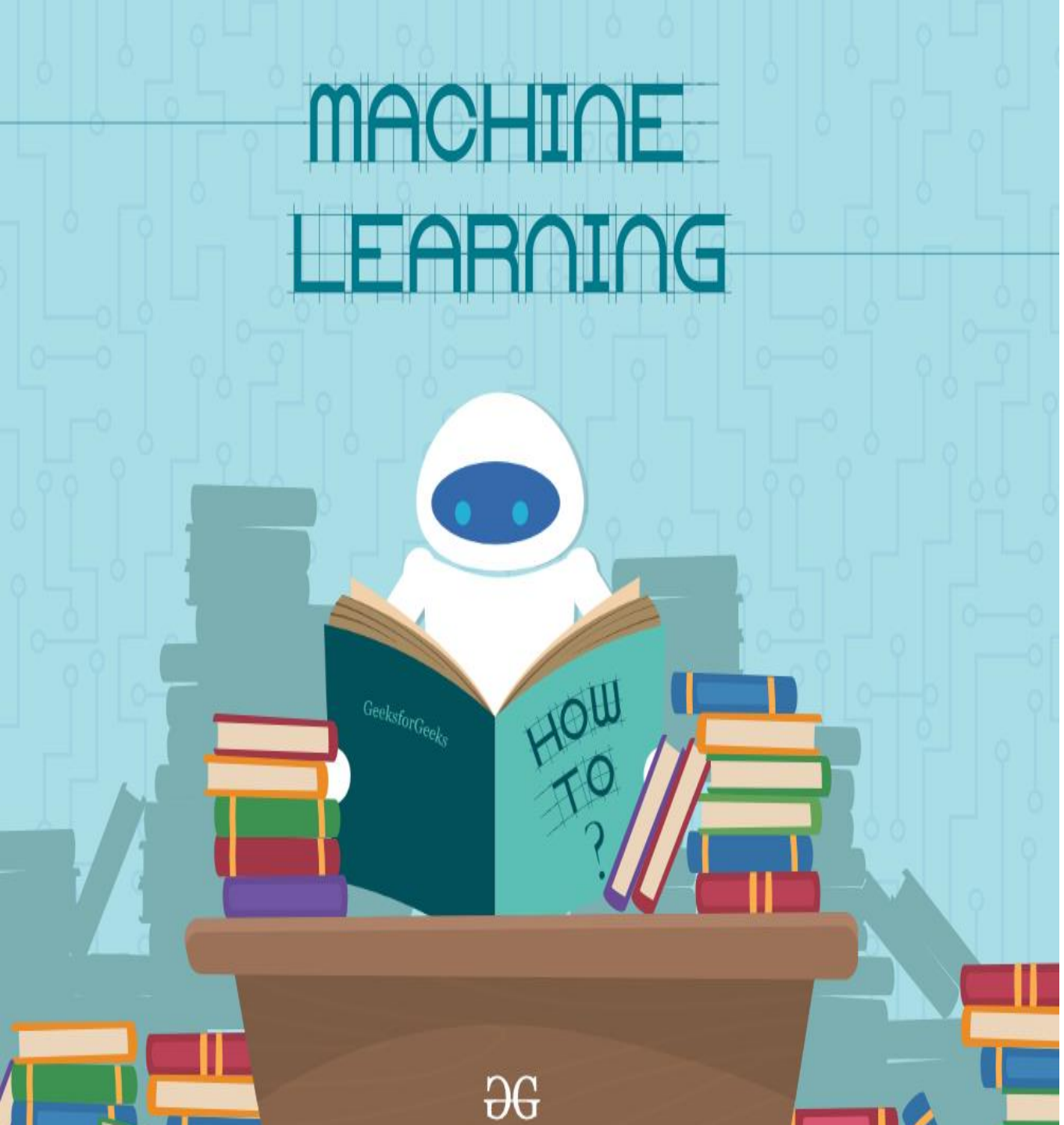


>TCC
THE CODERS' CLUB

Large Scale Machine Learning

WEEK 10

Learning with Large Datasets

Let's say our training set size is, m = 100,000,000

Suppose we want to train a linear regression/logistic regression model.

