Proceedings of International Mathematical Sciences
ISSN: 2717-6355, URL: https://dergipark.org.tr/tr/pub/pims
Volume 4 Issue 2 (2022), Pages 77-87.
DOI: https://doi.org/10.47086/pims.1170457

COMPARATIVE ANALYSIS OF FIRST AND SECOND ORDER METHODS FOR OPTIMIZATION IN NEURAL NETWORKS

AURAS KHANAL*, MEHMET DIK**
*BELOIT COLLEGE, BELOIT, UNITED STATES. 0000-0001-8621-9879
**BELOIT COLLEGE, BELOIT, UNITED STATES. 0000-0003-0643-2771

Abstract. Artificial Neural Networks are fine tuned to yield the best performance through an iterative process where the values of their parameters are altered. Optimization is the preferred method to determine the parameters that yield the minima of the loss function, an evaluation metric for ANN's. However, the process of finding an optimal model which has minimum loss faces several obstacles, the most notable being the efficiency and rate of convergence to the minima of the loss function. Such optimization efficiency is imperative to reduce the use of computational resources and time when training Neural Network models. This paper reviews and compares the intuition and effectiveness of existing optimization algorithms such as Gradient Descent, Gradient Descent with Momentum, RMSProp and Adam that implement first order derivatives, and Newton's Method that utilizes second order derivatives for convergence. It also explores the possibility to combine and leverage first and second order optimization techniques for improved performance when training Artificial Neural Networks.

1. Introduction

In Mathematics, **optimization** is the process of maximizing or minimizing a real function by finding the best set of input values under certain conditions or constraints. It can be defined as:

$$\arg\min_{\theta} f(\theta) \;\; or \;\; \arg\max_{\theta} f(\theta) \tag{1.1}$$

Here, θ represents the arguments for the function. The concept of optimization is used in abundance in real life: in GPS Systems, financial companies and airline reservations. Similarly, optimization methods are equally important in the field of Artificial Intelligence and Machine Learning, especially for their application to Artificial Neural Network Models.

2020 *Mathematics Subject Classification.* Primary: 68T07.
Key words and phrases. Optimization; Artificial Neural Networks; Gradient Descent.

Submitted on 03.09.2022, Accepted on 04.10.2022.
Communicated by Hacer ŞENGÜL KANDEMIR and Muhammed ÇINAR.

1.1. **Artificial Neural Networks.** Artificial Neural Networks are Machine Learning Models that mimic the functionality of biological neurons. They implement learning algorithms which are fine-tuned by training on data to improve their accuracy. ANN's comprise of an Input layer, a single or multiple hidden layers, and an output layer. Each layer consists of a particular number of artificial neurons or nodes and each node receives an input from all the nodes in the previous layer and outputs values to all the nodes in the next layer. Each of the individual connections between the nodes are assigned parameter values called weights and biases. These parameters are altered and tuned during the learning process using the input training data for optimal performance and accuracy.

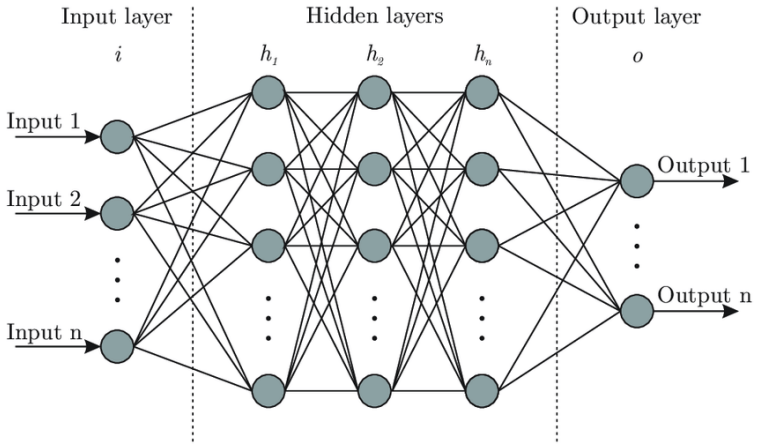

FIGURE 1. Artificial Neural Network with 3 hidden layers [1]

