Training AI on the MNIST Dataset Using CNN and LSTM Architecture

Jonathan D. Roberts

Department of Computer Science, College of Staten Island

Dr. Tianxiao Zhang

CSC 731: Artificial Intelligence & Knowledge Engineering

November 16th, 2025

Introduction

In this project, I will be training an AI model on the MNIST dataset, a dataset of handwritten digits (0-9) stored in 28x28 pixel images. When training an AI classification model for images, there are a few things that will need attention, including the architecture, hyperparameter tuning method and overfitting prevention method.

The architecture defines the way in which the model finds patterns and tunes its own weights and biases to become more accurate. Hyperparameter tuning changes the parameters within the model that affect its training. Finally, overfitting prevention is used to stop the model from fitting its results so closely to the training dataset that it cannot be used on new, unseen data.

In this case, the architecture that will be designed and trained are the Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) models. The hyperparameter tuning method will be the simple and easily implemented grid search. Lastly, I will be using early stopping to prevent overfitting.

CNN Architecture

This architecture uses two-dimensional feature maps to scan the image for patterns, and pooling layers extract the important ones to then finally flatten and classify the images. Figure 1.1 shows the source code for the CNN architecture.

```python
#CNN Class
class CNN_Classifier(nn.Module):
    #Setup for parameters
    def __init__(self, out_channels1 = 8, out_channels2 = 16, conv_stride = 1, pool_stride = 2, pooling_size = 2, padding = 1, kernel_size = 3):
        super(CNN_Classifier, self).__init__()
        final_size = (int)((((28 - kernel_size + (2*pooling_size)) / conv_stride) + 1) / (2*pooling_size))
        #First Convolution Layer ()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=out_channels1, stride=conv_stride, kernel_size=kernel_size, padding=padding)
        #Second Convolution Layer
        self.conv2 = nn.Conv2d(in_channels=out_channels1, out_channels=out_channels2, stride=conv_stride, kernel_size=kernel_size, padding=padding)
        #Pooling Layer (2 by 2)
        self.pool = nn.MaxPool2d(pooling_size, pool_stride)
        #Fully Connected Layer (16 * 14 * 14 is the final shape after pooling)
        self.fc1 = nn.Linear(out_channels2 * final_size * final_size, 10) #

    def forward(self, data):
        #Convolution Layer 1
        data = F.relu(self.conv1(data))
        #Pooling 1
        data = self.pool(data)
        #Convolution Layer 2
        data = F.relu(self.conv2(data))
        #Pooling 2
        data = self.pool(data)

        #Flatten the data
        data = data.view(data.size(0), -1)
        data = self.fc1(data)

        return data
```

Figure 1.1: CNN Classifier Source Code

Hyperparameter Tuning

Before training and testing the model, the first order of business should be to find out what hyperparameter values should be used. The hyperparameters we will explore in this case are going to be the batch size and learning rate. The batch size controls how much data to train on

at once. A large batch size may decrease noise while increasing overfitting and costing more memory. A smaller batch size may increase the noise that passes through, but it can decrease the risk of overfitting and uses less memory. The learning rate controls how large of a “step” the model takes to shift its weights/biases in the right direction. If it is too high, it may overstep and make training unstable. The losses will bounce up and down drastically. If set too low, training can take a long time or get stuck.

To find the optimal hyperparameters, we use a method called grid search. This method changes each hyperparameter, runs the model and records the best performance. Table 1.1 shows all the batch size and learning rate combinations tested with their results.

Table 1.1: CNN Hyperparameter Results

	lr = 0.001	lr = 0.002	lr = 0.005	lr = 0.010
bs = 64	98.86%	98.78%	98.83%	98.53%
bs = 128	98.98%	98.78%	98.70%	98.47%
bs = 256	98.92%	98.86%	98.78%	98.40%

It can be noted that the best most accurate combination is a learning rate of 0.001 and batch size of 128 with an accuracy of 98.98%. This is the combination that will be used for all future training, validation and testing.

Training

When training the model, we define a maximum number of epochs as well as an early stopping threshold. The maximum number of epochs will tell the program when to stop training once it exceeds that many passes through the training and validation stages. The threshold will stop it early, before reaching the maximum, if there have been too many epochs without improvement on the total loss. In this case, a maximum of 25 and a threshold of 3 was chosen. The large maximum and low threshold were chosen to give the model enough buffer so that it can reach its best state of total loss without running for too much time. This makes the training stage reasonably long.

Figure 1.2 shows the output of this stage. It can be observed that epoch 7, the best performing one, did not have the highest accuracy, but the lowest loss.

