Inf 264
project 2
Andreas Valen

Summary
In this project I made an effort to find the best hyperparameters for training a digit recognizer. I look at the different effects of the different layers and their parameters. At the end I train a model and test and analyze the performance.

Disclaimer:
It should be noted that I made a crucial mistake when doing this project which forced me to redo all of the training. There was no way I had time to do the same training as last time as I only noticed the error right before the deadline. I opted to heavily reduce the number of epochs and ranges of values I compared to finish in time. This resulted in a marginally different model structure (i.e the “best drop rate went from 0.3 to 0.2”), but the approach was the exact same.

Initial approach
I started by looking at examples of different image recognition models, most notably I watched a neural network video series by “StatQuest with Josh Starmer” on YouTube. The majority of models used a variation of convolutional layers with max pooling layer, dropout layers and dense layers. I played around with different examples and discovered that training time would be one of the most limiting factors of this project.

Due to time constraints I decided to:
- Only test the keras library as it is well documented
- Only use “relu” as activation function as it seems to be the most popular
- Use hold-out validation instead of cross validation
- Use standard batch size of 32
- Only test ranges for hyperparameters within popular ranges
- Only test a few combinations of number/types of layers

Accuracy and validation
The goal is to find the beast values for the parameters by comparing models that have been trained with different parameters. Since the test set should only ever be checked once, I decided to 80 - 20 split the training data into training and validation data.

Accuracy is measured in percentages of correct predictions when predicting labels for unseen data. Note that the accuracy we are trying to improve here is the accuracy on the validation set. Many of the examples I came across tweaked the parameters based on how the model performed on the test set. That makes the final accuracy test unfair as the models who perform better on the test set have been favored. As a way to visualize the performance

I used confusion matrices throughout the project to get an overview of the most common errors the classifiers made.

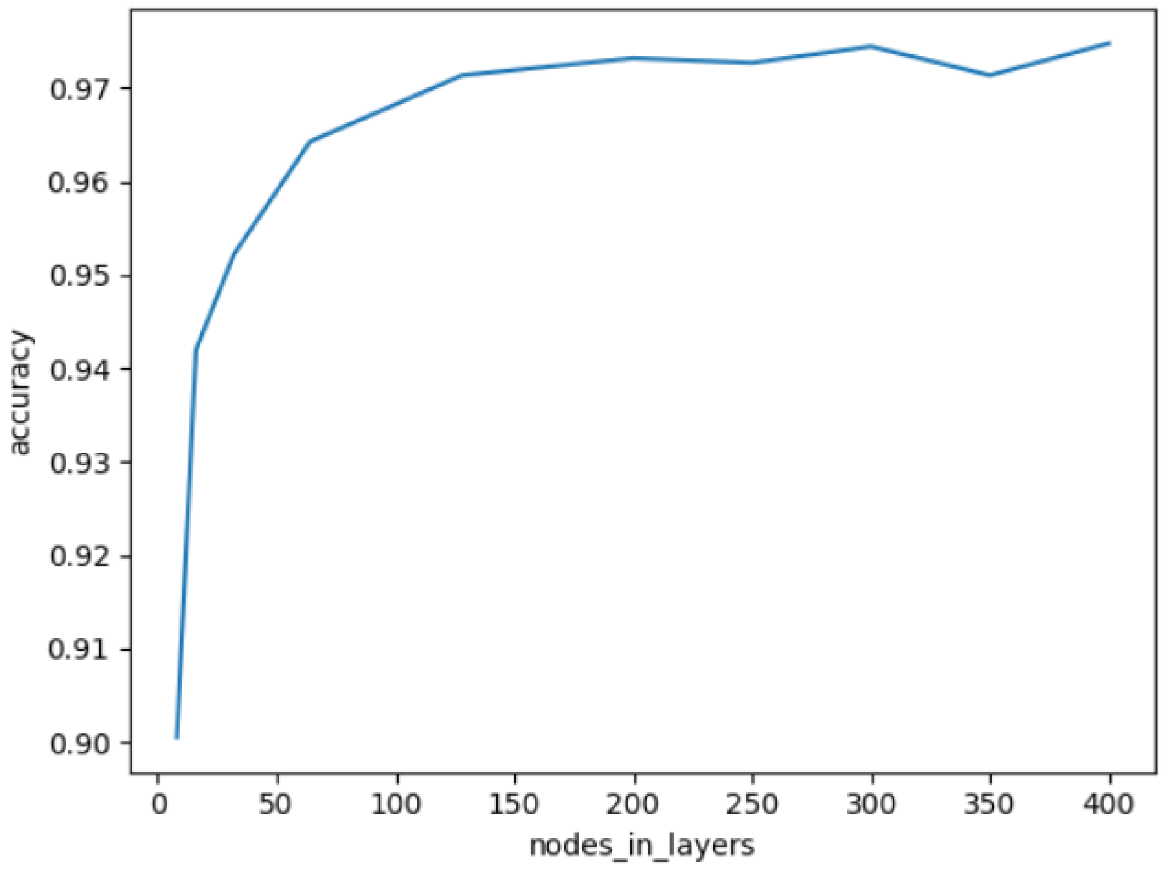
Comparing hyper parameters

I initially began by creating similar models with small differences to roughly test the performance of the layers. The first models I created only looked at the keras library's "dense" layer, and the difference in accuracy between number of nodes and number of layers. This became tedious and not very scalable, so I opted to make a function that created models based on given parameters and compared them.

