Table of Contents

1. ABSTRACT

We work on the classic MNIST dataset and evaluate the performances of various models. We look at Artificial Neural Networks (ANN) and Convolutional Neural Networks (CNN).

2. INTRODUCTION

The MNIST dataset is a classic classification problem. It involves images of handwritten digits and labels corresponding to those digits such as in Figure 1. The classification part revolves around learning and recognizing those digits. There are many variations or extensions, but we focus on single-digit MNIST.

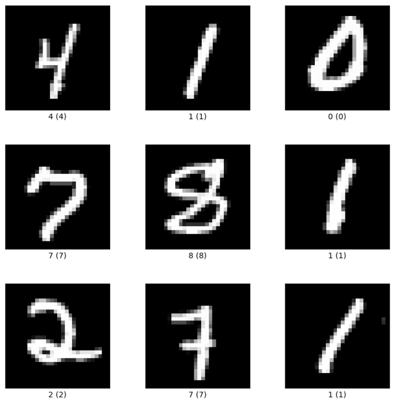


Figure 1. MNIST dataset images of handwritten digits [1].

3. PREPROCESSING AND MODELS

3.1. Dataset

The first step is checking the data. The images are 28 x 28 pixels. The photo is encoded as individual pixels within a 2D array. Each pixel is itself encoded by an array that stores RGB values. Each image is labeled with classes of digits from 0 to 9.

As seen in Figure 2., there does not appear to be a significant disparity between the number of digits in the dataset. The counts are not exactly the same, but they are relatively even compared to instances of rare diseases for instance. The exact counts are displayed in Table 1. All counts for the 10 classes lie within the range [6313, 7877] with 70,000 total in the dataset. Comparatively, this dataset may not have such a significant class imbalance. The worst imbalance ratio is 0.801447 between digits 5 and 1 as in Table 2. The dataset is relatively balanced.

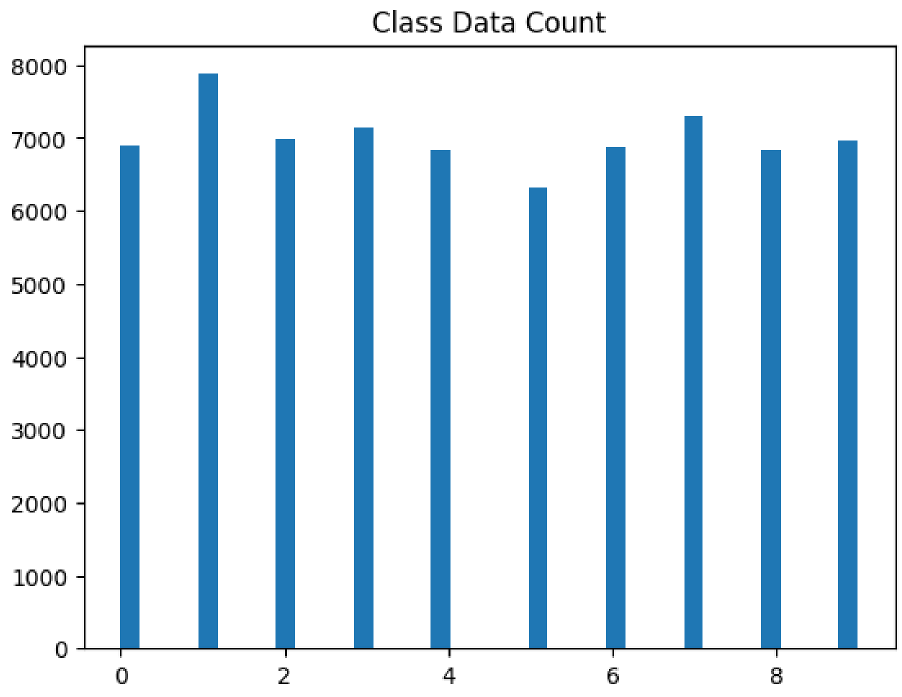


Figure 2. Histogram plot of occurrences of labeled digits in dataset.

Table 1. Exact count of occurrences of labeled digits in dataset. Counts highlighted in red are the minimum and maximum.

```
0: 6903,                                5: 6313,
1: 7877,                                6: 6876,
2: 6990,                                7: 7293,
3: 7141,                                8: 6825,
4: 6824,                                9: 6958
```

Table 2. Imbalance ratio calculation. All calculations are <minority class> over <majority class>. The ratio highlighted in red is the minimum.

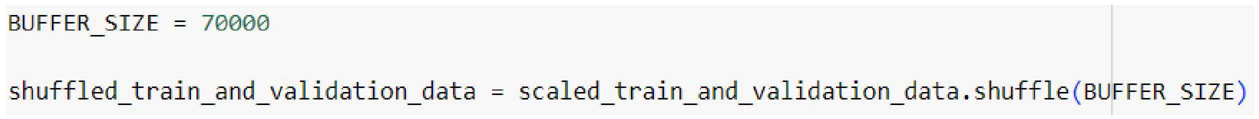
```
Imbalance ratio of classes:
0 over 1:    0.876349      3 over 7:    0.979158      5 over 0:    0.914530
0 over 2:    0.987554      4 over 0:    0.988556      5 over 1:    0.801447
0 over 3:    0.966671      4 over 1:    0.866320      5 over 2:    0.903147
0 over 7:    0.946524      4 over 2:    0.976252      5 over 3:    0.884050
0 over 9:    0.992095      4 over 3:    0.955608      5 over 4:    0.925117
2 over 1:    0.887394      4 over 6:    0.992437      5 over 6:    0.918121
2 over 3:    0.978855      4 over 7:    0.935692      5 over 7:    0.865625
2 over 7:    0.958453      4 over 8:    0.999853      5 over 8:    0.924982
3 over 1:    0.906563      4 over 9:    0.980742      5 over 9:    0.907301
```

