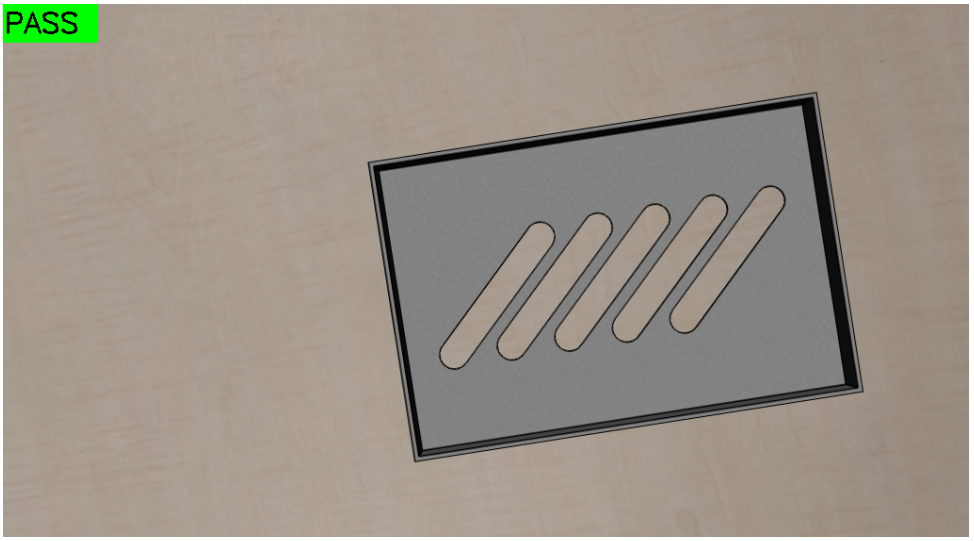


Machine Learning

Using machine learning algorithm to make an image classification model

DUE TO THE INADEQUACY OF STREAM'S SUBMISSION SERVICE, THE CODE FOR THIS PROJECT IS ONLY AVAILABLE ON GITHUB

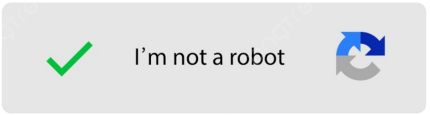
282772 Industrial Systems Design and Integration

Assessment 2: "Machine Learning Project"

Jamie Churchouse - 20007137

1. Contents

This report utilises hyperlinks frequently (even in these images)
and is *not* intended to be printed.



Despite this being an "*Artificial intelligence*" project,
it was completely human made!

The code for this project is ONLY available on GitHub.

2. Introduction

The massey engineering workshop is mass manufacturing CNC milled components. Due to the vast quantity of parts being made, the quality control needs to be done utilising an image classification Convolutional Neural Network.

The parts need to be categorised as either “Pass” or “Fail” with respect to whether the part meets quality control standards. To aid in the creation of the model, the workshop staff have provided 600 images to train the model - 200 of components that have passed, 200 of a class A failure, and 200 of a class B failure.

The model I have created uses the Keras API for the TensorFlow deep learning package for Python. It categorises images of parts from import to export in under a tenth of a second, and with a 100% efficacy.

3. Methodology

The machine learning package that is recommended everywhere is the Keras API for the TensorFlow deep learning package. This package is available for C++ which was tempting, but all the examples and reference material uses Python so that was the option I went with. As for IDE, Visual Studio Code is the Python IDE I am most familiar with so VS Code is what I used.

The workflow of this project is in three stages:
1. Research and familiarisation with the tools to make an initial prototype
2. Experimentation with different values for certain parameters
3. Optimisation of the original code with the learnings of the experimentation

3.1 Initial Prototype

The example code I found is from Nicholas Renotte on YouTube in his video *Build a Deep CNN Image Classifier with ANY Images*. Nicholas provided valuable instruction on how to start with Keras, and which features are most important for an image classifier as required for this project. Using his example code as a guide, I adapted functions to operate with my data and to output the results we require.

With images segregated into directories by category, the program imports all the data into arrays, randomises the order of the data and their respective labels, and then segregates the data into training, validation and testing groups. The model itself is predominantly 2D convolutional layers, with a Dense layer at the end. The model is compiled and trained on the training data set, and validated on the validation data set. Once the training is complete, the model is tested on the testing dataset. After some trial and error, I managed to get a model that did a decent job of classifying all the images.

With a baseline established, I now had to figure out how to enhance the algorithm.

3.2 Parameter Testing

The four parameters I wanted to test to see how the model responds were the number of layers in the model, the input size of the image, the number of neurons in the dense layers, and the number epochs the model is trained for. During the process of constructing this test platform, I found the videos from sentdex on TensorFlow and Keras to be of great help.

