Summer 2019

Optimization of Mathematical Functions Using Gradient Descent Based Algorithms

Hala Elashmawi

OPTIMIZATION OF MATHEMATICAL FUNCTIONS USING GRADIENT DESCENT BASED ALGORITHMS

by
Hala Elashmawi
B.S. in Architecture, College of Engineering, Mansoura University, Egypt, 1998

THESIS

Submitted in Partial Fulfillment of the Requirements

For the Degree of Master of Science,
With a Major in Mathematics

Governors State University
University Park, IL 60484

2019

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

Optimization problem involves minimizing or maximizing some given quantity for certain constraints. Various real-life problems require the use of optimization techniques to find a suitable solution. These include both, minimizing or maximizing a function. The various approaches used in mathematics include methods like Linear Programming Problems (LPP), Genetic Programming, Particle Swarm Optimization, Differential Evolution Algorithms, and Gradient Descent. All these methods have some drawbacks and/or are not suitable for every scenario. Gradient Descent optimization can only be used for optimization when the goal is to find the minimum and the function at hand is differentiable and convex. The Gradient Descent algorithm is applicable only in the case stated above. This makes it an algorithm which specializes in that task, whereas the other algorithms are applicable in a much wider range of problems. A major application of the Gradient Descent algorithm is in minimizing the loss functions in machine learning and deep learning algorithms. In such cases, Gradient Descent helps to optimize very complex mathematical functions. However, the Gradient Descent algorithm has a lot of drawbacks. To overcome these drawbacks, several variants and improvements of the standard Gradient Descent algorithm have been employed which help to minimize the function at a faster rate and with more accuracy. In this paper, we will discuss some of these Gradient Descent based optimization algorithms.

Keywords: Optimization, Gradient Descent, Convex optimization.

OPTIMIZATION OF MATHEMATICAL FUNCTIONS USING GRADIENT DESCENT BASED ALGORITHMS

1. Introduction

Many real-life problems involve modelling a real-life problem as mathematical functions, along with certain constraints with the goal of optimizing it. These optimization problems require the user to either minimize or maximize a given function. Some optimization problems also have certain constraints.

Optimization can be simply viewed as selecting certain inputs which result in best outputs or the best possible outputs that can be achieved. These optimization problems may also include certain constraints. A number of traditional and non-traditional approaches have been employed for this task. Some of them are specialized for certain cases, whereas others serve a general purpose which work in many scenarios. As an example, some of these approaches can be used for minimizing as well as maximizing a function. When it comes to constrained optimization, not all of these algorithms are applicable. The goal of discussion here is optimizing a function to find its minimum, when we know that the function is convex in nature.

Gradient Descent algorithm as the name suggests, works on the principle of descending along the direction given by gradient to attain the minimum value. However, sometimes, this may lead to local minima instead of global minima. This method also encounters other issues which will be discussed later. We will also review the techniques used for overcoming these drawbacks and then present the various modified versions of the Gradient Descent algorithm which were implemented to overcome these drawbacks.

Gradient Descent and its various modified forms as presented in [3], give an overview of all the optimization algorithms based on gradient descent. The basic idea behind the Gradient

Descent algorithm is to minimize the value of a convex function by moving along the gradients. This involves calculating the direction of the gradients from the given point and then taking a step along the direction of steepest descent. As a simple explanation, Gradient Descent can be considered as moving down the hill starting from any point. If we keep moving along the steepest downward slope, we will eventually reach the minimum value. This approach works well unless there is a saddle point or local minima, in which case, the algorithm may reach the wrong optimum value. These issues are discussed further in this paper in later sections. These problems associated with standard Gradient Descent led to the development of various other optimization algorithms which are based on gradient descent, which are discussed in later sections of this paper.

2. Gradient Descent

The Gradient Descent algorithm is an optimization algorithm for minimizing convex functions. It works on the principle that if we continue to move in the direction of the steepest descent from a particular starting point which may be randomly initialized, then we reach the minimum of the function, provided that the function to be minimized is differentiable at every point and convex. The direction of steepest descent is calculated by taking the negative of the gradient of the function, evaluated at that point. This can be simply viewed as finding the minimum altitude point in a geographical plane. If we continue to move along the direction having the steepest descent, then we will eventually reach the point at lowest altitude. The gradient of the function is taken with respect to the parameters we are updating. A convex function can be roughly considered as an 'U' shaped function with the bottom of the curve representing the minimum of the function. If we start at any point on this 'U' shaped function, we can see that the steepest descent at any point always points towards the bottom minimum of the function. We will see this in more detail in below.

If the function $J(\theta)$ is a convex function to be minimized, where θ is the set of parameters on which the function is dependent and θ_i is the *i-th* coordinate of the parameter, then we start by randomly initializing the values of θ_i. This value of θ_i gives us a starting point on the curve of the function. Now following the Gradient Descent algorithm, the next step is to find the direction of steepest descent so that we can move towards the minimum. This is done by finding the derivative of the function with respect to θ, which can be used to minimize $J(\theta)$. This derivative with respect to θ can be represented as a vector with i values. We take these derivatives by partially differentiating the function with respect to each component and treating the rest of these variables as constant. If we visualize the function, it would be a $(i+1)$ dimensional curve, with each θ_i representing an axis and the $i+1$st dimension would

represent the value of function. So, we can say that 'i' dimensions are the input of the function and the $(i+1)$st dimension is the output. The vector formed by the derivatives of the *i* input variables, represents the direction of steepest ascent, that is, it represents the direction in which the function increases at the maximum rate from that point. Reversing the direction of this vector gives the vector pointing to the direction of steepest descent. This direction represented by the reversed vector is the direction we move along, so as to reach the minimum value of the function. However, this reversed vector only gives the direction in which we are supposed to move. The magnitude by which we are supposed to take a step in that direction is calculated by multiplying the magnitude of the gradient in that direction by a constant called the learning rate.

This learning rate is a constant value, which is the same for updates along every direction. The learning rate is fixed through the entire algorithm. The use of the learning rate is to avoid overstepping in a particular direction, where the point leading to the actual minima is skipped.

If $\nabla_\theta J(\theta)$ is the gradient of $J(\theta)$ with respect to θ, then we update the variables in every iteration by using,

$\theta_i \leftarrow \theta_i - lr * \nabla_\theta J(\theta)$.

Here, *lr* is the learning rate and the negative sign reverses the direction of the gradients to get the direction of maximum descent.

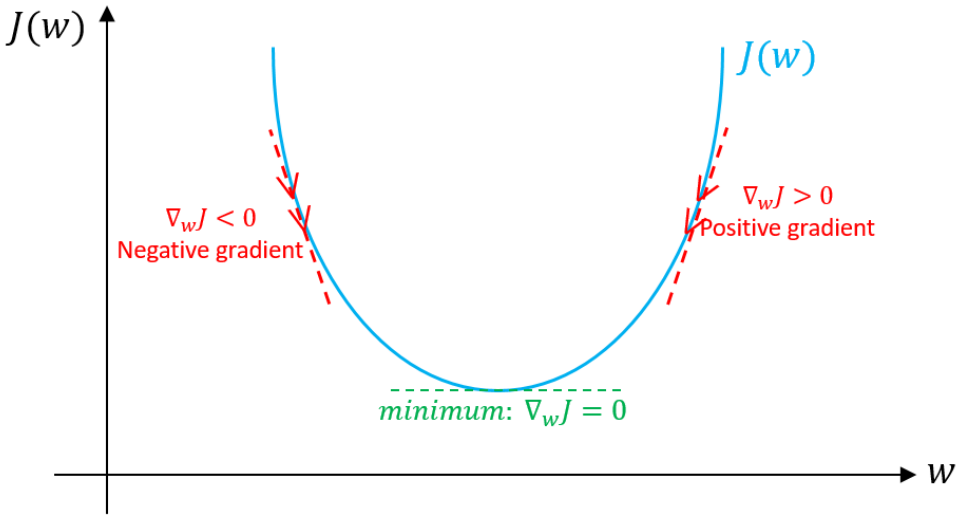

Figure (1) Shows the sign of derivatives for a function at various points. Source - (https://miro.medium.com/max/1400/1*jNyE54fTVOH1203IwYeNEg.png)