During the initial training or learning process of a Neural Network, the features or attributes of each individual data record are passed into the first hidden layer as an input. Each of these input features have some weight value attached to them and all the inputs are connected to each neuron in the hidden layer. Using the inputs, the output z of neuron j in the hidden layer is,

$$z_j = f\left(b + \sum_{i=1}^{n} x_i w_i\right), \tag{1.2}$$

where n is the total number of input features, b denotes the bias value, w denotes the weights for particular features and f is the activation or transfer function of the layer. The activation function is a linear or non linear function that determines the output of the neuron. Examples of activation functions are Linear function, Sigmoid function, Hyperbolic Tangent function, Rectified Linear Unit function, etc. The output of each neuron in a layer is passed as an input to all the neurons in the next layer with their own weights and biases values. Similarly, each layer receives its own input and calculates an output and passes it to the next layer. This process is also known as forward propagation. The last layer, which is the output layer receives the inputs from the neurons of the last hidden layer and provides the output of the neural network.

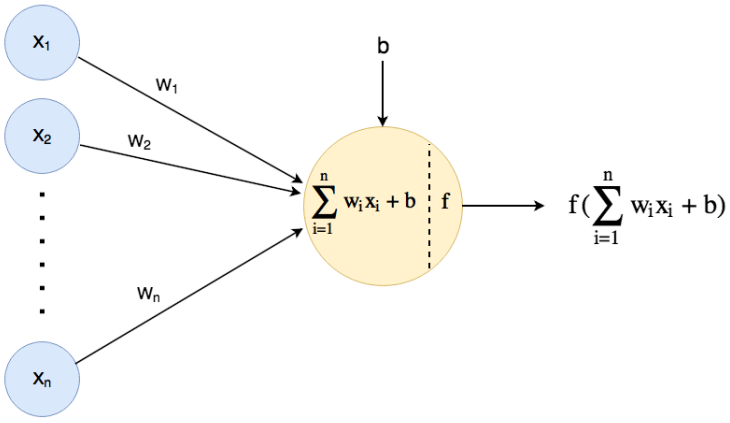

FIGURE 2. An artificial neuron with n inputs [2]

1.2. **Optimization in Neural Networks.** The output of a neural network represents the prediction for a particular input. The initialized values of the weights and biases do not usually produce an accurate prediction. Hence, ANN's use an iterative process where these parameters are adjusted in each iteration to increase the prediction accuracy. This step is also known as Backpropagation. Backpropagation is the process of updating and finding the optimal values of weights or coefficients which helps the model to minimize the error i.e difference between the actual and predicted values.[3] The difference between the predicted and the actual value for an individual data sample is calculated using a loss function. The most common loss function for regression problems is given by,

$$L\left(\widehat{y}\right) = \frac{1}{2}\left(\widehat{y} - y\right)^2, \tag{1.3}$$

where $\widehat{y}$ represents the neural network prediction and y denotes the actual value. The difference between the predicted and the actual values for all the records in the data set is given by a Cost Function J,

$$J = \frac{1}{2m}\sum_{k=1}^{m} L_k\left(\widehat{y}\right), \tag{1.4}$$

where m denotes the total number of records or data samples.

Neural Networks strive to adjust the parameters of the model to minimize the loss function. Hence, optimization is implemented to find the values of the weights and biases that will engender the minimum value of the loss function (closest to zero).

$$\arg\min_{(W,b)} \sum_{k=1}^{m} \frac{1}{2m}\left(f\left(W^T X_k + b\right) - y_k\right)^2, \tag{1.5}$$

where W denotes an $n \times j$ weight matrix of the output layer(n is the number of input features coming from the previous layer, and j is the number of outputs of the network), b denotes the bias of the output layer, and f denotes the activation

function of the output layer. Equation 1.5 can be understood as a combination of equations 1.2,1.3 and 1.4.

There are several different approaches of optimization to find the minimum of a function. These methods usually utilize the first and second derivatives of the function with respect to the parameters. The efficiency of such methods are evaluated through the computational cost (memory) and time cost for optimization.

