Machine Learning
Week 10
Large Scale Machine Learning

This week will cover large scale machine learning. Machine learning works best with a vast amount of data to leverage for training. As such, knowing how to handle 'big data' is a very sought after skill.

Gradient Descent with Large Datasets

Learning with Large Datasets

How do we handle big datasets?

Why do machine learning algorithms work so much better now than 5-10 years ago?

- One reason is that we have a LOT more data today than 5-10 years ago.
    - "It's not who has the best algorithm that wins, it's who has the most data.

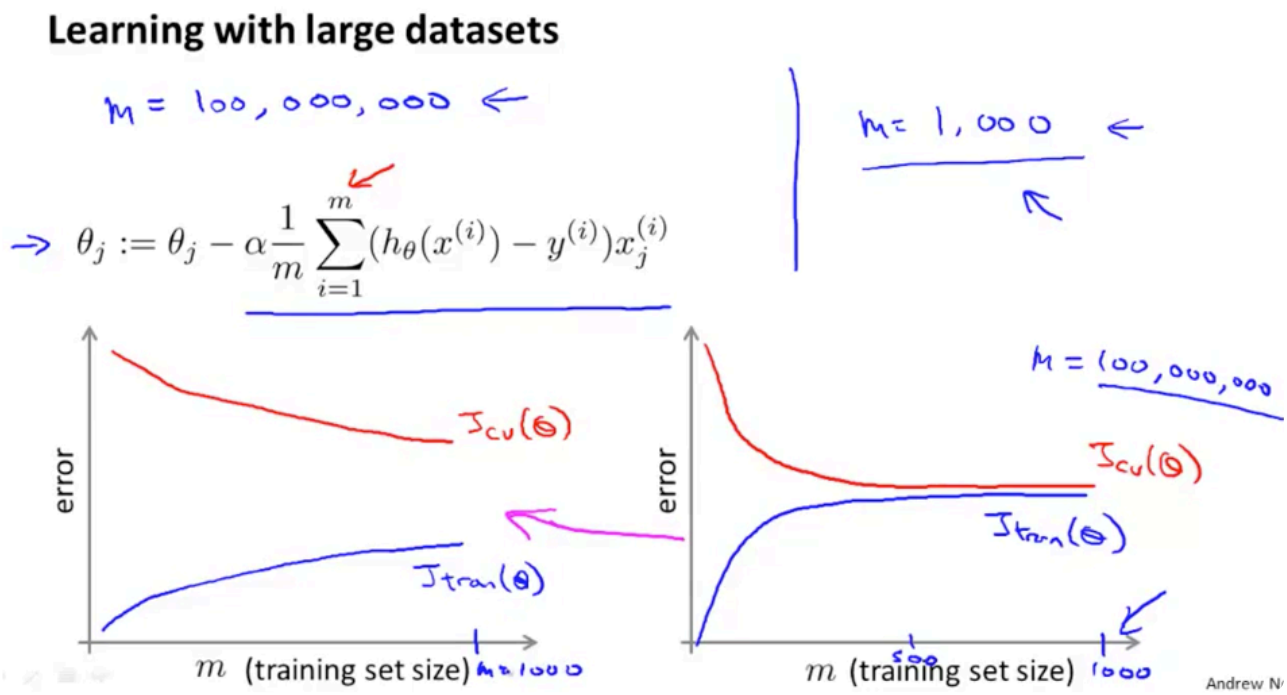
Lets talk about algorithms for dealing with massive data sets.

One of the best ways to achieve high performance machine learning systems is to take a low-bias learning algorithm and train it on a LOT of data.

But learning with large datasets has it's own problems, specifically computational problems.

Ex. The modern US census data has over 100,000,000 data points.

If we were to train a linear regression model on this data, we would need to compute a summation over a 100 million terms to perform a single step of gradient descent. This isn't practical.



It would be more efficient as a sanity check to develop a prototype on a smaller subset of the data.

- Recall plotting the learning curves:
    - If we see high variance (as on the left) on the smaller dataset it would be a valid assumption that more data would improve performance.
    - If we see low variance / high bias (as on the right) on the smaller dataset, it's probably unlikely that increases the amount of data would improve performance and instead you should investigate altering the features or changing the algorithm etc.

Stochastic Gradient Descent

For many learning algorithms like linear regression, logistic regression, neural networks, etc. The way we service the algorithm is by coming up with a cost function or an optimization object. Then, using an algorithm like gradient descent we try to minimize that cost function.

For very large data sets, gradient descent becomes a very computationally expensive procedure.