The images are processed into training data with image sizes of 32^2, 64^2 & 128^2 pixels. Models are created with 2, 3 or 4 dense layers with 16, 32 or 64 neurons. And all of these models run for 10, 15 or 20 epochs.

```
test_layers     = [2, 3, 4]
test_imsize     = [32, 64, 128]
test_neuros     = [16, 32, 64]
test_epochs     = [10, 15, 20]
```

```python
for L in test_layers:
    for I in test_imsize:
        for N in test_neuros:
            for E in test_epochs:

                name = GenName (L,I,N,E)
                params ["layers"] = L
                params ["imsize"] = I
                params ["neuros"] = N
                params ["epochs"] = E
                params [ "name" ] = name

                print(f"\nNow processing: {name}\n")

                model_names.append(Train.BuildModel(params))

                print(f"\nCompleted processing: {name}\n")
```

Figure 3.2A - Function that loops over each value of each parameter

```
model = Sequential()

# First layer
shape = (param_imsize,param_imsize,1)
model.add(Conv2D(16, (3,3), 1, activation='relu', input
model.add(MaxPooling2D())

model.add(Conv2D(16, (3,3), 1, activation='relu'))
model.add(MaxPooling2D())

# Intermediate layer(s) - decided by the input params
for i in range(1,param_layers):
    model.add(Dense(param_neuros, activation='relu'))

# Final layer
model.add(Flatten())
model.add(Dense(param_neuros*2, activation='relu'))

# Output layer
model.add(Dense(num_categs,activation="softmax"))

# Build
model.compile(
    loss = "sparse_categorical_crossentropy",
    optimizer = "adam",
    metrics = ["accuracy"]
    )
```

Figure 3.2B - Generic model used for the parameter testing

3.3 Final Prototype

With the result from stage two (discussed in section 4), it was now time to return to the original program and upgrade it with the features established to be beneficial.

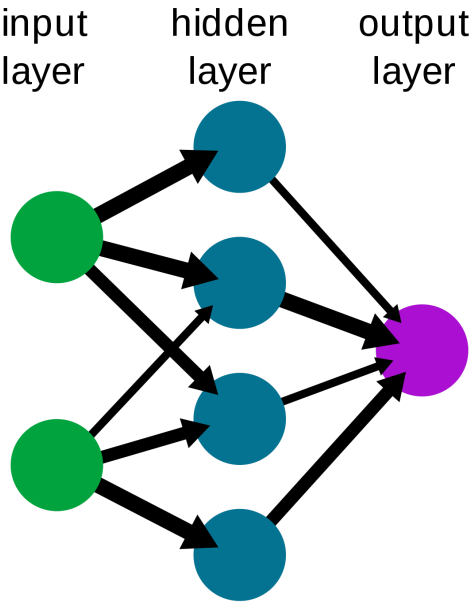
```
model = Sequential()

model.add(Conv2D(16, (3,3), 1, activation='relu', input_shape=(256,256,3)))
model.add(MaxPooling2D())

model.add(Conv2D(32, (3,3), 1, activation='relu'))
model.add(MaxPooling2D())

model.add(Conv2D(16, (3,3), 1, activation='relu'))
model.add(MaxPooling2D())

model.add(Flatten())
model.add(Dense(256, activation='relu'))

model.add(Dense(1, activation='sigmoid'))
model.compile('adam', loss=tf.losses.BinaryCrossentropy(), metrics=['accuracy'])
```

Figure 3.3 - Finalised classification model

3.4 How Neural Networks Work

A Neural Network (NN) is essentially a compilation of matrices. NNs have an input layer where the test subject is entered, a hidden layer or few or several, and then an output layer at the end which can be mapped into a category or categories. The two main layers used in this project are 2D Convolution layers and Dense layers. Neural Networks are often called Convolutional Neural Networks if several of the layers are convolutions.

A simple neural network



Convolutional layers take their input and convolve it with a given kernel. The size, step and cell weightings of the kernel can be adjusted by the creator.

Dense layers are a series of nodes or neurons. These neurons have “links" to previous and subsequent neurons. The links have weightings and the neurons have activation functions. When a neuron is processed, it takes the activation of the previous layer’s neurons, applies the link weightings and then processes its own activation function.

All of these layers combine along with support functions such as matrix shape changes to give a probability distribution at the output layer. For an example, the model may return [0.15, 0.91, 0.21] which when mapped to user friendly terms [“Pass”,”Fail_A”,”Fail_B”] means that this particular example is predicted to be “Fail_A” as it has the highest probability.

