The techniques we'll develop in this chapter include: a better choice of cost function, known as the cross-entropy cost function; four so-called "regularization" methods (L1 and L2 regularization, dropout, and artificial expansion of the training data), which make our networks better at generalizing beyond the training data; a better method for initializing the weights in the network; and a set of heuristics to help choose good hyper-parameters for the network.

we often learn fastest when we're badly wrong about something.

But we've just seen that our artificial neuron has a lot of difficulty learning when it's badly wrong - far more difficulty than when it's just a little wrong

We can see from this graph that when the neuron's output is close to 1

, the curve gets very flat, and so σ′(z)

gets very small. Equations (55) and (56) then tell us that ∂C/∂w and ∂C/∂b

get very small. This is the origin of the learning slowdown

It turns out that we can solve the problem by replacing the quadratic cost with a different cost function, known as the cross-entropy

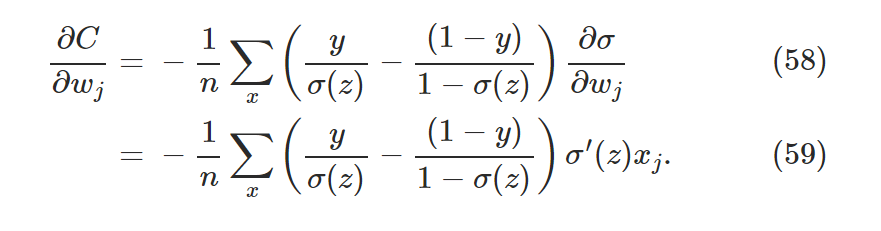
The output from the neuron is, of course, a=σ(z)a=σ(z), where z=∑jwjxj+bz=∑jwjxj+b is the weighted sum of the inputs. We define the cross-entropy cost function for this neuron by

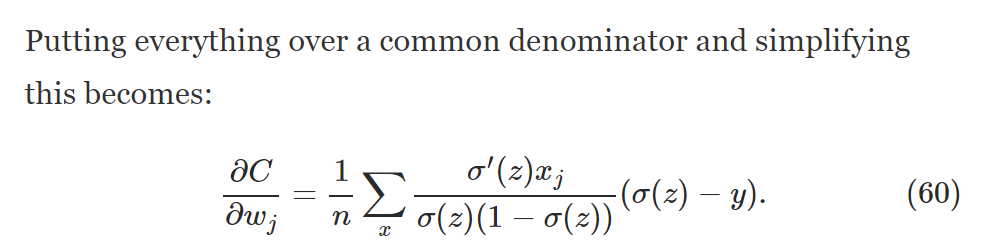
C=−1n∑x[ylna+(1−y)ln(1−a)],(57)

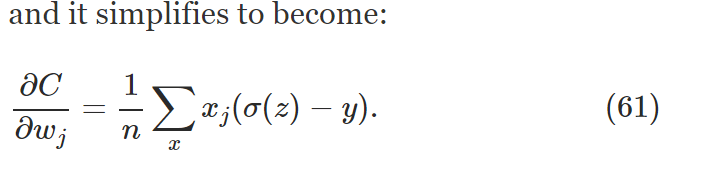
Two properties in particular make it reasonable to interpret the cross-entropy as a cost function. First, it's non-negative, that is, C>0C>0. To see this, notice that: (a) all the individual terms in the sum in (57) are negative, since both logarithms are of numbers in the range 00 to 11; and (b) there is a minus sign out the front of the sum. Second, it approaches zero when the NN do good job

 But the cross-entropy cost function has the benefit that, unlike the quadratic cost, it avoids the problem of learning slowing down

To see this, let's compute the partial derivative of the cross-entropy cost with respect to the weights. We substitute a=σ(z)a=σ(z) into (57), and apply the chain rule twice, obtaining:







 It tells us that the rate at which the weight learns is controlled by σ(z)−yσ(z)−y, i.e., by the error in the output. The larger the error, the faster the neuron will learn

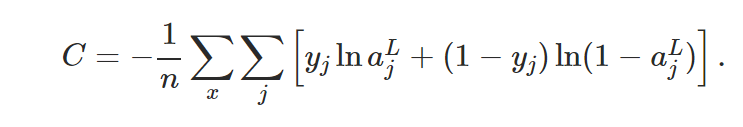
This cancellation (the simoid prime)is the special miracle ensured by the cross-entropy cost function. Actually, it's not really a miracle. As we'll see later, the cross-entropy was specially chosen to have just this property.

