An Empirical Study: Selection of Optimization Algorithms for Neural Networks

Yuan-Ting Hsieh
University of Wisconsin-Madison
yhsieh28@wisc.edu

Han Wang
University of Wisconsin-Madison
hwang729@wisc.edu

Wen-Fu Lee
University of Wisconsin-Madison
wlee256@wisc.edu

Abstract

This project presents our understanding and implementation of gradient descent optimization algorithms, including SGD, AdaGrad, and Adam. We illustrate some key insights of these methods on toy examples. We also report empirical results of these techniques when training a two-layer neural network on MNIST dataset.

1 Introduction

Nowadays, gradient descent techniques, such as SGD, AdaGrad [1], Adam [2], are extensively used in several areas to optimize parameters of learning models. However, there is no much study and survey to answer the question like this: given limited time and a certain size of the dataset, which optimization technique is recommended and should be adopted? In this project, we implement several state-of-the-art gradient descent algorithms for optimizing a neural network. Based on these optimization algorithms, we present the empirical result of the relationships between their performance, training time, and dataset sizes. This report is organized as follows: section 2 provides overview and insights of these methods. Section 3 describes how to apply them to neural networks. We give the evaluation results in section 4 and conclude our work in section 5.

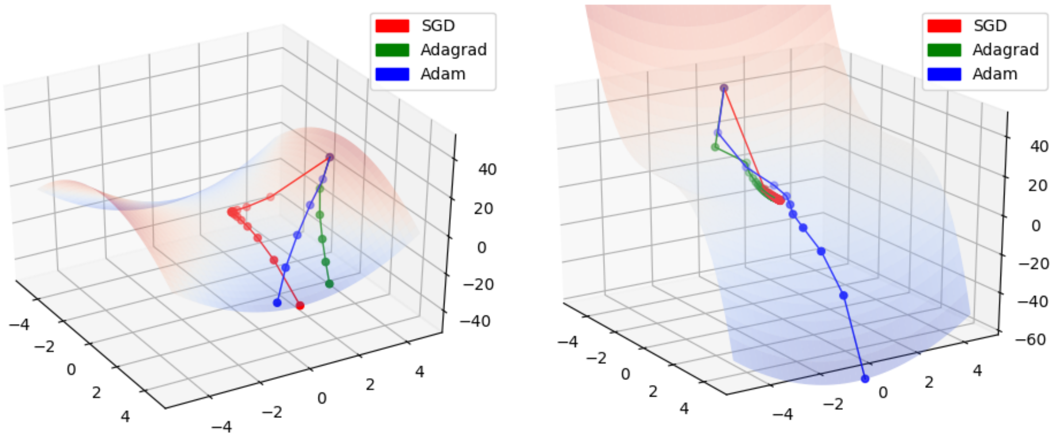
2 Overview of Gradient Descent Optimization Algorithms

2.1 Gradient Descent

Gradient descent is a first-order iterative optimization algorithms for finding the minimum of a function. Denoted our objective function as $J(\theta)$ where $\theta \in \mathbb{R}^d$, we want to find a θ that minimizes the function. To find it, we iteratively update θ such that $J(\theta_{t+1})$ is lower than $J(\theta_t)$. Utilizing first order Taylor expansion: $J(\theta_{t+1}) = J(\theta_t + v) \approx J(\theta_t) + v\nabla J(\theta)$. The optimal v is the opposite direction vector of the gradient, that is $v \propto -\frac{\nabla J(\theta_t)}{\|\nabla J(\theta_t)\|}$. Choose a special length of $v = -\eta \nabla J(\theta_t)$, we arrive at our regular GD update rules $\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$, where η is called the learning rate and $\nabla J(\theta_t)$ is the gradient of the function. If the gradient of cost function is calculated on the whole dataset, then that is called batch gradient descent. But that needs more time to calculated and slower convergence. Hence in this project, we consider the setting of stochastic gradient descent, that is we will update our parameters each time we see a sample.

2.2 Optimization Techniques

In this section, we are going to explain the intuition and update rules of SGD, AdaGrad, and Adam.



(a) Objective function $f(x) = y^2 - x^2$ and start point $(x, y) = (0.001, 4)$.

(b) Objective function $f(x) = y^2 - 0.7x^3$ and start point $(x, y) = (-4, 0.1)$.