During the training of a model, an epoch is a stage of learning. For instance if a NN is trained with 10 epochs, the NN would take a sample, train on it, evaluate, and repeat that process for a total of 10 runs. Usually NNs on this scale should be optimal around 10-20 epochs.

4. Results & Discussion

After running all 4^3 (64) tests - which took 19 minutes - we have a folder full of image classification models, along with their analytics. For ease of use, I have created a function that *automatically* launches the TensorBoard service and opens the interface in the system browser. TensorBoard offers a lot of analytics to help decide which parts of a model to change, but for the purposes of this exercise, we're only going to look at the accuracy and loss of the models with respect to epoch.

epoch_accuracy
The accuracy of the model with 0.00 being it got every prediction wrong, to 1.00 where it got every prediction correct. A good model will have an accuracy of 1.00 (100% correct).

epoch_loss
The loss of the model is essentially the penalty the model has attained for a particular epoch. This should trend down, and a good model will have a loss of 0.

All the graphs shown in this report are from the TensorBoard service, and we're using a smoothing factor of 20%.

Let's have a look at our results…

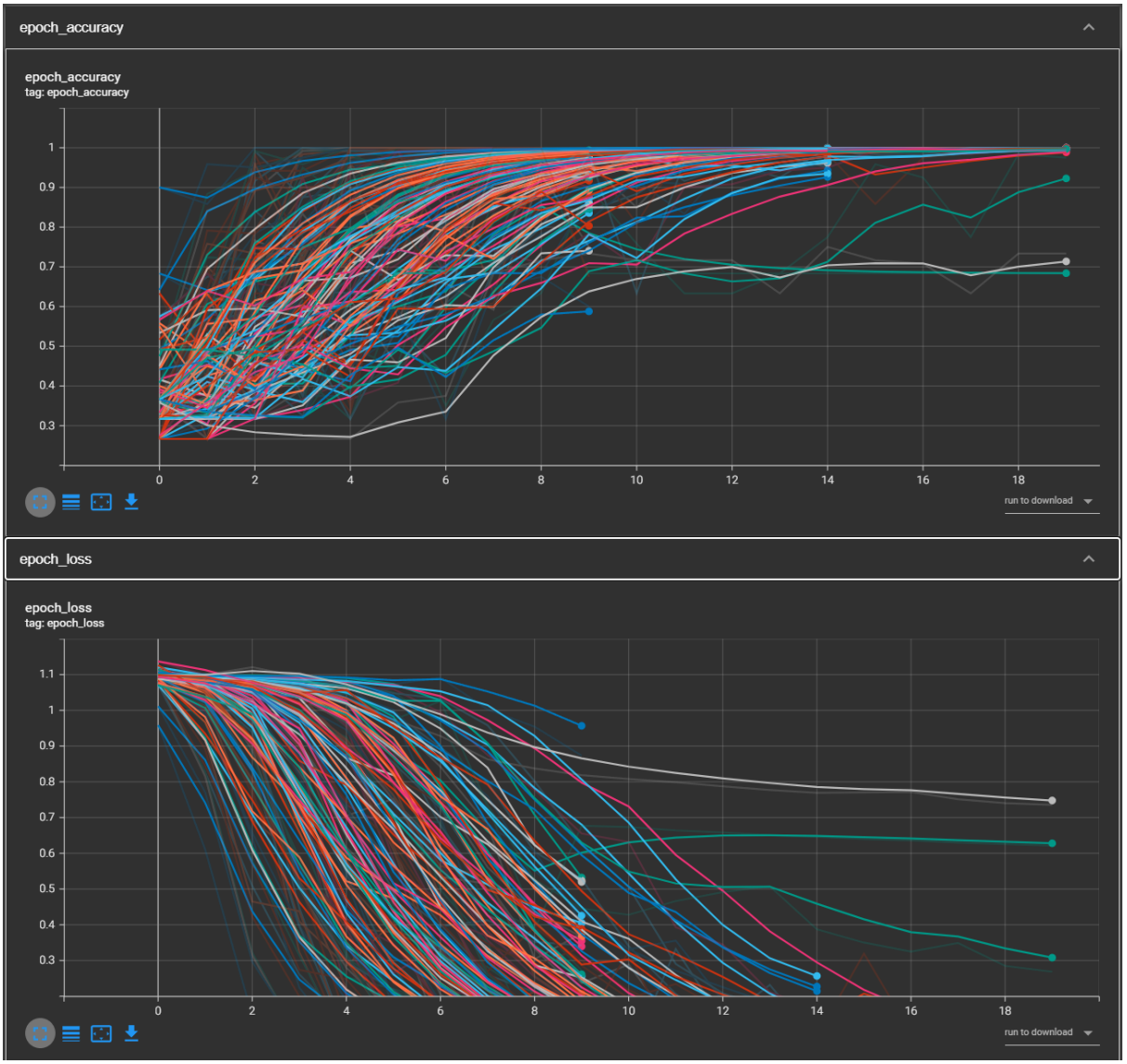


Figure 4.0 - All models comparison graph

Okaaay…
Ah, well, from looking at this graph, we can conclude that - without a doubt - having all the models displayed at the same time may be aesthetically appealing, but not very insightful. We're going to need to break it down by test parameter.

