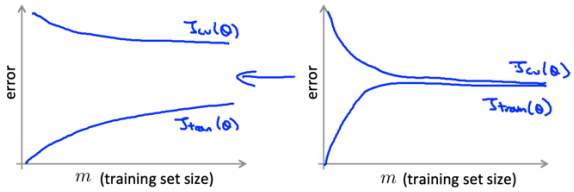
Machine Learning for Big Data:

- Large Datasets give us more basis to add performance to our models.
- However it is also more computationally expensive to run such datasets.
- Thus the overhead hardware requirements to train such a model are also thus consequently high.
- Hence before committing to training a model on a very large dataset, it is a good practice and often necessary to sanity check and train our model on a smaller subset first to indeed confirm that more data would improve the performance of our models
- Eg m =1000000000, check using m = 10000 images first to ensure viablility of model.

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



If the case is indeed on the left, then we proceed. Else re-evaluating the model would be a more fruitful means to find improvements.

Furthermore:
Batch Gradient Descent is a very computationally expensive algorithm when we are dealing with millions or billions of training examples.

Hence faster algorithms have been developed:

Stochastic Gradient descent:

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

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

$$The\ algorithm:$$

1. Randomly shuffle dataset
2. Repeat{

    For i=1:m {

$$\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$
$$for\ j = 0, \dots, n$$

    }

}

$$\frac{\partial}{\partial \theta_j} cost\left(\theta, (x^{(i)}, y^{(i)})\right) = (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

This algorithm is more computationally efficient than batch gradient descent:
As each training example causes θ_j to come closer to convergence.
However, it will never reach the absolute optimum, instead the value of theta will oscilate around the contours of convergence, based on the value of α used.

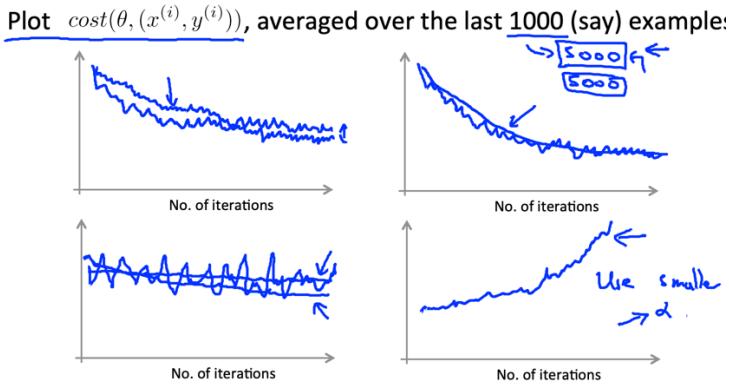
Mini-Batch Gradient Descent:
Use b examples in each iteration:

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

Hence we take every b interval of data, calculate the average cost function and update the theta values instead.
This algorithm can even perform slightly faster than Stochastic Gradient Descent, as it has a vectorized implementation.

Checking for convergence:

- Batch Gradient Descent:
  - ⇨ Plot $J_{train}(\theta)$ as a function of the number of iterations of gradient descent
- Stochastic Gradient Descent:
  - ⇨ Every eg: 1000 iterations plot $cost(\theta, \left(x^{(i)}, y^{(i)}\right))$ over the last 1000 examples processed by the algorithm.



In general a smaller α value will cause a closer oscillation around the convergence, and hence could cause marginal improvements in cost function.

Mini-BSGD would also cause the function plotted to have less noise and fluctuations, especially the larger the value of batch size b.

There is a way to get SGD to converge, or come very close to convergence throught adjusting the learning rate:

$$\alpha = \frac{const1}{iteration\ Number + const2}$$

But requires extra tuning and trial and error.

For Large Scale ML operations:
If there is a constant online stream of data we can perform what is called: Online learning. This is where a algorithm continuously learns new data from it's users.

Eg:
Infinite Loop {
Predict a set of products
Get (x,y) feedback corresponding to user
Update θ using (x,y){
$\theta_j := \theta_j - \alpha(gradient\ computation)$
}
}
⇨ Hence the algorithm is able to adapt to changing user preferences.

Map Reduce and Data Parallelism:
Condition: Model must be able to be expressed as computing the sum of functions over the training set.