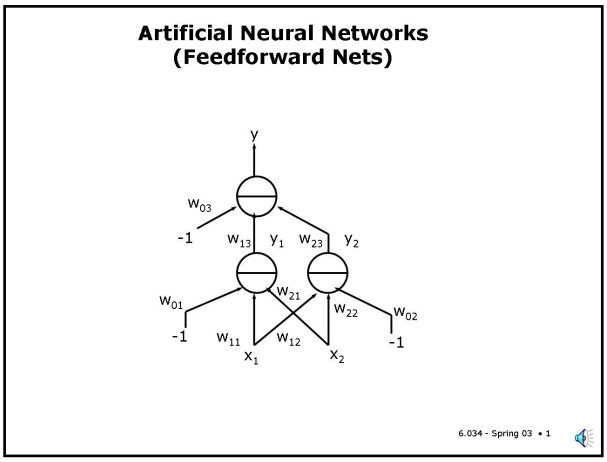
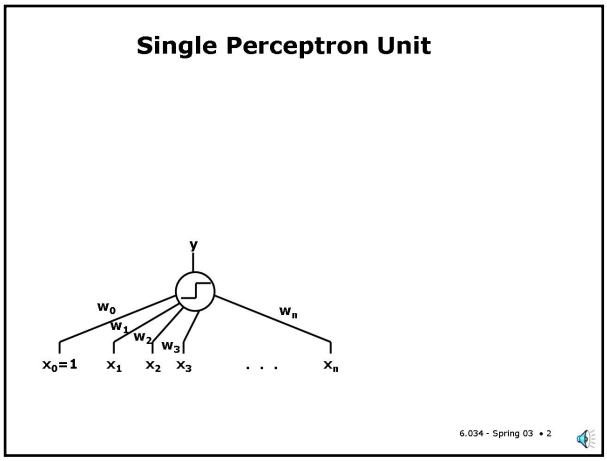
6.034 Notes: Section 4.1

Slide 4.1.1



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

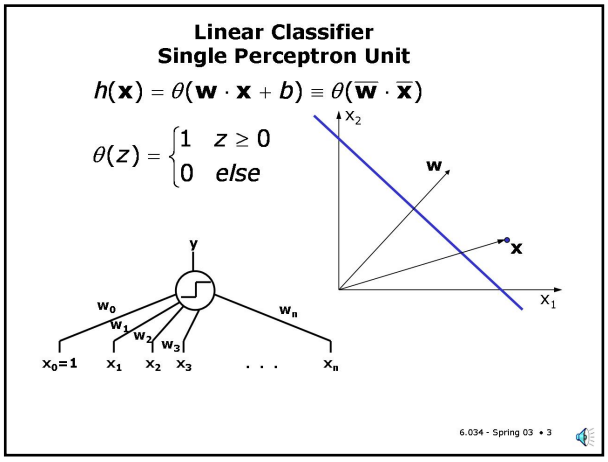
The basic idea in neural nets is to define interconnected networks of simple units (let's call them "artificial neurons") in which each connection has a weight. Weight w_{ij} is the weight of the i^{th} input into unit j. The networks have some inputs where the feature values are placed and they compute 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 4.1.2

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.

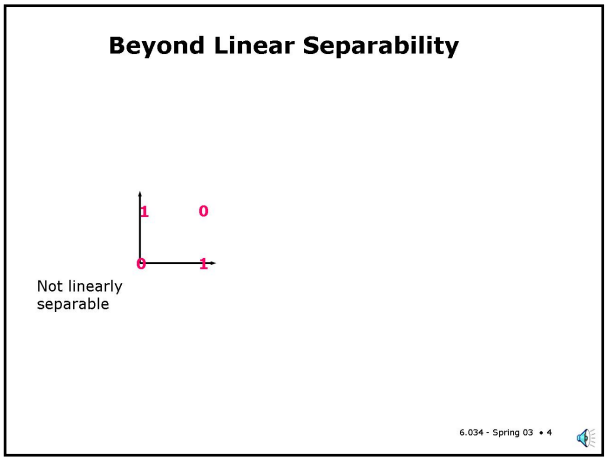
Slide 4.1.3



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.

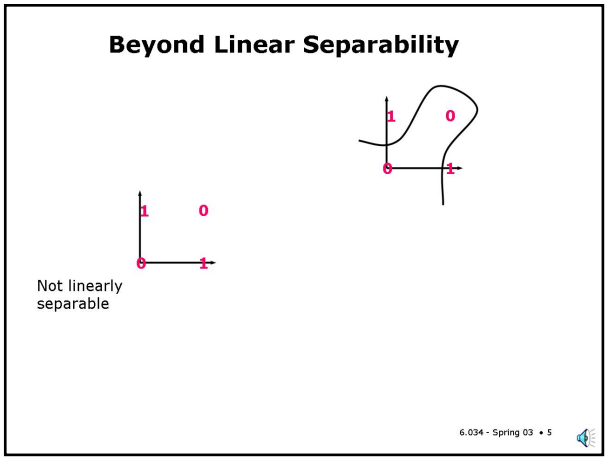


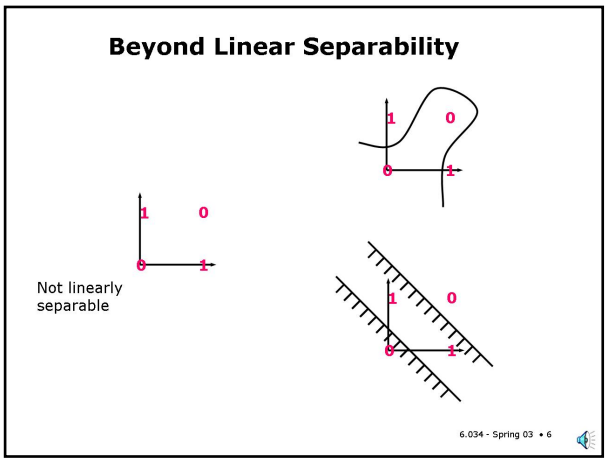
Slide 4.1.4

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 4.1.5

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.





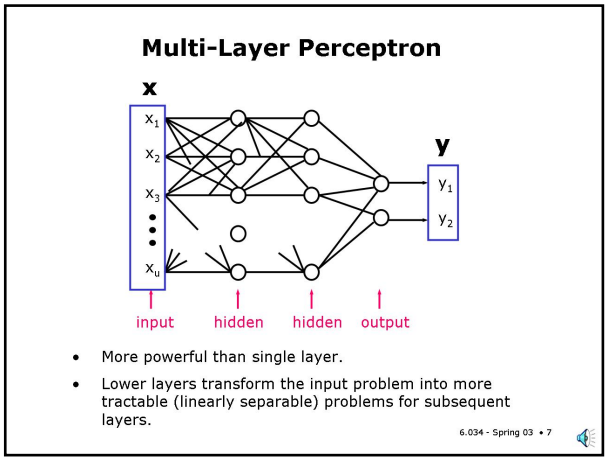
Slide 4.1.6

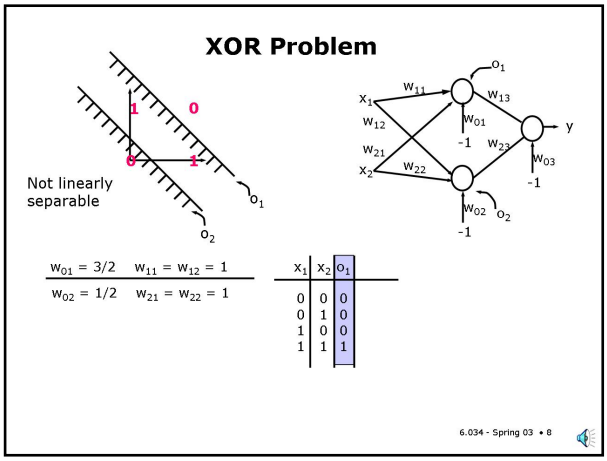
We will now explore a different approach (although later we will also introduce non-linear mappings). What about if we consider more than one linear separator and combine their outputs; can we get a more powerful classifier?

Slide 4.1.7

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.





Slide 4.1.8

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.

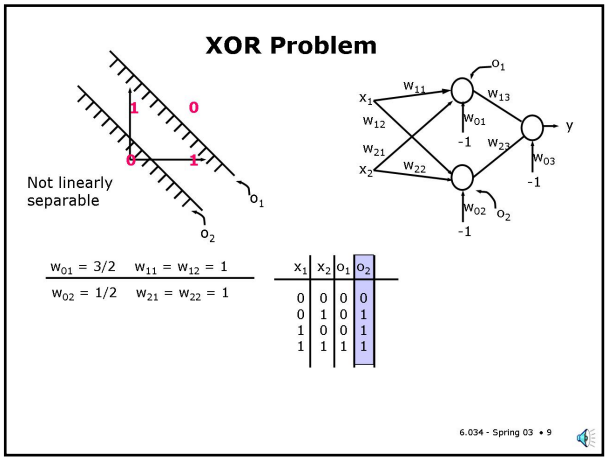
We see that each hidden unit defines its own "decision boundary" and the output from each of these units is fed to the output unit, which returns a solution to the whole problem. Let's look in detail at each of these boundaries and its effect.

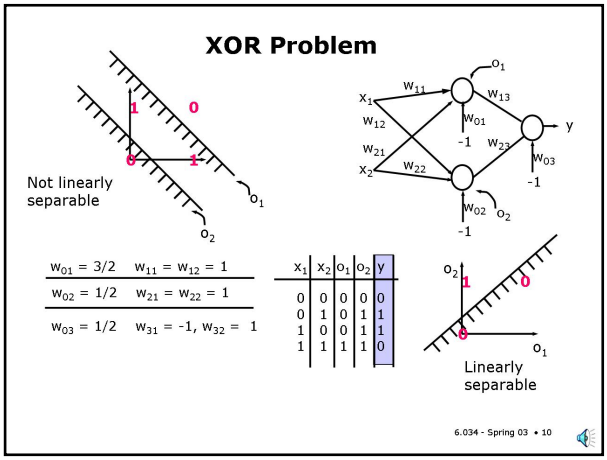
Note that each of the weights in the first layer, except for the offsets, has been set to 1. So, we know that the decision boundaries are going to have normal vectors equal to [1 1], that is, pointing up and to the right, as shown in the diagram. The values of the offsets show that the hidden unit labeled 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 4.1.9

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 4.1.10

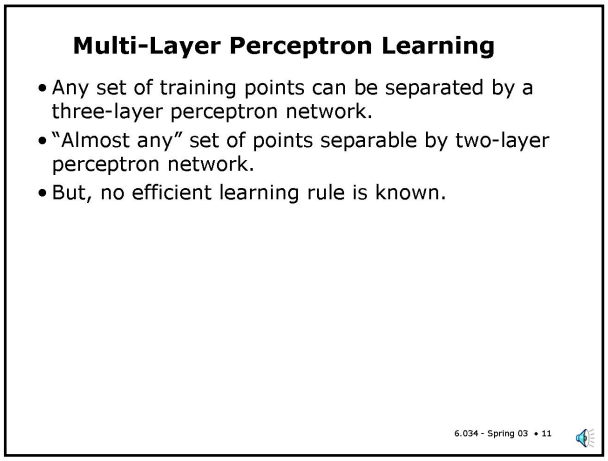
On the lower right, we see that the problem has been mapped into a linearly separable problem in the space of the outputs of the hidden units. We can now easily find a linear separator, for example, the one shown here. This mapping is where the power of the multi-layer perceptron comes from.

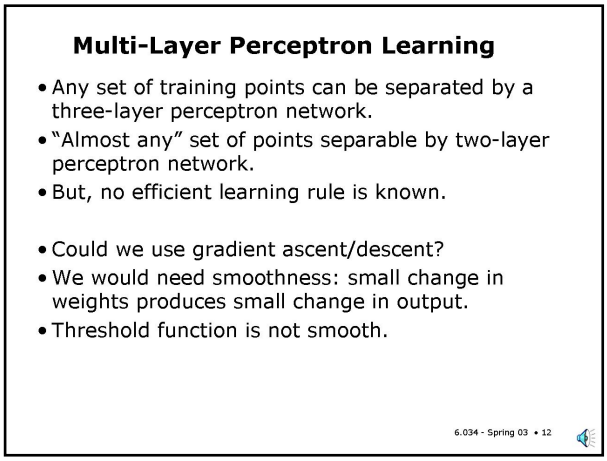
Slide 4.1.11

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.



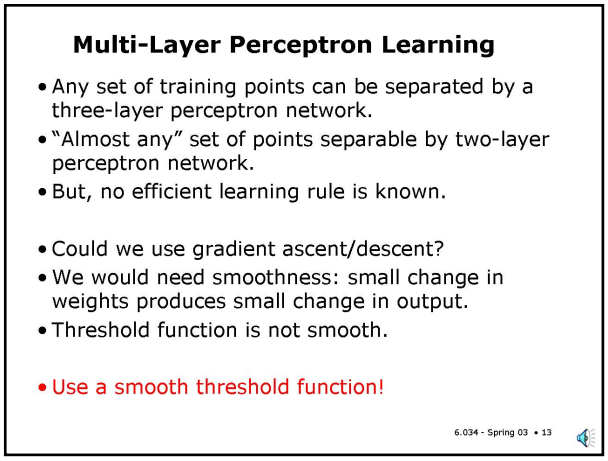


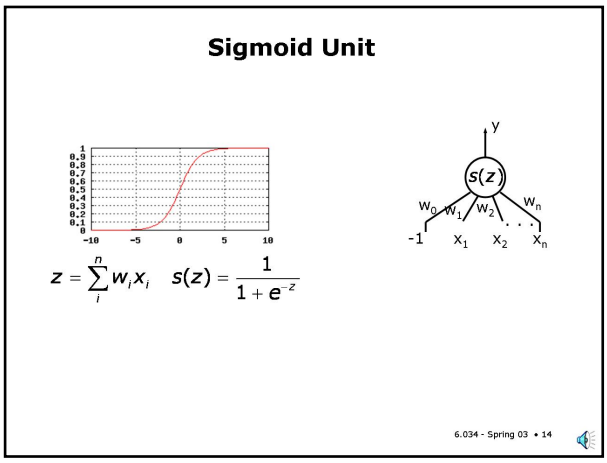
Slide 4.1.12

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 4.1.13

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.





Slide 4.1.14

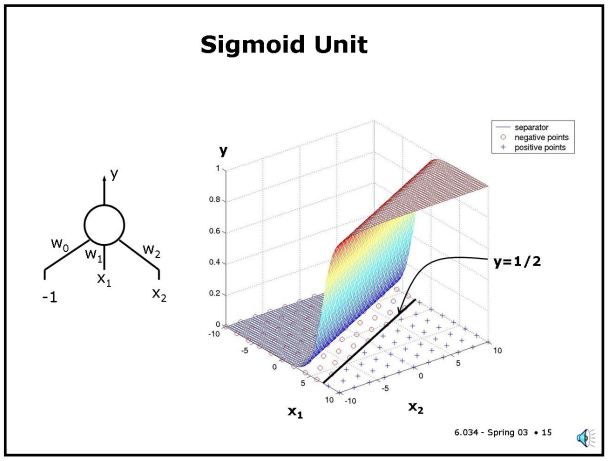
The classic "soft threshold" that is used in neural nets is referred to as a "sigmoid" (meaning S-like) and is shown here. The variable z is the "total input" or "activation" of a neuron, that is, the weighted sum of all of its inputs.

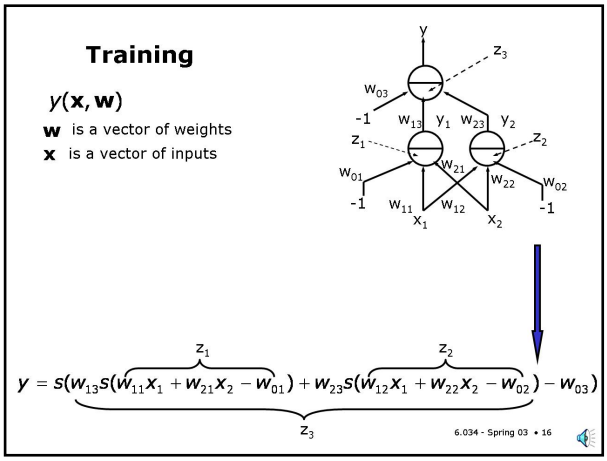
Note that when the input (z) is 0, the sigmoid's value is 1/2. The sigmoid is applied to the weighted inputs (including the threshold value as before). There are actually many different types of sigmoids that can be (and are) used in neural networks. The sigmoid shown here is actually called the **logistic**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

Slide 4.1.15

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.





Slide 4.1.16

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.

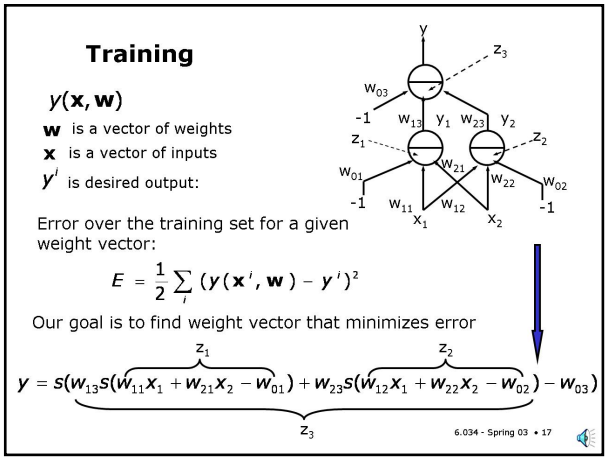
The output of a multi-layer net of sigmoid units is a function of two vectors, the inputs (x) and the weights (w). An example of what that function looks like for a simple net is shown along the bottom, where s() is whatever output function we are using, for example, the logistic function we saw in the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

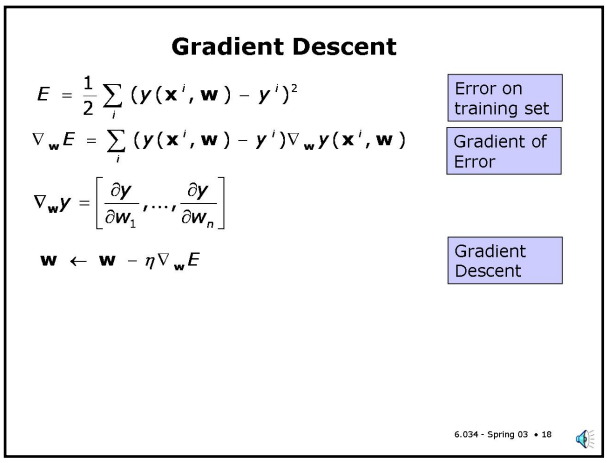
The output of this function (y) varies smoothly with changes in the input and, importantly, with changes in the weights. In fact, the weights and inputs both play similar roles in the function.

Slide 4.1.17

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.





Slide 4.1.18

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.

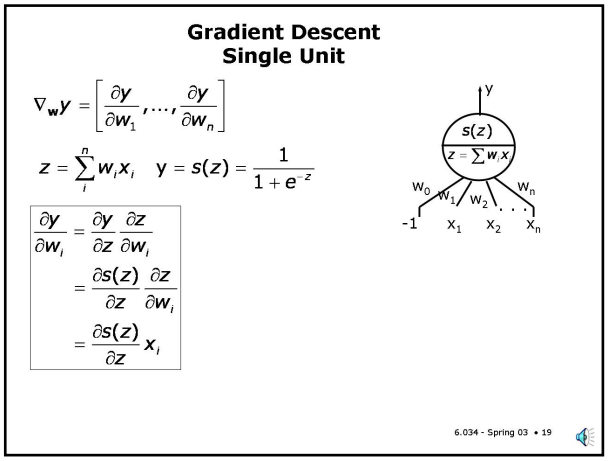
Slide 4.1.19

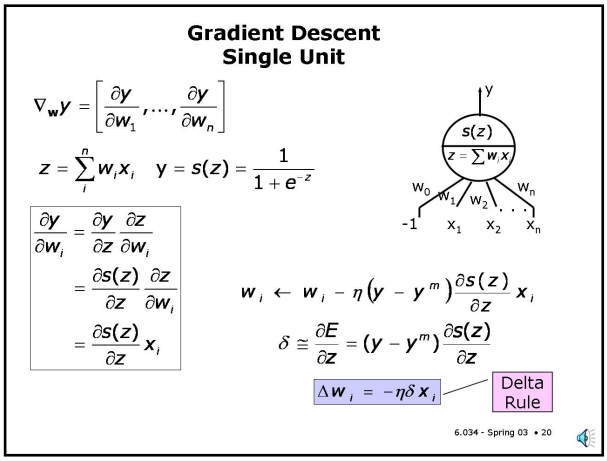
Let's look at a single sigmoid unit and see what the gradient descent rule would be in detail. We'll use the on-line version of gradient descent, that is, we will find the weight change to reduce the training error on a single training point. Thus, we will be neglecting the sum over the training points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





Slide 4.1.20

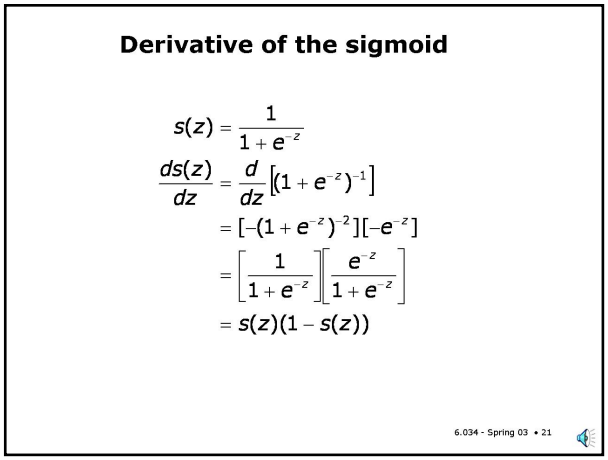
Now, we can substitute this result into the expression for the gradient descent rule we found before (for a single point).

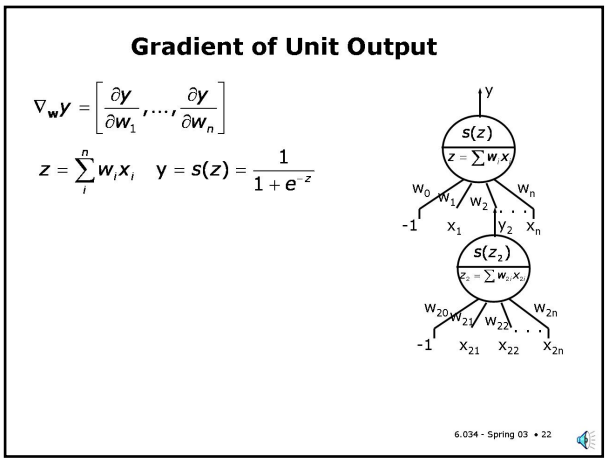
We will define a new quantity called delta, which is defined to be the derivative of the error with respect to a change in the activation z. We can think of this value as the "sensitivity" of the network output to a change in the activation of a unit.

The important result we get is that the change in the i^{th} weight is proportional to delta times the i^{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 4.1.21

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.



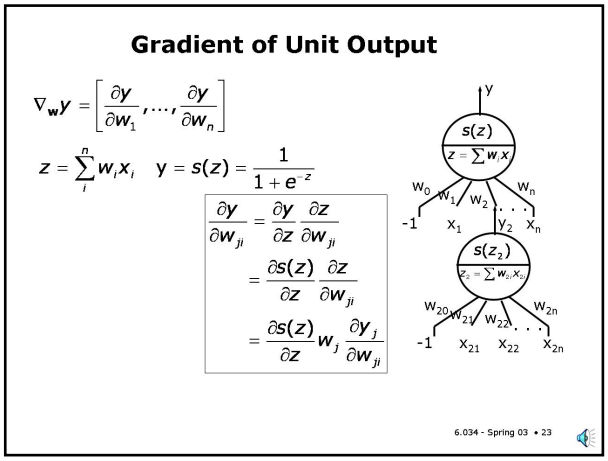


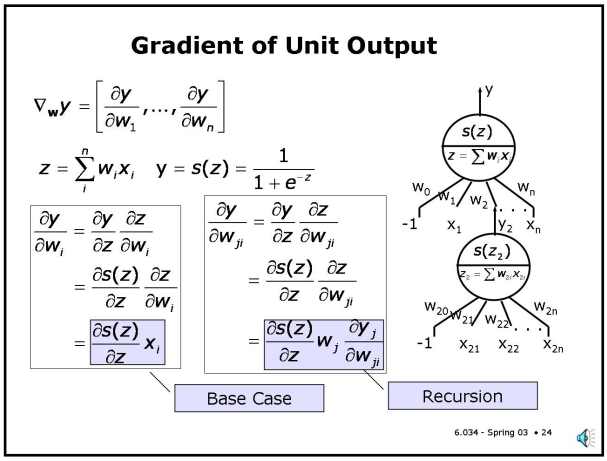
Slide 4.1.22

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 4.1.23

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.



