GRADIENT DESCENT

1. Gradient Descent

As the final topic of the course, we'll be studying our first Machine Learning algorithm – Gradient Descent. This is a very general-purpose algorithm and applicable in a variety of learning tasks, as we'll see shortly.

From basic calculus, we know that a function reaches its local extremum points (i.e. minima and maxima) when its derivative is zero. We can intuitively see this from the accompanying figure (Fig. 1). Using limits, we can prove that at local extrema, the derivative of the function is zero. Let's assume that our function $f(x)$ is differentiable in the interval (a, b) and reaches its local maximum at x_0 which lies in this interval. For a value $l < x_0$, we know that $f(l) \leq f(x_0)$. Similarly, for a value $u > x_0$, we know $f(u) \leq f(x_0)$. We can now show that $\lim_{l \to x_0} \frac{f(x_0)-f(l)}{x_0-l} \geq 0$. Similarly, we can show that $\lim_{u \to x_0} \frac{f(u)-f(x_0)}{u-x_0} \leq 0$. Both of these limits are approaching the derivative at $f(x_0)$. Therefore, $f(x_0) = 0$. Similarly, we can show that at local minima, we get the derivative to be zero.

Examining the two limits above, we conclude that in addition, the slopes (or more generally gradients) at points besides the local minima point away from the nearest minima. This intuition is the basis for the gradient descent algorithm. In gradient descent, we are interested in minimizing an objective function, and find the best parameters θ that minimizes this function $J(\theta)$. Here, we assume that we have a parameter vector θ whose components are $(\theta_0, \theta_1, \ldots, \theta_n)$, an n-dimensional parameter vector. When we look at a particular learning problem, we'll look at the specific form of this objective function. For now, this general formulation suffices.

We know that when $J(\Theta)$ is minimized, we must have complete gradient $\nabla J(\Theta) = 0$. In addition, we also get $\frac{\partial J(\Theta)}{\partial \theta_i} = 0$, where the partial derivative of the objective function $J(\Theta)$ is zero with respect to each of the components of Θ. Further, we also know that at points besides the minimum, the gradient points away from it. So, combining the above intuitions, we get a general algorithm that looks thus:

- Start with a random point $\Theta = \Theta_0$
- Compute the gradient $\nabla J(\Theta_0)$
- Take a step away from the direction of the gradient using a learning rate α. That is, set $\Theta = \Theta_0 - \alpha \nabla J(\Theta_0)$.
- Repeat the above steps till there is no change in the objective function.

This algorithm works for any objective function $J(\Theta)$ that can be differentiated. As we can surmise, gradient descent doesn't guarantee that you'll reach the global minimum. It'll

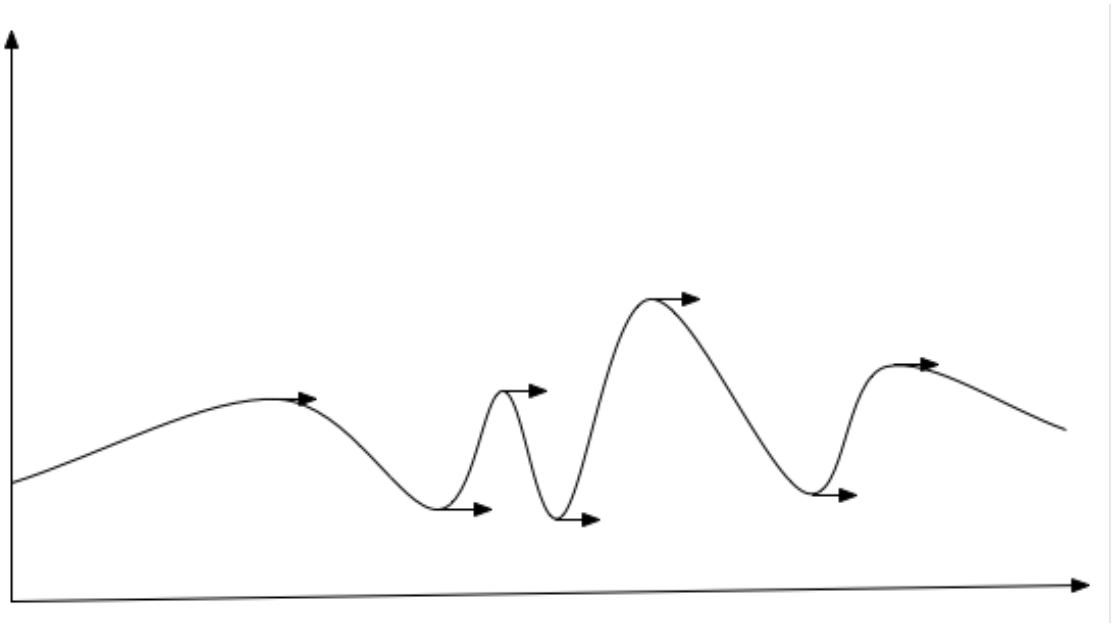

FIGURE 1. A simple 1-dimensional function with points where its first derivative is zero

converge to a local minimum. Further, small changes in your starting criteria can result in significantly different ending converged Θ estimates.