Neurons

I started by comparing the accuracy of similar models where the only difference was the number of neurons in the dense layer. The models had two hidden layers with 8, 16, 32, 64, 128, 200, 250, 300, 350 and 400 neurons each.

```
acc: 0.975, loss: 0.093, npl: 400, extra_layers: 0,
acc: 0.974, loss: 0.088, npl: 300, extra_layers: 0,
acc: 0.973, loss: 0.091, npl: 200, extra_layers: 0,
acc: 0.973, loss: 0.094, npl: 250, extra_layers: 0,
acc: 0.971, loss: 0.095, npl: 128, extra_layers: 0,
acc: 0.971, loss: 0.098, npl: 350, extra_layers: 0,
acc: 0.964, loss: 0.115, npl: 64, extra_layers: 0,
acc: 0.952, loss: 0.156, npl: 32, extra_layers: 0,
acc: 0.942, loss: 0.208, npl: 16, extra_layers: 0,
acc: 0.900, loss: 0.332, npl: 8, extra_layers: 0,
```

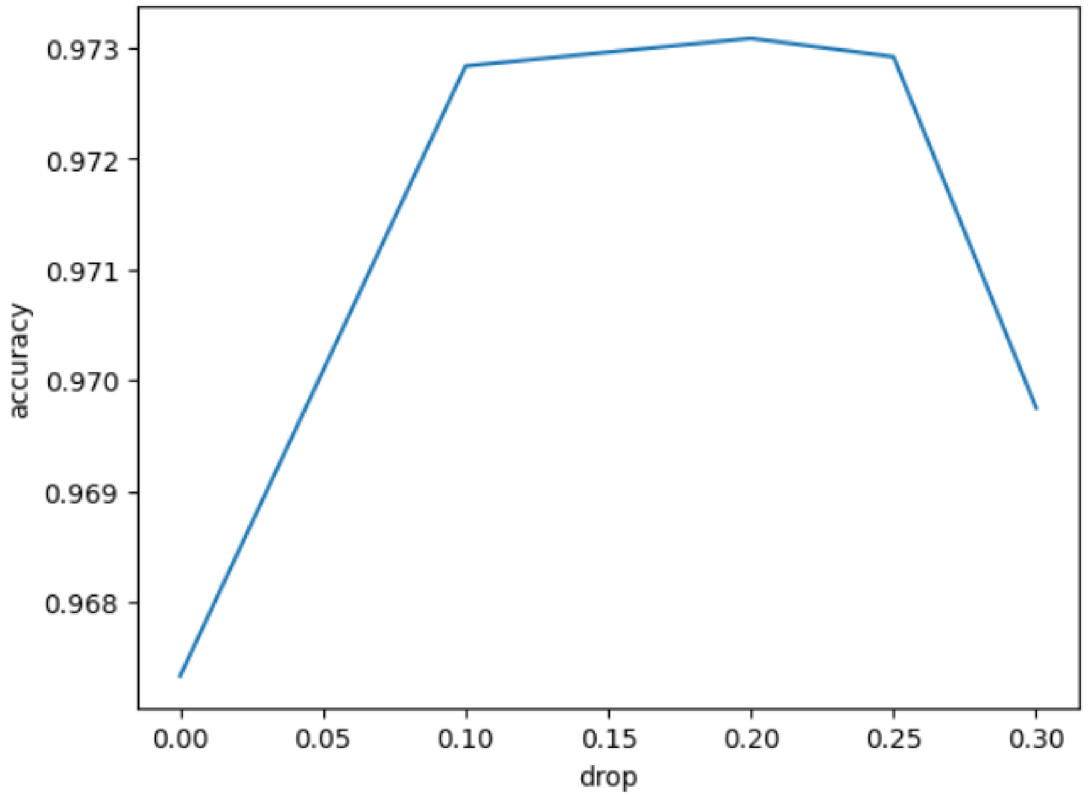


It seems accuracy increased alongside the number of neurons in each layer, however, there is a sharp decline in improvement after the neurons in each layer exceeds 128. Factoring in training time I decided to go with 128 neurons per layer.