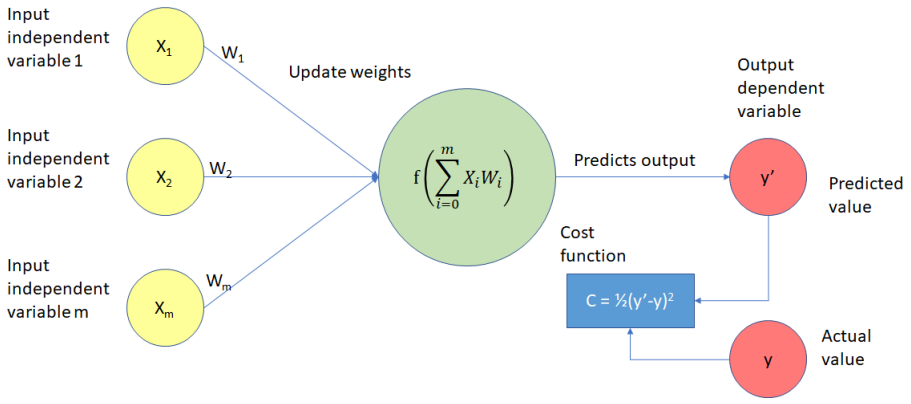

FIGURE 3. Forward and Backpropagation in a single neuron network.[4]

2. First Order Optimization Methods

First Order Optimization refers to methods that utilize the first derivative of the target function with respect to the parameters. It can only be applied to functions that are differentiable and continuous. One of the most commonly used first order methods is Gradient Descent where the gradient is used to descend down the curve of the function.

2.1. Gradient Descent.

Gradient Descent is a first order iterative method for optimization where the idea is to take repeated steps to update the parameters in the opposite direction of the gradient. For a vector $\theta = [\theta_1, \theta_2, \dots \theta_n]$ where θ_n represents the parameters for the cost function J, the updated values after a particular iteration is given by,

$$\theta = \theta - \eta \nabla_\theta J(\theta), \tag{2.1}$$

where η denotes the learning rate or the size of the steps that are taken to reach the minimum and

$$\nabla_\theta J(\theta) = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \dots \frac{\partial J}{\partial \theta_n} \right]$$

With respect to Neural Networks, there are three variants of gradient descent that are used for convergence: Batch Gradient Descent, Mini Batch Gradient Descent and Stochastic Gradient Descent. These variants differ in terms of the number of samples used to calculate the loss function gradient for each parameter update step.

2.1.1. *Batch Gradient Descent.* Batch or Vanilla Gradient Descent computes the gradient of the cost function with respect to the parameters θ for the entire training dataset.[5] Here, $\nabla_\theta J(\theta) = \sum_{i=1}^{m} \nabla_\theta J(\theta)$, where m denotes the total number of records in the dataset. Hence, the summation of the gradients of the entire dataset needs to be calculated to perform just one parameter update. In other words, the parameters are updated once in each epoch (one epoch refers to one iteration through the entire training data).[6] Therefore, for a large dataset, Batch Gradient Descent is really slow since a single update is performed after going through all the records. However, because the information of the entire dataset is being evaluated each time the parameters are updated, the convergence path taken using Batch Gradient Descent is smooth and free of noise which accounts for a more direct path towards the minimum.

2.1.2. *Stochastic Gradient Descent.* In contrast, Stochastic Gradient Descent performs a parameter update after each record in the dataset. In terms of SGD, $\nabla_\theta J(\theta) = \nabla_\theta^{(i)} J(\theta)$, where (i) denotes a random record. Hence, the parameters are updated m times in one epoch. As the gradient of the cost function with respect to the parameters for each record will vary largely, the convergence path using SGD is full of noise and oscillations in different directions compared to Batch Gradient Descent. Hence, SGD requires higher number of iterations to reach the minima. Furthermore, SGD performs frequent updates with a high variance that causes the objective function to fluctuate heavily. The random and drastic changes in parameter values due to the nature of SGD enables it to jump out of local minimas into new and potentially better minima. Due to the high variance of the gradients for each record, SGD never actually converges completely to the minima but rather oscillates around the region. However, SGD provides advantages of updating the parameters almost instantly and the escaping local minimas. Furthermore, it is less computationally expensive and converges faster than Batch Gradient Descent when the dataset is very large.