Figure (1) visualizes the Gradient Descent algorithm and shows how the gradient changes across the minima and the direction in which the algorithm moves. In this image, the parameters θ are represented by W. Gradient Descent works on convex functions because they have a unique minimum. Consider the function $J(\theta)$. Suppose that J has a local minimum at θ and also another local minimum at Φ with the condition that

$J(\theta) \leq J(\Phi)\ ;\ \theta \neq \Phi$. (equation 1)

The definition of strict convexity is,

$$J(h * \theta + (1 - h) * \Phi) < h * J(\theta) + (1 - h) * J(\Phi)\ ;\ 0 < h < 1$$

Since h is positive,

$$J(\theta) \leq J(\Phi) \Rightarrow h * J(\theta) \leq h * J(\Phi),$$

which justifies the condition below,

$h * J(\theta)+(1-h) * J(\Phi) \leq h * J(\Phi) + (1-h) * J(\Phi)$.

This gives,

$h * J(\theta)+(1-h) * J(\Phi) \leq J(\Phi)$.

Replacing this condition with the definition of convexity,

$J(h * \theta+(1-h) * \Phi) < J(\Phi)$.

Since $\lim_{h \to 0} h\Theta + (1-h)\Phi = \Phi$, for sufficiently small h, there is an $x = h\Theta + (1-h)\Phi$ that is arbitrarily close to Φ for which we have $J(x) < J(\Phi)$. However, this contradicts the minimality of $J(\Phi)$. Thus it must be that $\theta = \Phi$, which shows that J has at most one local minimum.

a. Impact of initialization on gradient descent

After seeing how the Gradient Descent algorithm works, now it's time to discuss some important factors related to the algorithm which may affect its performance. While starting the algorithm, the value of the input variables θ is randomly initialized. However, this random initialization can lead to the algorithm failing to optimize the given function.

Suppose, there is a function with many minima. The local minima are just minimum values when compared to their surrounding values. When a function is optimized to find its minimum value, the goal is to reach the global minimum, but the random initialization of the points in the first step of the algorithm may result in Gradient Descent converging to a local minimum.

As an example, consider the image shown below in Figure (2), it shows the graph of a random hypothetical function which is dependent on only one variable. The graph of the function can be represented in two dimensions. The x-axis in the figure represents the input variable θ and the y-axis in the figure represents the value of the function J, at that point. As can be seen, this function has two minima, the left minimum is the global minimum and the desired result after optimization, and the right minimum is just a local minimum. Four points are marked on this graph using different colors.

Suppose the variable is initialized to get the function value at the blue point; the direction of steepest descent at this point is towards the left side. Hence, the Gradient Descent algorithm would optimize the function to achieve the minimum value shown by the green point, and this indeed is the actual global minimum of the function. However, if the variable is initialized to get the point indicated by red point, the steepest descent is towards the right side. Following this rule of steepest descent, we would reach the value indicated by orange point as the minimum. However, this orange point is a local minimum and not the actual

optimum. In fact, any initialization resulting in the value of function to the right of the peak between blue and red point would result in the same local minima indicated by the orange point. This is a drawback of the algorithm, as the solution given by the algorithm does depends upon the initialization of the variables. A proper initialization would result in the actual global minima or the optimum value we were looking for, however a wrong initialization would result in wrong optimized value. However, if other factors are working fine, we will reach either a global or local minima in any case.

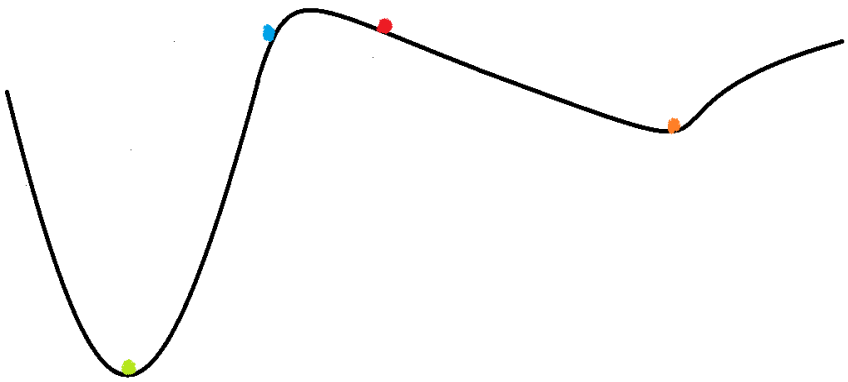

Figure (2) Graph of a function with four points marked having different gradients

b. Impact of the learning rate

The learning rate is a constant value which determines the length of the step the function is supposed to take while following the direction of steepest descent. The value of the learning rate can greatly impact the results of the Gradient Descent algorithm. In the conventional implementation of gradient descent, the value of learning rate is kept constant throughout the entire algorithm.

Why is the learning rate used? After calculating the gradient and reversing its direction, the direction of steepest descent from that point is reached. However, how much is the point supposed to move along this direction? Should it continue to move forever along this same

direction until we reach the minimum value? This problem is solved by the concept of learning rate. If we take a closer look at the function from the above example, we can see that the direction to the steepest descent in the function graph, changes after some significant interval. This implies that as soon as the point moves along the direction maximum descent, the direction to maximum descent at this new point has changed. And for a convex continuous function, this value may change at every infinitely small change in the parameters or the input variables. This problem of choosing the step size at every point so as to reach the minimum value is solved by learning rate. The learning rate multiplied by the gradient gives the updated value to which we are supposed to move. Then from this next point, the direction and the step size can be calculated. The learning rate can be considered as the rate at which the algorithm moves towards the minimum value.

Choosing an appropriate value for learning rate is important. If there is an extremely large value of learning rate, then it may happen that the algorithm never reaches the optimum value and just keeps bouncing over the minima. This can even lead to the function value increasing as the algorithm goes on, in case the learning rate is way too large.

Similarly, a learning rate that is extremely small isn’t good either as it can also lead to poor results (as gradient descent never reaches the minima, in case we define the algorithm to end after certain fixed number of steps) or unnecessarily large number of computation steps. A learning rate with a too small value would result in the algorithm, converging really slowly and it may happen that the algorithm never even reaches the minimum value and just gets stuck at a point which is really far from the desired value.