In this video we will discuss a modification of gradient descent called stochastic gradient descent which will allow us to scale these algorithms to much bigger training sets.

**Note: For this video we are using linear regression as a running example but the ideas of SGD are general and apply to other learning algorithms like logistic regression, neural networks, etc. that are based on minimizing w/ gradient descent.

Recall for linear an example of linear regression minimized with gradient descent we have the following:

Linear regression with gradient descent

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

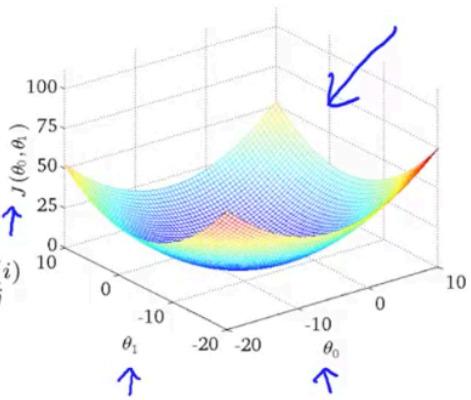
$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

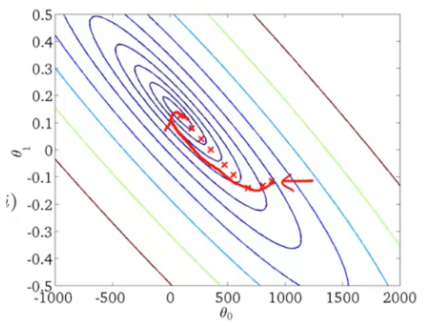
Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(for every $j = 0, \ldots, n$)

}





Batch Gradient Descent is GD using ALL of the training data

- Ex. US census data is 300,000,000 - batch gradient descent would use all 300,000,000 data points.
  - For a single step of gradient descent we would need to use each data point.

Instead, we are going to use a different algorithm (SGD) that only needs to look at one single data point per iteration.

Batch gradient descent

$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$\frac{\partial}{\partial \theta_j} J_{train}(\theta)$$

(for every $j = 0, \ldots, n$)

}

Stochastic gradient descent

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J_{train}(\theta) = \frac{1}{m} \sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

1. Randomly shuffle dataset.
2. Repeat {
   for $i = 1, \ldots, m$ {
   $\theta_j := \theta_j - \alpha (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$
   (for $j = 0, \ldots, n$)
   }
   }

$$\frac{\partial}{\partial \theta_j} cost(\theta, (x^{(i)}, y^{(i)}))$$

$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \ldots$

Andrew Ng

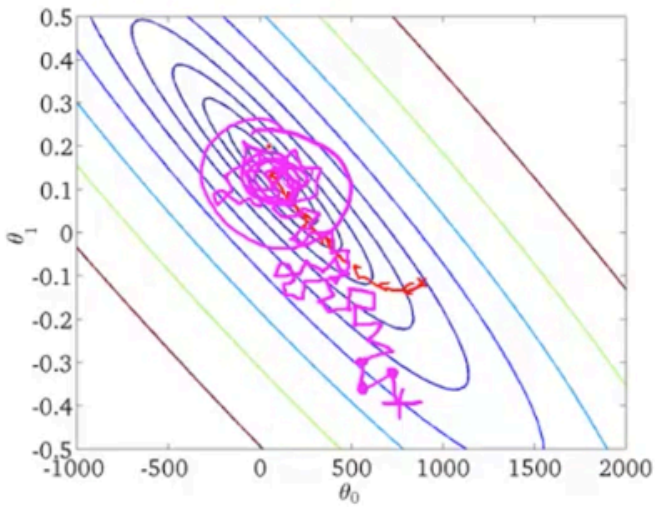
We make progress on each of our datapoint towards the ideal parameters instead of having to look at all data point on each iteration.

(Red == BGD, Purple == SGD)

Stochastic gradient descent

1. Randomly shuffle (reorder) training examples

2. Repeat {
   for $i := 1, \ldots, m$ {
   $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
   (for every $j = 0, \ldots, n$)
   }
   }



SGD doesn't converge like BGD does but instead eventually begins to wander around some region that the minimum is close to.