2.1.3. *Mini-Batch Gradient Descent.* Batch Gradient Descent updates parameters after going through the entire dataset while Stochastic Gradient Descent performs updates after each record. Mini-Batch Gradient Descent leverages the efficiency of both these methods. It divides the total dataset into small batches and updates the parameters after going through each batch. Hence, $\nabla_\theta J(\theta) = \sum_{i=1}^{k} \nabla_\theta J(\theta)$, where $k << m$. By taking a small sample of the total data, Mini-Batch Gradient Descent eliminates the heavy computational cost for large datasets

using Batch Gradient Descent while reducing the noise and variance of Stochastic Gradient Descent leading to a more stable convergence.

2.2. **Limitations of Gradient Descent.** Although it addresses the limitations of the previous two Gradient Descent methods, Mini-Batch Gradient Descent has several limitations.

(1) **Choice of Learning Rate**

Learning Rates denotes the size of the steps taken during convergence. If a learning rate is too small, the parameter updates will be insignificant and the number of iterations required to converge will be huge. If a learning rate is too big, the update might overshoot and jump over the minima to

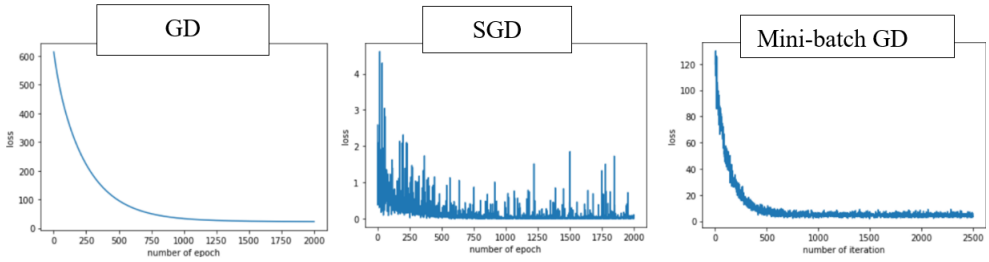

FIGURE 4. Loss Function fluctuations after parameter updates.[7]

the opposite side causing the loss function to fluctuate around the minima or in worst cases, even diverge.

(2) **Saddle Points and Local Minima**

Saddle points are flat regions of a function where the partial derivatives of the cost function with respect to its parameters are opposite in nature i.e areas where one dimension has a positive partial and another has a negative partial. Such points are usually surrounded by a plateau which makes it very difficult for Gradient Descent to escape as the gradient is close to zero in all dimensions.[5]

(3) **Noisy Convergence** The performance of Artificial Neural Networks increases with the size of the data. Hence, Batch Gradient Descent is rarely used due to its high computational cost. However, as SGD and Mini-Batch Gradient Descent update parameters using only a small portion of the total data, the convergence path using these methods have high variance. This increases the number of oscillations to reach the minimum as the path taken is not direct. The extent of noise depends on the size of the batch used (the larger the batch the smoother the convergence path).

Hence, due to the impracticality in application of Mini-Batch Gradient Descent, other first order methods were developed to address and eliminate such limitations.

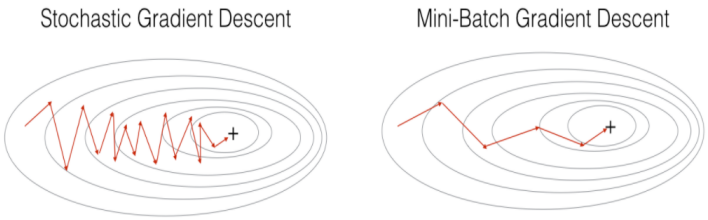

FIGURE 5. Convergence path using a contour map.[8]

2.3. **Gradient Descent with Momentum.** One of the major challenges with Gradient Descent is that the updated value of a parameter depends only on the gradient of the cost function at the previous parameter value. Therefore, it gets

stuck in areas where the gradient is very close to zero in all dimensions. Furthermore, the number of iterations to converge is higher due to the large variance in the gradient using SGD and Mini-Batch Gradient Descent. Gradient Descent with Momentum addresses both these challenges by using Exponentially Weighted Averages of the gradients to update the parameters. Exponentially Weighted Averages is used in sequential noisy data to reduce the noise and smoothen the data.[7] Referring to Figure 5 above, the vertical oscillations slows gradient descent and prevents the use of a high learning rate. By using the exponentially weighted averages, the partial derivative with respect to the vertical direction has an average closer to zero as it is in both (positive and negative) directions.[9] In contrast, the partial derivative with respect to the horizontal direction is always positive, hence the average in that direction will be large. This allows for a more direct convergence path to the minima. The exponentially weighted average of the gradient for a given iteration t is,