It's that steepness which the cross-entropy buys us, preventing us from getting stuck just when we'd expect our neuron to learn fastest, i.e., when the neuron starts out badly wrong.

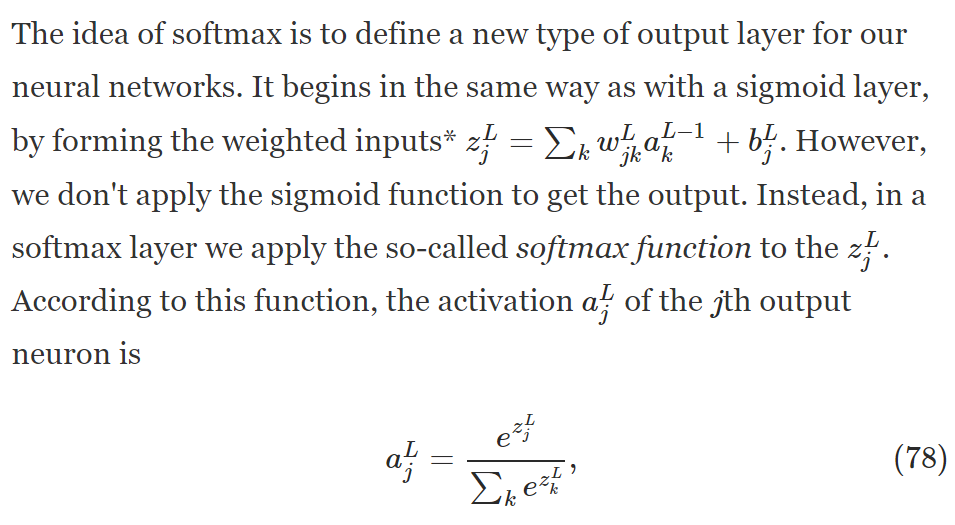
In fact, with the change in cost function it's not possible to say precisely what it means to use the "same" learning rate

Who cares how fast the neuron learns, when our choice of learning rate was arbitrary to begin with?! That objection misses the point. The point of the graphs isn't about the absolute speed of learning. It's about how the speed of learning changes.

In particular, when we use the quadratic cost learning is *slower* when the neuron is unambiguously wrong than it is later on, as the neuron gets closer to the correct output; while with the cross-entropy learning is faster when the neuron is unambiguously wrong. Those statements don't depend on how the learning rate is set.



more important reason is that neuron saturation is an important problem in neural nets, a problem we'll return to repeatedly throughout the book. And so I've discussed the cross-entropy at length because it's a good laboratory to begin understanding neuron saturation and how it may be addressed.



Equation (78) also implies that the output activations are all positive, since the exponential function is positive. Combining this with the observation in the last paragraph, we see that the output from the softmax layer is a set of positive numbers which sum up to 11. In other words, the output from the softmax layer can be thought of as a probability distribution.

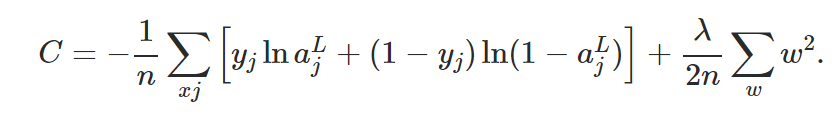
Even if such a model agrees well with the available data, that doesn't make it a good model. It may just mean there's enough freedom in the model that it can describe almost any data set of the given size, without capturing any genuine insights into the underlying phenomenon. When that happens the model will work well for the existing data, but will fail to generalize to new situations. The true test of a model is its ability to make predictions in situations it hasn't been exposed to before.