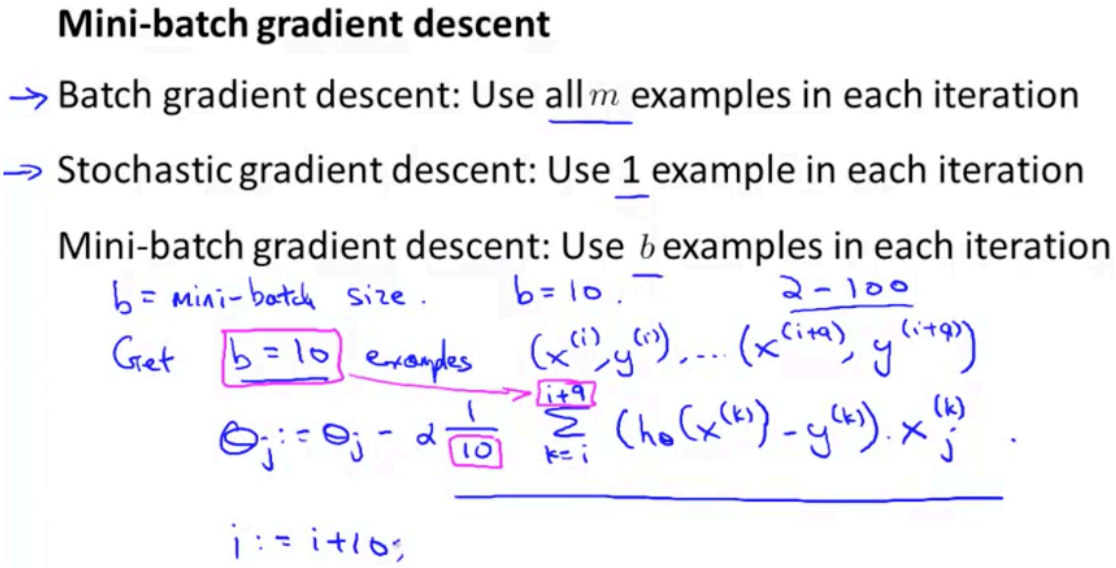
In practice this is usually good enough.

How many times do we perform the outer loop? It depends on the size of the training dataset.
Ex. for the US Census data of 300,000,000 entries, 1 pass through the dataset may likely be enough.

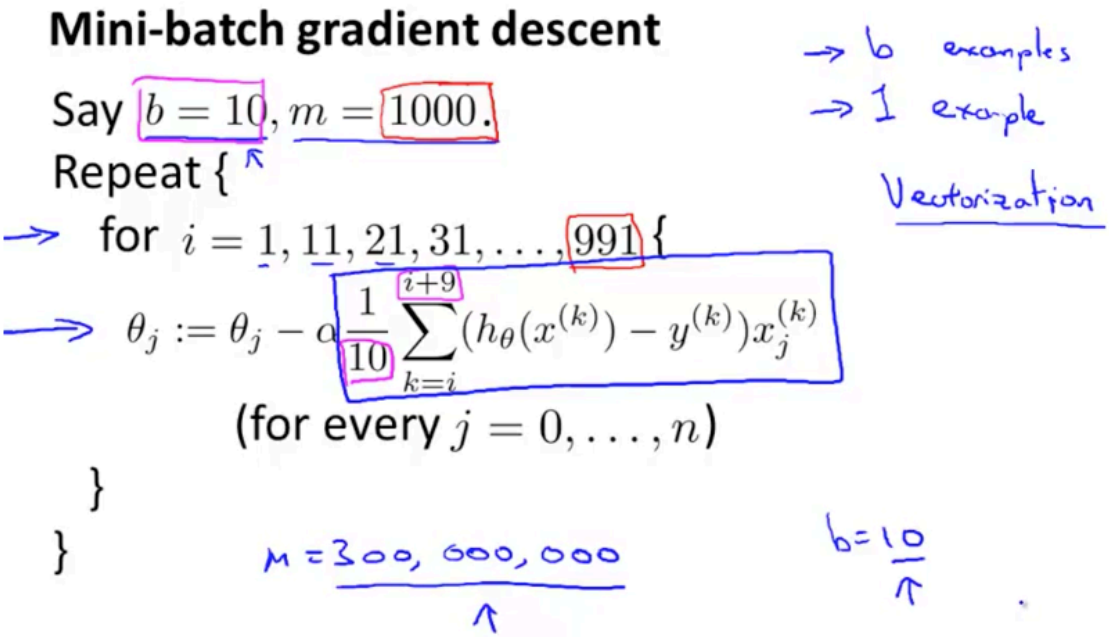
Mini-Batch Gradient Descent

Following the previous video, the obvious question is - can we do something between BDG and SGD by performing SGD looking at more than 1 training example per iteration to improve performance?

Yes! With Mini-batch gradient descent. We then use batch size 'b'.



More formally:



Compared to BGD, Mini BGD allows us to make progress much faster to improve our parameters.

