INTRODUCTION

Machine Learning methods have made remarkable impact in real world applications. A variety of these applications, including face recognition, machine translation, self-driving cars, search engines, autonomous robots, recommendation systems, computer games, etc., heavily rely on machine learning methods. Yet, theoretically, we do not have a good understanding of these methods.

Most of the problems that we come across in the field of Machine Learning is mainly non-convex. To understand the proper prediction and to have an accurate learning curve formulation we use optimization algorithms. Among these algorithms, the most widely used algorithm is Gradient Descent. But the Gradient Descent is best applied on a convex problem, but still, it does give a base for the formulation of advanced algorithms for different non-linear problems.

This paper attempts to discuss the complexities of a non-convex problem optimization in the field of Machine Learning. As in the real world, most of the scenarios don't fit in the convex problem representation. Most of the real-world problems are non-convex problems like face recognition, recommendation systems, autonomous robots and so on. In this is paper we will first attempt to understand the non-convex problem, Gradient Descent algorithm as an optimization algorithm and work towards understanding why it is hard to solve a non- convex problem and the possible problems to this NP-hard problem.

With the understanding of the non-convex problem optimization complexities with base Gradient Descent algorithm, other advanced algorithms are explored in order to have the best fit for the model function. This approach can then be applied to few real-world problems, from my organization, to formulate the model function that best fits the solution.

While Gradient Descent is a works best with a convex style problem, it may not be an ideal approach for a non-convex problem. Below graph represents a non-convex problem type.

Basically, in a non-convex problem statement there can be various local minima and single global minima or a flat region which makes it a little difficult to predict the global minimum point in the graph.

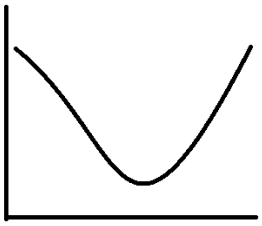
MAIN TEXT

BACKGROUND

Machine Learning (ML) is a field in AI (Artificial Intelligence) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine Learning methods have made huge impact in real world applications. A variety of these applications including face recognition, machine translation, self-driving cars, search engines, autonomous robots, recommendation systems, computer games, etc., heavily rely on machine learning methods. Yet, theoretically, we do not have a good understanding of these methods.

Machine learning focuses learning based on the experience gained doing several tasks. Hence, little related as a human's nature to do and learn. It is the scientific study of algorithms systems use to perform a specific task say T and gets the experiences E it gains over doing the tasks again and again. The process of learning begins with observations or instruction which essentially looks for patterns in data in order to make better decisions in the future based on the performance done till now.

A typical Convex problem in Machine Learning follows the below shown curve.



In the above graph it will easier enough to reach the global minima if anyone starts from any point of the curve. If the problem statement follows a convex pattern, we can use the Gradient Descent algorithm to get to the minima and the learning rate.

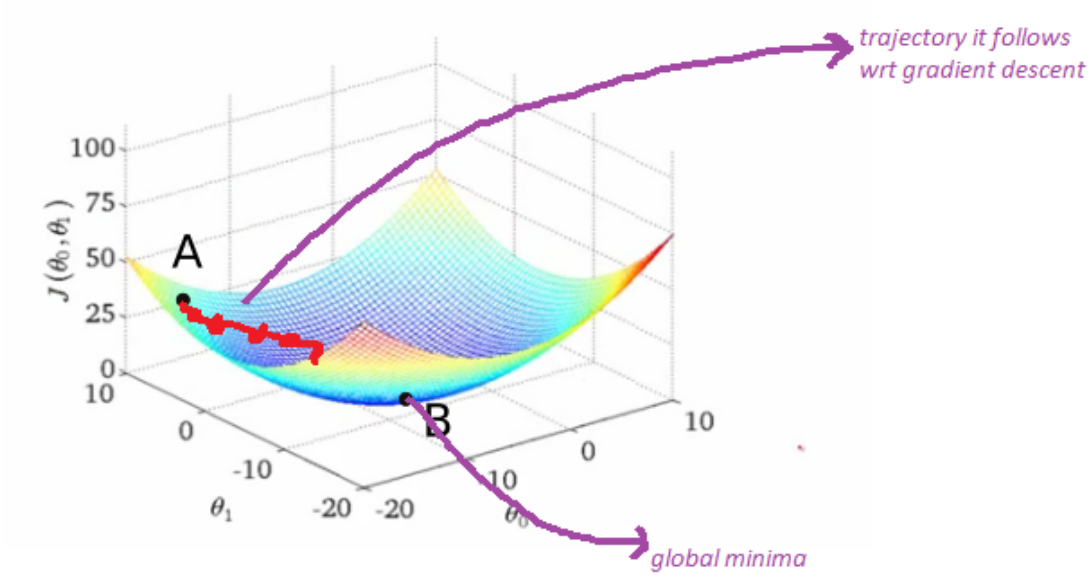
In Machine Learning, the learning algorithm can be called as the algorithm to generate the hypothesis gives a closely accurate predicted value.

COST FUNCTION AND GRADIENT DESCENT

Generally, a Cost Function is a function that measures the accuracy of the hypothesis function. It is a function that measures the performance of a Machine Learning model for given data. And Gradient descent has a direct relation with the Cost Function. When the gradient descent is at the global minima, this is the point where the cost or the squared error of the function is also the least. So, in order to have a hypothesis whose predicted values gives the least squared error with respect to the actual result we should aim at getting the global minima through the gradient descent optimization algorithm.

Now here as we saw the gradient descent is all about getting to the global minima, because that will be the point where in the squared error between the predicted and the actual value will be minimum (formulated as the Cost function). So, the goal is to find the value of weight for which the loss is minimum. Such a point is called a minima for the cost function.

With help of the below figure, let's say we initialized our weights at the point 'starting point, A'. So, now we need to figure out of all the direction from the 'starting point' that we can go in which one would be the best possible case so as it brings about the steepest decline in the value of the cost function. This is the direction we must move in. This direction is actually the direction opposite to that of the gradient. The gradient which is the higher dimensional derivative gives us the direction with the ascent.