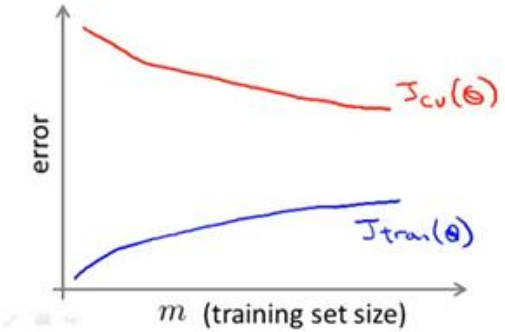
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

If we want to compute gradient descent for m=100,000,000, we need to carry out a summation over a hundred million terms in order to perform a single step of descent.

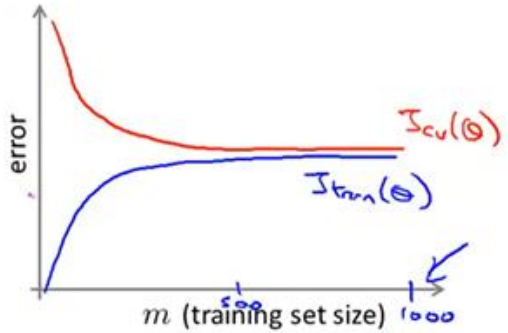
This is computationally very expensive.

Instead, we can randomly pick 1,000 examples out of the 100,000,000 training examples and train our algorithm on these 1,000 examples.

We can then perform a sanity check on the algorithm's performance by plotting a learning curve.



High Variance Algorithm



High Bias Algorithm

Suppose you are facing a supervised learning problem and have a very large dataset (m = 100,000,000). How can you tell if using all of the data is likely to perform much better than using a small subset of the data (say m = 1,000)?

- There is no need to verify this; using a larger dataset always gives much better performance.
- Plot $J_{train}(\theta)$ as a function of the number of iterations of the optimization algorithm (such as gradient descent).
- Plot a learning curve ($J_{train}(\theta)$ and $J_{CV}(\theta)$, plotted as a function of m) for some range of values of m (say up to m=1,000) and verify that the algorithm has bias when m is small.
- Plot a learning curve for a range of values of m and verify that the algorithm has high variance when m is small.

Linear Regression with Batch Gradient Descent:

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

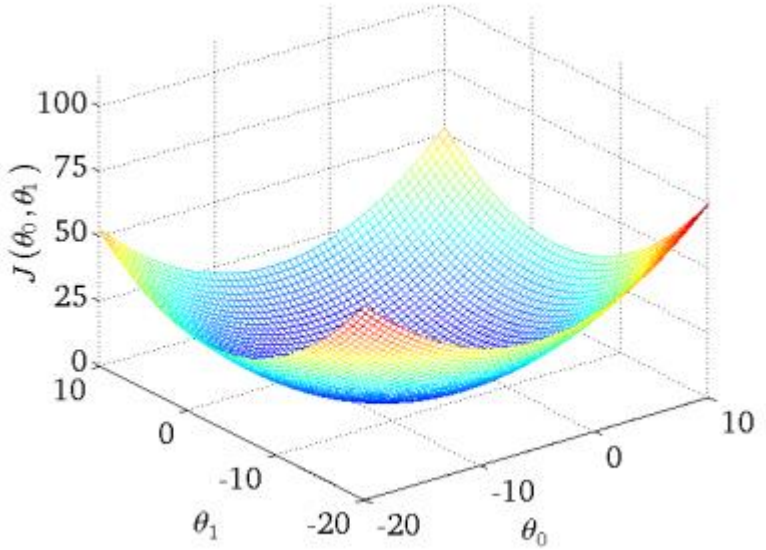
$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

