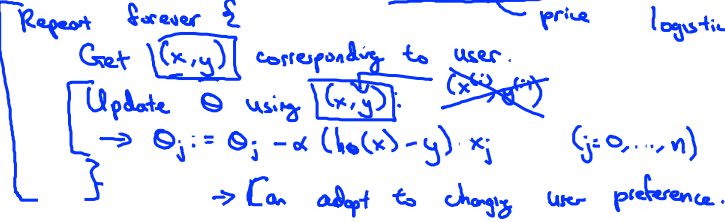
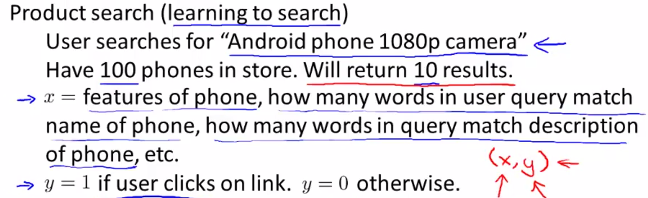
***Advanced Topics***

**I. Online Learning**

* New large-scale ML setting = the **online learning setting**, which allows us to model problems where we have a continuous flood/stream of data coming in + we’d like an algorithm to learn from that.
* Today, many of the largest website companies use different versions of online learning algorithms to learn from floods of users that keep coming back to the website.
* Specifically, if you have a continuous stream of data generated by a continuous stream of users coming to a website, you can sometimes use an online learning algorithm to learn user preferences from the stream use that to optimize decisions on your site.
* Suppose you run a shipping service site where users repeatedly come + tell you where they want to send a package from, + where they want to send it to (origin + destination)
* Your site offers to ship packages for some asking price + based on the price, users either choose to use your shipping service (positive example) + or to not purchase your shipping service (negative)
* We want a learning algorithm to help optimize the asking price to offer to our users
* Specifically, come up w/ features X that capture properties of users (demographics, origins, destinations, price) to learn the probability a user will elect to ship the package using our service
* Features X captures the price we're asking for