Dropout

I then tested the best performing model with dropout layers. I tested models with an increasing drop ratio from 0 to 0.6. I only used 5 epochs in the test, so the start accuracy doesn't match the previous model.

```
acc: 0.973, loss: 0.093, npl: 128, drop: 0.2, extra_layers: 0.0, weight_constraint: 0,
acc: 0.973, loss: 0.094, npl: 128, drop: 0.25, extra_layers: 0.0, weight_constraint: 0,
acc: 0.973, loss: 0.095, npl: 128, drop: 0.1, extra_layers: 0.0, weight_constraint: 0,
acc: 0.970, loss: 0.098, npl: 128, drop: 0.3, extra_layers: 0.0, weight_constraint: 0,
acc: 0.967, loss: 0.107, npl: 128, drop: 0.0, extra_layers: 0.0, weight_constraint: 0,
```

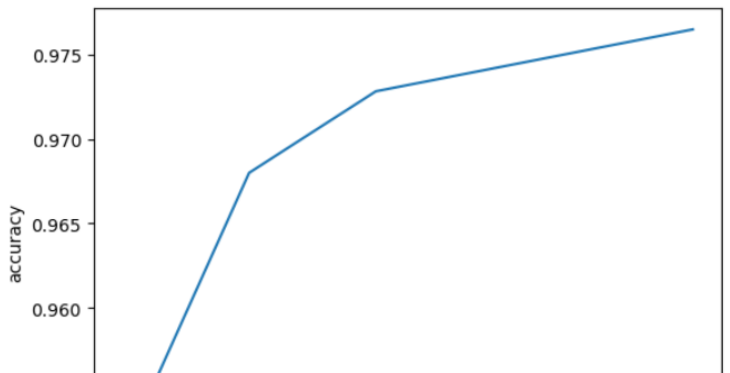


Accuracy seemed to peak neatly around the 0.2 mark so I used 0.2 as dropout rate going forward.

Epochs

The epochs variable denotes how many times the training algorithm performs a forward pass and backpropagation on each training data point.

```
acc: 0.976, loss: 0.098, time: 29.03, npl: 128, epochs: 10, drop: 0.2, extra_layers
acc: 0.973, loss: 0.095, time: 14.19, npl: 128, epochs: 5, drop: 0.2, extra_layers:
acc: 0.968, loss: 0.108, time: 9.54, npl: 128, epochs: 3, drop: 0.2, extra_layers:
acc: 0.951, loss: 0.169, time: 3.60, npl: 128, epochs: 1, drop: 0.2, extra_layers:
```



It again seems the accuracy increases with each epoch. Looking at the graph though, it looks like using 5 epochs is a good middle ground for testing. When training the final model, I will use more.

Adding a convolutional layer

I tested replacing a dense layer with a convolutional layer.

```
acc: 0.979, loss: 0.079, time: 197.14, npl: 128, drop: 0.2, extra_layers: 1.0, weight_constraint: 0, convolutional layer: yes,
acc: 0.969, loss: 0.099, time: 15.12, npl: 128, drop: 0.2, extra_layers: 0.0, weight_constraint: 0, convolutional layer: no,
```

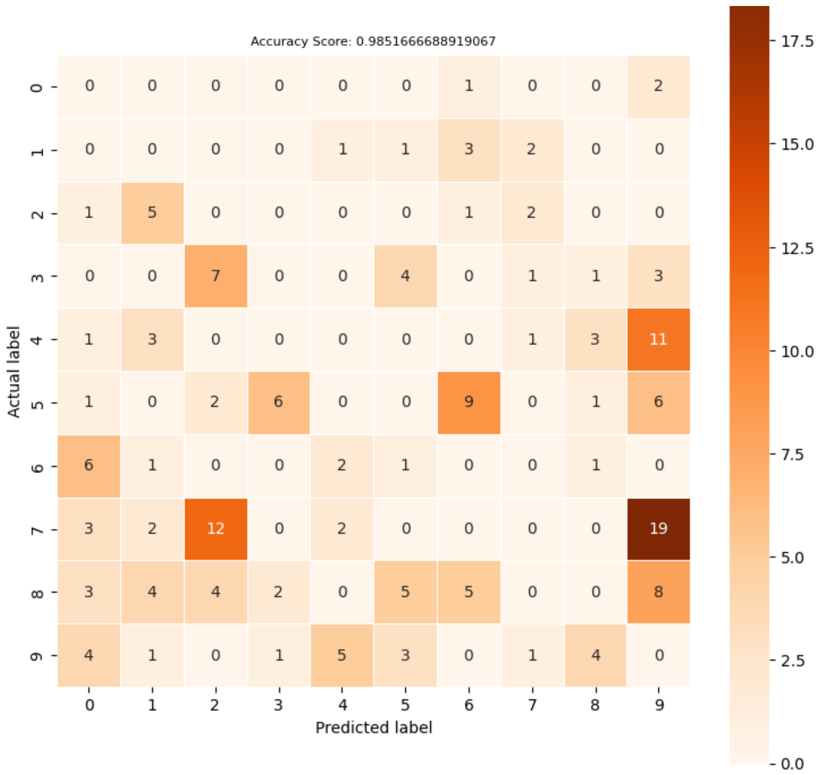
Looking at the data we see accuracy improvement is absolutely fantastic, plus 1 percent, with the convolutional layer addition, but it comes at a price. Training time is now a nightmare.

