COMP3105 A4 Report

Mario Pardo-101286566 Dante Farinon-Spezzano-101231566

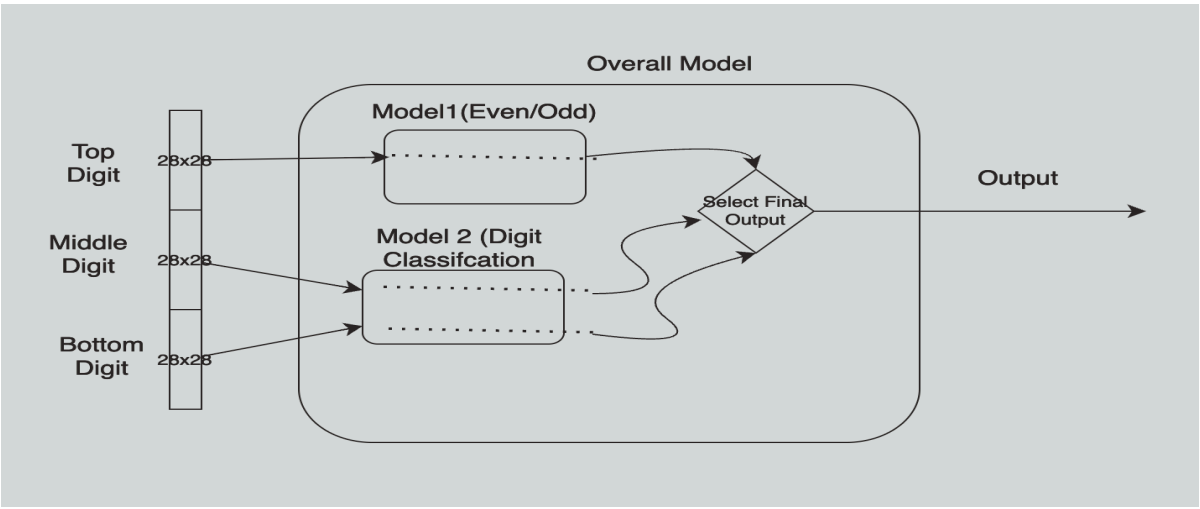
Algorithm Explanation

For this project, we chose to use a large hybrid neural network, with inner “submodels”, one to find if the top digit is even/odd, and then one that classifies the bottom and middle digits. The individual models use Convolutional Layers to best analyze the images, since they are image-like in nature.

The first 784 datapoints pertain to the top digit, this gets fed into the first model which outputs a number between 0 and 1, this is used as the even/odd “label”.

Independently, another model is created to perform actual digit classification for the middle/bottom images. It takes in one image, and outputs a vector of length 10 with the probability that this image is each digit. This model is used to predict the middle and bottom images **independently**

Once we have the probabilities of the middle and bottom image, we take the output of model 1 which tells us if even/odd and use this to mask, and therefore “select”, the desired output for the whole model. For example, if model 1 tells us it’s even, then the entire model will output what model 2 outputted.



Design Process

Finding the best algorithm took lots of trial and error, and analyzing what approaches we could use given the data we are given.

1 - Direct Digit Classification

We tried training a model directly for digit analysis. We know that the label of the image pertains to either the middle or bottom image so we trained a model where the label is actually “true” about 50% of the time. We knew that since the model would be getting trained on bad data, it would likely not perform well and might perform even worse on validation data.

As expected, this low quality training data lead to a model that could predict digits only about 30% of the time.

```
Epoch 1/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 19s 48ms/step - accuracy: 0.3359 - loss: 1.9104 - val_accuracy: 0.0320 - val_loss: 2.6313
Epoch 2/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 15s 46ms/step - accuracy: 0.5046 - loss: 1.4224 - val_accuracy: 0.2330 - val_loss: 2.4438
Epoch 3/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 23s 55ms/step - accuracy: 0.5263 - loss: 1.2985 - val_accuracy: 0.3190 - val_loss: 2.2338
Epoch 4/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 15s 46ms/step - accuracy: 0.5448 - loss: 1.2152 - val_accuracy: 0.2660 - val_loss: 2.4154
Epoch 5/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 15s 45ms/step - accuracy: 0.5491 - loss: 1.1752 - val_accuracy: 0.3520 - val_loss: 2.2307
Epoch 6/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 21s 46ms/step - accuracy: 0.5623 - loss: 1.1259 - val_accuracy: 0.3630 - val_loss: 2.1924
Epoch 7/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 15s 44ms/step - accuracy: 0.5764 - loss: 1.0863 - val_accuracy: 0.3890 - val_loss: 2.1761
Epoch 8/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 20s 43ms/step - accuracy: 0.5920 - loss: 1.0675 - val_accuracy: 0.3830 - val_loss: 2.3639
Epoch 9/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 21s 44ms/step - accuracy: 0.5977 - loss: 1.0436 - val_accuracy: 0.3170 - val_loss: 2.6722
Epoch 10/10
334/334 ━━━━━━━━━━━━━━━━━━━━ 15s 45ms/step - accuracy: 0.6066 - loss: 1.0266 - val_accuracy: 0.3190 - val_loss: 2.5991
<keras.src.callbacks.history.History at 0x79071c903730>
```

This is not the entire model but rather a single step of the model. From this, we decided to not go ahead with the rest of the strategy as we felt accuracies were not good enough to produce.

