RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

INTRODUCTION TO DEEP LEARNING PROJECT REPORT

Performance analysis of LeNet-5 architecture

Shounak Rangwala
May 12, 2020.

Table of Contents

Abstract:
In this project, I have studied the efficiency and the accuracy of the LeNet-5 CNN architecture based on different metrics. I will be running the MNIST dataset through my model for finding the performance. This report has 2 parts. For the first part, I have analysed the performance of the LeNet-5 model coupled with different optimizers. These optimizers include the 3 most popular optimizers which are RMSProp, AdaGrad and SGD (with momentum). Running simulations based on varying learning rates I found the best performance for the SGD optimizer with a learning rate = 0.04. The accuracy of the model was 99%. For the second part I included a dropout layer in the fully connected layer stack and a batch normalization layer in the convolutional network layer and analysed performance for different combination of these layers. I found that the FC layers with dropout without BN layer in CN stack has best performance with least test loss = 0.0323. The implementation has been done using the PyTorch Sequential module.

Introduction:
Before we discuss the CNN architecture I used in the project, we should examine the dataset which I was working on. The MNIST (Modified National Institute of Standards and Technology) is a large dataset of handwritten numbers ranging from 0-9. This was collected from the handwriting of students in American high schools. This has 60000 training images and 10000 test images. The dimension of each image is 32x32 pixels. We have to understand the size of the input data so that our model can effectively run matrix transformations on the input and give an accurate prediction.
The Lenet CNN architecture was proposed by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner in the 1990's. This was specifically designed to take in the input of handwritten digits and run classification on them. This is a simple convolution network that is working on the feed-forward mechanism where each neuron interacts with the results of the. surrounding neurons (neighbors) and is useful for large-scale image processing applications. We will discuss the architecture of this later in the report. Generally, the layers are of 2 stacks: the convolutional layers and the fully connected layers.
The optimizers play a very importantly role in deep learning specifically in the design of neural networks. We need these optimisers to work the gradient descent function properly so that we can converge on a particular set of weights which can satisfactorily help us classify with a good accuracy. Some of the most prominent are the SGD (stochastic gradient descent) with momentum which does not get stuck to local minimas , AdaGrad which adaptively sets a different gradient descent on sensing how slowly or quickly the values are changing and lastly RMSProp which is an unpublished algorithm for gradient descent optimisation. I jhave run simulation using each of these optimizers in my model for the same datasets and have compared the performance of the Lenet architecture.
Batch Normalization is a method that increases the smoothness and nthe stability of the CNN model. The input layer after batch normalization gets re-scaled and re-centered so that the matrix transformations on the input happen easily and the output shows no anomaly.

Machine Description:
The simulations (training and testing were done on this laptop):

Machine used: MacBook Air (Retina, 13-inch, 2019)
Processor - 1.6 Ghz intel Core i5
Memory - 8 GB 2133 MHz LPDDR3
Graphics - Intel UHD Graphics 617 1563 MB
Operating System - MacOS 10

Model Description:
The Lenet-5 model has 2 layer stacks:

1) The Convolutional layers:
    a. C1 is the first convolution layer with 6 kernels of size 5x5. It is followed by a ReLU activation function.
    b. S2 is the following pooling or subsampling layer which has a kernel size of 2x2 and stride of 2. This selects the samples that will be passed on as the input of the next convolution layer.
    c. C3 is the second convolution layer with 16 such kernels of size 5x5. It is also followed by a relu activation layer.
    d. Lastly the S4 layer selects the sub-samples and send the output ahead to the fully connected layer.

2) The fully connected layers:
    a. Linear transformation function which helps decrease the dimensions of the matrix.
    b. Alternated with ReLU activation functions.

Project Part A

Objective: Evaluate the performance of different types of optimizer on a LeNet-5 network using MNIST data. At least you need to evaluate SGD, AdaGrad, RMSprop.

Procedure: The Lenet model usually uses the SGD optimizer. Pytorch in its nn.optim module provides for different types of optimizers.
The 3 optmizers can be included with their individual parameters like follows:

1) SGD (with momentum) :

```
optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9, weight_decay=5e-4)
```

Model.parameters() = dictionary of the model layers
Momentum = value of the momentum that will be used in finding the gradient descent
Lr= learning rate

2) AdaGrad:

```
optimizer = optim.Adagrad(model.parameters(), lr=lr, lr_decay=0, weight_decay=5e-4, initial_accumulator_value=0, eps=1e-10)
```

Lr_decay = learning rate decay
Eps = term added to the denominator to improve numerical stability

3) RMSProp:

```
optimizer = optim.RMSprop(model.parameters(), lr=lr, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False)
```

Alpha = smoothing constant
Centered = if True, it computes the centeres RMSProp. Where the gradient is normalized by an estimation of its variance.

