} E$$

6.034 - Spring 03 - 18

Slide 7.3.18

We've seen that the simplest method for minimizing a differentiable function is **gradient descent** (or ascent if you're maximizing).

Recall that we are trying to find the weights that lead to a minimum value of training error. Here we see the gradient of the training error as a function of the weights. The descent rule is basically to change the weights by taking a small step (determined by the **learning rate**  $\eta$ ) in the direction opposite this gradient.

Note that the gradient of the error is simply the sum over all the training points of the error in the prediction for that point (given the current weights), which is the network output  $y$  minus the desired output  $y^i$ , times the gradient of the network output for that input and weight combination.

Slide 7.3.19

Let's look at a single sigmoid unit and see what the gradient descent rule would be in detail. We'll use the on-line version of gradient descent, that is, we will find the weight change to reduce the training error on a single training point. Thus, we will be neglecting the sum over the training points in the real gradient.

As we saw in the last slide, we will need the gradient of the unit's output with respect to the weights, that is, the vector of changes in the output due to a change in each of the weights.

The output ( $y$ ) of a single sigmoid unit is simply the output of the sigmoid function for the current activation (that is, total weighted input) of the unit. So, this output depends both on the values of the input features and the current values of the weights.

The gradient of this output function with respect to any of the weights can be found by an application of the chain rule of differentiation. The derivative of  $y$  with respect to  $w$  can be written as the product of the derivative with respect to  $z$  (the total activation) times the derivative of  $z$  with respect to the weight. The first term is the slope of the sigmoid function for the given input and weights, which we can write as  $ds(z)/dz$ . In this simple situation the total activation is a linear function of the weights, each with a coefficient corresponding to a feature value,  $x_i$ , for weight  $w_i$ .

### Gradient Descent Single Unit

$$\nabla_{\mathbf{w}} y = \left[ \frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

$$z = \sum_i w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} x_i$$

6.034 - Spring 03 - 19

So, the derivative of the activation with respect to the weight is just the input feature value,  $x_i$ .

### Gradient Descent Single Unit

$$\nabla_{\mathbf{w}} y = \left[ \frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

$$z = \sum_i^n w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} x_i$$

$$w_i \leftarrow w_i - \eta (y - y^m) \frac{\partial s(z)}{\partial z} x_i$$

$$\delta \equiv \frac{\partial E}{\partial z} = (y - y^m) \frac{\partial s(z)}{\partial z}$$

$\Delta w_i = -\eta \delta x_i$

Delta Rule

6.034 - Spring 03 • 20

Slide 7.3.20

Now, we can substitute this result into the expression for the gradient descent rule we found before (for a single point).

We will define a new quantity called delta, which is defined to be the derivative of the error with respect to a change in the activation  $z$ . We can think of this value as the "sensitivity" of the network output to a change in the activation of a unit.

The important result we get is that the change in the  $i^{\text{th}}$  weight is proportional to delta times the  $i^{\text{th}}$  input. This innocent looking equation has more names than you can shake a stick at: the delta rule, the LMS rule, the Widrow-Hoff rule, etc. Or you can simply call it the chain rule applied to the squared training error.

Slide 7.3.21

The derivative of the sigmoid plays a prominent role in these gradients, not surprisingly. Here we see that this derivative has a very simple form when expressed in terms of the **output** of the sigmoid. Then, it is just the output times 1 minus the output. We will use this fact liberally later.

### Derivative of the sigmoid

$$s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{ds(z)}{dz} = \frac{d}{dz} [(1 + e^{-z})^{-1}]$$

$$= [-(1 + e^{-z})^{-2}] [-e^{-z}]$$

$$= \left[ \frac{1}{1 + e^{-z}} \right] \left[ \frac{e^{-z}}{1 + e^{-z}} \right]$$

$$= s(z)(1 - s(z))$$

6.034 - Spring 03 • 21

### Gradient of Unit Output

$$\nabla_{\mathbf{w}} y = \left[ \frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

$$z = \sum_i^n w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

6.034 - Spring 03 • 22

Slide 7.3.22

Now, what happens if the input to our unit is not a direct input but the output of another unit and we're interested in the rate of change in  $y$  in response to a change to one of the weights in this second unit?

Slide 7.3.23

We use the chain rule again but now the change in the activation due to a change in the weight is a more complex expression: it is the product of the weight on the input times the rate of change in the output of the lower unit with respect to the weight. Notice that this new term is exactly of the same form as the one we are computing.

### Gradient of Unit Output

$$\nabla_{\mathbf{w}} y = \left[ \frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

$$z = \sum_i^n w_i x_i \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial y}{\partial w_{ji}} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_{ji}}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_{ji}}$$

$$= \frac{\partial s(z)}{\partial z} w_j \frac{\partial y_j}{\partial w_{ji}}$$

6.034 - Spring 03 • 23

### Gradient of Unit Output

$$\nabla_{\mathbf{w}} y = \left[ \frac{\partial y}{\partial w_1}, \dots, \frac{\partial y}{\partial w_n} \right]$$

$$z = \sum_i^n w_i x_i, \quad y = s(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_i}$$

$$= \frac{\partial s(z)}{\partial z} x_i$$

Base Case

$$\frac{\partial y}{\partial w_{ji}} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_{ji}}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_{ji}}$$

$$= \frac{\partial s(z)}{\partial z} w_j \frac{\partial y_j}{\partial w_{ji}}$$

Recursion

6.034 - Spring 03 • 24

Slide 7.3.24

We've just set up a recursive computation for the  $dy/dw$  terms. Note that these terms will be products of the slopes of the output sigmoid for the units times the weight on the input times a term of similar form for units below the input, until we get to the input with the weight we are differentiating with respect to. In the base case, we simply have the input value on that line, which could be one of the  $x_i$  or one of the  $y_j$ , since clearly the derivative of any unit with respect to  $w_i$  "below" the line with that weight will be zero.