I tried adding an additional convolution layer:

```
print(loss, accuracy)
```

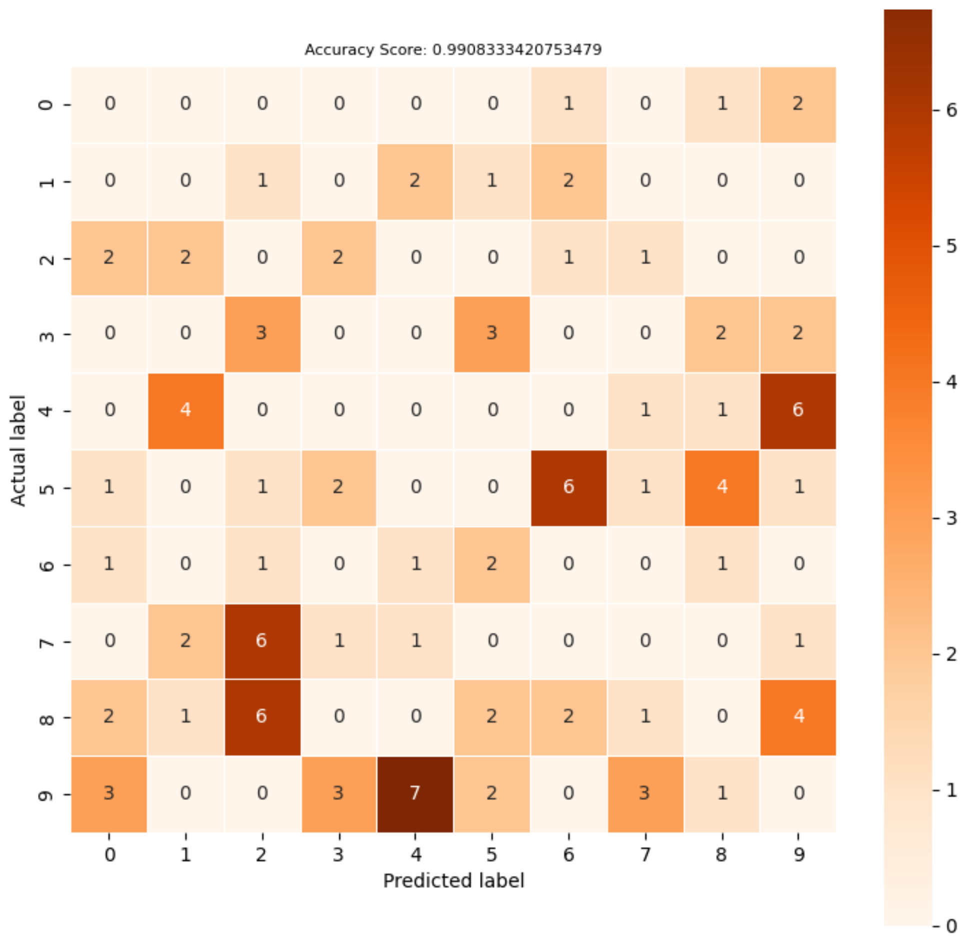
```
0.058848362416028976 0.9851666688919067
```

Accuracy is now as high as it has ever been. Here is the confusion matrix for the model:



This is a modified confusion matrix because I have removed the correct predictions to better highlight the mistakes. We clearly see the model often mistakes a 7 for a 9.