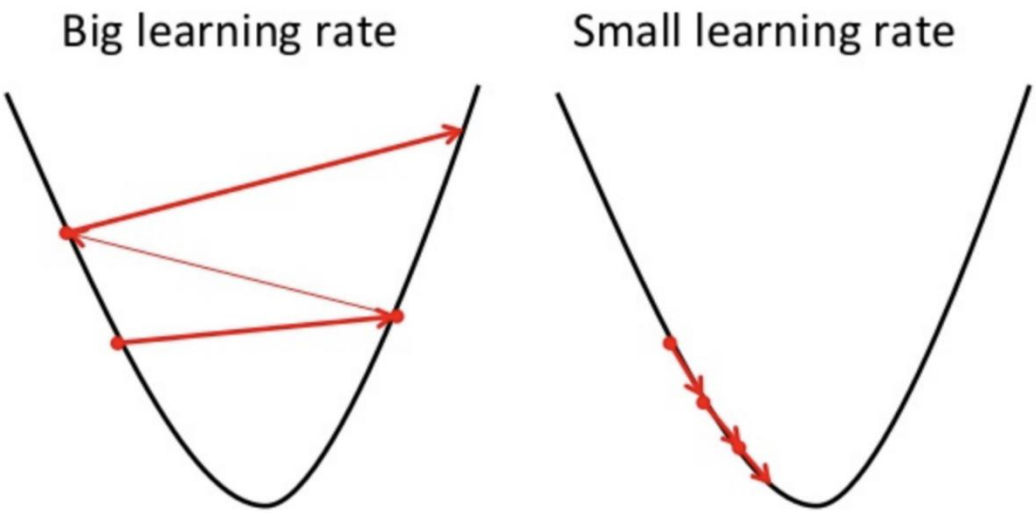

Figure (3) A image comparing Gradient Descent iteration on same function using big and small learning rate. Source - (https://miro.medium.com/max/1400/0*QwE8M4MupSdqA3M4.png)

Figure (3) represents the effect of learning rate while minimizing a simple 'U' shaped function using gradient descent. As we can see, in the first part, a large value of learning rate actually resulted in diverging away from the minima. And in the second part of the image, it resulted in updated being really small and not reaching the minimum value. Hence, it's important to choose the learning rate for the algorithm carefully or it may result in a non-optimum value.

c. Number of iterations

The number of iterations for the Gradient Descent algorithm refers to the number of times the value of the parameters or the input variables is updated to converge to the optimum value. It depends on the complexity of the function to be minimized and the distance of the initial random initialization from the optimum value. If the initialization was made close to the

minimum value, the optimum value can be reached using a small number of iterations. However, for more complex functions with initial value far from the optimum value, more iterations are needed. Hence, it is always a good idea to have a sufficiently large number of iterations, because the optimum value is reached, then further iteration would keep the value close to the minimum as the gradients become too small and almost zero near the minima, hence, the updates are small too.

After seeing how Gradient Descent works and the various aspects of this algorithm, now it is time to see how the problems mentioned above and other problems related to this algorithm are resolved.

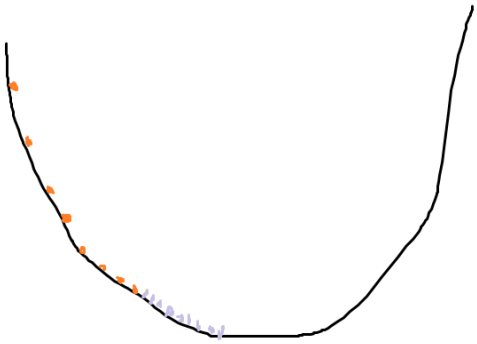

Figure (4) Figure showing the movement of Gradient Descent with red points for the first few iterations and blue for high number of iterations

Figure (4) shows the effect of number of iterations. If a small number of iterations is used, then the algorithm may stop at a point before reaching the minimum, which is shown by the orange points, whereas the purple points represent how the convergence would have continued for a sufficiently large value of number of iterations, which can be seen, successfully reaches the minimum of the function.

d. Solving random initialization problem:

In above section it was observed that the random initialization in the beginning of the algorithm can sometimes result in it converging to a local minimum. This problem can be solved by repeating the entire algorithm n number of times, using different random initialization from throughout the domain of the function. If n initialization points are used, which are randomly spread throughout the domain, then after running the gradient descent, if the function to be minimized has more than one minimum, then every initialization would converge to the minimum which is followed by the steepest descent from that point. The final optimum value could be the minimum of all these optimum values. A uniform probability distribution can be used to generate the initial random values and then optimize them using Gradient Descent and finally select the minimum of all the results [2]. The number of initializations to test depends on the task. If the function is highly complicated or has a high degree, then it's possible that it may have a large number of local minima. In this case, the value of n should be large as the chances of the result getting stuck to the local minima is very high. For fairly simple function, we can keep the value of n small.

If $J(\theta)$ is the function to be minimized and $\mathrm{opt}(J(\theta))$ represents the optimum value of the function $J(\theta)$ with an initialization of θ using Gradient Descent then,

Optimum value = min(opt($J(\theta^i)$)) for i in range 1 to n,

where, θ^i represents a sample from uniform probability distribution.

However, even this does not guarantee that the optimum value achieved would be the global minimum; this method just increases the chances of getting a global minima. We can never be really sure about the global minima, if the function to be optimized has a very high degree.

e. More effective learning rate:

One major issue with Gradient Descent algorithm is choosing a learning rate to descend in the direction of steepest slope. Having too large or small of a learning rate can prevent the algorithm from converging to the right point.

A solution to this problem is to modify the technique to decrease the learning rate as we get closer to the minimum. This is a modification done to make gradient descent work better. However, it is not known where the minima are until the algorithm converges. But it is known that the point in focus keeps on moving closer to the minimum value as the algorithm progresses, provided that we don't have a very large learning rate. This can be done by decreasing the learning rate with each iteration. This would lead to the algorithm taking smaller and smaller step as it reaches the minimum value. Various methods for decaying the learning rate with the passing iteration can be used. For example, we can decay the learning rate based on the number of iterations, the new learning rate decays with increasing number of iterations.

$$lr_t = \frac{k}{t}$$

Here, lr_t is the learning rate at iteration t and k represents a constant which is also the initial learning rate.

f. Other drawbacks with gradient descent:

Despite the solutions discussed above to some of the problems faced by gradient descent, there are still other problems which persist and are not solved by the methods discussed above. These problems are mainly associated with high dimensional functions and saddle points.

g. High dimensional function:

If the function to be optimized is a high dimensional function, that is, a function which is dependent on many variables, and the conventional Gradient Descent algorithm is used, the same learning rate will be applied to all components at once, as a vector, in every step. Even if decaying learning rate is used, the learning rate used at one particular step would be same for all directions.

But it may be the case that some components of variable affect the function at a far greater range than other variables. As in a slight change in some variable may result in a large change in the function, whereas, a drastic change in certain variables may lead to negligible change in the function. This may be because the variable affecting the function by greater amount may have a larger coefficient or the function might be dependent on higher degree for this variable. If that is the case, we might want to update the variables with different effects on the function by different learning rates. This is one major problem with the Gradient Descent algorithm.

h. Saddle points

A saddle point, also known as minimax point, is a point on the surface of the graph of a function where the derivatives or the slopes of the function in orthogonal directions are all zero, but this point is not a local extremum of the function. At a saddle point, the various directions disagree about whether the point is a minima or maxima. So even though the point is a stable point, and is not an inflection point, it cannot be a local minimum or a local

maximum. When we apply Gradient Descent algorithm at this point, the algorithm actually gets stuck at this point with no further update. This leads to wrong convergence.

To solve the above-mentioned problem with the Gradient Descent algorithm, various modified algorithms have been made available. This makes these modified versions of Gradient Descent work more efficiently and effectively. All these modified algorithms however still use the original Gradient Descent algorithm as their base. These algorithms have been derived by modifying certain aspects of the Gradient Descent algorithm, like the updating formula or the learning rate.