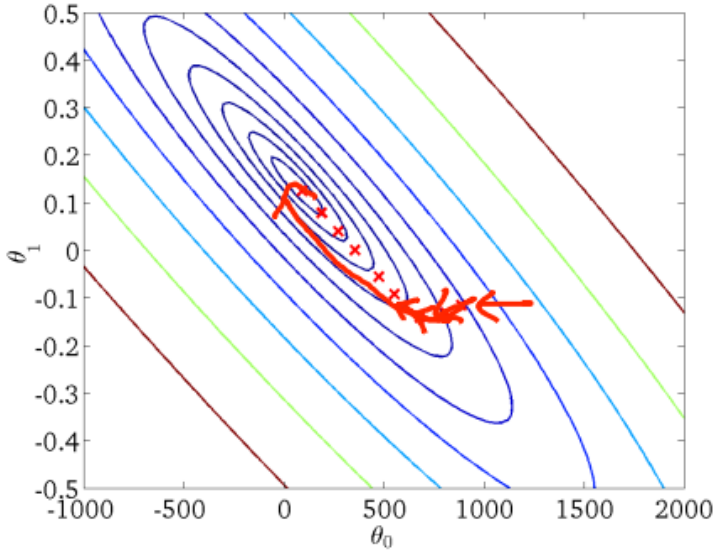
Repeat {

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

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

}

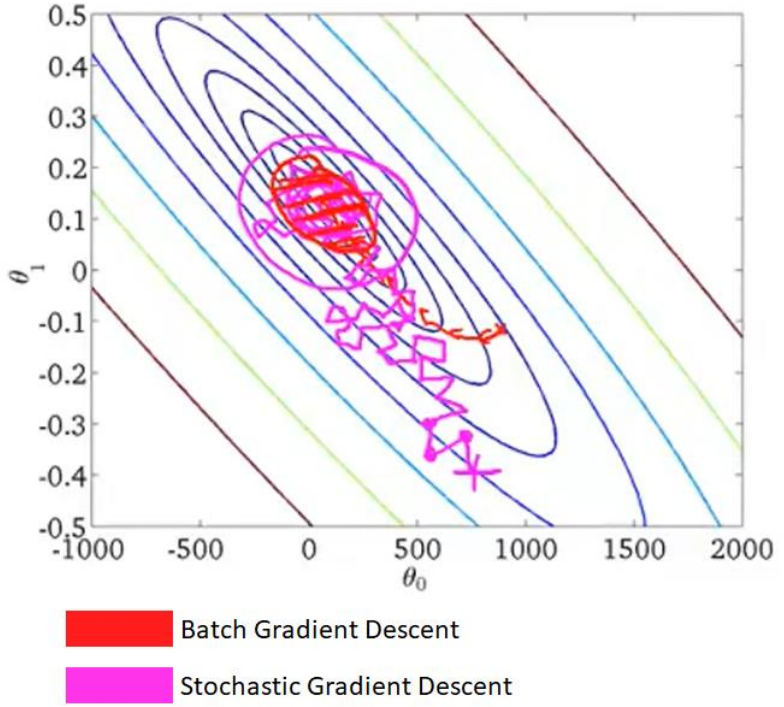




Stochastic Gradient Descent

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 $j = 0, \ldots, n$)
   }
   }



- As you run Stochastic gradient descent, what you find is that it will generally move the parameters in the direction of the global minimum, but not always.
- In fact as you run Stochastic gradient descent it doesn't actually converge in the same sense as Batch gradient descent does and what it ends up doing is wandering around continuously in some region that's in some region close to the global minimum.
- Stochastic Gradient Descent is faster than Batch Gradient Descent.

Which of the following statements about stochastic gradient descent are true? Check all that apply.

- When the training set size m is very large, stochastic gradient descent can be much faster than gradient descent.
- The cost function $J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$ should go down with every iteration of batch gradient descent (assuming a well-tuned learning rate α) but not necessarily with stochastic gradient descent.
- Stochastic gradient descent is applicable only to linear regression but not to other models (such as logistic regression or neural networks).
- Before beginning the main loop of stochastic gradient descent, it is a good idea to "shuffle" your training data into a random order.

Mini-Batch Gradient Descent

Batch Gradient Descent: Use all m examples in each iteration

Stochastic Gradient Descent: Use 1 example in each iteration

Mini-batch Gradient Descent: Use b (2-100) examples in each iteration

Example:

Say b=10, m=1000

Repeat {

for i = 1, 11, 21, 31, … , 991 {

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every j=0, … , n)

}

}

Suppose you use mini-batch gradient descent on a training set of size m, and you use a mini-batch size of b. The algorithm becomes the same as batch gradient descent if:

- b = 1
- b = m/2
- b = m
- None of the above

Stochastic Gradient Descent Convergence

Checking for convergence:

Batch Gradient Descent:

- Plot $J_{train}(\theta)$ as a function of the number of iterations of gradient descent.
- $J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

