Homework 1 - Classifying MNIST Data using DenseNet and CNN

Ben Fox
University of California, Santa Barbara

April 14, 2019

Abstract

This paper demonstrates the training and testing of the MNIST dataset utilizing two models, a general deep neural network (DNN), and a convolutional neural network. All models were built using python and Tensorflow. Keras was not used, because the goal of this assignment was to gain a full understanding of what is happening during training and easily visualize this via Tensorflow code. Overall, the two classifiers produced good results. The DNN, with 5 layers, each layer with correspondingly fewer neurons than the last had accuracy of 96.7% and precision of 97.8%. The CNN also performed well. The CNN contained three convolutional layers, and one fully connected layer. It had a test accuracy of accuracy of 97.9% and precision of 98.5%. Overall, this was a great exercise in building a DNN and CNN from scratch using tensorflow.

1 Data

For this assignment, the dataset used was the MNIST handwritten digit dataset (found here) comprised of 60,000 images of handwritten digits for training and 10,000 images of handwritten digits for testing. This dataset is perfect for people just getting involved with deep learning and convolutional neural networks (CNN). An example of the image from the dataset is shown in figure **??**.

Fortunately, this data is pre-proccessed to 28 by 28 images and anti-aliased, so no further pre-processing was done on the dataset, except normalizing each image by dividing by 255 to represent pixels between range zero and one.

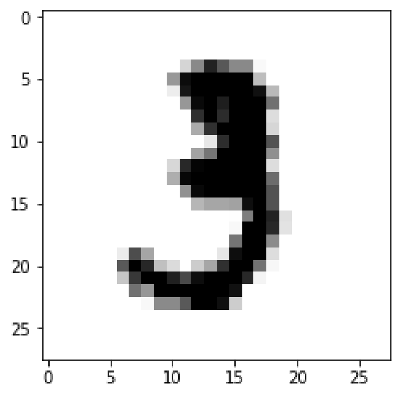


Figure 1: Example image from MNIST Dataset

Overall, this dataset was a good starting point for me, as this is my first time developing a deep learning method for image classification. Because of this, I did not use Keras, rather I used tensorflow to understand what's happening on the backend for the various calculations that are done in a deep learning process. This proved to be a great starting point for my understanding of deep learning and its multitude of applications.

2 Network

2.1 DNN

The DNN built was a five layer neural network. It was comprised of a total of 390 neurons. The associated network graph, scaled down by 10, is shown in figure **??**. Each inner layer utilized the Relu activation function. The output utilized a Softmax activation function. The layers had 200, 100, 50, 30, and 10 neurons respectively. Weights were initialized to random numbers following a normal distribution. Biases were initialized to 0.1, except for the last layer where biases were initialized to zero. Biases were 0.1 because of the chosen Relu activation function for the first four layers. The optimizer used was the Adams optimizer and the loss function used was cross entropy. The learning rate was adjusted following an exponential decay function each iteration.

$$lr = 0.0001 + 0.003 * \frac{1}{e}^{step/2000} \tag{1}$$

Finally, each iteration, a dropout rate of 0.7 was chosen to randomly dropout neurons in the model to generalize better to unseen data.

