Handwritten Digits Recognition

–

Manmeet Singh
ASUS

CS167 - Deep Learning

TABLE OF CONTENTS

For this Project I have decided to develop a model that can recognize handwritten digits.

I am using Torch to develop and train machine learning model.

Torchvision for importing and loading dataset.

Python TKinter library for user interface which leverages the use of Machine Learning model making it more user friendly by letting user write on canvas and process that as input to model.

X Dataset Exploration

For recognizing the digits, I have used MNIST dataset from torchvision originated from NIST. Which has over 70,000 samples 28×28 grayscale handwritten numbers.

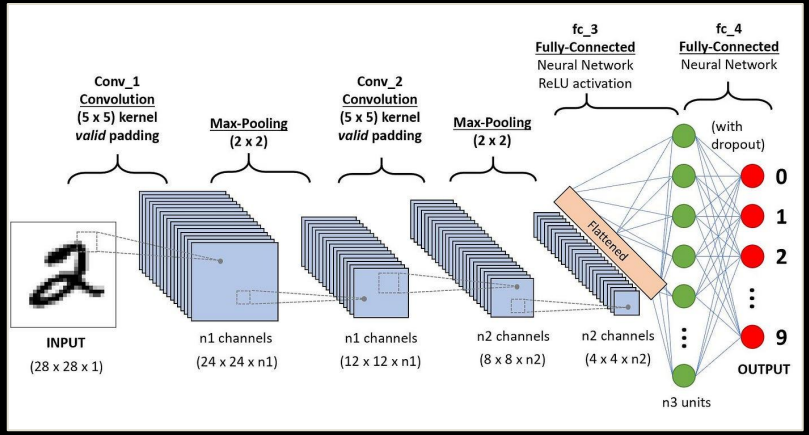
I am using 20% of the dataset to train the model. And 80% of it for testing. With batch size of 64 and shuffling the dataset after each epoch.

Doing around 50 epochs to well train the model so that I have low loss rate and high accuracy.

Using Transformers, transforming every single image to 28×28, greyscale and converting it to Tensor. Which then can be feed into model to train it.

Model Creation

- To Train my model I have used Convolution Neural Network with 3 layers, 1 input channel (grey scale) and 10 output channels (0-9). Using Kernel size of 3×3 carefully capturing every aspect of the input.
- By having 3 Conv2D layers then performing MaxPool of size 2×2 on each layer following ReLU function to filter negatives giving model more clear and detailed values to model.
- Since, this is a classification problem, I am using CrossEntropyLoss function which computes the logits of value between 0 and 1.
- On each of our 10 output channels we get probability between within that range and highest probability is what the model has predicted.

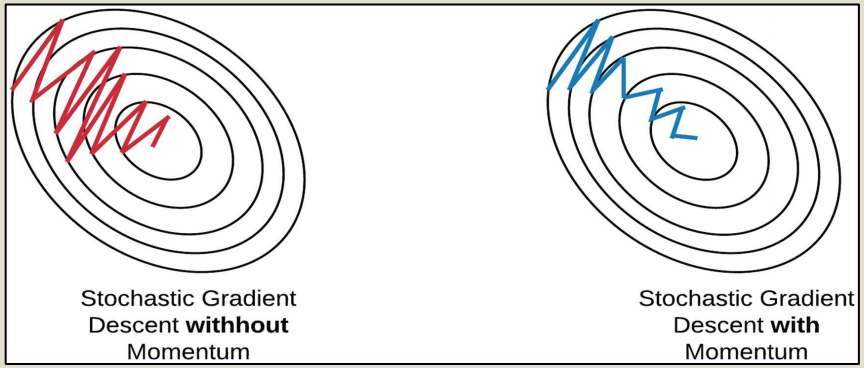


HyperParameter Exploration

Using SGD with learning rate of 0.01 keeping smooth for higher order curvatures along with momentum of 0.9 to reduce noise in dataset so that gradient doesn't get stuck at saddle point.

Changing the learning rate to higher would decrease the model accuracy because gradients might overshoot, and lowering the rate would increase the time it takes to lower the losses so for balanced learning rate 0.01 was the recommend by PyTorch as I was doing only 50 epochs of dataset.

Shuffling the dataset and dropping the last item from the batch has shown improved accuracy of model from 92% to 96% and doing about 60 epochs has given me around 98.42% of accuracy.