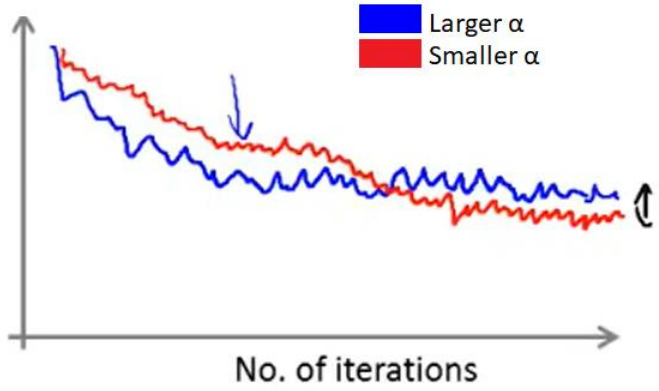
Stochastic Gradient Descent:

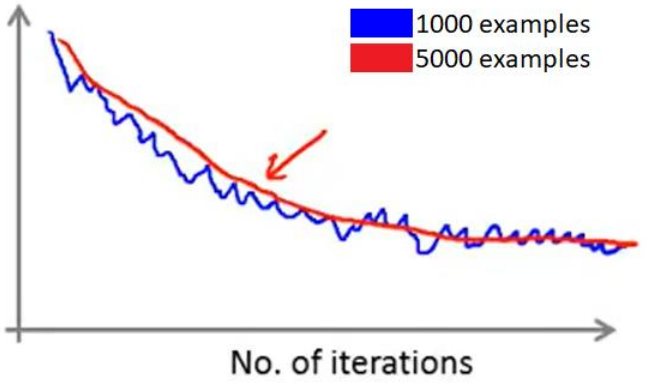
- $cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$

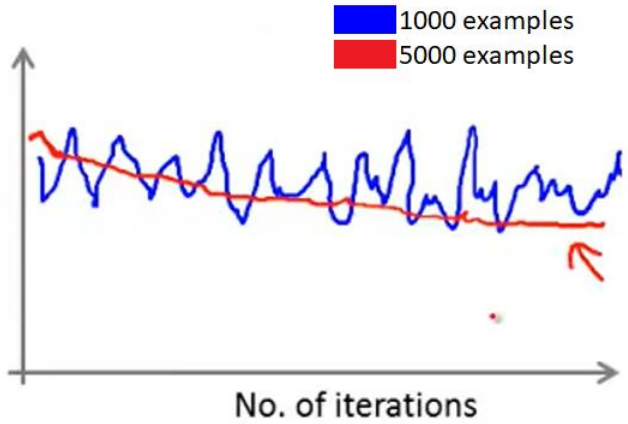
- During learning, compute cost(θ,($x^{(i)}$, $y^{(i)}$)) before updating θ using ($x^{(i)}$, $y^{(i)}$)
- Every 1000 iterations (say), plot cost(θ,($x^{(i)}$, $y^{(i)}$)) averaged over the last 1000 examples processed by algorithm.

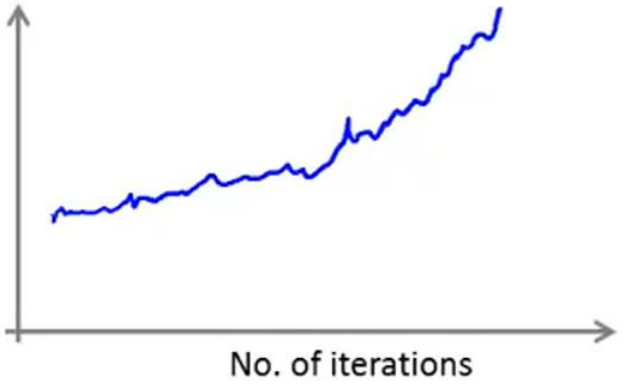
Checking for convergence:

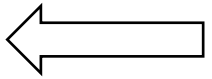
Plot cost(θ,($x^{(i)}$, $y^{(i)}$)), averaged over the last 1000 (say) examples











Algorithm diverges
(use smaller α)

