Contents

Learning with Large Datasets
Optical Character Recognition

March 26, 2018

Part I

Week 10

0.1 Stochastic Gradient Descent

0.1.1 Principle

Batch Gradient Descent	**Stochastic Gradient Descent**
$J_{\text{train}}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$	$cost(\theta,(x^i,y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)} - y^{(i)})^2$ $J_{\text{train}}(\theta) = \frac{1}{m}\sum_{i=1}^{m}\text{cost}(\theta,(x^i,y^{(i)})$
	setp1: Randomly shuffle the data set
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 $\}$	**step2**: Repeat $\{$ for $i = 1,2,3,...m$ $\{$ $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)} - y^{(i)})x_j^{(i)}$ for $j = 0,1,...,n$ $\}$

In Stochastic Gradient Descent, θ_j is adjusted after each training example. Typically, the outer loop can be repeated 1 to 10 times.

0.1.2 Stochastic Gradient Descent Convergence

To check for convergence:

- During learning, compute $\text{cost}(\theta_1,(x^{(i)},y^{(i)})$ before updating θ using $(x^{(i)},y^{(i)})$, where:

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

- Every 1000 iterations (say), plot $\text{cost}(\theta_1,(x^{(i)},y^{(i)})$ average over the last 1000 examples processed by the algorithm.

- Slowly decrease α overtime to get convergence:

$$\alpha = \frac{\text{const1}}{\text{iteration} \quad \text{Nbr} + \text{const2}} \tag{2}$$

  However, it is more common to keep α constant.

0.2 mini-batch gradient descent

1. **Batch Gradient Descent**: use all m examples in each iteration

2. **stochastic gradient descent**: use 1 example in each iteration

3. **mini-batch gradient descent**: use b examples (mini batch) in each iteration.
   Say $b = 10$ and $m = 1000$

$$
\begin{aligned}
&\text{Repeat } \{ \\
&\quad \text{for } \; k = i; i + 10: \\
&\quad \theta_j := \theta_j - \alpha \frac{1}{10} \sum_{i}^{i+9} (h_\theta(x^{(k)} - y^{(k)}) x_j^{(k)} \\
&\quad \text{for } \; \text{every } \; j = 0, ..., n \\
&\quad \} \\
&\}
\end{aligned} \tag{3}
$$

Minibatch can be faster than Stochastic if the loop is vectorized, and therefore enabling parallel calculation.

0.3 Online Learning Algorithm

Online Learning Algorithm allows to model problems when there is a continuous stream of data coming in (continuous learning).

- *For example, a shipping service website where the user specifies a origin and destination for a package. The company offer to ship the package for some asking price, and users sometimes choose to use the company service ($y = 1$) or 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.
- We would then use logistic regression to calculate $p(y = 1|x; \theta)$:
  Repeat forever {
  get (x, y) corresponding to a user on website.
  online learning algorithm: update θ using (x, y):

$$\theta_j := \theta_j - \alpha(h_\theta(x) - y)x_j \tag{4}$$

  where $j = 0, ...n$

0.4 MapReduce and Data Parallelism

Let's consider a problem with $m = 400$ training examples, for which we run batch gradient descent.

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{4} 00(h_\theta(x^{(i)} - y^{(i)})x_j^{(i)} \tag{5}$$

In **MapReduce**, the dataset is splitted in subset (for example 4), so every single small set would be used and ran simultenously on 4 machines:

$$\begin{aligned}
temp_j^{(1)} &= \sum_{i=1}^{1} 00(h_\theta(x^{(i)} - y^{(i)})x_j^{(i)} \\
temp_j^{(2)} &= \sum_{i=101}^{2} 00(h_\theta(x^{(i)} - y^{(i)})x_j^{(i)} \\
temp_j^{(3)} &= \sum_{i=201}^{3} 00(h_\theta(x^{(i)} - y^{(i)})x_j^{(i)} \\
temp_j^{(4)} &= \sum_{i=301}^{4} 00(h_\theta(x^{(i)} - y^{(i)})x_j^{(i)}
\end{aligned} \tag{6}$$

The data is then sent to a centralized master server for recombination:

$$\theta_j := \theta_j - \alpha \frac{1}{400}(temp1 + temp2 + temp3 + temp4) \tag{7}$$

If the learning algorithm can be expressed as a summation over the training set, then MapReduce can be used. However, MapReduce can be slow due to Network latency. Ther are a few open source implementation of MapReduce like **Hadoop** parallelism learning algorithm. Note that a similar approach can be used on **multi-core machine** where summation are splitted over several cores. This doe not have the issue of network latency,