$$M_{d\theta_t} = \beta M_{d\theta_{t-1}} + (1-\beta)\nabla_{\theta_t} J(\theta),$$

where $0 < \beta \leq 1$. Similarly,

$$M_{d\theta_{t-1}} = \beta M_{d\theta_{t-2}} + (1-\beta)\nabla_{\theta_{t-1}} J(\theta)$$
$$M_{d\theta_{t-2}} = \beta M_{d\theta_{t-3}} + (1-\beta)\nabla_{\theta_{t-2}} J(\theta),$$

and $M_{d\theta_1} = 0$. $M_{d\theta_1}$ is also called the momentum term. Expanding $M_{d\theta_t}$,

$$M_{d\theta_t} = \beta\left(\beta(\beta M_{d\theta_{t-3}} + (1-\beta)\nabla_{\theta_{t-2}} J(\theta)\right) + (1-\beta)\nabla_{\theta_{t-1}} J(\theta)) + (1-\beta)\nabla_{\theta_t} J(\theta) \tag{2.2}$$

Hence, using Gradient Descent with Momentum, for an iteration t, the new parameter value is dependent on the gradients of all previous iterations. Here, β is the hyper-parameter that determines the degree of smoothness of the convergence path and is usually 0.9. Consequently, the weight assigned to the averages of the previous iterations is larger compared to the weight assigned to the gradient at the current point. The parameters are then updated using the formula,

$$\theta = \theta - \eta M_{d\theta_t} \tag{2.3}$$

For partial derivatives with respect to dimensions whose values oscillate between positive and negative, as the averages are closer to zero, the oscillations in those dimensions are reduced. Inversely, the momentum term increases for dimensions whose gradients point in the same directions, resulting in a larger update step after each iteration and gaining faster convergence.

2.4. **Root Mean Square Propagation.** Although Gradient Descent with Momentum reduces oscillations in dimensions where the partials point in opposite directions in each iteration, it is more effective in accelerating the convergence in dimensions whose derivatives point in the same direction. However, this poses some challenges. For a large t (located closer to minima), the momentum term for these dimensions and consequently, the parameter updates will be very large. Hence, for parameter updates near the minima, the risk of overshooting and missing the minima due to the large momentum term is very high. Similar to a ball rolling down a large cliff and going back and forth across the bottom several times, GD with Momentum increases the number of iterations to settle down into the minima. Root Mean Square Propagation or RMSProp eliminates this risk. While momentum accelerates our search in the direction of the minima, RMSProp impedes our search in the direction of the oscillations.[10] RMSProp implements this by using

an adaptive learning rate, or a learning rate that changes in each iteration. In RMSProp, after each iteration the learning rate decreases independently for each dimension based on the partial derivative of that dimension at that particular iteration . If a dimension/ parameter has a higher partial derivative, than its learning rate is lower compared to a parameter with a lower partial derivative value. Using RMSProp, the new parameter values θ,

$$\theta = \theta - \frac{\eta}{\sqrt{V_{d\theta_t}} + \epsilon} \cdot \nabla_{\theta_t} J(\theta) \tag{2.4}$$

where

$$V_{d\theta_t} = \alpha V_{d\theta_{t-1}} + (1-\alpha) \nabla_{\theta_t} J(\theta)^2,$$

Here, ϵ is a very small value, usually 10^{-8} and $0 < \alpha \leq 1$.

Similar to GD with Momentum, RMSProp also uses an exponential weighted average. However, instead of taking the average of the gradient, it uses the exponentially weighted average of the squared gradient. This is to ensure the sole dependency of the adaptive learning rate on the magnitude of the partials of the different parameters and not the signs. Hence, in contrast to the Momentum term $M_{d\theta_t}$, if the partials of a parameter at different iterations has opposite signs, their exponentially weighted average won't cancel out but instead be added, increasing $V_{d\theta_t}$. The square root is added in the denominator of 2.4 to ensure that the learning rate isn't too small. Therefore for situations like the one shown in Figure 5, the step taken in both directions will decrease for increasing iterations. However, the direction that is steeper will have a significantly smaller step compared to shallower directions.