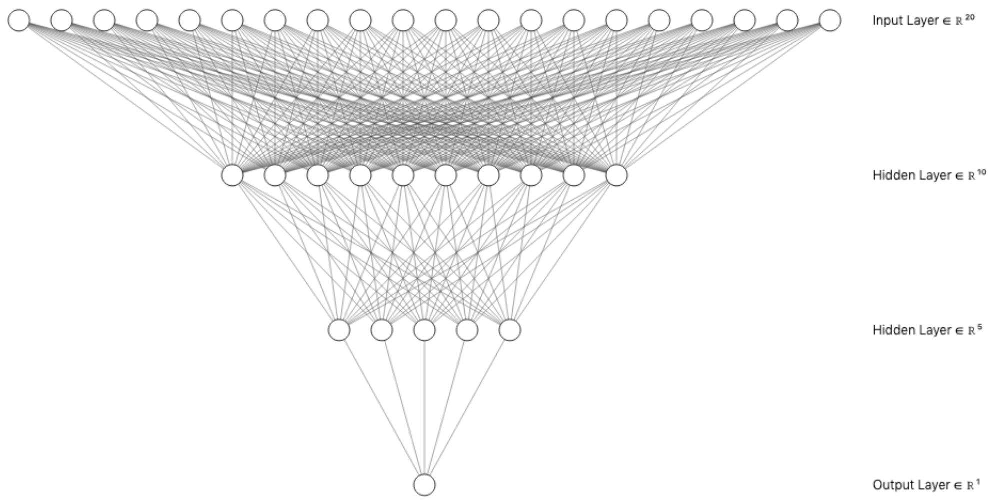


Figure 2: DNN Graph

Layer	Number of Neurons	Activation
Input Layer	200	Relu
Layer 1	100	Relu
Layer 2	50	Relu
Layer 3	30	Relu
Output Layer	10	Softmax

Table 1: DNN Model

2.2 CNN

The CNN built was also a five layer neural network, with three convolutions and one fully connected layer. The associated network graph is shown in figure **??**. Each inner layer utilized the Relu activation function. The output utilized a Softmax activation function. The layers' filter sizes are detailed in table **??**. As in the DNN, weights were initialized to random numbers following a normal distribution. Biases were initialized to a value close to 0 (0.1), except for the last layer where biases were initialized to zero. The optimizer used was the Adams optimizer and the loss function used was cross entropy. The learning rate was adjusted following an exponential decay function each iteration.

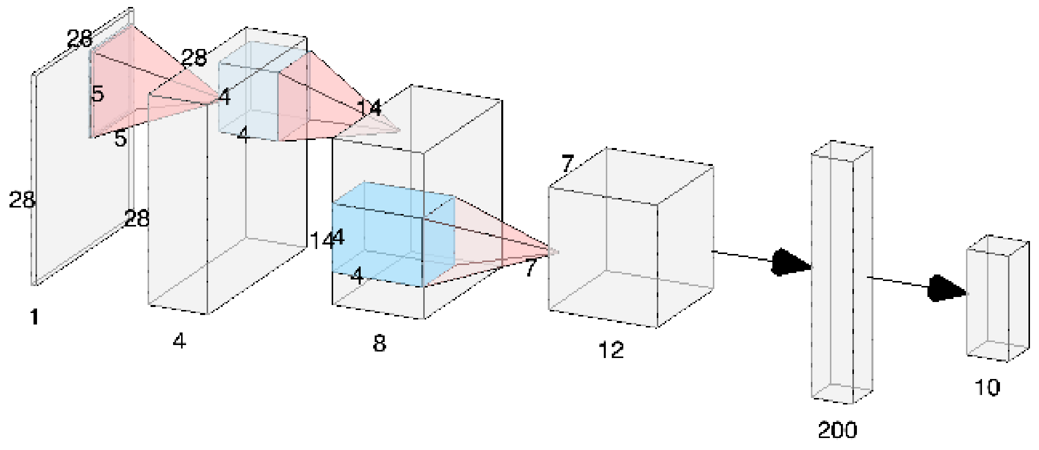


Figure 3: CNN Graph

Layer	Size	Filter Size	IO Channels	Stride	Activation
Input Image	[28,28,1]	NA	NA	NA	NA
Layer 1	[28,28,4]	[5,5]	[1,4]	1	Relu
Layer 2	[14,14,8]	[4,4]	[4,8]	2	Relu
Layer 3	[7,7,12]	[4,4]	[8, 12]	2	Relu
Layer 4 (fully connected)	[200]	NA	[7x7x12, 200]	NA	Relu
Output Layer	[10]	NA	[200,10]	NA	Softmax

Table 2: DNN Model

3 Training

Each mini batch size was 100 images. For both DNN and CNN, 20 epochs of training was done. The corresponding loss per epoch are shown in figure **??** below for both CNN and DNN.

