Deep Learning

Experimenting High Performance Aspects of Deep Learning

May 22, 2019

Lavanya Mandadapu

Contents

1 Introduction

In recent years, deep learning has been used to solve many difficult problems that traditional machine learning algorithms have failed to solve. However, it is always important to correctly tune(select) hyper parameters to obtain good accuracies. This work focuses on analysing different optimizers and learning rates in classification of MNIST dataset [1]. Although deep learning techniques has achieved significant results, for a very large data sets, it takes more computational time. So, the effect of using Multiple GPUs to reduce the training time is also studied. The source code and the plots are available in repo.

2 Analysing different optimizers on obtaining a linear model for a set of points

In this section, the effect of using different optimizers and learning rates on obtaining a linear model that best fits a set of points was studied. Every optimizer takes learning rate: size of steps, that takes into the direction of local minima. A set of 6 optimizers are tested:

- **Gradient Descent**: It is simple and powerful optimization algorithm based on convex function. It modifies the parameters iteratively to minimize the given function (loss) to its local minima.
- **Adagrad (Adaptive Gradient)**: This algorithm maintains a per-parameter learning rate to improve performance, especially when there are sparse gradients.
- **Adadelta**: This is an extension of Agagrad. Instead of taking all past squared gradients to update learning rate, a fixed window size is used.
- **RMSProp (Root Mean Square Propogation)**: This also maintains per-parameter learning rate, which are adapted using the average of recent magnitude of gradients.
- **Adam**: This can be seen as the combination of Adagrad and RMSProp. Unlike RMSProp, Adam takes into account the variance to update learning rate. This algorithm calculates the exponential moving average of gradient and squared gradient.
- **FTRL (Follow The Regularized Leader)**: The implementation of FTRL in Tensor Flow uses batch gradients: updating the learning rate once per batch.

Gradient Descent algorithm is tested with learning rates of 0.01, 0.001, 0.0001 and 0.3. When the learning rate is 0.01 (Figure 1), a good linear model is obtained, with less loss. When the learning rate is 0.001, the steps to the local minima will be slower, resulting the loss curve to show (have) the higher loss at first and gradually decreasing reaching a clear slope. The linear model obtained in this case is not as good as previous one. When the learning rate is 0.0001, which is too small, the linear model obtained is similar to the learning rate being 0.01, but much slower. Generally, with low learning rate, the model will get stuck in the local minima. When the learning rate is large (0.03), optimal is never reached, the steps takes back and forth resulting the increase of the loss for every iteration. When Adam is tested with 0.01 learning rate and 1000 iterations, it obtained not so good. From the Figure 2, it is clearly seen that for initial iterations, it got stuck in the local optima and eventually got out after some iterations. For Adagrad and Adadelta (Figure 3, 4), when tested with 0.01 learning rate, the obtained model is very bad. However, in the case of Adam, Adagrad and Adadelta, there is hope of reaching the optimal by increasing the number of iterations, but in RMSProp, when tested with 0.01 learning rate, it clearly got stuck at the local minima (Figure 5). Increasing the number of iterations for RMSProp with 0.01 learning rate will not improve the model. Given that there is only one batch, Adagrad and FTRL has behaved identically (Figure 3, 6), giving same curves.