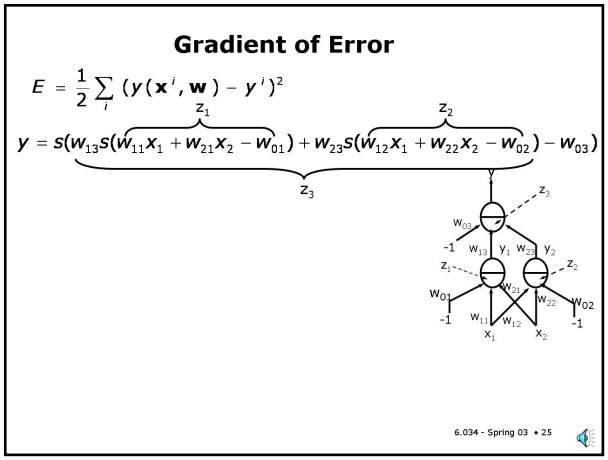


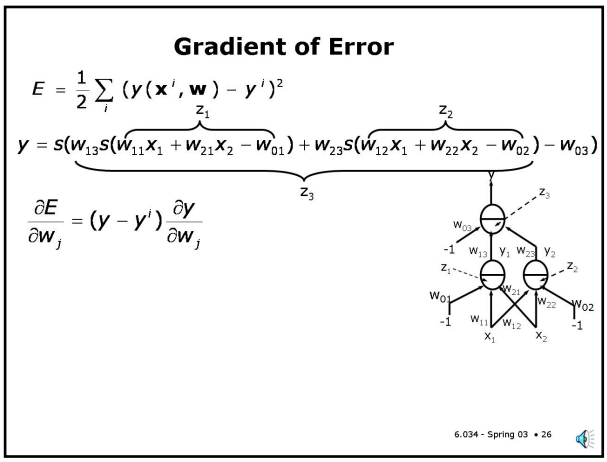
Slide 4.1.24

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_i, since clearly the derivative of any unit with respect to w_i "below" the line with that weight will be zero.

Slide 4.1.25

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.





Slide 4.1.26

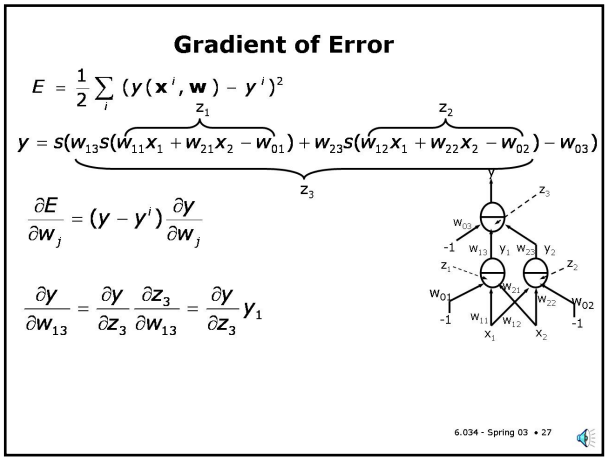
Recall that in the derivative of the error (**for a single instance**) with respect to any of the weights, we get a term that measures the error at the output (y-y^\i) times the change in the output which is produced by the change in the weight (dy/dw).

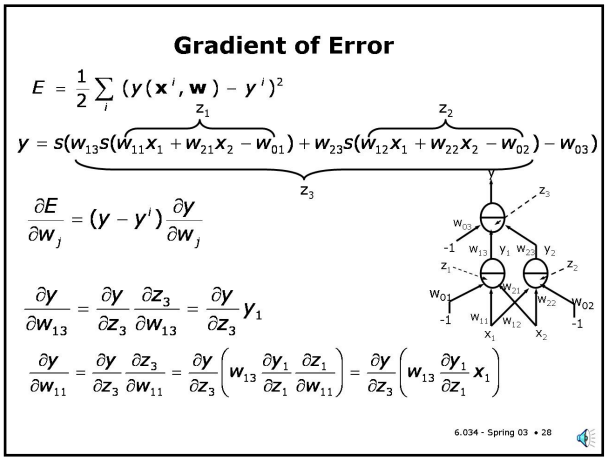
Slide 4.1.27

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.





Slide 4.1.28

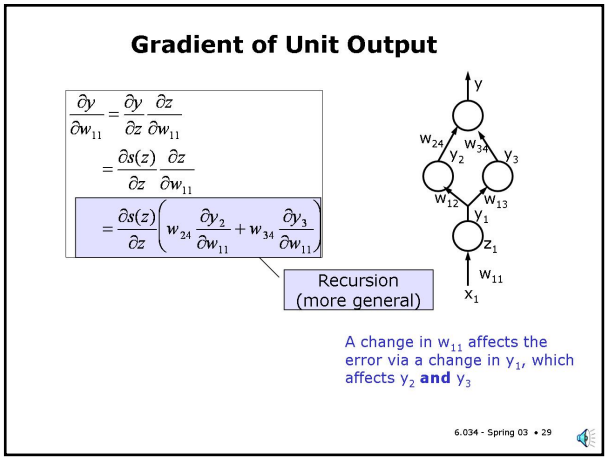
What happens when we pick a weight that's deeper in the net, say w_{11}? Since that weight affects y_1, we expect that the change in the final output will be affected by the value of w_{13} and the slope of the sigmoid at unit 3 (as when we were changing w_{13}). In addition, the change in y_1 will depend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

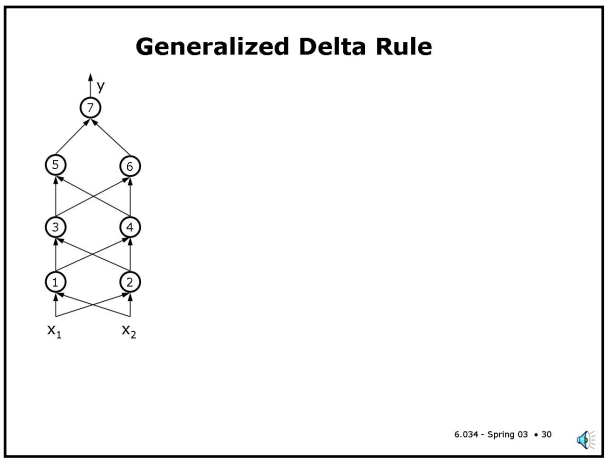
Note that in computing the gradients deeper in the net we will use some of the gradient terms closer to the output. For example, the gradient for weights on the inputs to unit 1 change the output by changing one input to unit 3 and so the final gradient depends on the behavior of unit 3. It is the realization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

Slide 4.1.29

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.





Slide 4.1.30

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 $w^{i->k}$ 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.

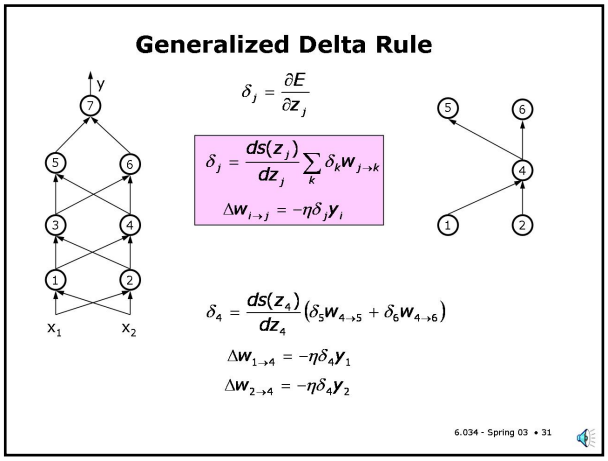
Slide 4.1.31

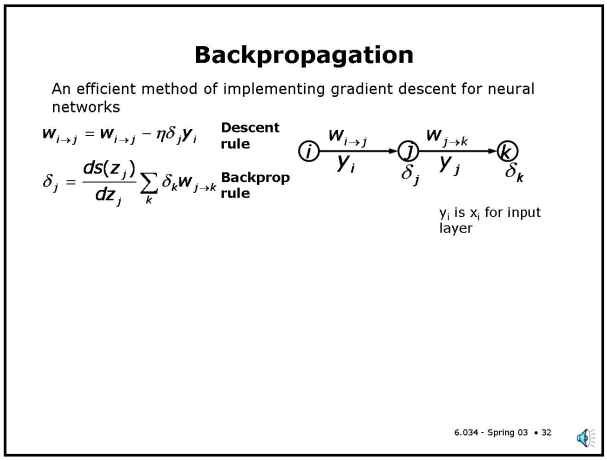
In this type of network we can define a generalization of the delta rule that we saw for a single unit. We still want to define the sensitivity of the training error (for an input point) to a change in the total activation of a unit. This is a quantity associated with the unit, independent of any weight. 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 gradient efficient.





Slide 4.1.32

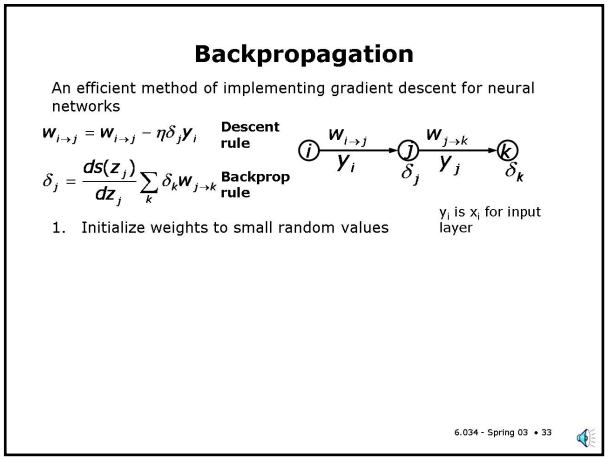
Thus, the algorithm for computing the gradients we need to update the weights of a net, with maximal re-using of intermediate results is known as backpropagation.

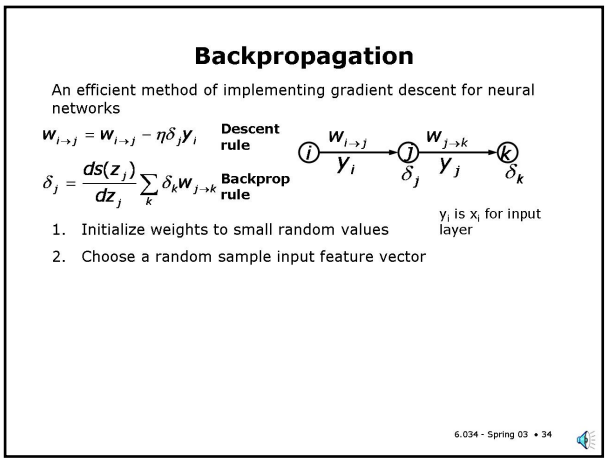
The two simple formulas we need are the ones we have just seen. One tells us how to change a weight. This is a simple gradient descent formula, except that it says that the gradient of the error is of the form delta_i times y_i where y_i is the signal on the wire with this weight, so it is either one 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 4.1.33

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.



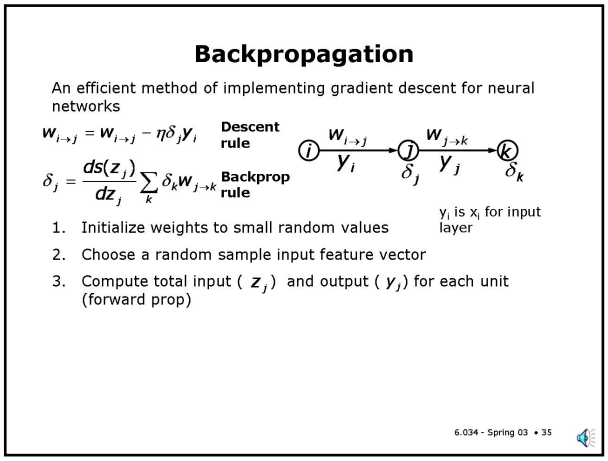


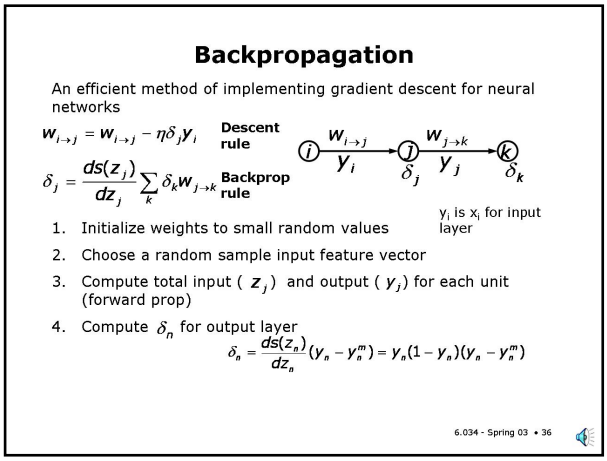
Slide 4.1.34

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 4.1.35

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.



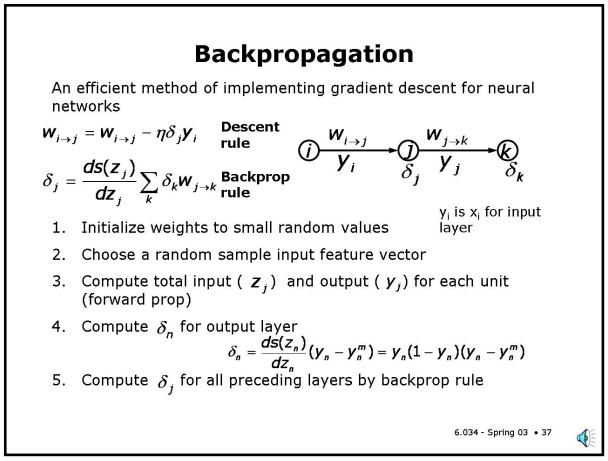


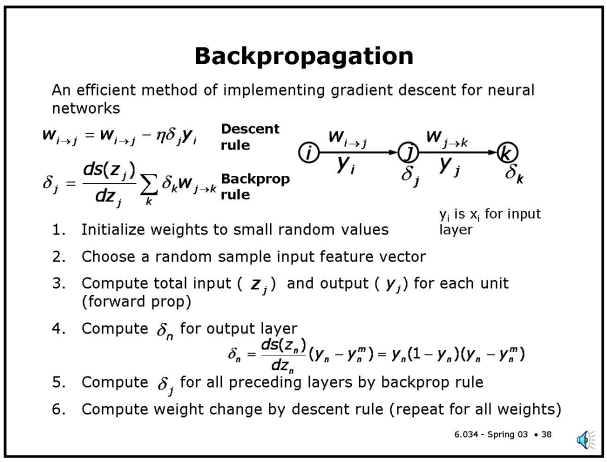
Slide 4.1.36

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 4.1.37

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





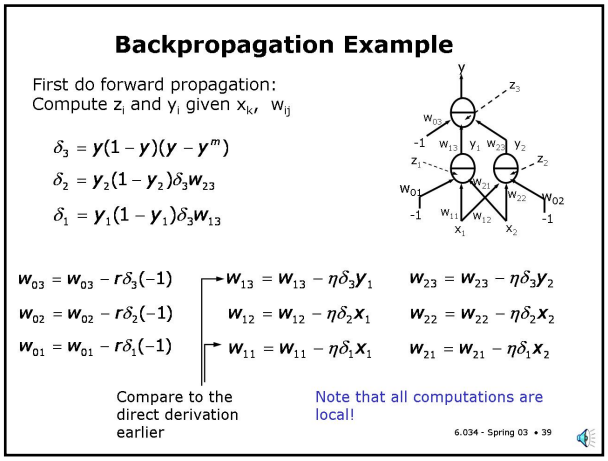
Slide 4.1.38

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

Slide 4.1.39

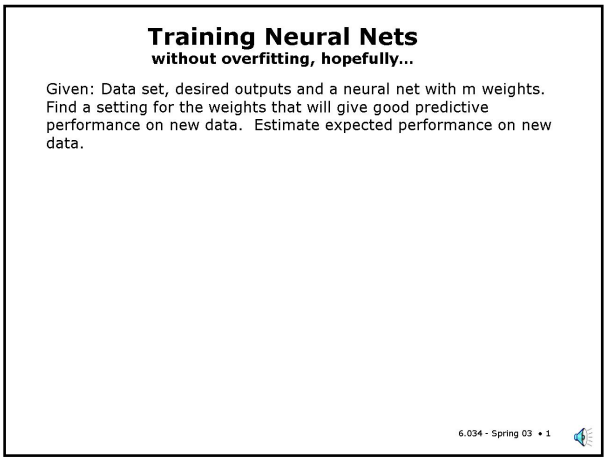
We can see what is involved in doing the simple three-unit example we saw earlier. Here we see the simple expressions for the deltas and the weight updates. Note that each expression involves data local to a particular unit, you don't have to look around summing things over the whole network, the delta's 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.



6.034 Notes: Section 4.2

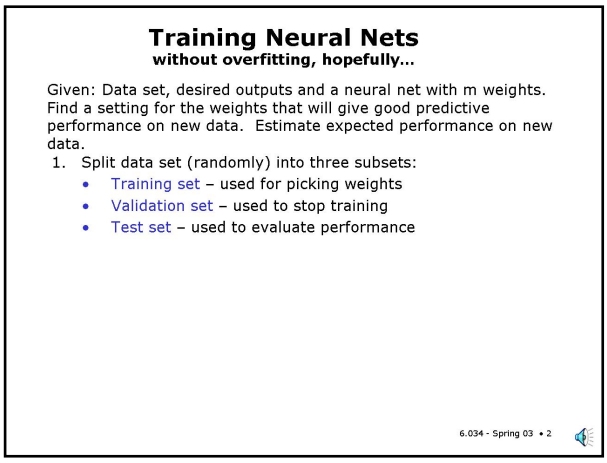
Slide 4.2.1



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.

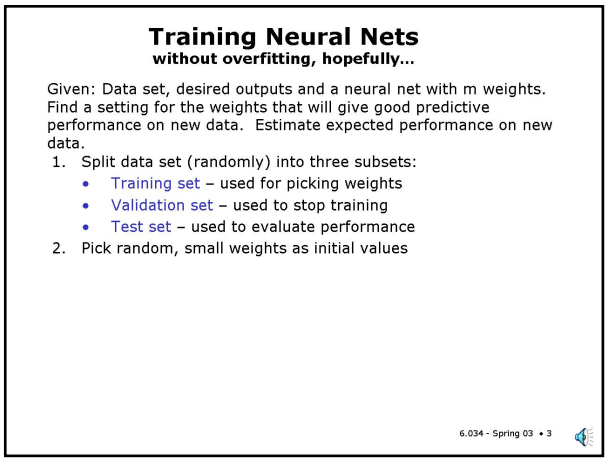
We need to worry about overfitting because of the generality of neural nets and the proliferation of parameters associated even with a relatively simple net. It is easy to construct a net that has more parameters than there are data points. Such nets, if trained so as to minimize the training error without 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.

Slide 4.2.2

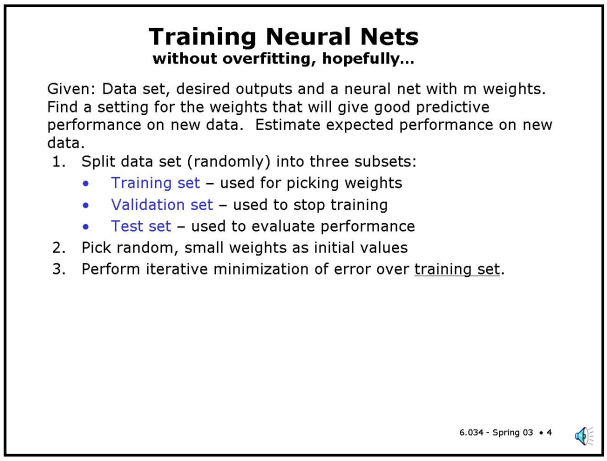


The first step (in the ideal case) is to separate the data into three sets. A training set for choosing the weights (using backpropagation), a validation set for deciding when to stop the training and, if possible, a separate set for evaluating the final results.

Slide 4.2.3



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.



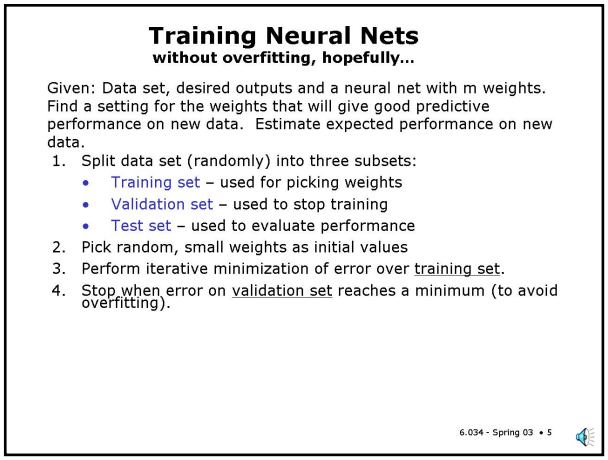
Slide 4.2.4

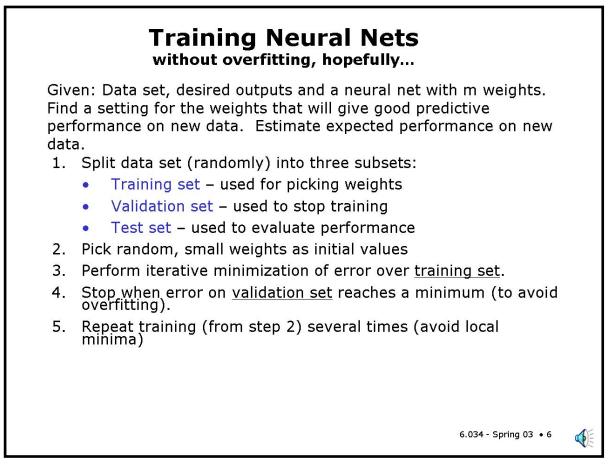
Then we perform the minimization of the training error, for example, using backpropagation. This will generally involve going through the input data and making changes to the weights many times. A common term used in this context is the **epoch**, which indicates how many times the algorithm has gone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

Slide 4.2.5

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.



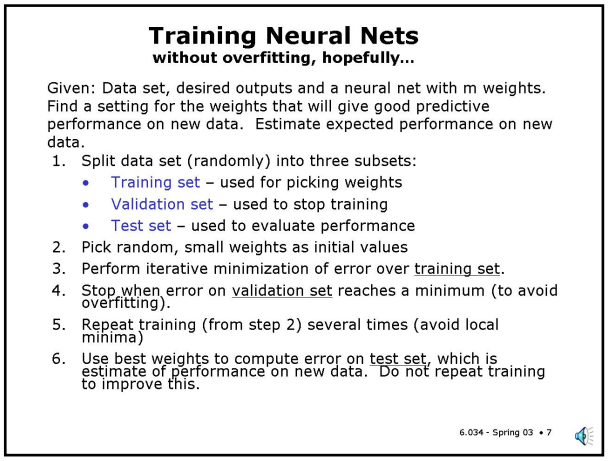


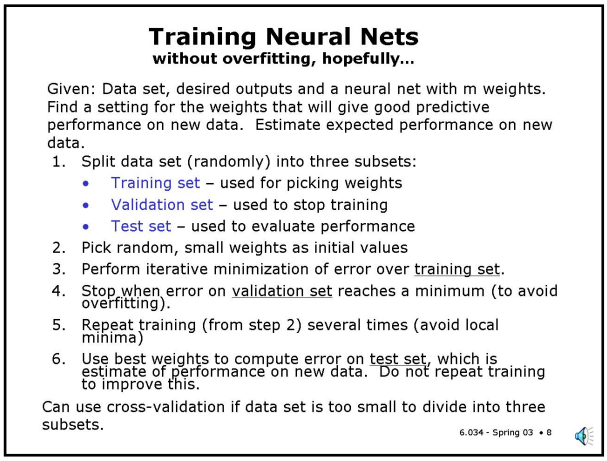
Slide 4.2.6

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 4.2.7

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.





Slide 4.2.8

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 4.2.9

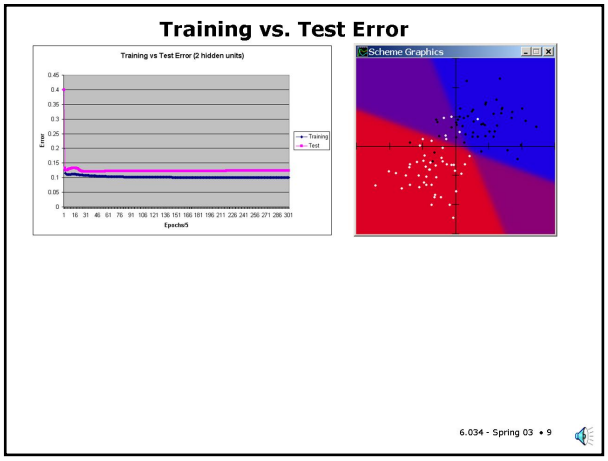
Let's look at the termination/overfitting issue via some examples.

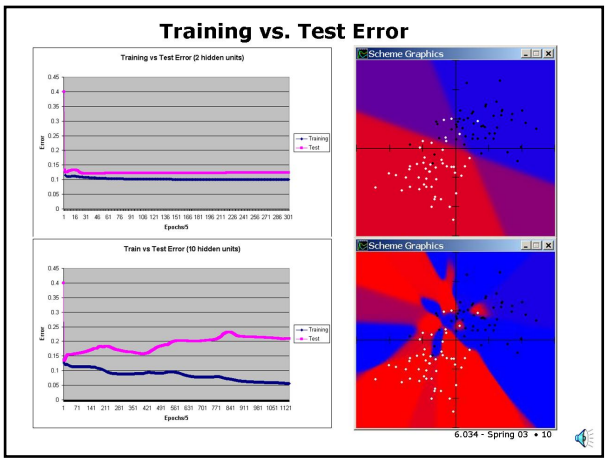
Here we see the behavior of a small neural net (two inputs, two hidden units and one output) when trained on the data shown in the picture on the right. The white and black points represent 50 instances from each of two classes (drawn from Gaussian distributions with different centers). An additional 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 4.2.10

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 drops, 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.

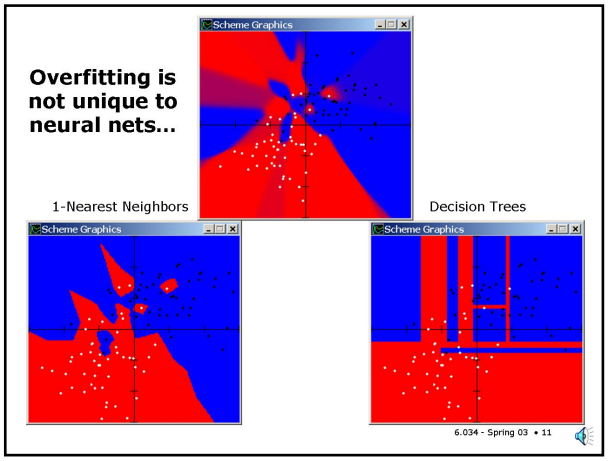
So, in fact, all that extra work in fitting the training set is wasted. Note that the test error with this net is much higher than with the simpler net. If we had used a validation set, we could have stopped training before it went too far astray.