To compare their performances, I ran the entire model with each optimizers for 5 epochs.
I varied the learning rates from 0.01 – 0.05. The test loss, accuracy and the training and testing time parameters were recorded.

Results:

1) Test Loss

learning rates	RMSProp	AdaGrad	SGD
0.01	0.0777	0.0571	0.0510
0.02	0.1581	0.0410	0.0410
0.03	0.2075	0.0396	0.0367
0.04	2.3035	0.0426	0.0338
0.05	0.4765	0.0540	0.0383

2) Time (testing + training) in seconds

learning rates	RMSProp	AdaGrad	SGD
0.01	124.4358720779419	136.5499711036682	134.809672832489
0.02	131.91999292373657	176.07710576057434	128.08544898033142
0.03	125.31782793998718	225.0645980834961	129.88648295402527
0.04	144.53284883499146	290.6422200202942	124.5013279914856
0.05	125.99141097068787	241.92013788223267	124.566565990448

3) Testing accuracy

learning rates	RMSProp	AdaGrad	SGD
0.01	9774/10000	9815/10000	9840/10000
0.02	9598/10000	9861/10000	9861/10000
0.03	9471/10000	9870/10000	9876/10000
0.04	1010/10000	9871/10000	9884/10000
0.05	8728/10000	9831/10000	9882/10000

Note: RMSprop for lr = 0.005

Average loss: 0.0561, Accuracy: 9853/10000 (99%)
Training and Testing total execution time is: 130.7267463207245 seconds

Conclusions:

SGD – By far the best performance seen on the dataset for all the learning ratres as compared to the other optimizers. As you can see from the trend, the loss and the time keep decreasing and the accuracy increases as we increase the learning rate from 0.01 to 0.04. At 0.04, we can assume that for the given set of parameters, this is the best performance the SGD optimizer can give on this model. After that you can see the at learning rate 0.05 the performance decreases and based on that we can assume that in subsequent learning rates the trend will be downwards.

AdaGrad – AdaGrad performed the second best in my model. I believe that its performance can be further tuned to be better than the SGD optimizer because theoretically, we can achieve that improvement. However for the parameters set, the performance of Adagrad optimizer also reaches optimal level at around leraning rate = 0.03. The time taken is more than that of SGD however both the test loss and the accuracy lie very close to SGD.

RMSProp – I started the learning rate at 0.01 and from there till the end the performance went down. The test loss and the accuracy have been consistently lower than the AdaGrad and the SGD optimizers. At the learning rate of 0.04 , I hit a big anomaly of having accuracy of only 10 %. TO be sure of this result, I ran the model again for this value and got the same result. My only inference is that I have overshot the minimum with this learning rate because in epochs 2 and 3 the loss was below 1 and then suddenly shot to 2. If I would have started at a learning rate lower than 0.01 for RMSProp maybe the accuracy would be better. To prove my hypothesis, I set the learning rate at 0.005 and the results as you can see are comparable to the better results of SGD and AdaGrad.

Project Part B

Objective: Evaluate the performance of dropout on fully-connected layers and batch normalization on convolutional layers. The model is LeNet-5 on MNIST dataset. Among four options 1) FC with dropout, CONV with BN; 2) FC with dropout, CONV without BN; 3) FC without dropout, CONV with BN; 4) FC without dropout, CONV without dropout, identify which one has the best performance?

Procedure:
As stated in the objective, I added a dropout and Batch normalization layer in the required stacks and ran the simulations.
The Pytorch nn module provided for the Dropout and BN methods.
The structure looked like this:
Dropout:

```
torch.nn.Dropout2d(p=0.5, inplace=False)
```

p = probability of an element of the matrix to be zero-ed out.