4 Validation

For both models, there was an indication of overfitting when the loss of the training set continued to decrease but the loss of the test remained the same and potentially increased. Figure **??** below demonstrates this for both CNN and DNN. Additionally, precision and

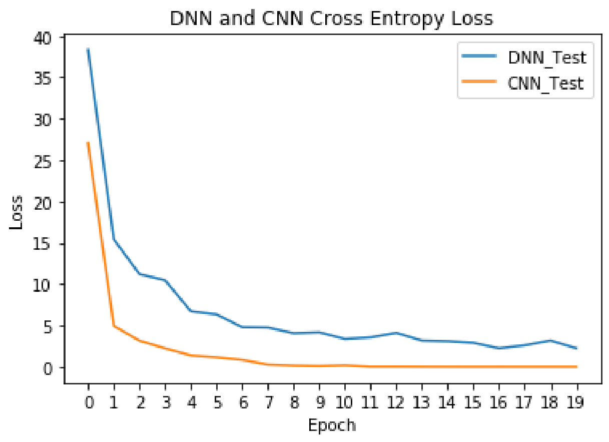
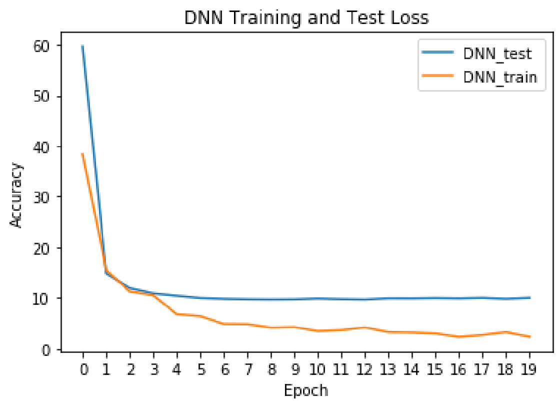
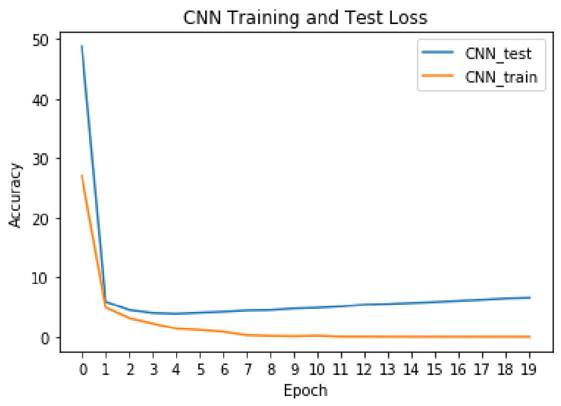


Figure 4: Cross Entropy Loss

accuracy measures are reported for the test set over the for both models. See table **??**.



(a) DNN



(b) CNN

Figure 5: DNN and CNN Train and Test Loss

Model	**Accuracy**	**Precision**
DNN	96.7%	97.8%
CNN	97.9%	98.5%

Table 3: DNN/CNN Stats

5 Discussion

Overall, this was a big learning experience for me. I have yet to implement a full DNN or CNN utilizing only tensorflow, and this proved to be very beneficial. Specifically, I learned new details around activation functions, initializations, and the similarities and differences between DNN and CNN.

First, I learned that utilizing the Relu activation function within networks instead of the classic sigmoid yields better results because of the vanishing gradient problem. Additionally, the differences between the three activation functions I looked at(sigmoid, relu, and softmax) are more clear (i.e. softmax better maximizes small differences between class outputs, thus giving a clearer picture of which class to select - this is why it's typically the final activation function).

Second, initialization of weights and biases is extremely important, not to mention with what values, one initializes these weights and biases to be. Utilizing random, normally distributed variables typically yields better results for weights, and utilizing non-zero values for biases, when using Relu, helps avoid the vanishing gradient problem as well.

Lastly, doing both the DNN and CNN was useful because I got to see the entire process of building a very simple DNN followed by a more complex CNN. This distinguished the differences between the two, namely that the CNN is able to examine pictures more in context than the DNN. The DNN essentially breaks down the image into an array of pixels and has no context for which pixels are next to each other. Overall, it was clear that the pipeline for both was very similar and tensorflow provides an easy way to implement both.