2.5. **Adaptive Moment Estimation.** In Gradient Descent with Momentum, the step size for parameters which have the same signed partial derivatives increases after each iteration. In contrast, in RMSProp the step size for parameters decreases after each iteration but to a greater extent for ones which have a higher magnitude partial derivative. Adaptive Moment Estimation or Adam is an optimization algorithm that is the combination of the features of both of these first order methods. Adam utilizes the acceleration that is provided by GD with Momentum, but to ensure that the step size doesn't infinitely increase towards latter iterations, it uses the RMSProp term to decrease the learning rate to limit the updates as the iterations increase. Hence, it leverages the increase in the Momentum term by decreasing the learning rate. The Momentum and RMSProp terms are given by,

$$M_{d\theta_t} = \beta M_{d\theta_{t-1}} + (1-\beta) \nabla_{\theta_t} J(\theta),$$

and

$$V_{d\theta_t} = \alpha V_{d\theta_{t-1}} + (1-\alpha) \nabla_{\theta_t} J(\theta)^2,$$

For a parameter θ, the updated value using Adam is given by,

$$\theta = \theta - \frac{\eta}{\sqrt{V_{d\theta_t}} + \epsilon} \cdot M_{d\theta_t} \tag{2.5}$$

Adam is the most widely used optimization method because it performs really well in optimization test functions compared to other algorithms.

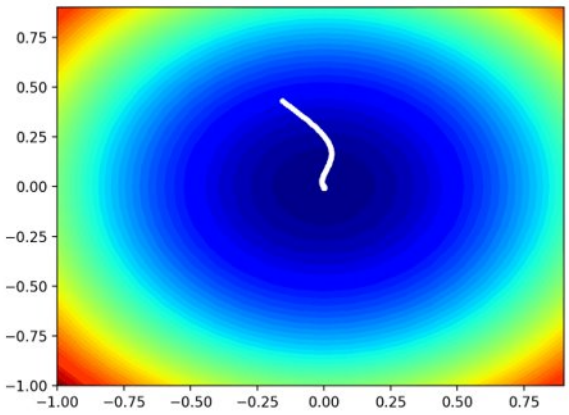

FIGURE 6. Contour Plot of the Test Objective Function With Adam [11]

3. Second Order Optimization Methods

Second Order Optimization methods are a separate set of methods for optimization that differ from the traditional gradient descent ideology. Instead of using the gradient of the objective function at a particular point to update the parameters, second order methods utilize the Hessian of the function. The gradient of a function is a vector where each element represents the partial derivative of the function with respect to the individual parameters. For a function with a total of p parameters, the Hessian is a $p \times p$ matrix where each element represents the second partial derivative of the functions with respect to the parameters. The Hessian of function $f(x_1, x_2, \ldots, x_n)$ is given by,

$$H_f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

3.1. Newton's Method.

Newton's method is an optimization technique that utilizes the approximation of a function using Taylor's Expansion to the second order. For a function $f(x)$, the Taylor expansion to the second order about a certain point x_0 in the domain is given by,

$$f_2(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2$$

The approximation function f_2 is a quadratic function about point x_0. Newton's method finds the value of x, where f_2 is a minimum, and assigns that value to x_0,

i.e $x_0 = \arg\min_x f_2(x)$. Taylor's expansion is used to approximate a new f_2 about the updated x_0. The minima of the new approximation is found and x_0 is updated again. Hence, Newton's method iteratively updates the parameter x_0 to values where newly approximated Taylor Functions are a minimum until the minimum of the actual function f is reached.