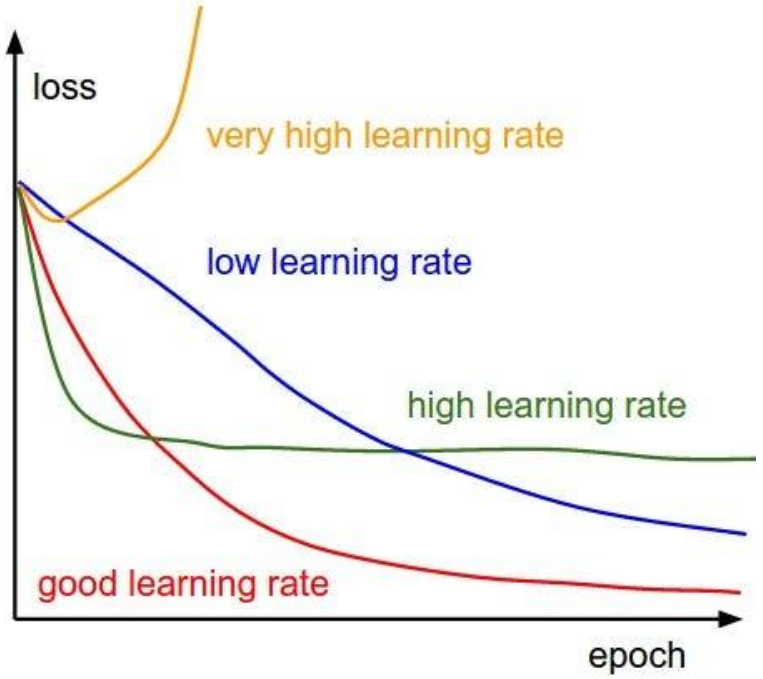


Performance get much better while training on GPU, 1 epoch of batch size 64 iterating over 11,850 samples takes around 6 minutes on RTX 3060 where on CPU i7 9900F takes around 14-15 minutes.

Effect of learning Rate

Learning rate helps to lower the error rate.

A good learning rate can be vary based on our size of data set, complexity of model and number of Epochs, having balanced approach can lower the loss by exponential rate. Which will predict more accurate results.



Results Documentation

```
Epoch 17, Loss: 0.4874
Epoch 18, Loss: 0.4858
Epoch 19, Loss: 0.4865
Epoch 20, Loss: 0.4855
Model Trained and Saved
Testing a model
Accuracy of the model on the test images: 82.68184279979025%
Model Trained.
Time taken to train the model: 96.81692179838817 minutes
Testing user given input images
Predicted Class: 3
```

Fig 1.4

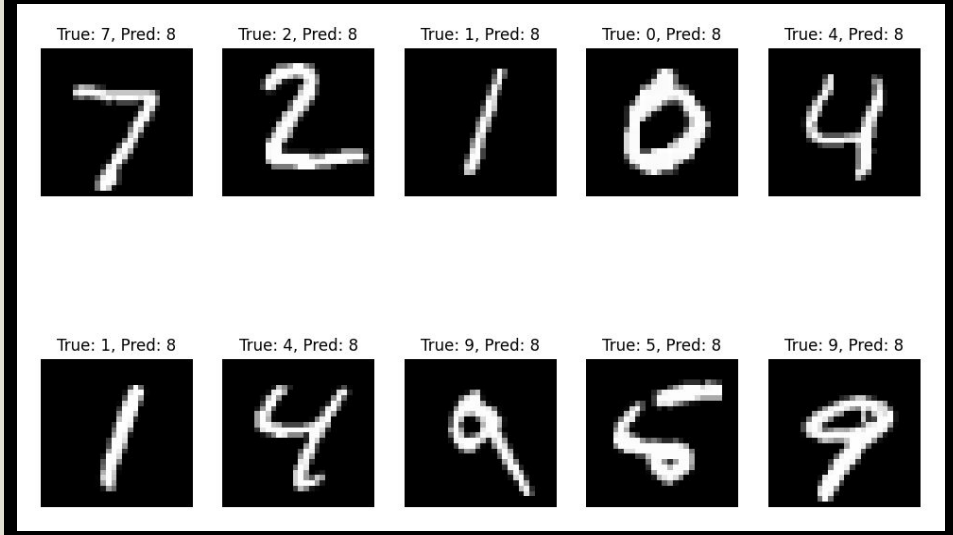
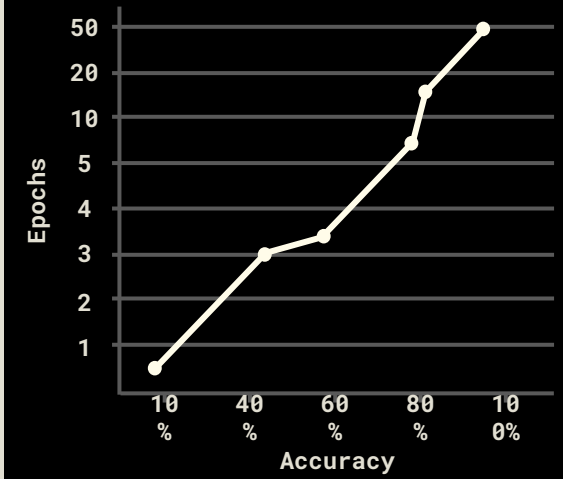


