Rutgers, the State University Of New Jersey

Optimizer Performance Analysis on LeNet5 CNN Architecture

Final Project

ECE-579: Intro to Deep Learning

Pranav Shivkumar

RUID: 194007405

NetID: ps1029

Abstract

Artificial intelligence and data science has taken today's world by storm. It finds use in many diverse fields, be it in technology, medicine, agriculture, autonomous vehicles and many more. While artificial intelligence on its own is hugely popular, deep learning, a subset of artificial intelligence, is the most popular, with the use of neural networks in many fields. While research is extensively going on, convolutional neural networks, or CNNs, find many applications in image classification or identification. The most popular dataset used to train CNNs is the MNIST dataset, that is considered a standard in deep learning. This project aims to determine the effect of optimizers on the LeNet5 CNN model trained with the MNIST dataset as well as analyze the performance of these optimizers on the model, considering the test accuracy, test loss and training time.

Introduction

The aim of this project is to analyze the performance of various optimizers on the LeNet5 convolutional neural network (CNN) architecture. The dataset used to train and test the model was the MNIST dataset and the model was constructed using the PyTorch sequential model in Python.

Data Description

The MNIST[1], or Modified National Institute of Standards and Technology, database, is a large database of handwritten digits that is used to train various image processing systems as well as training and testing machine learning models. The dataset consists of 60,000 training images and 10,000 test images and was created by re-mixing the samples from NIST's original datasets. Half of the training set images and half of the test set images were taken from NIST's training dataset, while the other half of the both sets were taken from NIST's test dataset.

System Specifications

The system specifications that were used for the project are listed below:

Device: Dell XPS 15

Processor: Intel (R) Core (TM) i7-9750H CPU @ 2.60GHz, 2592 Mhz, 6 Core(s)

Memory: 16GB RAM

GPU: NVIDIA GeForce GTX 1650

Operating System: Windows 10

LeNet5 Architecture

The LeNet5 architecture is a neural network architecture proposed by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner for the purpose of handwritten and machine-printed character recognition. It basically consists of two sets of convolutional and average pooling layers followed by a flattening convolutional layer, two fully connected layers and finally a SoftMax classifier. The input to the LeNet5 CNN architecture is a 32x32 grayscale image.

a. First Layer: The first layer of the LeNet5 CNN consists of 6 filters having a size of 5x5 and a stride of 1. On passing the input image through this layer, the input goes from 32x32x1 to a size of 28x28x6.

b. Second Layer: This layer applies maximum pooling to the output of the first layer with a filter size of 2x2 and a stride of two. The resulting image is reduced to a size of 14x14x6.

c. Third Layer: This layer forms the second convolutional layer with 16 filters, each having a filter size of 5x5 and a stride of 1.

d. Fourth Layer: This layer is another max pooling layer with a filter size of 2x2 and a stride of 2. This is like the second layer, with the difference that it has 16 filters, so the output is reduced to 5x5x16.

e. Fifth Layer: This is a fully connected convolutional layer with 120 filters, each of size 1x1.

f. Sixth Layer: This layer is a fully connected layer with 84 units.

g. Output: The final layer consists of a fully connected SoftMax output layer with 10 possible values, which correspond to the 10 digits (0 to 9).

   For the project, the CNN was constructed using 2 stacks, the first of which consists of the convolutional layers (the convolutional layers and max pooling layers) and the second stack consists of the fully connected layers.

Optimizers

The optimizers used for training the model are listed below:

1. Stochastic Gradient Descent (SGD)
2. Stochastic Gradient Descent with Momentum (SGDM)
3. Adaptive Gradient Descent (Adagrad)
4. RMS Propagation
5. Adam Optimizer

Stochastic Gradient Descent (SGD)

The stochastic gradient optimizer [2], or SGD, implements the stochastic gradient descent algorithm on the model parameters. Basically, it performs a parameter update for each training example and label. The below line declares the SGD optimizer for the model:

```
optimizer = optim.SGD(model.parameters(), lr=lr)
```

Stochastic Gradient Descent with Momentum (SGDM)