Gradient Descent

Once we have the direction we need to move in, we'd need the learning rate now. Learning rate (α) is the parameter for gradient descent. For a sufficiently small α, the cost function (J(ɵ)) should decrease with every iteration, but if the learning rate is too small then the gradient descent may

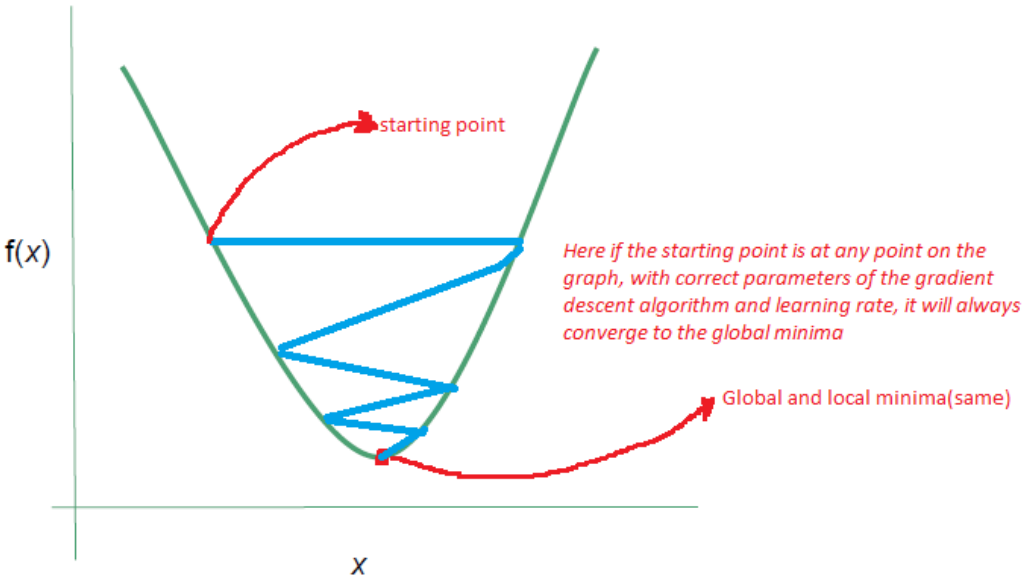
become extremely slow to converge. Go too slow, and the training might turn out to be too long to be feasible at all. Even if that's not the case, very slow learning rates make the algorithm more prone to get stuck in a minima. On the other hand, if we go too fast, we might overshoot the minima, and keep bouncing along the ridges of the "valley" without ever reaching the minima. If learning rate is too large, cost function may not decrease on every iteration thus may not converge at all.

After taking gradient and the learning rate, we take a step, and recompute the gradient to get to a new position and repeat the process till convergence.

The trajectory we take is entire confined to the x-y plane, the plane containing the weights. As depicted above, gradient descent doesn't involve moving in z direction at all. This is because only the weights are the free parameters, described by the x and y directions. The actual trajectory that we take is defined in the x-y plane as the slope of the curve (convex structured function).

While the direction of the gradient tells us, which direction has the steepest ascent, its magnitude tells us how steep the steepest ascent/descent is. So, at the minima, that shows an almost flat region, the gradient will be almost zero. In fact, it's precisely zero for the point of minima.

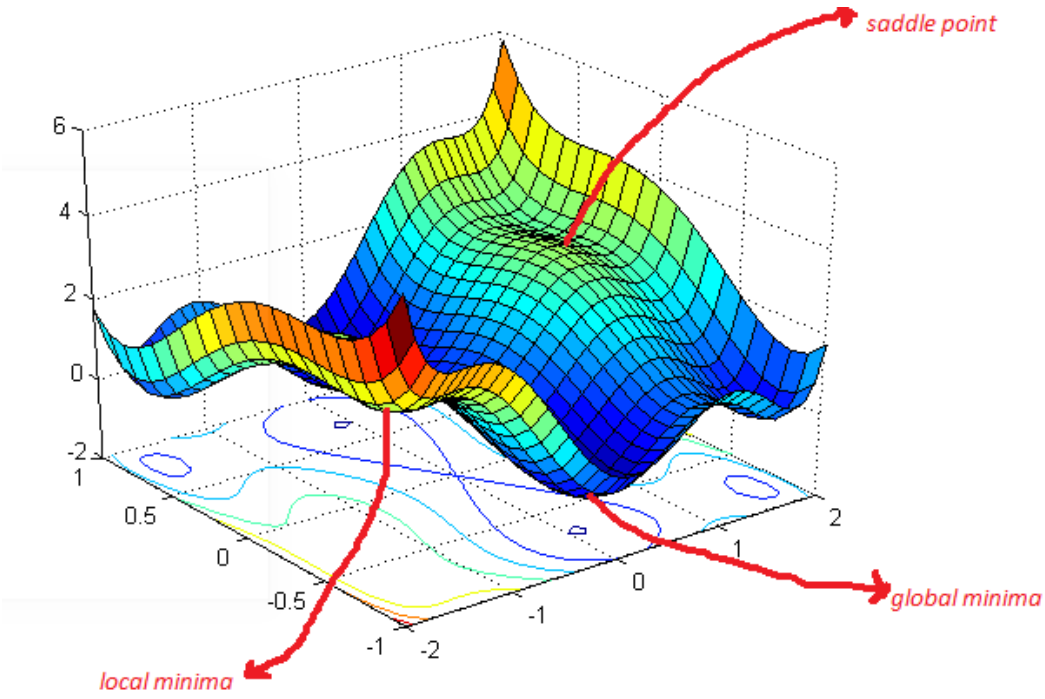
For a convex function:



NON-CONVEX PROBLEMS

In today's world the real-world applications operate in high dimensional spaces. The usage is as sparse use cases as from document classification problems where n-gram-based representations can have dimensionalities in the millions or more, recommendation systems with millions of items being recommended to millions of users, and recognition tasks such as face recognition and image processing and bio-engineering tasks such as splice and gene detection, all these represents a high dimensional data.

So, a typical non-convex problem is a complicated function, consisting of millions of parameters, that represent a mathematical solution to a problem. By training/optimizing these non-convex functions we essentially mean that we are trying to minimize a loss function, which gives us a score displaying how far from perfect is the performance on the dataset.



The problem with non-convex problem could be

- Potentially many local minima
- Saddle points
- Very flat regions

- Widely varying curvature

The above points out various issues in the potential graph that can be formed that doesn't results to a convex function.

GRADIENT DESCENT

Basic equations for the Gradient Descent are below,

Cost Function is represented as:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

The aim is to minimize the above cost function over the given parameters $\theta 0, \theta 1$.

Here, Gradient Descent is:

repeat until convergence:

{

$\theta 0 := \theta 0 - \alpha(1/m) \sum i=1 m(h\theta(xi) - yi)$

$\theta 1 :=$

$\theta 1 - \alpha(1/m) \sum i=1 m((h\theta(xi) - yi)xi)$

}

Where $\theta 0$ and $\theta 1$ are updated simultaneously.

This is performed every iteration. Here, 'θ' is the weights vector, which lies in the x-y plane. From this vector, we subtract the gradient of the loss function with respect to the weights multiplied by alpha (α), the learning rate. The gradient is a vector which gives us the direction in which cost function is almost zero. The direction of descent is the direction opposite to the gradient that was the slope, and that is why we are subtracting the gradient vector from the weights vector in order to calculate a descent. The above update, as written above, is simultaneously done for all the weights.

Before subtracting the weights, the gradient vector is multiplied by the learning rate. Even if we keep the learning rate constant, the size of step can change owing to changes in magnitude of the

gradient. As minimum is approached and the gradient approaches zero. Here, we take smaller and smaller steps towards the minima.