Slide 7.3.25

Let's see how this works out for the simple case we've looked at before. There are two types of weights, the ones on the output unit, of the form,  $w_{*3}$ . And the weights on the two lower level units,  $w_{*1}$  and  $w_{*2}$ . The form of  $dy/dw$  for each of these two weights will be different as we saw in the last slide.

### Gradient of Error

$$E = \frac{1}{2} \sum_l (y(\mathbf{x}^l, \mathbf{w}) - y^l)^2$$

$$y = s(w_{13}s(w_{11}x_1 + w_{21}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02}) - w_{03})$$

6.034 - Spring 03 • 25

### Gradient of Error

$$E = \frac{1}{2} \sum_l (y(\mathbf{x}^l, \mathbf{w}) - y^l)^2$$

$$y = s(w_{13}s(w_{11}x_1 + w_{21}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02}) - w_{03})$$

$$\frac{\partial E}{\partial w_j} = (y - y^l) \frac{\partial y}{\partial w_j}$$

6.034 - Spring 03 • 26

Slide 7.3.26

Recall that in the derivative of the error (for a single instance) with respect to any of the weights, we get a term that measures the error at the output ( $y - y^l$ ) times the change in the output which is produced by the change in the weight ( $dy/dw$ ).

Slide 7.3.27

Let's pick weight  $w_{13}$ , that weights the output of unit 1 ( $y_1$ ) coming into the output unit (unit 3). What is the change in the output  $y_3$  as a result of a small change in  $w_{13}$ ? Intuitively, we should expect it to depend on the value of  $y_1$ , the "signal" on that wire since the change in the total activation when we change  $w_{13}$  is scaled by the value of  $y_1$ . If  $y_1$  were 0 then changing the weight would have no effect on the unit's output.

Changing the weight changes the activation, which changes the output. Therefore, the impact of the weight change on the output depends on the slope of the output (the sigmoid output) with respect to the activation. If the slope is zero, for example, then changing the weight causes no change in the output.

When we evaluate the gradient (using the chain rule), we see exactly what we expect -- the product of the sigmoid slope ( $dy/dz_3$ ) times the signal value  $y_1$ .

### Gradient of Error

$$E = \frac{1}{2} \sum_l (y(\mathbf{x}^l, \mathbf{w}) - y^l)^2$$

$$y = s(w_{13}s(w_{11}x_1 + w_{21}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02}) - w_{03})$$

$$\frac{\partial E}{\partial w_j} = (y - y^l) \frac{\partial y}{\partial w_j}$$

$$\frac{\partial y}{\partial w_{13}} = \frac{\partial y}{\partial z_3} \frac{\partial z_3}{\partial w_{13}} = \frac{\partial y}{\partial z_3} y_1$$

6.034 - Spring 03 • 27

### Gradient of Error

$$E = \frac{1}{2} \sum_i (y(\mathbf{x}^i, \mathbf{w}) - y^i)^2$$

$$y = s(w_{13}s(z_1) + w_{21}x_2 - w_{01}) + w_{23}s(w_{12}x_1 + w_{22}x_2 - w_{02}) - w_{03}$$

$$\frac{\partial E}{\partial w_j} = (y - y^i) \frac{\partial y}{\partial w_j}$$

$$\frac{\partial y}{\partial w_{13}} = \frac{\partial y}{\partial z_3} \frac{\partial z_3}{\partial w_{13}} = \frac{\partial y}{\partial z_3} y_1$$

$$\frac{\partial y}{\partial w_{11}} = \frac{\partial y}{\partial z_3} \frac{\partial z_3}{\partial w_{11}} = \frac{\partial y}{\partial z_3} \left( w_{13} \frac{\partial y_1}{\partial z_1} \frac{\partial z_1}{\partial w_{11}} \right) = \frac{\partial y}{\partial z_3} \left( w_{13} \frac{\partial y_1}{\partial z_1} x_1 \right)$$

6.034 - Spring 03 - 28

Slide 7.3.28

What happens when we pick a weight that's deeper in the net, say  $w_{11}$ ? Since that weight affects  $y_1$ , we expect that the change in the final output will be affected by the value of  $w_{13}$  and the slope of the sigmoid at unit 3 (as when we were changing  $w_{13}$ ). In addition, the change in  $y_1$  will depend on the value of the "signal" on the wire ( $x_1$ ) and the slope of the sigmoid at unit 1. Which is precisely what we see.

Note that in computing the gradients deeper in the net we will use some of the gradient terms closer to the output. For example, the gradient for weights on the inputs to unit 1 change the output by changing one input to unit 3 and so the final gradient depends on the behavior of unit 3. It is the realization of this reuse of terms that leads to an efficient strategy for computing the error gradient.

Slide 7.3.29

The cases we have seen so far are not completely general in that there has been only one path through the network for the change in a weight to affect the output. It is easy to see that in more general networks there will be multiple such paths, such as shown here.

This means that a weight can affect more than one of the inputs to a unit, and so we need to add up all the effects before multiplying by the slope of the sigmoid.