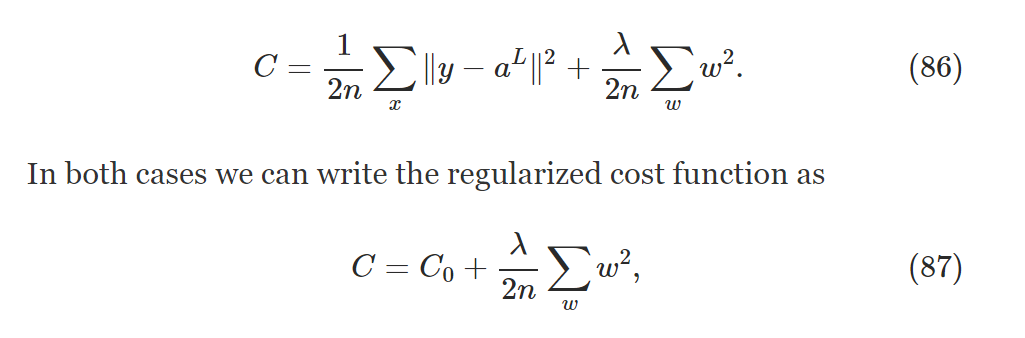
Overfitting is a major problem in neural networks. This is especially true in modern networks, which often have very large numbers of weights and biases. To train effectively, we need a way of detecting when overfitting is going on, so we don't overtrain. And we'd like to have techniques for reducing the effects of overfitting.

Overfitting is a major problem in neural networks. This is especially true in modern networks, which often have very large numbers of weights and biases. To train effectively, we need a way of detecting when overfitting is going on, so we don't overtrain. And we'd like to have techniques for reducing the effects of overfitting.

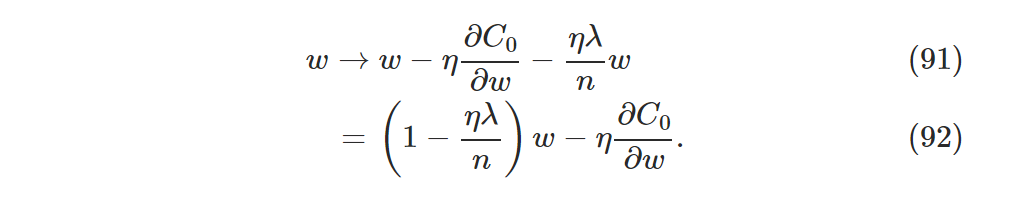
why we're using the validation\_data rather than the test\_data to set good hyper-parameters? To understand why, consider that when setting hyper-parameters we're likely to try many different choices for the hyper-parameters. If we set the hyper-parameters based on evaluations of the test\_data it's possible we'll end up overfitting our hyper-parameters to the test\_data. That is, we may end up finding hyper-parameters which fit particular peculiarities of the test\_data, but where the performance of the network won't generalize to other data sets. We guard against that by figuring out the hyper-parameters using the validation\_data. Then, once we've got the hyper-parameters we want, we do a final evaluation of accuracy using the test\_data. That gives us confidence that our results on the test\_data are a true measure of how well our neural network generalizes. To put it another way, you can think of the validation data as a type of training data that helps us learn good hyper-parameters. This approach to finding good hyper-parameters is sometimes known as the *hold out* method, since the validation\_data is kept apart or "held out" from the training\_data

technique sometimes known as *weight decay* or *L2 regularization*. The idea of L2 regularization is to add an extra term to the cost function, a term called the *regularization term*. Here's the regularized cross-entropy

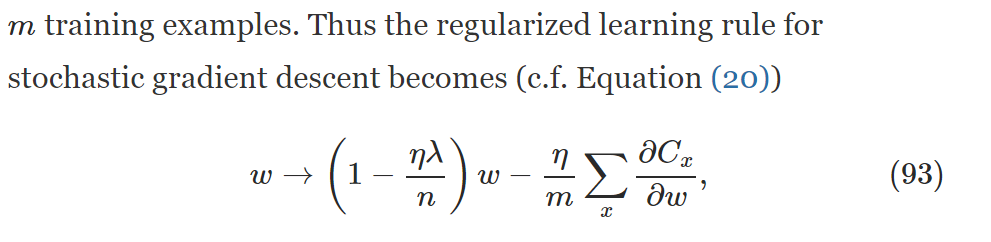




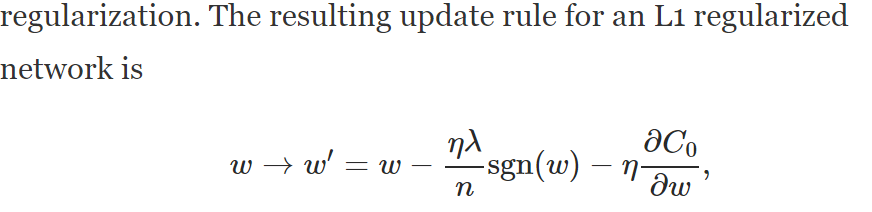
Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowed if they considerably improve the first part of the cost function. Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function. The relative importance of the two elements of the compromise depends on the value of λλ: when λλ is small we prefer to minimize the original cost function, but when λλ is large we prefer small weights.