For an approximation f_2, a point x is a maximum or minimum only if the gradient at that point is 0.

$$\frac{df_2}{d\left(x - x_0\right)} = 0$$

$$\frac{d\left[f\left(x_0\right) + f'\left(x_0\right)\left(x - x_0\right) + \frac{1}{2}f''\left(x_0\right)\left(x - x_0\right)^2\right]}{d\left(x - x_0\right)} = 0$$

$$f'\left(x_0\right) + f''\left(x_0\right)\left(x - x_0\right) = 0$$

$$x = x_0 - \frac{f'\left(x_0\right)}{f''\left(x_0\right)} \tag{3.1}$$

For an approximation of a multi-variable function, $f_2(\theta)$ where θ is a vector of the parameters,

$$\theta_{t+1} = \theta_t - \left[H\left(\theta_t\right)\right]^{-1} \nabla_\theta f\left(\theta_t\right) \tag{3.2}$$

Compared to gradient descent, Newton's method is extremely fast. For a suitably chosen learning , gradient descent takes 229 steps to converge to the minimum whereas Newton's method converges to the minimum in only 6 steps.[12]

Although it is very efficient, Newton's method has numerous limitations in application to Neural Networks. ANN's usually have thousands of parameters and are non-convex by nature. However, Newton's method isn't applicable to non-convex functions. The initialization of the parameters to areas closer to the maximum of the function or at points where the Hessian is negative-definite can lead to a quadratic approximation that is concave. For such an approximation, the parameter update will lead towards the maximum point of the approximated concave function instead of the minimum that is required. Hence, Newton's method can lead to an increase in the value of the loss which is undesired. Furthermore, Newtonian methods are very computationally expensive. The calculation of the Hessian is itself an $0(N^2)$ and inverting the Hessian is $0(N^3)$ compared to Gradient Descent methods which scale at $O(N)$.[12] Additionally, saddle points where the Gradient and Hessian are almost zero might lead to computationally inaccurate values and slow updates of the parameters. Such extreme limitations render the use of Newtonian methods in Neural Networks useless.

However, there are methods not covered in this paper called Quasi-Newton methods that eliminate the large computational cost of traditional Newton's method while preserving optimization efficiency. Quasi-Newton methods utilize an approximation of the Hessian using a generalized secant method, eliminating the need to invert the Hessian. Hence, they have a computational complexity of $O(N^2)$ compared to $O(N^3)$ for Newton's method, while retaining most of the efficiency when converging using Newton's method.

4. Conclusion and Future Work

Although Quasi-Newton methods such as the Broyden—Fletcher—Goldfarb—Shanno algorithm represent significant progress in the field of second order optimization, the lack of precision in the calculation of the Hessian can sometimes lead to slower convergence.[13] Hence, instead of approximating the Hessian, leveraging Gradient Descent and Newton's Method can possibly lead to better performance. An optimization algorithm can be introduced which consists of two stages, beginning with Gradient Descent and ending with Newton's method. The transition into Newton's method can be implemented at a point where the Hessian is a positive-definite matrix. However, since the calculation of the Hessian is computationally expensive, this prompts further research on finding a general method to determine the point of transition into Newton's Method.

References

[1] F. Bre, J.M. Gimenez, and V.D. Fachinotti, Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158 (2017).
[2] Hvidberrrg, Activation functions in artificial neural networks.
[3] Deepanshi, Artificial neural network: Beginners guide to ann. *Analytics Vidhya*, (2021).
[4] M. Z. Mulla, Cost, activation, loss function || neural network || deep learning. what are these? *Medium*, (2020).
[5] S.Ruder, An overview of gradient descent optimization algorithms. *Ruder.io*, (2020).
[6] K. Pykes, Gradient descent. *Towards Data Science*, (2020).
[7] G. Mayanglambam, Deep learning optimizers. *Towards Data Science*, (2020).
[8] i2tutorials, Explain brief about mini batch gradient descent. *i2tutorials*, (2019).
[9] B. S. Shankar, Gradient descent with momentum. *Medium*, (2020)
[10] A. Kathuria, Intro to optimization in deep learning: Momentum, rmsprop and adam. *Paperspace Blog*, (2018).
[11] J.Brownlee, Code adam optimization algorithm from scratch. *Machine Learning Mastery*, (2021).
[12] A.Lam, Bfgs in a nutshell: An introduction to quasi newton methods. *Towards Data Science*, (2020).
[13] V.Cericola, Quasi-Newton methods. *Northwestern University Open Text Book on Process Optimization*, (2015)

Auras Khanal,
700 College St, Beloit, WI 53511, United States, Phone: +1 4159108043 0000-0001-8621-9879
Email address: khanalauras@outlook.com

Mehmet Dik,
700 College St, Beloit, WI 53511, United States, Phone: +1 8152264135 0000-0003-0643-2771
Email address: dikm@beloit.edu