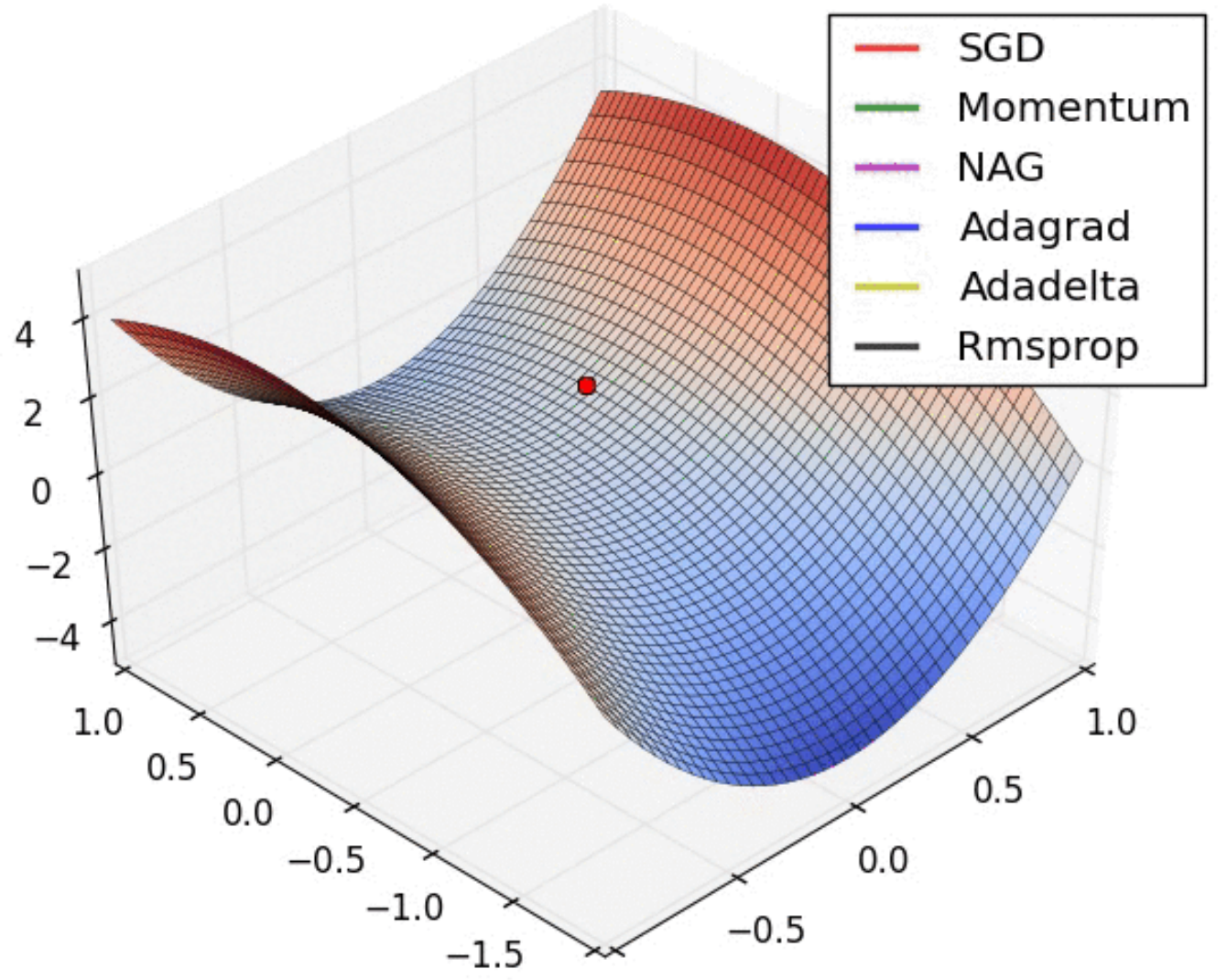
Optimization Algorithms in Neural Networks

This article presents an overview of some of the most used optimizers while training a neural network.

