Classifying Handwritten Digits (MNIST) with a Multi-Layer Perceptron

Pranav Mucharla Santosh Kandula Aayush Prakash Kaiji Fu

Abstract

This paper will outline the process of designing the architecture for the classifier, training, and testing the handwritten number classifier developed by the authors. The paper also explains the model's design and its goal of finding features in images of handwritten numbers, and reports the accuracy and loss values for each training epoch. Additionally, there is a focus on the first layer of perceptrons in the model, analyzing importance maps for each number to explain the differences between classifications and generation in this scenario. Finally, the paper will conclude with a summarized discussion of the model architecture, the inherent idea and design of this classifier, and why that prevents it from being able to generate.

1. Introduction

In the realm of artificial intelligence, a multitude of architectures, techniques, methodologies, and theories have been implemented to build various types of AI in recent years. In this paper, the focus is on one of these types: classifying images, specifically handwritten numbers. The goal is to understand how classifiers analyze pixels in images to identify features and how the specific classification method used here prevents the model from generating handwritten numbers on demand, instead restricting it to recognizing them. This analysis utilizes heatmaps to visualize the weights assigned to each pixel in the image, enabling the reader to better understand why the weights don't exactly map to numbers in an MLP and why this limitation prevents the model from generating numbers.

2. Data

This classifier was trained on the data provided by the MNIST dataset. MNIST provides 70,000 labeled images of handwritten numbers, pairing each image with the digit it represents, which allows models to train and test easily using this dataset. Additionally, MNIST data has one number per image, and keeps it roughly centered within the picture. The dataset has an approximately 85/15 split between training and testing, with 60,000 images allocated for training and 10,000 images for testing. Using PyTorch, the training program for this model can automatically download, separate, and load the MNIST data into distinct training and testing sets.

3. Model Architecture and Rationale

The model used to classify the handwritten numbers is an MLP (Fig. 1) with four total layers (one input layer, 2 hidden layers, and one output layer). At each of the hidden layers, the non-linearization of choice is RELU, which has become a standard in many modern MLP architectures and generally has better results than other non-linearities. The input layer has 784 neurons, as the image consists of a total of 784 pixels (28 x 28). This is fed into the first hidden layer of 512 neurons (with RELU applied after), followed by another hidden layer of 256 neurons (which also has RELU applied after). The output layer consists of 10 neurons, corresponding to the numbers one through nine, and each number is assigned a specific value (the number with the highest value determines the model's decision). The idea behind using such an architecture, as is common with the motivation for similar MLP architectures, is that the weights in the first hidden layer will eventually reach a point where it is classifying different small features in the image, and the second layer will combine these features to decide which number an image is. Although this may not be the model's ultimate thought process, using multiple layers and non-linearities allows for combinations of components at different layers, making the model more complex and better at classifying different types of numbers and various ways to represent each number.

4. Model Training

The PyTorch library was used to code the training loops and fetch the data from the MNIST dataset. The model architecture used for training was as described in the previous section. The loss function used was Cross-Entropy Loss, and the optimizer used was an Adam optimizer, which updates all the trainable parameters in the model with a learning rate of 0.001 (1e-3). To feed the data during train-

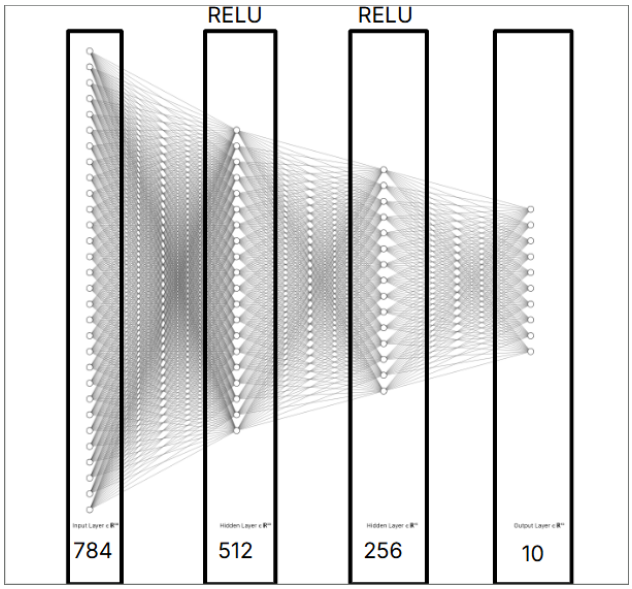

Figure 1. The model architecture for the MNIST classifier in this paper, visualized.

```
Epoch 1
-------------------------------
loss: 2.307217  [    0/60000]
loss: 0.611163  [ 6400/60000]
loss: 0.206281  [12800/60000]
loss: 0.172908  [19200/60000]
loss: 0.195918  [25600/60000]
loss: 0.177992  [32000/60000]
loss: 0.063362  [38400/60000]
loss: 0.176489  [44800/60000]
loss: 0.105998  [51200/60000]
loss: 0.095575  [57600/60000]
Test Error:
 Accuracy: 96.8%, Avg loss: 0.100689
```

