ELEC4010N Assignment 1 Report

Bo-Rong LAI (20737984)

Abstract

This report can be separated into two main parts. In problem 1, I completed a four-linear layer multi-layer perceptron and trained it on the MNIST dataset. In problem 2, I successfully implemented the LeNet to build a classifier for MNIST dataset. Inside problem 2, by treating some hyper-parameters and optimization methods as variables, I conducted a series of training over different sets of variable combinations and stored the training process for further analysis. I have reviewed several recent articles and explanations on how will these variables impact the training process. Finally, several potential improvements are also included in the final discussion.

1. Problem 1

For this section, I constructed a four-layer linear network in Pytorch. Inside the Init function, I specified the type of different layers and later implemented the network architecture inside the forward function. The main difference between network inputs and raw images is the dimension. The batch size of each batch is 64 where it contains 64, 28*28 pixels, images with a single channel. In order to feed raw images into the network, I flattened all the input before forwarding them into upcoming linear layers. The original input dimension is [64, 1, 28, 28]. After flattening from the second to the fourth dimension, the input dimension became [64, 784]. As a result, the input size of the first linear layer needed to match 784. Following the question, the output size of the last layer was set to 10 which means there are ten classes. Afterward, I applied the cross-entropy loss function to this model since the problem here is a classification problem. The cross-entropy function is generally considered useful for classification tasks as it describes the likelihood and the errors between the model and each data point. By feeding the outputs and their corresponding ground truth labels into the loss function, I obtained a loss tensor that can be utilized later while running the gradient descending. One hint for implementation is that as the built-in cross-entropy function in Pytorch has already embedded the softmax function, it will automatically bind your output to [0, 1]. So while calculating the running accuracy, I only need to find the index of the output with the maximum probability and validate whether the index matches the label. The results turned out to be feasible with around 98 percent for training accuracy and around 97 percent for testing accuracy. We can see that the test accuracy is slightly worse than the training accuracy as we are testing the model on unseen data during the inference time.

2. Problem 2

In problem 2, the question asked us to implement the LeNet model and train it on the MNIST dataset. After successfully implementing the code, I additionally tried several sets of hyper-parameters. To find out the best combination, I store all the values in a python list and iterate all the variables through the training. In this section, several optimization methods, batch size, and learning rate are treated as variables. Empirically, instead of training over 10 epochs, I decided to increase the number to 20. Looking at the results, 0.005 seemed to be the optimal learning rate among almost optimization methods considering all circumstances. In addition, a smaller batch size generally performs better than a larger batch size. And I will dive into detail later.

3. Problem 2 Related Work

During the experiment, there is a trend showing insufficient learning rates are not able to lead the model to an optimal solution. Table 1, 2, and 3 show when the learning rate is equal to 5e-7, the performance are all undesired regardless of which batch size I choose. Furthermore, from table 1 and 2, they indicate that there is a positive correlation between batch size and learning rate. We can see that if we increase the batch size, it is better to have relatively high learning rates to obtain a better performance. One possible interpretation is that when batch size increase, we are more confident about the general gradient descent direction. This enables increasing the learning rate and helps us search for more generalized minima rather than sharp minima that can not represent the overall data well.