Figure 1: Comparison of the learning behavior among SGD, AdaGrad, and Adam on toy objectives.

2.2.1 Stochastic Gradient Descent

The update function is given as:

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla J(\theta_t)$$

This is the most simple version with fix learning rate and just subtract the gradient. However, this typical gradient descent optimization algorithm has the following drawbacks:

(1) Learning rate won't change w.r.t. time, it keeps constant

(2) Learning rate is the same in all direction

If the initial learning rate is too big, (1) would make it more difficult to converge when we are close to the minima, even leads to diverging; On the other hand, if the initial learning rate is too small, then the converging time and runtime will take too long. So we want an adaptive learning rate, which is larger in the beginning and shrinks while time proceeds. While (2) would make it harder to escape from saddle points.

These behaviors can be seen in Fig. 1a, as Adagrad and Adam can escape easily from the saddle point, SGD spends more time there. As the function surfaces become more complicated, it will have problems escaping from the saddle point. We can verify this in Fig. 1b.

2.2.2 AdaGrad

The authors of AdaGrad [1] is trying to solve the problems of SGD. The update rule is written as:

$$\begin{gathered} G_{t+1} \leftarrow G_t + \nabla J(\theta_t)^2 \\ \theta_{t+1} \leftarrow \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \nabla J(\theta_t) \end{gathered}$$

We can see that it replaces fixed learning rate to "adaptive" learning rate, which is the original learning rate divide by G_t. Where each element in G_t is the squared sum of all past gradients in that direction. The intuition behind this method is that for the direction that hasn't update much (which will have smaller G_t value in that direction), the learning rate of that direction will be bigger. Which the authors state that it allows us to find needles in haystacks in the form of very predictive but rarely seen features.

This behavior can also be seen in Fig. 1a. AdaGrad is not following the steepest direction to go down only x-direction first. Also, it goes down y-direction simultaneously. We can see it escapes from the saddle point faster than SGD.

2.2.3 Adam

While the results and proofs of AdaGrad look very promising, it suffers from a major problem. Since it keeps adding the square gradient to G. Its learning rate will keep decreasing, so it might become unreasonably small after some iterations and stuck at some point. This behavior can be seen in Fig. 1b. Adam [2] aims at solving this issue. Instead of using the total sum of square gradients in that direction, Adam wants to use a sliding window to consider only recent updates. That is it only consider the sum of square gradient from iteration $t - w$ to t, but this requires the memorization of each time step's gradient, which is costly as time expands. So it uses an exponential moving average model to get the sum of square gradients term (v_t), the gradient of the time that too far from now will have little effect on current updates. It also does the same moving average for the original gradient (m_t). And because it initialize m_t and v_t with 0, it requires bias correction. The update formula of Adam is given as:

$$m_{t+1} \leftarrow \beta_1 m_t + (1-\beta_1)\nabla J(\theta_t)$$
$$v_{t+1} \leftarrow \beta_2 v_t + (1-\beta_2)\nabla J(\theta_t)^2$$

Then do bias correction on these terms

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^t}, \hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^t}$$

Finally we get,

$$\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

As Fig. 1a shows, the step of AdaGrad will shrink as time goes by, while Adam have bigger steps in later iterations. Also, seeing from Fig. 1b, Adam will not stuck at some points that AdaGrad struggles.

3 Gradient Descent for Neural Networks

Since we use neural networks as our evaluation model, we first derive the gradients of cost function w.r.t. parameters in neural networks. Then we can plug those gradients into our update rules provided above. Finally, we use the back-propagation algorithm, which updates the weights from the deepest layer back to the shallowest layer, to avoid repeated calculations.