This is exactly the same as the usual gradient descent learning rule, except we first rescale the weight ww by a factor 1−ηλn1−ηλn. This rescaling is sometimes referred to as *weight decay*, since it makes the weights smaller. At first glance it looks as though this means the weights are being driven unstoppably toward zero. But that's not right, since the other term may lead the weights to increase, if so doing causes a decrease in the unregularized cost function



Other regularization techniques: L1, dropout and, bigger training set



In particular, suppose we have a training input xx and corresponding desired output yy. Ordinarily, we'd train by forward-propagating xx through the network, and then backpropagating to determine the contribution to the gradient. With dropout, this process is modified. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. After doing this, we'll end up with a network along the following lines. Note that the dropout neurons, i.e., the neurons which have been temporarily deleted, are still ghosted in:

We forward-propagate the input xx through the modified network, and then backpropagate the result, also through the modified network. After doing this over a mini-batch of examples, we update the appropriate weights and biases. We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network.

What's this got to do with dropout? Heuristically, when we dropout different sets of neurons, it's rather like we're training different neural networks. And so the dropout procedure is like averaging the effects of a very large number of different networks. The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting.

A related heuristic explanation for dropout is given in one of the earliest papers to use the technique"This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons." In other words, if we think of our network as a model which is making predictions, then we can think of dropout as a way of making sure that the model is robust to the loss of any individual piece of evidence. In this, it's somewhat similar to L1 and L2 regularization, which tend to reduce weights, and thus make the network more robust to losing any individual connection in the network.

In particular, we can see from this graph that it's quite likely that |z||z| will be pretty large, i.e., either z≫1z≫1 or z≪−1z≪−1. If that's the case then the output σ(z)σ(z) from the hidden neuron will be very close to either 11 or 00. That means our hidden neuron will have saturated

if the weights in later hidden layers are initialized using normalized Gaussians, then activations will often be very close to 00 or 11, and learning will proceed very slowly.

Suppose we have a neuron with ninnin input weights. Then we shall initialize those weights as Gaussian random variables with mean 00 and standard deviation 1/nin−−−√1/nin. That is, we'll squash the Gaussians down, making it less likely that our neuron will saturate

Such a neuron is much less likely to saturate, and correspondingly much less likely to have problems with a learning slowdown.

In fact, it doesn't much matter how we initialize the biases, provided we avoid the problem with saturation. Some people go so far as to initialize all the biases to 00, and rely on gradient descent to learn appropriate biases. But since it's unlikely to make much difference, we'll continue with the same initialization procedure as before.

**Use early stopping to determine the number of training epochs:** As we discussed earlier in the chapter, early stopping means that at the end of each epoch we should compute the classification accuracy on the validation data. When that stops improving, terminate. This makes setting the number of epochs very simple. In particular, it means that we don't need to worry about explicitly figuring out how the number of epochs depends on the other hyper-parameters. Instead, that's taken care of automatically. Furthermore, early stopping also automatically prevents us from overfitting. This is, of course, a good thing, although in the early stages of experimentation it can be helpful to turn off early stopping, so you can see any signs of overfitting, and use it to inform your approach to regularization