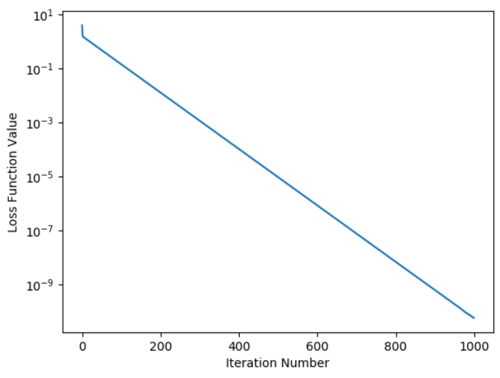

Figure 1: GD with 0.01 learning rate

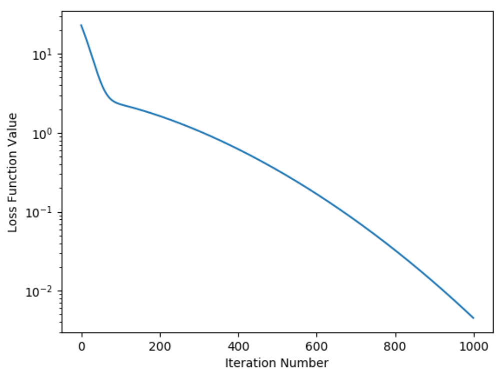

Figure 2: Adam with 0.01 learning rate

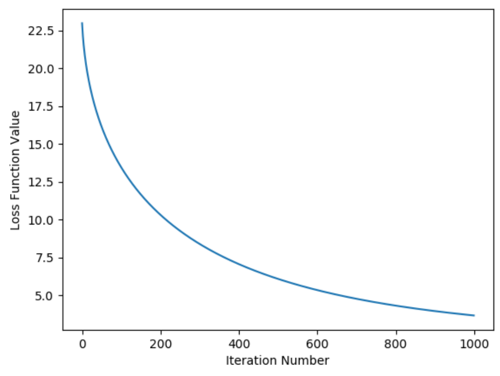

Figure 3: Adagrad with 0.01 learning rate

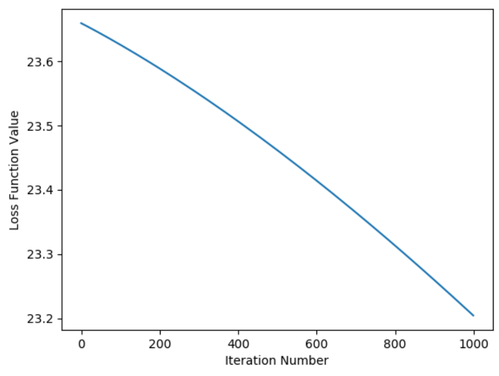

Figure 4: Adadelta with 0.01 learning rate

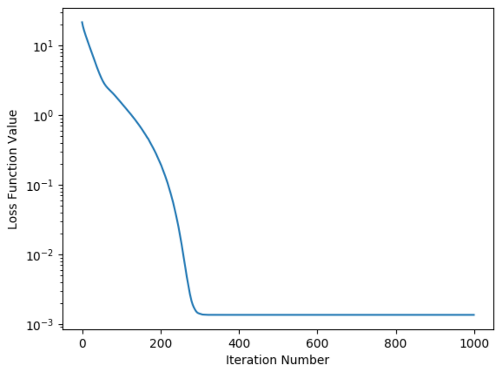

Figure 5: RMSProp with 0.01 learning rate

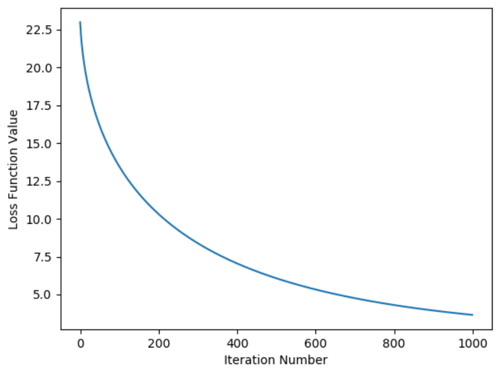

Figure 6: FTRL with 0.01 learning rate

3 Classification of MNIST data set using Single Layer Network

In this section, a single layer network is used to solve the MNIST classification task. Different optimizers and learning rates are tested to see their impact on this classification task. Table 1 depicts the test accuracy obtained with different optimizers and learning rates. A simple single layer network with Gradient descent as optimizer and learning rate being 0.5 has produced a test accuracy of 90.79%. When the learning rate for the Gradient descent is further increased (0.9), there is no significant improvement in the accuracy, but when the learning rate is decreased, the test accuracy is decreased (stuck at local minima). Accuracy of 91.66%is obtained using Adam and 0.01 learning rate. When the learning rate is further decreased, the model didn't improve, and when the learning rate is increased, it is not reaching the optimal at all, resulting in a very low accuracy. Adagrad with 0.5 has given 91.02% accuracy. Increasing or decreasing the learning rate in this case has reduced the accuracy, leaving out 0.5 as best learning rate for Adagrad. For Adadelta, initially 0.5 learning rate is used, which produced an accuracy of 85.35%. On decreasing the learning rate, the accuracy also decreased. But, on increasing the learning rate, the accuracy increased. So, higher values for learning rate up to 9 are tested. For learning rate being 9, Adadelta produced an accuracy of 91.17%. Further increase in learning rate didn't improve the model. The effect of learning rate change on test accuracy in RMSProp algorithm is similar to Adam whereas in FTRL algorithm is similar to Adagrad as mentioned in previous section. Figures 7, 8, 9 and 10 depicts the loss plots of optimizers (Adam, GD, RMSProp and Adagrad) and learning rates obtained for single layer network. Other loss plots are available in the repo.