By **Nagesh Singh Chauhan**, KDnuggets on December 18, 2020 in **Gradient Descent**, **Neural Networks**, **Optimization**

comments



Credits

Introduction

In deep learning, we have the concept of loss, which tells us how poorly the model is performing at that current instant. Now we need to use this loss to **train** our network such that it performs better. Essentially what we need to do is to take the loss and try to **minimize** it, because a lower

loss means our model is going to perform better. The process of minimizing (or maximizing) any mathematical expression is called **optimization.**

Optimizers are algorithms or methods used to change the attributes of the neural network such as **weights** and **learning rate** to reduce the losses. Optimizers are used to solve optimization problems by minimizing the function.

How do Optimizers work?

For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on. It's impossible to know which direction to go in, but there's one thing she can know: if she's going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she'll reach the base.

Similarly, it's impossible to know what your model's weights should be right from the start. But with some trial and error based on the loss function (whether the hiker is descending), you can end up getting there eventually.

How you should change your weights or learning rates of your neural network to reduce the losses is defined by the optimizers you use. Optimization algorithms are responsible for reducing the losses and to provide the most accurate results possible.

Various optimizers are researched within the last few couples of years each having its advantages and disadvantages. Read the entire article to understand the working, advantages, and disadvantages of the algorithms.

We'll learn about different types of optimizers and how they exactly work to minimize the loss function.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini Batch Stochastic Gradient Descent (MB-SGD)
4. SGD with momentum
5. Nesterov Accelerated Gradient (NAG)

6. Adaptive Gradient (AdaGrad)
7. AdaDelta
8. RMSprop
9. Adam

Gradient Descent

Gradient descent is an optimization algorithm that's used when training a machine learning model. It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.

WHAT IS GRADIENT DESCENT? Gradient Descent is an optimization algorithm for finding a local minimum of a differentiable function. Gradient descent is simply used to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible.

You start by defining the initial parameter's values and from there gradient descent uses calculus to iteratively adjust the values so they minimize the given cost-function.

The weight is initialized using some initialization strategies and is updated with each epoch according to the update equation.

$$\text{Repeat until convergence } \{$$

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$

The above equation computes the gradient of the cost function **J(θ)** w.r.t. to the parameters/weights **θ** for the entire training dataset:

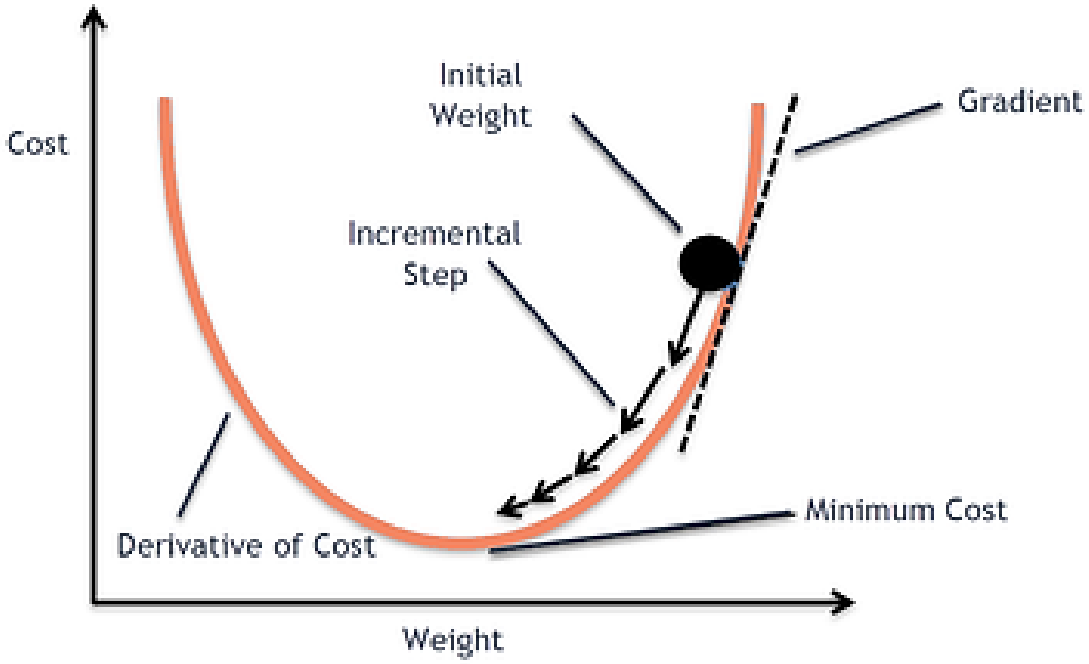


Image source

Our aim is to get to the bottom of our graph(Cost vs weights), or to a point where we can no longer move downhill–a local minimum.

Okay now, what is Gradient?

"A gradient measures how much the output of a function changes if you change the inputs a little bit." —Lex Fridman (MIT)

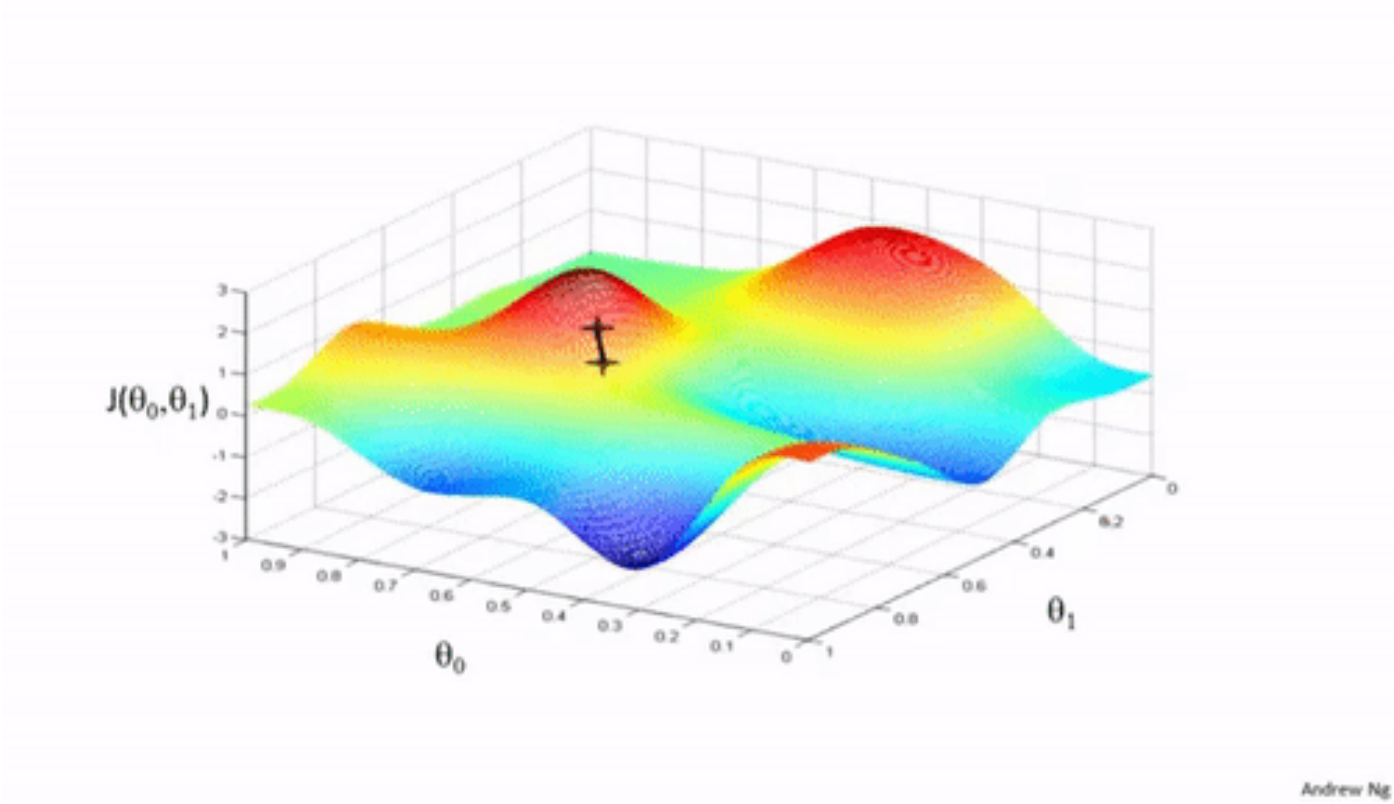


Image source

Importance of Learning rate

How big the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slows we will move towards the optimal weights.