6 over 0:	0.996089	7 over 1:	0.925860	8 over 7:	0.935829
6 over 1:	0.872921	8 over 0:	0.988701	8 over 9:	0.980885
6 over 2:	0.983691	8 over 1:	0.866447	9 over 1:	0.883331
6 over 3:	0.962890	8 over 2:	0.976395	9 over 2:	0.995422
6 over 7:	0.942822	8 over 3:	0.955748	9 over 3:	0.974373
6 over 9:	0.988215	8 over 6:	0.992583	9 over 7:	0.954066

3.2. Methodology

3.2.1 Shuffling

The dataset saved on TensorFlow should already have been shuffled. However, it is best to shuffle it ourselves as well. We do so with the following code.



```
BUFFER_SIZE = 70000

shuffled_train_and_validation_data = scaled_train_and_validation_data.shuffle(BUFFER_SIZE)
```

Figure 3. Python to shuffle loaded training dataset from TensorFlow MNIST

We have 70,000 elements in the data. These elements in the MNIST dataset are already split among a “train” and “test” sets. The training (and validation) set must strictly have fewer than 70,000 elements. We set BUFFER_SIZE to that value to shuffle among the entire training dataset rather than shuffling chunks at a time. We want to avoid scenarios where chunks contain only one or very few distinct labels. Shuffling these chunks would be mostly ineffective and pointless. Moreover, it seems that my computer can handle it, so there is no harm in doing so.

3.2.2 Training, Testing and Validation Datasets

The TensorFlow MNIST dataset already partitions the data into “train” and “test” groups. Of the 70,000 elements total, 60,000 are in the “train” set and 10,000 are in the “test” set.

We use the “test” set as our testing set. However, we need to create a validation set. We will select 10% of the “train” group to be our validation set which will be 6,000 elements. We extracts the first 6,000 elements after shuffling and saves it as the validation set. The rest of the 54,000 elements will be our training set. This can be done easily in Python through slicing or extracting.

3.2.3 Batching

We want to divide up the datasets into batches. The specific size of each batch can be experimented with. We batch all datasets with the same batch size for each to match and remain consistent. While the training dataset should be batched, the other datasets will also be batched because the model will expect data to be in batches.

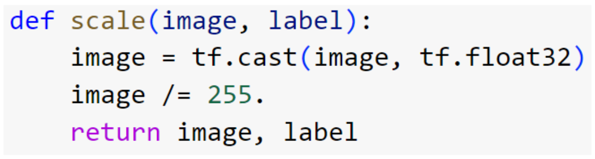
Regarding training, the idea of batching is to update weights after each batch instead of each epoch. In doing so, we reach a minimum quicker with a little loss in our accuracy. Theoretically, if we use bigger batches, we might increase our accuracy. At the same time, smaller batches may be able to help reach the minimum quicker and stopping us immediately as we are about to overfit.

3.3. ANN

Beyond the above, there are more steps that we take for the ANN as detailed below.

3.3.1 Scaling

Given that the images are encoded as individual pixels, these pixels could range from black (RGB [0,0,0]) to white (RGB [255,255,255]). We want to scale these values down while preserving the inherent color distribution. To that end, we simply divide all pixel values by the max value 255.



```python
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255.
    return image, label
```

Figure 4. Python code to scale down each image.

3.3.2 Flattening

For the input to the ANN, it expects a vector. We cannot simply input the 2D array image as is. Thus, we apply Flatten. This reduces (or flattens) the 2D image into a long vector that we can input into the ANN and train. It effectively looks like the rows have been concatenated together along the ends. The images in MNIST are 28x28 pixels. The resulting vector will be 784 elements as our input. This can be done in TensorFlow with the Flatten layer.

3.4. CNN

Detailed below are further steps that we take for CNN.

3.4.1 Convolution

The images are encoded as individual pixels in a 2D matrix, but not all of these pixels are entirely useful. In fact, some of them may end being noise. Convolution is taking the input image and passing it through filters that will find where critical features are within the input image. We simultaneously reduced the size of the original matrix and the noise while maintaining the crucial features. These saved positions are what is called a feature map. We create many of these feature maps.

We can also apply activation functions to apply non-linearity. We do so and choose relu to force the image to produce a more stark contrast among color gradients. These can be done with Conv2D in TensorFlow.

3.4.2 Pooling

Pooling is taking a sliding window across the input and taking the most prevalent feature. It is common done after convolution. In doing so, pooling is further reducing the size to only the most prevalent features. This thereby reduces overfitting as the train dataset and the actual input after convolution and pooling are different.

Pooling gives an extra advantage missing from convolution. It gives the model to ability to learn spatial invariance. Convolution mainly detected features, but pooling takes a subwindow of the input and reduces it down to the most prevalent element. Inputs that are slightly rotated or shifted within this window will remain familiar to the model. These can be done with MaxPooling2D in TensorFlow.

3.4.3 Flattening

The mechanics of Flattening remains the same as mentioned for ANN. The output from pooling will eventually be input into ANN. ANN expects a vector input, so we need to Flatten. Due to the convolution and pooling, the output from Flattening in CNN will look different than in ANN.

4. RESULTS

4.1. ANN

The following are the model details for the ANN.

- Learning rate = 0.001
- Optimizer = ADAM
- Input size = 784
- Output size = 10
- Output layer activation function = softmax
- Batch size = 100
- Early stopping patience = 2
- # of hidden layers = 2
    - Hidden Layer #1:
        - Activation Function = relu
        - Size = 100
    - Hidden Layer #2:

    - Activation Function = relu
    - Size = 100
- Loss function = sparse_categorical_crossentropy
- Maximum epochs (may not hit from early stopping) = 50
- Initializer = Xavier Glorot initialization

The plots below show the history of accuracy and loss values during model training. We can see a few points where the validation loss increases (i.e. could be overfitting). We can see them at epochs 11, 13, and 19 on the Loss plot in Figure 5. We have set the patience to continue training despite the validation loss increasing at epochs 11 and 13. At these epochs, we can see that both the training and validation loss remained decreasing as training went on.