### Gradient of Unit Output

$$\frac{\partial y}{\partial w_{11}} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_{11}}$$

$$= \frac{\partial s(z)}{\partial z} \frac{\partial z}{\partial w_{11}}$$

$$= \frac{\partial s(z)}{\partial z} \left( w_{42} \frac{\partial y_2}{\partial w_{11}} + w_{43} \frac{\partial y_3}{\partial w_{11}} \right)$$

Recursion (more general)

A change in  $w_{11}$  affects the error via a change in  $y_1$ , which affects  $y_2$  and  $y_3$

6.034 - Spring 03 - 29

### Generalized Delta Rule

6.034 - Spring 03 - 30

Slide 7.3.30

In general we will be looking at networks connected in the fashion shown on the left, where the output of every unit at one level is connected to an input of every unit at the next level. We have not shown the bias inputs for each of the units, but they are there!

A word on notation. To avoid having to spell out the names of all the weights and signals in these networks, we will give each unit an index. The output of unit  $k$  is  $y_k$ . We will specify the weights on the inputs of unit  $k$  as  $w^{i \rightarrow k}$  where  $i$  is either the index of one of the inputs or another unit. Because of the "feedforward" connectivity we have adopted this terminology is unambiguous.

Slide 7.3.31

In this type of network we can define a generalization of the delta rule that we saw for a single unit. We still want to define the sensitivity of the training error (for an input point) to a change in the total activation of a unit. This is a quantity associated with the unit, independent of any weight. We can express the desired change in a weight that feeds into unit  $k$  as (negative of) the product of the learning rate, delta, for unit  $k$  and the value of the input associated with that weight.

The tricky part is the definition of delta. From our investigation into the form of  $dy/dw$ , the form of delta in the pink box should be plausible: the product of the slope of the output sigmoid times the sum of the products of weights and other deltas. This is exactly the form of the  $dy/dw$  expressions we saw before.

The clever part here is that by computing the deltas starting with that of the output unit and moving **backward** through the network we can compute all the deltas for every unit in the network in one pass (once we've computed all the  $y$ 's and  $z$ 's during a forward pass). It is this property that has led to the name of this algorithm, namely **backpropagation**.

It is important to remember that this is still the chain rule being applied to computing the gradient of the error. However, the computations have been arranged in a clever way to make computing the

### Generalized Delta Rule

$$\delta_j = \frac{\partial E}{\partial z_j}$$

$$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$$

$$\Delta w_{i \rightarrow j} = -\eta \delta_j y_i$$

$$\delta_4 = \frac{ds(z_4)}{dz_4} (\delta_5 w_{4 \rightarrow 5} + \delta_6 w_{4 \rightarrow 6})$$

$$\Delta w_{1 \rightarrow 4} = -\eta \delta_4 y_1$$

$$\Delta w_{2 \rightarrow 4} = -\eta \delta_4 y_2$$

6.034 - Spring 03 - 31

gradient efficient.

### Backpropagation

An efficient method of implementing gradient descent for neural networks

$w_{i \rightarrow j} = w_{i \rightarrow j} - \eta \delta_j y_i$  **Descent rule**

$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$  **Backprop rule**

$y_i$  is  $x_i$  for input layer

6.034 - Spring 03 • 32

#### Slide 7.3.32

Thus, the algorithm for computing the gradients we need to update the weights of a net, with maximal re-using of intermediate results is known as backpropagation.

The two simple formulas we need are the ones we have just seen. One tells us how to change a weight. This is a simple gradient descent formula, except that it says that the gradient of the error is of the form  $\delta_j$  times  $y_i$  where  $y_i$  is the signal on the wire with this weight, so it is either one of the inputs to the net or an output of some unit.

The delta of one unit is defined to be the slope of the sigmoid of that unit (for the current value of  $z$ , the weighted input) times the weighted sum of the deltas for the units that this unit feeds into.

#### Slide 7.3.33

The backprop algorithm starts off by assigning random, small values to all the weights. The reason we want to have small weights is that we want to be near the approximately linear part of the sigmoid function, which happens for activations near zero. We want to make sure that (at least initially) none of the units are saturated, that is, are stuck at 0 or 1 because the magnitude of the total input is too large (positive or negative). If we get saturation, the slope of the sigmoid is 0 and there will not be any meaningful information of which way to change the weight.

### Backpropagation

An efficient method of implementing gradient descent for neural networks

$w_{i \rightarrow j} = w_{i \rightarrow j} - \eta \delta_j y_i$  **Descent rule**

$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$  **Backprop rule**

$y_i$  is  $x_i$  for input layer

1. Initialize weights to small random values

6.034 - Spring 03 • 33

### Backpropagation

An efficient method of implementing gradient descent for neural networks

$w_{i \rightarrow j} = w_{i \rightarrow j} - \eta \delta_j y_i$  **Descent rule**

$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$  **Backprop rule**

$y_i$  is  $x_i$  for input layer

1. Initialize weights to small random values
2. Choose a random sample input feature vector

6.034 - Spring 03 • 34

#### Slide 7.3.34

Now we pick a sample input feature vector. We will use this to define the gradients and therefore the weight updates. Note that by updating the weights based on one input, we are introducing some randomness into the gradient descent. Formally, gradient descent on an error function defined as the sum of the errors over all the input instances should be the sum of the gradients over all the instances. However, backprop is typically implemented as shown here, making the weight change based on each feature vector. We will have more to say on this later.

#### Slide 7.3.35

Now that we have weights and inputs, we can do a **forward propagation**, that is, we can compute the values of all the  $z$ 's and  $y$ 's, that is, the weighted inputs and the outputs for all the units. We will need these values, so let's remember them for each unit.