4.1 Layers

My initial thought was that because the model is fairly small, that the more layers the model had, the more resources the model had to learn quicker.

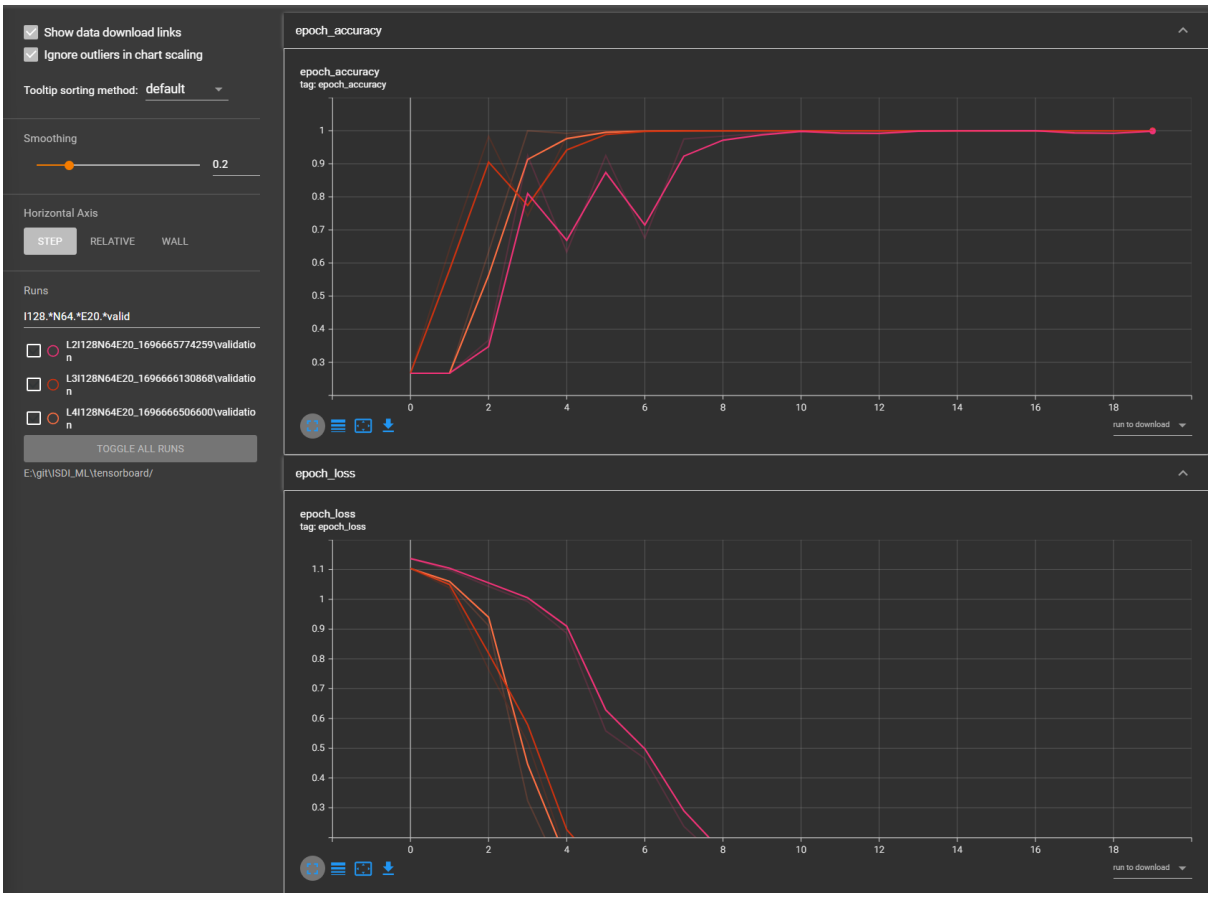


Figure 4.1 - Number of layers comparison graph

Pink:	Low	(2 layers)
Red:	Mid	(3 layers)
Orange:	High	(4 layers)

The high model learns the quickest, but the mid model is not too far behind. The low model, however, learns much slower but does reach 100% accuracy in about 10 epochs. This shows that the number of layers is an important factor, but changing it may not have such a huge effect as other parameters.