(a) Epoch 1

```
Epoch 3
-------------------------------
loss: 0.025318  [    0/60000]
loss: 0.022461  [ 6400/60000]
loss: 0.104844  [12800/60000]
loss: 0.055862  [19200/60000]
loss: 0.089604  [25600/60000]
loss: 0.167934  [32000/60000]
loss: 0.065166  [38400/60000]
loss: 0.033530  [44800/60000]
loss: 0.054354  [51200/60000]
loss: 0.112229  [57600/60000]
Test Error:
 Accuracy: 97.7%, Avg loss: 0.074034
```

(b) Epoch 3

```
Epoch 5
-------------------------------
loss: 0.008972  [    0/60000]
loss: 0.010252  [ 6400/60000]
loss: 0.045671  [12800/60000]
loss: 0.014685  [19200/60000]
loss: 0.074669  [25600/60000]
loss: 0.006757  [32000/60000]
loss: 0.013741  [38400/60000]
loss: 0.034146  [44800/60000]
loss: 0.047102  [51200/60000]
loss: 0.019541  [57600/60000]
Test Error:
 Accuracy: 98.0%, Avg loss: 0.075748
```

(c) Epoch 5

Figure 2. The training epochs one, three, and five, with the loss for every 100th batch, and the results of testing each time.

ing, the model utilizes a PyTorch DataLoader, which creates an iterable to facilitate easy looping through during training. Additionally, the DataLoader breaks up the MNIST training data into batches of 64 and shuffles the data. During training, the program runs for five epochs (with testing after each epoch), printing the loss every 100 batches and displaying the test accuracy and average loss.

5. Results

After training was completed, the model achieved promising performance on the MNIST classification task. By the end of the first epoch of training, the loss had already decreased significantly, and when tested, the model accurately classified 96.8% of the images. Over the five epochs, testing accuracy increased to 98%, with an average loss of 0.075748. Figure 2 illustrates the training loop in greater detail for epochs 1, 3, and 5.

6. Interpreting the Model

While this model has shown promising results for classifying the numbers in the MNIST dataset, the question becomes whether the model can generate a drawing of a number based on everything it has learned. A few things can be determined when examining the weights for the first layer of neurons for each number, which were intended to encapsulate certain features of the drawings of numbers. Figure 3 contains the importance maps, based on the weights applied to the first layer, for each of the numbers in the classifier. Upon examination, it becomes apparent that the maps don't accurately represent the numbers they are intended to classify, except for some minor features. Additionally, this importance map is likely to differ significantly if a new model is trained, due to the randomness of the starting weights. This means there is no way for the model to use these weights to generate an image, even though these weights are perfectly capable of classifying an image. Additionally, there are two other matrices of parameters that combine the first-layer weights in different ways and apply further non-linearities, which makes it even harder to generate the requested image, but once again supports high-accuracy classification. This limitation in the MNIST model shows one major distinction between classification models (the model in this paper) and generative models, such as LLMs. Though the training processes may be similar, they are inherently different, and the data used and optimization done to the model function much differently in LLMs than in this MNIST model.

7. Conclusion

For this project, the MNIST classifier was built not only as a tool but also as a subject to analyze how decisions of architecture and training procedure affect its representations. This full cycle of tasks, starting from data management and optimization processes to visualizing the weights, helped to realize the need for robust analysis while making conclusions regarding behavior at all levels of classification or object-detection tasks like MNIST classification or object detection. This paper also showed how the classification

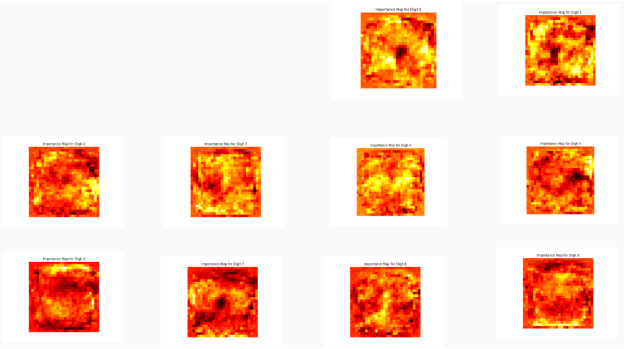

Figure 3. The importance map for each digit is shown here, ranging from 0 to 9, from left to right and top to bottom. Lighter colors indicate that the models assign more importance to that pixel relative to other pixels, and the scale ranges from black to white.

and training techniques utilized by the model are only suited for classification, not generation. By using visual maps of the weights at the first layer of the model, it becomes more apparent and easy for the reader to understand why this limitation exists in this type of model. At the same time, the analysis was limited to a very simple MLP network and just one of its layers, which may limit the conclusions one can make.