2 - K Means

Understanding that we could not force direct digit classification, we then tried grouping images through k means and using that in a neural network. K of 10, being 1 per digit.

The idea was, while that this grouping would not classify what each digit is, it would group it to others like it. Then the NN model would be inputted which “kind” of digit this for each digit and use this to make predictions. It would not have to learn the visual information of each digit, but rather work on which “type” of digit this is. The K means would do the heavy lifting of grouping

and then the NN would learn patterns on much more simple data. One could consider it as a form of extreme dimensionality reduction.

Unfortunately, accuracies of around 15% showed us that this would not work.

From this moment we had to re-analyze possible approaches and do some research to see what tools/methods could be available to us.

3 - Model inside Model

We found out that it would be possible to have “models inside models”. From this, we realized we found the “missing piece” where now the even/odd classification would be linked to the digit recognition and we could learn as a whole, instead of in separate pieces.

Having submodels allows us to tailor the shape and design of these models to best suit the respective tasks, and them being included in a single larger model means the backpropagation would impact both models at the same time - thus they learn together and learn to work together to best suit our needs.

Refining our Model

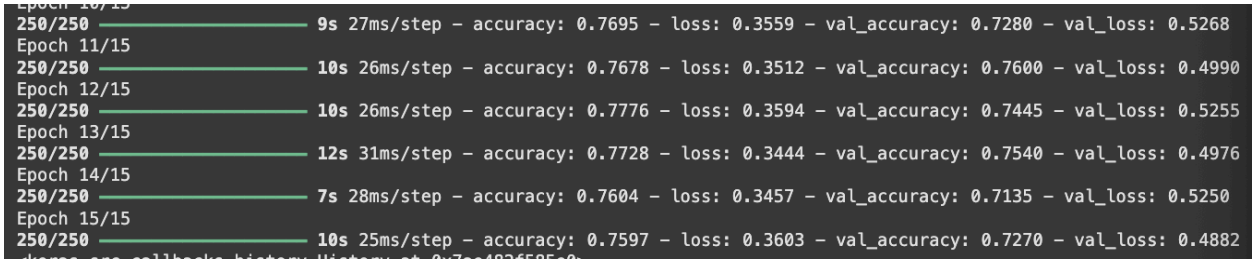
The first attempt on this configuration worked well and showed us that this was likely the way to go. We got accuracies of nearly 70% so we started refining this model, hyperparameters, and its inner configuration.

NN Hyperparameters used for the layers were taken from the in class example, and worked well. Since model 1 outputs a number between 0 and 1, the output layer was changed to be sigmoid.We noticed slight overfitting so added the standard dropoff of 0.2.

```
Epoch 10/15
250/250 ━━━━━━━━━━━━━━━━━━━━ 9s 38ms/step - accuracy: 0.7859 - loss: 0.3491 - val_accuracy: 0.7735 - val_loss: 0.5274
Epoch 11/15
250/250 ━━━━━━━━━━━━━━━━━━━━ 9s 35ms/step - accuracy: 0.8101 - loss: 0.3370 - val_accuracy: 0.7380 - val_loss: 0.5485
Epoch 12/15
250/250 ━━━━━━━━━━━━━━━━━━━━ 9s 32ms/step - accuracy: 0.7715 - loss: 0.3484 - val_accuracy: 0.7220 - val_loss: 0.5765
Epoch 13/15
250/250 ━━━━━━━━━━━━━━━━━━━━ 10s 39ms/step - accuracy: 0.7855 - loss: 0.3396 - val_accuracy: 0.7175 - val_loss: 0.5718
Epoch 14/15
250/250 ━━━━━━━━━━━━━━━━━━━━ 8s 34ms/step - accuracy: 0.7778 - loss: 0.3634 - val_accuracy: 0.7255 - val_loss: 0.6309
Epoch 15/15
250/250 ━━━━━━━━━━━━━━━━━━━━ 10s 34ms/step - accuracy: 0.7980 - loss: 0.3441 - val_accuracy: 0.7565 - val_loss: 0.5478
```

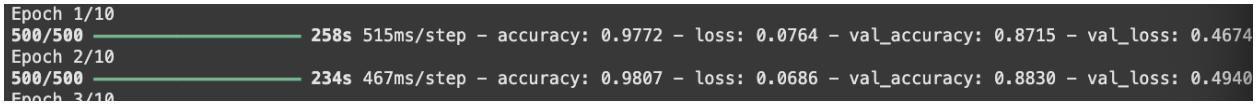
To start we were using three inner models but we quickly realized that we could use one single model for both the middle and bottom images since the models performed the same task. This way the digit recognition model gets trained on each digit (not only half the time, based on even/odd) which leads to much more training volume.