**Note, for Mini BGD to outperform SGD, we need to have a good vectorized implementation in order to partially parallelize the summation.

Stochastic Gradient Descent Convergence

When using SGD, how do we debug it and ensure it is converging okay?
How do we tune the learning parameter alpha for SGD?

This video will discuss the techniques to answer these questions.

Recall for BGD, to check for convergence we plot J over the number of iterations of gradient descent and ensure it is decreasing on each iteration.

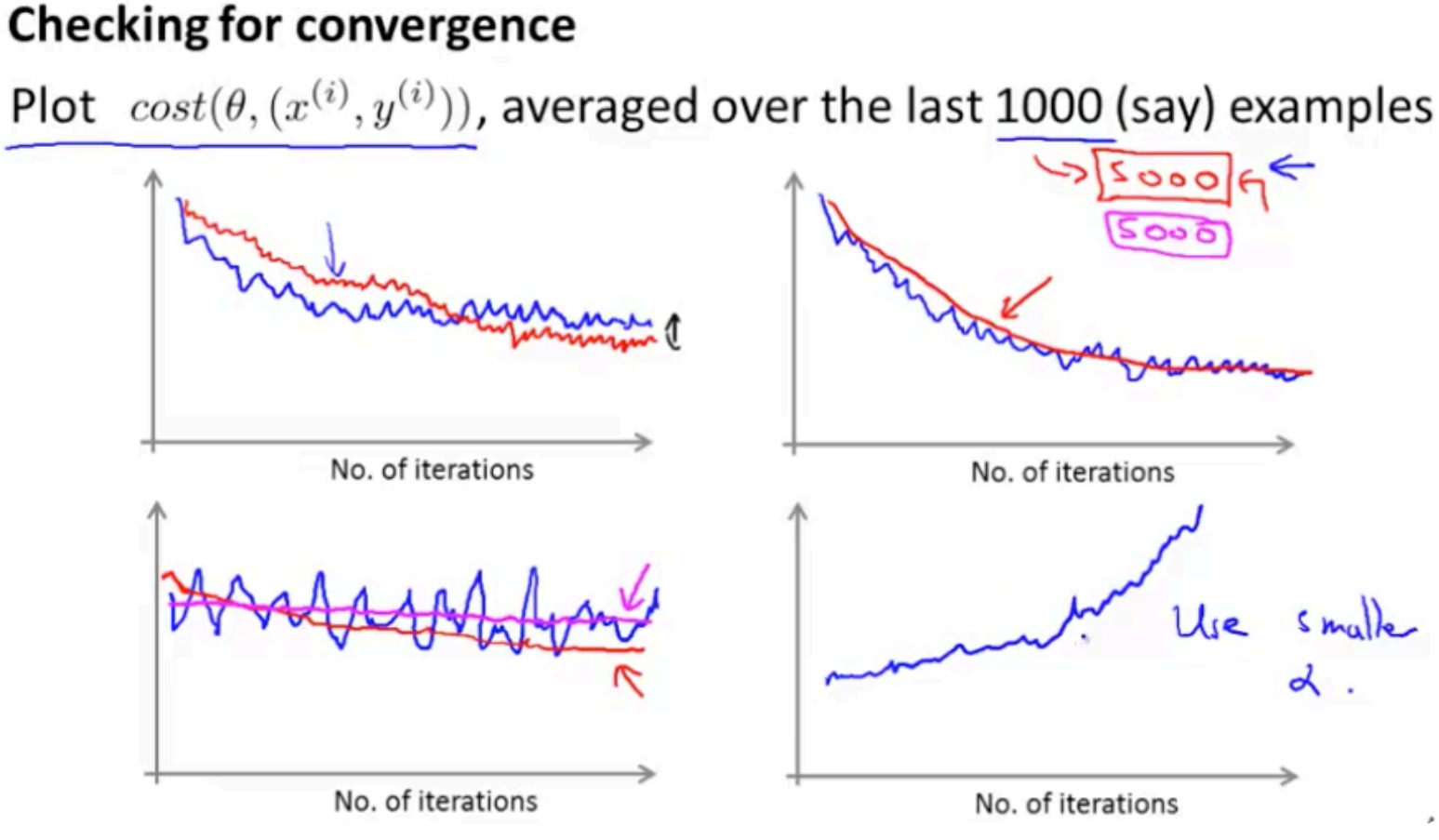
For SGD, while the algorithm is training through the dataset, do the following:

Checking for convergence

- Batch gradient descent:
  - Plot $J_{train}(\theta)$ as a function of the number of iterations of gradient descent.
  - $J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$ M = 300,000,000
- Stochastic gradient descent:
  - $cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$ $(x^{(i)}, y^{(i)})$, $(x^{(i+1)}, y^{(i+1)})$, ...
  - During learning, compute $cost(\theta, (x^{(i)}, y^{(i)}))$ before updating θ using $(x^{(i)}, y^{(i)})$.
  - Every 1000 iterations (say), plot $cost(\theta, (x^{(i)}, y^{(i)}))$ averaged over the last 1000 examples processed by algorithm.