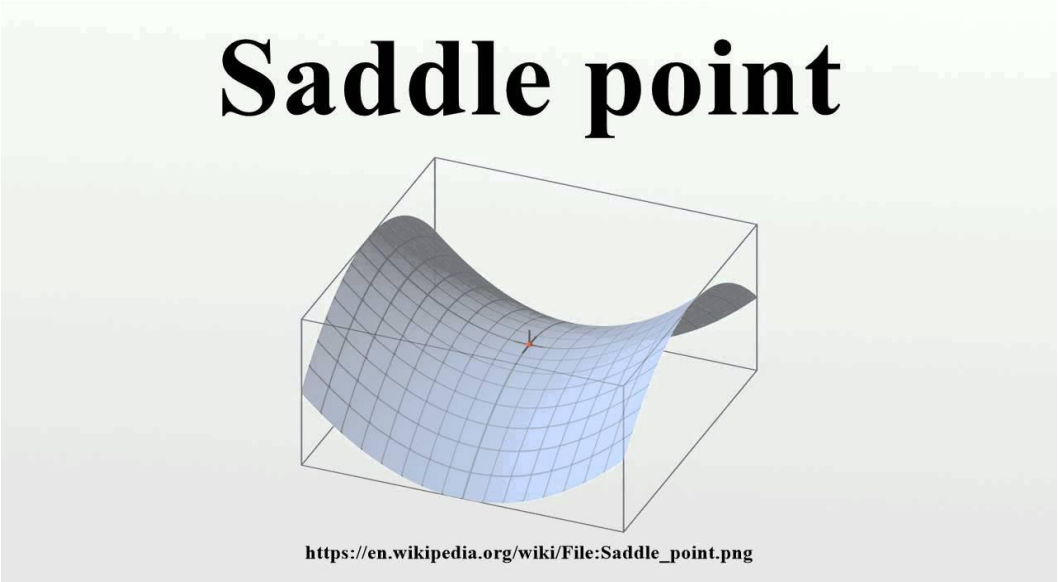

Figure (5) A surface with a saddle point.

3. Application of gradient descent

The following is an example minimizing the function *J* using standard Gradient Descent (sgd).

$$J(\theta) = (2\theta_1^2 - 5\theta_1 + 4 + \theta_3^2\theta_4 - \frac{2\theta_3 + 3\theta_4}{\theta_2} + 6\theta_2)/(\theta_3 * \theta_4)$$

From the value of *J(θ)* it can be observed that Gradient Descent works well for this equation and the value of *J(θ)* continuously keeps on decreasing with every step. The code shown in Appendix A-1 iterates the standard gradient descent algorithm.

4. Momentum

Gradient Descent algorithm encounters difficulties when the surface of the function to be minimized has steeper curves in one direction as compared to others. These surfaces are also called ravines. When this happens, Gradient Descent keeps on oscillating across the slopes in the ravine and makes slow progress towards the actual minima.

When momentum is applied to the algorithm, it reduces the oscillations as it dampens them. Momentum is applied by adding a fraction of the last update vector to the current update vector. This gives Gradient Descent with momentum algorithm [6].

This can be compared to a ball rolling downhill. When a ball rolls downhill, it keeps on gaining velocity as it goes down. This gain in velocity results because of the moment of the ball going downward. At any given point, the movement of the ball not only depends on the slope at that point, but also on the direction of velocity attained by the ball because of the previous points.

The momentum term increases the update for the dimensions where the gradients point in the same direction and it reduces the update for the dimensions whose gradients direction keeps on changing. This results in less oscillation and makes the algorithm converge faster.

The updating formula used for momentum with Gradient Descent is,

$v_t = \gamma * v_{t-1} + lr * \nabla_\theta J(\theta)$

$\theta \leftarrow \theta - v_{t-1.}$

Here, γ is the momentum term which decides the fraction of the previous term to be added to the next term, lr represents the learning rate, and v_t is called the momentum term at step t. The value of γ determines how much the last update will affect the current update. A high value of γ would mean that the next update would be highly dependent on the last update and a low value of γ would mean the opposite. If we put γ as zero, we get the standard Gradient

Descent algorithm, that is, the past update will no longer affect the current update. Usually a value of 0.9 is used for γ.

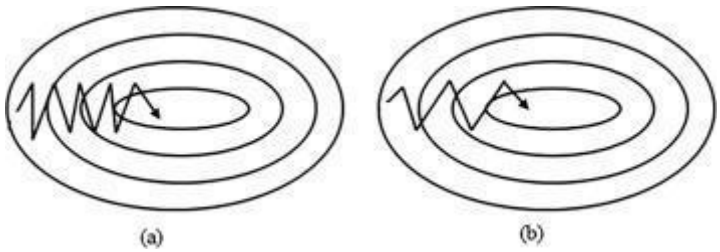

Figure (6) Figure comparing Gradient Descent without(a) and with(b) momentum. Source - (https://ars.els-cdn.com/content/image/1-s2.0-S1319157818300636-gr2.jpg)

Figure (6) shows the effect of adding momentum to the Gradient Descent algorithm. Part a. is Gradient Descent without momentum. Part b. is Gradient Descent with momentum. We can see that the second figure converges faster as it reaches the minima point with fewer updates. Consider the function,

$$J(\theta) = (2 * {\theta_1}^2 - 5 * \theta_1 + 4 + {\theta_3}^2 * \theta_4 - \frac{\theta_3 * 2 + \theta_4 * 3}{\theta_2} + \theta_2 * 6)/(\theta_3 * \theta_4)$$

From the value of *J(θ)* it can be observed that the momentum works better than standard Gradient Descent and reaches a more minimal point in the same number of iterations. See Appendix A-2.

5. Nesterov accelerated gradient

Gradient Descent with momentum helped in solving certain problems. However, one problem remaining is that if momentum is given to the point by considering the past update, it may happen that after reaching the minima, the point would continue to move upward because of the momentum due to its past update. This problem is solved by using Nesterov accelerated Gradient Descent algorithm [1], as it also computes an approximate future value of the next position of the parameters.

Comparing this Nesterov accelerated gradient with momentum gradient descent, it can be seen that momentum Gradient Descent calculates the current gradient value and then moves the point in the direction of the accumulated gradient formed by combining this past and present gradients. Whereas, Nesterov momentum first takes a move in the direction of the accumulated gradients from the past and then calculates a rough approximate value of the future gradient. It then uses this future approximate value to correct its path. This prevents the algorithm to move too fast in a particular direction. This proves helpful in case the direction of the gradient along the direction with maximum momentum changes.

As an analogy, if a ball keeps on falling down the hill, when it reaches the minimum point, it doesn’t stop there, but instead continues to move upwards because of the past accumulated gradients. This happens because the momentum achieved by the ball has no sense of future position of the ball, but just the past. In the Nesterov accelerated Gradient Descent algorithm, this problem is solved by finding an approximate value for the future position. So, the ball rolling downhill actually has an idea that the direction of gradient is going to change, so it corrects the direction of its movement. This solves the problem faced by momentum Gradient Descent and results in more sensible updates in the direction of the movement of the point.

Modifying the update methods used for momentum, the updating formula used for Nesterov accelerated Gradient Descent are,

$v_t = \gamma * v_{t-1} + lr * \nabla_\theta J(\theta - \gamma * v_{t-1})$

$\theta \leftarrow \theta - v_t$

The notations used in the above equation are the same as that used for momentum algorithm. As can be seen when the term $\theta - \gamma * v_{t-1}$ is computed, which gives a rough approximation of the updated position of the parameters. This is called an approximate value, as it does not contain the gradient term. With the exception of the gradient term, this formula is similar to the momentum Gradient Descent updating formula. The value of gradients is then found at this point, which is approximated future value and multiply it by learning rate, then add it to the term which represents past update, the same as in momentum with gradient descent. This final value calculated is then used for updating the parameters.