* If we could estimate the chance they'll agree to use our service for any given price, we can try to pick a price so they have a pretty high probability of choosing our site while simultaneously hopefully offering us a fair return/profit for shipping their package.
* If we can learn this P(y = 1) given any price + the other features -- P(y = 1 | X; Ө) – we could use this to choose appropriate prices as new users come to us.
* In order to model P(y=1), can use logistic regression, NN’s, or some other algorithm like
* Start w/ logistic regression
* If you have a site that runs *continuously*, here's what an online learning algorithm would do:
* 
* A user comes to the site + for that user, we'll get some (x, y) pair corresponding to them
* The features X are: origin + destination specified by user + the price offered to them this time around, + Y is either 1 or 0 depending whether or not they chose to use our service
* The online learning algorithm does updates the parameters Ө using *just this example* (x, y)
* In particular, update parameters Ө as: **Өj := Өj – α \* gradient descent rule for logistic regression**
* Do this update for j = 0-n
* So, for other learning algorithms, instead of writing (X, Y), we had things like (X(i), Y(i)), but in this online learning setting we’re discarding the notion of a *fixed* training set
* Instead we have an algorithm + as we get an example, we learn using that example, + then *we throw that example away* + *never use it again*
* Look at 1 example at a time 🡪 learn from that example 🡪 discard it.
* We're also doing away w/ the notion of there being a fixed training set indexed by i
* If you run a major site w/ a continuous stream of users, this sort of online learning algorithm is a pretty reasonable algorithm b/c the data is essentially free
* And if you have *so much* data (essentially unlimited), there’s really no need to look at a training example more than once.
* W/ only a small number of users, rather than using an online learning algorithm, might be better off saving away all data in a fixed training set + running some algorithm over that training set.
* 1 interesting effect of this sort of online learning algorithm is that it can adapt to changing user preferences.
* If, over time, maybe users become more price sensitive + less-willing to pay high prices, become less price sensitive + more willing to pay higher prices, different things become more important to users, or you start to have new types of users coming to your site, this sort of online learning algorithm can also adapt to these changing user preferences
* It can keep track of what changing populations of users may be willing to pay b/c if the pool of users changes, updates to parameters Ө will slowly adapt to whatever the latest pool of users looks like
* Another example of an application to of online learning 🡺 product search in which we want to apply learning algorithms to learn to give good search listings to a user.
* Online store that sells phones w/ a UI where a user can come to the site + type in a query
* Suppose we have 100 phones + when a user types in a query, our site would to find a choice of 10 different phones to offer to the user.
* Want a learning algorithm to help us figure out what are the 10 phones out of the 100 to return the user in response to a search query
* For each phone + specific user query, construct a feature vector X (captures different properties of the phone + capture things like how similar a query is to the phones (same words in phone name or description)
* Want to estimate probability a user will click on a link for a specific phone b/c we want to users phones they have high probability of clicking on/are likely to want to buy
* Define y = 1 if user clicks on a link for a phone + y = 0 otherwise
* Want to learn the probability a user will click on a specific phone given features X (properties of the phone + how well the query matches the phone)
* This learning problem is called the problem of learning **predicted click-through rate**, or **predicted CTR** 🡪 learning the probability a user will click on the specific link you offer them
* If you can estimate the predicted CTR for any particular phone, we can use this to show a user the 10 phones they’re most likely to click on
* Out of 100 phones, we can compute P(y = 1 | x; Ө) for each + select the 10 phones a user is most likely to click on
* This will be a pretty reasonable way to decide what 10 results to show to the user.
* Just to be clear, every time a user does a search, we return 10 results 🡪 actually give us 10 (x, y) pairs/10 training examples every time a user comes to our site
* This is b/c for each of the 10 phone we chose to show a user, we get a feature vector X + also get a value for Y/observe the value of Y
* 1 way to run a site like this would be to continuously show a user your 10 best guesses for what other phones they might like
* So, each time a user comes, you get 10 examples/(X, Y)pairs 🡪 use online learning algorithm to update parameters using 10 steps of gradient descent on these 10 examples 🡪 throw the data away
* If you have a continuous stream of users, this would be a pretty reasonable way to learn parameters for your algorithm so as to show 10 phones to users they’re the most likely to click on

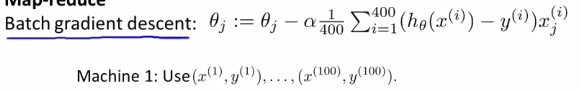


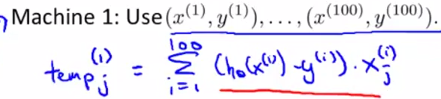


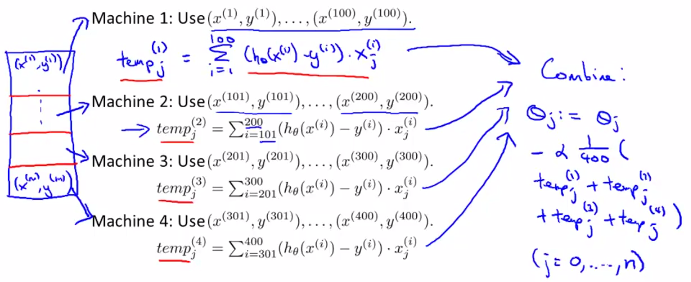
* Another example = trying to decide on a special offer to show a user or you show different users different news articles on a news aggregator site, you can use a similar system to show users news articles they’re most likely to be interested in/offers they’re most likely to click on.
* Can even imagine a collaborative filtering system giving you additional features to feed into a logistic regression classifier to try to predict CTR for different products you might recommend to a user
* Any of these problems could have been formulated as a standard ML problem (fixed training set saved after running the site for a few days)
* But these are the actual sorts of problems done in large companies that get so much data so there’s really no need to save away a fixed training set,
* Instead you can use an online learning algorithm to just learn continuously from data users are generating on your site.
* This algorithm is really very similar to Stochastic GD, only instead of scanning through a fixed training set, we get 1 example from a user, learning from that example, then discard it + move on.
* If you have a continuous stream of data for some application, this sort of algorithm may be well worth considering.
* An advantage of online learning = if you have a changing pool of users or if the things you're trying to predict are slowly changing (user tastes, i.e. P(Y | X; Ө), the online learning algorithm can slowly adapt your learned hypothesis to whatever the latest sets of user behaviors are like as well.

