Gradient Descent

From Deep Learning Course Wiki

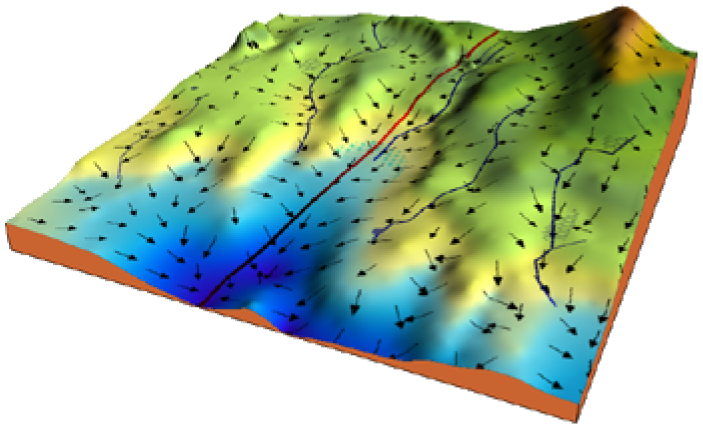
Contents

Gradient Descent

Gradient Descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in linear regression and weights in neural networks.

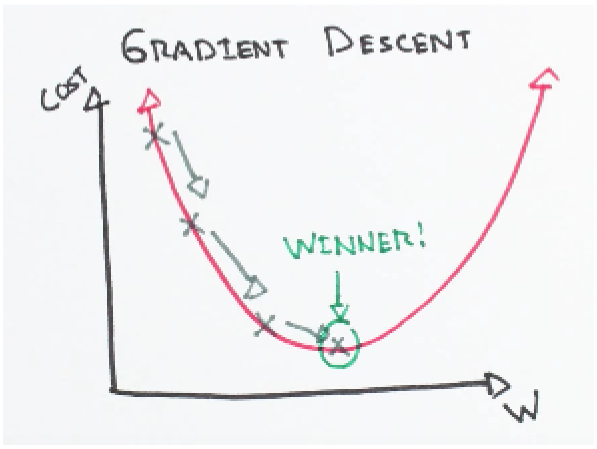
How it works

Consider the 3-dimensional graph below in the context of a cost function. Our goal is to move from the mountain in the top right corner (high cost) to the dark blue sea in the bottom left (low cost). The arrows represent the direction of steepest descent (negative gradient) from any given point--the direction that decreases the cost function as quickly as possible.



Source (http://www.adalta.it/Pages/-GoldenSoftware-Surfer-010.asp)

Starting at the top of the mountain, we take our first step downhill in the direction specified by the negative gradient. Next we recalculate the negative gradient (passing in the coordinates of our new point) and take another step in the direction it specifies. We continue this process iteratively until we get to the bottom of our graph, or to a point where we can no longer move downhill--a local minimum.



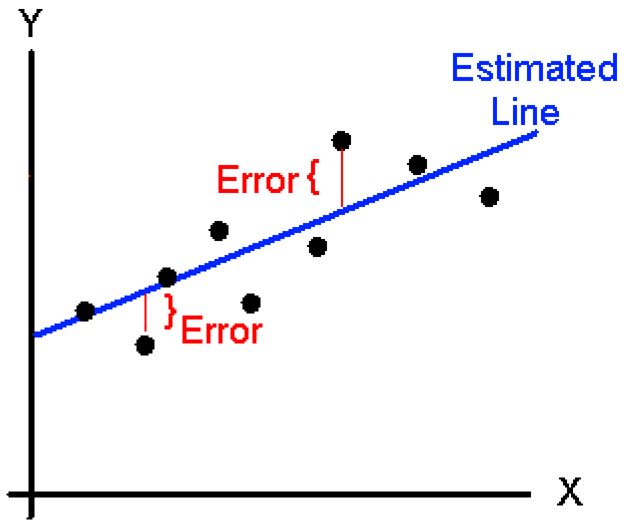
Great Video on Gradient Descent (https://youtu.be/5u0jaA3qAGk)

Learning Rate

The size of these steps is called the **learning rate**. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

Cost Function

A cost function is a wrapper around our model function that tells us "how good" our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to change our parameters to make the model more accurate! We use the model to make predictions. We use the cost function to update our parameters. Our cost function can take a variety of forms as there are many different cost functions available. Popular cost functions include: Mean Squared Error, Root Mean Squared Error, and Log Loss.



source (http://www.ken-szulczyk.com/misc/statistical_lecture_10.php)

Let's take an example from linear regression where our model is $f(x) = mx + b$, where m and b are the parameters we can tweak.

$$y = mx + b$$

Formula

Let's use Mean Squared Error as our cost function:

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

- N is the total number of observations (data points)
- $\frac{1}{N}\sum_{i=1}^{n}$ is the mean
- y_i is the actual value of an observation and $(mx_i + b)$ is our prediction

Code

Calculating the mean squared error in python.

```
# MSE for y = mx + b
def cost_function(x, y, m, b):
    N = len(x)
    total_error = 0.0
    for i in range(N):
        total_error += (y[i] - (m*x[i] + b))**2
    return total_error / N
```

Algorithm

Now let's run gradient descent using our new cost function. There are two "parameters" (i.e. weights) in our cost function we can control: m and b. Since we need to consider the impact each one has on the final prediction, we need to use partial derivatives. We calculate the partial derivatives of the cost function with respect to each parameter and store the results in a gradient.