Table 1: MNIST classification test accuracy for different combinations of optimizers and learning rates.

Optimizer	Learning Rate	Test Accuracy
Adagrad	0.5	0.9102
Adagrad	0.01	0.8779
Adagrad	0.7	0.908
Adam	0.01	**0.9166**
Adam	0.1	0.098
Gradient Descent	0.5	0.9079
Gradient Descent	0.01	0.85
Gradient Descent	0.9	0.9095
Adadelta	0.5	0.8535
Adadelta	0.01	0.6922
Adadelta	0.9	0.8741
Adadelta	3	0.9023
Adadelta	9	0.9117
RMSProp	0.5	0.098
RMSProp	0.01	0.9123
RMSProp	0.001	0.9053
FTRL	0.5	0.9102
FTRL	0.01	0.8779
FTRL	0.7	0.908

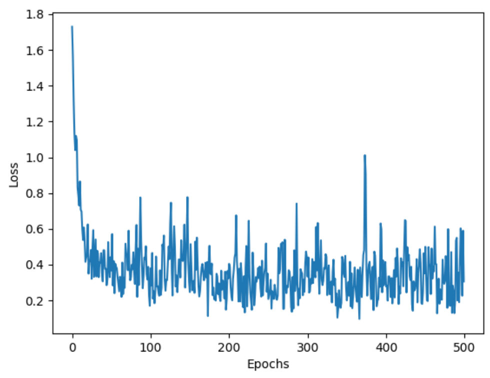

Figure 7: Adam with 0.01 learning rate

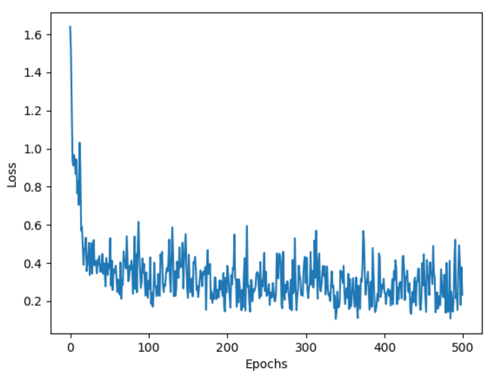

Figure 8: GD with 0.5 learning rate

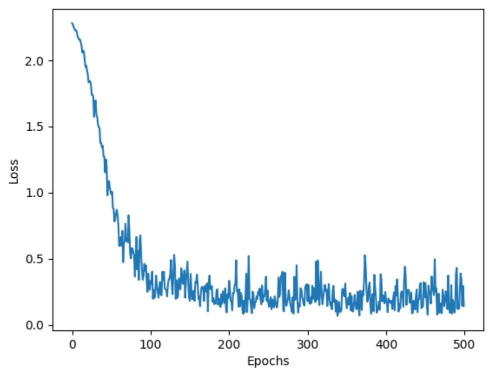

Figure 9: RMSProp with 0.01 learning rate

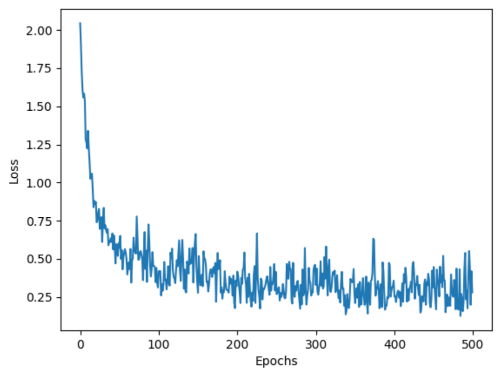

Figure 10: Adagrad with 9 learning rate

4 Classification of MNIST data set using Multiple Layer Network

In this section, Multiple layer network is used to solve the MNIST classification problem. Multi layer network used in this work consists of two convolution layers and fully connected layer with a dropout of 0.5. Optimizer Adam with 0.0001 learning rate is used. A batch size of 100 has produced an accuracy of 95.4%. Batch size determines how much we feed the model to learn at a time out of the whole data. Generally, lower batch sizes allows the model to learn better at the cost of more computational time. When the batch size is reduced to 50, an accuracy of 96.35% is obtained. Further decreasing to 20 and 10 didn't give any significant results. But, when the batch size is reduced to 4, a test accuracy of **98.27%** is obtained.