This gives us a running estimate on how well the algorithm is doing.

Here are some examples of how these plots may look:



Top Left:
- 1000 examples, blue is larger learning rate, red is smaller learning rate.
    - For the smaller learning rate, the oscillations around the global minimum are smaller.

Top Right:
- Increasing examples per cost computation can have better results but requires large batch sizes I.e. slower.

Bottom Left:
- Too much noise may imply the batch size is too small to see through the noise.
- Try increasing batch size which may show:
    - (Red) The the cost is in fact decreasing.
    - (Purple) The cost is in fact flat and the algorithm isn't learning.

Bottom Right:
- Algorithm is diverging, use a smaller alpha.

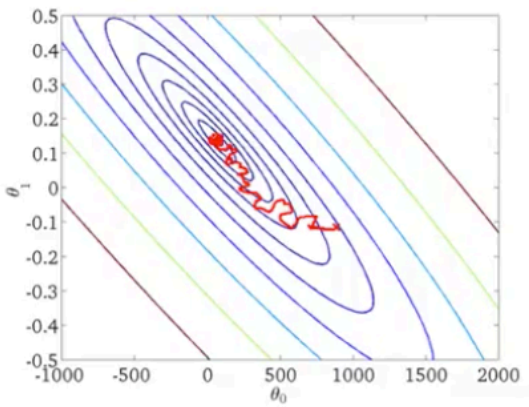
To attempt to converge SGD:

Stochastic gradient descent

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

1. Randomly shuffle dataset.
2. Repeat {
   for $i := 1, \ldots, m$ {
   $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
   (for $j = 0, \ldots, n$)
   }
   }



Learning rate α is typically held constant. Can slowly decrease α over time if we want θ to converge. (E.g. $\alpha = \frac{\texttt{const1}}{\texttt{iterationNumber} + \texttt{const2}}$) $\alpha \to 0$

Note that you need to tune/fiddle around with const1 and const2 for this approach which can make the algorithm for finicky, but if you can tune them well it could be well worth it.

Practically, this approach usually isn't done and alpha is held constant.