Formula

Given the cost function:

$$f(m,b) = \frac{1}{N}\sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

The gradient of this cost function would be:

$$f'(m,b) = \begin{bmatrix} \frac{df}{dm} \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N}\sum -2x_i(y_i - (mx_i + b)) \\ \frac{1}{N}\sum -2(y_i - (mx_i + b)) \end{bmatrix}$$

To solve for the gradient, we iterate through our data points using our new m and n values and compute the partial derivatives. This new gradient tells us the slope of our cost function at our current position (i.e. parameters) and the direction we should move to update our parameters. The size of our update is controlled by the learning rate.

Code

Example python code for finding the optimal local minimum.

```python
def update_weights(m, b, X, Y, learning_rate):
    m_deriv = 0
    b_deriv = 0
    N = len(X)
    for i in range(N):
        # Calculate partial derivatives
        # -2x(y - (mx + b))
        m_deriv += -2*X[i] * (Y[i] - (m*X[i] + b))

        # -2(y - (mx + b))
        b_deriv += -2*(Y[i] - (m*X[i] + b))

    # We subtract because the derivatives point in direction of steepest ascent
    m -= (m_deriv / float(N)) * learning_rate
    b -= (b_deriv / float(N)) * learning_rate

    return m, b
```

Convergence

The above algorithm calculates the gradient for one set of m and b parameters. To find the best possible parameters we need to run our gradient descent algorithm through multiple iterations. But when do we stop?

[INSERT EXPLANATION OF STOPPING POINT}

Batch Gradient Descent

Stochastic Gradient Descent

SGD is an optimisation technique - a tool used to update the parameters of a model. Most optimisation techniques (including SGD) are used in an iterative fashion: The first run adjusts the parameters a bit, and consecutive runs keep adjusting the parameters (hopefully improving them).

Why use SGD?

SGD is an optimisation technique. It is an alternative to *Standard* Gradient Descent and other approaches like batch training or BFGS. It still leads to fast convergence, with some advantages:

- Doesn't require storing all training data in memory (good for large training sets)

- Allows adding new data in an "online" setting

How is it different to other techniques?

Minibatches

Rather than evaluating a "cost function" over the entire training set (as in *Standard* Gradient Descent), SGD uses a subset of the training data (a minibatch). The subset that gets used should change each iteration, and it should be selected randomly each iteration.

The number of datapoints to use in the minibatch is not set in stone, the considerations are:

Too small

- Slow convergence (the parameters may jump around a lot, rather than smoothly approaching the optimum outcome).

- The model parameters and the minibatch data can be stored in matrices. There are some efficiency wins by doing smart matrix operations, so having a small minibatch means we aren't making the most of these smart matrix operations.

Too big

- The speed advantages get lost as more of the training data gets used.

Learning rate

Optimisation techniques often have a variable or parameter called a learning rate. The learning rate specifies how aggressively the optimisation technique should jump between each iteration.

- If the learning rate is too small, the optimisation will need to be run a lot of times (taking a long time and potentially never reaching the optimum).
- If the learning rate is too big then the optimisation may be unstable (bouncing around the optimum, and maybe even getting worse rather than better).

Because SGD uses a subset of all the training data, there will be more variance in the parameters for each iteration than batch optimisation techniques. Because of this the learning rate should be set to be smaller than the learning rate for batch techniques.

Dynamic Learning Rates

Initially a model's parameters may be far from their optimum values. That means a high learning rate can be used, to make big jumps in the right direction. As the parameters get closer to their optimum values, big jumps in a parameter's value may actually jump past the optimum. In this case the optimum parameter values may never be reached. One solution to this is to change what the learning rate is as the parameters are optimised. There are a range of ways this can be done, here are a couple of examples:

- Use a large learning rate for the first couple of iterations, and then shrink the learning rate for each consecutive iteration.

- Use a held-out set to assess learning:

1. Hold a set of the training data out of the optimisation process.
2. Once an iteration is complete, run the held-out data through the model and look at the performance of the model.
3. If the performance improvement between iterations has dropped below a certain threshold, shrink the learning rate.

- "backtracking line search"???

See dynamic learning rates for more info.

Momentum

As parameters approach a local optimum, the improvements can slow down, taking a long time to finally get to the local optimum. Adding "momentum" to the optimisation technique can help to keep improving the model parameters towards the end of the optimisation process. Momentum will look at how the parameters were changing over the last few iterations, and use that information to keep moving in the same direction. The number of prior iterations to take into account changes as the initial learning stabilises.

Tutorials

- Gradient Descent Demystified (Excellent) (https://www.youtube.com/watch?v=5u0jaA3qAGk)
- Gradient and stochastic gradient descent (https://www.youtube.com/watch?v=WnqQrPNYz5Q)

References

- Gradient Descent Demystified (https://www.youtube.com/watch?v=5u0jaA3qAGk)
- http://sebastianruder.com/optimizing-gradient-descent/
- Unsupervised Feature Learning and Deep Learning Tutorial (http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/)
- https://www.coursera.org/learn/machine-learning/supplement/2GnUg/gradient-descent
- http://machinelearningmastery.com/gradient-descent-for-machine-learning/
- https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/