Fig 1.4 shows that after doing about 20 epoches, our model only gives about 82.6% of accuracy which is not enough to work with user given data. It it took about 1 hour and 47 mins to complete.

By saving this model in pytorch and running again with N epochs might lower it. To work this out and lower the loss rate which can be done by adjusting the learning rate and momentum.



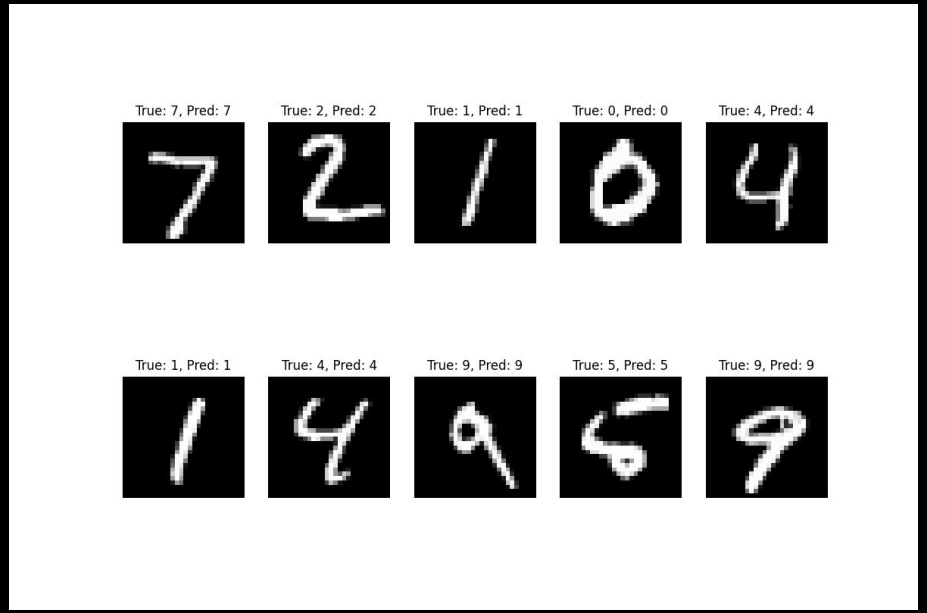
End Results

```
Epoch 10, Train Loss: 0.0105, Val Loss: 0.0336
Model Trained and Saved
Model Trained.
Time taken to train the model: 8.721180295944214 minutes
Testing a model
Accuracy: 99.08%
```

By Running another 10 epochs with learning rate of 0.01 and momentum of 0.09, and randomizing the batch of size 64 and dropping the last sample after each pass through to improve the performance of our model. So that model can explore other possibilities and not get stuck in same place in batch every time.

After little bit of tweaking the parameters I was able to get the most out of model which was about 99.35% of accuracy.

Fig 1.7 - Improved model




```
Epoch 56, Train Loss: 0.0049, Val Loss: 0.0238
Epoch 57, Train Loss: 0.0049, Val Loss: 0.0238
Epoch 58, Train Loss: 0.0049, Val Loss: 0.0238
Epoch 59, Train Loss: 0.0049, Val Loss: 0.0238
Epoch 60, Train Loss: 0.0049, Val Loss: 0.0238
Model Trained and Saved
Model Trained. MNIST [0-9]
Time taken to train the model: 35.68084814945857 minutes
Accuracy: 99.35%
```