3.1 Neural Network Structure

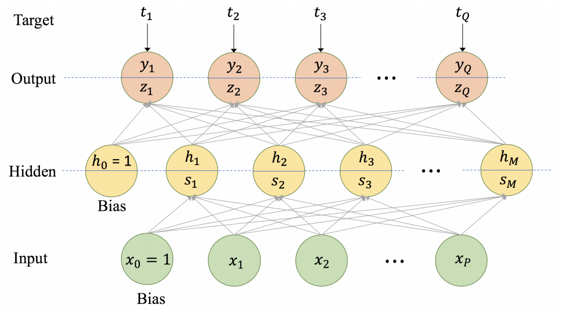


Figure 2: Neural Network

Specifically, our neural network has one hidden layer and one output layer. We choose logistic function as activation function. The structure is shown as Fig. 2. The outputs h of the hidden layer are computed by applying the logistic function to the weighted input sums s.

$$h_j = \frac{1}{1+e^{-s_j}}, \; s_j = \sum_{i=0}^{P} x_i w_{ji}$$

where w_{ji} is the weight connecting the input unit x_i to the j unit in the hidden layer. Likewise, the outputs y are computed by applying the logistic function to the weighted sums z of the hidden units.

$$y_k = \frac{1}{1+e^{-z_k}}, \; z_k = \sum_{j=0}^{M} h_j w_{kj}$$

where w_{kj} is the weight connecting the j unit in the hidden layer to the k unit in the output layer.

3.2 Gradients Derivation of the Neural Network

In a binary classification task, it is standard to use the cross-entropy as the loss function. Thus, the cross-entropy loss for a single training sample in our model is calculated as follows:

$$E = -\sum_{k=1}^{Q}(t_k log(y_k) + (1-t_k)log(1-y_k))$$

The derivative of the loss with respect to each weight connecting the j hidden unit to the k output unit can be computed using the chain rule.

$$\begin{aligned}\frac{\partial E}{\partial w_{kj}} &= \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}\frac{\partial z_k}{\partial w_{kj}} \\ &= (\frac{-t_k}{y_k} + \frac{1-t_k}{1-y_k})(y_k(1-y_k))(h_j) \\ &= (\frac{y_k - t_k}{y_k(1-y_k)})(y_k(1-y_k))(h_j) \\ &= (y_k - t_k)(h_j)\end{aligned}$$

Likewise, the derivative of the loss with respect to each weight connecting the i input unit to the j hidden unit can be also computed using the chain rule.

$$\begin{aligned}\frac{\partial E}{\partial w_{ji}} &= \sum_{k=1}^{Q}\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial h_j}\frac{\partial h_j}{\partial w_{ji}} \\ &= \sum_{k=1}^{Q}(\frac{y_k - t_k}{y_k(1-y_k)})(y_k(1-y_k)w_{kj})(h_j(1-h_j)x_i) \\ &= \sum_{k=1}^{Q}(y_k - t_k)(w_{kj})(h_j(1-h_j)x_i)\end{aligned}$$

4 Evaluation

To explore the trade-off behavior of each method, we set up several experiments using a 2-layer neural network to do a binary classification task on MNIST dataset. We choose hidden node size as 56, which is the dimension of a picture's total pixels (28 x 28) divided by 14. The reason behind this is because we try to extract a compactly embedded vector from sparse pixel information, the dimension needs to be reasonably small.