In theory, this is good, since we want the algorithm to take smaller steps when it approaches a minima. Having a step size too large may cause it to overshoot a minima and bounce between the ridges of the minima.

A technique used in gradient descent is to have a variable learning rate based on the progress of the steps towards the minimum. Initially, we can use a large learning rate. But later on, we want to slow down as we approach a minima, as if we don't there is a risk of overshooting the minima point. An approach that implements this strategy is called Simulated annealing where learning rate is reduced every fixed number of iterations.

ISSUES WITH NON-CONVEX PROBLEM OPTIMIZATION

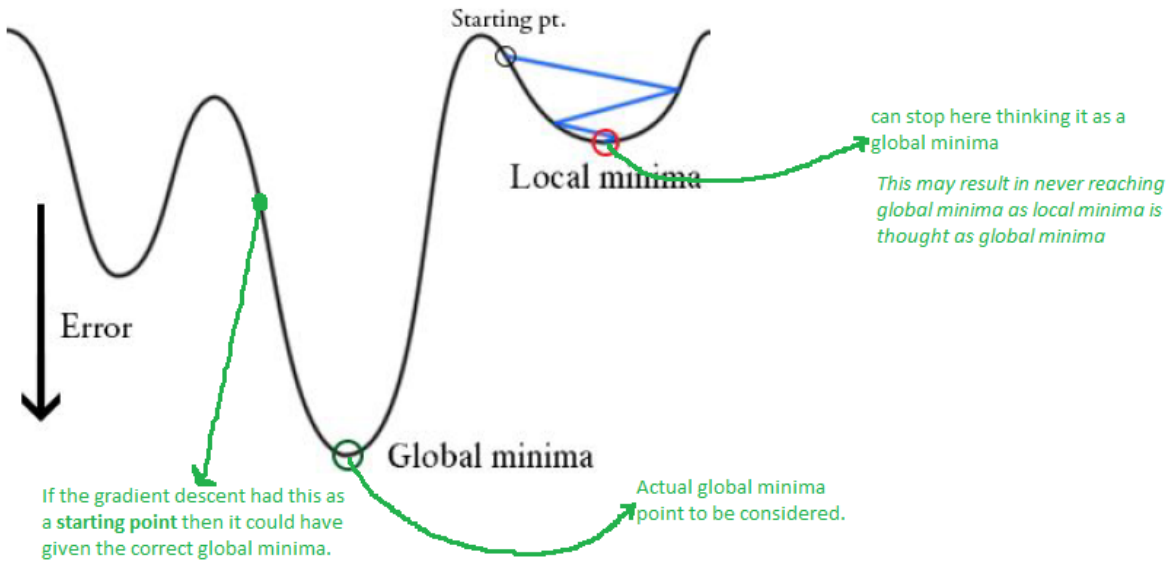
While we saw that among optimization algorithms, gradient descent (GD) is perhaps the simplest one and acts as the prototype of many advanced optimization algorithms like stochastic gradient decent (SGD) or momentum methods. These gradient-based methods provide the core optimization methodology in machine learning problems.

Local Minima

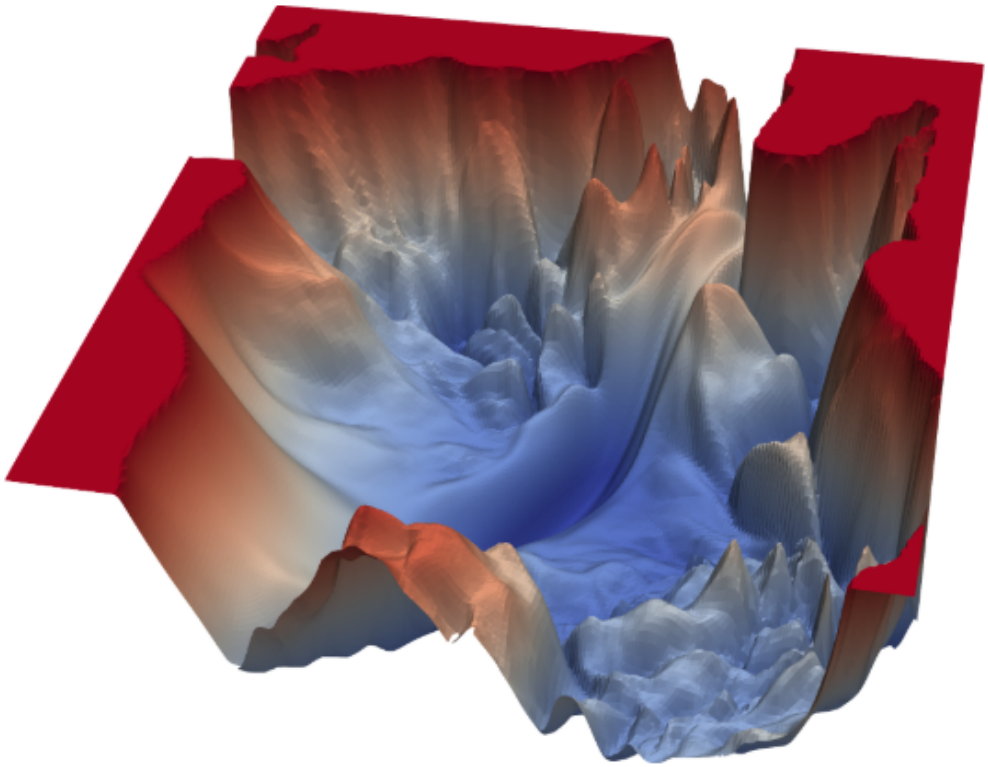
In the below figure we can see and understand the problem with local minima in a typical non-convex function. There exists a local minimum where the gradient is zero. However, we can see that they are not the lowest cost we can achieve, which is the point corresponding to the global minima. While calculating Gradient descent we expect to get to a point where the gradient will be zero, as this will be the minima. Local minimum is called so since the value of the loss function is minimum at that point in a local region. Whereas, a global minimum is called so since the value of the loss function is minimum there, globally across the entire domain the cost function.

Here, if the gradient descent starts at the 'starting pt' then it surely loses the global minimum and gets stuck at the local minima. These scenarios are highly likely to appear in a non -convex problem optimization.

Now if we see a non-convex function, the gradient descent



Although, the below figure is a simplified version of the non-convexity that can be seen. In real life examples may have a billon weights, even hard to visualize such high dimensional example. Escaping the local minima in those high dimensional problem statement on apply the plain Gradient Descent is highly unlikely. For example, the following figure 3-D representation for loss contour loss function on the CIFAR-10 dataset. As we can see it is ridden with the local minima.



