WEEK-10

⇒ When using a very big dataset ($m = 100$ million) we can randomly pick a small no. of examples ($m = 1000$) and perform ML algos on it

→ If the graph of algo on small m, shows high variance, then adding more examples will be good.

→ If the graph of algo on small m, shows high bias, then adding more features or changing algo might help.

⇒ STOCHASTIC GRADIENT DESCENT

⇒ Batch Gradient Descent (one we have been using till now)

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

Repeat {

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

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

Becomes very computationaly expensive for $m = 100,000,000$

}

⇒ STOCHASTIC G.D. (we compute θ on one example)

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

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

⇒ Algo

1) = Randomly shuffle the dataset

2) = Repeat {

for $i = 1, \ldots, n$ {

$\theta_j := \theta_j - \alpha \underbrace{(h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}}$

for $j = 0, \ldots, n$

this executes over every example and computes a minimum & after every example it reaches closer to minima.

}

}

→ Its path is very random, depends on every example but still it manages to reach to very close to the local minima.

⇒ Mini Batch GD

→ Batch GD → Uses all m examples in each direction.

→ Stochastic GD → Use 1 example in each direction

⇒ Mini Batch GD → use b examples in each direction

b = mini-batch size. , b = 10 range → 2 - 100

Get b = 10 examples → $(x^{(i)}, y^{(i)}) \dots (x^{(i+9)}, y^{(i+9)})$

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

$i = i + 10;$

⇒ Algo →

Say b = 10, m = 1000

Repeat {

for $i = 1, 11, 21, 31 \dots 991$ {

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

for every $j = 0 \dots n$

}

}

⇒ Mini batch GD can only outperform Stochastic GD with a good vectorized implementation.

⇒ To find that the cost function is decreasing we would plot the cost function against ~~number~~ number of iterations.

⇒ Checking for convergence

⇒ Stochastic GD →

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

→ 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.

⇒ Increasing averaging over the no of examples to 5000, may help to get a smoother curve.

⇒ If the graph is increasing, decrease the α (learning rate).

⇒ Can slowly decrease α overtime if we want θ to converge) $\alpha = \frac{const1}{iterationNumber + const2}$

Online Learning Algorithms

⇒ For when we have unlimited supply of data (like of users to major e-commerce website).

Repeat forever {

→ Get (x, y) corresponding to user

→ Update θ using (x, y)

→ $\theta_j : \theta_j - \alpha (h_\theta(x) - y) \cdot x_j$

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

}

⇒ we train on an example and then discard the example as we have unlimited supply.

⇒ The above algo can adapt to ~~change~~ changing user preferences.

⇒ Example → Product Search → Predicted CTR
aka → Predicted Click Through Rate

→ Customized selection of news articles
→ choosing special offers to show user

⇒ Map-Reduce technique helps us to parallelize the work, i.e. we can divide the dataset in any number of parts depending on the number of computers we have and compute the cost f^n & minimization of cost f^n on every part of dataset on different computers.

→ Then after computing $J(\theta)$ & $\frac{\partial}{\partial \theta_j} J(\theta)$ for every part separately, we can add them up and then this way, our computational speed increases many times.

⇒ Even on a single computer, we can ~~decide~~ divide the dataset for every core in our CPU and then we can sum over the computed things in the same computer

→ We can apply the map reduce method to train a neural network on n machines. In each iteration, each machine will compute the forward P & back P on 1/10 of data to compute the derivative with respect to that 1/10 of the data