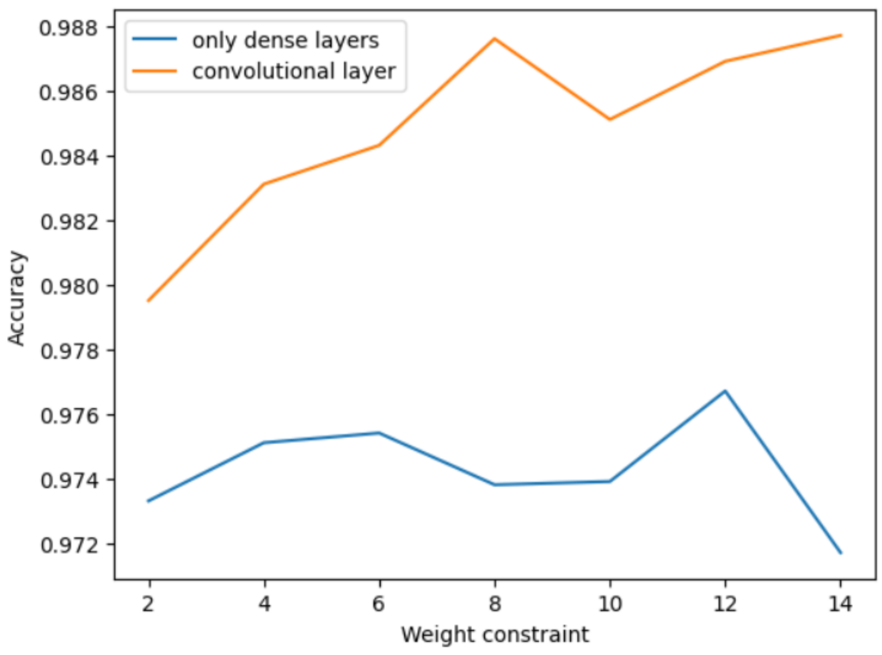
Let's try adding max pooling layers and add another epoch.



	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	1	0	1	2
1	0	0	1	0	2	1	2	0	0	0
2	2	2	0	2	0	0	1	1	0	0
3	0	0	3	0	0	3	0	0	2	2
4	0	4	0	0	0	0	0	1	1	6
5	1	0	1	2	0	0	6	1	4	1
6	1	0	1	0	1	2	0	0	1	0
7	0	2	6	1	1	0	0	0	0	1
8	2	1	6	0	0	2	2	1	0	4
9	3	0	0	3	7	2	0	3	1	0

We officially breached the 90% barrier. These are the layers and parameters I will use for my final model.

Weight constraints (result gotten using wrong testing data, but still intresting)
I went back a few steps and tried adding weight constraints. It took over an hour to test and it looks like the less constraints, the more accurate my models become. This might be due to the placement of the constraints and the number of layers with constraints, but testing all possible combinations is not feasible due to time constraints.
Graph of accuracy of the models with respect to weight_constraints:



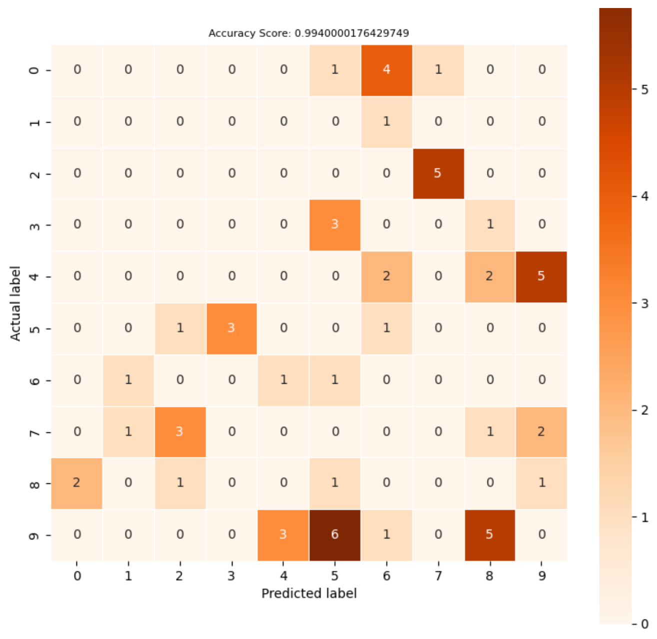
Additional parameters
There are a number of other hyperparameters that might improve accuracy. Notably batch size, activation function, loss function and optimizing function. I have decided to not test them as each new model now takes 20 minutes to train. The increase in accuracy would also be so incremental that cross validation would be in order. With k=4 that gives approximately 1 hour of train time per model. Testing between ‘relu’ and ‘leaky_relu’ would then be at least a multiple hour ordeal.

Training the final model

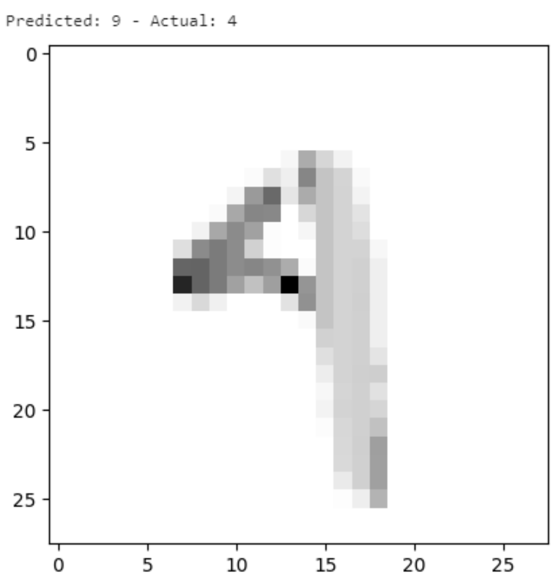
Based on what I found I tested these values:

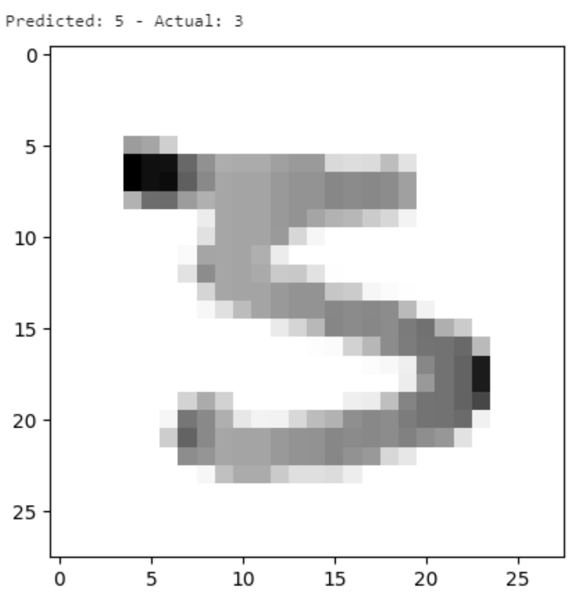
- nodes=128
- drop=0.2
- epochs=10
- extra convolutional layer
- max pooling layers

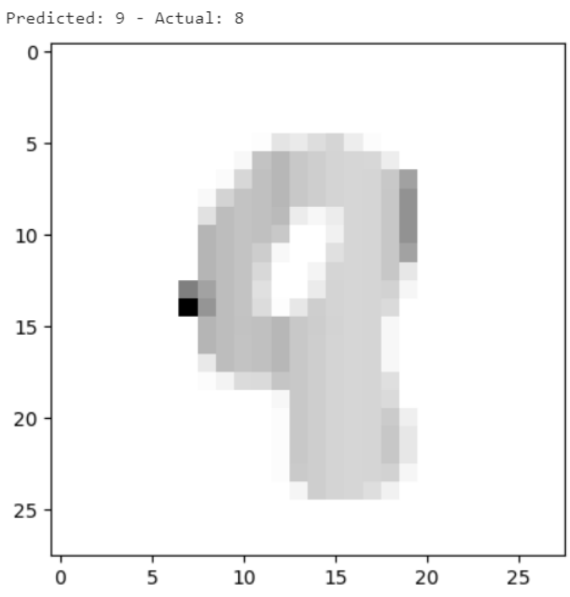
The resulting classifier had an accuracy of 99.40% when predicting the labels of the unseen test data. I am thrilled with that result as I predicted it to hit in the mid 98% range

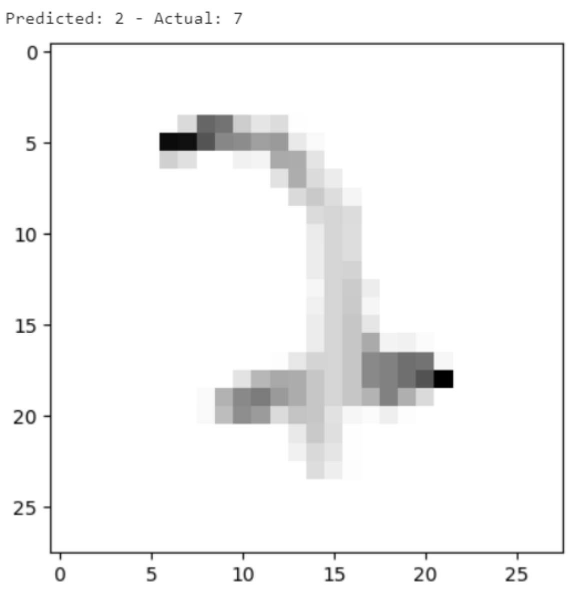


Evaluation
99.40% is fantastic, but it doesn’t tell us where it goes wrong. I wrote a function that finds the errors and prints them out. Let’s look at a few examples of where the classifier fails:









Run block 2 in the ‘final_v2’ file and it prints all of the numbers it got wrong. After looking at the images printed out it’s clear that the performance of the classifier is sufficient. Appart from a couple few examples, it only messes up on numbers that even humans would struggle with. I for one would, along with the classifier, say the bottom left picture above is a “nine” and not an “eight”.