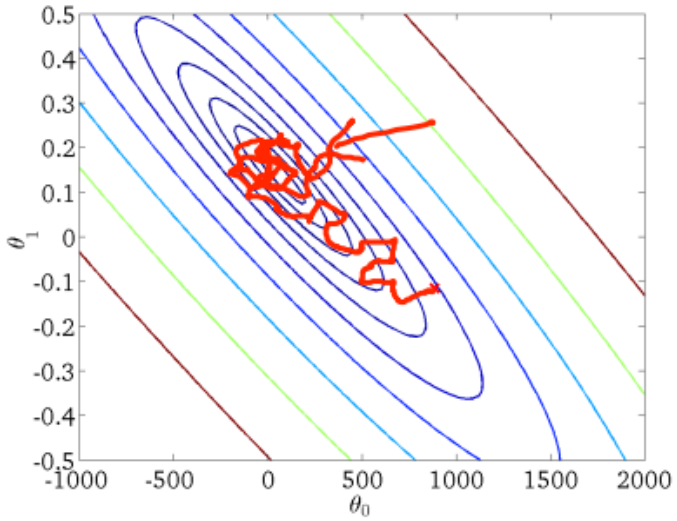
Stochastic Gradient Descent

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

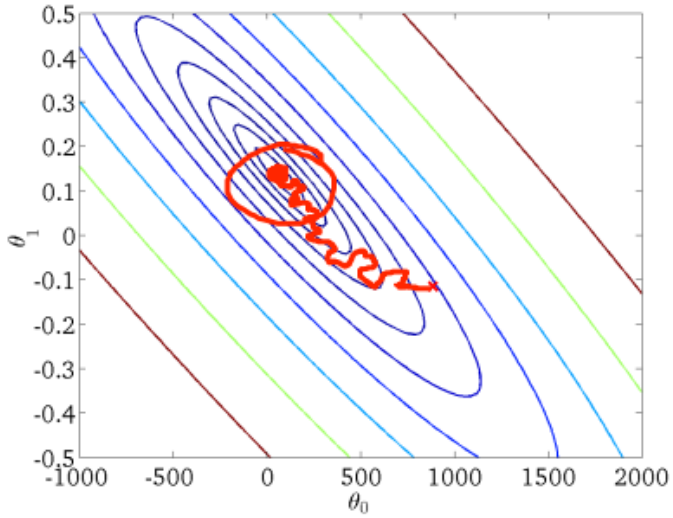
$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

1. Randomly shuffle (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 $j = 0, \ldots, n$)
   }
   }



For larger α



For smaller α

- Learning rate α is typically held constant.
- Can slowly decrease α over time if we want θ to converge.

Example: $\alpha = \frac{\texttt{const1}}{\texttt{iterationNumber + const2}}$

Which of the following statements about stochastic gradient descent are true? Check all that apply.

- Picking a learning rate α that is very small has no disadvantage and can only speed up learning.

- If we reduce the learning rate α (and run stochastic gradient descent long enough), it's possible that we may find a set of better parameters than with larger α.

- If we want stochastic gradient descent to converge to a (local) minimum rather than wander of "oscillate" around it, we should slowly increase α over time.

- If we plot cost(θ,($x^{(i)}$, $y^{(i)}$)) (averaged over the last 1000 examples) and stochastic gradient descent does not seem to be reducing the cost, one possible problem may be that the learning rate α is poorly tuned.

Online Learning

Suppose you run a shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service (y=1) and sometimes not (y=0).

Features x capture properties of user, of origin/destination and asking price. We want to learn $p(y=1|x;\theta)$ to optimize price.

Algorithm:

Repeat forever {

Get (x, y) corresponding to user

Update θ using (x, y)

$\theta_j := \theta_j - \alpha(h_\theta(x) - y)\, x_j$ (for $j = 0, \ldots, n$)

}

- An interesting property of Online Learning algorithm is that it can adapt to changing user preferences.

Other Online Learning Examples:

Product Search (learning to search):

- User searches for “Android Phone 1080p camera”
- There are 100 phones in store, out of which 10 results would be returned upon searching.
- x = features of phone, how many words in user query match name of phone, how many words in query match description of phone, etc.
- y = 1 if user clicks on link and y = 0 otherwise.
- Learn $p(y=1|x;\theta)$, meaning, Predict **Click Through Rate (CTR)**
- The predicted CTR is then used to show user the 10 phones they’re most likely to click on.

Other examples:
Choosing special offers to show user, customized selection of news articles, product recommendation, etc.

Some of the advantages of using an online learning algorithm are:

- It can adapt to changing user tastes (i.e., if $p(y|x;\theta)$ changes over time).
- There is no need to pick a learning rate α.
- It allows us to learn from a continuous stream of data, since we use each example once then no longer need to process it again.
- It does not require that good features be chosen for the learning task.