The loss of an epoch represents how close the predictions are to being correct, zero being correct with one-hundred percent confidence. While in the training and validation stage, the loss is valued more than the accuracy itself since it guides the gradients that determine the next step of the model’s training. A high loss shows that the model is very far off and should be tuned appropriately. The accuracy in this case does not matter because we want to guide the model towards the correct classification for when it is deployed on an unseen dataset. The scale and range of the loss may depend on the loss function you use. In this case, the loss function used for both models is the cross entropy loss function which is a very common loss function used in simple examples such as this one.

```
PS C:\Users\jonny\OneDrive\Documents\Python Projects\AI&KE_Assignment_2> py main.py --network cnn --batch_size 128 --lr 0.001
CNN Model: max_epochs=25, early_stop_threshold=3, batch_size=128, lr=0.001
 Epoch: 1
  Loss: 1.151
  Accuracy: 96.62
 Epoch: 2
  Loss: 0.756
  Accuracy: 97.70
 Epoch: 3
  Loss: 0.631
  Accuracy: 97.88
 Epoch: 4
  Loss: 0.608
  Accuracy: 98.07
 Epoch: 5
  Loss: 0.561
  Accuracy: 98.15
 Epoch: 6
  Loss: 0.577
  Accuracy: 98.25
 Epoch: 7
  Loss: 0.498
  Accuracy: 98.35
 Epoch: 8
  Loss: 0.512
  Accuracy: 98.40
 Epoch: 9
  Loss: 0.502
  Accuracy: 98.41
 Epoch: 10
  Loss: 0.565
  Accuracy: 98.22
[Stopping Early]
 Best Epoch: 7
 Best Val Loss: 0.498
 Best Accuracy: 98.35
```

Figure 1.2: CNN Model Training Outputs

Testing

After training the model, we restore the model back to the state in which it performed the best (epoch 7 as seen in Figure 1.2) and use it on the initial training dataset, the validation dataset as well the test dataset which it has yet to see. Figure 1.3 shows these results.

```
Best CNN Model Test (Training Dataset): size=50000, batch_size=128, lr=0.001
 Total Loss: 12.664
 Accuracy: 98.97

Best CNN Model Test (Validation Dataset): size=10000, batch_size=128, lr=0.001
 Total Loss: 0.498
 Accuracy: 98.35

Best CNN Model Test (Test Dataset): size=10000, batch_size=128, lr=0.001
 Total Loss: 0.435
 Accuracy: 98.63

Average Accuracy of CNN Model with: batch_size=128, lr=0.001
> 98.65%
PS C:\Users\jonny\OneDrive\Documents\Python Projects\AI&KE_Assignment_2>
```

Figure 1.3: CNN Model Testing Outputs

It should be noted that the total loss in the training dataset is far higher due to the dataset being five times larger. Remember that this is a total loss, not an average. To find the average, sum the total loss and divide it by 70,000 which is about 0.000194.

The accuracy is also higher in the training dataset which is to be expected since the model is trained on it which makes it better at classifying it. The average accuracy across the datasets is 98.65% as shown. Overall, the CNN model performed well and as expected.

LSTM Architecture

The LSTM architecture is originally designed to analyze sequences such as paragraphs of text while remembering context and information from previous lines known as timesteps. When analyzing an image, we treat each row of pixels as a timestep. In this case, we will have 28 timesteps of 28 pixels each. Figure 2.1 shows the source code for the LSTM model.

```
class LSTM_Classifier(nn.Module):
    def __init__(self, input_size = 28, hidden_size = 128, num_layers = 2, num_classes = 10):
        super(LSTM_Classifier, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True
        )
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, data):
        #data: (batch_size, seq_len, input_size)
        h0 = torch.zeros(self.num_layers, data.size(0), self.hidden_size).to(data.device)
        c0 = torch.zeros(self.num_layers, data.size(0), self.hidden_size).to(data.device)

        out, _ = self.lstm(data, (h0, c0))
        out = out[:, -1, :]
        out = self.fc(out)
        return out
```

Figure 2.1: LSTM Classifier Source Code

Hyperparameter Tuning

As seen previously with the CNN architecture, hyperparameter tuning is important for the LSTM architecture as well. The same hyperparameters, batch size and learning rate, need to be tuned.

Table 2.1 shows all the combinations and results of these parameters.

As seen in Table 2.1, the best performing accuracy has a batch size of 64 and learning rate of 0.001 at an accuracy of 99.18%. These are the hyperparameters that will be used for future training, validation and testing.