Documentation / Code showcase

plotCorrelationGraph

```python
def plotCorrelationGraph(modelDataList, x="nodes_in_layers", y="accuracy"):
    x_arr = []
    y_arr = []
    for modelData in modelDataList:
        x_arr.append(modelData[x])
        y_arr.append(modelData[y])
    plt.ylabel(y)
    plt.xlabel(x)
    plt.plot(x_arr, y_arr)
    plt.show()
```

The parameter value had to be extracted from the resulting list of dictionaries. I made this function to quickly plot the correlation between two parameters. Note that the modelDataList has to be sorted based on x value before being passed to the plotting function.

cleanPredictions

```python
def cleanPredictions(predictions):
    _predictions = []
    for p in predictions:
        _predictions.append(np.argmax(p))
    return _predictions
```

The model.predict function returns a list of arrays of the last layer. Argmax finds the label that the classifier deemed most likely to be correct. This function turns the 2d array into a list of the actual predicted labels.

```python
def trainAndEvaluateModel(x_train, y_train, x_test, y_test, use_cnn=0,
                          nodes_in_layers=32,
                          epochs=10,
                          drop=0,
                          add_dense_layer=0,
                          weight_constraint=0):
    model = tf.keras.models.Sequential()
    if use_cnn==1:
        model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu',
                                         input_shape=(28, 28, 1)))
        model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
        model.add(tf.keras.layers.Dropout(drop, seed=3))
        model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
    else:
        model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
        model.add(tf.keras.layers.Dense(units=nodes_in_layers, activation=tf.nn.relu,))
        model.add(tf.keras.layers.Dropout(drop, seed=1))
    model.add(tf.keras.layers.Dense(units=nodes_in_layers, activation=tf.nn.relu,
                                    kernel_constraint= tf.keras.constraints.max_norm(weight_constraint)))
    if add_dense_layer==1:
        model.add(tf.keras.layers.Dense(units=nodes_in_layers, activation=tf.nn.relu))
    model.add(tf.keras.layers.Dropout(drop, seed=2))
    model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    start = time.time()

    model.fit(x_train, y_train, epochs=epochs, verbose=1)
    end = time.time()
    elapsed_time = end - start
```

This is the function I made to test the different hyper parameters. I made a mapping function and fed it an array of different combinations to test which it then trained and tested. This made it easy to compare different results.
Example of testing array and printing the wanted values:

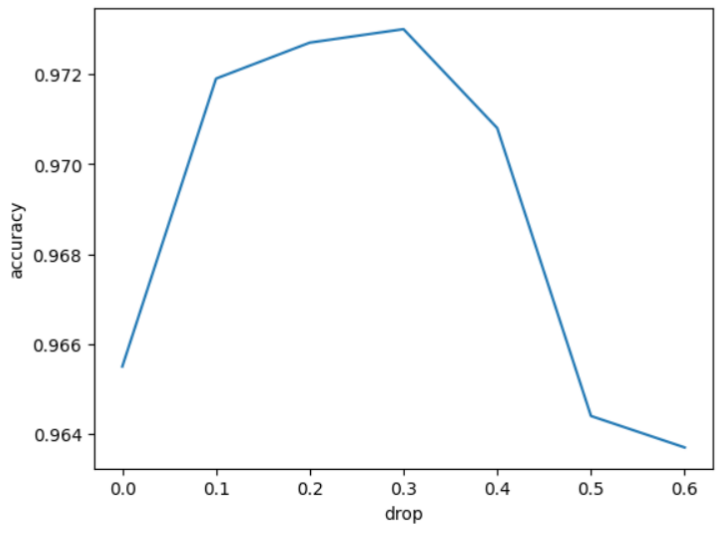
```
####################################
## This block is testing dropout ##
####################################
nodes=128
epochs=5

#[use_cnn = 0 or 1, nodes_in_layers, epochs, drop = float between 0 and 1, add_dense_layer = 0 or 1]
d_parameters_to_test = np.array([[0, nodes, epochs, 0, 0],
                                [0, nodes, epochs, 0.1, 0],
                                [0, nodes, epochs, 0.2, 0],
                                [0, nodes, epochs, 0.3, 0],
                                [0, nodes, epochs, 0.4, 0],
                                [0, nodes, epochs, 0.5, 0],
                                [0, nodes, epochs, 0.6, 0],
                               ])
#Testing the significance of the number of neurons and if an extra layer adds to the accuracy
d_tested_models = mapThen_trainAndEvaluateModel(x_train, y_train, x_test, y_test, d_parameters_to_test)

#Printing models from best to worst
for modelData in d_tested_models:
    printChosenParams(modelData, use_cnn=False, epochs=False, time=False)
```

Result (removed the keras printout):

```
acc: 0.973, loss: 0.091, npl: 128, drop: 0.3, extra_layers: 0.0,
acc: 0.973, loss: 0.091, npl: 128, drop: 0.2, extra_layers: 0.0,
acc: 0.972, loss: 0.092, npl: 128, drop: 0.1, extra_layers: 0.0,
acc: 0.971, loss: 0.101, npl: 128, drop: 0.4, extra_layers: 0.0,
acc: 0.965, loss: 0.115, npl: 128, drop: 0.0, extra_layers: 0.0,
acc: 0.964, loss: 0.113, npl: 128, drop: 0.5, extra_layers: 0.0,
acc: 0.964, loss: 0.125, npl: 128, drop: 0.6, extra_layers: 0.0,
```

