Gradient Descent Introduction

Carlos E. Carrillo-Gallegos

July 2022

1 Conceptual Overview

1.1 Introduction

Gradient Descent is an algorithm often used in Machine Learning to accurately estimate the parameters of a function that aims to model a specific training data set. For example, if you have a data set of NBA statistics, with every player, how many points per game they scored, and what their yearly salary was, you might want to predict a player's salary given their points per game (PPG). In this case, you could have a function, $f(x) = wx + b$, where x is the points per game, and $f(x)$ is predicting y, the salary. The gradient descent algorithm will attempt to find the optimal w and b values such that the model gives an accurate prediction. For a multi-parameter model (say you want to predict salary based on points, assists, games played, etc...), then the function would be $f(x_1, x_2, ...x_n) = w_1x_1 + w_2x_2 + ...w_3x_3$, and gradient descent would fit for n w parameters and b.

1.2 Methodology

So, how does gradient descent magically determine the appropriate parameters for best fit? It all comes down to calculus! For this discussion, we'll focus on single variable gradient descent, with $f(x) = wx + b$, as visualizing and conceptualizing this is much simpler.

First, the algorithm needs to be fed initial values for all w parameters and b. The accuracy of these is not incredibly important on their own, because they will be moved on from relatively quickly. Once the initial values are given, gradient descent computes $f(x)$ at every x value. In our NBA analogy, this means estimating the salary of every player in the data set using their PPG data. This estimate will likely be awful. That's okay for now!

Next, the cost function (or loss function) is computed. This starts by finding the difference between every $f(x^i)$, the estimate of the y values, and y^i, the real data set values. In the cost function, we square each difference and sum over all i values to get the sum cost. By convention, we also divide the sum cost by m, the number of entries in the data (in this example, the number of NBA players).

This function gives us a measure of how accurate our algorithm is with respect to the data we know to be true.

You might notice that the cost function is a function of the parameters, w, b. Indeed, the cost function will change with different choices for w, b. Given that we want the cost to be minimized, because that would indicate a more accurate estimate, it makes sense that we want to find the minimum of the cost function, $J(w, b)$. It might seem tempting to simply derive the function and set it equal to 0, and while that may work in simple cases, it would be inconclusive in a function with multiple local minima, which will almost certainly be the case for a cost function of decent complexity. In that case, we don't search for where the derivative is 0, but rather for where it is minimized. The algorithm will loop through iterations of w, b until it finds that $\frac{\partial J(x)}{\partial w}$ and $\frac{\partial J(x)}{\partial b}$ are at their minimums.

Each iteration ends with an update of the w and b values, subtracting the derivatives of the cost function divided by the number of data points, m. When subtracting, we'll also need to scale this term by the learning rate, α, which we will define as a small value, usually somewhere between 0.0001 and 0.1, depending on the specific case. This let's us determine how big the steps taken with every new iteration are. If we were to use no learning rate, or one that is too large, then gradient descent is likely to take extremely large steps and avoid nearing the minima entirely. If the learning rate is too small, gradient descent will take too many small steps and take too long to converge.

How do we know when we need to stop the iterations? Setting a convergence condition for $abs(w_{new} - w)$ and $abs(b_{new} - b)$ is usually a great method. This way, you ensure you get to a small enough step size such that the derivative of the cost function must be close to 0. You can also set a maximum number of iterations, say 1000, after which the code will terminate. This should also work for most simple implementations of gradient descent; if it does not converge in 1000 iterations, there may be an error with the code. Once the code has terminated, the model is defined by the final w and b values.

To give an example, let's return to the NBA data set. Say you are trying to estimate the salary of each player given their PPG. You choose an initial w and b for the initial $f(x)$ which you will use to get your first iteration of the cost function, which represents the difference between the model and the data set. Next, you take the derivative of the cost function, $J(x)$, with respect to every w and b value, and proceed to update each of those values with their respective derivatives. Lastly, you check to see if a convergence condition of a minimum change in the parameters or maximum iterations has been met. If so, the new model has been defined.

2 Mathematical Overview

In this section, we'll outline the algorithm more clearly and mathematically, to serve as a useful reference point. We'll showcase multi variable gradient descent.

Implementing Gradient Descent

X_1	X_2	X_3	Y
x_1^1	x_2^1	x_3^1	y^1
x_1^2	x_2^2	x_3^2	y^2
x_1^3	x_2^3	x_3^3	y^3
x_1^4	x_2^4	x_3^4	y^4
x_1^5	x_2^5	x_3^5	y^5

Table 1: Data Structure for Gradient Descent

First, let's look at the data. It's a small example here, but can be extended to any number of parameters. We define x_j as vectors of each parameters (PPG, rebounds, assists for the NBA example). Each vector x_j has that information for every data point (player) in the data set. Every x_j^i is an individual value of a parameter for a specific player. Y is the dependent variable we want to model. Given this information, we define our modeling function, $f(x)$, as follows:

$$f(x) = w_1x_1 + w_2x_2 + w_3x_3 + b \tag{1}$$

Where x are arrays of length 5, and ws and b are floats. Note that in practice, all w values are stored in a w array. With this, we want to compute the cost function, considering that there are m data points (in this case, 5) and n parameters (in this case, 3).

$$J(w,b) = \frac{1}{2m}\sum_{i=0}^{m}(f(x^i) - y^i)^2 \tag{2}$$

Note that $f(x^i)$ takes as input all values of x at level data point i. For example, it's like taking in a player's points, rebounds, and assists.

We need the derivative of the cost function to update the w, b values. We derive the cost function here 4 times, once for each value of w (corresponding to a value of x) and once for b. In a more general case, the derivative is computed $n + 1$ times.

For j = 1,2,3....:

$$\frac{\partial J(x)}{\partial w_j} = \frac{1}{m}\sum_{i=0}^{m}(f(x^i) - y^i)x_i \tag{3}$$

For b...:

$$\frac{\partial J(x)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m}(f(x^i) - y^i) \tag{4}$$

Now that we have the derivatives, we can update w and b with these values, given that we choose a learning rate α.

For j = 1,2,3...:

$$w_{j,new} = w_{j,old} - \alpha\frac{\partial J(x)}{\partial w_j} \tag{5}$$

For b...:

$$b_{new} = b_{old} - \alpha\frac{\partial J(x)}{\partial b} \tag{6}$$

This process will keep iterating until the convergence conditions are met. Once they are, then you can take the final w and b values to be the ideal values for the model $f(x)$.

Looking forward, there are many different variations of gradient descent used for a wide array of applications. The foundations of the algorithm are the same, however.