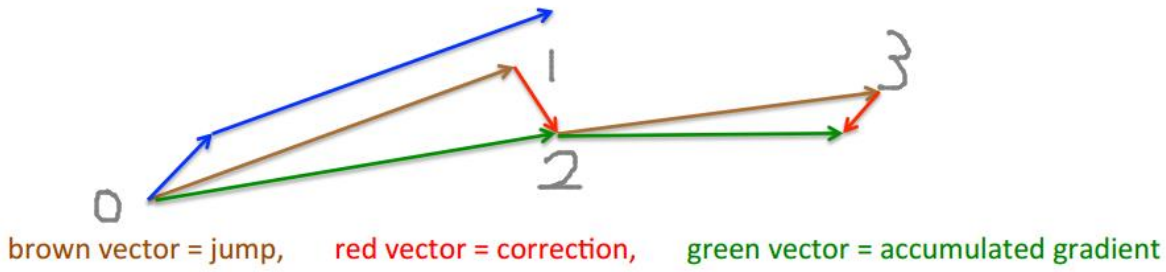

Figure (7) Comparing Momentum and Nesterov accelerated gradient. Source - (https://i.stack.imgur.com/wBIpz.png)

In Figure (7) the combined blue vector shows the update using normal momentum algorithm. The first small blue vector from point 'O' represents the value of movement from the current gradient. The next blue vector from the tip of current blue vector, represents the movement because of past updates. The combined blue vectors give the direction of the movement. The brown vector shows the update because of the accumulated movements from the past

updates. The red small vector gives the correction vector, which is the approximate value of the future. This is then added to the previous brown vector to correct its direction. The green vector then gives the direction of the final update. Two updates are illustrated in the above image for Nesterov momentum.

Consider the function,

$$J(\theta) = (2 * {\theta_1}^2 - 5 * \theta_1 + 4 + {\theta_3}^2 * \theta_4 - \frac{\theta_3 * 2 + \theta_4 * 3}{\theta_2} + \theta_2 * 6)/(\theta_3 * \theta_4)$$

From this output it can be observed that Nesterov worked better than the Momentum algorithm, reaching a minimum value of 4.9902086

From the value of *J(θ)* it can be observed that Nesterov worked better than the Momentum algorithm, reaching a minimum value of 4.9902086. See Appendix A-3.

6. Adagrad

The next update in the Gradient Descent algorithm is Adagrad [2]. This variant of Gradient Descent uses different learning rates for different components of the variable. The previous versions performed the update for every direction at the same time, as the learning rate was the same for all of them. But as Adagrad uses different learning rate for each component, we perform individual updates for all the directions.

One benefit of using Adagrad is that it does not require the learning rate *lr* to be changed for every time step *t* when an update is made as the learning rate is modified for every component at every time step.

The update formula used in Adagrad is,

$g_{t,i} = \nabla_{\theta t} J(\theta_{t,i})$

Here, $g_{t,i}$ is the gradient of the function with respect to parameter θ_i at time step *t*. This gradient is then used to update the value of θ_i to get the next updated value which is θ_{i+1} using the given formula,

$\theta_{t+1,i} = \theta_{t,i} - lr * g_{t,i}$

The value of *lr* in the above formula is updated for every value of θ_i at every time step *t*, so that the learning rate is different for each θ_i. The modified learning rate for every parameter is computed by,

$\theta_{t+1,i} = \theta_{t,i} - lr /((G_{t,ii} +\varepsilon)^{(1/2)}) * g_{t,i}$

Here, the extra term $(G_{t,ii} + \varepsilon)^{\frac{1}{2}}$ takes care of changing the learning rate.

In the above mentioned formula, $G_{t,ii}$ is the sum of squares of gradients of the function to be minimized with respect to θ, for all time step up to *t*. It also may happen that this term may become zero, hence, to avoid division by zero, we add the term ε.

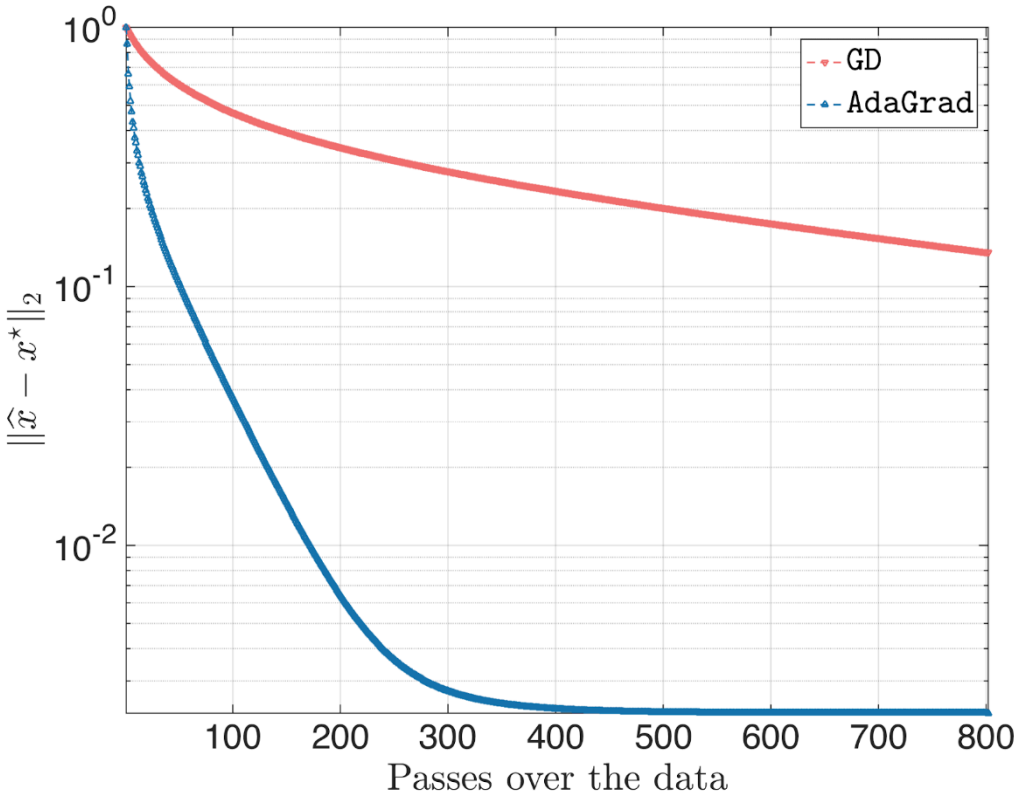

Figure (8) Graph comparing Gradient Descent and Adagrad for minimizing the error in a neural network where $\hat{x}$ is the predicted value and x^* is the true value which the neural network is trying to learn. Source - (https://akyrillidis.github.io/notes/AdaGrad/GDvsAdaGrad2.png)

Figure (8) shows a comparison between Gradient Descent and Adagrad for minimizing a function. As we can see, the Adagrad algorithm performs a lot better than the standard Gradient Descent algorithm and reaches an error value which is much smaller than the gradient descent.

One major drawback with this algorithm lies in its adaptive learning rate. The learning rate is divided by the square root of the term $G_{t,ii}$ which keeps on accumulating, that is, with every

update or iteration the gradient keeps on adding, which makes this value very large. Dividing *lr* by this very large value decreases the value of learning rate to an infinitesimally small number. Eventually when the number of updates is too large, this value becomes so small that the algorithm is not able to update the parameters further as the update becomes almost negligible.