### Backpropagation

An efficient method of implementing gradient descent for neural networks

$w_{i \rightarrow j} = w_{i \rightarrow j} - \eta \delta_j y_i$  **Descent rule**

$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k w_{j \rightarrow k}$  **Backprop rule**

$y_i$  is  $x_i$  for input layer

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input ( $z_j$ ) and output ( $y_j$ ) for each unit (forward prop)

6.034 - Spring 03 • 35

### Backpropagation

An efficient method of implementing gradient descent for neural networks

$W_{i \rightarrow j} = W_{i \rightarrow j} - \eta \delta_j y_i$     **Descent rule**

$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k W_{j \rightarrow k}$     **Backprop rule**

y<sub>i</sub> is x<sub>i</sub> for input layer

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input (  $z_j$  ) and output (  $y_j$  ) for each unit (forward prop)
4. Compute  $\delta_n$  for output layer
 
$$\delta_n = \frac{ds(z_n)}{dz_n} (y_n - y_n^m) = y_n(1 - y_n)(y_n - y_n^m)$$

6.034 - Spring 03 • 36

Slide 7.3.36

Now, we start the process of computing the deltas. First we do it for the output units, using the formula shown here, that is, the product of the gradient of the sigmoid at the output unit times the error for that unit.

Slide 7.3.37

Then we compute the deltas for the other units at the preceding layer using the backprop rule.

### Backpropagation

An efficient method of implementing gradient descent for neural networks

$W_{i \rightarrow j} = W_{i \rightarrow j} - \eta \delta_j y_i$     **Descent rule**

$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k W_{j \rightarrow k}$     **Backprop rule**

y<sub>i</sub> is x<sub>i</sub> for input layer

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input (  $z_j$  ) and output (  $y_j$  ) for each unit (forward prop)
4. Compute  $\delta_n$  for output layer
 
$$\delta_n = \frac{ds(z_n)}{dz_n} (y_n - y_n^m) = y_n(1 - y_n)(y_n - y_n^m)$$
5. Compute  $\delta_j$  for all preceding layers by backprop rule

6.034 - Spring 03 • 37

### Backpropagation

An efficient method of implementing gradient descent for neural networks

$W_{i \rightarrow j} = W_{i \rightarrow j} - \eta \delta_j y_i$     **Descent rule**

$\delta_j = \frac{ds(z_j)}{dz_j} \sum_k \delta_k W_{j \rightarrow k}$     **Backprop rule**

y<sub>i</sub> is x<sub>i</sub> for input layer

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input (  $z_j$  ) and output (  $y_j$  ) for each unit (forward prop)
4. Compute  $\delta_n$  for output layer
 
$$\delta_n = \frac{ds(z_n)}{dz_n} (y_n - y_n^m) = y_n(1 - y_n)(y_n - y_n^m)$$
5. Compute  $\delta_j$  for all preceding layers by backprop rule
6. Compute weight change by descent rule (repeat for all weights)

6.034 - Spring 03 • 38

Slide 7.3.38

With the deltas and the unit outputs in hand, we can update the weights using the descent rule.

Slide 7.3.39

We can see what is involved in doing the simple three-unit example we saw earlier. Here we see the simple expressions for the deltas and the weight updates. Note that each expression involves data local to a particular unit, you don't have to look around summing things over the whole network, the delta's capture the recursion that we observed earlier. It is for this reason, simplicity, locality and, therefore, efficiency that backpropagation has become the dominant paradigm for training neural nets.

As mentioned before, however, the difficult choice of the learning rate and relatively slow convergence to a minimum are substantial drawbacks. Thousands of variations of backprop exist, aimed at addressing these drawbacks. More sophisticated minimization strategies, for example, do a search along the gradient direction (or related directions) to find a step that achieves a reduction in the error. Nevertheless, for these methods one still needs to derive the gradient of the network and a backprop-like computation can be used to do that.

### Backpropagation Example

First do forward propagation:  
Compute  $z_i$  and  $y_i$  given  $x_i$ ,  $W_{ij}$

$$\delta_3 = y_3(1 - y_3)(y_3 - y_3^m)$$

$$\delta_2 = y_2(1 - y_2)\delta_3 W_{23}$$

$$\delta_1 = y_1(1 - y_1)\delta_3 W_{13}$$

$$W_{03} = W_{03} - r\delta_3(-1)$$

$$W_{02} = W_{02} - r\delta_2(-1)$$

$$W_{01} = W_{01} - r\delta_1(-1)$$

$$\rightarrow W_{13} = W_{13} - \eta\delta_3 y_1$$

$$W_{12} = W_{12} - \eta\delta_2 x_1$$

$$W_{11} = W_{11} - \eta\delta_1 x_1$$

$$W_{23} = W_{23} - \eta\delta_3 y_2$$

$$W_{22} = W_{22} - \eta\delta_2 x_2$$

$$W_{21} = W_{21} - \eta\delta_1 x_2$$

Compare to the direct derivation earlier      Note that all computations are local!

6.034 - Spring 03 • 39

## 6.034 Notes: Section 7.4

### Slide 7.4.1

Now that we have looked at the basic mathematical techniques for minimizing the training error of a neural net, we should step back and look at the whole approach to training a neural net, keeping in mind the potential problem of overfitting.

We need to worry about overfitting because of the generality of neural nets and the proliferation of parameters associated even with a relatively simple net. It is easy to construct a net that has more parameters than there are data points. Such nets, if trained so as to minimize the training error without any additional constraints, can very easily overfit the training data and generalize very poorly. Here we look at a methodology that attempts to minimize that danger.

### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

6.034 - Spring 03 • 1



### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
  - Training set – used for picking weights
  - Validation set – used to stop training
  - Test set – used to evaluate performance

6.034 - Spring 03 • 2



### Slide 7.4.2

The first step (in the ideal case) is to separate the data into three sets. A training set for choosing the weights (using backpropagation), a validation set for deciding when to stop the training and, if possible, a separate set for evaluating the final results.

### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
  - Training set – used for picking weights
  - Validation set – used to stop training
  - Test set – used to evaluate performance
2. Pick random, small weights as initial values

6.034 - Spring 03 • 3



### Slide 7.4.3

Then we pick a set of random small weights as the initial values of the weights. As we explained earlier, this reduces the chance that we will saturate any of the units initially.

### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
  - Training set – used for picking weights
  - Validation set – used to stop training
  - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.

6.034 - Spring 03 • 4

#### Slide 7.4.4

Then we perform the minimization of the training error, for example, using backpropagation. This will generally involve going through the input data and making changes to the weights many times. A common term used in this context is the **epoch**, which indicates how many times the algorithm has gone through every point in the training data. So, for example, one can plot the training error as a function of the training epoch. We will see this later.

#### Slide 7.4.5

An important point is that we do not want to simply keep going until we reduce the training error to its minimum value. This is likely to overfit the training data. Instead, we can use the performance on the validation set as a way of deciding when to stop; we want to stop when we get best performance on the validation set. This is likely to lead to better generalization. We will look at this in more detail momentarily.

This type of "early termination" keeps the weights relatively small. Keeping the weights small is a strategy for reducing the size of the hypothesis space. It's informally related to the idea of maximizing the margin by minimizing the magnitude of the weight vector in an SVM. It also reduces the variance of the hypothesis since it limits the impact that any particular data point can have on the output.

### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
  - Training set – used for picking weights
  - Validation set – used to stop training
  - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).

6.034 - Spring 03 • 5

### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
  - Training set – used for picking weights
  - Validation set – used to stop training
  - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).
5. Repeat training (from step 2) several times (avoid local minima)

6.034 - Spring 03 • 6

#### Slide 7.4.6

In neural nets we do not have the luxury that we had in SVMs of knowing that we have found the global optimum after we finished learning. In neural nets, there are many local optima and backprop (or any other minimization strategy) can only guarantee finding a local optimum (and even this guarantee depends on careful choice of learning rate). So, it is often useful to repeat the training several times to see if a better result can be found. However, even a single round of training can be very expensive so this may not be feasible.

#### Slide 7.4.7

Once we have a final set of weights, we can use them once on a held out test set to estimate the expected behavior on new data. Note the emphasis on doing this once. If we change the weights to improve this behavior, then we no longer have a held out set.

### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
  - Training set – used for picking weights
  - Validation set – used to stop training
  - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).
5. Repeat training (from step 2) several times (avoid local minima)
6. Use best weights to compute error on test set, which is estimate of performance on new data. Do not repeat training to improve this.

6.034 - Spring 03 • 7

### Training Neural Nets without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with  $m$  weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
  - Training set – used for picking weights
  - Validation set – used to stop training
  - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set.
4. Stop when error on validation set reaches a minimum (to avoid overfitting).
5. Repeat training (from step 2) several times (avoid local minima)
6. Use best weights to compute error on test set, which is estimate of performance on new data. Do not repeat training to improve this.

Can use cross-validation if data set is too small to divide into three subsets.

6.034 - Spring 03 • 8

#### Slide 7.4.8

In many cases, one doesn't have the luxury of having these separate sets, due to scarcity of data, in which case cross-validation may be used as a substitute.

#### Slide 7.4.9

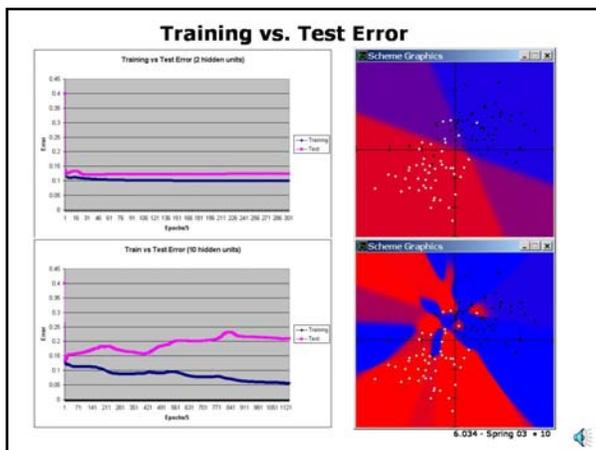
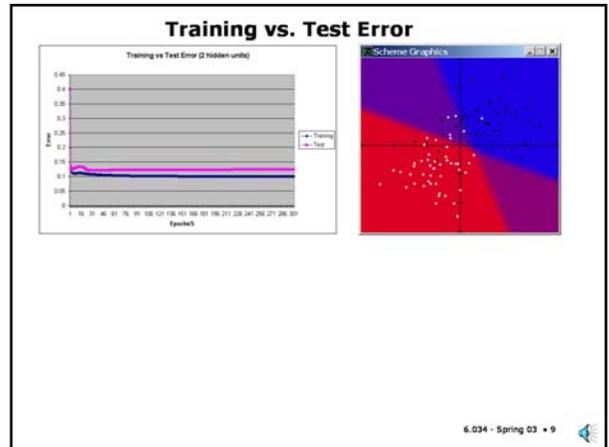
Let's look at the termination/overfitting issue via some examples.