**Learning rate schedule**

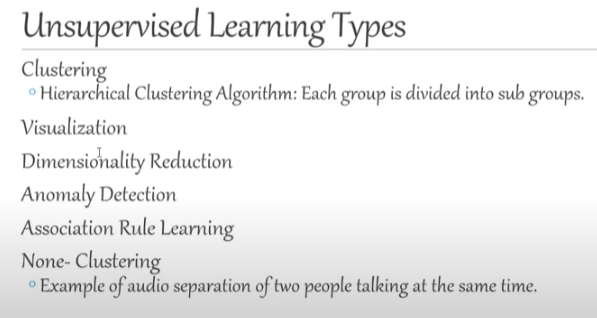
**Automated techniques:** I've been describing these heuristics as though you're optimizing your hyper-parameters by hand. Hand-optimization is a good way to build up a feel for how neural networks behave. However, and unsurprisingly, a great deal of work has been done on automating the process. A common technique is *grid search*, which systematically searches through a grid in hyper-parameter space. A review of both the achievements and the limitations of grid search (with suggestions for easily-implemented alternatives) may be found in a 2012 paper. Many more sophisticated approaches have also been proposed. I won't review all that work here, but do want to mention a particularly promising 2012 paper which used a Bayesian approach to automatically optimize hyper-parameters. The code from the paper is [publicly available](https://github.com/jaberg/hyperopt), and has been used with some success by other researchers.

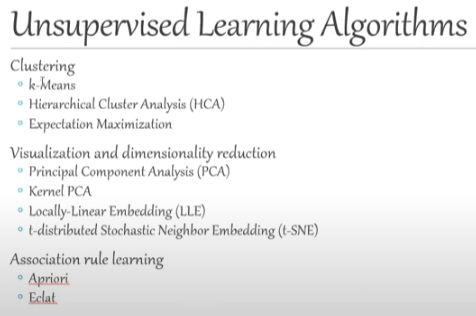
Lecture Notes

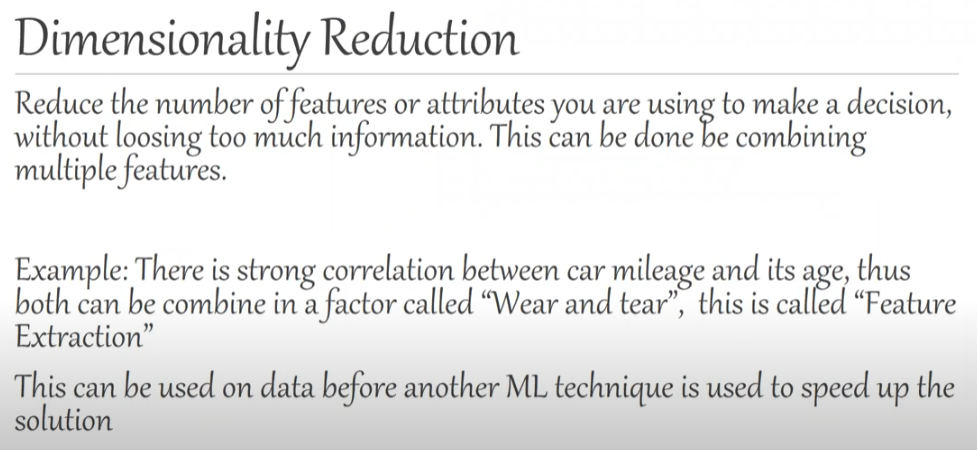
Supervised Learning types: classification, prediction

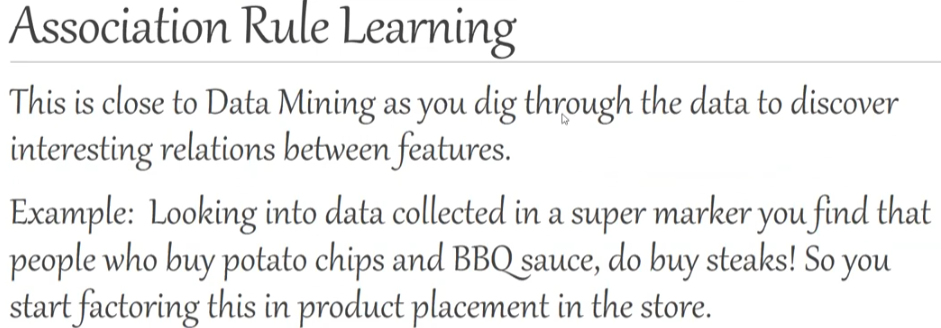
Supervised Learning Algorithms:

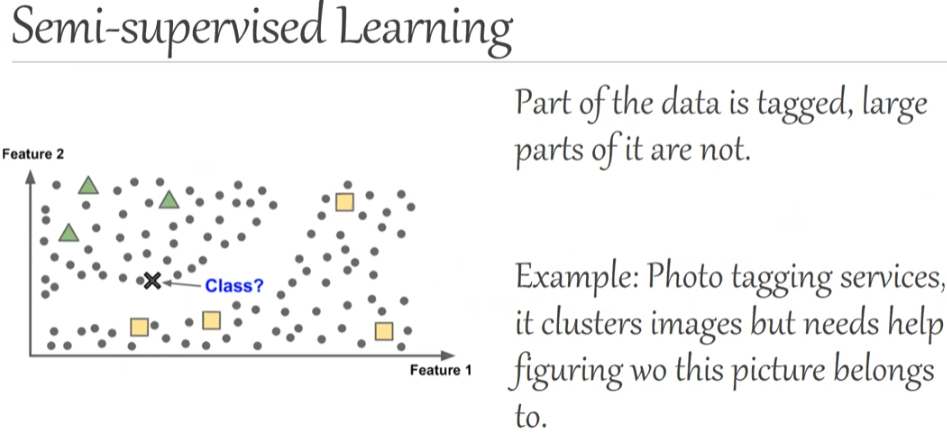
* K-nearest neighbors
* Linear regression
* Logistic regression
* SVM
* Decision trees and random forests
* Neural networks