Consider the function,

$$J(\theta) = (2 * {\theta_1}^2 - 5 * \theta_1 + 4 + {\theta_3}^2 * \theta_4 - \frac{\theta_3 * 2 + \theta_4 * 3}{\theta_2} + \theta_2 * 6)/(\theta_3 * \theta_4)$$

From the value of *J(θ)* it can be observed that Adagrad did not provide any exceptional result while minimizing the function. See Appendix A-4.

7. RMSprop

RMSprop has been developed with the goal overcoming the problem of diminishing updates in Adagrad [4]. In the presence of saddle points, the RMSprop algorithm goes straight down the slope even if the gradients are very small. The scaling of learning rate in RMSprop helps it to move through saddle points faster than any other algorithm seen above.

The update formula used in RMSprop are,

$E[g^2]_t = 0.9 * E[g^2]_{t-1} + 0.1 * g_t^2$

$\theta_{t+1} = \theta_t - lr * p/((E[g^2]_t + \varepsilon) \text{^}(1/2))* g_t$

Here, the term $E[g_2]_t$ accumulates the sum of squares of gradients up to time t and *lr* is the learning rate used for updating the parameters.

Consider the function,

$$J(\theta) = (2 * {\theta_1}^2 - 5 * \theta_1 + 4 + {\theta_3}^2 * \theta_4 - \frac{\theta_3 * 2 + \theta_4 * 3}{\theta_2} + \theta_2 * 6)/(\theta_3 * \theta_4)$$

From the value of *J(θ)* it can be observed that RMSprop actually attains the lowest minimum value of all the algorithms discussed until now. See Appendix A-5.

RMSprop suffers from the same problem as the standard gradient descent. When the slope becomes steeper in one dimension as compared to other, the algorithm keeps on oscillating in that direction because of larger updates in that dimension and smaller in others. The momentum algorithm solved this problem with gradient descent. This problem in RMSprop is solved by combining it with the RMSprop algorithm, which gives the Adam algorithm. Adam uses momentum to solve this drawback of RMSprop by combining both of them in its updation rule.

8. Adam

Adam stands for Adaptive Moment Estimation. This method also computes and adapts learning rate for each parameter separately [7]. Adam can be considered as a union of Adadelta and Momentum. Like the Adadelta algorithm, Adam also uses the exponentially decaying average of sum of squares of past gradients and similar to momentum algorithm, it also uses exponentially decaying average of gradients from the past iterations.

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$$
$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_2$$

In the above formulas, m_t computes the exponentially decaying sum of gradients from the past, and v_t computes the sum of squares of gradients from past with an exponential decay. In the initial iteration, m_t and v_t are initialized with zero, they become biased towards zero in the initial iterations, especially when the decay rates β_1 and β_2 are too small, where β_1 and β_2 are manually chosen constants. To overcome this problem, instead of using m_t and v_t for updating the value of parameters, we slightly modify both of these vectors by,

$$\hat{m}_t = \frac{m_t}{(1 - \beta_1^t)}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The values of $\hat{m}_t$ and $\hat{v}_t$ are then used to update the value of the parameters associated with the function by using,

$$\theta_{t+1} = \theta_t - \frac{lr}{(\hat{v}_t + \varepsilon)^{\frac{1}{2}}} * (\hat{m}_t)$$

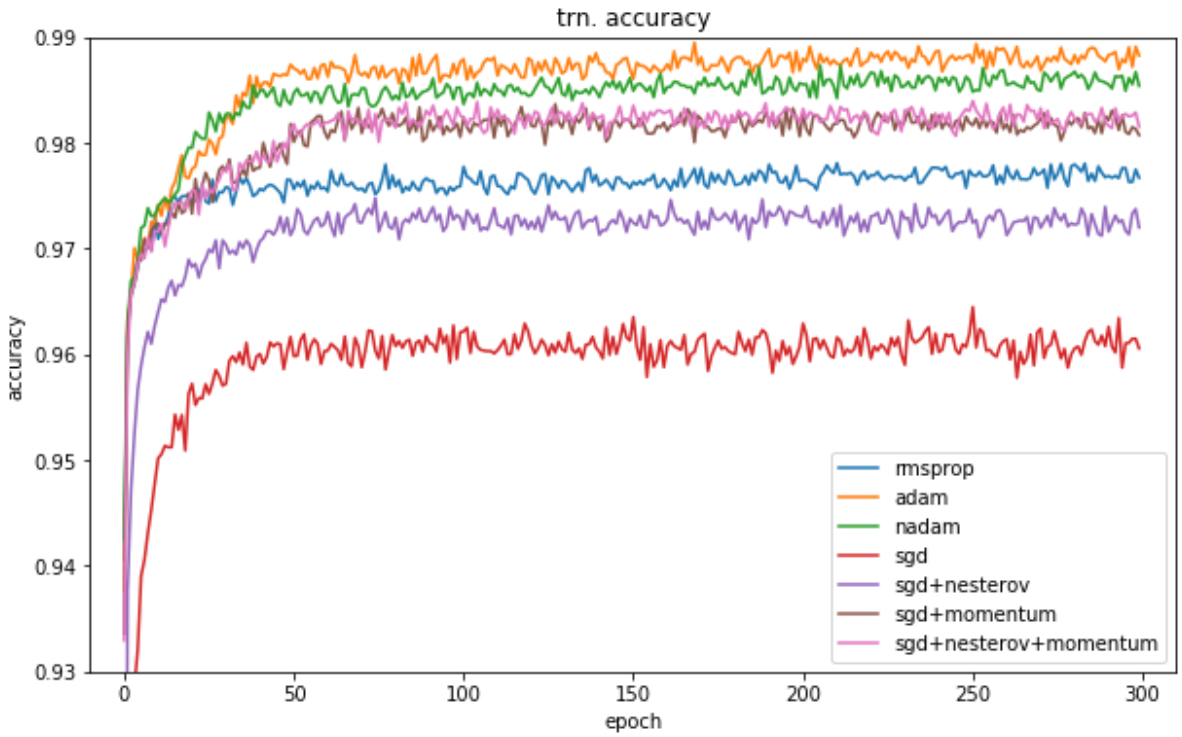

Figure (9) Graph comparing accuracy of various algorithms for optimizing a neural network.

Source- (https://shaoanlu.files.wordpress.com/2017/05/trn_acc.png?w=788)

Figure (9) shows the graph of training accuracy of a neural network with the number of epochs which is nothing but number of iterations for minimizing the error in a neural network, by using various optimization algorithms for a machine learning problem. As we can see, the Adam optimization performs the best and beats every other optimization algorithm.

Consider the function,

$$J(\theta) = (2 * {\theta_1}^2 - 5 * \theta_1 + 4 + {\theta_3}^2 * \theta_4 - \frac{\theta_3 * 2 + \theta_4 * 3}{\theta_2} + \theta_2 * 6)/(\theta_3 * \theta_4)$$

From the value of *J(θ)* it can be observed that Adam worked better than most of the algorithms discussed above and reached a minimum value of 2.8517976. See Appendix A-6.

9. AdaMax

AdaMax is a variant of the Adam optimizer [3]. AdaMax offers the advantage of being much less sensitive to the hyper-parameters, that is, the learning rate and the constants that are manually chosen while using an optimizer.