For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image).

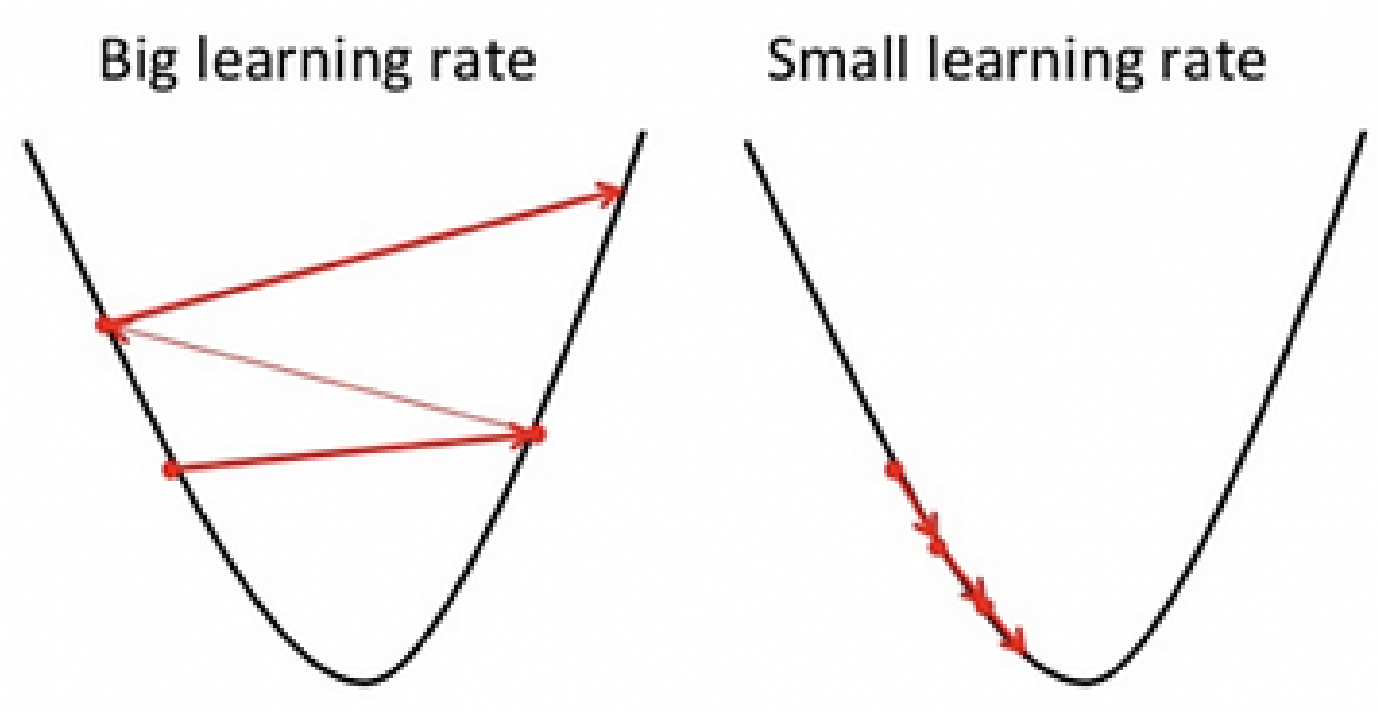


Image Source

So, the learning rate should never be too high or too low for this reason. You can check if you're learning rate is doing well by plotting it on a graph.