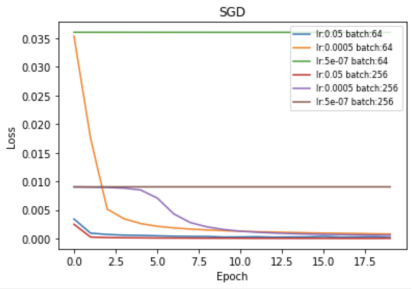

Figure 1. SGD + Momentum

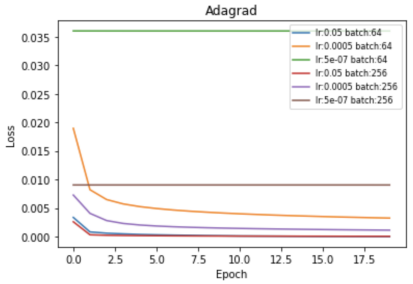

Figure 2. AdaGrad

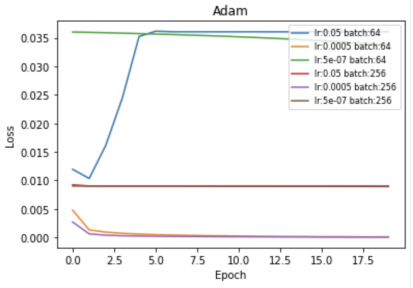

Figure 3. Adam

3.1. Stochastic Gradient Descent

Theoretically, SGD has problems when locating at the ravine. We can see a ravine in a 2D dimension as a valley where its x direction is steep while y direction is gentle. SGD will struggle along the steep slope and form a zigzagging trajectory, hence incurring slow training processes. To solve this problem, I adapt the SGD + momentum method to alleviate the phenomenon. From the physics point of view, the momentum value gives the friction force to prevent exceeding velocity. Generally, selecting the momentum equal to 0.9 is considered a feasible value. I also stuck with this value throughout the experiments.

3.2. AdaGrad and Adam

Both AdaGrad and Adam are adaptive optimization methods which imply that they may generate different step sizes depending on the situation. This feature helps the progress accelerate in the flat area where the gradient is relatively small. We may see an obvious difference between adaptive optimization methods and SGD when the learning rate and batch size are set to 0.0005 and 256 respectively. While SGD was struggling to converge the loss even after 5 epochs, both Adam and AdaGrad had already settled. This suggests that an 0.0005 learning rate is inadequate for such a big batch size. Yet, the adaptive optimization methods demonstrated that they were able to overcome the initial situation and reduce the loss effectively. However, the theory also suggested the feature of adaptive methods may also invoke insufficient driving force to reach minima over time. Fortunately, I did not see this happen during the training. Finally, one thing worth noticing about Adam is, despite the other two types of optimizers, that an 0.05 learning rate is extremely high for the optimizer to learn the data well. From Figure 3, the loss curve diverges severely when the learning rate is 0.05 and with a batch size of 64. Loss divergence is a typical phenomenon when setting an excessive learning rate.

64 Batch Size	Train Acc (%)	Test Acc (%)
lr = 0.05	99.70	98.82
lr = 0.0005	98.47	98.49
lr = 0.0000005	9.74	9.82
256 Batch Size	Train Acc (%)	Test Acc (%)
lr = 0.05	99.75	99.03
lr = 0.0005	95.55	95.95
lr = 0.0000005	7.48	7.37

Table 1. SGD + Momentum

64 Batch Size	Train Acc (%)	Test Acc (%)
lr = 0.05	99.99	98.96
lr = 0.0005	93.92	94.30
lr = 0.0000005	10.22	10.26
256 Batch Size	Train Acc (%)	Test Acc (%)
lr = 0.05	99.75	99.00
lr = 0.0005	91.75	92.30
lr = 0.0000005	9.78	9.84

Table 2. AdaGrad

64 Batch Size	Train Acc (%)	Test Acc (%)
lr = 0.05	10.70	11.35
lr = 0.0005	99.735	**99.07**
lr = 0.0000005	62.35	63.51
256 Batch Size	Train Acc (%)	Test Acc (%)
lr = 0.05	10.72	10.28
lr = 0.0005	99.54	98.93
lr = 0.0000005	36.13	37.53

Table 3. Adam

4. Conclusion and Discussion

In practice, a bigger batch size may provide a better performance in practice. However. in the meanwhile, we may also need to examine if the learning rates are feasible to allow the model to learn. Throughout the experiment, it has also shown us the feature of each optimization method. And I suggest that Adam be the most robust optimizer when compares to SGD and AdaGrad. In the future, I may also want to try learning decay or increasing the batch size more. Learning decay helps the model to have a larger step size in the beginning, but prevents it from overshooting when the target is close. Moreover, some latest studies argued that increasing the batch size is equivalent to decaying the learning rate. Since a larger batch size could reduce the number of total updated parameters, hence abridging the training time.