A Complicated Loss Landscape *Image Credits: https://www.cs.umd.edu/~tomg/projects/landscapes/*

Saddle Points

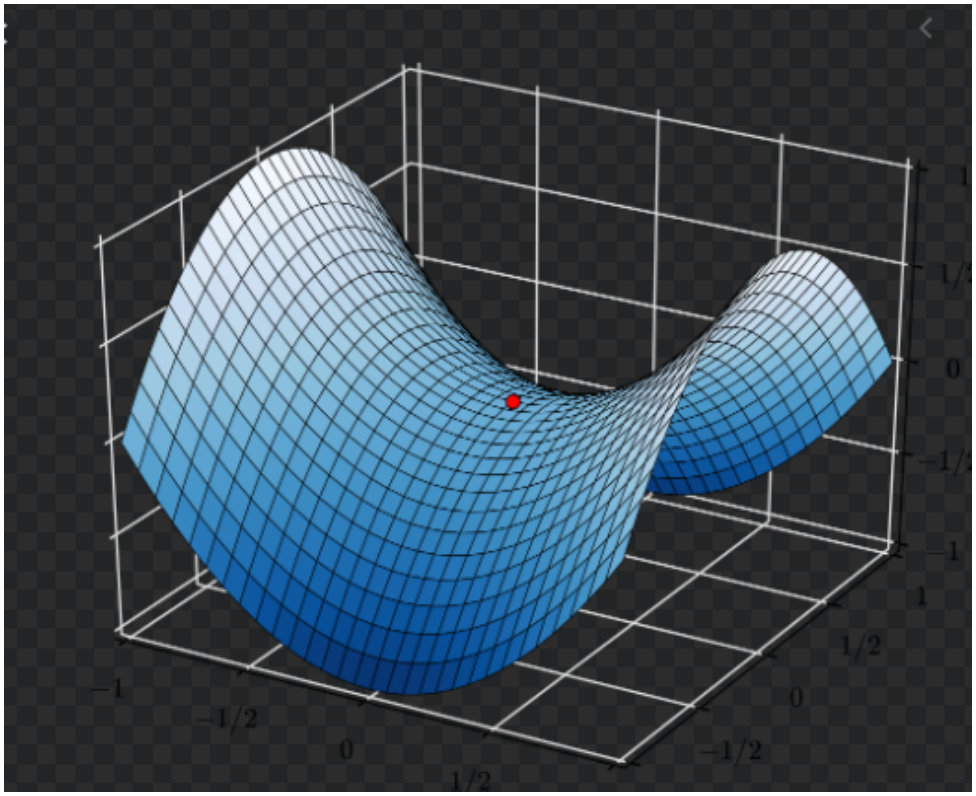
With the previous cases we worked on we know now the issue is stationary points in a non-convex problem. These points are essentially the point where the region gives a gradient of zero, which can't be escaped. Another such case is saddle points. The below figure explains the saddle point, the meeting point of two humps.

Essentially the issue with non-convex problem is it converges to stationary points (local minima, local maxima, saddle points)

$$\nabla f(x) = 0.$$

While it's a minima in one direction, it's a local maxima in another direction, and if the contour is flatter towards the x direction, GD would keep oscillating back and forth in the y - direction, and

give us the illusion that we have converged to a minimum. In non-convex scenario, Gradient Descent can take exponential time to escape saddle point.



So, with this it is established that traditional Gradient Descent can-not be applied to the non-convex problems; but it can surely serve as the base for the complex algorithm that can attempt to solve the non-convex problem optimization efficiently.

SOLVING USING STOCHASTIC GRADIENT DESCENT

Gradient Descent is the most popular and perhaps the simplest optimization technique for the machine learning problems. These gradient-based methods provide the core optimization methodology in machine learning problems.

For convergence of SGD to a stationary point in non-convex problems, we undertake few assumptions

• Second-differentiable objective

• Lipschitz-continuous gradients

• Noise has bounded variance

$$-LI \preceq \nabla^2 f(x) \preceq LI$$

$$\mathbf{E}\left[\left\|\nabla \tilde{f}_t(x) - f(x)\right\|^2\right] \leq \sigma^2$$

So, Stochastic Gradient Descent for non-convex problem converges in the sense to getting to points where the gradient is arbitrarily small.

But this doesn't mean that its stochasticity ensures it finds global minimum all the time. Also, it doesn't ensure it doesn't go to the saddle point or a local maxima or it goes to a region of very flat but nonzero gradients.

In this approach we consider the slope of a function, the degree of change of a parameter with the amount of change in another parameter.

$repeat\ until\ convergence$: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right).x_j^{(i)} \qquad \text{for j= 0..n}$$

}

Where m is the number of training examples.

And, α is the learning rate.

This helps to Gradient Descent to converge properly. The learning rate is generally set for getting the convergence properly in Gradient Descent. If learning rate is too small, then Gradient Descent

may be too slow to converge; and if the learning rate is too high then Gradient Descent may not decrease on every iteration and hence may not converge at all. It can be described as the partial derivatives of a set of parameters with respect to its inputs. The more the gradient, the steeper the slope. Gradient Descent can be described as an iterative method which is used to find the values of the parameters of a function that minimizes the cost function as much as possible. The parameters are initially defined a value and from that, Gradient Descent is run in an iterative fashion to find the optimal values of the parameters, using calculus, to find the minimum possible value of the given cost function.

STOCHASTIC GRADIENT DESCENT

Stochastic Gradient Descent is a form of advanced Gradient Descent algorithm that can help us in non-convex problem optimization. Till now working with gradient descent with the cost function and this cost function is essentially created by summing loss over all possible examples of the training set. So in a non-convex scenario if we get into a local minima or saddle point, we are stuck. A way to help gradient descent escape these is to use what is called Stochastic Gradient Descent. In stochastic gradient descent, instead of taking a step by computing the gradient of the loss function creating by summing all the cost functions, we take a step by computing the gradient of the loss of only one randomly sampled example. In contrast to Stochastic Gradient Descent, where each example is stochastically chosen, Gradient Descent processed all examples in one single batch, and therefore, is known as Batch Gradient Descent.

Stochastic Gradient Descent would randomly pick one of the samples for each step and then calculate the derivative. Hence, it'll reduce the number of terms for calculation by greater factor. Although, using the whole dataset is useful for getting to the minima in a less noisy or less random manner, but the problem arises when our datasets get huge. In SGD, we find out the gradient of the cost function of just one example at each iteration instead of summing up the gradient of the cost function of all the examples. Since only one sample from the dataset is chosen at random for each iteration, SGD is usually noisier than the typical Gradient Descent algorithm. SGD can escape saddle points in polynomial time (Gradient Descent can take exponential time to escape saddle points)

SGD is noisier. Because of its randomness it takes higher number of iterations to reach the minima but computationally is less expensive than a regular Gradient Descent. The cycle of taking the values and adjusting them based on different parameters in order to reduce the loss function is called back-propagation.