Now, let's look at some common problems that we have encountered in the past and see how this algorithm can be applied.

2. Gradient Descent applied to Linear Regression (or Ordinary Least Squares)

Let's start with Linear Regression. Let's assume that have an n dimensional feature space or n variables that are all predicting one response variable y. For instance, in the *auto* data, we have a 4-dimensional feature space or 4 variables all used to predict the target *mpg* value. In addition, we have M data points. And the objective is to minimize the regression error. That is, we want to find the best coefficients for each of the n variables so that the squared error is minimized. We saw earlier that this is the *Least Squares* solution when this squared error is minimized. The linear model takes the following form:

$$y = \Theta^T x \tag{1}$$

Here, we assume that θ_0 is the slope and correspondingly, we assume that $x_0 = 1$, a dummy variable. This allows us to write the linear equation $y = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$ in a convenient matrix form with the above assumption of $x_0 = 1$.

We can write the sum of squared error across all M samples as follows:

$$SSE = \sum_{i=0}^{M} [\Theta^T x(i) - y(i)]^2 \tag{2}$$

Where $x(i)$ is the ith data sample and it is $n+1$ dimensional: $x(i) = (1, x_1(i), \ldots, x_n(i))$, including the dummy variable. Let's consider $J(\Theta) = \frac{1}{2M} SSE$ as this results in some minor simplifications.

$$J(\Theta) = \frac{1}{2M} \sum_{i=0}^{M} [\Theta^T x(i) - y(i)]^2 \tag{3}$$

From this we get the partial derivative for jth dimension as

$$\frac{\partial J(\Theta)}{\theta_j} = \frac{1}{M} \sum_{i=0}^{M} (\Theta^T x(i) - y(i)) \cdot x_j(i) \tag{4}$$

where $x_j(i)$ is the jth component of the i sample. From this we get an update rule for linear regression as:

$$\theta_j = \theta_j - \alpha \frac{1}{M} \sum_{i=0}^{M} (\Theta^T x(i) - y(i)) \cdot x_j(i) \tag{5}$$

And here is the complete algorithm:

(1) Start with a random initial Θ. For example $\Theta = 0$. Set an initial learning rate α
(2) Update each $\theta_j = \theta_j - \alpha \frac{1}{M} \sum_{i=0}^{M} (\Theta^T x(i) - y(i)) \cdot x_j(i)$. It is important to note that all θ_j are updated in lock-step.
(3) Repeat steps 1 and 2 until $J(\Theta)$ has not converged (or up to some max iterations).

You may recall that we can find a closed-form solution via Ordinary Least Squares. That works well for reasonably small data sets. However, when the data set sizes grow, gradient descent becomes much more efficient compared with inverting matrices, which OLS requires.

3. Some variants of Gradient Descent

Gradient descent is the workhorse of many modern Machine Learning systems. However, as the data set size increases, the iterations become very expensive. Imagine if you have 100 million data points. These are not uncommon data set sizes today. Each iteration of the gradient descent requires a summation over these 100 million data points and can be very expensive.

One variation that is frequently used in Big Data situations is *Stochastic Gradient Descent*. Here, instead of computing the gradient over all the data points and then taking a step, the algorithm starts by randomly shuffling the data set. Then, it computes the gradient over each data point and takes a step right away to adjust the parameters. The advantage with this approach is that as the algorithm completes one pass over the data set, it would have taken many small steps to reduce the objective function. Then, an outer

loop is run for a handful of iterations and it is found that in many cases, SGD converges much more rapidly than regular gradient descent.

(1) Randomly shuffle the M data points
(2) For i = 1 to M, adjust each $\theta_j = \theta_j - \alpha(\Theta^T x(i) - y(i)) \cdot x_j(i)$.
(3) Repeat for K iterations

One frequent criticism with SGD is that its gradient estimates are very noisy as it takes only one data sample at a time. A simple alternative is to take m samples at a time, where $m << M$. This is called a *mini batch* SGD. Since we take several samples together, we get a much less noisy estimate of the gradient but also retain the efficiency of SGD in terms of speed-up for Big Data problems.