Advanced Topics

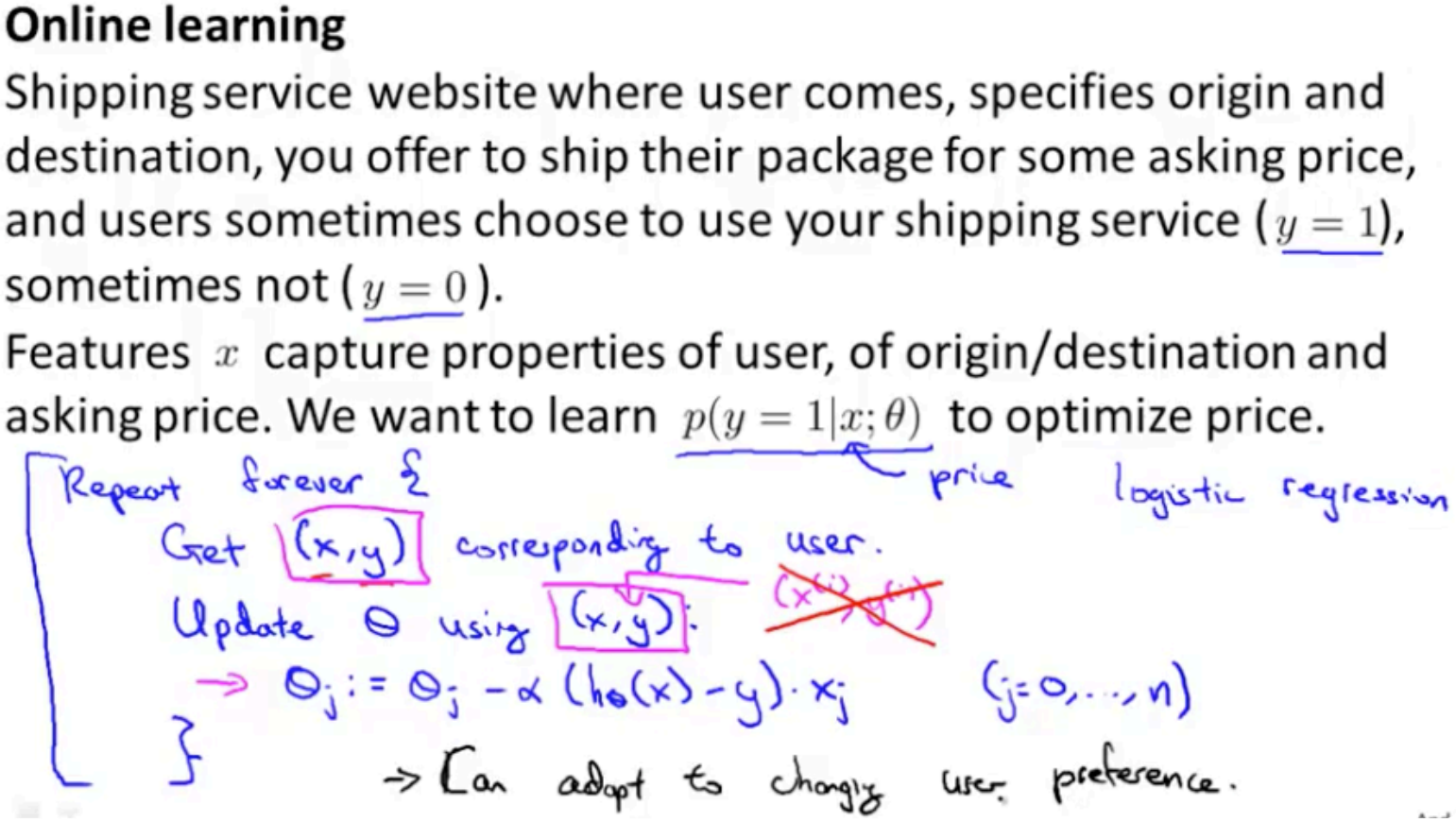
Online Learning

Online learning allows us to model problems where we have a continuous flood/stream of data coming in that we would like an algorithm to learn from.

Ex. For large websites or apps that have floods of users that keep returning, and in term there is a steady stream of data being generated by this continuous stream of users interacting and use the website/app, you can use an online learning algorithm to learn user preferences or behavior and then in turn optimize some of the decisions or features of the app/website. (To get even more people to return more often, slightly nefarious)

Ex. Suppose you run a shipping service and website, depending on the price of shipping for a given origin and destination, the price changes, and with that users either use your service or go to another service.

Let's use logistic regression to optimize the price.



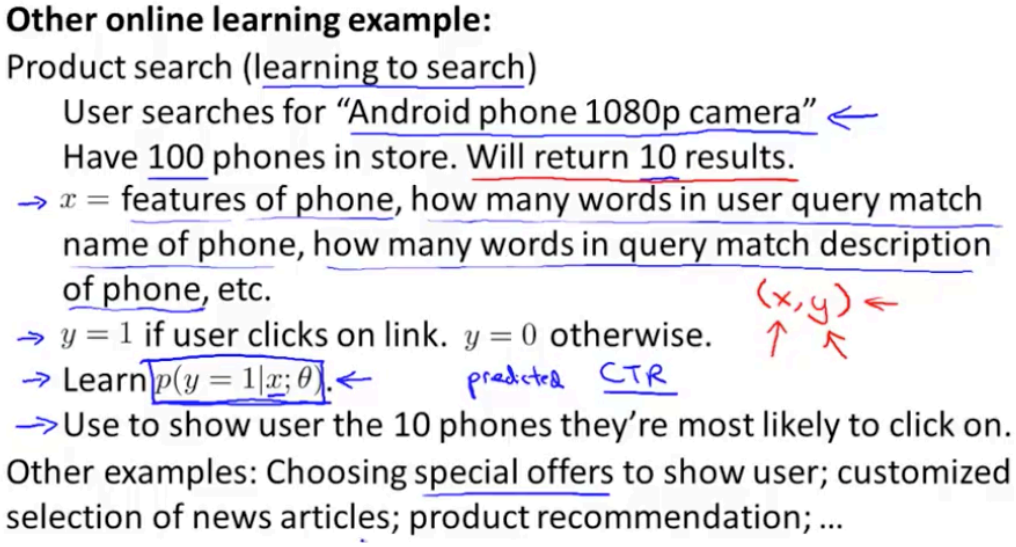
After we have updated theta from an example that has come in, we then discard that example. There is not fixed training set.