SGD is helpful in data which have redundancies (by first marking the clusters)

In SGD we use only one sample per step. But commonly a small subset of data or mini batch for each step; this can help us to lead to more stable estimates of the parameters in fewer steps.

With the introduction of new data in SGD, we don't have to start from the scratch, the new entry can be utilized in the next step for the parameter estimates. SGD is great when we have tons of data and lots of parameters.

Stochastic Gradient Descent finds "Ɛ-approximate" local minima in O(1/Ɛ^4) rounds (back propagation

$$cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right) = \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m} cost\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$$

1. $Randomly\ shuffle\ dataset$
2. $Repeat$ {

$for\ i = 1, .., m$

{

$$\theta_j = \theta_j - \alpha\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

}

}

This means, at every step, we are taking the gradient of a cost function, which is different from our actual loss function (which is a summation of loss of every example). The gradient of this one example loss at a may point in a direction slightly different to the gradient of taking all example loss.

While the gradient of the all example loss may direct to a local minima, or gets stuck at a saddle point, the gradient taking single example might point in a different direction, and might help us escape the local minima.

One could also consider a point that is a local minima for the all example loss. Batch Gradient Descent may stuck here since the gradient will always point to the local minima. If we are using

Stochastic Gradient Descent, this point may not lie around a local minima and allow us to move away from it.

We can take a small number of data points instead of just one point at each step and that is called "mini-batch" gradient descent. Mini-batch tries to give the best of both goodness of gradient descent and speed of SGD.

Even if we get stuck in a minima for the one example loss, the loss landscape for the one example loss for the next randomly sampled data point might be different, allowing us to keep moving. When it does converge, it converges to a point that is a minima for almost all the one example losses. It's also been empirically shown the saddle points are extremely unstable, and a slight nudge may be enough to escape one.

When we perform gradient descent with a cost function that is created by adding all the individual losses, the gradient of the individual losses can be calculated in parallel, whereas in stochastic gradient descent it must calculate sequentially step by step.

So, here instead of using the entire dataset, or just a single example to construct our cost function, we can use a fixed number of examples to form what is called a mini-batch. The size of the mini-batch is chosen as to ensure we get enough stochasticity to ward off local minima, while leveraging enough computation power from parallel processing.

Batch Gradient Descent

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

$$J_{train}(\theta) = \frac{1}{2m} \sum_{i=0}^{m} (h_\theta(x^{(i)} - y^{(i)})^2$$

REPEAT {

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$$

(For every j = 0,…,n)

}

OPTIMAL LOCAL MINIMA

There has been researches that shows we can still go ahead and depend on good local minima. In the network good local minima might just perform as good as a global minimum. There still exist bad local minima which are just the resultant of few erratic training examples. Such good local minima which performs like a global minimum are also referred as optimal local minima

For example, when dealing classification problem, a nearer score of the good local minima to a global minimum can be considered too. So, if a local minimum corresponds to it producing scores between 0.7-0.8 for the correct labels, while the global minima has it producing scores between 0.95-0.98 the same case then the output class prediction is going to be same for both.

Flat minima are easy to converge to, as there is chance to overshoot the minima and be bouncing between the ridges of the minima. So, flatter minima generalize better.

CONVERGENCE IN SGD

- Start with the update rule:

$$w_{t+1} = w_t - \alpha_t \nabla \tilde{f}_t(w_t)$$

- At the next time step, by Taylor's theorem, the objective will be

$$\begin{aligned} f(w_{t+1}) &= f(w_t - \alpha_t \nabla \tilde{f}_t(w_t)) \\ &= f(w_t) - \alpha_t \nabla \tilde{f}_t(w_t)^T \nabla f(w_t) + \frac{\alpha_t^2}{2} \nabla \tilde{f}_t(w_t)^T \nabla^2 f(y_t) \nabla \tilde{f}_t(w_t) \end{aligned}$$

- So now we know how the expected value of the objective evolves.

$$\mathbf{E}\left[f(w_{t+1})|w_t\right] \leq f(w_t) - \left(\alpha_t - \frac{\alpha_t^2 L}{2}\right) \|\nabla f(w_t)\|^2 + \frac{\alpha_t^2 \sigma^2 L}{2}.$$

- If we set α small enough that 1 - αL/2 > 1/2, then

$$\mathbf{E}\left[f(w_{t+1})|w_t\right] \leq f(w_t) - \frac{\alpha_t}{2}\left\|\nabla f(w_t)\right\|^2 + \frac{\alpha_t^2 \sigma^2 L}{2}.$$

- Now taking the full expectation,

$$\mathbf{E}\left[f(w_{t+1})\right] \leq \mathbf{E}\left[f(w_t)\right] - \frac{\alpha_t}{2}\mathbf{E}\left[\left\|\nabla f(w_t)\right\|^2\right] + \frac{\alpha_t^2 \sigma^2 L}{2}.$$

- And summing up over an epoch of length **T**

$$\mathbf{E}\left[f(w_T)\right] \leq f(w_0) - \sum_{t=0}^{T-1} \frac{\alpha_t}{2}\mathbf{E}\left[\left\|\nabla f(w_t)\right\|^2\right] + \sum_{t=0}^{T-1} \frac{\alpha_t^2 \sigma^2 L}{2}.$$

$$\alpha_t = \frac{\alpha_0}{t+1}$$

- So our bound on the objective becomes

$$\mathbf{E}\left[f(w_T)\right] \leq f(w_0) - \sum_{t=0}^{T-1} \frac{\alpha_0}{2(t+1)}\mathbf{E}\left[\left\|\nabla f(w_t)\right\|^2\right] + \sum_{t=0}^{T-1} \frac{\alpha_0^2 \sigma^2 L}{2(t+1)^2}.$$

- Rearranging the terms,

$$\sum_{t=0}^{T-1} \frac{\alpha_0}{2(t+1)}\mathbf{E}\left[\left\|\nabla f(w_t)\right\|^2\right] \leq f(w_0) - \mathbf{E}\left[f(w_T)\right] + \sum_{t=0}^{T-1} \frac{\alpha_0^2 \sigma^2 L}{2(t+1)^2}$$

output some randomly chosen wi from the history

Let $z_T = w_t$ with probability $\frac{1}{H_T(t+1)}$, where $H_t = \sum_{t=0}^{T-1} \frac{1}{t+1}$

Let $z_T = w_t$ with probability $\frac{1}{H_T(t+1)}$, where $H_t = \sum_{t=0}^{T-1} \frac{1}{t+1}$

- So the expected value of the gradient at this point is

$$\mathbf{E}\left[\|\nabla f(z_T)\|^2\right] = \sum_{t=0}^{T-1} \mathbf{P}\left(z_T = w_t\right) \cdot \mathbf{E}\left[\|\nabla f(w_t)\|^2\right]$$
$$= \sum_{t=0}^{T-1} \frac{1}{H_T(t+1)} \mathbf{E}\left[\|\nabla f(w_t)\|^2\right].$$