This did not lead to noticeable increases of accuracy, but the reduced size of the overall model and decreased complexity lead to us concluding that this was worth keeping.



CNN

Since we are dealing with image data, we realized that we could leverage the power of CNNs to obtain useful data that the regular portion of the NN could use to train. This brought our accuracies up to the 90s. This is where the “hybrid” portion comes in, we use CNN to find useful features of the images, which the regular NN uses to learn from.



You’ll notice Model1 has more simple NN layers - since it only needs to learn even/odd and not digit classification, it should be a more simple model.

Then we got to refining hyperparameters related to the CNN - filter size and dept, pooling, etc.

CNN Hyperparamters

Padding: Valid (no padding) since the edges of the images are usually useless, this helps reduce the impact of edges

You will notice model 1 has less filters than model 2, this is because model 1 only needs to know even/odd and not the digit itself. Therefore, it does not need to be as complex.

Accuracies were not impacted by making model 1 more so following the rule of more simple=better, we decided to keep is at this.

ChatGPT recommended to start with fewer filters and increase, as features get more abstract. As for the size, the logic is that to start we want to get a rougher idea of the image's features, so a larger size, and later on get finer details, so smaller kernel size. These were recommended by ChatGPT and through testing, proved to be good and reliable.

With just these, we were getting nearly 99% training accuracy but only about 89% validation accuracy - so I knew there was some performance left on the table. Increasing dropout did not help, so then I found out abou**t max pooling** which could help smooth out variations in data and therefore *should* help reduce overfitting. After testing, it did exactly this. Testing accuracy reduce slightly but was nearly 1:1 with testing accuracy - which is the metric that actually matters.

```
final_model.fit(ModelModelInput, ModelModelLabels, epochs=10, batch_size=8, validation_split=0.2)

Epoch 1/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 59s 54ms/step - accuracy: 0.2540 - loss: 1.9995 - val_accuracy: 0.6800 - val_loss: 0.8803
Epoch 2/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 80s 53ms/step - accuracy: 0.7095 - loss: 0.8333 - val_accuracy: 0.8665 - val_loss: 0.4095
Epoch 3/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 83s 54ms/step - accuracy: 0.8488 - loss: 0.4879 - val_accuracy: 0.8935 - val_loss: 0.3306
Epoch 4/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 82s 55ms/step - accuracy: 0.8746 - loss: 0.3988 - val_accuracy: 0.9215 - val_loss: 0.2502
Epoch 5/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 55s 55ms/step - accuracy: 0.8938 - loss: 0.3271 - val_accuracy: 0.9125 - val_loss: 0.2639
Epoch 6/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 54s 54ms/step - accuracy: 0.9089 - loss: 0.2968 - val_accuracy: 0.9215 - val_loss: 0.2505
Epoch 7/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 82s 55ms/step - accuracy: 0.9131 - loss: 0.2810 - val_accuracy: 0.9225 - val_loss: 0.2448
Epoch 8/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 55s 55ms/step - accuracy: 0.9163 - loss: 0.2590 - val_accuracy: 0.9360 - val_loss: 0.2249
Epoch 9/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 81s 54ms/step - accuracy: 0.9273 - loss: 0.2422 - val_accuracy: 0.9215 - val_loss: 0.2604
Epoch 10/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 82s 54ms/step - accuracy: 0.9316 - loss: 0.2260 - val_accuracy: 0.9175 - val_loss: 0.2839
<keras.src.callbacks.history.History at 0x7ae474db63b0>
```

Originally, data was handled using complicated tensors but after reviewing code I was skeptical of how this was being done and decided to restructure. I made it more simple, not using tensors with channels and directly indexing into the data. My assumptions were correct and this shot up our accuracies to up to 97%.

Pre/Post Processing

Pre-Processing: Normalizing the data to be [0,1] instead of [0,255]. This simplifies the data a **very significant** amount and simplifies learned for the NN.

Post-Processing: The model outputs a oneHot vector, so we simply turn this to a single digit. The reverse of oneHot, this is done in the classify function.

Conclusion

This assignment took A LOT of trial and error, and thinking of different strategies. The big eureka moment came when we realized that model inside model was possible (or this general idea) and then came time to tuning the model and optimizing it for our use case!

Reference/Tools Used

- **Lectures**
- **In Class NN Example**
- **ChatGPT** Used to figure out syntax issues, how to use tensorflow, and for general ideas/questions that helped us figure out tools available to us. For example, maxPooling to fix overfitting problems.