Here we see the behavior of a small neural net (two inputs, two hidden units and one output) when trained on the data shown in the picture on the right. The white and black points represent 50 instances from each of two classes (drawn from Gaussian distributions with different centers). An additional 25 instances each (drawn from the same distributions) have been reserved as a test set.

As you can see, the point distributions overlap and therefore the net cannot fully separate the data. The red region represents the area where the net's output is close to 1 and the blue region represents where the output is close to 0. Intermediate colors represent intermediate values.

The error on the training set drops quickly at the beginning and does not drop much after that. The error on the test set behaves very similarly except that it is a bit bigger than the error on the training set. This is to be expected since the detailed placement of points near the boundaries will be different in the test set.

The behavior we see here is a good one; we have not overfit the data.



#### Slide 7.4.10

Here we see the behavior of a larger neural net (with 10 hidden units) on the same data.

You can see that the training error continues to drop over a much longer set of training epochs. In fact, the error goes up slightly sometimes, then stagnates and drops again. This is typical behavior for backprop.

Note, however, that during most of the time that the training error is dropping, the test error is **increasing**. This indicates that the net is overfitting the data. If you look at the net output at the end of training, you can see what is happening. The net has constructed a baroque decision boundary to capture precisely the placement of the different instances in the training set. However, the instances in the test set are (very) unlikely to fall into that particular random arrangement.

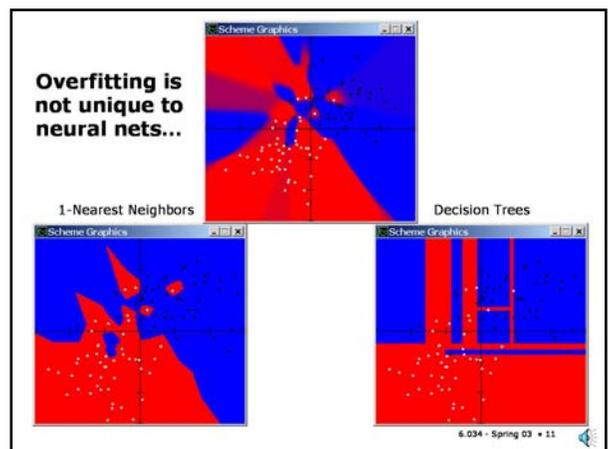
So, in fact, all that extra work in fitting the training set is wasted. Note that the test error with this net is much higher than with the simpler net. If we had used a validation set, we could have stopped training before it went too far astray.

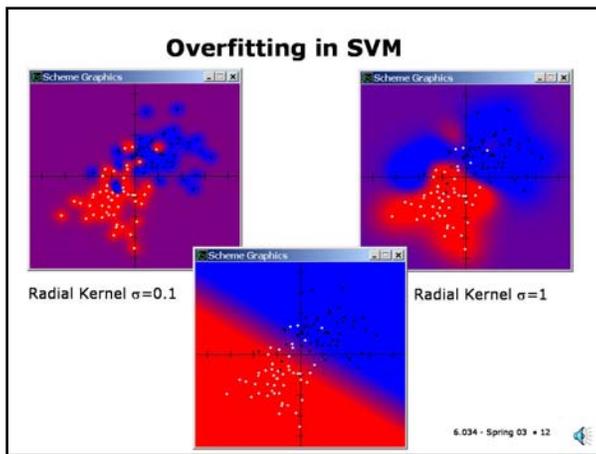
#### Slide 7.4.11

Note that this type of overfitting is not unique to neural nets. In this slide you can see the behavior of 1-nearest-neighbor and decision trees on the same data. Both fit it perfectly and produce classifiers that are just as unlikely to generalize to new data.

For K-nearest-neighbors, on this type of data one would want to use a value of K greater than 1. For decision trees one would want to prune back the tree somewhat. These decisions could be based on the performance on a held out validation set.

Similarly, for neural nets, one would want to choose the number of units and the stopping point based on performance on validation data.





Slide 7.4.12

Even SVMs, which have relatively few parameters and a well-deserved reputation for being resistant to overfitting can overfit. In the center panel we see the output for a linear SVM. This is, in fact, the optimal type of separator for this data. On the upper left we see a fairly severe overfitting stemming from the choice of a too-small sigma for a radial kernel. On the right is the result from a larger choice of sigma (and a relatively high C). Where the points are densest, this actually approximates the linear boundary but then deviates wildly in response to an outlier near (-2, 0). This illustrates how the choice of kernel can affect the generalization ability of an SVM classifier.

Slide 7.4.13

We mentioned earlier that backpropagation is an on-line training algorithm, meaning that it changes the weights for each input instance. Although this is not a "correct" implementation of gradient descent for the total training error, it is often argued that the randomization effect is beneficial as it tends to push the system out of some shallow local minima. In any case, the alternative "off-line" approach of actually adding up the gradients for all the instances and changing the weights based on that complete gradient is also used and has some advantages for smaller systems. For larger nets and larger datasets, the on-line approach has substantial performance advantages.

### On-line vs off-line

There are two approaches to performing the error minimization:

- **On-line training** – present  $\mathbf{x}^i$  and  $y^i$  (chosen randomly from the training set). Change the weights to reduce the error on this instance. Repeat.
- **Off-line training** – change weights to reduce the total error on training set (sum over all instances).

On-line training is an approximation to gradient descent since the gradient based on one instance is "noisy" relative to the full gradient (based on all instances). This can be beneficial in pushing the system out of shallow local minima.