- Now we can continue to bound this with

$$\mathbf{E}\left[\|\nabla f(z_T)\|^2\right] = \frac{2}{\alpha_0 H_T} \sum_{t=0}^{T-1} \frac{\alpha_0}{2(t+1)} \mathbf{E}\left[\|\nabla f(w_t)\|^2\right]$$

- This means that for some fixed constant **C**

$$\mathbf{E}\left[\|\nabla f(z_T)\|^2\right] \leq \frac{C}{\log T}$$

- And so in the limit

$$\lim_{T \to \infty} \mathbf{E}\left[\|\nabla f(z_T)\|^2\right] = 0.$$

CODE EXAMPLES

So, the above explains how non-convex problems the actual real-world problems and a plain GD may not help in optimizing the problem. In our organization various such non-convex scenarios can be tackled. Such as a budget estimation module in out project. Each project taken by the contractor/owner are first asked to decide the type of the project and allocate the budget adding the needed items. Estimating the budget based on the type of project can be a non-convex problem that we can handle. As the data can not be directly presented below are the code snippets using randomly generated data to give a solution on optimizing such non-convex problems.

SAMPLE CODE FOR GRADIENT DESCENT

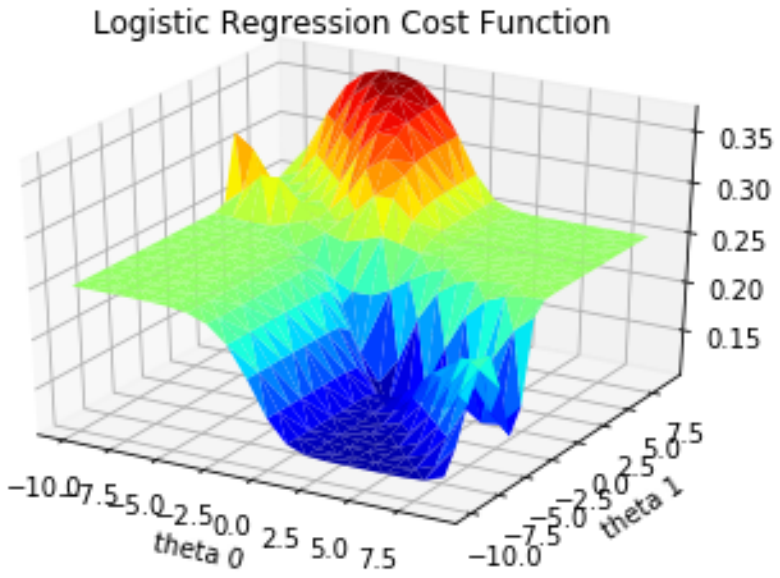
```
function [theta, Jl] = GD(X, y, theta, alpha, i)
m = length(y); % training examples
Jl = zeros(i, 1);

for iter = 1:i
    Jl(iter) = 0;

    hyp =  X*theta;
    th_zero = theta(1) - alpha*(1/m) * sum(hyp-y);
    th_one = theta(2) - alpha*(1/m) * sum((hyp-y).*X(:,2));

    theta(1)=th_zero;
    theta(2)= th_one;

    % Computing the cost function (squared errors)
    predictions = X*theta;
    sqrErrors   = (predictions - y).^2;

    Jl(iter) = 1/(2*length(y)) * sum(sqrErrors);

end
    disp(min(Jl));
end
```

SAMPLE CODE FOR STOCHASTIC GRADIENT DESCENT

```
@author: Nidhi.singh
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
import csv
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

def costFunction(h_theta_x):
    def getCost(thetas, train_set):
        result = 0
        for training_sample in train_set:
                result += pow(h_theta_x(thetas, training_sample[:-1]) - training_sample[-1],
        result *= (1/(2 * len(train_set)))
        return result
    return getCost


def getHypothesis(thetas, train_set):
    z = thetas[0] + sum(train_set * thetas[1:])
    return 1 / (1 + np.e ** (-z))

data_file =  "DataSampleCSV.csv"
data = pd.read_csv(data_file)
#sample_training_set = pd.DataFrame(data)

sample_training_set = []
with open("DataSampleCSV2.csv",encoding='utf-8-sig') as csvfile:
    reader = csv.reader(csvfile)#,quoting=csv.QUOTE_NONNUMERIC) # change contents to floats
    for row in reader: # each row is a list
        sample_training_set.append(row)
```

```
sample_training_set1 = np.array([0, 1,
                                 1, 0,
                                 2, 1,
                                 3, 0]).reshape(4, 2)
arrThetas = []

cost_lst_logistic_regression = []

start = -10
stop = 10
step = 1

for theta_0 in np.arange(start, stop, step):
    for theta_1 in np.arange(start, stop, step):
        arrThetas.append(np.array([theta_0, theta_1]))

logisticCostFunction = costFunction(getHypothesis)
for my_thetas in arrThetas:
    cost_logistic_non_convex = logisticCostFunction(my_thetas, sample_training_set1)
    cost_lst_logistic_regression.append(cost_logistic_non_convex)

theta_0_lst = [x[0] for x in arrThetas]
theta_1_lst = [x[-1] for x in arrThetas]

plt.close('all')


fig = plt.figure()
fig.patch.set_facecolor('white')
ax = fig.add_subplot(111, projection='3d')
ax.set_title('Logistic Regression Cost Function')
ax.plot_trisurf(theta_0_lst, theta_1_lst, cost_lst_logistic_regression, cmap=cm.jet, linewidt
plt.xlabel('theta 0')
plt.ylabel('theta 1')
```