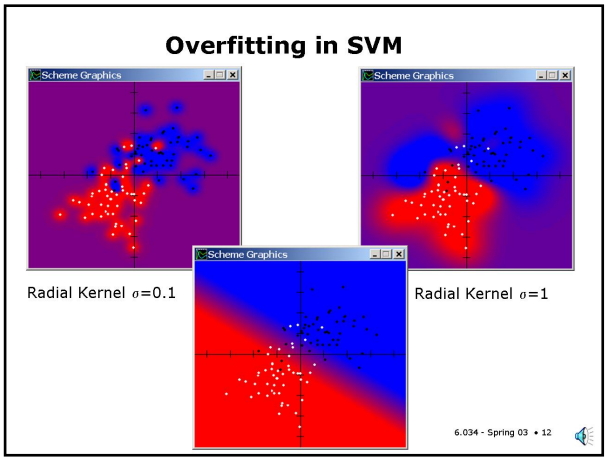
Slide 4.2.11

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.



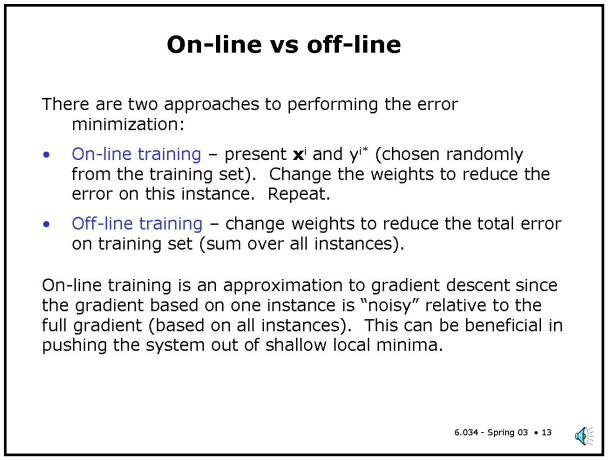


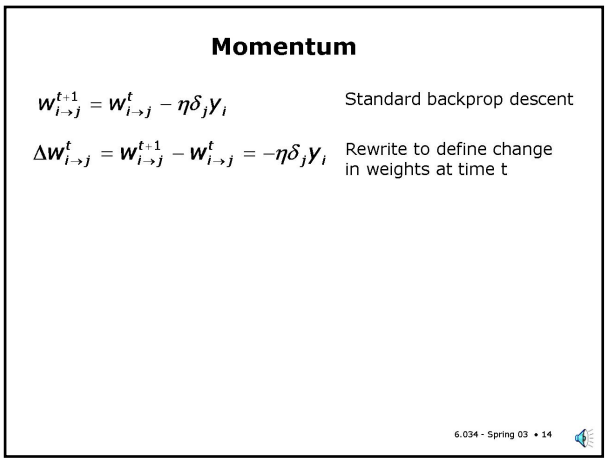
Slide 4.2.12

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 seaprator 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 4.2.13

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.





Slide 4.2.14

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 4.2.15

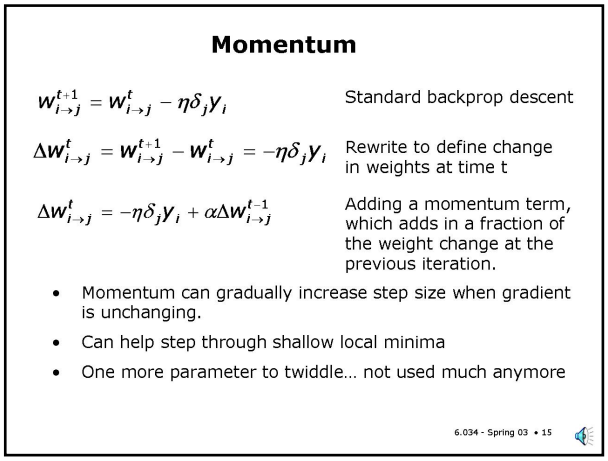
We can keep around the most recent change to the weights (at time t-1) and add some fraction of that weight change to the current delta. The fraction, alpha, is the momentum weight.

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.



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 4.2.16

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.

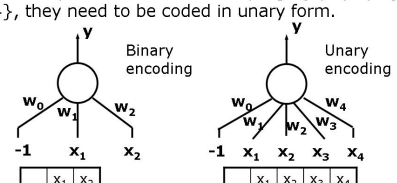
One issue that is prominent in neural nets is the fact that the behavior of these nets are dependent on the scale of the input. In particular, one does not want to saturate the units, since at that point it becomes impossible to train them. Note that all "signals" in a sigmoid-unit neural net are in 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 4.2.17

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.



	x_1	x_2
A	0	0
B	0	1
C	1	0
D	1	1

	x_1	x_2	x_3	x_4
A	0	0	0	0
B	0	1	0	0
C	0	0	1	0
D	0	0	0	1

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 4.2.18

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 4.2.19

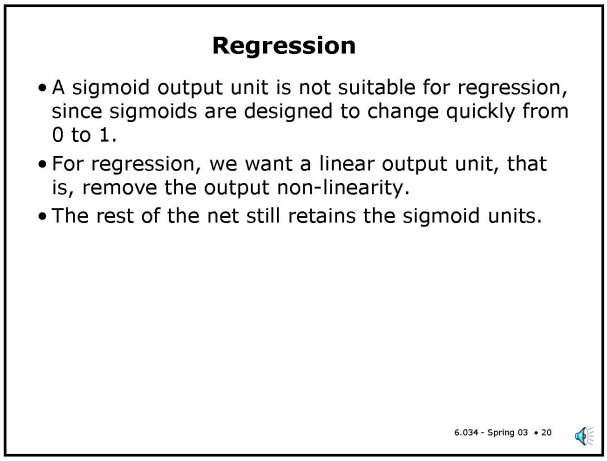
Another detail to consider in training is what to use as the desired (target) value for the network outputs. We have been talking as if we would use 0 or 1 as the targets. The problem with this is that those are the asymptotic values of the sigmoid only achieved for infinite values of the weights. So, 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

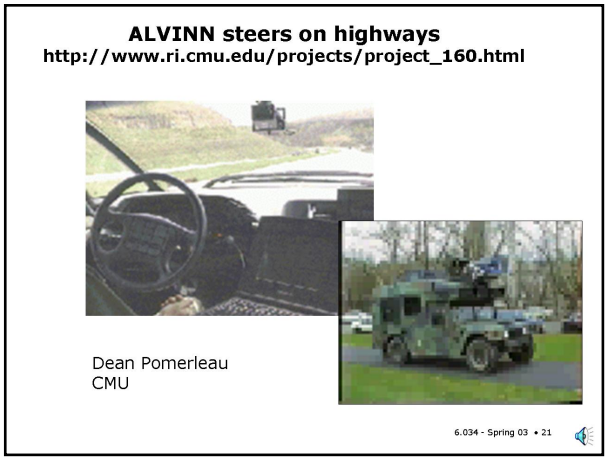


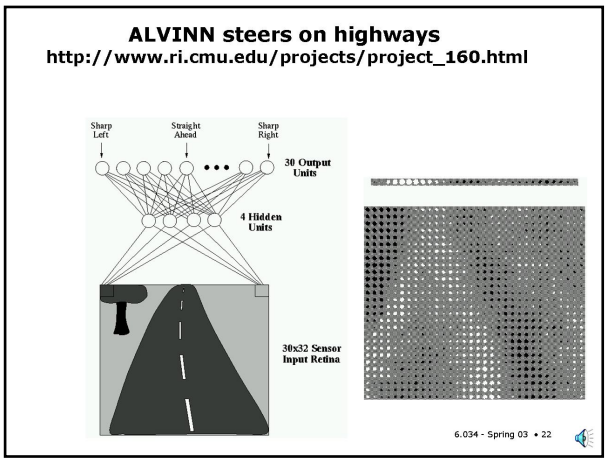
Slide 4.2.20

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 4.2.21

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 succesfully demonstrated on a variety of real roads in actual traffic. A successor to ALVINN, which unfortunately was not based on machine learning, has been used to steer a vehicle in a cross-country trip.





Slide 4.2.22

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.

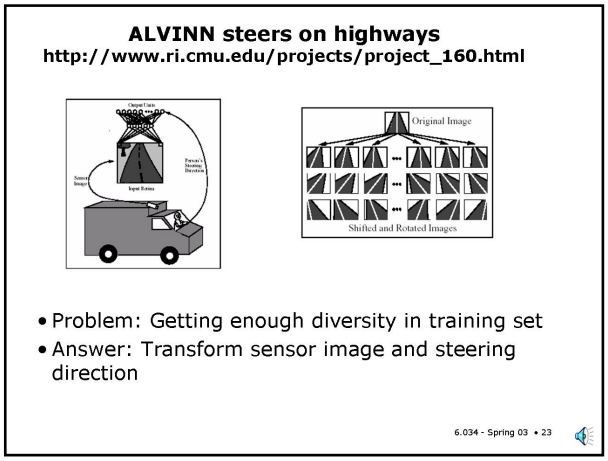
Slide 4.2.23

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.

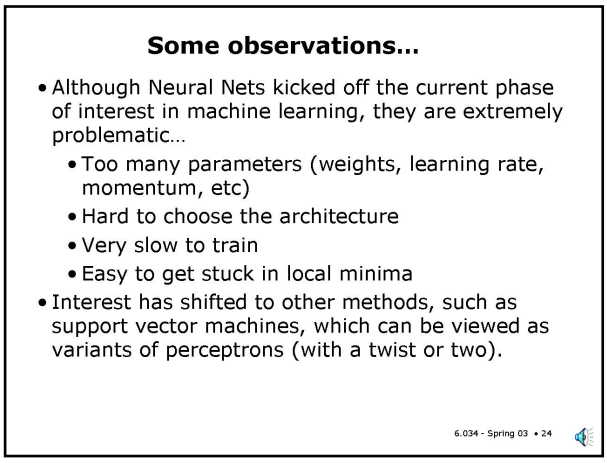
The researchers considered having the vehicle drive in a wobbly path during training, but that posed the danger of having the system learn to drive that way. They came up with a clever solution. Simulate what the sensory data would have looked like had ALVINN been off-center or headed in a slightly wrong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



Slide 4.2.24

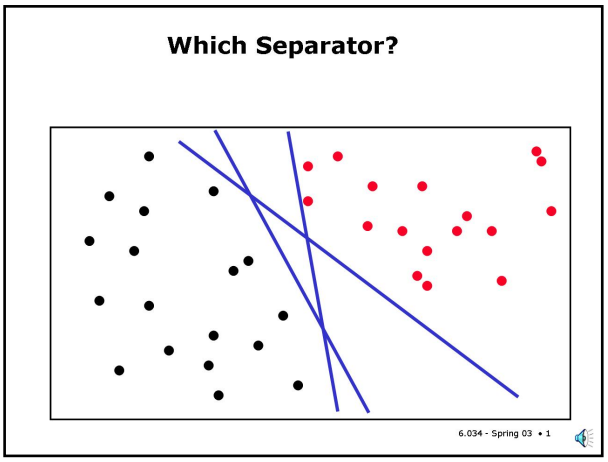
Neural nets are largely responsible for the current interest in statistical methods for machine learning. For a while, they were wildly popular in the research community and saw many applications. Over time, the enthusiasm has waned somewhat.

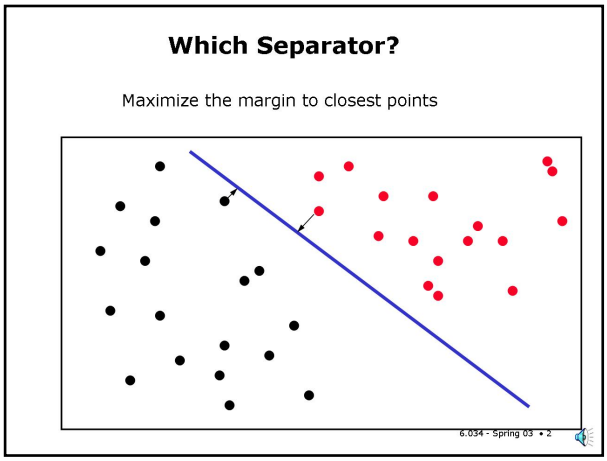
The big problem with neural nets is that they are very complex and have many many parameters that have to be chosen. Furthermore, training them is a bit of a nightmare. So, recent interest has shifted towards methods that are simpler to use and can be characterized better. For example, support vector 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.

6.034 Notes: Section 4.3

Slide 4.3.1

There is no easy way to characterize which particular separator the perceptron algorithm will end up with. In general, there can be many separators for a data set. Even in the tightly constrained bankruptcy data set, we saw two runs of the algorithm with different starting points ended up with slightly different hypotheses. Is there any reason to prefer one separator over the others?





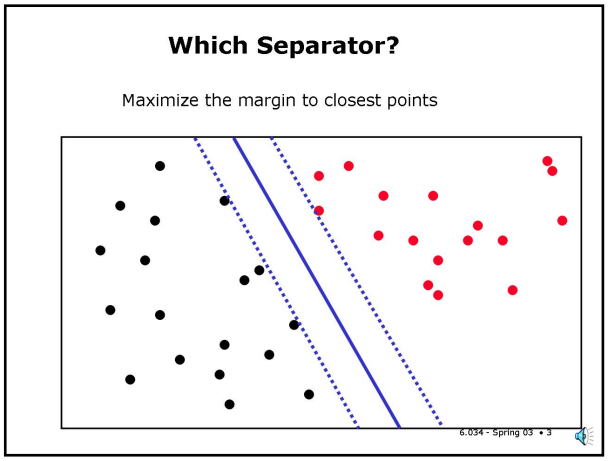
Slide 4.3.2

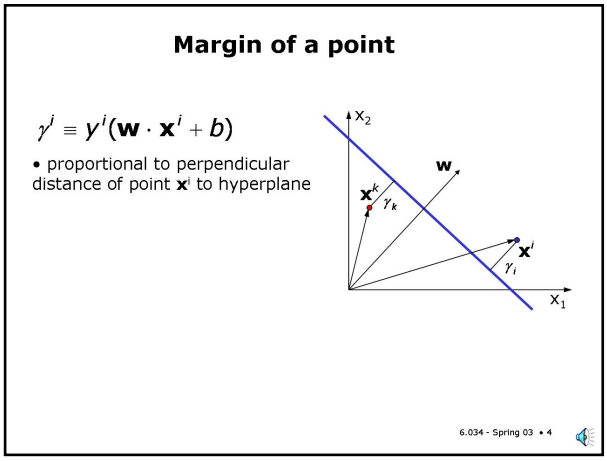
Yes. One natural choice is to pick the separator that has the maximal margin to its closest points on either side. This is the separator that seems most conservative. Any other separator will be "closer" to one class than to the other. The one shown in this figure, for example, seems like it's closer to the black points on the lower left than to the red ones.

Slide 4.3.3
This one seems safer, no?

Another way to motivate the choice of the maximal margin separator is to see that it reduces the "variance" of the hypothesis class. Recall that a hypothesis has large variance if small changes in the data result in a very different hypothesis. With a maximal margin separator, we can wiggle the data quite a bit without affecting the separator. Placing the separator very close to positive or negative points is a kind of overfitting; it makes your hypothesis very dependent on details of the input data.

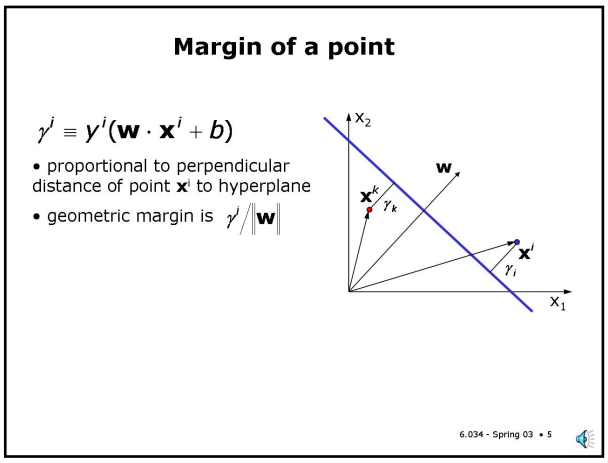
Let's see if we can figure out how to find the separator with maximal margin as suggested by this picture.

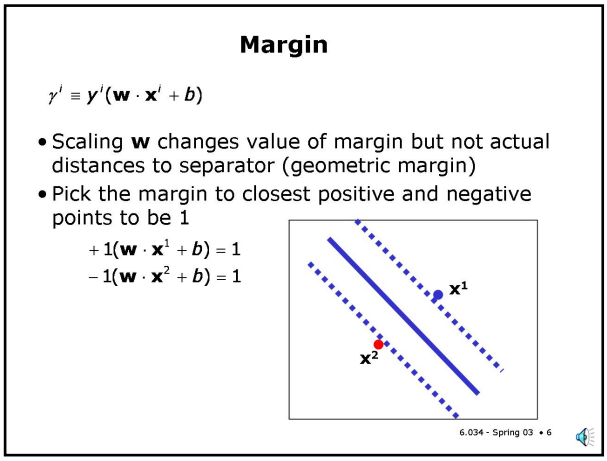




Slide 4.3.4
First we have to define what we are trying to optimize. Clearly we want to use our old definition of margin, but we'll have to deal with a couple of issues first. Note that we're using the **w**, *b* notation instead of **w** bar, because we will end up giving *b* special treatment in the future.

Slide 4.3.5
Remember that any scaling of **w** and *b* defines the same line; but it will result in different values of gamma. To get the actual geometric distance from the point to the separator (called the *geometric margin*), we need to divide gamma through by the magnitude of **w**.





Slide 4.3.6
The next issue is that the we have defined the margin for a point relative to a separator but we don't want to just maximize the margin of some particular single point. We want to focus on one point on each side of the separator, each of which is closest to the separator. And we want to place the separator so that the it is as far from these two points as possible. Then we will have the maximal margin between the two classes.

Since we have a degree of freedom in the magnitude of **w** we're going to just define the margin for each of these points to be 1. (You can think of this 1 as having arbitrary units given by the magnitude of **w**.)

You might be worried that we can't possibly know which will be the two closest points until we know what the separator is. It's a reasonable worry, and we'll sort it out in a couple of slides.

Slide 4.3.7
Having chosen these margins, we can add the two equations to get that the projection of the weight vector on the difference between the two chosen data points has magnitude 2. This is obvious from the setup, but it's nice to see it follows.

Then, we divide through by the magnitude of the weight vector and we have a simple expression for the margin, simply 2 over the magnitude of **w**.

Margin

- Pick the margin to closest positive and negative points to be 1

$$+1(\mathbf{w}\cdot\mathbf{x}^1+b)=1$$
$$-1(\mathbf{w}\cdot\mathbf{x}^2+b)=1$$

- Combining these

$$\mathbf{w}\cdot(\mathbf{x}^1-\mathbf{x}^2)=2$$

- Dividing by length of **w** gives perpendicular distance between dashed lines (2 **x** geometric margin)

$$\frac{\mathbf{w}}{\|\mathbf{w}\|}\cdot(\mathbf{x}^1-\mathbf{x}^2)=\frac{2}{\|\mathbf{w}\|}$$

6.034 - Spring 03 • 7

Picking w to Maximize Margin

- Pick **w** to maximize geometric margin

$$\frac{2}{\|\mathbf{w}\|}$$

- or, equivalently, minimize

$$\|\mathbf{w}\|=\sqrt{\mathbf{w}\cdot\mathbf{w}}$$

- or, equivalently, minimize

$$\frac{1}{2}\|\mathbf{w}\|^2=\frac{1}{2}\mathbf{w}\cdot\mathbf{w}=\frac{1}{2}\sum_j w_j^2$$

6.034 - Spring 03 • 8

Slide 4.3.8
So, we want to pick **w** to maximize the geometric margin, that is, to maximize 2 over the magnitude of **w**. To maximize this expression, we want to minimize the magnitude of **w**. If we minimize 1/2 the magnitude squared that is completely equivalent in effect but simpler analytically.

Of course, this is not enough, since we could simply pick **w** = 0 which would be completely useless.

Slide 4.3.9
We'd like to find the **w** that specifies the maximum margin separator. To be a separator, **w** needs to classify the points correctly. So, we'll maximize the margin, subject to a set of constraints that require the points to be classified correctly. We will require each point in the training set to have a margin greater than or equal to 1. Requiring the margins to be positive will ensure that they are classified correctly. Requiring them to be greater than or equal to 1 will ensure that the margin of the closest points wil be greater than or equal to 1. The fact that we are minimizing the magnitude of **w** will force the margins to be as small as possible, so that in fact the margins of the closest points will equal 1.

Picking w to Maximize Margin

- Pick **w** to maximize geometric margin

$$\frac{2}{\|\mathbf{w}\|}$$

- or, equivalently, minimize

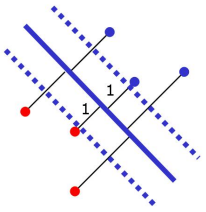
$$\frac{1}{2}\|\mathbf{w}\|^2=\frac{1}{2}\mathbf{w}\cdot\mathbf{w}=\frac{1}{2}\sum_j w_j^2$$

- while classifying points correctly

$$y^i(\mathbf{w}\cdot\mathbf{x}^i+b)\geq 1$$

- or, equivalently,

$$y^i(\mathbf{w}\cdot\mathbf{x}^i+b)-1\geq 0$$



6.034 - Spring 03 • 9

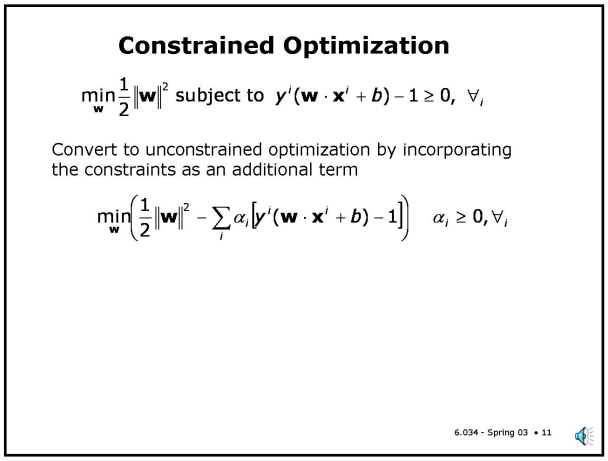
Constrained Optimization

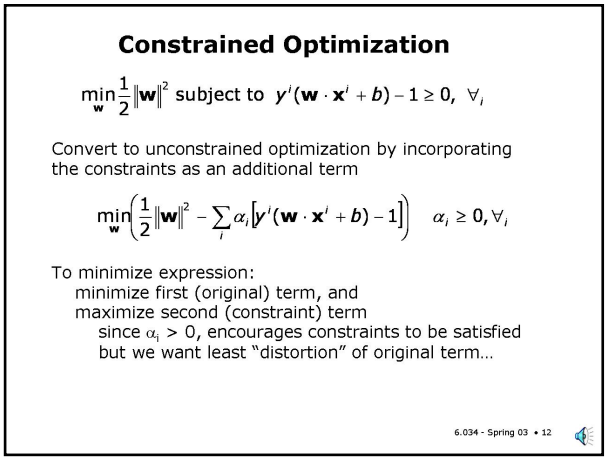
$$\min_{\mathbf{w}}\frac{1}{2}\|\mathbf{w}\|^2 \text{ subject to } y^i(\mathbf{w}\cdot\mathbf{x}^i+b)-1\geq 0,\ \forall_i$$

6.034 - Spring 03 • 10

Slide 4.3.10
So, to summarize, we have defined a constrained optimization problem as shown here. It involves minimizing a quadratic function subject to a set of linear constraints. These kinds of optimization problems are very well studied. When the function to be minimized is linear, it is a particularly easy case that can be solved by a "linear programming" algorithm. In our case, it's a bit more complicated.

Slide 4.3.11
The standard approach to solving this type of problem is to convert it to an unconstrained optimization problem by incorporating the constraints as additional terms in the function to be minimized. Each of the constraints is multiplied by a weighting term $alpha_i$. Think of these terms as penalty terms that will penalize values of **w** that do not satisfy the constraints.

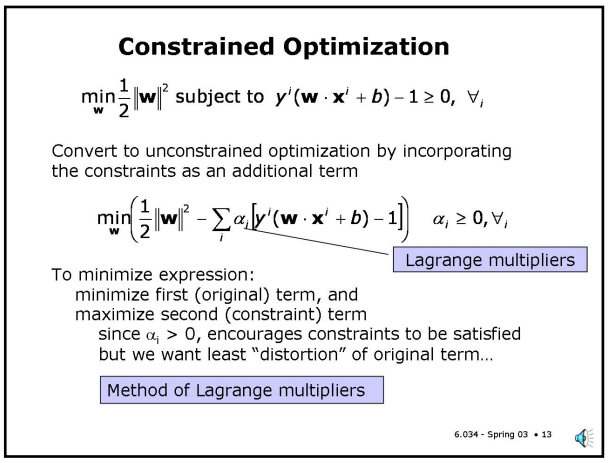




Slide 4.3.12
To minimize the combined expression we want to minimize the first term (the magnitude of the weight vector squared) but we want to maximize the constraint term since it is negated. Since $alpha_i > 0$, making the constraint terms bigger encourages them to be satisfied (we want the margins to be bigger than 1).