An interesting effect of an online learning algorithm is that it can adapt to changes in user preferences.

Ex. As the economy strengthens or declines and users spending changes, this can adapt to those changes.

Here is another example application that we may apply online learning algorithms to - Product Search.

Estimate what a user will click on based on what they searched for and the options provided.



This is called the problem of the 'predicted click through rate (CTR)'

If you can learn the CTR, you can provide better search results.

Combined with a Collaborative Filtering Algorithm this can become very powerful to predict the different CTR for different products and different users.

Map Reduce and Data Parallelism

Some machine learning problems are just too big to run on one machine no matter what algorithm you use.

This video will discuss the map reduce approach to large scale machine learning for these types of situations that need more than 1 computer.

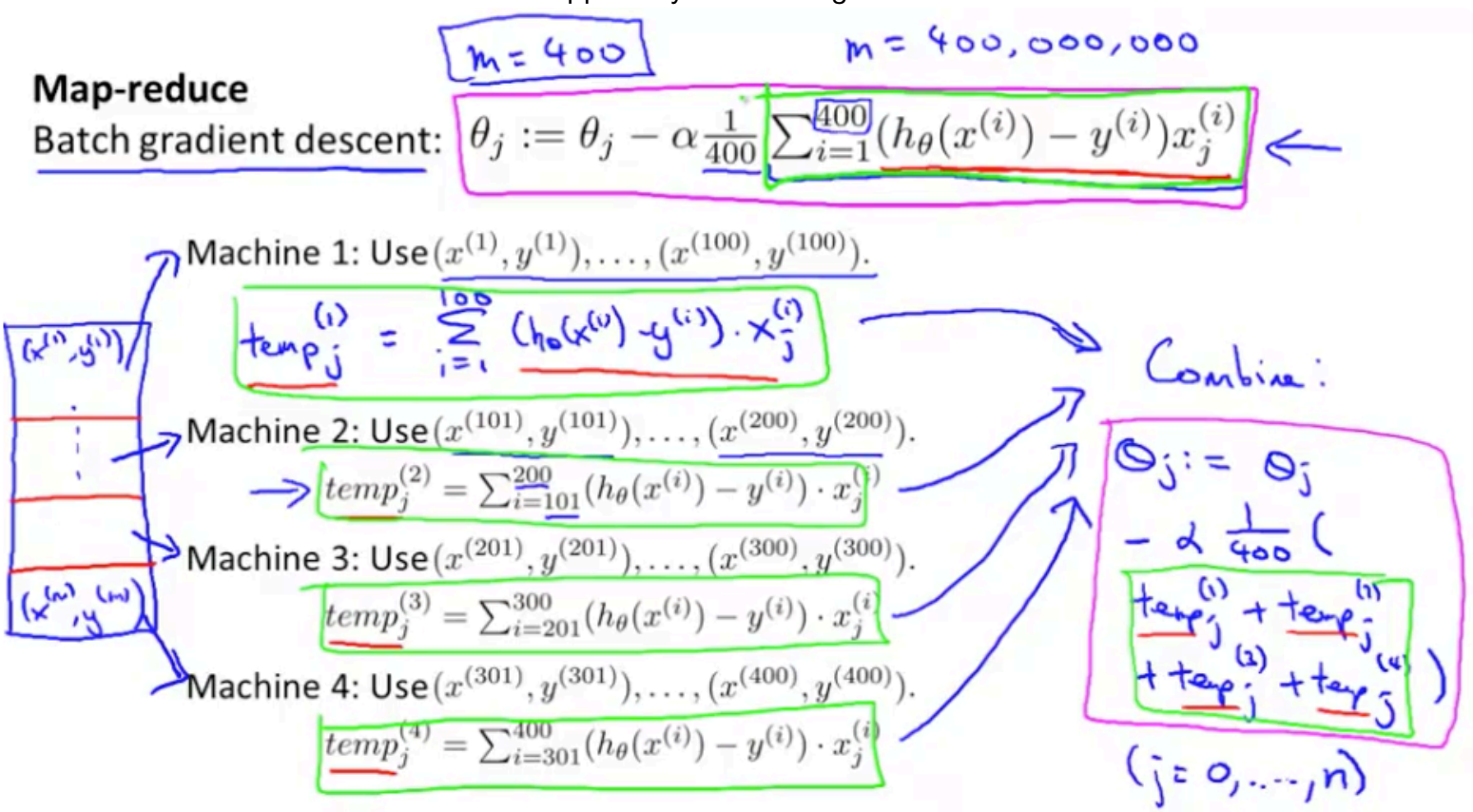
**Note this is an equally important idea to SGD