4.1 Reducing Training Time using Multiple GPU

MNIST is a small data set with 50000 samples. Eventhough an accuracy of 98.27% is reached, the computation time is quite high (nearly a minute). Using parallel processing can reduce this time. In this section, Multiple GPU's are used to reduce the training time needed for the MNIST dataset. Figures 11,12 and 13 illustrate the training time plots of different number of GPU's (1,2 and 4) with different batch sizes. From the plot it is clear that with less batch size, using multiple GPU is no benefitial [2]. It is important to note that multiple GPU's only help if the requested computation has already saturated a single GPU (or the GPU's at use). In the context of deep learning, demand for the computation is increased when there is a larger network, or when the batch size is larger. The model used for this work is not so complex, to demand high computation. Likewise, with a smaller batch size, there is no demand for the high computation, resulting in the increase of training time. The training time required for multiple GPU's in the case of batch size being 4 illustrate this fact. For a batch size of 4, using 1 GPU took 59 seconds where using 2 GPU's took 65 sec and using 4 GPU's took 71 seconds (clearly more time than 1). The main reason is that, running a smaller batches on multiple GPU's just make then idle for most of the time. For larger batches (48), multiple GPU's have shown clear improvement in the training time, using 1 GPU took 8.44 seconds, using 2 GPU's took 7.22 seconds and using 4 GPU's 7.02 seconds.

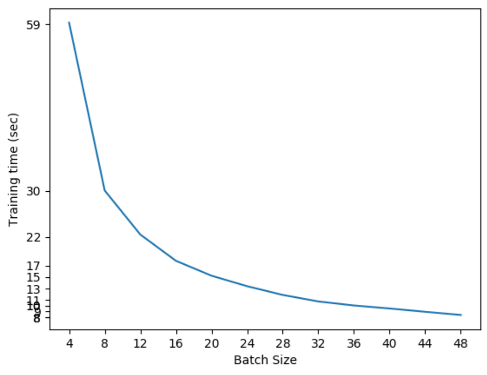

Figure 11: Training time plot of using 1 GPU

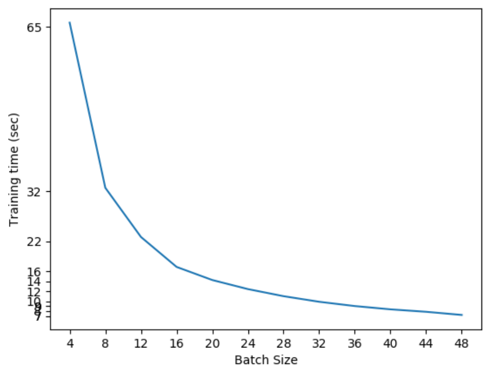

Figure 12: Training time plot of using 2 GPU's

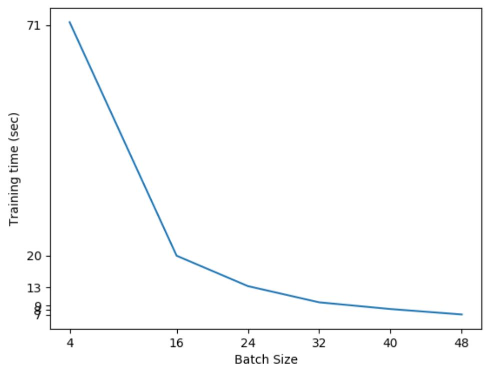

Figure 13: Training time plot of using 4 GPU's

5 Conclusions

Choosing right hyper-parameters is an important and necessary task to achieve better results. Among many hyper parameters, the effect of learning rate and optimizers is studied here. Using multiple GPU's can significantly decrease the training time only when the demand for computation is large. In order to utilize the use-fullness of parallel processing, it is required to study the type, complexity and the computation power needed for the problem at hand.

References

[1] Kaggle:. https://www.kaggle.com/ngbolin/mnist-dataset-digit-recognizer.

[2] Chuan Li:. https://medium.com/@c$_6$1011/$why - multi - gpu - training - is - not - faster - f439fe6dd6ec$.