This optimizer implements the stochastic gradient descent algorithm with momentum, which is provided as a parameter to the optimizer. Since SGD essentially gets stuck in areas where the surface curves more deeply in one dimension than another, leading to oscillations that hesitantly approach the local optimum, adding momentum helps accelerate SGD in the relevant direction. The below line declares the SGD optimizer with momentum:

```
optimizer =  optim.SGD(model.parameters(), lr=lr, momentum=0.9)
```

Adaptive Gradient Descent (AdaGrad)

The adaptive gradient descent [2], or AdaGrad, basically adapts the learning rate to the model parameters by penalizing the learning rate for parameters that need to be updated frequently and providing more learning rate to parameters that are not updated frequently. The below line demonstrates the AdaGrad optimizer in PyTorch:

```
optimizer = optim.Adagrad(model.parameters(), lr=lr)
```

RMSProp Optimizer (RMSProp)

The RMSProp [2] learning rate method basically utilizes the idea behind AdaDelta, an extension of Adagrad. Instead of accumulating all past squared gradients, this restricts the window of accumulated past gradients to some fixed size. The below line demonstrates the RMSProp optimizer in PyTorch:

```
optimizer = optim.RMSprop(model.parameters(), lr=lr)
```

Adam Optimizer

The Adam optimizer[2][3], also known as the adaptive moment estimation, is one of the most popular optimizers. It combines the best properties of Adagrad and the RMSProp optimizers, which is why it performs better for most of the problems. The below line demonstrates the Adam optimizer in PyTorch:

```
optimizer = optim.Adam(model.parameters(), lr=lr)
```

Experimental Procedure

The first step in the procedure was to construct the LeNet5 CNN model using PyTorch. This was done using the Sequential model in PyTorch, by defining the necessary layers within an ordered dictionary within this model. Separate scripts were written, each one corresponding to a separate optimizer, and the model was trained with the MNIST dataset.

Once this was set up, the model was trained with the MNIST training dataset in batches of 128 samples each for 5 epochs. The test set accuracy and loss were printed for each epoch and at the end, the total time taken for training and testing the dataset was printed in the terminal. To analyze the performance of each optimizer, the model was trained for each learning rate ranging from 0.01 to 0.05, incremented by steps of 0.01. After training the model, the model is tested on the MNIST test dataset with a test batch size of 100 samples.

Additionally, I added a menu in the main script for a user to enter any optimizer that the user wishes to train the model with, so that the results for that specific optimizer can be viewed at a time. This menu is shown below:

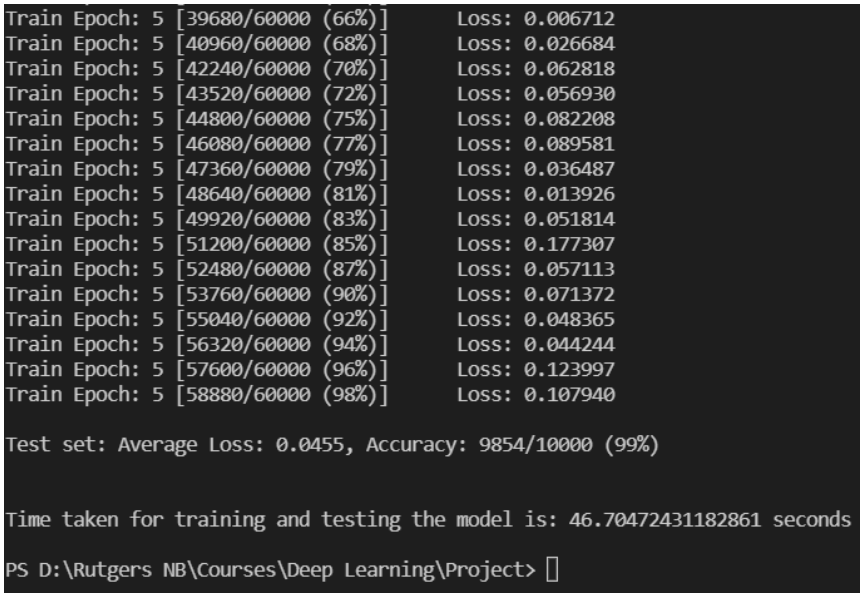
```
PS D:\Rutgers NB\Courses\Deep Learning\Project> C:\Users\prana\AppData\Local\Programs\Python\Python38\python.exe main.py
Preparing data...
Preparing data...
Preparing data...
Preparing data...
Preparing data...
1. SGD
2. SGD w/ Momentum (SGDM)
3. Adagrad
4. Adam
5. RMSProp (RMS)
6. All
Enter the model to train:
```