Here is the idea:

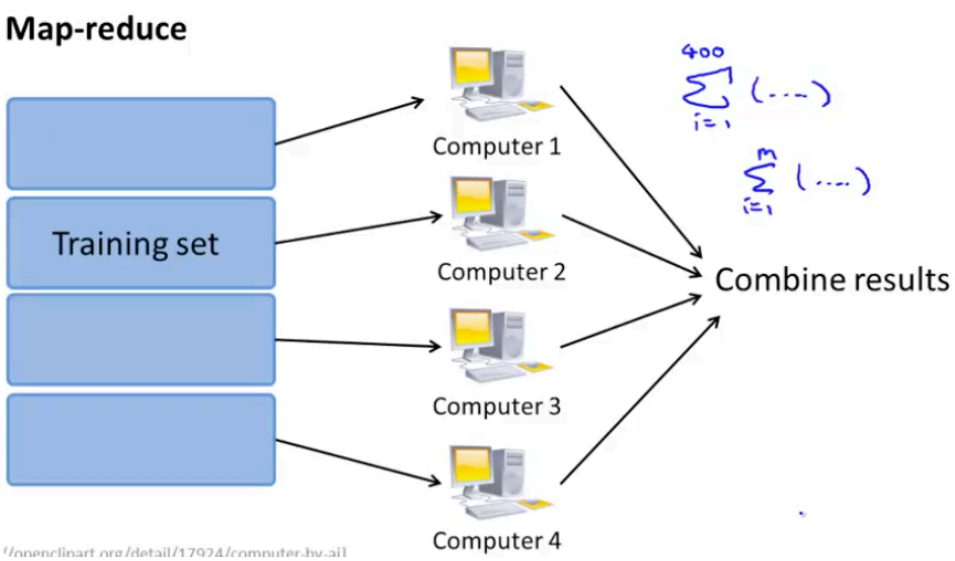
Let us say we want to fit a linear regression model using Batch Gradient Descent w/ ~400,000,000 examples. [400 on slides to make it easy to see]

We can split this training set into separate subsets to run on 4 machines.

Thus each machine does 1/4 the work an supposedly makes things 4 times faster.



Thus this is identical to the Batch Gradient Descent algorithm but with the summation work parallelized.



If you want to perform this parallelization speed up on an ML algorithm to speed up training time, the key question to ask is if the machine learning algorithm you want to use can be expressed as computing sums of functions over the training set.

Lets look at another example.

Say we want to train a logistic regression ML algorithm but we want to use advanced optimization (like fminunc).

Map-reduce and summation over the training set

Many learning algorithms can be expressed as computing sums of functions over the training set.

E.g. for advanced optimization, with logistic regression, need:

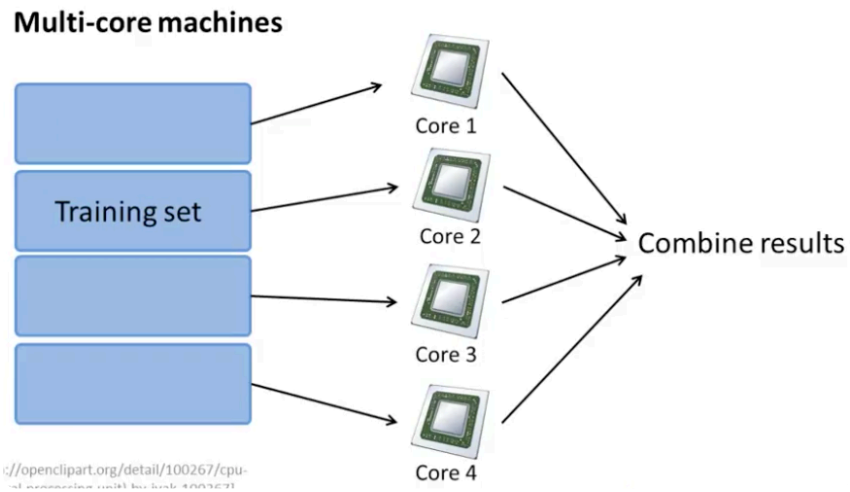
$$J_{train}(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) - (1-y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

$$\frac{\partial}{\partial \theta_j} J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

$temp^{(i)}$ $temp_j^{(i)} \leftarrow$

By breaking up the summations again we are able to parallelize the training process and scale to very large training sets.

Map Reduce can also be applied to multi-core machines.



Some linear algebra libraries have multi-core support built in and by calling their functions you don't need to worry about the multi-core parallelization as they do it automatically.