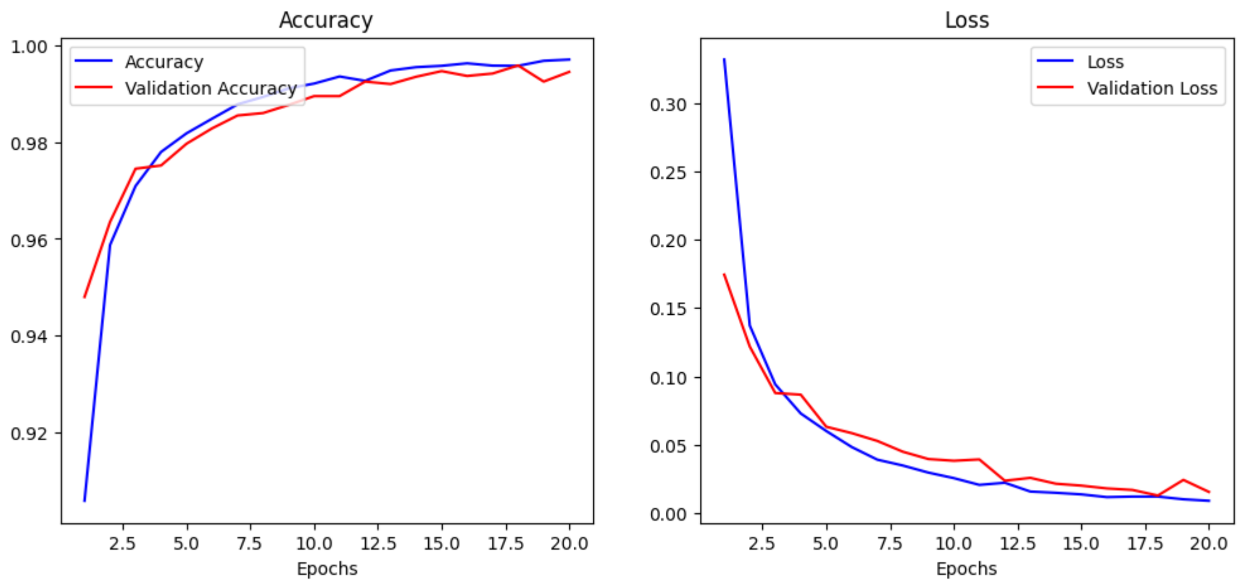


Figure 5. Accuracy and loss plots of ANN model training.

Figure 6 shows the confusion matrix for all ten classes (digits 0 to 9). The associated Precision, Recall, F1-Score, and Accuracy are display in Table 3. All the metrics seem to perform quite well; many are close to 1 and all are least 95%. It appears that the worst scores are recall for digit 9 with a recall score of 0.95342 and digit 8 with a recall score of 0.95277. Looking at the confusion matrix in Figure 6, the most misclassified digits of 9 were predicted as 7 with 16 errors and as 4 with 13 errors. The most misclassifications of 8 were predicted as 5 with 13 errors and as 0 with 12 errors. It's somewhat understandable that these similar-looking digits would be confused, especially with different handwriting styles. We will try with CNN to see if convolution and pooling layers can help the model with this.

As previously mentioned, there is not a significant class imbalance in the dataset. We can even see in the support column that there is a relatively nice, even distribution of labeled digits in the test dataset ranging from [892, 1,135] with a mean of 1,000. However, this does not mean the datasets are perfectly balanced. Thus, we check the F1 scores, accuracy, macro average, and weighted average.

The macro and weighted averages take into account F1 scores across the many classes. The weighted average applies a weight based on class occurrence within the dataset. The macro average weighs all classes evenly. The test accuracy, macro average, and weighted average are 0.97570, 0.97545, 0.97568 respectively. These values are nearly the same, meaning that either the dataset is sufficiently balanced or that any imbalance does not severely impact the model. I suspect the reason that any imbalance does not have a major effect is because the classes may be separable.

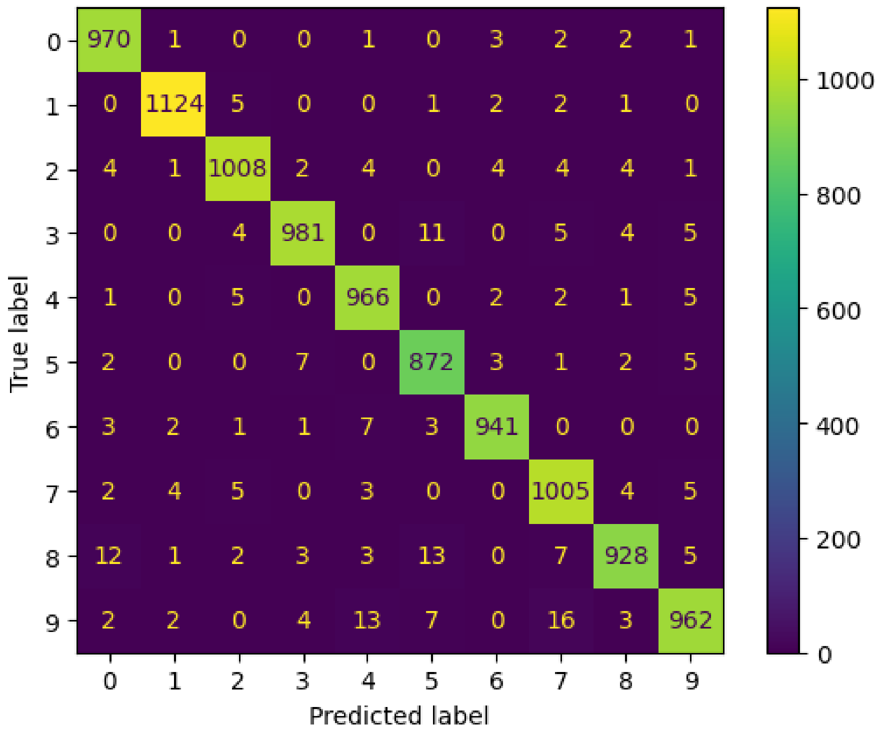


Figure 6. Confusion matrix of ANN predictions on test dataset.