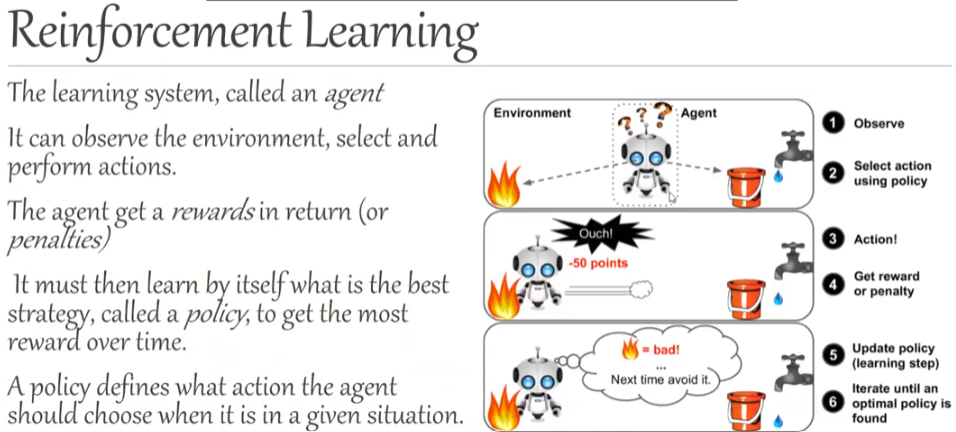




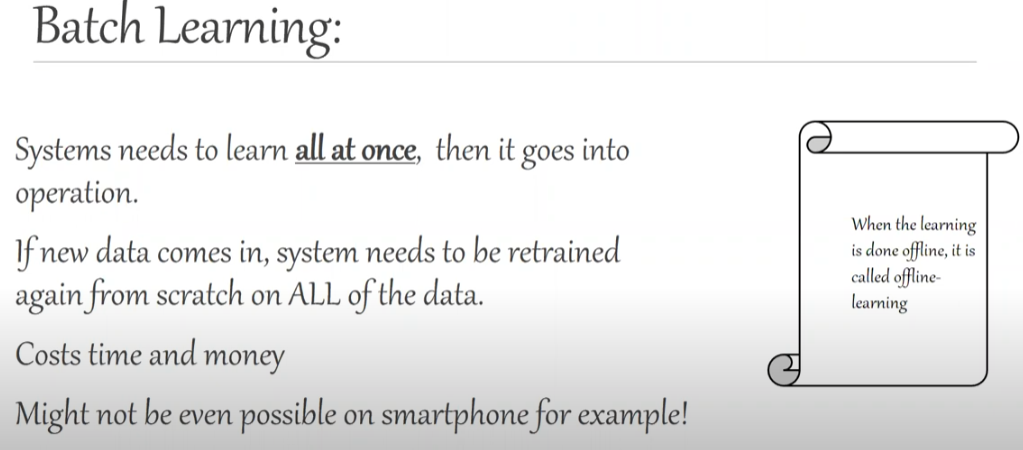


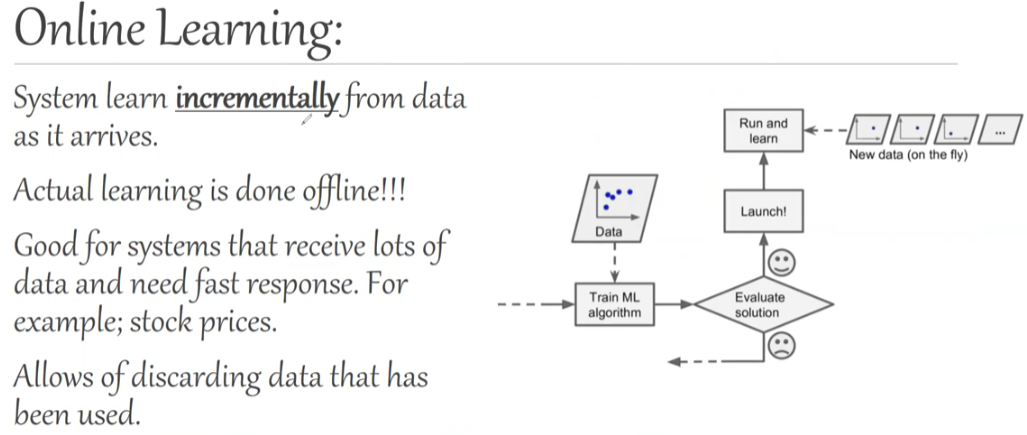




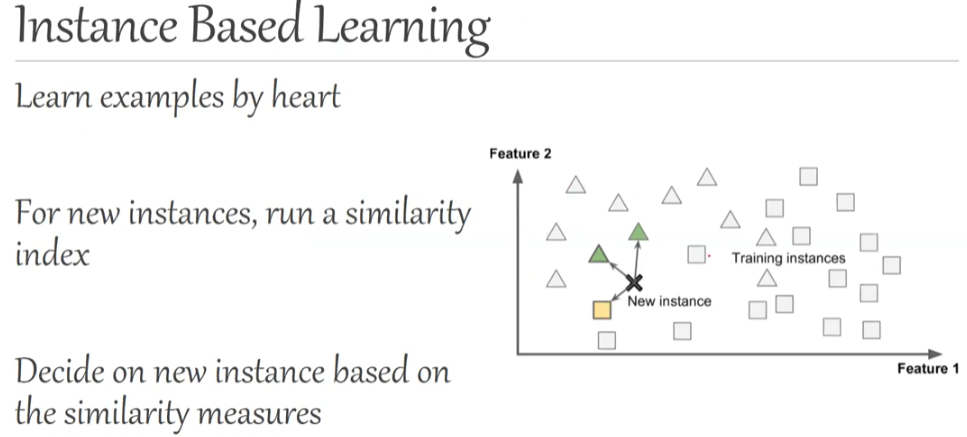


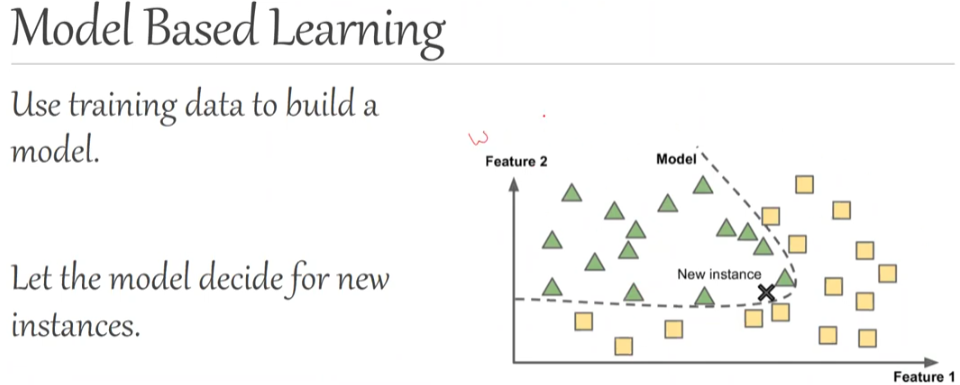
types of ML based on ability to learn while operation





Types of ML based on how they generalize:





Limited data should not be biased (“sampling bias”)