In code, gradient descent looks something like this:

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
```

```
    params = params - learning_rate * params_grad
```

For a pre-defined number of epochs, we first compute the gradient vector params_grad of the loss function for the whole dataset w.r.t. our parameter vector params.

Advantages:

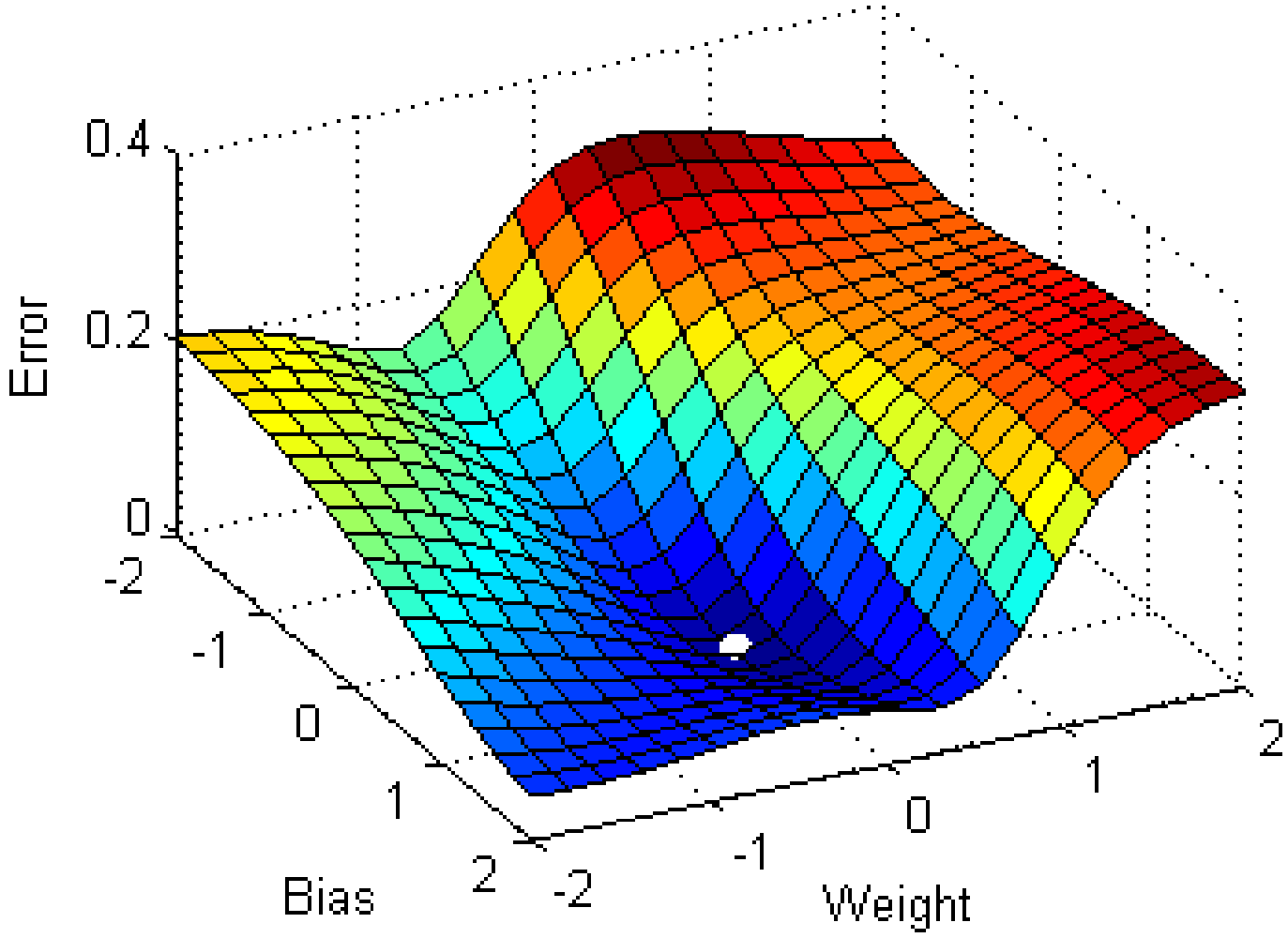
1. Easy computation.
2. Easy to implement.
3. Easy to understand.

Disadvantages:

1. May trap at local minima.
2. Weights are changed after calculating the gradient on the whole dataset. So, if the dataset is too large then this may take years to converge to the minima.
3. Requires large memory to calculate the gradient on the whole dataset.

Stochastic Gradient Descent (SGD)

SGD algorithm is an extension of the Gradient Descent and it overcomes some of the disadvantages of the GD algorithm. Gradient Descent has a disadvantage that it requires a lot of memory to load the entire dataset of n-points at a time to compute the derivative of the loss function. **In the SGD algorithm derivative is computed taking one point at a time.**



Stochastic Gradient Descent Image Source

SGD performs a parameter update for *each* training example **x(i)** and label **y(i)**:

$$\theta = \theta - \alpha \cdot \partial(J(\theta; x(i), y(i)))/\partial\theta$$

where **{x(i) ,y(i)}** are the training examples.

To make the training even faster we take a Gradient Descent step for each training example. Let's see what the implications would be in the image below.