**MapReduce and Data Parallelism**

* Some ML problems are just too big to run on 1 machine
* Sometimes you just have so much data you don't ever want to run of it through a single CPU, no matter the algorithm
* A different approach to large-scale ML = **MapReduce**
* Many people will say MapReduce is at least an equally important (some say more) compared to stochastic GD, but it's only relatively simpler to explain
* But using these ideas, you might be able to scale learning algorithms to even far larger problems than possible using stochastic GD.
* Say we want to fit a linear regression or a logistic regression model 🡪 start w/ batch GD + assume m = 400 examples (in terms of large-scale ML, you this is small so this might be more commonly applied to problems where you have maybe closer to 400Mexamples)



* Batch GD learning sums from i = 1-400 🡪 If m is large, this is a computationally expensive step.
* **MapReduce** is due to 2 researchers, Jeff Dean (1 of the most legendary engineers in all of Silicon Valley + built a large fraction of the architectural infrastructure that all of Google runs on today) + Sanjay Gimawat.
* MapReduce idea.
* Have some training set, {(X(i), Y(i)), … (X(m), Y(m))} + split this in to 4 different subsets
* Assuming we have 4 CPUs/machines to run **in parallel** on this training set
* The 1st of my 4 CPUs uses just the 1st ¼ of my training set (1st 100 training examples) + computes the summation for just those examples.
* 
* Then take the 2nd ¼ of my data + send it to my 2nd CPU which will use training examples 101-200
* Each machine will sum over 100 examples instead of 400 🡪 only has to do a ¼ of the work + thus presumably could do it about 4X as fast.
* After all these machines have done this work, take their temp\_j variables + put them back together
* i.e. Take these variables + send them all to a centralized master server that combine these results together
* In particular, it will update my parameters Өj according to Өj := Өj – α\*)1/400)\*(temp\_J(1) + temp\_J(2) + temp\_J(3) + temp\_J(4))