As shown above, if the user enters SGD, SGDM, Adagrad, Adam or RMS as the inputs, the program trains the model with that corresponding optimizer and displays the tabular results. However, if the user enters ‘All’ (or any other input) as the input, the program trains the model on all the optimizers and displays graphs plotting the test accuracy, test loss and training time versus the learning rate.

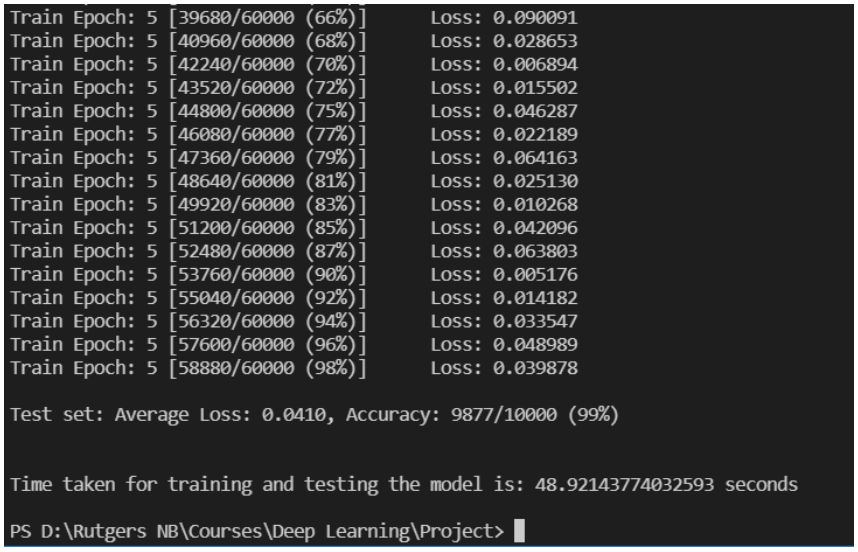
Results

The results obtained on training the model with all 5 optimizers are shown below:

a. Stochastic Gradient Descent (SGD)



b. Stochastic Gradient Descent with Momentum (SGDM)



c. Adaptive Gradient Descent (Adagrad)

```
Train Epoch: 5 [39680/60000 (66%)]      Loss: 0.030584
Train Epoch: 5 [40960/60000 (68%)]      Loss: 0.037154
Train Epoch: 5 [42240/60000 (70%)]      Loss: 0.049011
Train Epoch: 5 [43520/60000 (72%)]      Loss: 0.011240
Train Epoch: 5 [44800/60000 (75%)]      Loss: 0.034624
Train Epoch: 5 [46080/60000 (77%)]      Loss: 0.031656
Train Epoch: 5 [47360/60000 (79%)]      Loss: 0.032288
Train Epoch: 5 [48640/60000 (81%)]      Loss: 0.026142
Train Epoch: 5 [49920/60000 (83%)]      Loss: 0.088645
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.076503
Train Epoch: 5 [52480/60000 (87%)]      Loss: 0.035163
Train Epoch: 5 [53760/60000 (90%)]      Loss: 0.010486
Train Epoch: 5 [55040/60000 (92%)]      Loss: 0.029343
Train Epoch: 5 [56320/60000 (94%)]      Loss: 0.041269
Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.066359
Train Epoch: 5 [58880/60000 (98%)]      Loss: 0.030890

Test set: Average Loss: 0.0481, Accuracy: 9849/10000 (98%)


Time taken for training and testing the model is: 48.12712502479553 seconds

PS D:\Rutgers NB\Courses\Deep Learning\Project>
```