Adam updating formula scales the gradient by the l_2 norm of past gradients, as well as the current gradients by calculating,

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\|g_t\|_2$$

For large order norms, the values become numerically unstable, which is why norms of order 1 and 2 are usually used. However, it is observed that a norm of infinite order is also stable. This property is used to modify adam optimization to get the AdaMax optimizer. The updating formula for AdaMax is,

$$u_t = \beta_2 * v_{t-1} + (1 - \beta_2)\|g_t\|_\infty$$

The above can be simplified to get,

$$u_t = max\,(\beta_2 * v_{t-1}, \|g_t\|)$$

This value of u_t can then be used in the updating formula rule of adam to get the final updating formula of AdaMax,

$\theta_{t+1} = \theta_t - (lr / u_t) * m^t$

It can be seen that the value of u_t depends on the max operation, it is not biased towards zero, hence it is not necessary to update the value of u_t to avoid this formula unlike Adam.

Consider the function,

$$J(\theta) = (2 * \theta_1{}^2 - 5 * \theta_1 + 4 + \theta_3{}^2 * \theta_4 - \frac{\theta_3 * 2 + \theta_4 * 3}{\theta_2} + \theta_2 * 6)/(\theta_3 * \theta_4)$$

From the value of $J(\theta)$ it can be observed that Adamax performed exceptionally well by reaching a minimum value of -1551.8826 but then the value started to move upwards which can be due to large learning rate. See Appendix A-7.

10. Nadam

As observed in the previous sections, Adam is a combination of Adadelta and momentum algorithm. It was also observed that Nesterov accelerated momentum works better than the simple momentum algorithm. Nadam [6] combines Adam and Nesterov accelerated momentum. This is done by simple modifying the momentum term in Adam.

The momentum update rule is,

$$g_t = \nabla_{\theta t} J(\theta_t)$$
$$m_t = \gamma * m_{t-1} + lr * g_t$$
$$\theta_{t+1} = \theta_t - m_t$$

This momentum formula is then modified to get the formula for Nesterov accelerated momentum formula,

$$g_t = \nabla_{\theta t} J(\theta_t - \gamma * m_{t-1})$$
$$m_t = \gamma * m_{t-1} + lr * g_t$$
$$\theta_{t+1} = \theta_t - m_t$$

The writers of this paper were advised to use look ahead momentum directly, to modify the parameters instead of using the momentum step twice. This results in the modified formula for Nesterov accelerated momentum,

$$g_t = \nabla_{\theta t} J(\theta_t)$$
$$m_t = \gamma * m_{t-1} + lr * g_t$$
$$\theta_{t+1} = \theta_t - (\gamma * m_t + lr * g_t)$$

Instead of using the momentum from the past term to update the parameters, the update from the current term was used.

If the Adam updation rule is expanded, the result is,

$$\theta_{t+1} = \theta_t - lr\left((v^t + \varepsilon)^{1/2}\right) * \left(\beta_1 \hat{m}_{t-1} + (1 - \beta_1) * \frac{g_t}{1 - \beta_1}\right)$$

Nesterov accelerated momentum can be added to this formula by replacing the past momentum term $\hat{m}_{t-1}$ with the current momentum term of $\hat{m}_t$ to get the Nadam updation rule,

$$\theta_{t+1} = \theta_t - lr\left((v^t + \varepsilon)^{1/2}\right) * \left(\beta_1 \hat{m}_t + (1 - \beta_1) * \frac{g_t}{1 - \beta_1}\right)$$

Consider the function,

$$J(\theta) = (2 * {\theta_1}^2 - 5 * \theta_1 + 4 + {\theta_3}^2 * \theta_4 - \frac{\theta_3 * 2 + \theta_4 * 3}{\theta_2} + \theta_2 * 6)/(\theta_3 * \theta_4)$$

From the value of *J(θ)* it can be observed that Nadam reached a minimum value close to 1.53 but then again, it started to move upwards like Adamax and ended up with a value of 2.88 by the end of 10 iterations. See Appendix A-8.

Conclusion

In this paper, the Gradient Descent algorithm, which is an optimization algorithm for finding the minimum value of a function was analyzed. The drawbacks of this method and how they are overcome were observed. Problems still remaining with the Gradient Descent algorithm and various other modified algorithms for optimization which are based on Gradient Descent and their problems and strengths were observed.

Appendix A

1. The code shown below iterates the standard gradient descent algorithm

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 - (theta3*2
+theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.train.GradientDescentOptimizer(0.1).minimize(f_x)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(10):
        print(sess.run([theta1,theta2,theta3,theta4,loss]))
        sess.run(opt)
```

If this function is minimized using Gradient Descent algorithm, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$].

Output:

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.922222, 1.9583334, 3.0062964, 15.024148, 6.0384607]
[9.845421, 1.9155577, 3.009418, 15.047722, 5.955703]
[9.769498, 1.8715187, 3.0096257, 15.07076, 5.8734064]
[9.6943655, 1.8260425, 3.0071359, 15.093296, 5.7910576]
[9.619946, 1.7789302, 3.0021262, 15.11536, 5.7081537]
[9.546166, 1.7299496, 2.9947407, 15.136979, 5.624166]
[9.472961, 1.6788257, 2.9850924, 15.158175, 5.5385194]
[9.4002695, 1.6252266, 2.9732645, 15.17897, 5.4505467]
[9.328033, 1.568744, 2.9593098, 15.199381, 5.3594427]
```

2. The code shown below iterates the gradient descent algorithm with momentum.

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 - (theta3*2
+theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.train.MomentumOptimizer(0.1, momentum = 0.3).minimize(f_x)
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   for i in range(10):
      print(sess.run([theta1,theta2,theta3,theta4,loss]))
      sess.run(opt)
```

If this function is minimized using the momentum algorithm, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$],

Output:

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.922222, 1.9583334, 3.0062964, 15.024148, 6.0384607]
[9.822087, 1.9030577, 3.0113068, 15.054966, 5.930835]
[9.7164135, 1.8420639, 3.0120802, 15.087081, 5.816583]
[9.6101885, 1.7773033, 3.0078099, 15.118888, 5.6998477]
[9.504784, 1.7089179, 2.9985049, 15.149948, 5.580802]
[9.400476, 1.6364149, 2.984409, 15.180176, 5.4583063]
[9.297221, 1.5589616, 2.9657722, 15.209587, 5.33055]
[9.194885, 1.4753927, 2.9427607, 15.238225, 5.194999]
[9.093315, 1.3840795, 2.915415, 15.26613, 5.0479383]
```

3. The code shown below iterates the Nesterov accelerated gradient descent algorithm

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 - (theta3*2
+theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.train.MomentumOptimizer(0.1, 0.3,use_nesterov =
True).minimize(f_x)
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   for i in range(10):
      print(sess.run([theta1,theta2,theta3,theta4,loss]))
      sess.run(opt)
```

If this function is minimized using Nesterov accelerated gradient, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$],

Output:

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.898889, 1.9458333, 3.0081851, 15.031393, 6.013454]
[9.792426, 1.8860236, 3.0115807, 15.063991, 5.898785]
[9.685525, 1.8225658, 3.0096164, 15.096223, 5.7823315]
[9.579504, 1.7556536, 3.0024717, 15.127678, 5.664274]
[9.474617, 1.6848506, 2.9904754, 15.158282, 5.5435586]
[9.37081, 1.609394, 2.9739182, 15.188058, 5.4185534]
[9.267948, 1.5282156, 2.9529932, 15.217049, 5.287042]
[9.16588, 1.4398322, 2.9277678, 15.245298, 5.14588]
[9.06445, 1.3421049, 2.8981497, 15.272844, 4.9902086]
```