Sorting and printing out the result:

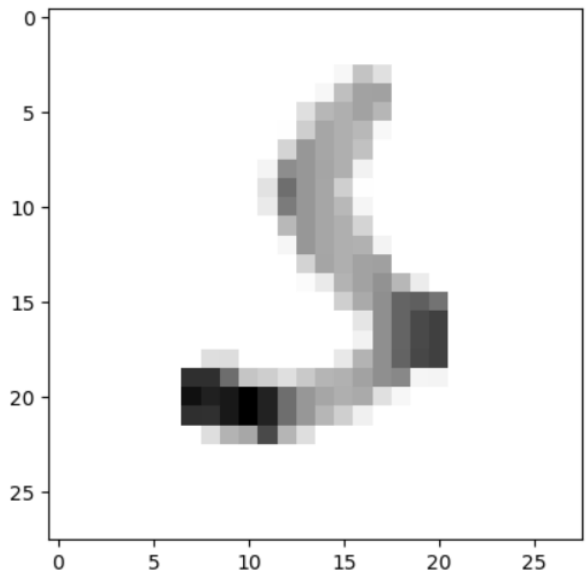
```
[106]: sortedList = sorted(d_tested_models, key=lambda d: d['drop'], reverse=False)
       plotCorrelationGraph(sortedList, x="drop", y="accuracy")
```



Locating the wrongly predicted images:

```python
predictions = cleanPredictions(model.predict(x_test))

for i in range(0, len(y_test)-1):
    if predictions[i]!=y_test[i]:
        print("Predicted: {} - Actual: {}".format(predictions[i], y_test[i]))
        plt.imshow(x_test[i], cmap='Greys')
        plt.show()
```

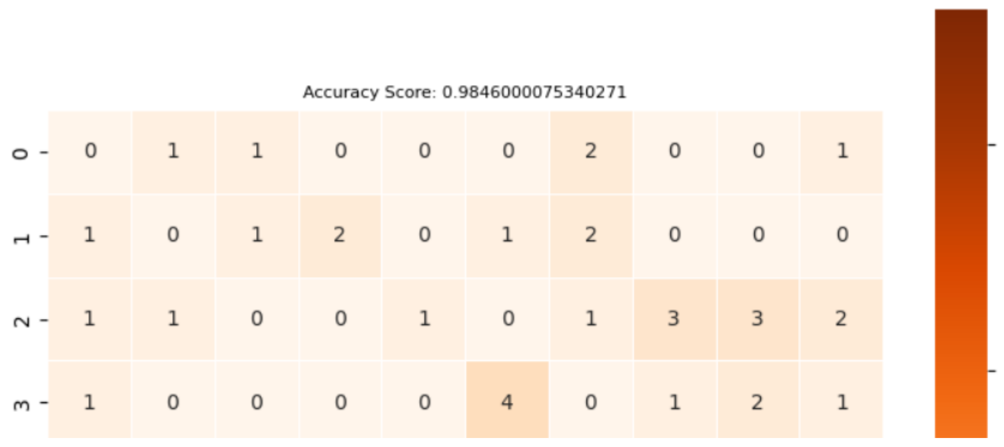
```
313/313 [==============================] - 3s 10ms/step
Predicted: 3 - Actual: 5
```



This is how I located the images of the numbers that failed.

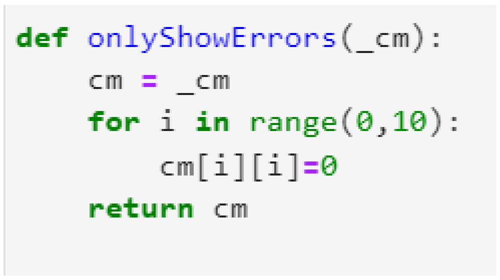
Here is the code i used to print out confusion matrices:

```python
_modelData = c_tested_models[0]

predictions = _modelData['predictions']
cm = metrics.confusion_matrix(y_test, predictions)
error_cm = onlyShowErrors(cm)

plt.figure(figsize=(9,9))
sns.heatmap(error_cm, annot=True, fmt=".0f", linewidths=.5, square = True, cmap = 'Oranges');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(_modelData['accuracy'])
plt.title(all_sample_title, size = 8);
```



0	0	1	1	0	0	0	2	0	0	1
1	1	0	1	2	0	1	2	0	0	0
2	1	1	0	0	1	0	1	3	3	2
3	1	0	0	0	0	4	0	1	2	1

Note that I made a function that removes the correct answers to easier see where the errors are.



```python
def onlyShowErrors(_cm):
    cm = _cm
    for i in range(0,10):
        cm[i][i]=0
    return cm
```