Table 3. Metrics report (includes Accuracy, Precision, Recall, F1-Score)

```
              precision    recall  f1-score   support

           0    0.97390   0.98980   0.98178       980
           1    0.99031   0.99031   0.99031      1135
           2    0.97864   0.97674   0.97769      1032
           3    0.98297   0.97129   0.97709      1010
           4    0.96891   0.98371   0.97625       982
           5    0.96141   0.97758   0.96943       892
           6    0.98534   0.98225   0.98380       958
           7     0.96264    0.97763    0.97008       1028
           8     0.97787    0.95277    0.96516        974
           9     0.97270    0.95342    0.96296       1009

    accuracy                           0.97570      10000
   macro avg     0.97547    0.97555    0.97545      10000
weighted avg     0.97577    0.97570    0.97568      10000
```

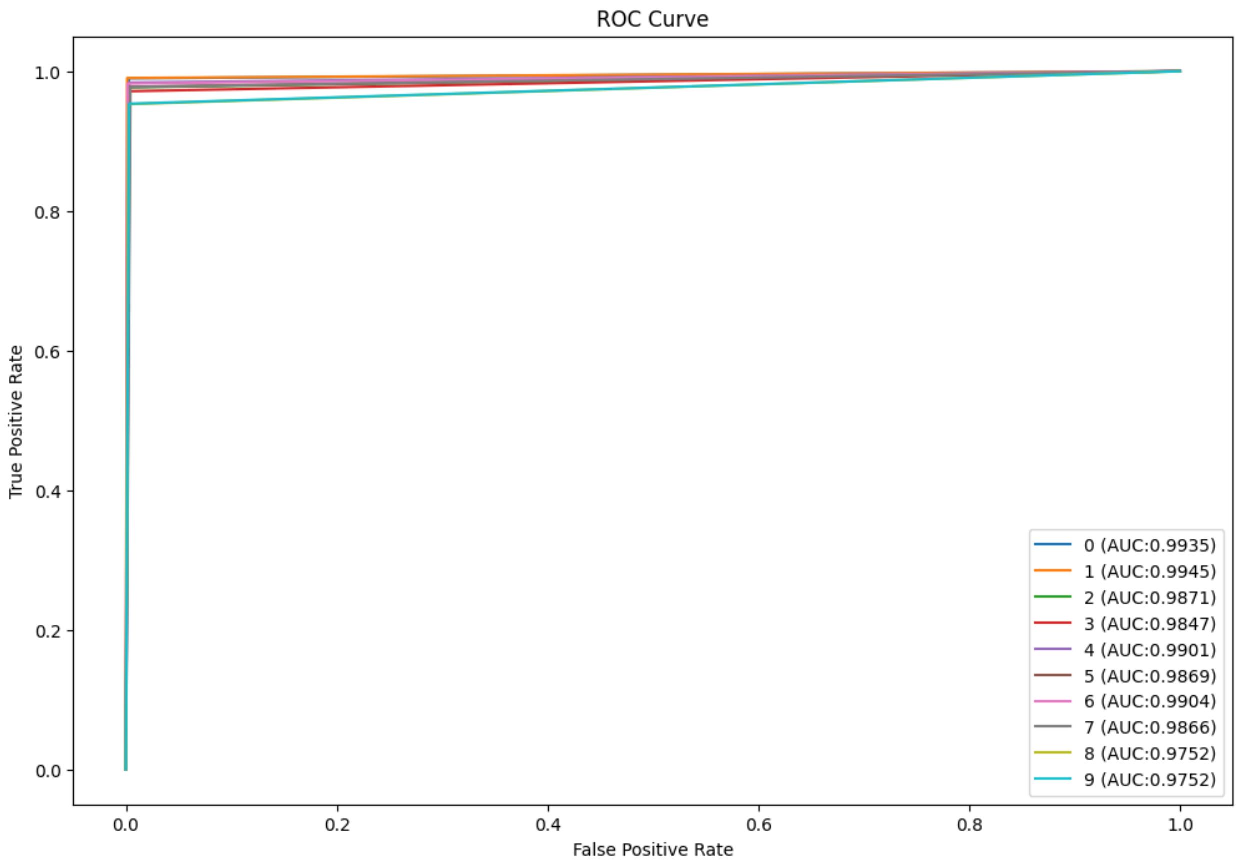


Figure 7. ROC Curve of ANN. AUC are displayed in the legend along with associated classes.