* Have to do this separately for temp\_j(0)
* What the batch GD equation is doing is exactly the same as a centralized master server that takes the results 4 CPU’s temp\_j results + adds them up
* It’s exactly equivalent to the batch GD algorithm, only instead of needing to sum over all 400 hundred training examples on just 1 machine, instead divide up the workload onto 4 machines.
* So, here's what the general picture of the MapReduce technique looks like. We have some training sets, + if we want to paralyze across four machines, we are going to take the training set + split it, you know, equally. Split it as evenly as we can into four subsets. Then we are going to take the 4 subsets of the training data + send them to 4 different computers. + each of the 4 computers can compute a summation over just one quarter of the training set, + then finally take each of the computers takes the results, sends them to a centralized server, which then combines the results together. So, on the previous line in that example, the bulk of the work in GD, was computing the sum from i equals 1 to 400 of something. So more generally, sum from i equals 1 to m of that formula for GD. + now, b/c each of the four computers can do just a quarter of the work, potentially you can get up to a 4x speed up. In particular, if there were no network latencies + no costs of the network communications to send the data back + forth, you can potentially get up to a 4x speed up. Of course, in practice, b/c of network latencies, the overhead of combining the results afterwards + other factors, in practice you get slightly less than a 4x speedup. But, none the less, this sort of macro juice approach does offer us a way to process much larger data sets than is possible using a single computer. If you are thinking of applying MapReduce to some learning algorithm, in order to speed this up. By paralleling the computation over different computers, the key question to ask yourself is, can your learning algorithm be expressed as a summation over the training set? + it turns out that many learning algorithms can actually be expressed as computing sums of functions over the training set + the computational expense of running them on large data sets is b/c they need to sum over a very large training set. So, whenever your learning algorithm can be expressed as a sum of the training set + whenever the bulk of the work of the learning algorithm can be expressed as the sum of the training set, then map reviews might a good candidate for scaling your learning algorithms through very, very good data sets. Lets just look at one more example. Let's say that we want to use one of the advanced optimization algorithm. So, things like, you know, l, b, f, g, s constant gradient + so on, + let's say we want to train a logistic regression of the algorithm. For that, we need to compute two main quantities. One is for the advanced optimization algorithms like, you know, LPF + constant gradient. We need to provide it a routine to compute the cost function of the optimization objective. + so for logistic regression, you remember that a cost function has this sort of sum over the training set, + so if youre paralizing over ten machines, you would split up the training set onto ten machines + have each of the ten machines compute the sum of this quantity over just one tenth of the training data. Then, the other thing that the advanced optimization algorithms need, is a routine to compute these partial derivative terms. Once again, these derivative terms, for which it's a logistic regression, can be expressed as a sum over the training set, + so once again, similar to our earlier example, you would have each machine compute that summation over just some small fraction of your training data. + finally, having computed all of these things, they could then send their results to a centralized server, which can then add up the partial sums. This corresponds to adding up those tenth i or tenth ij variables, which were computed locally on machine number i, + so the centralized server can sum these things up + get the overall cost function + get the overall partial derivative, which you can then pass through the advanced optimization algorithm. So, more broadly, by taking other learning algorithms + expressing them in sort of summation form or by expressing them in terms of computing sums of functions over the training set, you can use the MapReduce technique to parallelize other learning algorithms as well, + scale them to very large training sets. Finally, as one last comment, so far we have been discussing MapReduce algorithms as allowing you to parallelize over multiple computers, maybe multiple computers in a computer cluster or over multiple computers in the data center. It turns out that sometimes even if you have just a single computer, MapReduce can also be applicable. In particular, on many single computers now, you can have multiple processing cores. You can have multiple CPUs, + w/in each CPU you can have multiple proc cores. If you have a large training set, what you can do if, say, you have a computer w/ 4 computing cores, what you can do is, even on a single computer you can split the training sets into pieces + send the training set to different cores w/in a single box, like w/in a single desktop computer or a single server + use MapReduce this way to divvy up work load. Each of the cores can then carry out the sum over, say, one quarter of your training set, + then they can take the partial sums + combine them, in order to get the summation over the entire training set. The advantage of thinking about MapReduce this way, as paralyzing over cause w/in a single machine, rather than parallelizing over multiple machines is that, this way you don't have to worry about network latency, b/c all the communication, all the sending of the [xx] back + forth, all that happens w/in a single machine. + so network latency becomes much less of an issue compared to if you were using this to over different computers w/in the data sensor. Finally, one last caveat on parallelizing w/in a multi-core machine. Depending on the details of your implementation, if you have a multi-core machine + if you have certain numerical linear algebra libraries. It turns out that the sum numerical linear algebra libraries that can automatically parallelize their linear algebra operations across multiple cores w/in the machine. So if you're fortunate enough to be using one of those numerical linear algebra libraries + certainly this does not apply to every single library. If you're using one of those libraries +. If you have a very good vectorizing implementation of the learning algorithm. Sometimes you can just implement you standard learning algorithm in a vectorized fashion + not worry about parallelization + numerical linear algebra libararies could take care of some of it for you. So you don't need to implement [xx] but. for other any problems, taking advantage of this sort of map reducing commentation, finding + using this MapReduce formulation + to paralelize a cross coarse except yourself might be a good idea as well + could let you speed up your learning algorithm. In this video, we talked about the MapReduce approach to parallelizing ML by taking a data + spreading them across many computers in the data center. Although these ideas are critical to paralysing across multiple cores w/in a single computer as well. Today there are some good open source implementations of MapReduce, so there are many users in open source system called Hadoop + using either your own implementation or using someone else's open source implementation, you can use these ideas to parallelize learning algorithms + get them to run on much larger data sets than is possible using just a single machine.