6.034 - Spring 03 • 13

### Momentum

$$W_{i \rightarrow j}^{t+1} = W_{i \rightarrow j}^t - \eta \delta_j y_i \quad \text{Standard backprop descent}$$

$$\Delta W_{i \rightarrow j}^t = W_{i \rightarrow j}^{t+1} - W_{i \rightarrow j}^t = -\eta \delta_j y_i \quad \text{Rewrite to define change in weights at time } t$$

6.034 - Spring 03 • 14

Slide 7.4.14

We have mentioned the difficulty of picking a learning rate for backprop that balances, on the one hand, the desire to move speedily towards a minimum by using a large learning rate and, on the other hand, the need to avoid overstepping the minimum and possibly getting into oscillations because of a too-large learning rate. One approach to balancing these is to effectively adjust the learning rate based on history. One of the original approaches for this is to use a **momentum** term in backprop.

Here is the standard backprop gradient descent rule, where the change to the weights is proportional to delta and y.

Slide 7.4.15

We can keep around the most recent change to the weights (at time  $t-1$ ) and add some fraction of that weight change to the current delta. The fraction, alpha, is the momentum weight.

The basic idea is that if the changes to the weights have had a consistent sign, then the effect is to have a larger step size in the weight. If the sign has been changing, then the net change may be smaller.

Note that even if the delta times y term is zero (denoting a local minimum in the error), in the presence of a momentum term, the change in the weights will not necessarily be zero. So, this may cause the system to move through a shallow minimum, which may be good. However, it may also lead to undesirable oscillations in some circumstances.

In practice, choosing a good value of momentum for a problem can be nearly as hard as choosing a learning rate and it's one more parameter to twiddle with.

Momentum is not that popular a technique anymore; people will tend to use more complex search strategies to ensure convergence to a local minimum.

### Momentum

$$W_{i \rightarrow j}^{t+1} = W_{i \rightarrow j}^t - \eta \delta_j y_i \quad \text{Standard backprop descent}$$

$$\Delta W_{i \rightarrow j}^t = W_{i \rightarrow j}^{t+1} - W_{i \rightarrow j}^t = -\eta \delta_j y_i \quad \text{Rewrite to define change in weights at time } t$$

$$\Delta W_{i \rightarrow j}^t = -\eta \delta_j y_i + \alpha \Delta W_{i \rightarrow j}^{t-1} \quad \text{Adding a momentum term, which adds in a fraction of the weight change at the previous iteration.}$$

- Momentum can gradually increase step size when gradient is unchanging.
- Can help step through shallow local minima
- One more parameter to twiddle... not used much anymore

6.034 - Spring 03 • 15

### Input Representation

- All the signals in a neural net are  $[0, 1]$ . Input values should also be scaled to this range (or approximately so) so as to speed training.

6.034 - Spring 03 • 16

### Slide 7.4.16

One question that arises in neural nets (and many other machine learning approaches) is how to represent the input data. We have discussed some of these issues before.

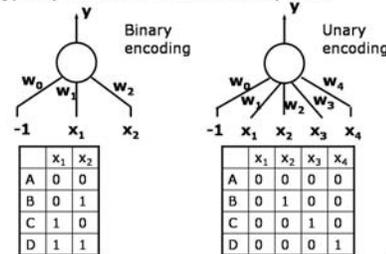
One issue that is prominent in neural nets is the fact that the behavior of these nets are dependent on the scale of the input. In particular, one does not want to saturate the units, since at that point it becomes impossible to train them. Note that all "signals" in a sigmoid-unit neural net are in the range  $[0,1]$  because that is the output range of the sigmoids. It is best to keep the range of the inputs in that range as well. Simple normalization (subtract the mean and divide by the standard deviation) will almost do that and is adequate for most purposes.

### Slide 7.4.17

Another issue has to do with the representation of discrete data (also known as "categorical" data). You could think of representing these as either unary or binary numbers. Binary numbers are generally a bad choice; unary is much preferable if the number of values is small since it decouples the values completely.

### Input Representation

- All the signals in a neural net are  $[0, 1]$ . Input values should also be scaled to this range (or approximately so) so as to speed training.
- If the input values are discrete, e.g.  $\{A, B, C, D\}$  or  $\{1, 2, 3, 4\}$ , they need to be coded in unary form.



6.034 - Spring 03 • 17

### Output Representation

- A neural net with a single sigmoid output unit is aimed at binary classification. Class is 0 if  $y < 0.5$  and 1 otherwise.
- For multi-class problems
  - Can use one output per class (unary encoding)
  - There may be confusing outputs (two outputs  $> 0.5$  in unary encoding).
  - More sophisticated method is to use special **softmax** units, which force outputs to sum to 1.

6.034 - Spring 03 • 18

### Slide 7.4.18

Similar questions arise at the output end. So far, we have focused on binary classification. There, the representation is clear - a single output and we treat an output of 0.5 as the dividing value between one class and the other.

For multi-class problems, one can have multiple output units, for example, each aimed at recognizing one class, sharing the hidden units with the other classes.

One difficulty with this approach is that there may be ambiguous outputs, e.g. two values above the 0.5 threshold when using a unary encoding. How do we treat such a case? One reasonable approach is to choose the class with the largest output.

A more sophisticated method is to introduce a new type of unit (called "softmax") that forces the sum of the unary outputs to add to 1. One can then interpret the network outputs as the probabilities that the input belongs to each of the classes.

### Slide 7.4.19

Another detail to consider in training is what to use as the desired (target) value for the network outputs. We have been talking as if we would use 0 or 1 as the targets. The problem with this is that those are the asymptotic values of the sigmoid only achieved for infinite values of the weights. So, if we were to attempt to train a network until the weights stop changing, then we'd be in trouble. It is common to use values such as 0.1 and 0.9 instead of 0 and 1 during neural network training.