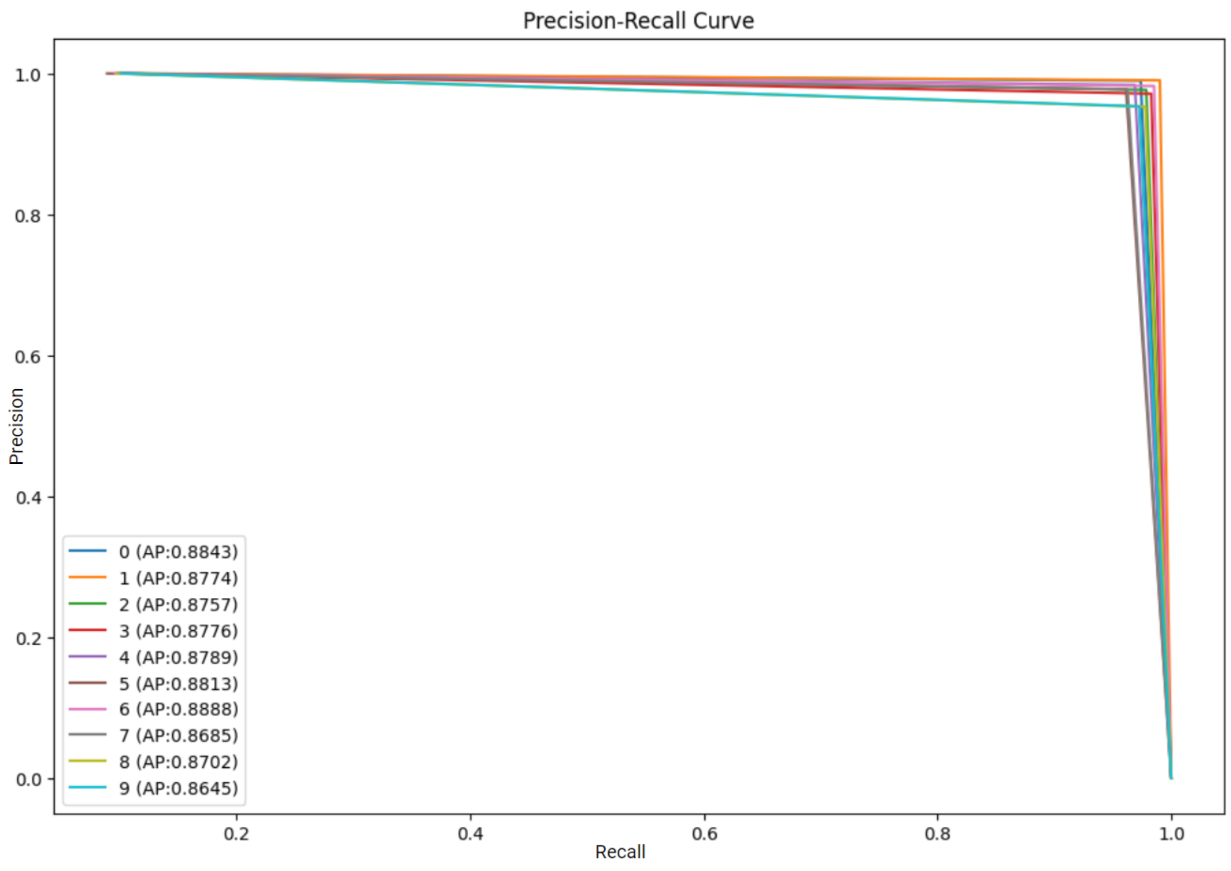


Figure 8. PR Curve of ANN. AP (Average Precision) are displayed in the legend along with associated classes.

Above are ROC and PR curves. Given that this is multiclass classification, we have used a One vs Rest strategy to plot the curves for each class. We'll assess how the model is and the balance of the dataset. Looking at the ROC curve in Figure 7, the model seems to perform quite well. It looks very close to a perfect classifier that has the vertical cliff along the edges of the graph. Each curve has an AUC close to 1. Curves for digits 8 and 9 performed the worst at 0.9752, corroborating our observations within the metric calculations and confusion matrix. We also check PR curve since ROC curve can be unrepresentative for imbalanced data.

The dataset is decently balanced. Looking at the PR curve in Figure 8, all the curves seem roughly similar to each other. Despite diverging, the curves do follow a same trend of their cusps. Additionally, we can see that the area under the curves calculated as AP (Average Precision) hover around [0.86, 0.88]. So, the curves are very similar and seem to show that the dataset is decently balanced.

4.2. CNN

The following are the model details for the CNN.

- Learning rate = 0.001

- Optimizer = ADAM
- Input size = 28x28 images
- Output size = 10
- Output layer activation function = softmax
- Batch size = 100
- Early stopping patience = 2
- # of layers = 8
    - Layer #1: 2D Convolution Layer
        - Activation Function = relu
        - Size = 32
        - Stride = (3, 3)
    - Layer #2: 2D Pooling Layer
        - Stride = (2, 2)
    - Layer #3: 2D Convolution Layer
        - Activation Function = relu
        - Size = 64
        - Stride = (3, 3)
    - Layer #4: 2D Pooling Layer
        - Stride = (2, 2)
    - Layer #5: Flatten
    - Layer #6: Dense Hidden Layer #1
        - Activation Function = relu
        - Size = 100
    - Layer #7: Dense Hidden Layer #2
        - Activation Function = relu
        - Size = 100
    - Layer #8: Output layer
        - Activation Function = softmax
        - Size = 10
- Loss function = sparse_categorical_crossentropy
- Maximum epochs (may not hit from early stopping) = 50
- Initializer = Xavier Glorot initialization

The plots below show the history of accuracy and loss values during model training. There are a few points where the validation loss increases (i.e. could be overfitting) at epochs 5, 10, and 11 on the Loss plot in Figure 9. Our current setting of patience may have been a good choice since after each of these epochs, both the training and validation loss remained decreasing as training went on.