Map-Reduce and Data Parallelism

Map-Reduce:

- Map-Reduce divides the training set into x subsets and computation is performed on each subset by a different machine.
- The results of computation are then combined together at a central server.
- This increases the efficiency and speeds up the process by approximately x times.

Example: Let's say we want to fit a linear regression/logistic regression model for m=400
The dataset for m=400 is $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(400)}, y^{(400)})$

The batch gradient descent is given by: $\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

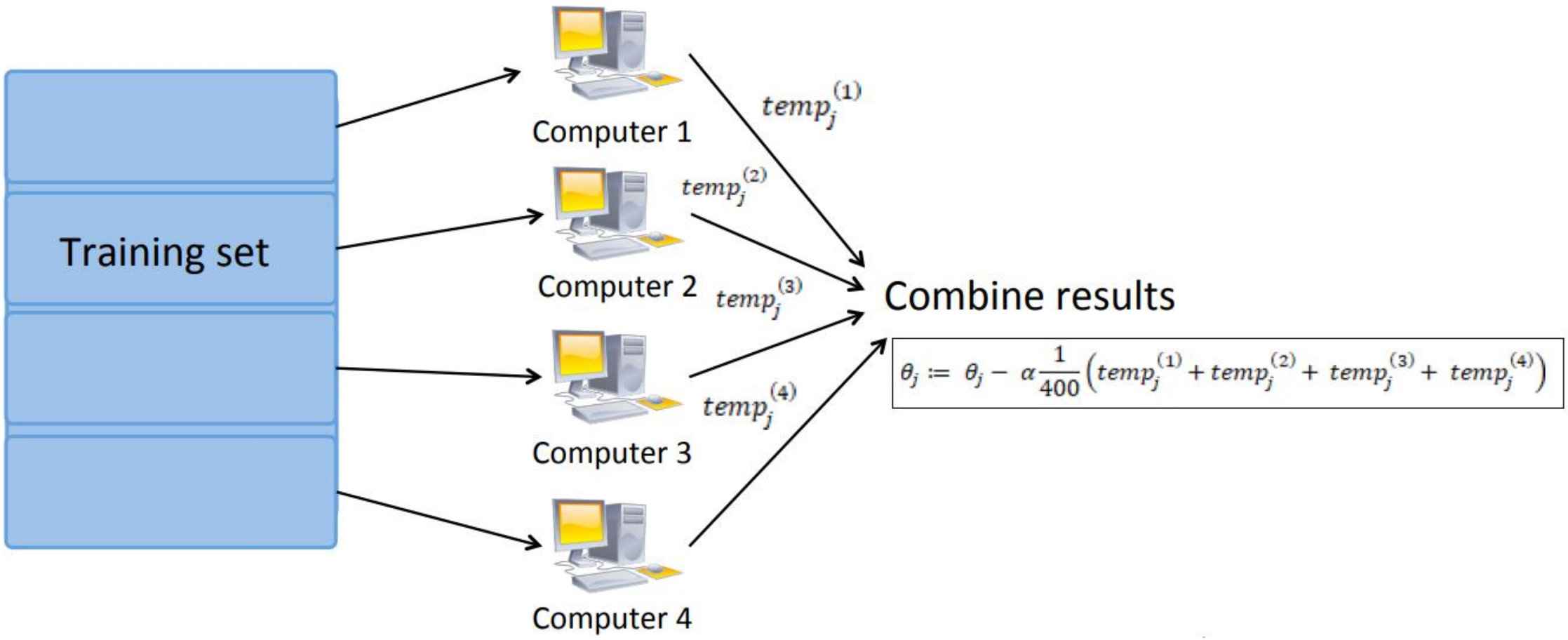
Machine 1: Use $(x^{(1)}, y^{(1)}), \dots, (x^{(100)}, y^{(100)})$ and compute $temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

Machine 2: Use $(x^{(101)}, y^{(101)}), \dots, (x^{(200)}, y^{(200)})$ and compute $temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

Machine 3: Use $(x^{(201)}, y^{(201)}), \dots, (x^{(300)}, y^{(300)})$ and compute $temp_j^{(3)} = \sum_{i=201}^{300} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

Machine 4: Use $(x^{(301)}, y^{(301)}), \dots, (x^{(400)}, y^{(400)})$ and compute $temp_j^{(4)} = \sum_{i=301}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

Map-Reduce using Multiple Machines



Map-Reduce and summation over the training set:

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

Example:

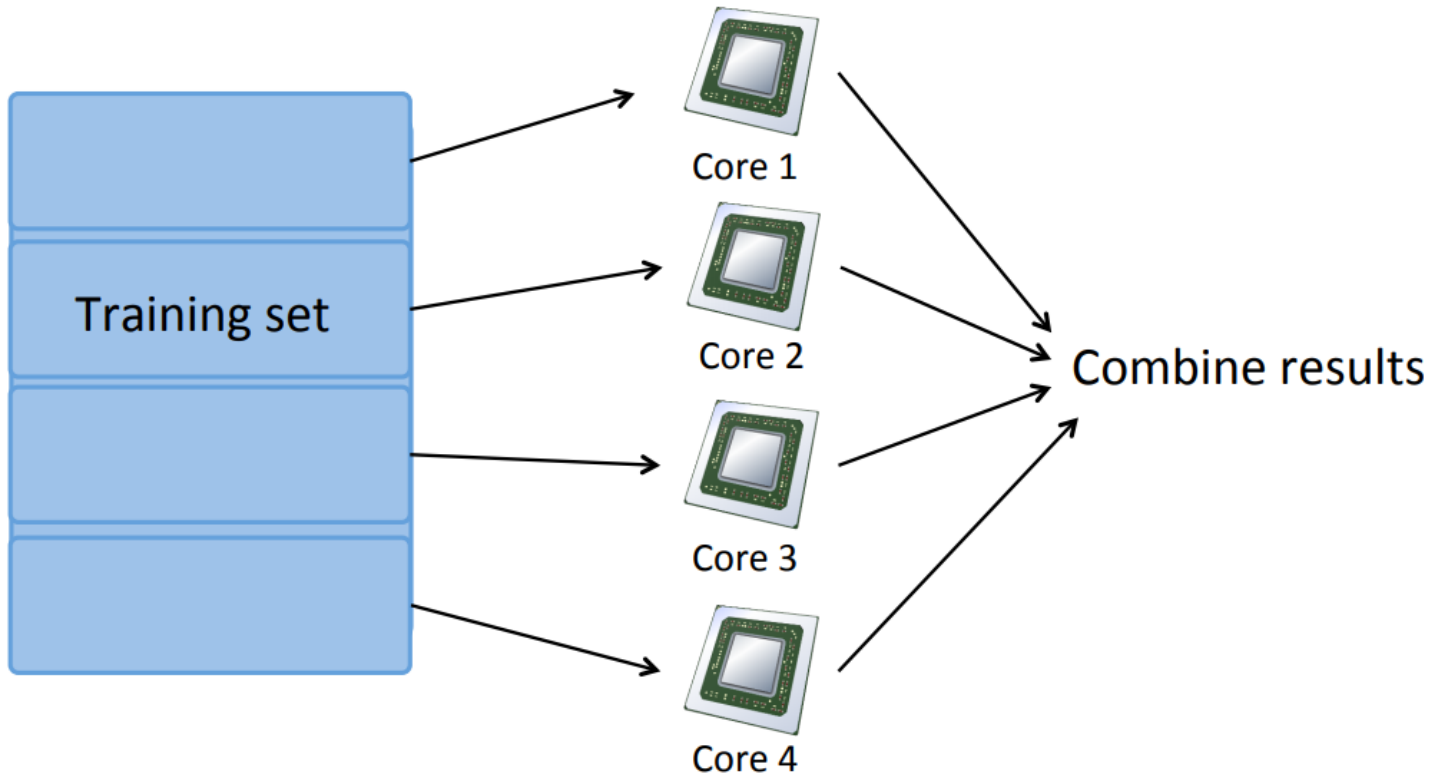
For advanced optimization, with logistic regression, need:

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

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

Multi-core Machines:

Just like multiple machines, map-reduce can also be performed on one single machine having multiple cores. Thus, computation is done separately on each core and the results of computation are then combined together.



Suppose you apply the map-reduce method to train a neural network on ten machines. In each iteration, what will each of the machines do?

- Compute either forward propagation or back propagation on 1/5 of the data.
- Compute forward propagation and back propagation on 1/10 of the data to compute the derivative with respect to that 1/10 of the data.
- Compute only forward propagation on 1/10 of the data. (The centralized machine then performs back propagation on all the data)
- Compute back propagation on 1/10 of the data (after the centralized machine has computed forward propagation on all of the data).