4. The code shown below iterates the Adagrad variant of the gradient descent algorithm

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 - (theta3*2
+theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.train.AdagradOptimizer(0.1).minimize(f_x)
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   for i in range(10):
      print(sess.run([theta1,theta2,theta3,theta4,loss]))
      sess.run(opt)
```

If this function is minimized using Adagrad, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$],

Output:

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.907364, 1.9203434, 3.0195274, 15.060691, 6.0018377]
[9.840183, 1.8561982, 3.0228057, 15.111122, 5.910739]
[9.784887, 1.8004539, 3.016301, 15.155236, 5.833094]
[9.736764, 1.7500898, 3.0034652, 15.194971, 5.763172]
[9.693542, 1.7034986, 2.9865, 15.231447, 5.6982193]
[9.653928, 1.659709, 2.9668767, 15.265378, 5.6366906]
[9.617109, 1.618082, 2.9455829, 15.297251, 5.5776215]
[9.582537, 1.5781717, 2.9232743, 15.327414, 5.520354]
[9.549818, 1.5396513, 2.90038, 15.356125, 5.4644027]
```

5. The code shown below iterates the RMSprop variation of the gradient descent algorithm.

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 - (theta3*2
+theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.train.RMSPropOptimizer(0.8).minimize(f_x)
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   for i in range(10):
      print(sess.run([theta1,theta2,theta3,theta4,loss]))
      sess.run(opt)
```

If this function is minimized using RMSprop, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$],

Output:

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.36511, 1.6519765, 3.0530837, 15.202978, 5.4486365]
[8.779703, 1.186784, 2.8852398, 15.376843, 4.635596]
[8.196805, 0.35474062, 2.5559142, 15.535678, 1.3905575]
[7.582406, -2.0904992, 1.4112455, 15.67072, 5.587976]
[6.604772, -2.2830892, 2.8558655, 15.878063, 4.3530188]
[6.1856503, -2.3753843, 2.4671373, 15.948829, 3.930082]
[5.7242384, -2.4813306, 2.115982, 16.016907, 3.505566]
[5.2225194, -2.6035602, 1.8013016, 16.078365, 3.0693698]
[4.685563, -2.7453246, 1.5117476, 16.127918, 2.6082675]
```

6. The code shown below iterates the Adam variation of the gradient descent algorithm.

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 - (theta3*2
+theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.train.AdamOptimizer(0.8).minimize(f_x)
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   for i in range(10):
      print(sess.run([theta1,theta2,theta3,theta4,loss]))
      sess.run(opt)
```

If this function is minimized using Adam, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$],

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.200001, 1.2000005, 3.799996, 15.799999, 5.2764587]
[8.421813, 0.41791004, 3.26558, 16.558508, 2.7432792]
[7.6467776, -0.17172569, 2.6189187, 17.284792, 11.76774]
[6.8686733, -0.7001622, 2.7730923, 18.02325, 5.673423]
[6.100538, -1.1717173, 2.9014716, 18.729305, 4.6276717]
[5.3514, -1.5871048, 2.939927, 19.388006, 4.086396]
[4.626879, -1.954987, 2.8761785, 19.992569, 3.6687298]
[3.930489, -2.2829752, 2.715311, 20.540276, 3.2696524]
[3.2646527, -2.577186, 2.4678833, 21.030418, 2.8517976]
```

7. The code shown below iterates the AdaMax variation of the gradient descent algorithm.

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 - (theta3*2
+theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.contrib.opt.AdaMaxOptimizer(0.5).minimize(f_x)
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   for i in range(10):
      print(sess.run([theta1,theta2,theta3,theta4,loss]))
      sess.run(opt)
```

If this function is minimized using Adamax, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$],

Output:

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.5, 1.5, 3.5, 15.5, 5.5337944]
[9.056901, 1.0570564, 3.2746582, 15.920429, 4.765695]
[8.634653, 0.7260843, 2.9798636, 16.29936, 3.7775757]
[8.22086, 0.47218934, 2.6771438, 16.649302, 2.3131359]
[7.8092017, 0.27379906, 2.4065537, 16.975315, -0.40879026]
[7.396519, 0.11861443, 2.1885688, 17.277468, -8.303569]
[6.9820037, 0.0010245052, 2.0271308, 17.546543, -1551.8826]
[6.566657, -0.086773515, 1.9383408, 17.467293, 22.774347]
[6.152319, -0.16032444, 1.8650105, 17.403164, 14.093249]
```

8. The code shown below iterates the Nadam variant of the gradient descent algorithm.

Python Code:

```
import tensorflow as tf
theta1 = tf.Variable(10.0, trainable=True)
theta2 = tf.Variable(2.0, trainable=True)
theta3 = tf.Variable(3.0, trainable=True)
theta4 = tf.Variable(15.0, trainable=True)
f_x =( 2 * theta1* theta1 - 5 *theta1 + 4 + (theta3*theta3) * theta4 -
(theta3*2 +theta4*3)/theta2 + theta2*6)/(theta3*theta4)
loss = f_x
opt = tf.contrib.opt.NadamOptimizer(0.5).minimize(f_x)
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for i in range(10):
    print(sess.run([theta1,theta2,theta3,theta4,loss]))
    sess.run(opt)
```

If this function is minimized using Nadam, the following output is obtained for the first 10 iterations in the format of [θ_1, θ_2, θ_3, θ_4, $J(\theta)$],

Output:

```
[10.0, 2.0, 3.0, 15.0, 6.1222224]
[9.05, 1.0500008, 3.9499953, 15.949999, 5.1524863]
[8.415877, 0.28182697, 3.2746027, 16.535938, 1.5387307]
[7.822502, -0.3482744, 2.6253784, 17.038128, 8.147704]
[7.2439523, -0.8920035, 2.6691291, 17.566727, 5.4938083]
[6.701645, -1.2484236, 2.5782728, 18.03905, 4.7349067]
[6.1808834, -1.559493, 2.4457068, 18.475872, 4.1901164]
[5.6763115, -1.8375407, 2.2745233, 18.880936, 3.7260313]
[5.1844797, -2.0891466, 2.0679297, 19.255558, 3.296759]
[4.7027717, -2.318992, 1.8304031, 19.599789, 2.882332]
```

References

[1] Botev, Aleksandar. *Nesterov's Accelerated Gradient and Momentum as Approximations to Regularised Update Descent*. arxiv.org/abs/1607.01981.

[2] Duchi, John. *Adaptive Subgradient Methods ForOnline Learning and Stochastic Optimization*. www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[3] Fan, Tommy. *Co-Attention with Answer-Pointer for SQuADReading Comprehension Task*. web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/reports/2761004.pdf.

[4] Graves, Alex. *Generating Sequences With Recurrent Neural Networks*. arxiv.org/abs/1308.0850.

[5] Jaggi, Seema. *[PDF] PROBABILITY AND SAMPLING DISTRIBUTIONS - Free Download PDF*. slidex.tips/download/probability-and-sampling-distributions.

[6] Kim, Jihyun. *An Effective Intrusion Detection Classifier Using Long Short-Term Memory with Gradient Descent Optimization*. www.researchgate.net/publication/315638592_An_Effective_Intrusion_Detection_Classifier_Using_Long_Short-Term_Memory_with_Gradient_Descent_Optimization.

[7] Kingma, Diederik P. *Adam: A Method for Stochastic Optimization*. arxiv.org/abs/1412.6980.

[8] Ruder, Sebastian. *An Overview of Gradient Descent Optimization Algorithms*. arxiv.org/abs/1609.04747.

[9] Sutskever, Ilya. *On the Importance of Initialization and Momentum in Deep Learning*. www.cs.toronto.edu/~fritz/absps/momentum.pdf.