However, the bigger the constraint term, the farther we move from the original minimal value of **w**. In general we want to minimize this "distortion" of the original problem. We want to introduce just enough distortion to satisfy the constraints. We'll look at this in more detail momentarily.

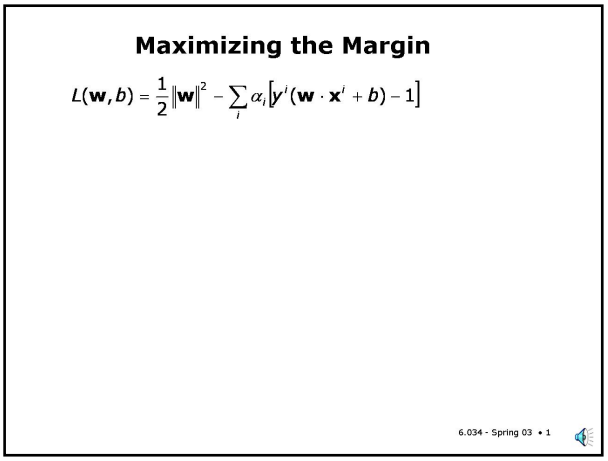
Slide 4.3.13
This method we have begun to outline here is called the **method of Lagrange multipliers** and the $alpha_i$ are the individual Lagrange multipliers.

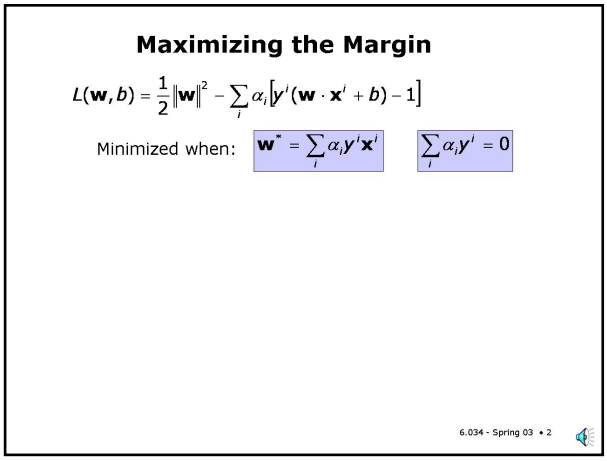


6.034 Notes: Section 4.4

Slide 4.4.1
The details of solving a Lagrange multiplier problem are a little bit complicated. But we are going to go through the derivation at a somewhat abstract level here, because it gives us some insights and intuitions about the resulting solution.

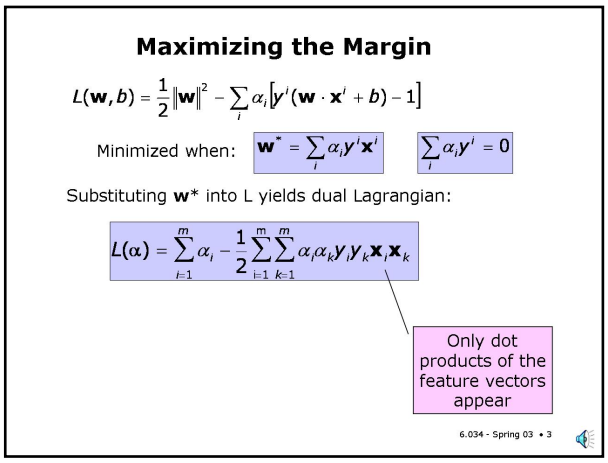
We have an expression, *L(***w**,*b)*, that also involves parameters alpha. If we knew what the values of alpha should be, we could just fix them, minimize *L* with respect to **w** and *b*, and be done. The big problem is that we don't know what the alphas are supposed to be.

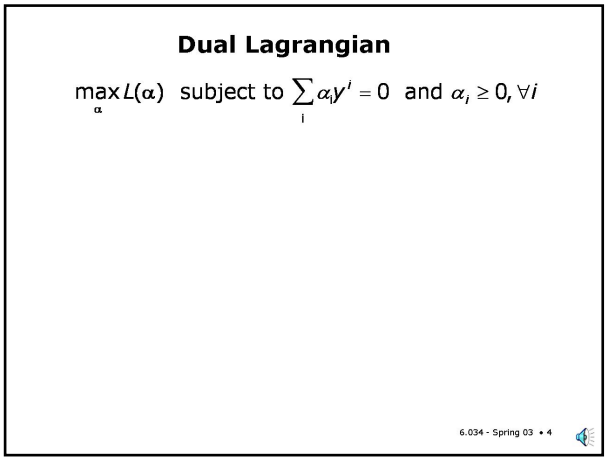




Slide 4.4.2
So, we're going to start by imagining that we know what we want the alphas to be. We'll hold them constant for now, and figure out what values of **w** and *b* would optimize *L* for those fixed alphas. We can do this by taking the partial derivatives of *L* with respect to **w** and *b* and setting them to zero, getting two constraints. We find that the best value of **w**, **w*** is a weighted sum of the input points (in the same form as the dual perceptron); and we get an extra constraint that the sum of the alphas for the positive points has to equal the sum of the alphas for the negative points.

Slide 4.4.3
We can substitute this expression for the optimal **w**'s back into our original expression for L, getting L as a function of alpha. Now we have an expression involving only alphas, which we don't know, and x's and y's, which we do know. This function is known as the *dual Lagrangian*. One of the most important things about it, from our perspective, is that the feature vectors only appear in dot products with other feature vectors. We'll come back to this point later on.





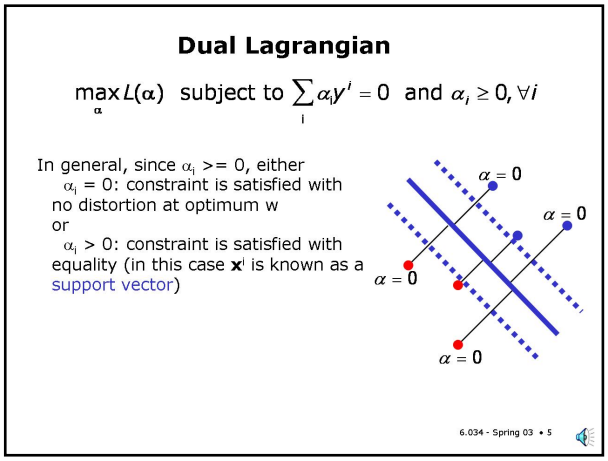
Slide 4.4.4
Now, it's time to pick the best values for the alphas. We do so (for reasons that you'll have to learn in a math class) by choosing the alpha values that maximize this expression. We will retain the constraints that the sum of the alpha values for positive points is equal to the sum of the alpha values for negative points, and that the alphas must be positive.

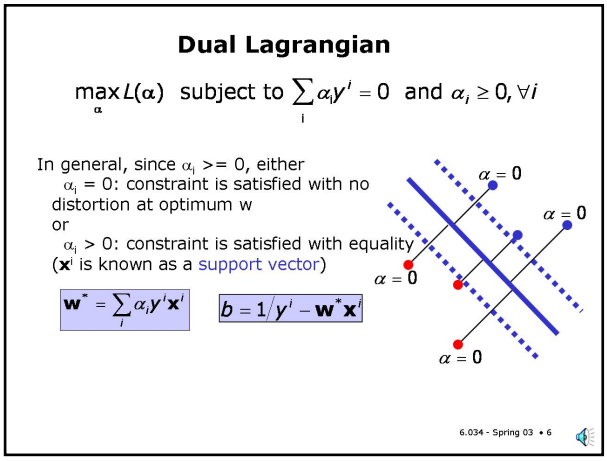
Note that we will be solving for *m* alphas. We started with *n+1* (the number of features, plus one) variables in the original Lagrangian and now we have *m* (the number of data points) variables in the dual Lagrangian. For the low-dimensional examples we have been dealing with this seems like a horrible tradeoff. We will see later that this can be a very good tradeoff in some circumstances.

We have two constraints, but they are much simpler. One constraint is simply that the alphas be non-negative---this is required because our original constraints were >= inequalities. The constraint on the alphas comes from the setting to zero the derivative of the Lagrangian with respect to the offset *b*.

This problem is not trivial to solve in general; we'll talk more about this later. For now, let us assume that we can solve it and get the optimal values of alphas.

Slide 4.4.5
In the solution, most of the alphas will be zero, corresponding to data points that do not provide binding constraints on the choice of the weights. A few of the data points will have their alphas be nonzero; they will all satisfy their constraints with equality (that is, their margin is equal to 1). These are called **support vectors** and they are the ones used to define the maximum margin separator. You could remove all the other data points and still get the same separator. Because the sparsity of support vectors is so important, this learning method is called a **support vector machine**, or SVM.

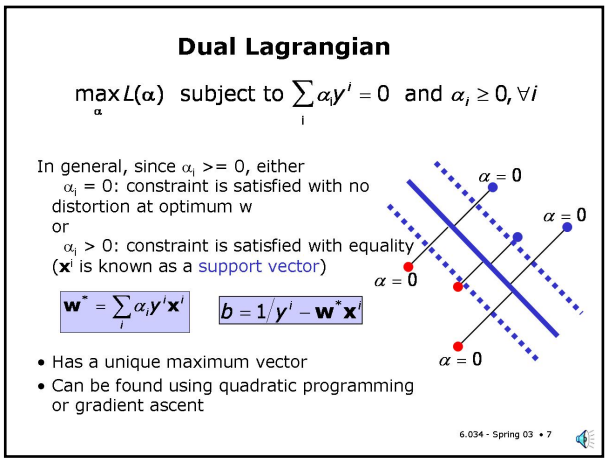


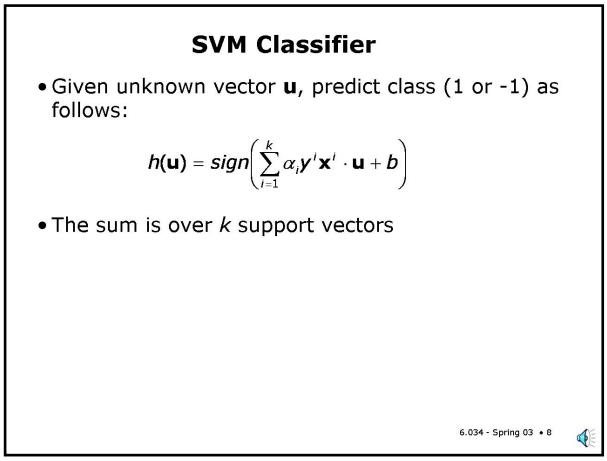


Slide 4.4.6
Given the optimal alphas, we can compute the weights. but this time, the coefficients in the sum are the Lagrange multipliers, the alphas, which are mostly zero. This means that the equation of the maximum margin separator depends only on the handful of data points that are closest to it. It makes sense that all the rest of the points would be irrelevant.

We can use the fact that at the support vectors the constraints hold with equality to solve for the value of the offset *b*. Each such constraint can be used to solve for this scalar.

Slide 4.4.7
We have not discussed actual algorithms for finding the maxima of the dual Lagrangian. It turns out that the optimization problem we defined is a relatively simple form of the general class of **quadratic programming** problems, which are known to (a) have a unique maximum and (b) can be found using existing algorithms. A number of variations on these algorithms exist but they are beyond our scope.

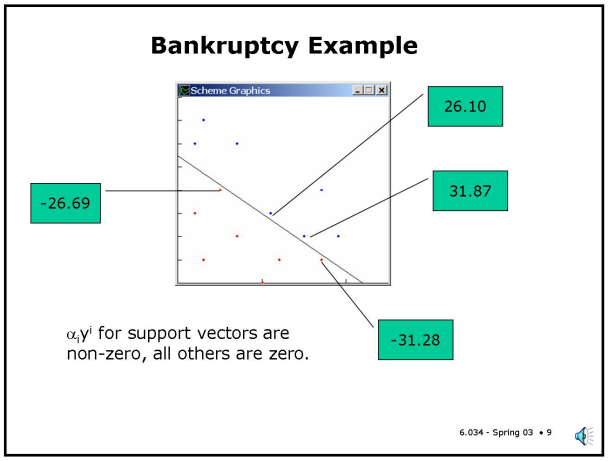


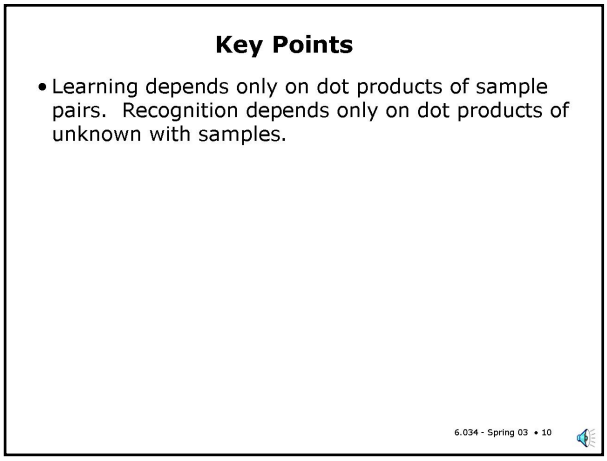


Slide 4.4.8

With the values of the optimal alpha$_i$'s and *b* in hand, and the knowledge of how **w** is defined, we now have a classifier that we can use on unknown points. Crucially, notice that once again, the only thing we care about are the dot products of the unknown vector with the data points.

Slide 4.4.9
Here's the result of running a quadratic programming algorithm to find the maximal margin separator for the bankruptcy example. Note that only four points have non-zero alpha's. They are the closest points to the line and are the ones that actually define the line.



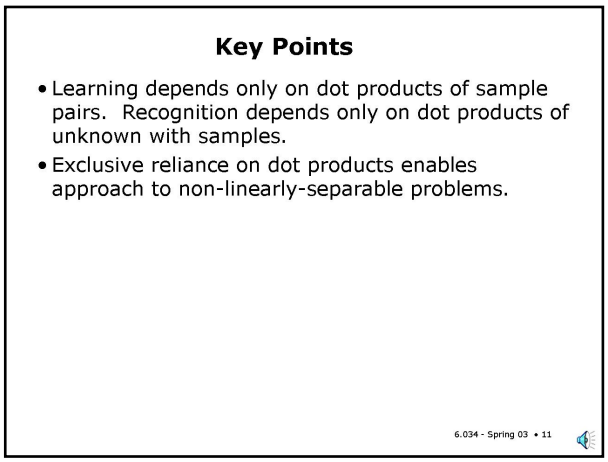


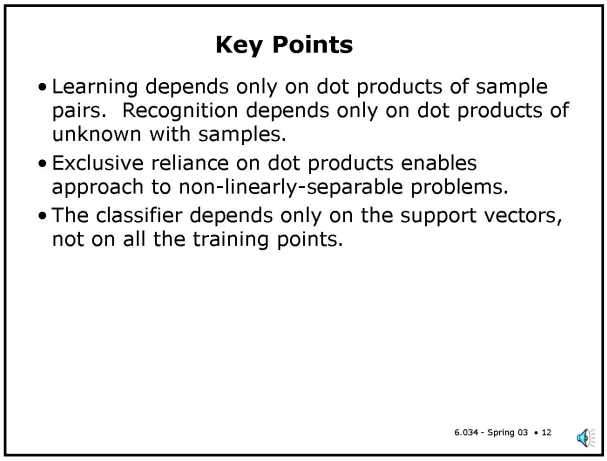
Slide 4.4.10

Let's highlight once again a few of the key points about SVM training and classification. First and foremost, and at the risk of repeating myself, recall that the training and classification of SVMs depends only on the value of the dot products of data vectors. That is, if we have a way of getting the dot products, the computation does not otherwise depend explicitly on the dimensionality of the feature space.

Slide 4.4.11

The fact that we only need dot products (as we will see next) means that we will be able to substitute more general functions for the traditional dot product operation to get more powerful classifiers without really changing anything in the actual training and classification procedures.



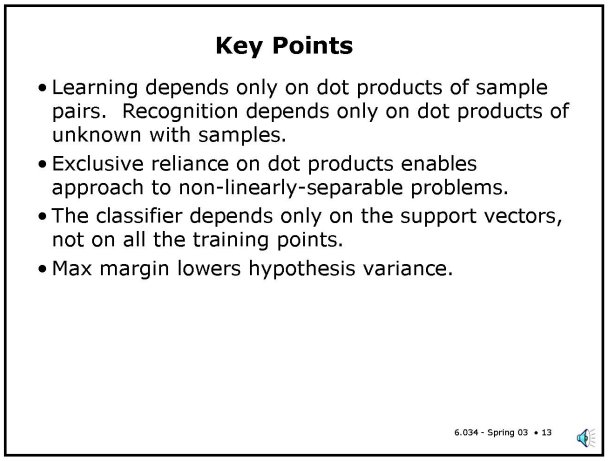


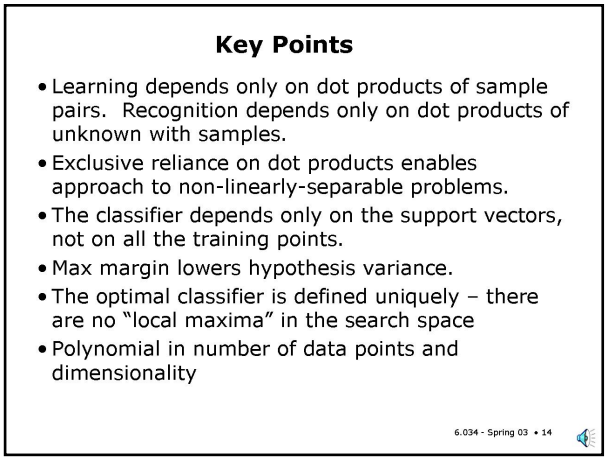
Slide 4.4.12

Another point to remember is that the resulting classifier does not (in general) depend on all the training points but only on the ones "near the margin", that is, those that help define the boundary between the two classes.

Slide 4.4.13

The maximum margin constraint helps reduce the variance of the SVM hypotheses. Insisting on a minimum magnitude weight vector drastically cuts down on the size of the hypothesis class and helps avoid overfitting.





Slide 4.4.14

Finally, we should keep firmly in mind that the SVM training process guarantees a unique global maximum. And it runs in time polynomial in the number of data points and the dimensionality of the data.

6.034 Notes: Section 4.5

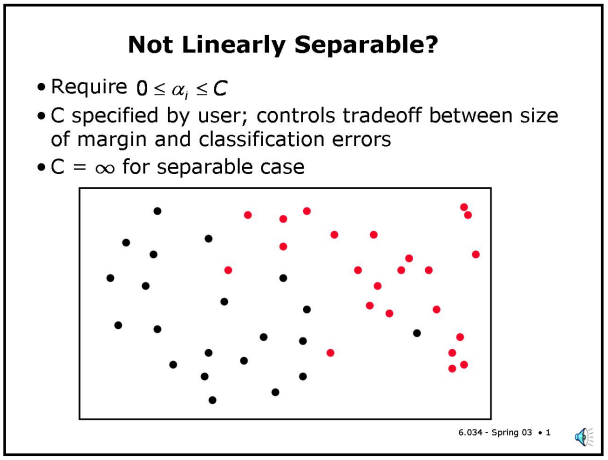
Slide 4.5.1

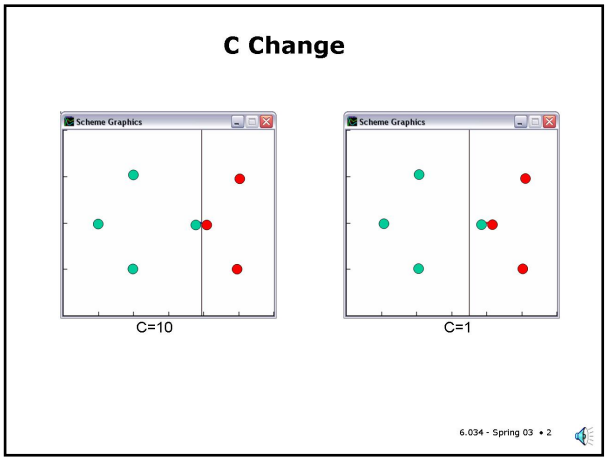
Thus far, we have only been talking about the linearly separable case. What happens for the case in which we have a "nearly separable" problem? That is, some "noise points" that are bound to be misclassified by a linear separator.

It is useful to think about the behavior of the dual perceptron on this type of problem. In that algorithm, the value of the alpha_i for a point is incremented proportionally to its distance to the separator. In fact, if the point is classified correctly, no change is made to the multiplier. We can see that if point i stubbornly resists being classified, then the value of alpha_i will continue to grow without bounds.

The alpha's in the dual perceptron are analogous to the values of the Lagrange multipliers in the SVM. In both cases, the separator is defined as a linear combination of the input points, with the alphas being the weights.

So, one strategy for dealing with these noise points in an SVM is to limit the maximal value of any of the alpha_i's (the Lagrange multipliers) to some C. And, furthermore, to ignore the points with this maximal value when computing the margin. Clearly, if we ignore enough points, we can always get back to a linearly separable problem. By choosing a large value of C, we will work very hard at correctly classifying all the points, a low value of C will allow us to give up more easily on many of the points so as to achieve a better margin.





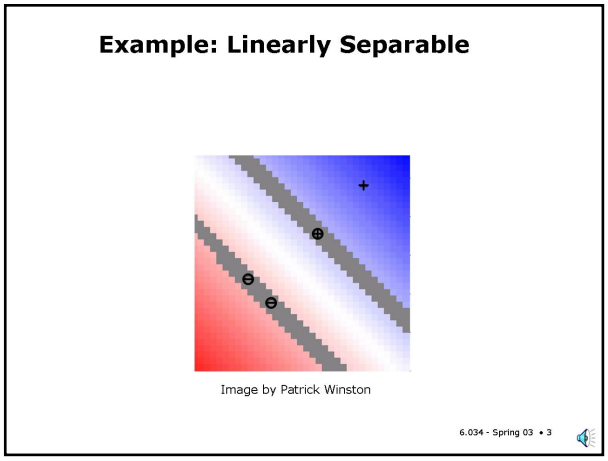
Slide 4.5.2
This simple example shows how changing C causes the geometric margin to change. High values of C penalize misclassifications more. Low values may permit misclassifications to achieve better margin.

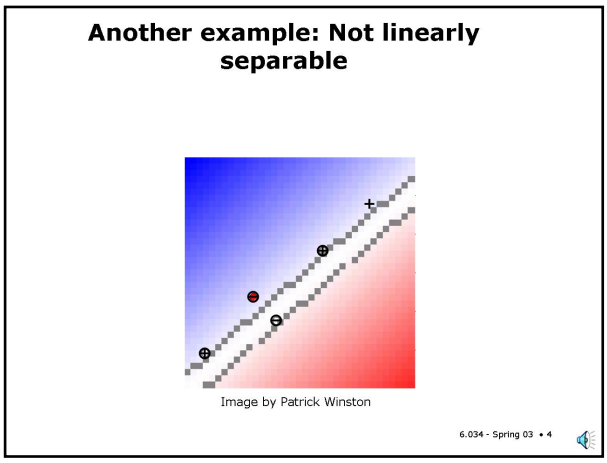
Slide 4.5.3

Here is an example of a separator on a simple data set with four points, which are linearly separable. The colors show the result returned by the classification function on each point in the space. Gray means near 1 or -1. The more intense the blue, the more positive the result; the more intense the red, the more negative. Points lying between the two gray lines return values between -1 and +1.

Note that only three of the four samples are actually used to define **w**, the ones circled. The other plus sample might as well not be there; its coefficient alpha is zero.

The samples that are actually used are the **support vectors**.





Slide 4.5.4

The next example is the same as the previous example, but with the addition of another plus sample in the lower left corner. There are several points of interest.

First, the optimization has failed to find a separating line, as indicated by the minus sample surrounded by a red disk. The alphas were bounded and so the contribution of this misclassified point is limited and the algorithm converges to a global optimum.

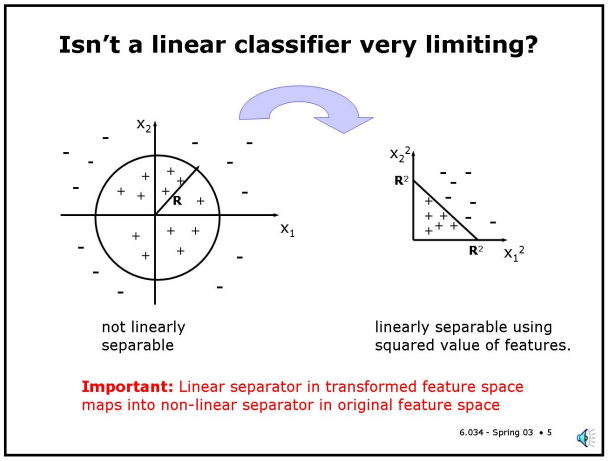
Second, the added point produced quite a different solution. The algorithm is looking for best possible dividing line; a tradeoff between margin and classification error defined by C. If we had kept a solution close to the one in the previous slide, the rogue plus point would have been misclassified by a lot, while with this solution we have reduced the misclassification margin substantially.

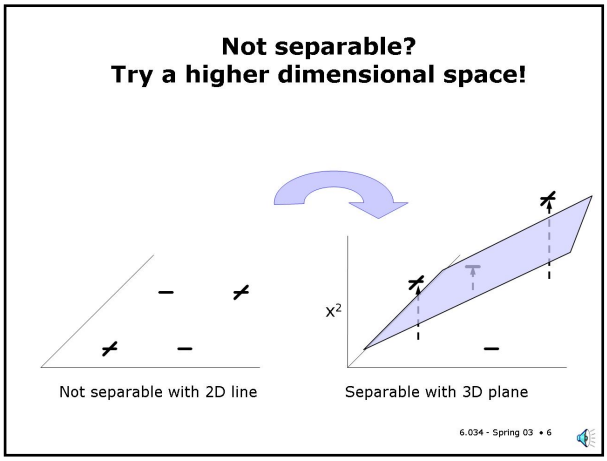
Slide 4.5.5

However, even if we provide a mechanism for ignoring noise points, aren't we really limited by a linear classifier? Well, yes.