d. Adam Optimizer

```
Train Epoch: 5 [39680/60000 (66%)]      Loss: 2.331651
Train Epoch: 5 [40960/60000 (68%)]      Loss: 2.290055
Train Epoch: 5 [42240/60000 (70%)]      Loss: 2.304076
Train Epoch: 5 [43520/60000 (72%)]      Loss: 2.318943
Train Epoch: 5 [44800/60000 (75%)]      Loss: 2.295058
Train Epoch: 5 [46080/60000 (77%)]      Loss: 2.297462
Train Epoch: 5 [47360/60000 (79%)]      Loss: 2.303374
Train Epoch: 5 [48640/60000 (81%)]      Loss: 2.310011
Train Epoch: 5 [49920/60000 (83%)]      Loss: 2.301451
Train Epoch: 5 [51200/60000 (85%)]      Loss: 2.308040
Train Epoch: 5 [52480/60000 (87%)]      Loss: 2.306528
Train Epoch: 5 [53760/60000 (90%)]      Loss: 2.299439
Train Epoch: 5 [55040/60000 (92%)]      Loss: 2.323711
Train Epoch: 5 [56320/60000 (94%)]      Loss: 2.313268
Train Epoch: 5 [57600/60000 (96%)]      Loss: 2.304372
Train Epoch: 5 [58880/60000 (98%)]      Loss: 2.302754

Test set: Average Loss: 2.3039, Accuracy: 1135/10000 (11%)


Time taken for training and testing the model is: 48.708762407302856 seconds

PS D:\Rutgers NB\Courses\Deep Learning\Project>
```

e. RMSProp Optimizer

```
Train Epoch: 5 [39680/60000 (66%)]      Loss: 2.298211
Train Epoch: 5 [40960/60000 (68%)]      Loss: 2.289312
Train Epoch: 5 [42240/60000 (70%)]      Loss: 2.298586
Train Epoch: 5 [43520/60000 (72%)]      Loss: 2.313435
Train Epoch: 5 [44800/60000 (75%)]      Loss: 2.308671
Train Epoch: 5 [46080/60000 (77%)]      Loss: 2.298384
Train Epoch: 5 [47360/60000 (79%)]      Loss: 2.316932
Train Epoch: 5 [48640/60000 (81%)]      Loss: 2.305611
Train Epoch: 5 [49920/60000 (83%)]      Loss: 2.292687
Train Epoch: 5 [51200/60000 (85%)]      Loss: 2.316898
Train Epoch: 5 [52480/60000 (87%)]      Loss: 2.321708
Train Epoch: 5 [53760/60000 (90%)]      Loss: 2.299334
Train Epoch: 5 [55040/60000 (92%)]      Loss: 2.305658
Train Epoch: 5 [56320/60000 (94%)]      Loss: 2.304228
Train Epoch: 5 [57600/60000 (96%)]      Loss: 2.303854
Train Epoch: 5 [58880/60000 (98%)]      Loss: 2.303864

Test set: Average Loss: 2.3072, Accuracy: 1010/10000 (10%)


Time taken for training and testing the model is: 47.58112096786499 seconds

PS D:\Rutgers NB\Courses\Deep Learning\Project>
```

These graphs and tables are shown below:

a. For the stochastic gradient descent (SGD) optimizer:

Learning Rate	Test Accuracy (%)	Test Loss	Training Time (s)
0.01	96.07	0.13	68.58
0.02	97.25	0.08	64.47
0.03	98.02	0.06	65.82
0.04	98.02	0.06	109.32
0.05	98.63	0.04	131.83

b. For the stochastic gradient descent with momentum (SGDM) optimizer:

Learning Rate	Test Accuracy (%)	Test Loss	Training Time (s)
0.01	98.9	0.04	81.93
0.02	98.59	0.04	65.23
0.03	99.0	0.03	63.47
0.04	98.74	0.04	63.67
0.05	98.53	0.05	64.39

c. For the adaptive gradient descent (Adagrad) optimizer:

Learning Rate	Test Accuracy (%)	Test Loss	Training Time (s)
0.01	98.73	0.04	66.15
0.02	98.71	0.04	79.96
0.03	98.72	0.04	110.16
0.04	98.48	0.04	132.78
0.05	98.18	0.05	140.39

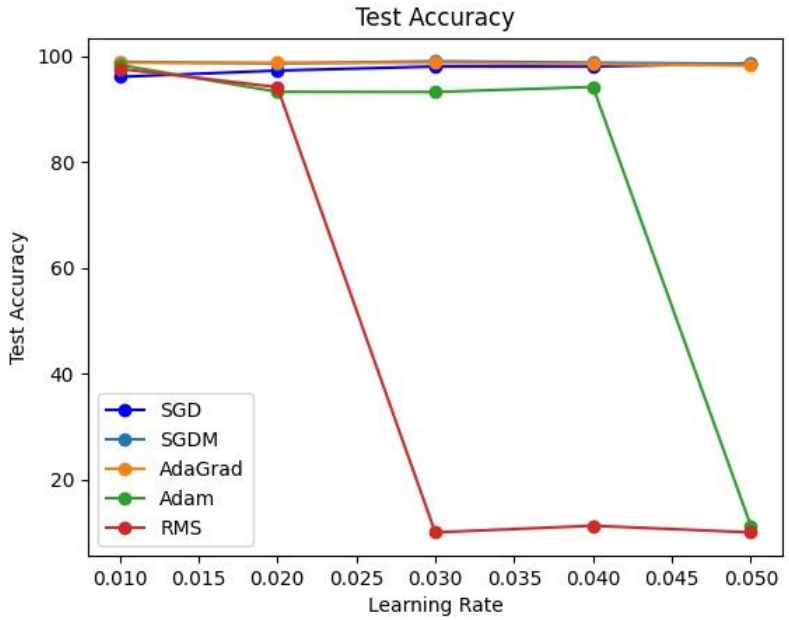
d. For the Adam optimizer:

Learning Rate	Test Accuracy (%)	Test Loss	Training Time (s)
0.01	98.33	0.08	117.36
0.02	93.24	0.22	65.33
0.03	93.21	0.28	121.32
0.04	94.17	0.27	134.22
0.05	11.35	2.3	92.34

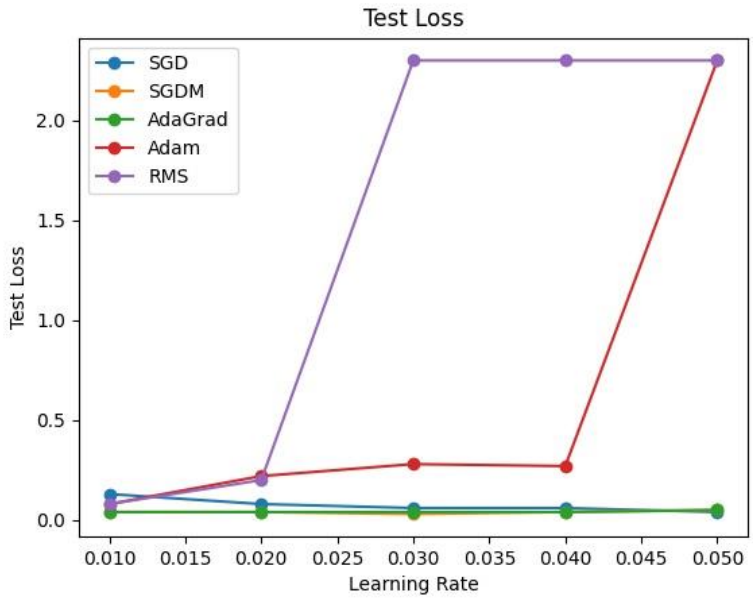
e. For the RMSprop optimizer:

Learning Rate	Test Accuracy (%)	Test Loss	Training Time (s)
0.01	97.58	0.08	67.47
0.02	94.13	0.2	110.81
0.03	10.1	2.3	134.36
0.04	11.35	2.3	142.56
0.05	10.09	2.3	99.18