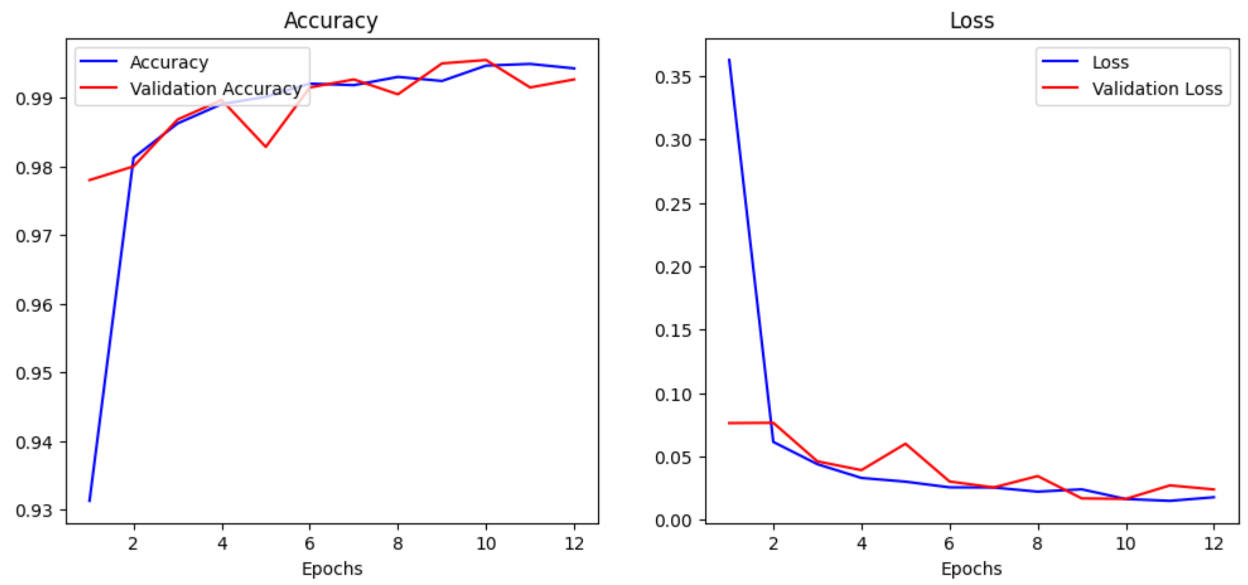


Figure 9. Accuracy and loss plots of CNN model training.

Figure 10 shows the confusion matrix for all ten classes (digits 0 to 9). The associated Precision, Recall, F1-Score, and Accuracy are display in Table 4.

Comparing this CNN and the ANN, the confusion matrix and metrics seem to reveal a better performance than ANN. The lowest score on Table 7 appears to be precision for digit 4 with a precisiomn score of 0.97702. However, ANN was even worse with 0.96891 as shown in Table 3. In fact, CNN outperformed ANN on all metrics, particularly the worst metric from ANN with recall on digits 9 and 8.

In confusion matrix for ANN as shown in Figure 6, the most misclassified digits of 9 were predicted as 7 with 16 errors and as 4 with 13 errors. The confusion matrix for CNN in Figure 10 shows misclassifying 9 as 7 with 0 errors and 4 with 10 errors. In response to a proposed earlier question, it appears that adding the convolution and pooling served to help the model better distinguish between 7 and 9 especially. It also seems to have helped to a lesser extent with distinguishing between 4 and 9. Furthermore, the other most misclassifications for ANN were of 8 predicted as 5 with 13 errors and as 0 with 12 errors. CNN misclassified 8 as 5 with 0 errors and 0 with 2 errors.

We now look at the F1 scores in terms of the macro and weighted averages. The accuracy, macro average, and weighted average are 0.98800, 0.98793, 0.98800 respectively. These values are nearly the same, and we reach the same conclusion as before. As repeated, either the dataset is sufficiently balanced or that any imbalance does not severely impact the model. I suspect the reason that any imbalance does not have a major effect is because the classes may be separable.

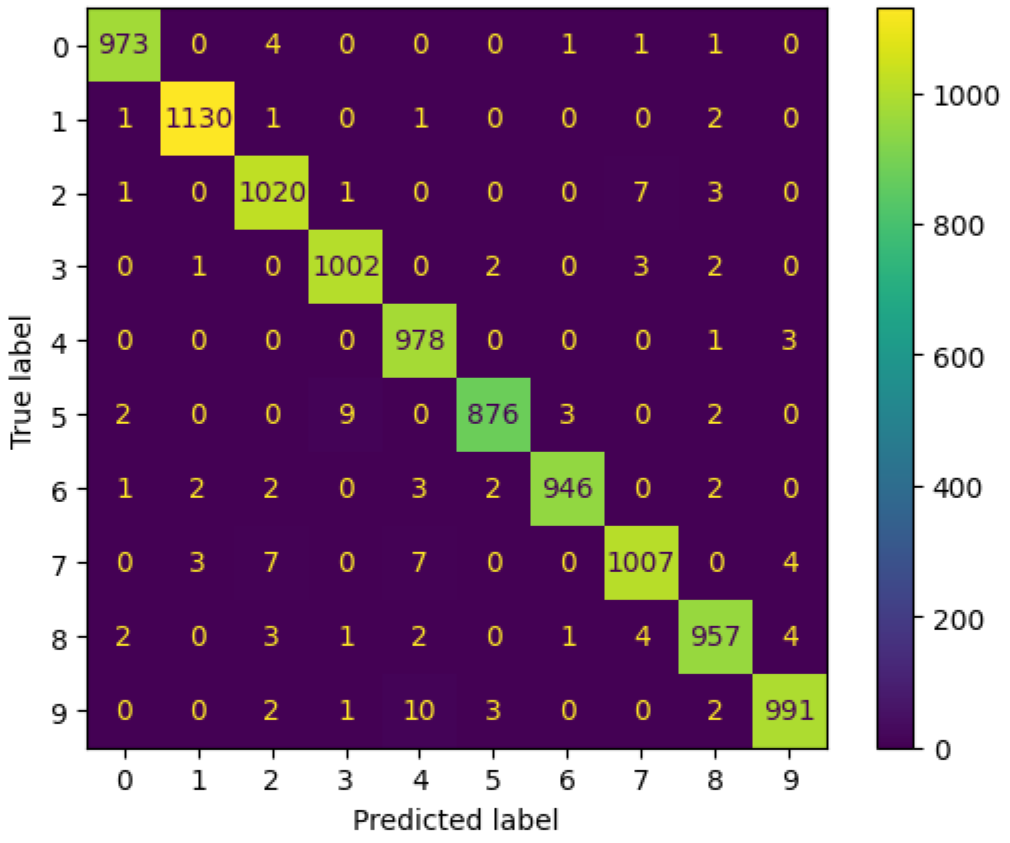


Figure 10. Confusion matrix of ANN predictions on test dataset.