However, in many cases, if we transform the feature values in a non-linear way, we can transform a problem that was not linearly separable into one that is. This example, shows that we can create a circular separator by finding a linear classifier in a feature space defined by the squares of the original feature values. That is, we can obtain a non-linear classifier in the original space by finding a linear classifier in a transformed space.

Hold that thought.





Slide 4.5.6

Furthermore, when training samples are not separable in the original space they may be separable if you perform a transformation into a higher dimensional space, especially one that is a non-linear transformation of the input space.

For the example shown here, in the original feature space, the samples all lie in a plane, and are not separable by a straight line. In the new space, the samples lie in a three dimensional space, and happen to be separable by a plane.

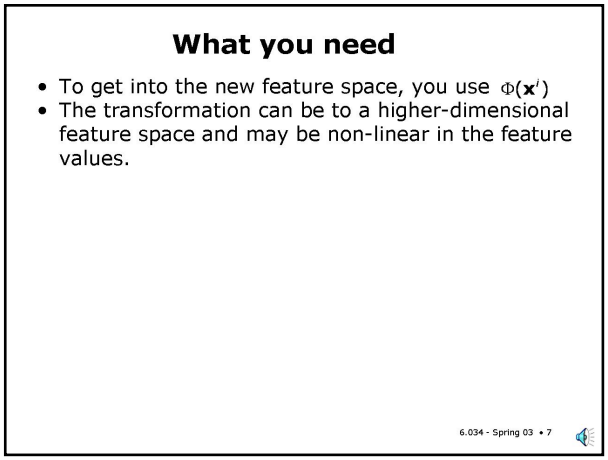
The heuristic of moving to a higher dimensional space is general, and does not depend on using SVMs.

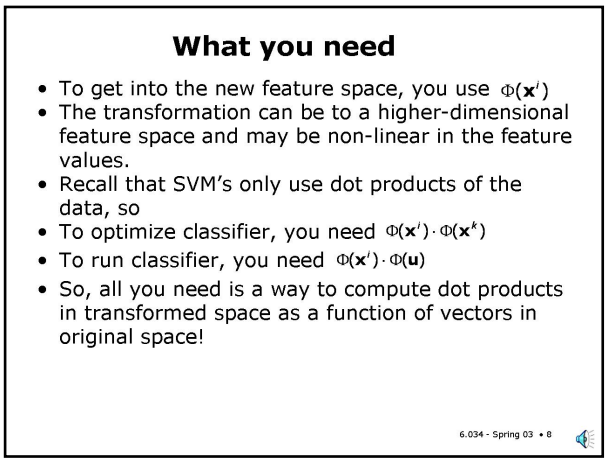
However, we will see that the support vector approach lends itself to movement into higher dimensional spaces because of the exclusive dependence of the support vector approach on dot products for learning and subsequent classification.

Slide 4.5.7

First, suppose there is a function, phi, that puts the vectors into another, higher-dimensional space, which will also typically involve a non-linear mapping of the feature values. In general, the higher the dimensionality, the more likely there will be a separating hyperplane.

By moving to a higher-dimensional feature space, we are also moving to a bigger hypothesis class, and so we might be worried about overfitting. However, because we are finding the maximum margin separator, the danger of overfitting is greatly reduced.





Slide 4.5.8
Even if we aren't in danger of overfitting, there might be computational problems if we move into higher dimensional spaces. In real applications, we might want to move to orders of magnitude more features, or even (in some sense) infinitely many features! We'll need a clever trick to manage this...

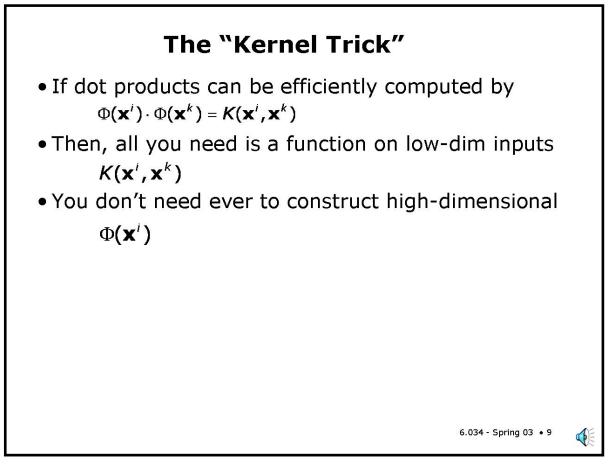
You have learned that to work in any space with the support vector approach, you will need (only) the dot products of the samples to train and you will need the dot products of the samples with unknowns to classify.

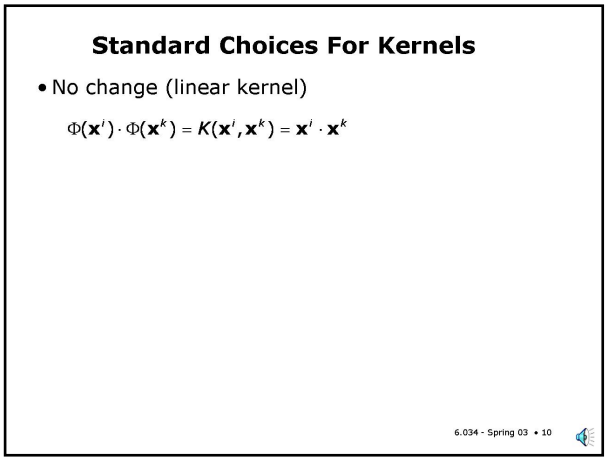
Note that you don't need anything else. So, all we need is a way of computing the dot product between the transformed feature vectors.

Slide 4.5.9

Let's assume that we have a function that allows us to compute the dot products of the transformed vectors in a way that depends only on the original feature vectors and not directly on the transformed vectors. We will call this the **kernel function**. (This usage of the term "kernel" is related to kernel functions we saw in regression; they are both about measuring effective distances between points in different spaces.)

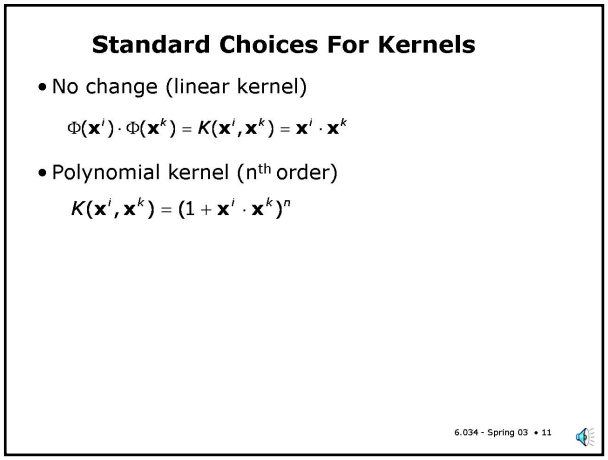
Then you do not need to know how to do the transformations themselves! This is why the support-vector approach is so appealing. The actual transformations may be computationally intractable, or you may not even know how to do the transformations at all, but you can still learn and classify without ever moving explicitly up into the high-dimensional space.

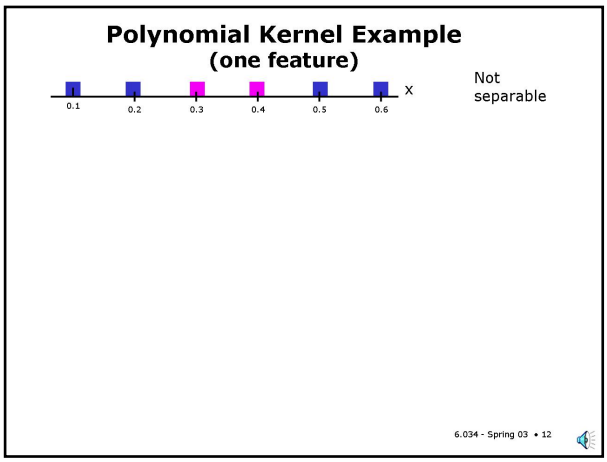




Slide 4.5.10
So now we need to find some phis (mappings from low to high-dimensional space) that have a convenient kernel function associated with them. The simplest case is one where phi is the identity function and K is just the dot product.

Slide 4.5.11
One such other kernel function is the dot product raised to a power; the actual power is a parameter of the learning algorithm that determines the properties of the solution.





Slide 4.5.12

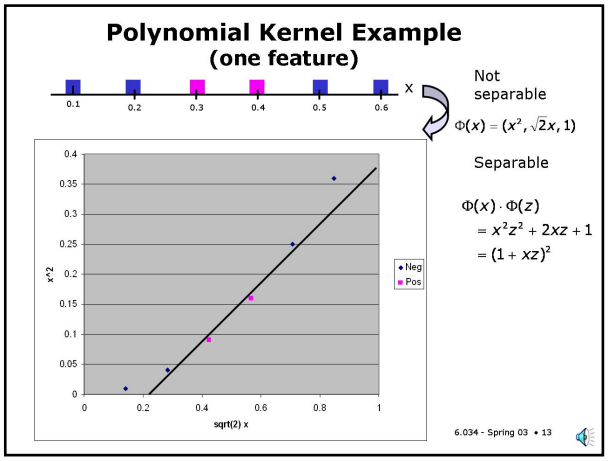
Let's look at a simple example of using a polynomial kernel. Consider the one dimensional problem shown here, which is clearly not separable. Let's map it into a higher dimensional feature space using the polynomial kernel of second degree (n=2).

Slide 4.5.13

Note that a second degree polynomial kernel is equivalent to mapping the single feature value x to a three dimensional space with feature values x^2, sqrt(2)x, and 1. You can see that the dot product of two of these feature vectors is exactly the value computed by the polynomial kernel function.

If we plot the original points in the transformed feature space (using just the first two features), we see in fact that the two classes are linearly separable. Clearly, the third feature value (equal to 1) will be irrelevant in finding a separator.

The important aspect of all of this is that we can find and use such a separator without ever explicitly computing the transformed feature vectors - only the kernel function values are required.



Polynomial Kernel

- Polynomial kernel for n=2 and features $\mathbf{x}=[x_1\ x_2]$

$$K(\mathbf{x},\mathbf{z}) = (1+\mathbf{x}\cdot\mathbf{z})^2$$

is equivalent to the following feature mapping:

$$\Phi(\mathbf{x}) = [x_1^2\ x_2^2\ \sqrt{2}x_1x_2\ \sqrt{2}x_1\ \sqrt{2}x_2\ 1]$$

- We can verify that:

$$\begin{aligned}\Phi(\mathbf{x})\cdot\Phi(\mathbf{z}) &= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 + 2x_1z_1 + 2x_2z_2 + 1\\ &= (1 + x_1z_1 + x_2z_2)^2\\ &= (1+\mathbf{x}\cdot\mathbf{z})^2\\ &= K(\mathbf{x},\mathbf{z})\end{aligned}$$

6.034 - Spring 03 • 14

Slide 4.5.14

Here is a similar transformation for a two dimensional feature vector. Note that the dimension of the transformed feature vector is now 6. In general, the dimension of the transformed feature vector will grow very rapidly with the dimension of the input vector and the degree of the polynomial.

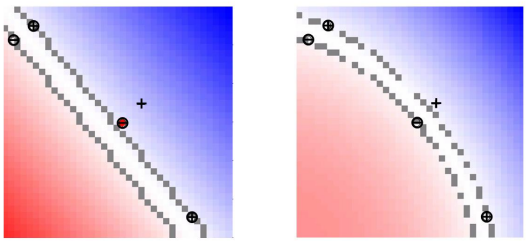
Slide 4.5.15

Let's look at the behavior of these non-linear kernels.

The decision surface produced by the non-linear kernels is curved. Here is an example for which the (unsuccessful) attempt on the left is with a simple dot product; the attempt on the right is done with a polynomial kernel of degree 3. Note the curve in the solution, and note that four of the samples have become support vectors.

Generally, the higher-dimensional the transformed space, the more complex the separator is in the original space, and the more support vectors will be required to specify it.

Polynomial Kernel



Images by Patrick Winston

6.034 - Spring 03 • 15

Standard Choices For Kernels

- No change (linear kernel)

$$\Phi(\mathbf{x}^i)\cdot\Phi(\mathbf{x}^k) = K(\mathbf{x}^i,\mathbf{x}^k) = \mathbf{x}^i\cdot\mathbf{x}^k$$

- Polynomial kernel (n^{th} order)

$$K(\mathbf{x}^i,\mathbf{x}^k) = (1+\mathbf{x}^i\cdot\mathbf{x}^k)^n$$

- Radial basis kernel (σ is standard deviation)

$$K(\mathbf{x}^i,\mathbf{x}^k) = e^{\frac{-\|\mathbf{x}^i-\mathbf{x}^k\|^2}{2\sigma^2}} = e^{\frac{-(\mathbf{x}^i-\mathbf{x}^k)\cdot(\mathbf{x}^i-\mathbf{x}^k)}{2\sigma^2}}$$

6.034 - Spring 03 • 16

Slide 4.5.16
Another popular kernel function is an exponential of the square of the distance between vectors, divided by sigma squared. This is the formula for a Gaussian bump in the feature space, where sigma is the standard deviation of the Gaussian. Sigma is a parameter of the learning that determines the properties of the solution.

Slide 4.5.17

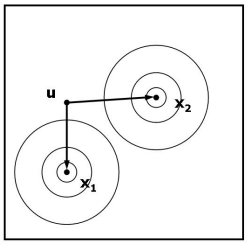
You can get a curved separator if you use radial basis functions, which give us a classifier that is a sum of the values of several Gaussian functions.

Let's pause a minute to observe something that should strike you as a bit weird. When we used the polynomial kernels, we could see that each input feature vector was being mapped into a higher-dimensional, possibly very high dimensional, feature vector. With the radial-basis kernel each input feature vector is being mapped into a **function** that is defined over the whole feature space! In fact, each input feature point is being mapped into a point in an infinite-dimensional feature space (known as a Hilbert space). We then build the classifier as sum of these functions. Whew!

The actual operation of the process is less mysterious than this "infinite-dimensional" mapping view, as we will see by a very simple example.

Radial-basis kernel

- Classifier based on sum of Gaussian bumps with standard deviation σ, centered on support vectors.

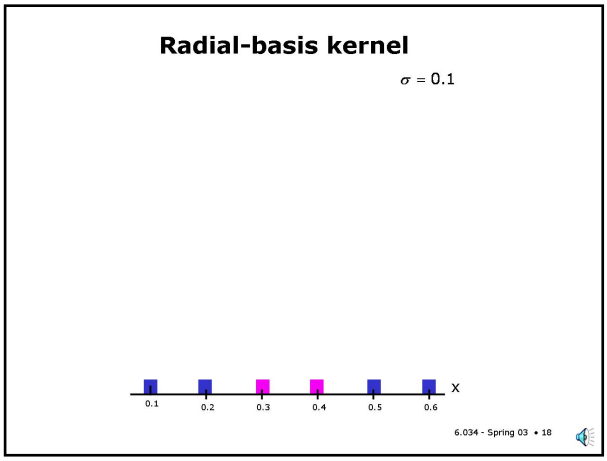


$$h(\mathbf{u}) = sign[h'(\mathbf{u})]$$

$$h'(\mathbf{u}) = \sum_{i=1}^{k}\alpha_i y^i K(\mathbf{x}^i,\mathbf{u}) + b$$

$$K(\mathbf{x}^i,\mathbf{u}) = e^{\frac{-\|\mathbf{x}^i-\mathbf{u}\|^2}{2\sigma^2}}$$

6.034 - Spring 03 • 17



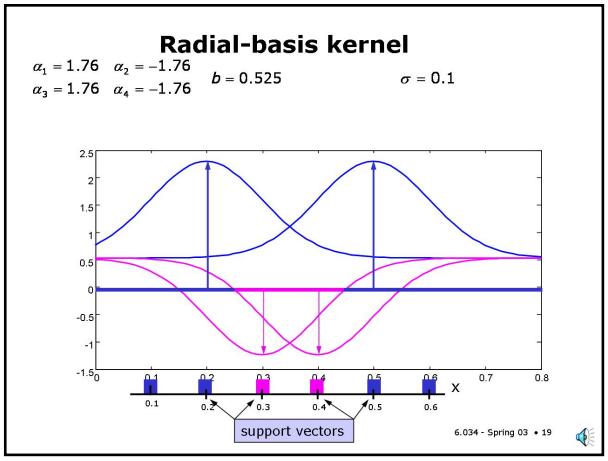
Slide 4.5.18
Along the bottom you see that we're dealing with the simple one-dimensional example that we looked at earlier using a polynomial kernel. The blue points are possitive and the pinkish purple ones are negative. Clearly this arrangement is not linearly separable.

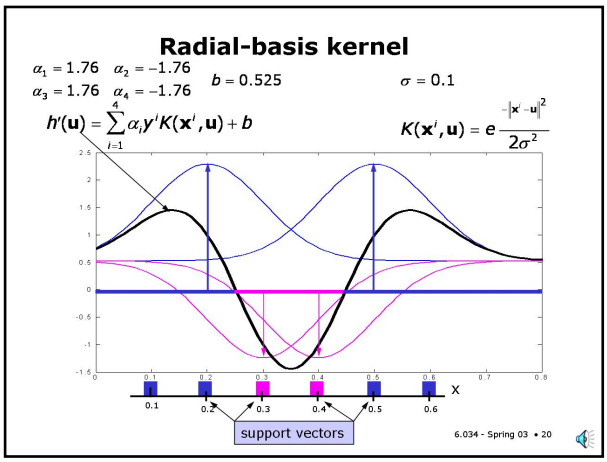
$K(x^i,u)$ can be seen as a "Gaussian bump"; that is, as a function with a maximum at $u = x^i$, that decreases monotonically with the distance between u and x^i, but is always positive and goes to 0 at infinite distance. The parameter sigma specifies how high the bump is and how fast it falls off (the area under the curve of each bump is 1, no matter what the value of sigma is). The smaller the sigma, the more sharply peaked the bump.

With a radial-basis kernel, we will be looking for a set of multipliers for Gaussian bumps with the specified sigma (here it is 0.1) so that the sum of these bumps (plus an offset) will give us a classification function that's positive where the positive points are and negative where the negative points are.

Slide 4.5.19
Here is the solution obtained from an SVM quadratic optimization algorithm. Note that four points are support vectors, as expected, the points near where the decision boundary has to be. The farther positive points receive alpha=0. The value of the offset, *b* is also shown.

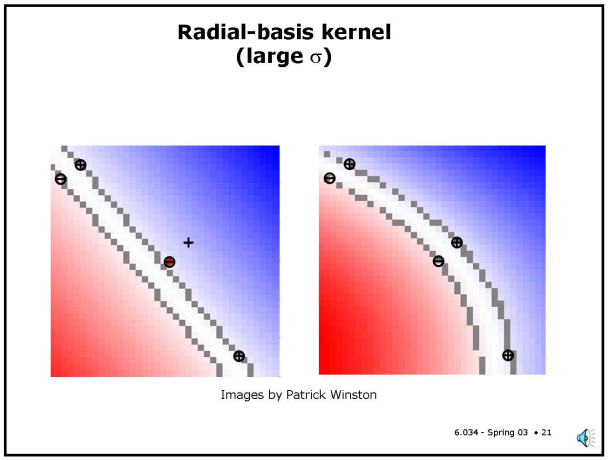
The blue and pink Gaussian bumps correspond to copies of a Gaussian with standard deviation of 0.1 scaled by the corresponding alpha values.

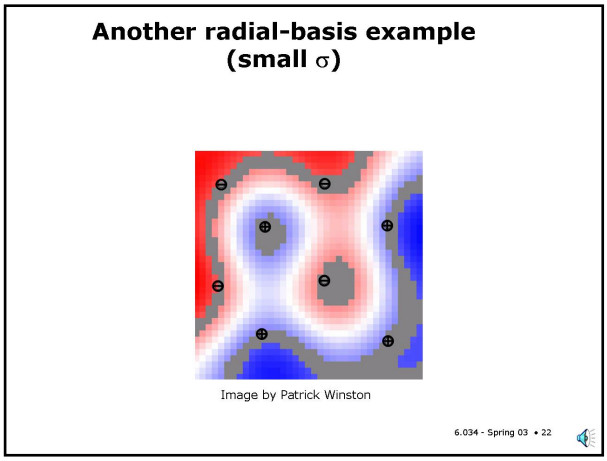




Slide 4.5.20
The black line corresponds to the sum of the four bumps (and the offset). The important point is to notice where this line crosses zero since that's the decision surface (in one dimension). Notice that, as required, it succeeds in separating the positive from the negative points.

Slide 4.5.21
Here we see a separator for our simple five point example computed using radial basis kernels. The solution on the left, for reference, is the original dot product. The solution on the right is for a radial basis function with a sigma of one. Note that all the points are now support vectors.





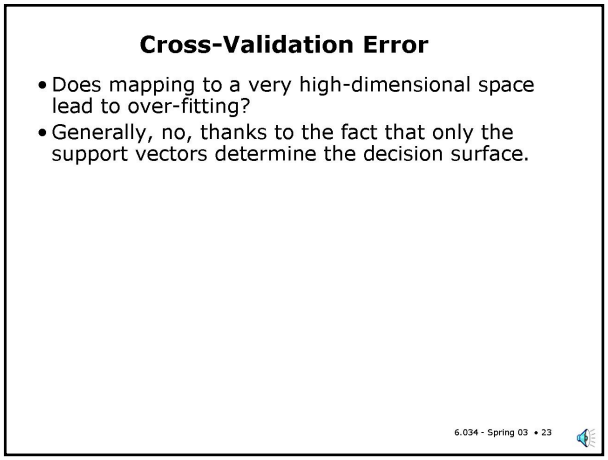
Slide 4.5.22

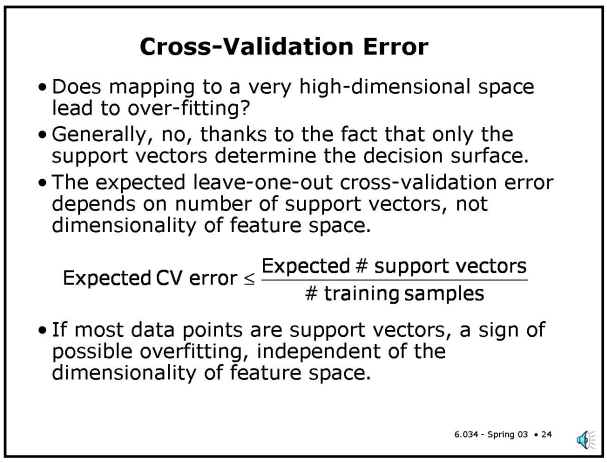
If a space is truly convoluted, you can always cover it with a radial basis solution with small-enough sigma. In extreme cases, like this one, each of the four plus and four minus samples has become a support vector, each specialized to the small part of the total space in its vicinity. This is basically similar to 1-nearest neighbor and is just as powerful and subject to overfitting.

Slide 4.5.23

At this point alarm bells may be ringing. By creating these very high dimensional feature vectors, are we just setting ourselves up for severe overfitting? Intuitively, the more parameters we have the better we can fit the input, but that may not lead to better performance on new data.

It turns out that the fact that the SVM decision surface depends only on the support vectors and not directly on the dimensionality of the space comes to our rescue.





Slide 4.5.24

We can estimate the error on new data by computing the cross-validation error on the training data. If we look at the linearly separable case, it is easy to see that the expected value of leave-one-out cross-validation error is bounded by the proportion of support vectors.

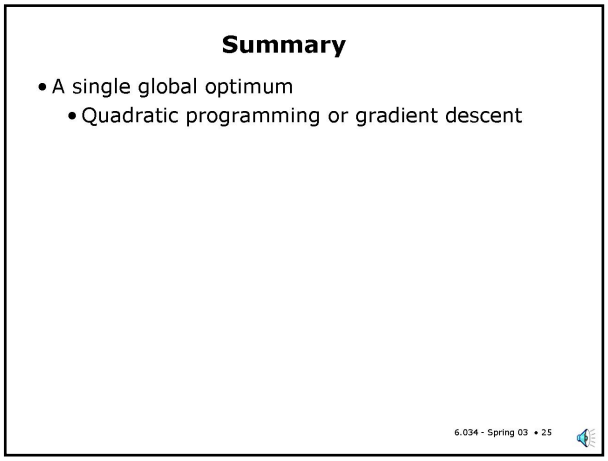
If we take a data point that is not a support vector from the training set, the computation of the separator will not be affected and so it will be classified correctly. If we take a support vector out, then the classifier will in general change and there may be an error. So, the expected generalization error depends on the number of support vectors and not on the dimension.

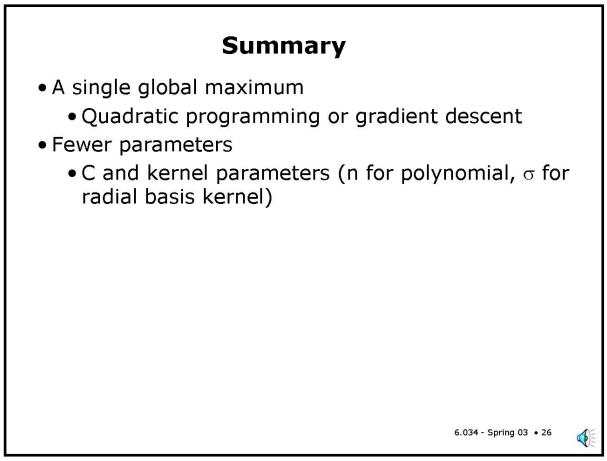
Note that using a radial basis kernel with very small sigma gives you a high expected number of support vectors and therefore a high expected cross-validation error, as expected. Yet, a radial basis kernel with large sigma, although of similar dimensionality, has fewer expected support vectors and is likely to generalize better.