4.1 Data Collection and Preprocessing

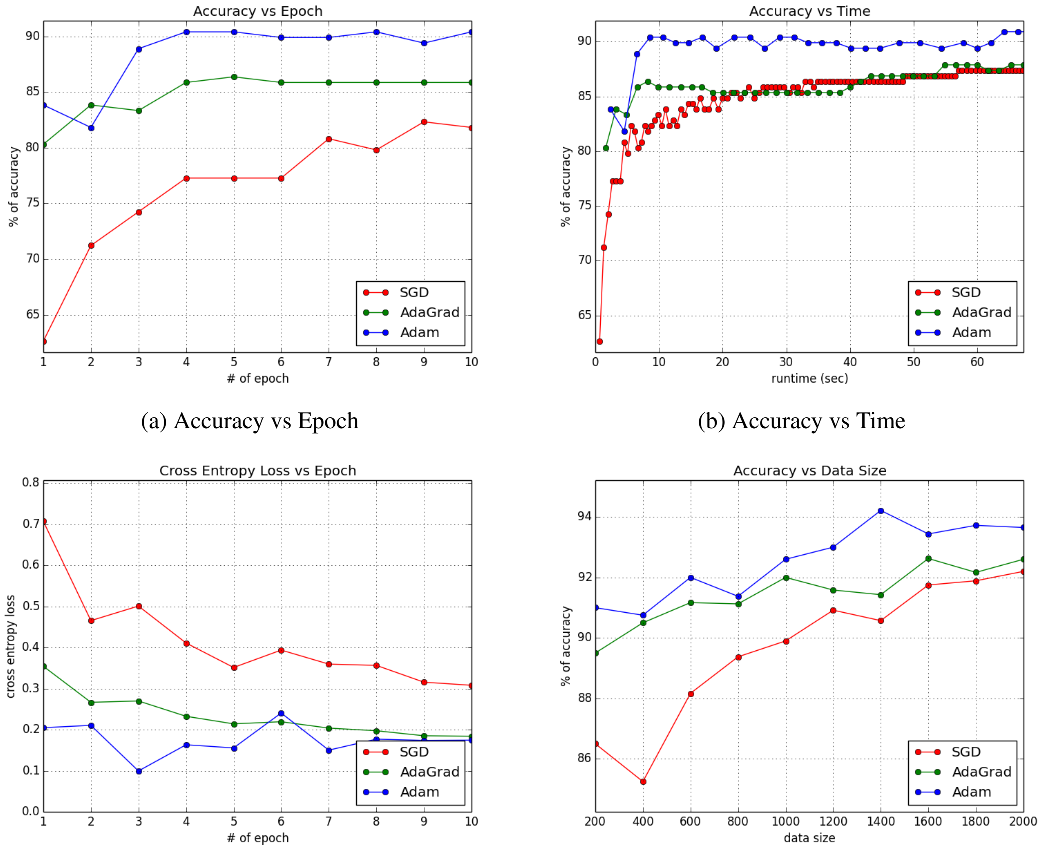
We evaluate the performance of different algorithms on MNIST handwritten digit dataset [3]. In this project, we simplify the problem to binary classification. We set a target digit and choose half of our data with a target digit data while another half of our data with the MNIST data that is not a target digit evenly. For example, if the data size = N and target digit = 0, we would choose $N/2$ digit-0 data and $N/18$ digit-n data for each n ranged from 1 to 9.

4.2 Experiment Result

We have conducted four experiments to compare these methods: accuracy vs epoch, time, dataset size, and cross entropy loss vs epoch.

Fig. 3a shows the performance of classification accuracy over epoch for the three optimization algorithms. Overall, their accuracy increase over epochs. Moreover, under this setting, Adam performs the best regarding accuracy vs epoch.

However, the number of epochs does not reflect the actual computation time, which is an important factor we are also interested in. For example, in our implementation, the average computation time per epoch for SGD, AdaGrad, and Adam is 0.56, 1.58, and 2.07 seconds, respectively. An interesting question



(a) Accuracy vs Epoch

(b) Accuracy vs Time

(c) Cross Entropy Loss vs Epoch

(d) Accuracy vs Data Size

Figure 3: Comparison of SGD, AdaGrad, and Adam

arises that which algorithm can have the best accuracy given the same computation time? Fig. 3b shows some insights into this question. Given the same computation time, SGD can achieve similar results as AdaGrad, but Adam still has the best performance.

In a classification task, typically what we care the most is the classification accuracy of the model. Nevertheless, hard classification error is a step function and not differentiable, which makes it difficult to optimize. This is exactly why we usually use squared error or cross entropy error. In Fig. 3c , we can see that the cross-entropy loss has a decreasing trend. Also, from Fig. 3a we can see that the classification accuracy increases as the loss decreases. This validates the effectiveness of cross entropy loss on neural networks for classification tasks.

Finally, in Fig. 3d we can see that the performance increases as the dataset size increases. The reason is that more data lead to a better learning model and also better generalization ability. An interesting fact we can see here is that when the dataset size is small (200, 400), the difference between the performance of each algorithm is bigger. While the dataset size grows, their difference becomes smaller, but Adam still performs the best. This suggests that in a small dataset, we favor Adam and AdaGrad more than SGD.

5 Conclusion

Among all the methods, Adam performs the best. From SGD to AdaGrad, we know that adaptive learning rate and different learning rate in different directions is very important. From AdaGrad to Adam, we know that sometimes forgetting the past would benefit our learning. This is analogous to life. We have to keep learning and adaptive to the environment, and also we should not be bounded too much by our experience. Next time, when we are trapped in a hard situation (saddle point), we always can escape from it by following these rules. In this way, we can arrive at the optimal of our life.

References

[1] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[2] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.