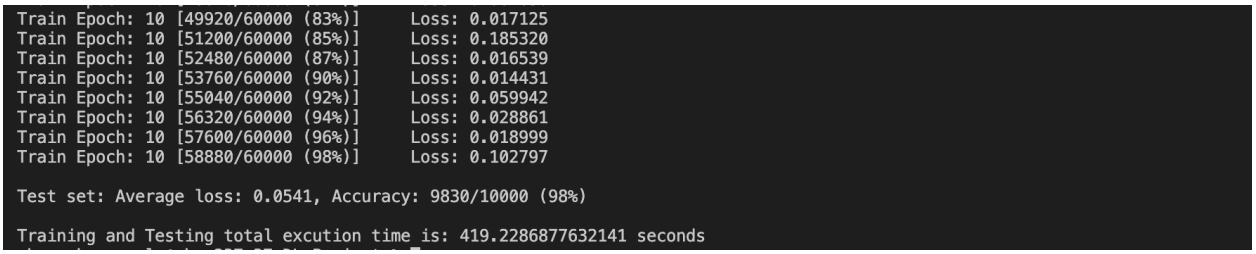
Batch Normalization:

```
torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

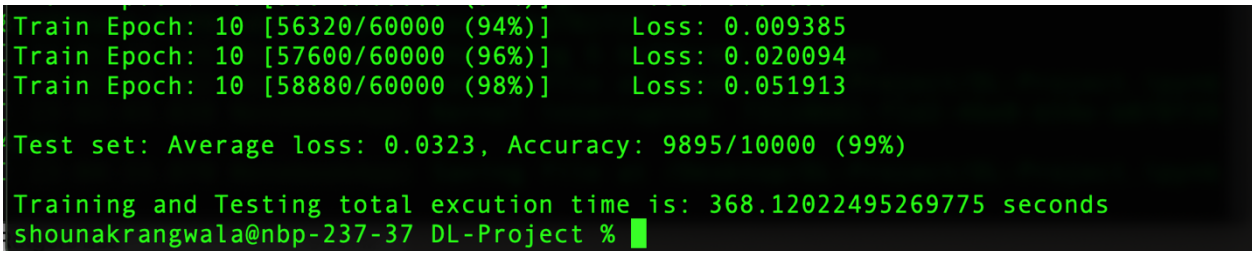
Num_features = the dimension over which batch's are made. This is also called spatial batch normalization.

Result:

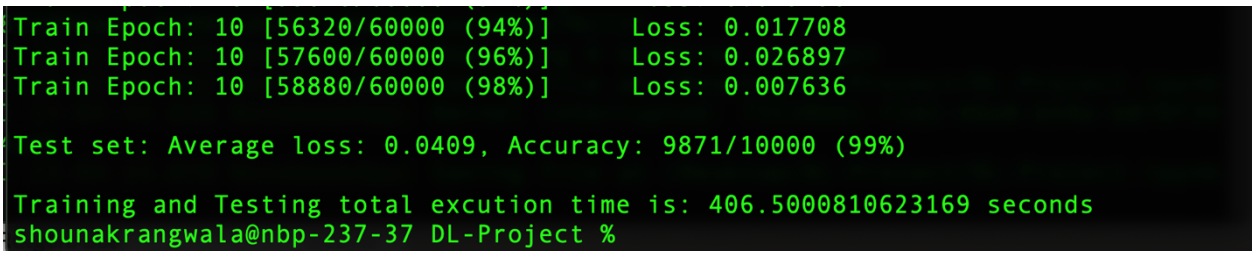
1) Case1: FC with dropout, CONV with BN



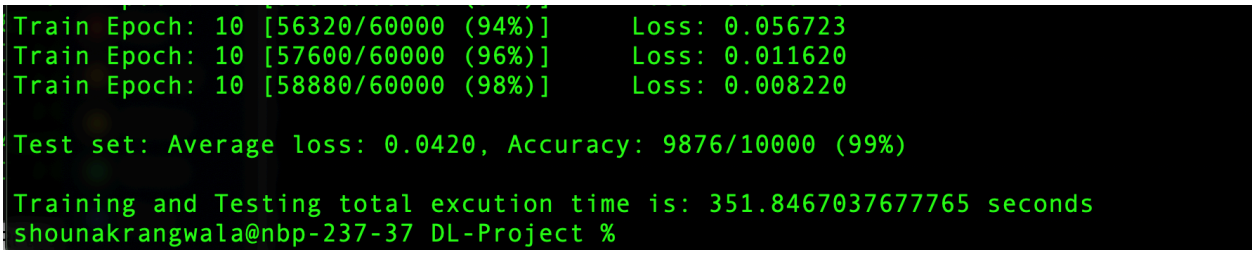
2) Case 2: FC with dropout, CONV without BN



3) Case 3: FC without dropout, CONV with BN



4) Case 4: FC without dropout, CONV without dropout



Conclusion:
Comparing on the basis of accuracy after 10 epochs: we can see that the fully connected layer with dropout and convolutional layer without Batch normalization.
Comparing on the basis of time taken to completion : fully connected layer without dropout and convolutional layer without batch normalization.

References:

1) https://pytorch.org/docs/stable/nn.html#torch.nn.Dropout
2) https://en.wikipedia.org/wiki/MNIST_database
3) https://en.wikipedia.org/wiki/Batch_normalization
4) https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
5) https://ruder.io/optimizing-gradient-descent/
6) https://pytorch.org/docs/stable/optim.html