OUTPUT

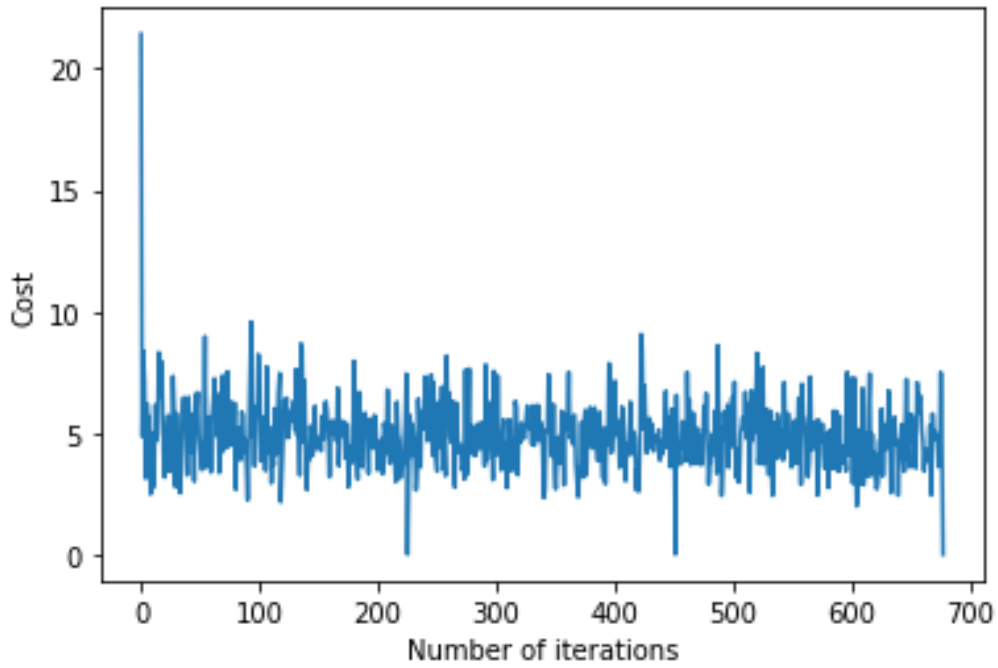


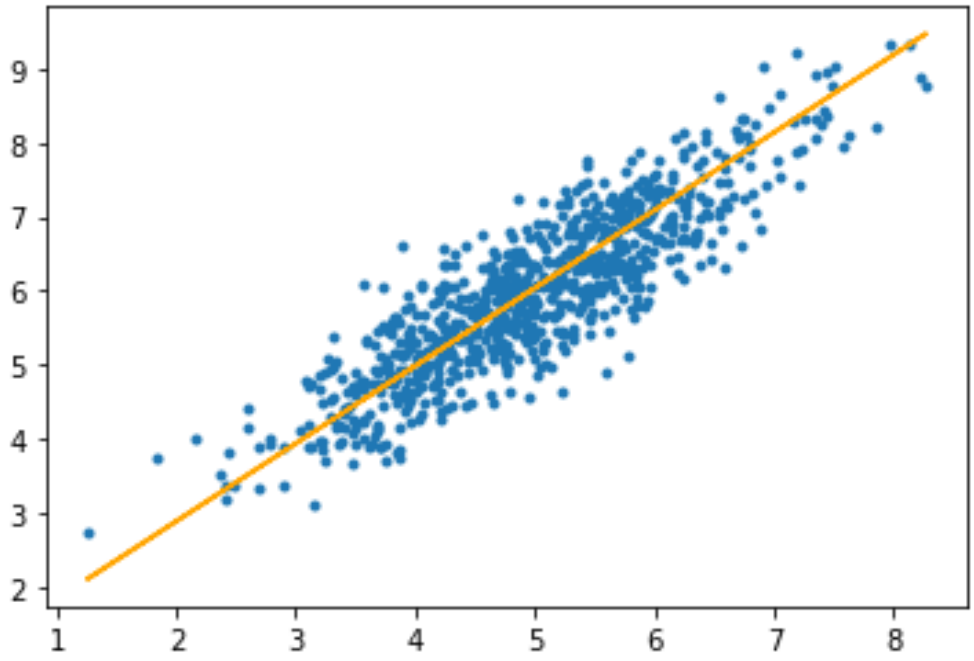

```
# -*- coding: utf-8 -*-
"""
Created on Sun Apr  5 13:43:36 2020

@author: Nidhi.singh
"""

import numpy as np
import matplotlib.pyplot as plt
mean = np.array([5.0, 6.0])
cov = np.array([[1.0, 0.95], [0.95, 1.2]])
data = np.random.multivariate_normal(mean, cov, 8000)

plt.scatter(data[:500, 0], data[:500, 1], marker = '.')
plt.show()
data = np.hstack((np.ones((data.shape[0], 1)), data))

split_factor = 0.90
split = int(split_factor * data.shape[0])

X_train = data[:split, :-1]
y_train = data[:split, -1].reshape((-1, 1))
X_test = data[split:, :-1]
y_test = data[split:, -1].reshape((-1, 1))

print("Number of examples in training set = % d"%(X_train.shape[0]))
print("Number of examples in testing set = % d"%(X_test.shape[0]))
```

```
def miniBatches(X, y, batch_size):
    mini_batches = []
    data = np.hstack((X, y))
    np.random.shuffle(data)
    n_minibatches = data.shape[0] // batch_size
    i = 0

    for i in range(n_minibatches + 1):
        mini_batch = data[i * batch_size:(i + 1)*batch_size, :]
        X_mini = mini_batch[:, :-1]
        Y_mini = mini_batch[:, -1].reshape((-1, 1))
        mini_batches.append((X_mini, Y_mini))
    if data.shape[0] % batch_size != 0:
        mini_batch = data[i * batch_size:data.shape[0]]
        X_mini = mini_batch[:, :-1]
        Y_mini = mini_batch[:, -1].reshape((-1, 1))
        mini_batches.append((X_mini, Y_mini))
    return mini_batches

# function to perform mini-batch gradient descent
def stochasticGradientDescent(X, y, learning_rate = 0.001, batch_size = 32):
    theta = np.zeros((X.shape[1], 1))
    max_iters = 3
    for itr in range(max_iters):
        mini_batches = miniBatches(X, y, batch_size)
        for mini_batch in mini_batches:
            X_mini, y_mini = mini_batch
            h = np.dot(X_mini, theta)
            grad = np.dot(X_mini.transpose(), (h - y_mini))
            theta = theta - learning_rate * grad

    return theta
```

```
theta = stochasticGradientDescent(X_train, y_train)
print("Bias = ", theta[0])
print("Coefficients = ", theta[1:])

#-------------------------------------------------------------------------------
#-------------------------------------------------------------------------------

y_pred = np.dot(X_test, theta)
plt.scatter(X_test[:, 1], y_test[:, ], marker = '.')
plt.plot(X_test[:, 1], y_pred, color = 'orange')
plt.show()

error = np.sum(np.abs(y_test - y_pred) / y_test.shape[0])
print("Mean absolute error = ", error)
```

```
Number of examples in training set =  7200
Number of examples in testing set =  800
Bias =  [0.78770556]
Coefficients =  [[1.05032598]]
```






```
Mean absolute error =  0.4415671578329669
```