In practice, however, the usual termination for training a network is when the training or, preferably, validation error either achieves an acceptable level, reaches a minimum or stops changing significantly. These outcomes generally happen long before we run the risk of the weights becoming infinite.

### Target Value

- During training it is impossible for the outputs to reach 0 or 1 (with finite weights).
- Customary to use 0.1 and 0.9 as targets
- But, most termination criteria, e.g. small change in training or validation error will stop training before targets are reached.

6.034 - Spring 03 • 19

### Regression

- A sigmoid output unit is not suitable for regression, since sigmoids are designed to change quickly from 0 to 1.
- For regression, we want a linear output unit, that is, remove the output non-linearity.
- The rest of the net still retains the sigmoid units.

6.034 - Spring 03 • 20

#### Slide 7.4.20

Neural nets can also do regression, that is, produce an output which is a real number outside the range of 0 to 1, for example, predicting the age of death as a function of packs of cigarettes smoked. However, to do regression, it is important that one does not try to predict a continuous value using an output sigmoid unit. The sigmoid is basically a soft threshold with a limited dynamic range. When doing regression, we want the output unit to be linear, that is, simply remove the sigmoid non-linearity and have the unit returned a weighted sum of its inputs.

#### Slide 7.4.21

One very interesting application of neural networks is the ALVINN project from CMU. The project was the brainchild of Dean Pomerleau. ALVINN is an automatic steering system for a car based on input from a camera mounted on the vehicle. This system was successfully demonstrated on a variety of real roads in actual traffic. A successor to ALVINN, which unfortunately was not based on

### ALVINN steers on highways

[http://www.ri.cmu.edu/projects/project\\_160.html](http://www.ri.cmu.edu/projects/project_160.html)

Images removed due to copyright restrictions.

Dean Pomerleau  
CMU

6.034 - Spring 03 • 21

### ALVINN steers on highways

[http://www.ri.cmu.edu/projects/project\\_160.html](http://www.ri.cmu.edu/projects/project_160.html)

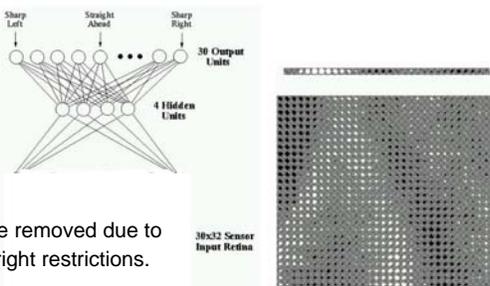


Image removed due to copyright restrictions.

6.034 - Spring 03 • 22

#### Slide 7.4.22

The ALVINN neural network is shown here. It has 960 inputs (a 30x32 array derived from the pixels of an image), four hidden units and 30 output units (each representing a steering command). On the right you can see a pattern of the brightnesses of the input pixels and right above that you can see the pattern of the outputs, representing a "steer left" command.

#### Slide 7.4.23

One of the most interesting issues that came up in the ALVINN project was the problem of obtaining training data. It's not difficult to get images of somebody driving correctly down the middle of the road, but if that were **all** that ALVINN could do, then it would not be safe to let it on the road. What if an obstacle arose or there was a momentary glitch in the control system or a bump in the road got you off center? It is important that ALVINN be able to recover and steer the vehicle back to the center.

The researchers considered having the vehicle drive in a wobbly path during training, but that posed the danger of having the system learn to drive that way. They came up with a clever solution. Simulate what the sensory data would have looked like had ALVINN been off-center or headed in a slightly wrong direction and also, for each such input, simulate what the steering command should have been.

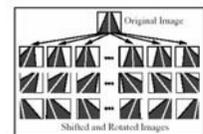
Now, you don't want to generate simulated images from scratch, as in a video game, since they are insufficiently realistic. What they did, instead, is transform the real images and fill in the few missing pixels by a form of "interpolation" on the actual pixels. The results were amazingly good.

However, it turned out that once one understood that this whole project was possible and one

### ALVINN steers on highways

[http://www.ri.cmu.edu/projects/project\\_160.html](http://www.ri.cmu.edu/projects/project_160.html)

Image removed due to copyright restrictions.



- Problem: Getting enough diversity in training set
- Answer: Transform sensor image and steering direction

6.034 - Spring 03 • 23

understood what ALVINN was learning, it became possible to build a special purpose system that was faster and more reliable and involved no explicit on-line learning. This is not an uncommon side-effect of a machine-learning project.

### **Some observations...**

- Although Neural Nets kicked off the current phase of interest in machine learning, they are extremely problematic...
  - Too many parameters (weights, learning rate, momentum, etc)
  - Hard to choose the architecture
  - Very slow to train
  - Easy to get stuck in local minima
- Interest has shifted to other methods, such as support vector machines, which can be viewed as variants of perceptrons (with a twist or two).

6.034 - Spring 03 - 24



### **Slide 7.4.24**

Neural nets are largely responsible for the current interest in statistical methods for machine learning. For a while, they were wildly popular in the research community and saw many applications. Over time, the enthusiasm has waned somewhat.

The big problem with neural nets is that they are very complex and have many many parameters that have to be chosen. Furthermore, training them is a bit of a nightmare. So, recent interest has shifted towards methods that are simpler to use and can be characterized better. For example, support vector machines, which at one level can be viewed as a variation of the same perceptrons that neural nets superseded, is the current darling of the machine learning research and application community.