We shouldn't take this bound too seriously; it is not actually very predictive of generalization performance in practice but it does point out an important property of SVMs - that generalization performance is more related to expected number of support vectors than to dimensionality of the transformed feature space.

Slide 4.5.25

So, let's summarize the SVM story. One key point is that SVMs have a training method that guarantees a unique global optimum. This eliminate many headaches in other approaches to machine learning.





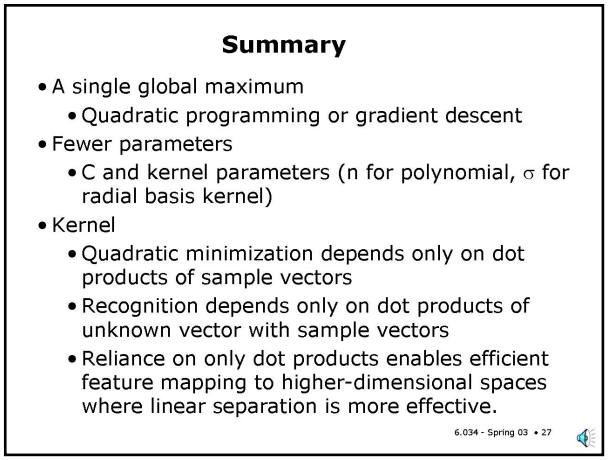
Slide 4.5.26

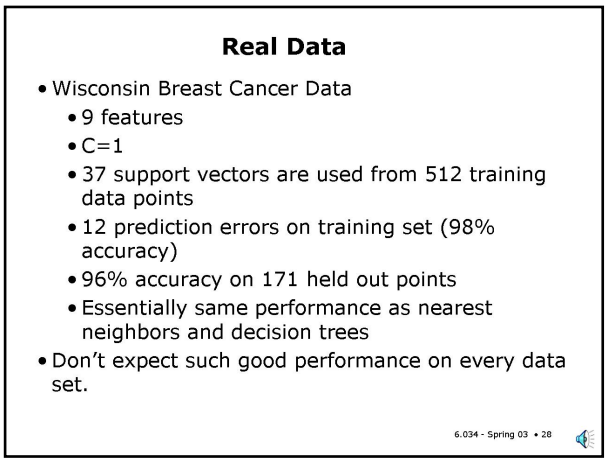
The other advantage of SVMs is that there are relatively few parameters to be chosen: C, the constant used to trade off classification error and width of the margin; and the kernel parameter, such as sigma in the radial basis kernel.

These can both be continuous parameters and so there still remains a search requiring some form of validation, but these are few parameters compared to some of the other methods.

Slide 4.5.27

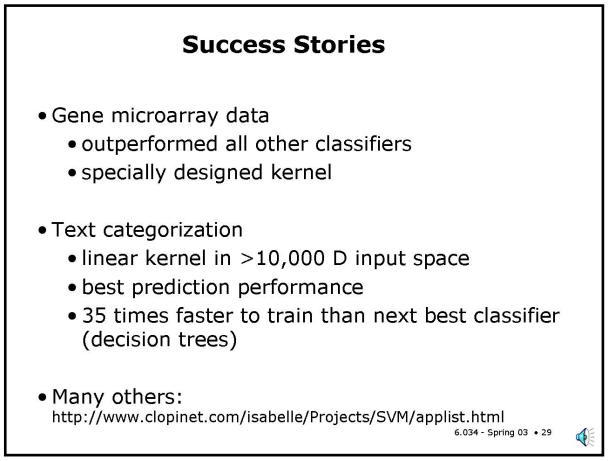
And, last but not least, is the kernel trick. That is, that the whole process depends only on the dot products of the feature vectors, which is the key to the generalization to non-linear classifiers.





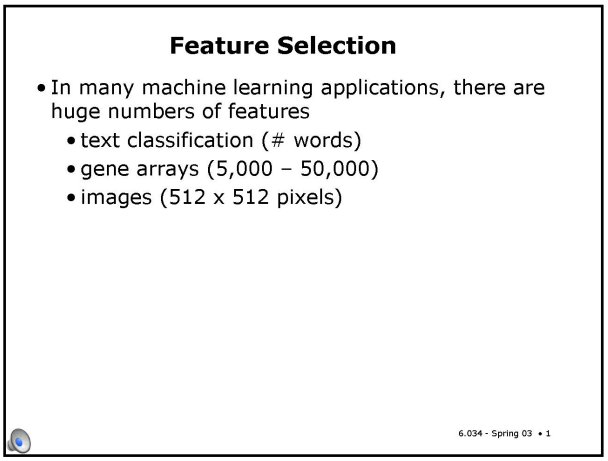
Slide 4.5.28
The linear separator is very simple hypothesis class but it can perform very well on appropriate data sets. On the Wisconsin breast cancer data, the maximal margin classifier, with a linear kernel, does as well or better as any of the other classifiers we have seen on held-out data. Note that only 37 of the 512 training points are support vectors.

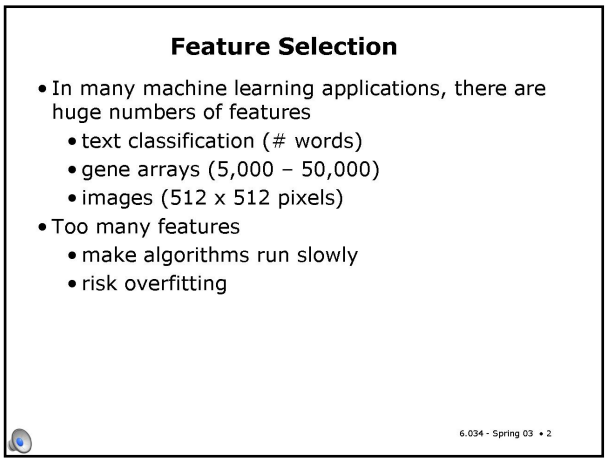
Slide 4.5.29
SVMs have proved useful in a wide variety of applications, particularly those with large numbers of features, such as image and text recognition problems. They are the method of choice in text classification problems, such as categorization of news articles by topic, or spam detection, because they can work in a huge feature space (typically with a linear kernel) without too much fear of overfitting.



6.034 Notes: Section 4.6

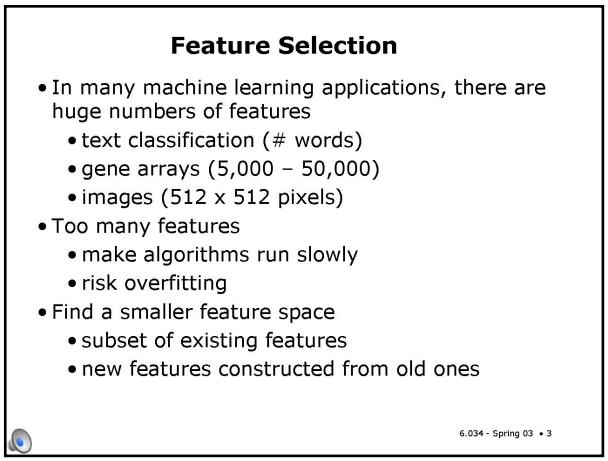
Slide 4.6.1
In many machine-learning applications, there are huge numbers of features. In text classification, you often have as many features as there are words in the dictionary. Gene expression arrays have five to fifty thousand elements. Images can have as many as 512 by 512 pixels.

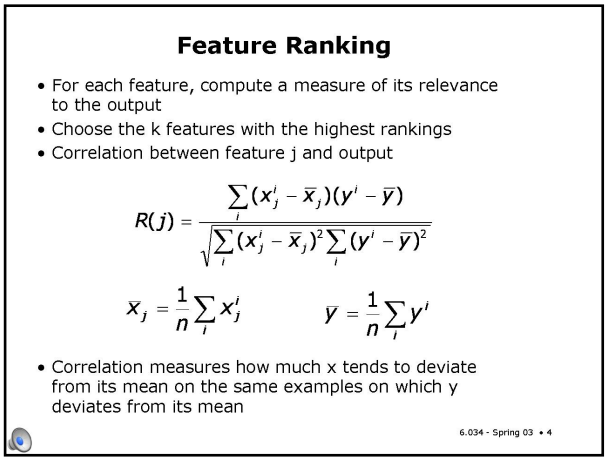




Slide 4.6.2
When there are lots of features in a domain, it can make some machine learning algorithms run much too slowly. Worse, it often causes overfitting problems: most classifiers have a complexity related to the number of features, and in many of these cases we can have many more features than training examples, which doesn't give us much confidence in our parameter estimates.

Slide 4.6.3
There are two approaches to dealing with very large feature spaces: one is to select a subset of the given set of features to work with; the other is to make new features that are supposed to describe the input space more efficiently than the given set of features.



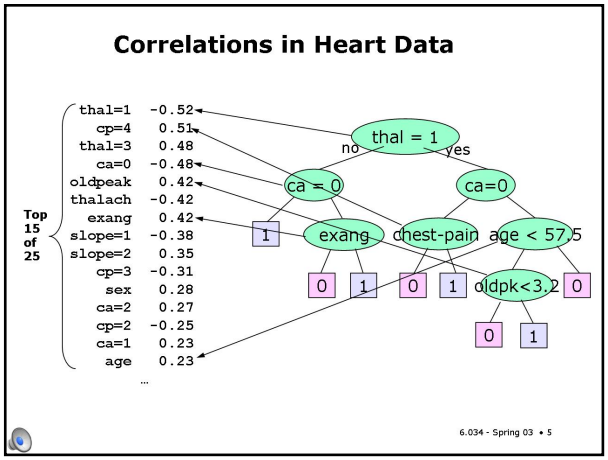


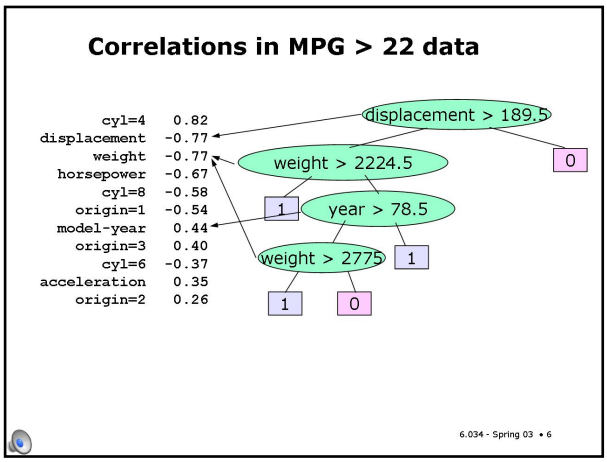
Slide 4.6.4
The simplest feature-selection strategy is to compute some score for each feature, and then select the k features with the highest rankings.

A popular feature score is the correlation between a feature and the output variable. It measures the degree to which a feature varies with the output, and is usable when the output is discrete or continuous.

Slide 4.6.5
We computed the correlations of each of the features in the heart disease data set with the output. They are shown here in sorted order, with reference to the decision tree we learned on this data.

We can see that most of the features used in the tree show up among the top features, ranked according to correlation. You can see the features with a positive correlation score indicate that heart disease is more likely, and those with a negative score indicate that it is less likely.

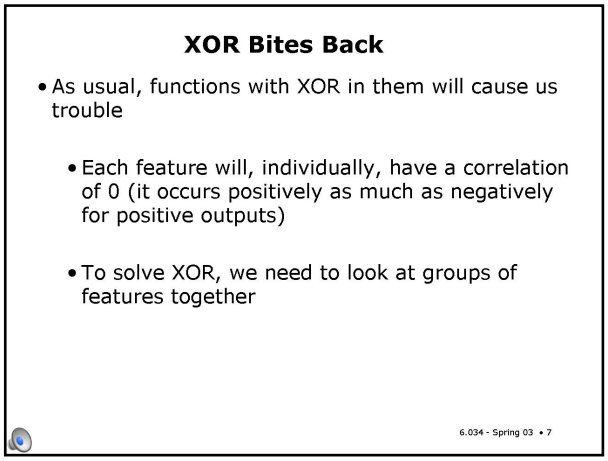


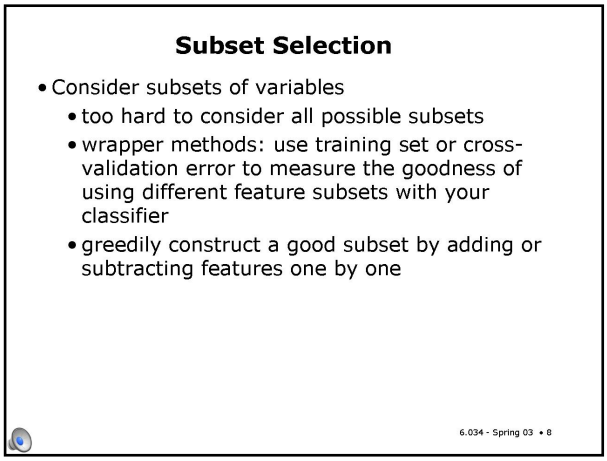


Slide 4.6.6
Here's a similar figure for the auto fuel efficiency data. It's interesting to see that the highest-correlation feature is binary choice about whether there are 4 cylinders. It looks like binary features have a tendency to be preferred (since the output is binary, as well, and so they often match up perfectly). But displacement is also very highly ranked, and probably contains more information than the number of cylinders.

Slide 4.6.7
As usual, XOR will cause us trouble if we do scoring of single features. In an XOR problem, each feature will, individually, have a correlation of 0 with the output.

To solve xor problems, we need to look at groups of features together.



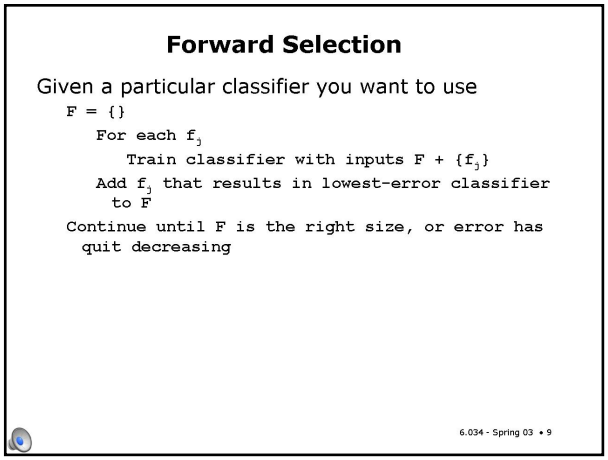


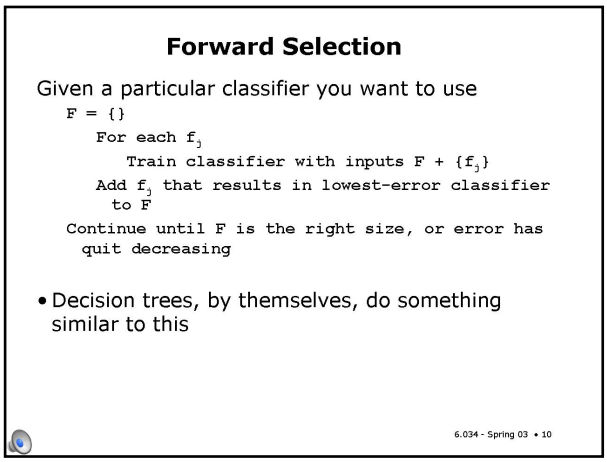
Slide 4.6.8
Ideally, we'd like to try all possible subsets of the features and see which one works best. We can evaluate a subset of features by training a classifier using just that subset, and then measuring the performance using training set or cross-validation error.

Instead of trying all subsets, we'll consider greedy methods that add or subtract features one at a time.

Slide 4.6.9
In the forward selection method, we start with no features at all in our feature set. Then, for each feature, we consider adding it to the feature set: we add it, train a classifier, and see how well it performs (on a separate validation set or by using cross-validation). We then add the feature that generated the best classifier to our existing set and continue.

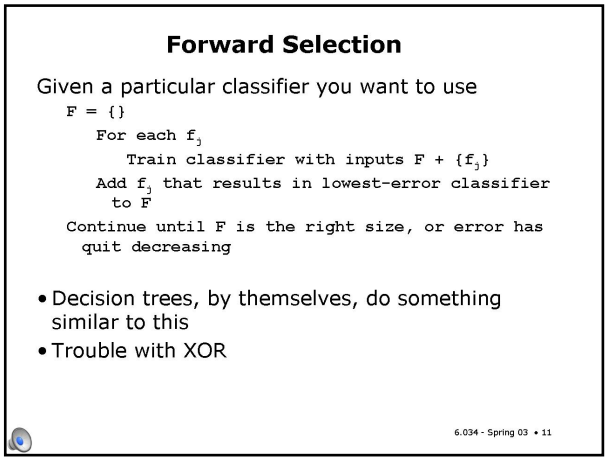
We'll terminate the algorithm when we have as many features as we can handle, or when the error has quit decreasing.

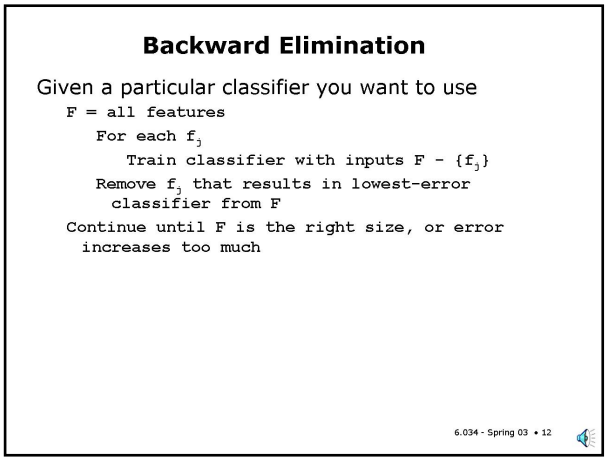




Slide 4.6.10
Decision trees work sort of like this: they add features one at a time, choosing the next feature in the context of the ones already chosen. However, they establish a whole tree of feature-selection contexts.

Slide 4.6.11
Even if we do forward selection, XOR can cause us trouble. Because we only consider adding features one by one, neither of the features will look particularly attractive individually, and so we would be unlikely to add them until the very end.



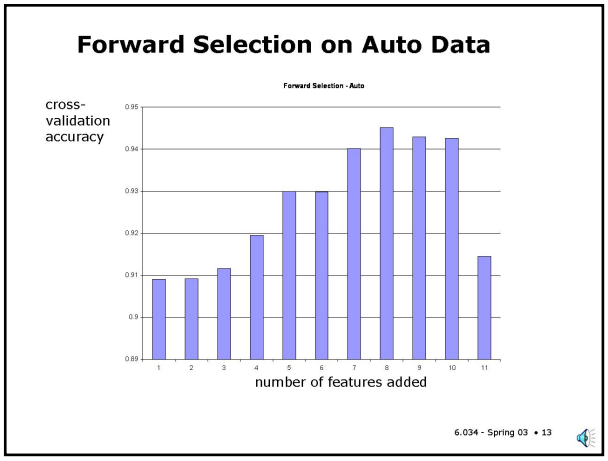


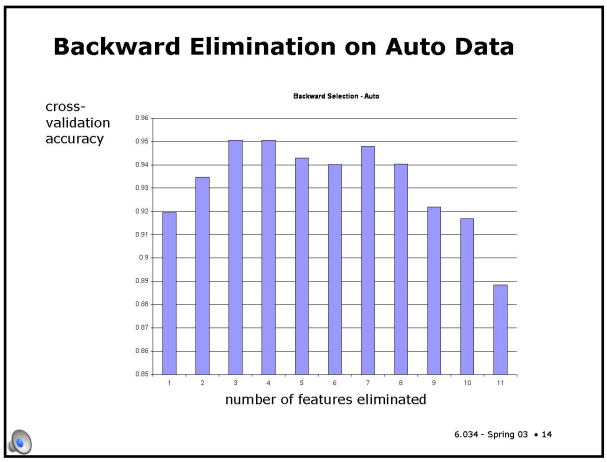
Slide 4.6.12
Backward elimination works in the other direction. It starts with all the features in the feature set and eliminates them one by one, removing the one that results in the best classifier at each step.

This strategy can cope effectively with XOR-like problems. But it might be impractical if the initial feature set is so large that it makes the algorithm to slow to run.

Slide 4.6.13
Here's a plot of the cross-validation accuracy against number of features chosen by forward selection on the auto data. The classifier we used was nearest neighbor.

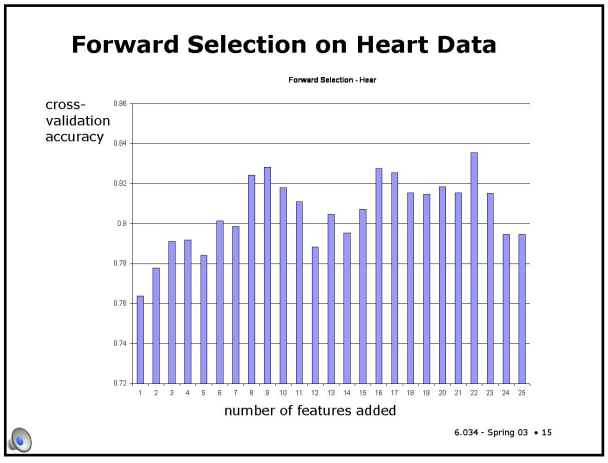
Here we can see that adding features improves cross-validation accuracy until the last couple of features. If there were a large number of relatively noisy features, adding them would make the performance go down even further, as they would give the classifier further opportunity for overfitting.

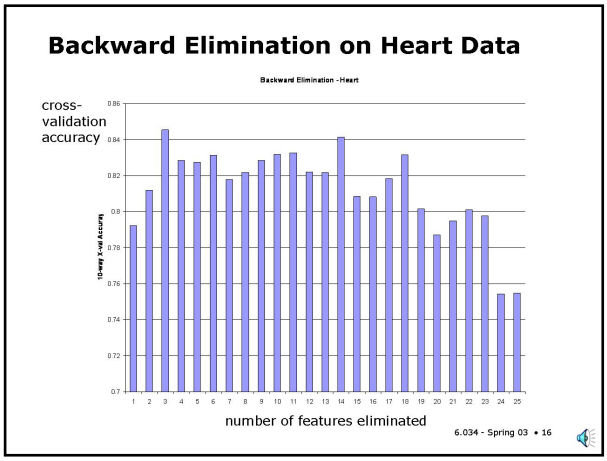




Slide 4.6.14
The picture for backward elimination is similar. But notice that it seems to work a bit better, even when we are eliminating a lot of features. This may be because it can decide which features to eliminate in the context of all the other features. Forward selection, especially in the early phases, picks features without much context.

Slide 4.6.15
On the heart data, we need about 8 features before we're getting reasonably good performance. The accuracies are pretty erratic after that; it's probably an indication of overall variance in the performance estimates.



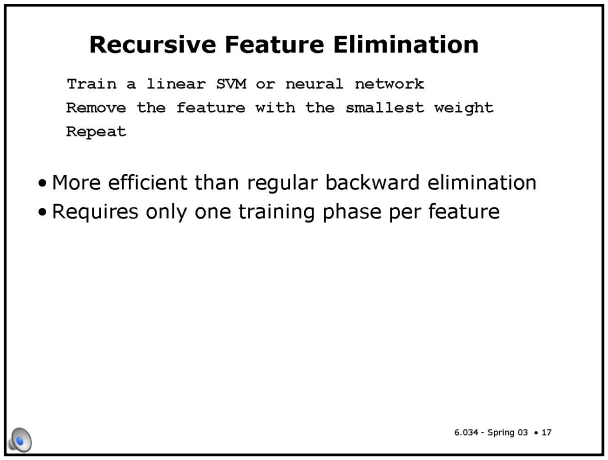


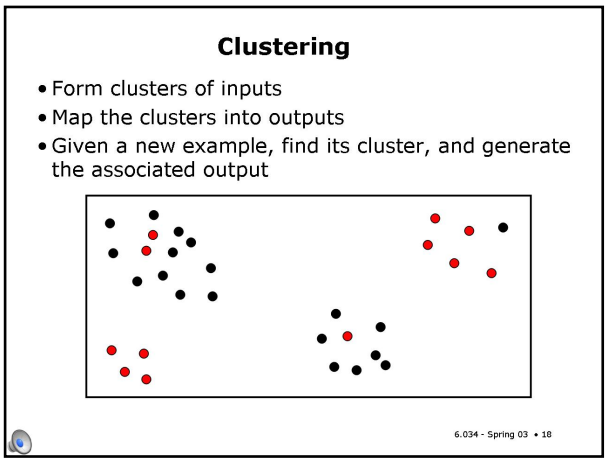
Slide 4.6.16
We can see similar performance with backward elimination. It's possible to get rid of a lot of features before performance suffers dramatically. And, it really seems to be worthwhile to eliminate some of the features, from a performance perspective.

Slide 4.6.17
Backward elimination and forward selection can be computationally quite expensive, because they require you, on each iteration, to train approximately as many classifiers as you have features.

In some classifiers, such as linear support-vector machines and linear neural networks, it's possible to do backward elimination more efficiently. You train the classifier once, and then remove the feature that has the smallest input weight.

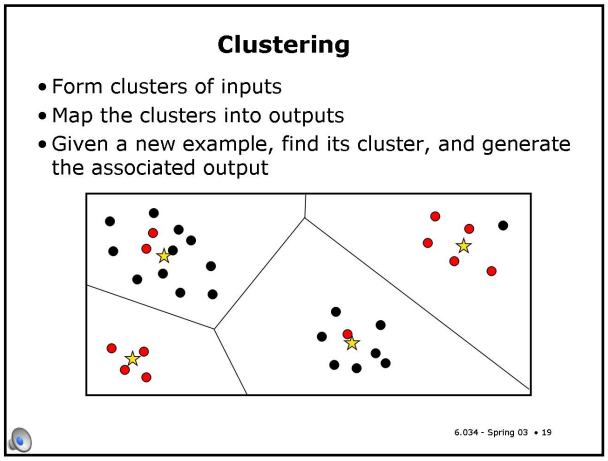
These methods can be extended to non-linear SVMs and neural networks, but it gets somewhat more complicated there.