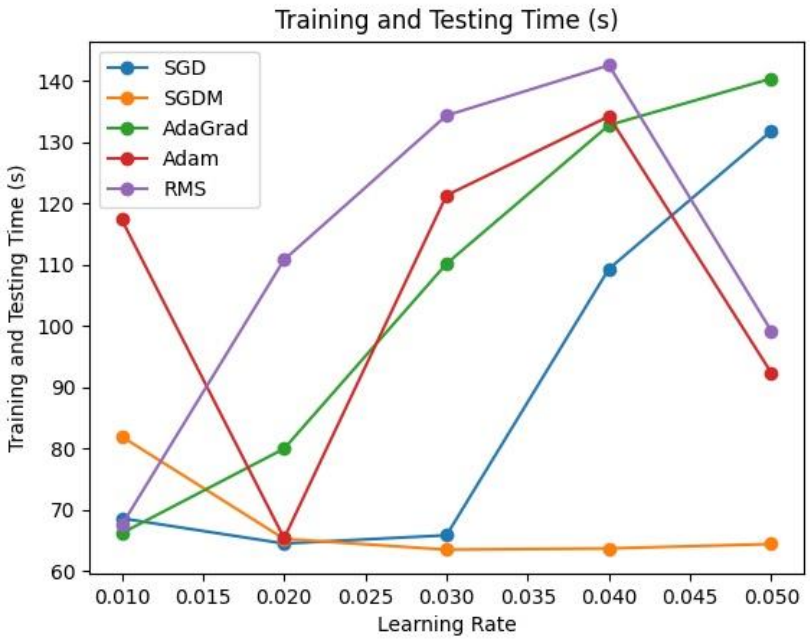
The variation of the test accuracy was plotted against the learning rate values to observe the trend. This plot is shown below:



The variation of the test loss with the learning rate is shown below:



The variation of the training time versus the learning rate is shown below:



Observations & Conclusion

In order to observe the effect of momentum on stochastic gradient descent and the role it plays, I have run the same code for SGD – one includes the momentum parameter to add momentum to the optimizer and the other doesn't.

As seen in the results above, for all the learning rates, stochastic gradient descent with momentum (SGDM) offers the highest test accuracy for all the learning rates tested for. While stochastic gradient descent on its own offers a relatively higher test accuracy, the addition of momentum to the optimizer starts the accuracy off at 98%, as compared to 97%, and goes until 99% accuracy.

As for the adaptive gradient descent optimizer (Adagrad), the test accuracy appears to be nearly constant over all the learning rates. The Adam optimizer appears to perform well for lower learning rates, but as the learning rate increases, the test accuracy falls, plummeting to 11% accuracy for a learning rate of 0.05.

The RMSprop optimizer falls in a similar category as the Adam optimizer, however, in this case, the test accuracy falls to 10% at a learning rate of 0.03 and reaches a low accuracy at 9.74% for a learning rate of 0.04.

The above observations are further corroborated with the graphs plotted for all optimizers. The test accuracy for stochastic gradient descent, with and without momentum, as well as adaptive gradient descent, is nearly constant and provide near equal test accuracies, while the test accuracies obtained with the RMSprop and Adam optimizers decline with increase in learning rate. As for the test loss with each epoch the model is trained, the loss from the SGD, SGDM and Adagrad optimizers decrease slightly; however in the case of RMSProp and Adam optimizers, the loss increases to nearly 2.

Finally, considering the training time, the SGD and Adagrad optimizers experience a steady increase in training time with increase in learning rate, but for the Adam and RMS optimizers, the training time increases and then decreases towards a learning rate of 0.05. However, the time taken to train the model with the SGDM optimizer decreases with increasing learning rate. This shows that the most efficient optimizer is the stochastic gradient descent optimizer with momentum (SGDM).

References

1. https://en.wikipedia.org/wiki/MNIST_database
2. https://analyticsindiamag.com/ultimate-guide-to-pytorch-optimizers/
3. https://ruder.io/optimizing-gradient-descent/index.html#rmsprop