Table 4. Metrics report for CNN

	precision	recall	f1-score	support
0	0.99286	0.99286	0.99286	980
1	0.99472	0.99559	0.99516	1135
2	0.98171	0.98837	0.98503	1032
3	0.98817	0.99208	0.99012	1010
4	0.97702	0.99593	0.98638	982
5	0.99207	0.98206	0.98704	892
6	0.99474	0.98747	0.99109	958
7	0.98532	0.97957	0.98244	1028
8	0.98457	0.98255	0.98356	974
9	0.98902	0.98216	0.98558	1009
accuracy			0.98800	10000
macro avg	0.98802	0.98786	0.98793	10000
weighted avg	0.98803	0.98800	0.98800	10000

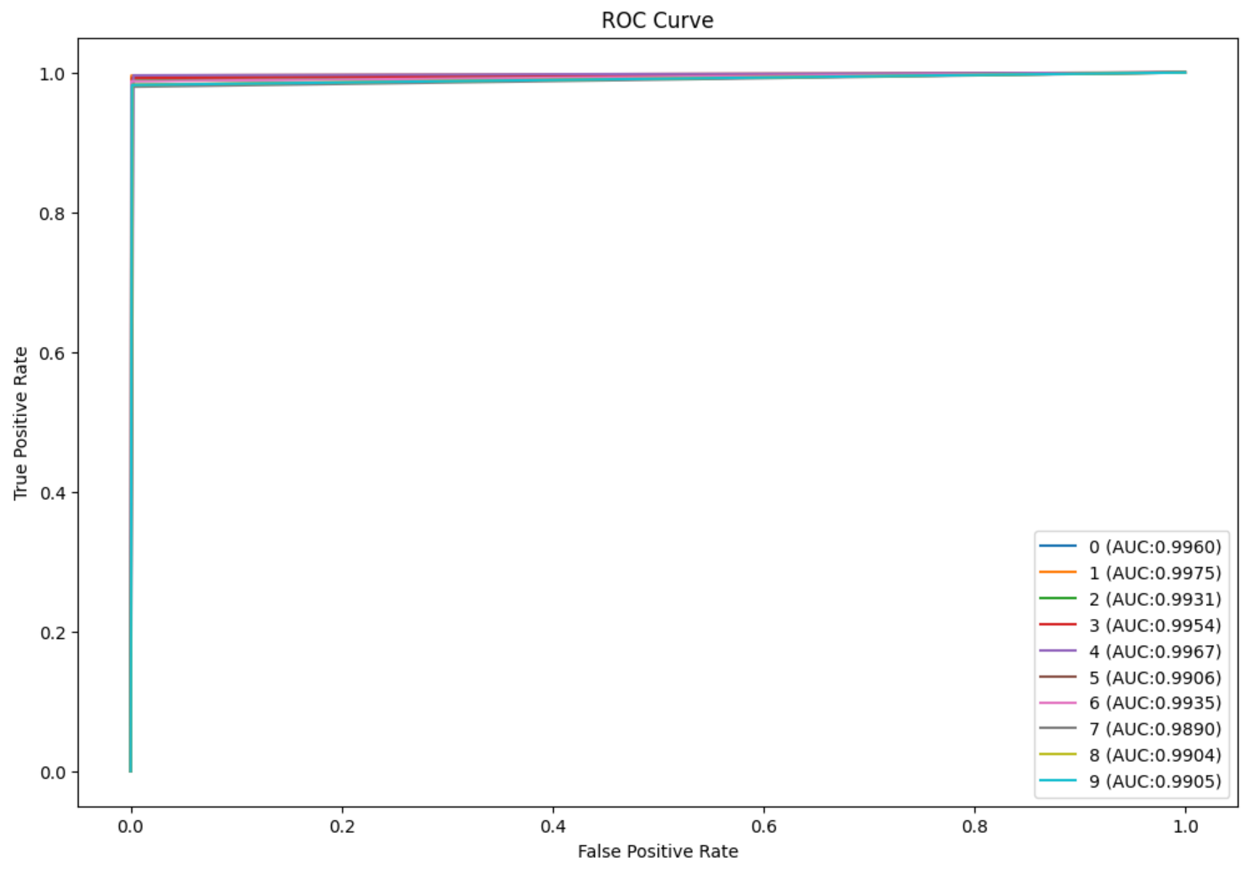


Figure 11. ROC Curve of CNN. AUC are displayed in the legend along with associated classes.