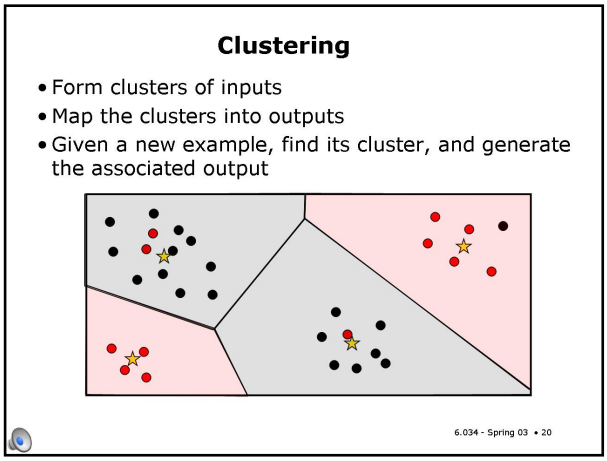




Slide 4.6.18
Another whole strategy for feature selection is to make new features. One very drastic method is to try to cluster all of the inputs in your data set into a relatively small number of groups, and then learn a mapping from each group into an output.

Slide 4.6.19
So, in this case, we might divide the input points into 4 clusters. The stars indicate the cluster centers.



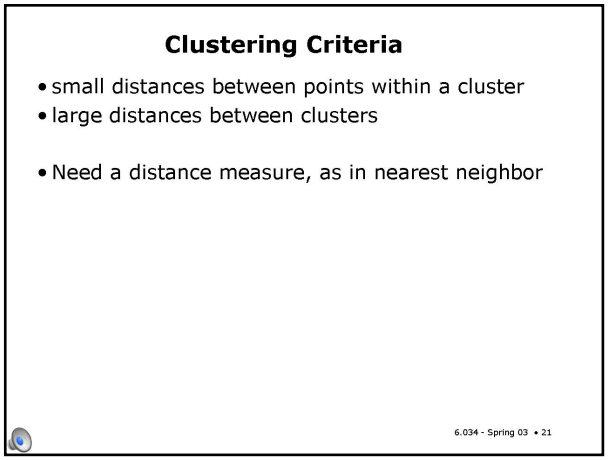


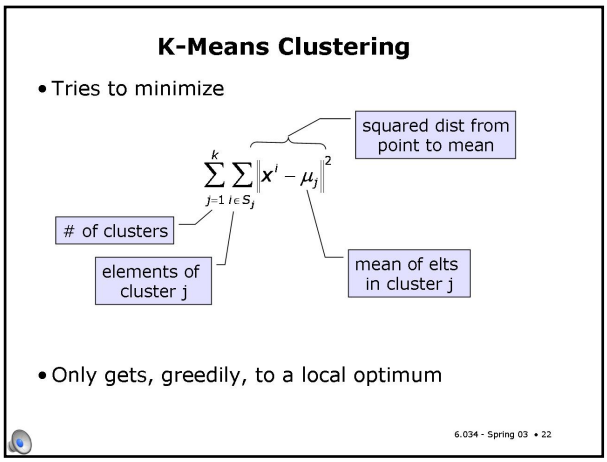
Slide 4.6.20
Then, for each cluster, we would assign the majority class. Now, to predict the value of a new point, we would see which region it would land in, and predict the associated class.

This is different from nearest neighbors in that we actually discard all the data except the cluster centers. This has the advantage of increasing interpretability, since the cluster centers represent "typical" inputs.

Slide 4.6.21
So, what makes a good clustering? There are lots and lots of different technical choices. The basic idea is usually that you want to have clusters in which the distance between points in the same group is small and the distance between points in different groups is large.

Clustering, like nearest neighbor, requires a distance metric, and the results you get are as scale-sensitive as they are in nearest-neighbor.

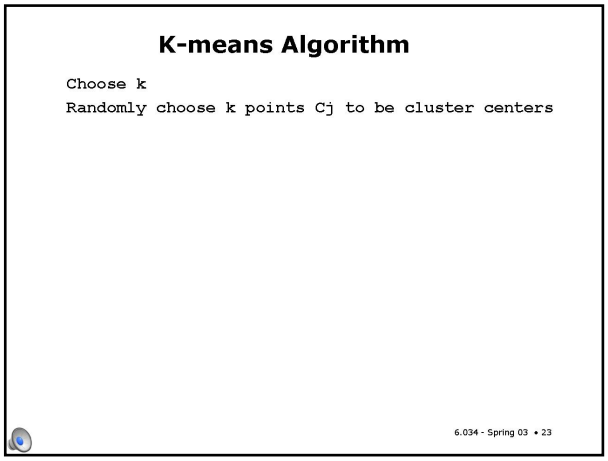


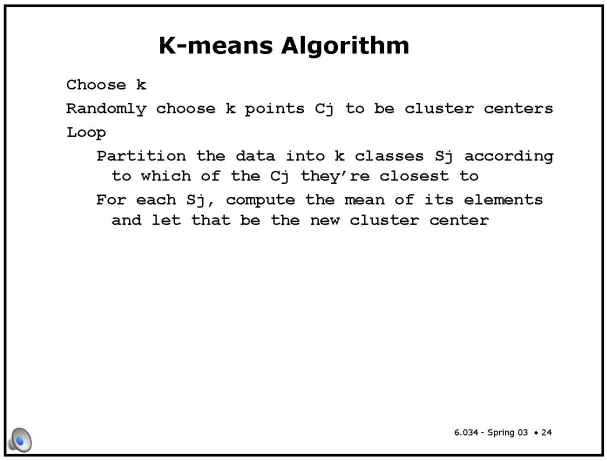


Slide 4.6.22
One of the simplest and most popular clustering methods is K-means clustering. It tries to minimize the sum, over all the clusters, of the variance of the points within the cluster (the distances of the points to the geometric center of the cluster).

Unfortunately, it only manages to get to a local optimum of this measure, but it's usually fairly reasonable.

Slide 4.6.23
Here is the code for the k-means clustering algorithm. You start by choosing k, your desired number of clusters. Then, you can randomly choose k of your data points to serve as the initial cluster centers.

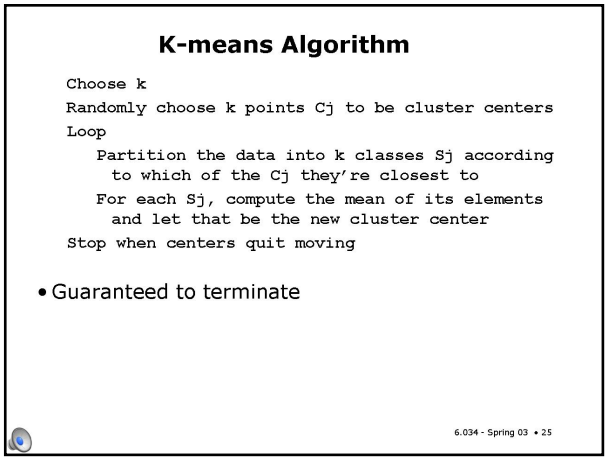


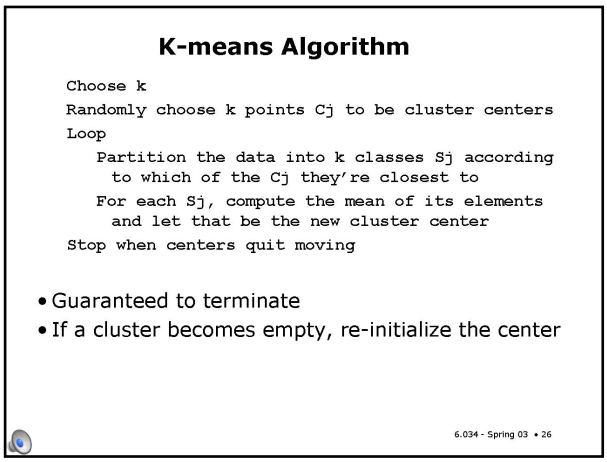


Slide 4.6.24
Then, we enter a loop with two steps. The first step is to divide the data up into k classes, using the cluster centers to make a Voronoi partition of the data. That is, we assign each data point to the cluster center that it's closest to.

Now, for each new cluster, we compute a new cluster center by averaging the elements that were assigned to that cluster on the previous step.

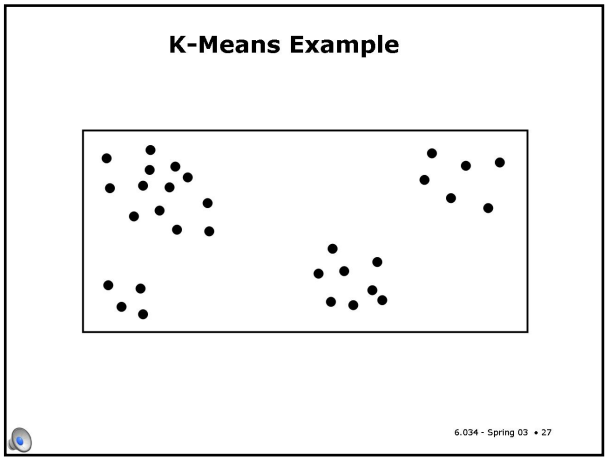
Slide 4.6.25
We stop when the centers quit moving. This process is guaranteed to terminate.

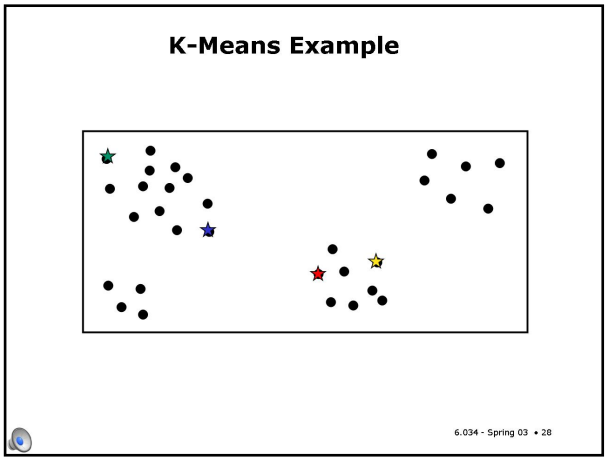




Slide 4.6.26
One possible problem is that cluster centers can become "orphaned". That is, they no longer have any points in them (or perhaps just a single point). A standard method for dealing with this problem is simply to randomly re-initialize that cluster center.

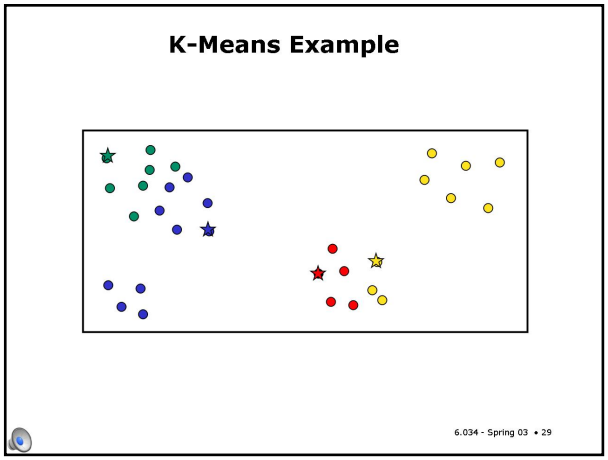
Slide 4.6.27
Here's a running example simulation of the k-means algorithm. We start with this set of input points.

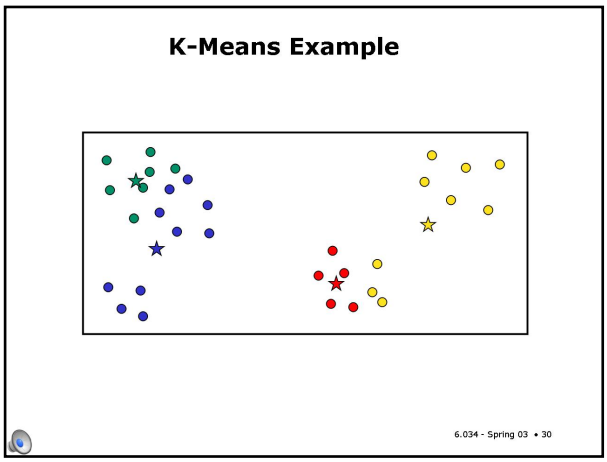




Slide 4.6.28
And randomly pick 4 of them to be our cluster centers.

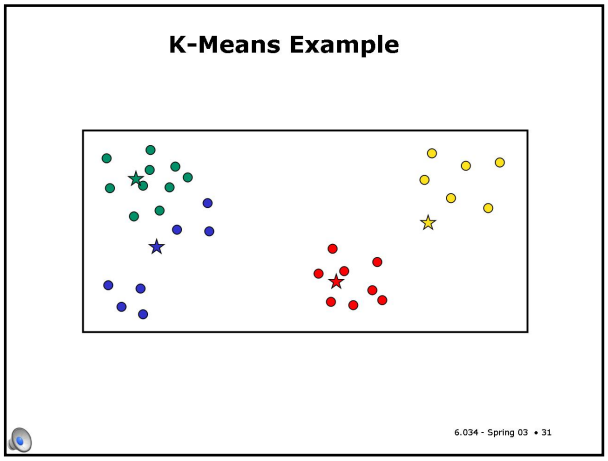
Slide 4.6.29
Now we partition the data, assigning each point to the center to which it is closest.

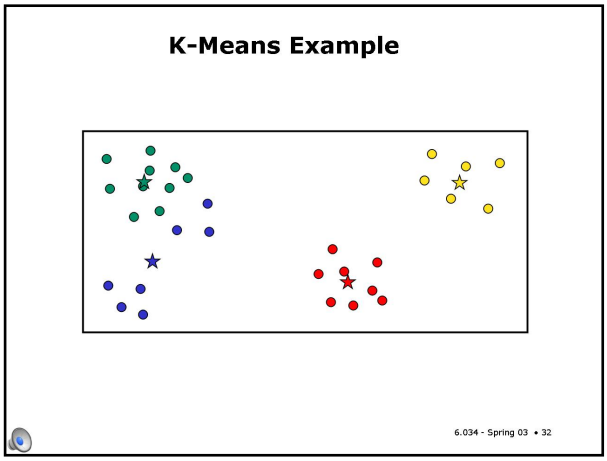




Slide 4.6.30
We move each center to the mean of the points that belong to it.

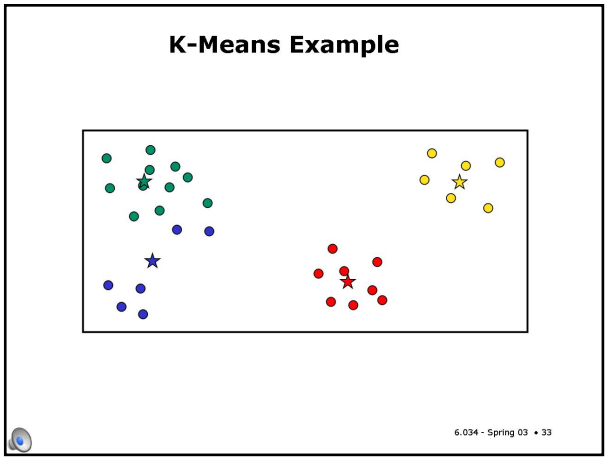
Slide 4.6.31
Having moved the means, we can now do a new reassignment of points.

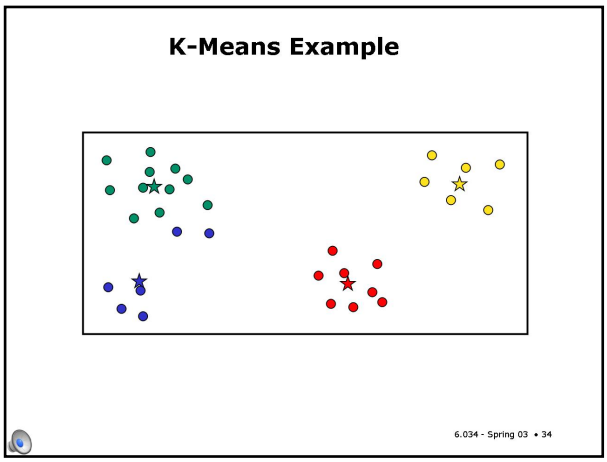




Slide 4.6.32
And recompute the centers.

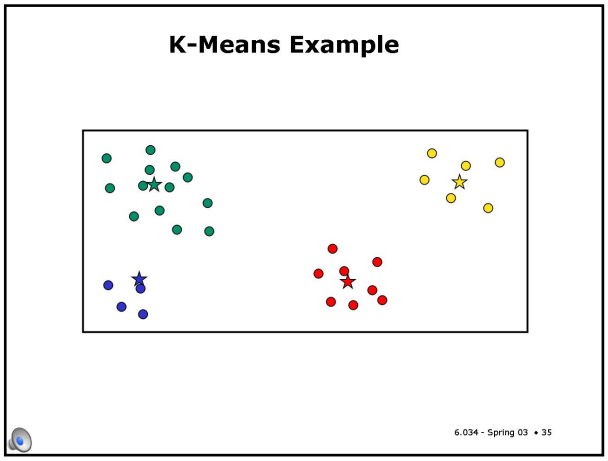
Slide 4.6.33
Here we reassign one more point to the green cluster,

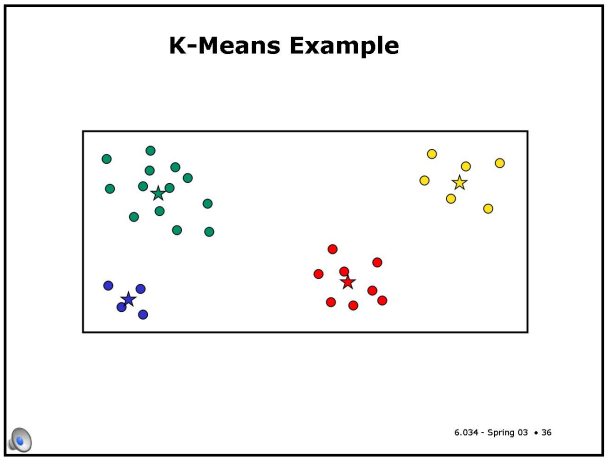




Slide 4.6.34
Which causes the green and blue centers to move a bit. At this point, the red and yellow clusters are stable.

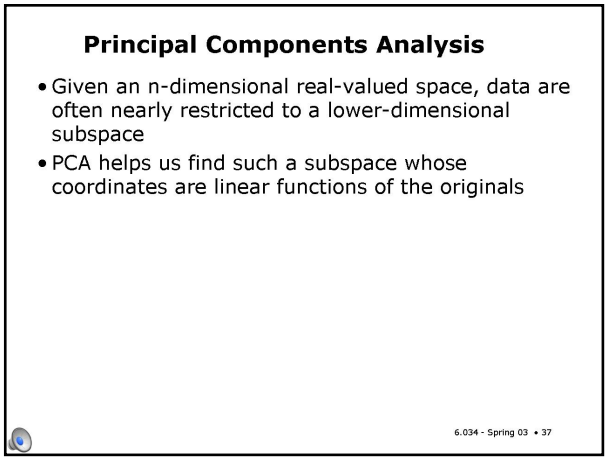
Slide 4.6.35
Now two more points get reassigned to green,

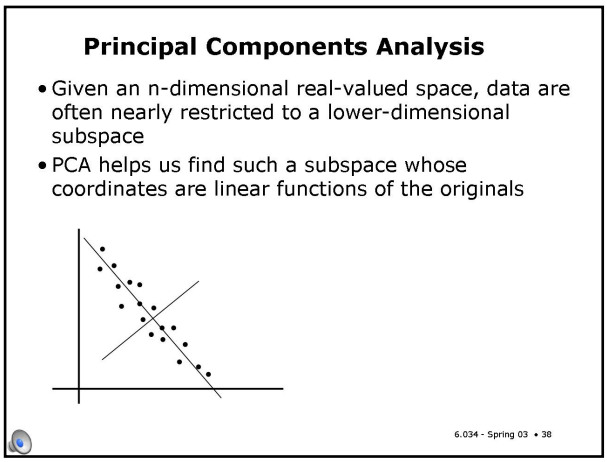




Slide 4.6.36
And we recompute the centers, to get a clustering that is stable, and will not change under further iterations.

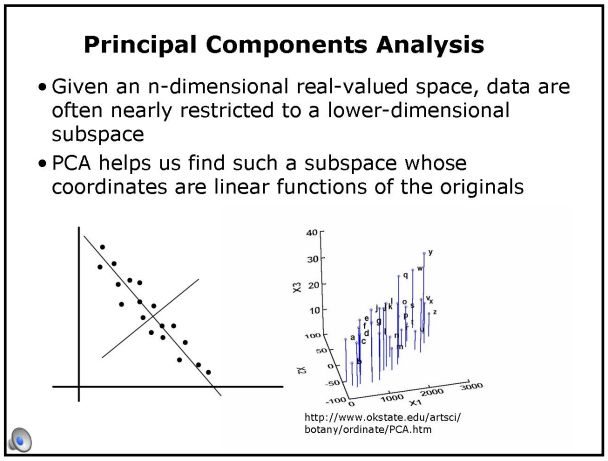
Slide 4.6.37
The k-means algorithm takes a real-valued input space and generates a one-dimensional discrete description of the inputs. In principal components analysis, we take a real-valued space, and represent the data in a new multi-dimensional real-valued space with lower dimensionality. The new coordinates are linear combinations of the originals.

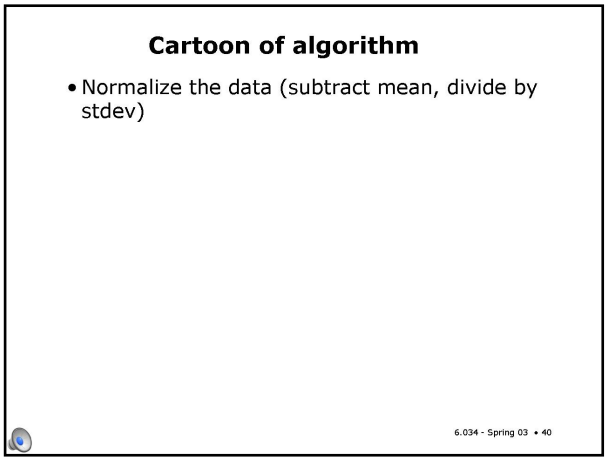




Slide 4.6.38
The idea is that even if your data are described using a large number of dimensions, they may lie in a lower-dimensional subspace of the original space. So, in this figure, the data are described with two dimensions, but a single dimension that runs diagonally through the data would describe it without losing too much information.

Slide 4.6.39
It's harder to see in three dimensions, but here's a data set that might be effectively described using only two dimensions.

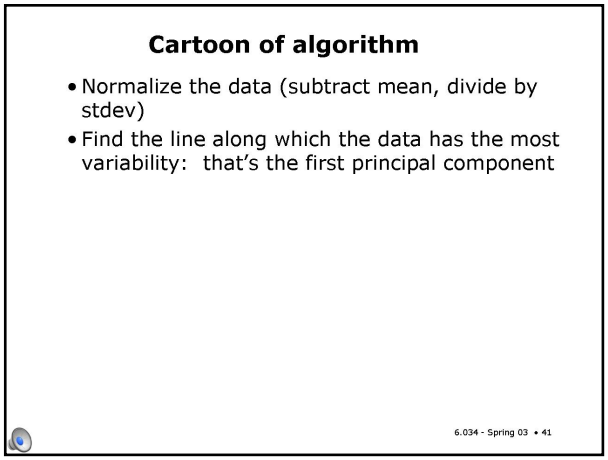


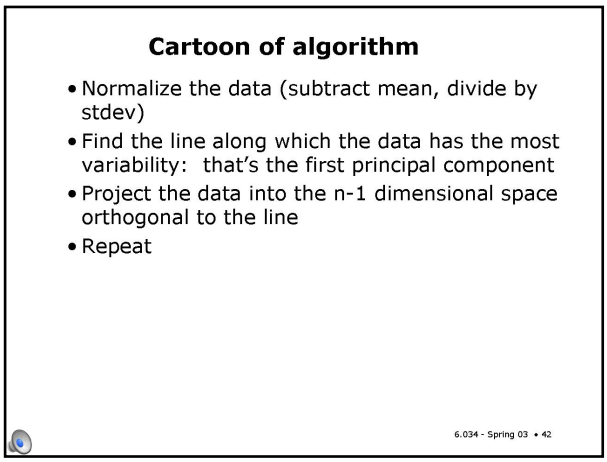


Slide 4.6.40
To really understand what's going on in this algorithm, you need to have had linear algebra. We'll just give you a "cartoon" idea of how it works.

We start out by normalizing the data (subtracting the mean and dividing by the standard deviation). The new set of coordinates we construct will have its origin at the centroid of the data.

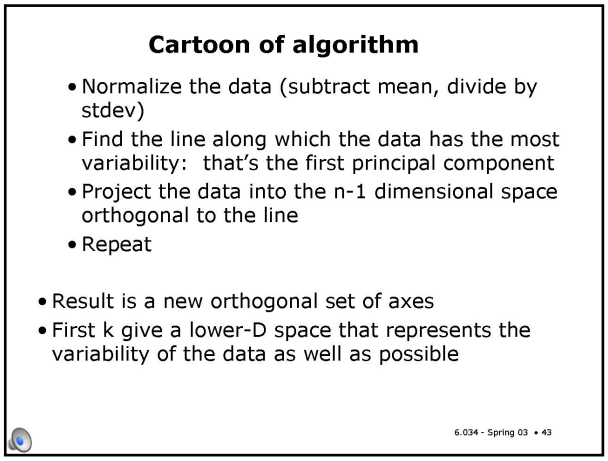
Slide 4.6.41
Now, we find the single line along which the data have the most variance. It's the dimension that, were we to project the data onto it, would result in the most "spread" of the data. We'll let this be our first principal component.