STOCHASTIC WEIGHT AVERAGING

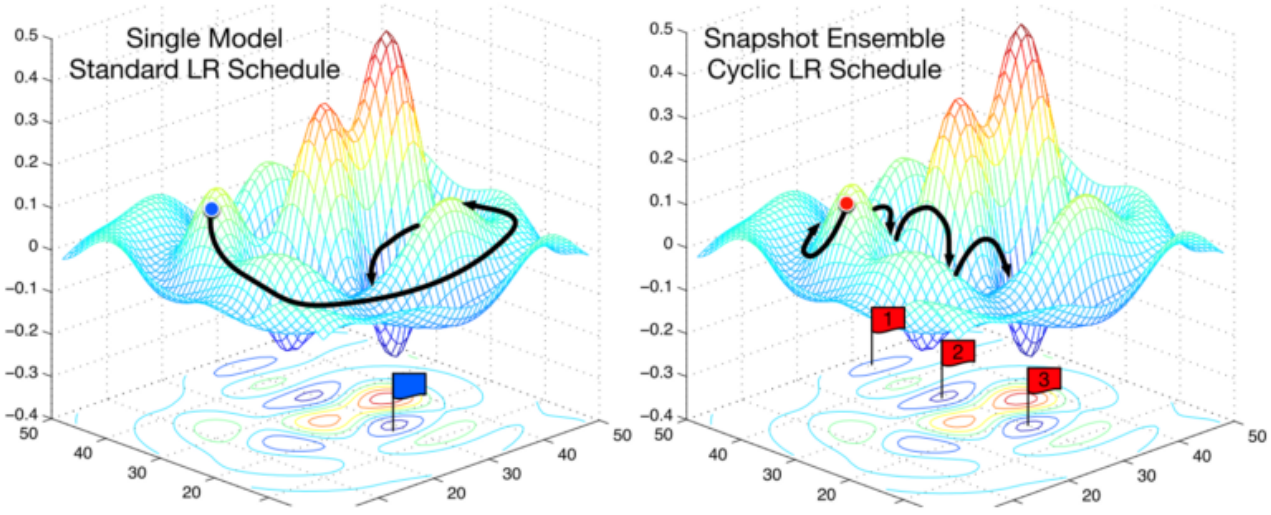
A very recent paper introduces a technique called Stochastic Weight Averaging. In which an approach where they first converge to a minima, cache the weights and then restore the learning rate to a higher value is introduced. Then the algorithm is made to converge to another minima. This is repeated for a few times. Finally we get the averaged prediction made by all the set of cached weights to as the final prediction.

Exponentially weighed averages deal with sequences of numbers. Suppose, we have some sequence S which is noisy. In this process we add some moving average that would help denoise the data. This moves the new mapped function with the momentum move closer to the actual function plot itself.

Exponentially weighed averages define a new sequence V with the following equation:

$$V_t = \beta V_{t-1} + (1-\beta)S_t$$
$$\beta \in [0,1]$$

Here beta β, is another hyperparameter which takes values from 0 to 1. Generally, a value of 0.9 is used for beta. It is used most often in SGD with momentum. With smaller numbers of beta, the new sequence turns out to be fluctuating a lot, because we're averaging over smaller number of examples and therefore are 'closer' to the noisy data. With bigger values of beta, like beta=0.98, we get much smother curve, but it's a little bit shifted to the right, because we average over larger number of example (around 50 for beta=0.98). Beta = 0.9 provides a good balance between these two extremes.



A technique called Stochastic Weight Averaging

STOCHASTIC GRADIENT DESCENT WITH MOMENTUM

We have observed that when it comes to surfaces which have more steep curves in one dimension in comparison to another, SGD faces trouble in navigation. In this case, SGD try to make hesitant progress along the bottom towards local optimum, as SGD oscillates across the sloped of the ravine.

Momentum:- A method that helps SGD accelerate in the required direction and dampens oscillation.

It is done by adding a fraction γ of the update vector of the past time step to the current update vector:

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

$$\theta = \theta - v_t$$

Here γ is usually set to 0.9 or a similar value.

When we use momentum and push a ball down a hill, the ball accumulates momentum as it rolls downhill and keeps spinning faster, the speed becomes constant, once it reaches the terminal velocity. The momentum term increases where the gradients point in the same directions and reduces for dimensions where it changes directions.

The same thing happens to our parameter updates; the momentum term increases for dimensions whose gradients is in the same directions and reduces for dimensions whose gradients change directions. This results in faster convergence and reduced oscillation.

With Stochastic Gradient Descent the exact derivate of cost function is not computed. Instead, we're estimating it on a small batch. Which means it doesn't always go in the optimal direction, because our derivatives can be noisy. Hence it can be deduced that exponentially weighed averages can provide us a better estimate. This estimate is closer to the actual derivate than the noisy calculations. Because of this momentum can work better than classic SGD.

Ravine is the area where the surface curves much more steeply in one dimension as in the real world. Ravines are common near local minima in deep learning and SGD has troubles navigating them. SGD will tend to give output that are across this area since the negative gradient will point down one of the steep sides and not along the ravine towards the optimum. Thus, momentum helps steer gradients in the right direction.

RECURSIVE STOCHASTIC GRADIENT ESTIMATES

This handles the non-convex problem optimization by recursive/adaptive updates of gradient estimates gathered stochastically. It uses a fixed snapshot point for the entire epoch.

For many problems, recursive gradient estimate g^t may decay fast; gt may quickly deviate from the target gradient ∇F(xt), and progress stalls as gt cannot guarantee enough descent.

To tackle this, we can reset gt every few iterations to calibrate with the true batch gradient.

gt is not an unbiased estimate of ∇F(xt)

$$\mathbb{E}[\boldsymbol{g}^t \mid \text{everything prior to } \boldsymbol{x}_s^t] = \nabla F(\boldsymbol{x}^t) \underbrace{-\nabla F(\boldsymbol{x}^{t-1}) + \boldsymbol{g}^{t-1}}_{\neq \mathbf{0}}$$

STOCHASTIC RECURSIVE GRADIENT ALGORITHM (SARAH)

Algorithm 12.4 SARAH (Nguyen et al. '17)

1: **for** $s = 1, 2, \ldots, S$ **do**
2: $\quad \boldsymbol{x}_s^0 \leftarrow \boldsymbol{x}_{s-1}^{m+1}$, and compute $\underbrace{\boldsymbol{g}_s^0 = \nabla F(\boldsymbol{x}_s^0)}_{\text{batch gradient}}$ // restart $\boldsymbol{g}$ anew
3: $\quad \boldsymbol{x}_s^1 = \boldsymbol{x}_s^0 - \eta \boldsymbol{g}_s^0$
4: $\quad$ **for** $t = 1, \ldots, m$ **do**
5: $\quad\quad$ choose i_t uniformly from $\{1, \ldots, n\}$
6: $\quad\quad \boldsymbol{g}_s^t = \underbrace{\nabla f_{i_t}(\boldsymbol{x}_s^t) - \nabla f_{i_t}(\boldsymbol{x}_s^{t-1})}_{\text{stochastic gradient}} + \boldsymbol{g}_s^{t-1}$
7: $\quad\quad \boldsymbol{x}_s^{t+1} = \boldsymbol{x}_s^t - \eta \boldsymbol{g}_s^t$

iteration complexity for finding ε-approximate stationary point (i.e. k∇F(x)k2 ≤ ε):

$$O\left(n + \frac{L\sqrt{n}}{\varepsilon^2}\right) \qquad (\text{setting } m \asymp n, \eta \asymp \tfrac{1}{L\sqrt{m}})$$