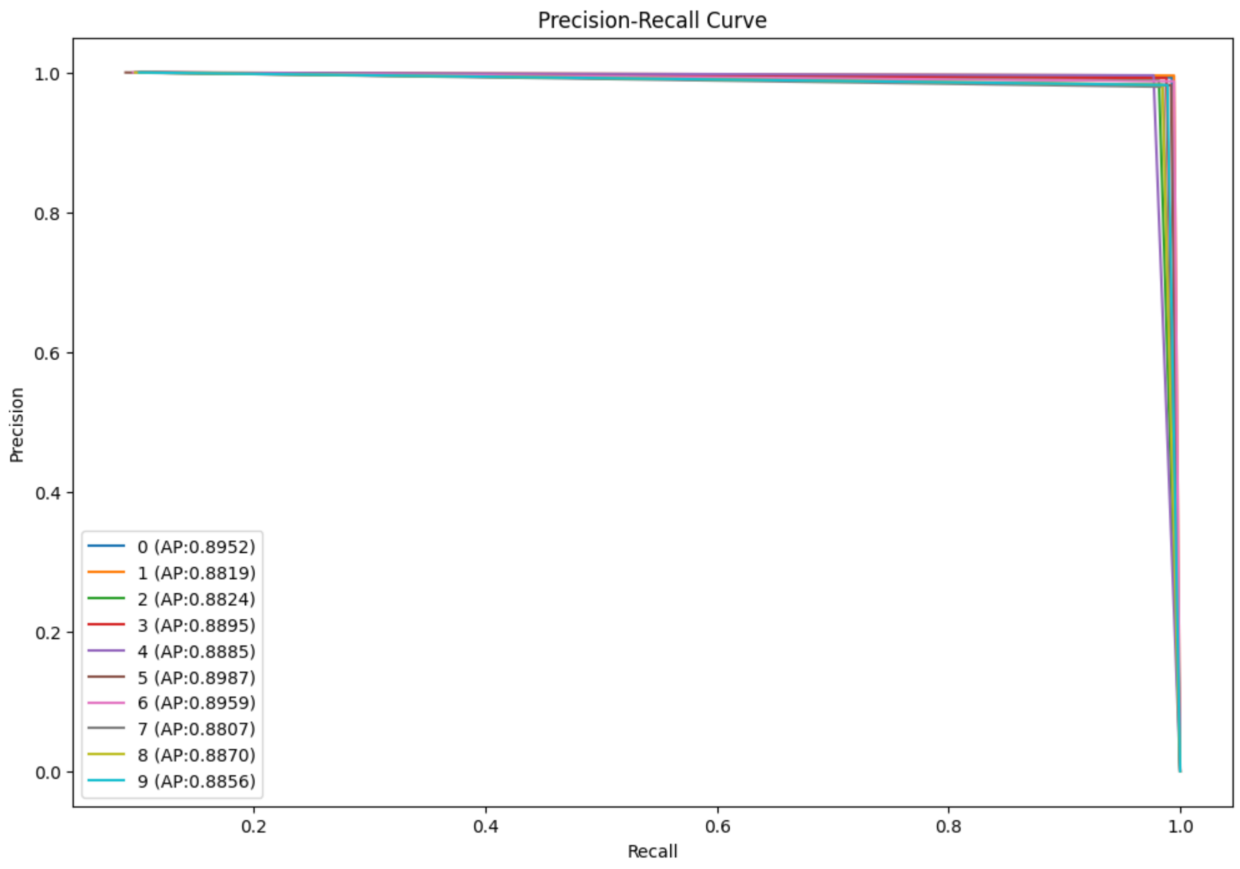


Figure 12. PR Curve of CNN. AP (Average Precision) are displayed in the legend along with associated classes.

Above are the ROC and PR curves for CNN using a One vs Rest strategy to plot the curves for each class. We'll assess model performance and dataset balance. Corroborating our observation from the metrics, the ROC curve for CNN in Figure 11 outperforms the ANN in Figure 7. The AUC is greater for every curve and is thereby closer to 1. The worst curves remain to be digits 8 and 9 with the lowest AUC of 0.9904 and 0.9905 respectively. Regardless, the ROC curve seems to imply that the model performs better and well overall. However, once again, the ROC curve can be unrepresentative for imbalanced data, so we check the PR curve.

The PR curves for CNN in Figure 12 seems to be tight-knit as opposed to those in Figure 8 for ANN. These curves for CNN primarily look to have a more narrow range of divergence and have their cusps much closer to each other. The AP for all curves for CNN also outperform their counterparts for ANN.

Taking into account all metrics and plots from all tests, it seem to point to that the MNIST dataset is nicely well-balanced or any imbalance was insignificant to severely impact the model or the classes are separable. The CNN model has a better model performance than the ANN, particularly attributing to the convolution and pooling.

5. CONCLUSION AND FUTURE WORK

The CNN seems to work better than the ANN. Unfortunately due to time constraints, there are additional experiments I would have liked to do and additional models such as CRNN. I leave those experiments as future work.

Naturally, future work would be best suited to expand to further variations and extensions of MNIST. However, there are a couple experiments that could maybe utilize MNIST as more of a helpful analogy. In the context of image-based malware classification, I had a few ideas. Of course, there are valid ways of applying noise or image manipulation to simulate distortion or obfuscation. I think there may be merit in trying to look for blocking out chunks to test recognition, simulating something similar to malware packet fragmentation.

MNIST takes photos of handwritten digits. I would like to try recognizing digits based on the handwriting motion. There does exist research into the actual motion of handwriting. Though, they may be a little more morally grey since recognizing and authenticating signatures or wills is great, whereas generation is a little different. Within the context of malware classification, I think one potential is to look at control flows - especially since the program and files can be internally obfuscated but malware families should share a similar central core. Following and learning a central control flow seems similar to the handwriting motion. That said, getting the control flow definitely sees more hurdles with encryption and modification, but it’s a nice thought to have.

Most programs are pretty well modularized. (I’d assume malware is too, but I don’t know. I have not sandboxed and willing downloaded and disassembled a virus on my only computer nor am I a malware writer.) Does this mean that the specific order of malware binary files is not crucial? Could I shuffle around small chunks of the binary-converted image files and yield the same recognition results? If so, does that mean that the image files within malware families can be rearranged to create a unique footprint? I’m not sure, but one of my guesses is to use CRNN to test mandatory sequential order first. In a way, testing order and modularity may come about from something similar to multi-digit MNIST, trying to recognize a sequence of digits. In addition, it may be interesting to test with recognizing characters in languages with common stroke patterns.

6. REFERENCES

[1] https://www.tensorflow.org/datasets/catalog/mnist