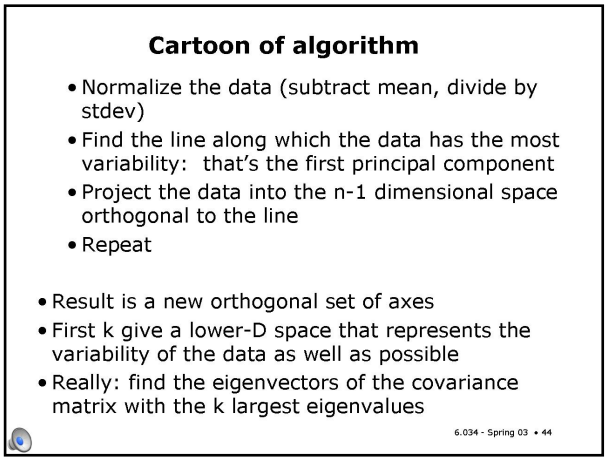




Slide 4.6.42
Now, we project the data down into the n-1 dimensional space that's orthogonal to the line we just chose, and repeat.

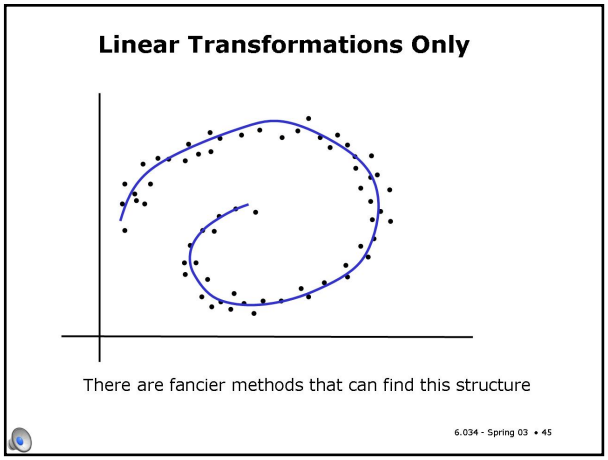
Slide 4.6.43
The result of this process is a new set of orthogonal axes. The first k of them give a lower-dimensional space that represents the variability of the data as well as possible.

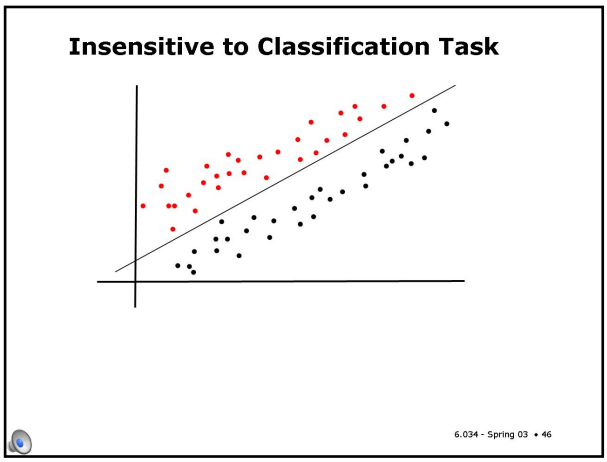




Slide 4.6.44
If you have some experience with linear algebra, then I can tell you that what we really do is find the eigenvectors of the covariance matrix with the k largest eigenvalues.

Slide 4.6.45
One problem with PCA (as it's called by its friends) is that it can only produce a set of coordinates that's a linear transformation of the originals. But here's a data set that seems to have a fundamentally one-dimensional structure. Unfortunately, we can't express its axis as a linear combination of the original ones. There are some other cool dimensionality reduction techniques that can actually find this structure!

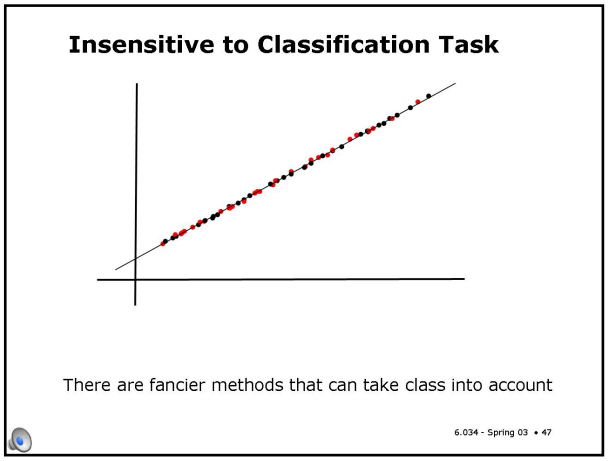


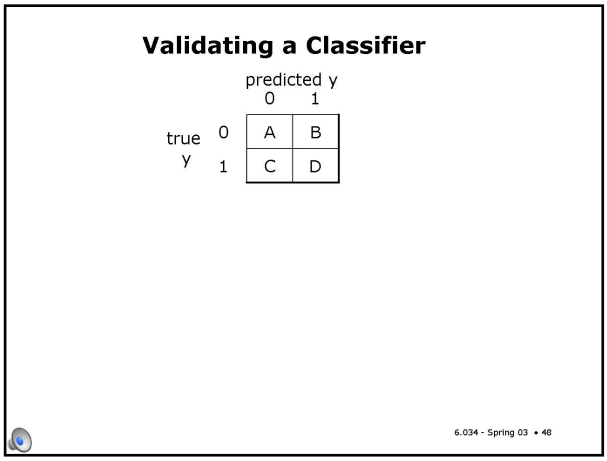


Slide 4.6.46
Another problem with PCA is that it (like k-Means clustering) ignores the classes of the points. So, in this example, the principal component is the line that goes between the two classes (it's a great separator, but that's not what we're looking for right now).

Slide 4.6.47
Now, if we project the data onto that line (which is what would happen if we wanted to reduce the dimensionality of our data set to 1), the positive and negative points are completely intermingled, and we can never get a separator.

There are dimensionality-reduction techniques, also, sadly beyond our scope, that try to optimize the discriminability of the data rather than its variability, which don't suffer from this problem.

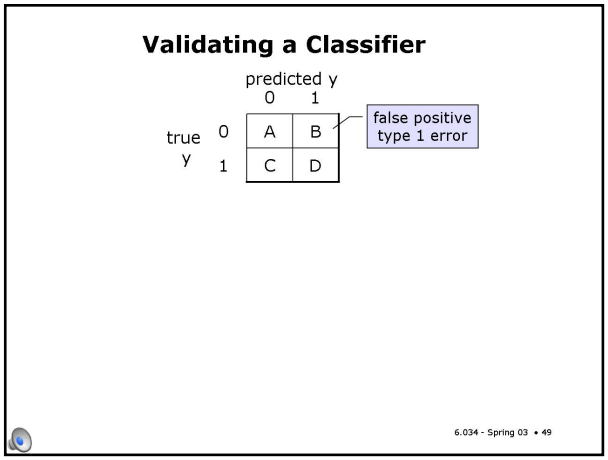


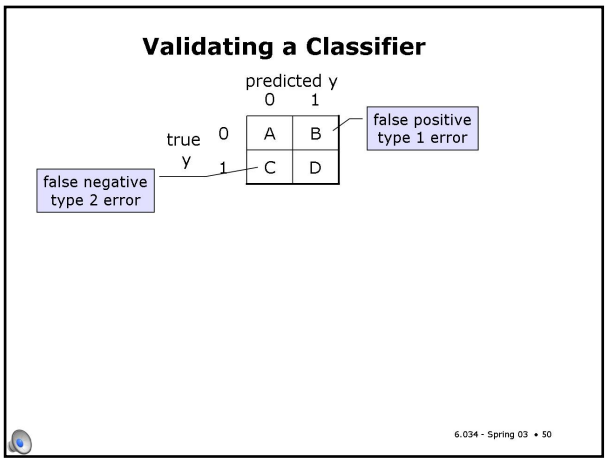


Slide 4.6.48
We're just going to tack one additional topic onto the end of this section. It has to do with understanding how well a classifier works. So far, we've been thinking about optimizing training error or cross-validation error, where "error" is measured as the number of examples we get wrong. Let's examine this a little more carefully.

In a binary classification problem, on a single example, there are 4 possible outcomes, depending on the true output value for the input and the predicted output value. In this table, we'll assign values A through D to be the number of times each of these outcomes happens on a data set.

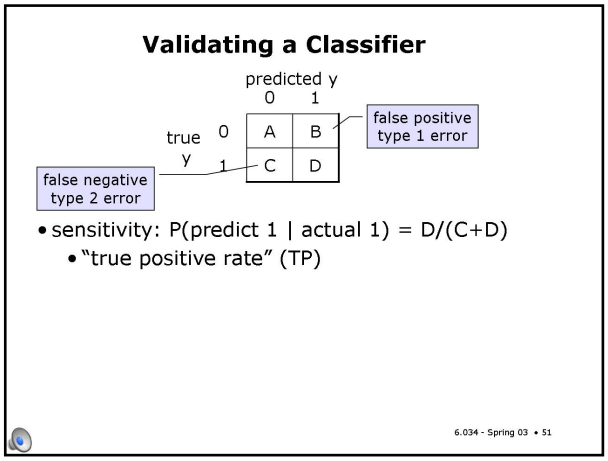
Slide 4.6.49
Case B, in which the answer was supposed to be 0 but the classifier predicted 1 is called a "false positive" or a type 1 error.

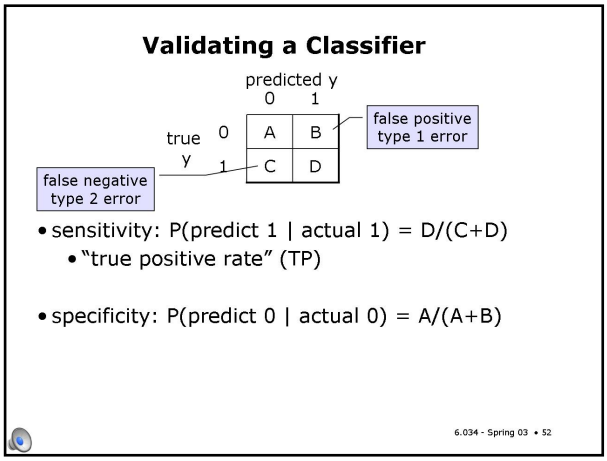




Slide 4.6.50
Case C, in which the answer was supposed to be 1 but the classifier predicted 0 is called a "false negative" or a type 2 error.

Slide 4.6.51
Given these 4 numbers, we can define different characterizations of the classifier's performance. The **sensitivity** is the probability of predicting a 1 when the actual output is 1. This is also called the **true positive** rate, or TP.

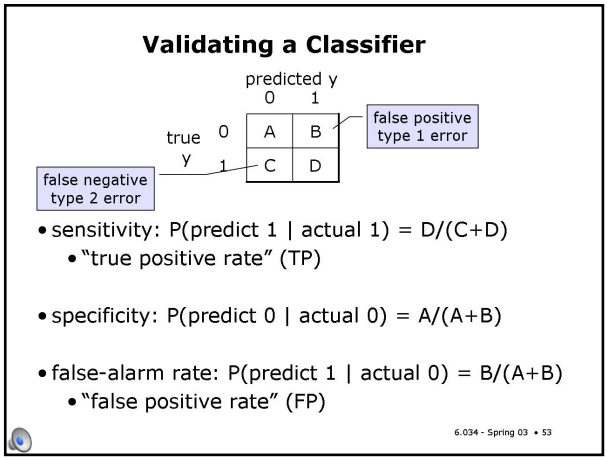


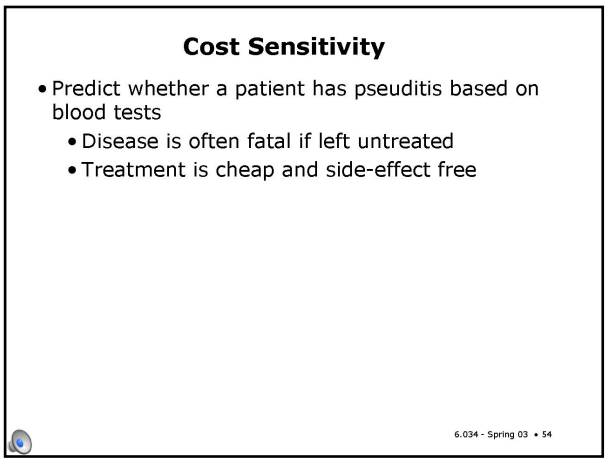


Slide 4.6.52
The **specificity** is the probability of predicting a 0 when the actual output is 0.

Slide 4.6.53
The **false-alarm rate** is the probability of predicting a 1 when the actual output is 0. This is also called the **false positive** rate, or FP.

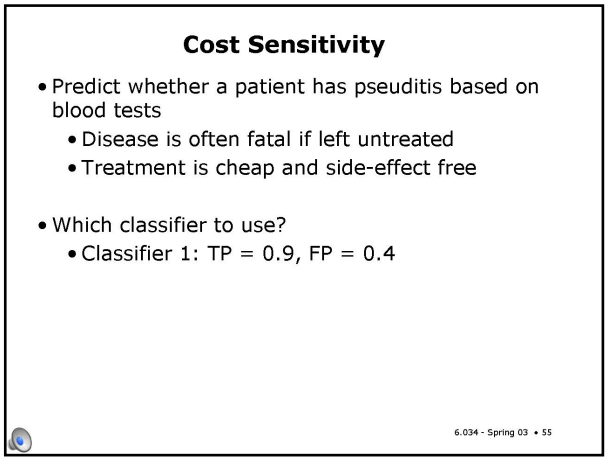
Classifiers are usually characterized using sensitivity and specificity, or using TP and FP.

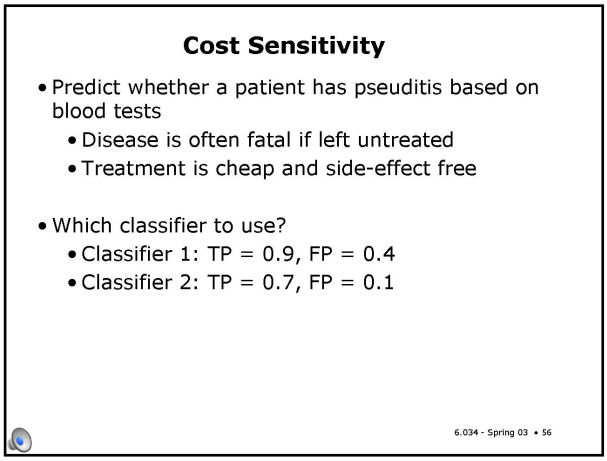




Slide 4.6.54
Imagine that you're a physician and you need to predict whether a patient has pseuditis based on the results of some blood tests. The disease is often fatal if it's left untreated, and the treatment is cheap and relatively side-effect free.

Slide 4.6.55
You have two different classifiers that you could use to make the decision. The first has a true-positive rate of 0.9 and a false-positive rate of 0.4. That means that it will diagnose the disease in 90 percent of the people who actually have it; and also diagnose it in 40 percent of people who don't have it.



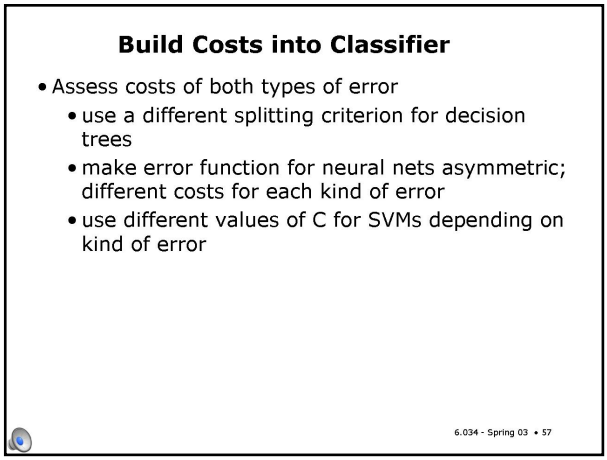


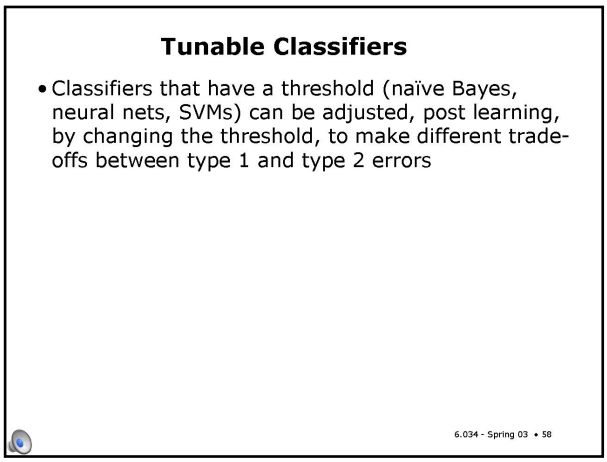
Slide 4.6.56
The second classifier only has a true-positive rate of 0.7, but a more reasonable false positive rate of 0.1.

Given the set-up of the problem, we might choose classifier 1, since all those false positives aren't too costly (but if it causes too much hassle, per patient, we might not want to bring 40 percent of them back for treatment).

Slide 4.6.57
One way to address this problem is to start by figuring out the relative costs of the two types of errors. Then, for many classifiers, we can build these costs directly into the choice of classification.

In decision trees, we could use a different splitting criterion. For neural networks we could change the error function to be asymmetric. In SVM's, we could use two different values of C.



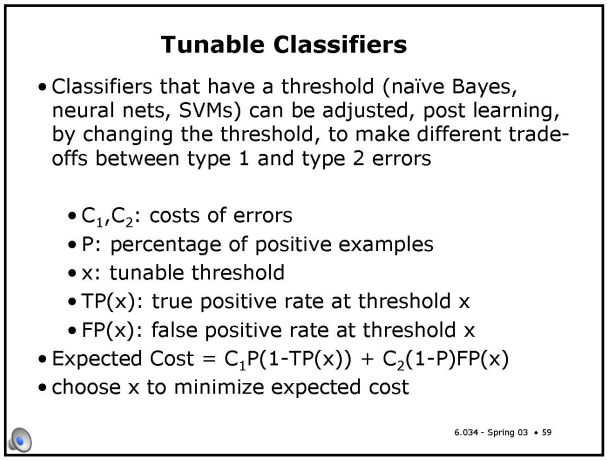


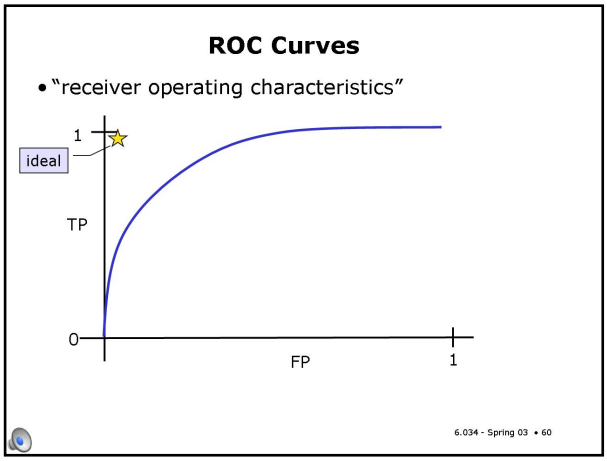
Slide 4.6.58
Often it's useful to deliver a classifier that is tunable. That is, a classifier that has a parameter in it that can be used, at application time, to change the trade-offs made between type 1 and type 2 errors. Most classifers that have a threshold (such as naive Bayes, neural nets, or SVMs), can be tuned by changing the threshold. At different values of the threshold the classifier will tend to make more errors of one type versus the other.

Slide 4.6.59
In a particular application, we can choose a threshold as follows.

Let c1 and c2 be the costs of the two different types of errors; let p be the percentage of positive examples, let x be the threshold parameter that we are allowed to tune, and let TP(x) and FP(x) be the true-positive and false-positive rates, respectively, of the classifier when the threshold is set to have value x.

Then, we can characterize the average, or expected, cost based on this formula, as a function of x. We should choose the value of x that will minimize expected cost.

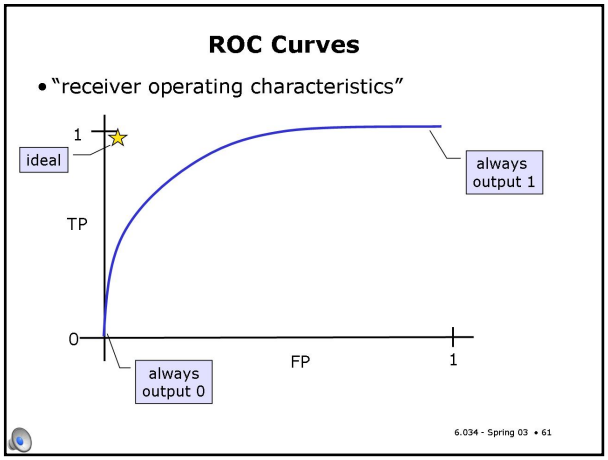


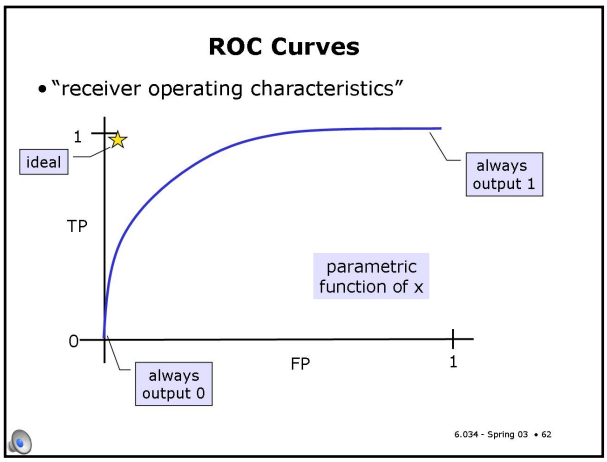


Slide 4.6.60
One way to see the overall performance of a tunable classifier is with a ROC curve. ROC stands for "receiver operating characteristics" from the days of the invention of radar.

An ROC curve is plotted on two axes; the x axis is the false positive rate and the y axis is the true positive rate. In an ideal world, we would have a false positive rate of 0 and a true positive rate of 1, which would put our performance up near the star on this graph.

Slide 4.6.61
In reality, as we adjust the parameter in the classifier, we typically go from a situation in which the classifier always outputs 0, which generates no false positives and no true positives, to a situation in which the classifier always outputs 1, in which case we have both false positive and true positive rates of 1.

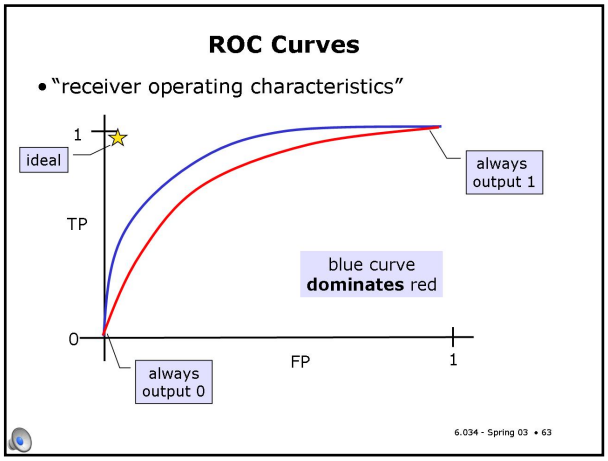


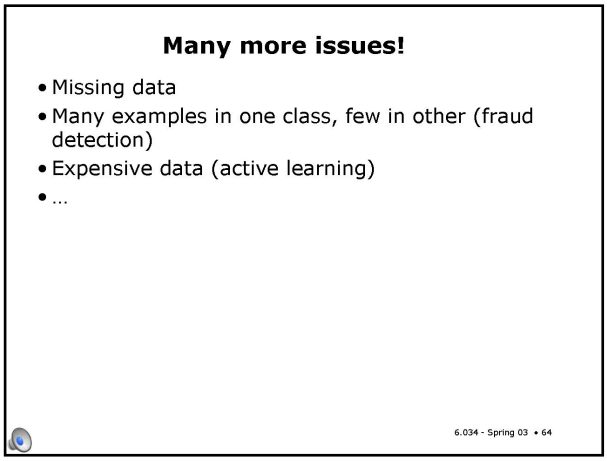


Slide 4.6.62
The ROC curve itself is a parametric curve; for each value of x, we plot the pair FP(x), TP(x). The curve shows the range of possible behaviors of the classifier. It is typically shaped something like this blue curve; the higher the false positive rate we can stand, the higher the rate of detecting true positives we can achieve.

Slide 4.6.63
Often it is useful to compare two different classifiers by comparing their ROC curves. If we're lucky, then one curve is always higher than the other. In such a situation, we'd say that the blue curve **dominates** the red curve. That means that, no matter what costs apply in our domain, it will be better to use the blue classifier (because, for any fixed rate of false positives, the blue classifier can achieve more true positives; or for any fixed rate of true positives, the blue classifier can always achieve fewer false positives).

If the curves cross, then it will be better to use one classifier in some cost situations and the other classifier in other situations.





Slide 4.6.64
Machine learning is a huge field that we have just begun to cover. Even in the context of supervised learning, there are a variety of other issues, including how to handle missing data, what to do when you have very many negative examples and just a few positives (such as when you're trying to detect fraud), what to do when getting y values for your x's is very expensive (you might actively choose which y's you'd like to have labeled), and many others.

If you like this topic, take a probability course, and then take the graduate machine learning course.