Table 2.1: LSTM Hyperparameter Results

	lr = 0.001	lr = 0.002	lr = 0.005	lr = 0.010
bs = 64	99.18%	98.81%	98.54%	96.57%
bs = 128	98.78%	99.13%	98.03%	96.34%
bs = 256	99.13%	98.73%	98.44%	96.29%

Training

To train the LSTM model, similarly to the CNN model, a max number epochs of 25 and patience threshold of 3 have been chosen. Figure 2.2 shows the results.

```
PS C:\Users\jonny\OneDrive\Documents\Python Projects\AI&KE_Assignment_2> py main.py --network lstm --batch_size 64 --lr 0.001
LSTM Model: max_epochs=25, early_stop_threshold=3, batch_size=64, lr=0.001
 Epoch: 1
  Total Loss: 1.185
  Accuracy: 96.62
 Epoch: 2
  Total Loss: 0.917
  Accuracy: 97.37
 Epoch: 3
  Total Loss: 0.743
  Accuracy: 97.67
 Epoch: 4
  Total Loss: 0.571
  Accuracy: 98.44
 Epoch: 5
  Total Loss: 0.650
  Accuracy: 98.09
 Epoch: 6
  Total Loss: 0.529
  Accuracy: 98.45
 Epoch: 7
  Total Loss: 0.519
  Accuracy: 98.64
 Epoch: 8
  Total Loss: 0.545
  Accuracy: 98.48
 Epoch: 9
  Total Loss: 0.500
  Accuracy: 98.57
 Epoch: 10
  Total Loss: 0.493
  Accuracy: 98.61
 Epoch: 11
  Total Loss: 0.518
  Accuracy: 98.61
 Epoch: 12
  Total Loss: 0.481
  Accuracy: 98.88
 Epoch: 13
  Total Loss: 0.486
  Accuracy: 98.66
 Epoch: 14
  Total Loss: 0.553
  Accuracy: 98.72
 Epoch: 15
  Total Loss: 0.496
  Accuracy: 98.77
[Stopping Early]
 Best Epoch: 12
 Best Val Loss: 0.481
 Best Accuracy: 98.88
```

Figure 2.2: LSTM Model Training Outputs

Testing

Using the best performing state from the training stage (epoch 12 as seen in Figure 2.2) all the datasets are tested. Out of the three datasets, the average accuracy obtained is 99.06%.

This is shown in Figure 2.3.

```
Best LSTM Model Test (Train Dataset): size=50000, batch_size=64, lr=0.001
 Total Loss: 5.450
 Accuracy: 99.57

Best LSTM Model Test (Validation Dataset): size=10000, batch_size=64, lr=0.001
 Total Loss: 0.481
 Accuracy: 98.88

Best LSTM Model Test (Test Dataset): size=10000, batch_size=64, lr=0.001
 Total Loss: 0.457
 Accuracy: 98.74

Average Accuracy of LSTM Model with: batch_size=64, lr=0.001
> 99.06%
```

Figure 2.3: LSTM Model Testing Outputs

The total loss, once again, is much higher in the training dataset because the sample size is five times larger than the other two. To find the average, we once again sum them and divide by the total sample size of 70,000. This would be about 0.000091.

Conclusion

The results show that the LSTM model had a marginally better performance than the CNN model. This may be explained by the small and grayscale nature of the images within the MNIST dataset. This would benefit the LSTM model while being a detriment to the CNN model because the LSTM model could easily find strong shapes and patterns due to the sharp lines and curves. The CNN model, on the other hand, benefits from images that it can reduce using more feature maps. Unfortunately, in this case, there were only two convolutional layers due to the small size of the images. This results in a total of 16 feature maps of 7x7 pixels each as noted in the source code in Figure 1.1.

Another observation is that the total loss in the CNN model is far more than the total loss from the LSTM model. This can be explained similarly to the accuracy being slightly lower, in that the size of the images affect the ability of the CNN model to perform while the LSTM model is not harmed. The reason for the difference being so large can be due to the fact that loss is far more “sensitive” than accuracy. When the CNN model predicts correctly with low confidence, the loss will be very high whereas when the LSTM model predicts correctly with high confidence, the loss will not be nearly as high.

Overall, the outcomes for both models went well with relatively high accuracies. However, it did not go quite as expected. The expectation was for the CNN model to outperform the LSTM model since it is designed for images unlike the LSTM model, but as seen above, it did not.

Recreating the Results
While exact results may not be repeatable, the following instructions will detail the process of downloading the correct dependencies and source code to run the same design yourself.

First, ensure that you have python installed by opening up your terminal and entering:

```
py --version
```

If it is not installed, install it on windows using:

```
winget install Python.python.3
```

After it is installed, you must install PyTorch by typing:

```
pip install torch
```

or if you have an NVIDIA GPU:

```
pip install torch --index-url https://download.pytorch.org/whl/cu121
```

After this is finished, extract the .py files into a folder and navigate there within your terminal.

To run the model that you want, enter the following:

```
py main.py --network (nw) (others)
```

To specify hyperparameters, replace (others) with your choice of the following:

```
--epochs (ep) --patience (p) --batch_size (bs) --lr (lr)
```

If you would like to find the most optimal hyperparameters yourself, there is a tuning mode that can be used by entering:

```
py main.py --network (nw) --tuning True
```

This takes a long time so be prepared to wait.

This should allow you to train and test with the models detailed in this report.
Enjoy your experiments!