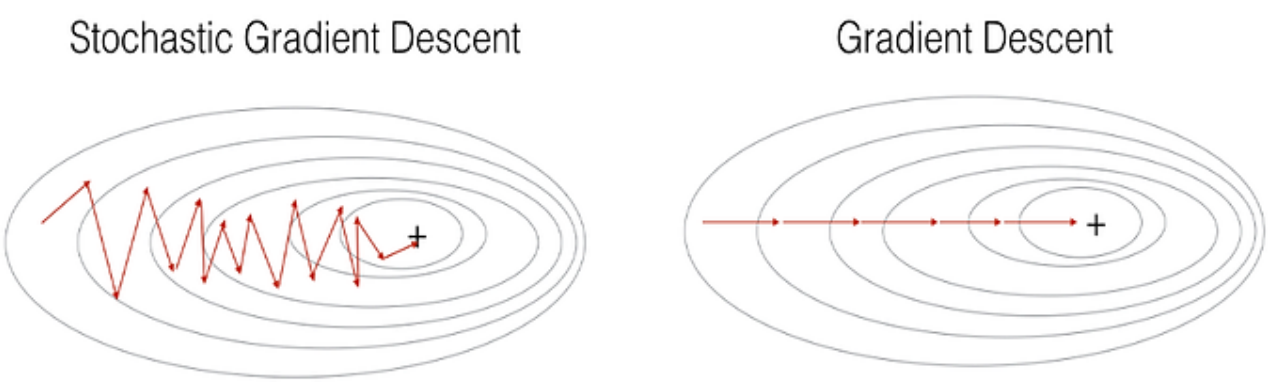


Figure 1 : SGD vs GD

"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

Image Source

1. On the left, we have Stochastic Gradient Descent (where m=1 per step) we take a Gradient Descent step for each example and on the right is Gradient Descent (1 step per entire training set).
2. SGD seems to be quite noisy, at the same time it is much faster but may not converge to a minimum.
3. Typically, to get the best out of both worlds we use Mini-batch gradient descent (MGD) which looks at a smaller number of training set examples at once to help (usually power of 2 - 2^6 etc.).
4. Mini-batch Gradient Descent is relatively more stable than Stochastic Gradient Descent (SGD) but does have oscillations as gradient steps are being taken in the direction of a sample of the training set and not the entire set as in BGD.

It is observed that in SGD the updates take more number iterations compared to gradient descent to reach minima. On the right, the Gradient Descent takes fewer steps to reach minima but the SGD algorithm is noisier and takes more iterations.

Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example.

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

Advantage:

Memory requirement is less compared to the GD algorithm as the derivative is computed taking only 1 point at once.

Disadvantages:

1. The time required to complete 1 epoch is large compared to the GD algorithm.
2. Takes a long time to converge.
3. May stuck at local minima.