While my prediction was not far off, I had anticipated the number of layers would have a stronger effect than it really did. The optimal number of layers for a model with good speed/accuracy would be a middle value.

4.2 Image Size

My prediction for the best image size to use was that smaller images may lose too much data, but large images would be too slow therefore a moderate image size would be optimal.

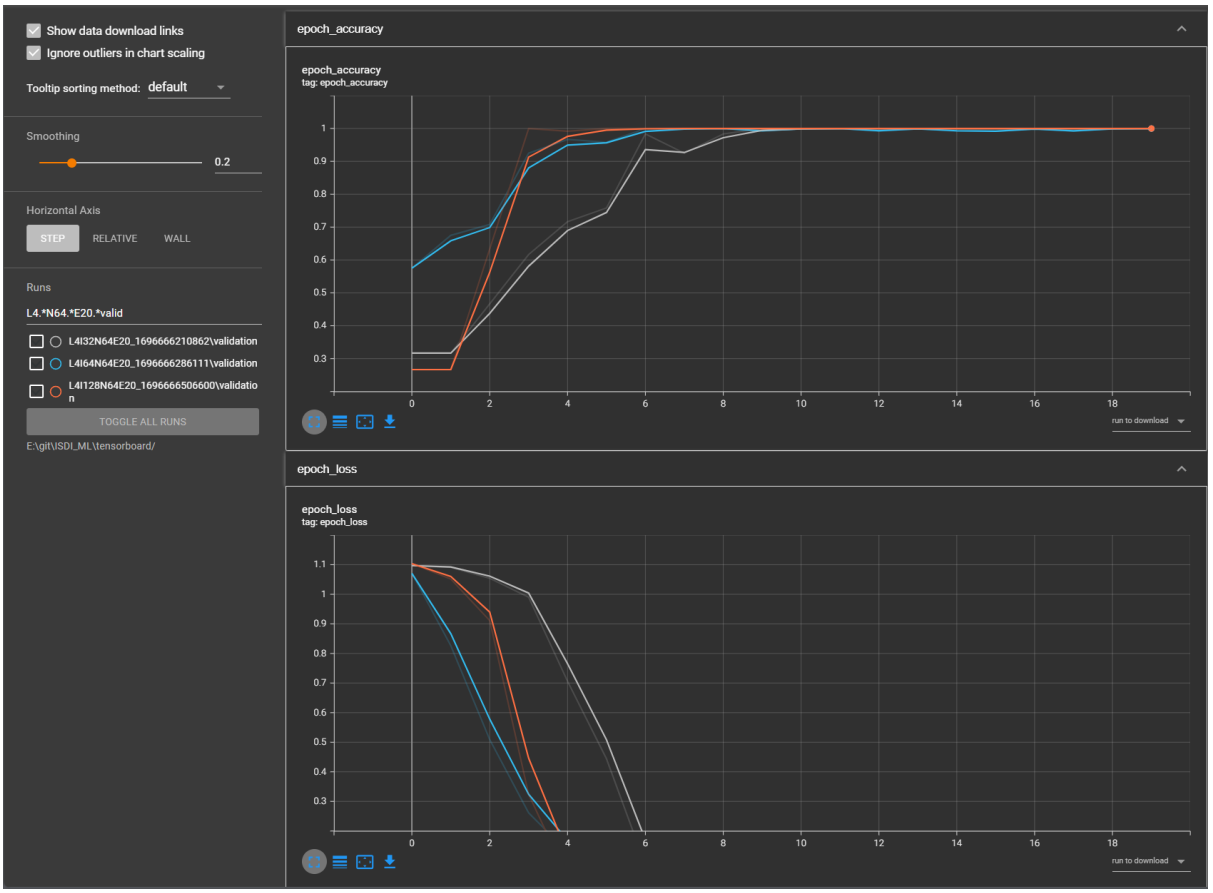


Figure 4.2 - Image size comparison graph

Grey:	low	(32² pixels)
Blue:	mid	(64² pixels)
Orange:	high	(128² pixels)

The differences between the models are not too dissimilar. The higher models are quicker to learn, but not by much - even the low model is in the 90% accuracy range by epoch 6. This shows that while image size does have an impact, it's not a large one, and the time saved by processing a smaller image may well be more advantageous than processing a larger image in only a couple of fewer epochs.

Again, my prediction for this specific application was a little off, but the justification is fairly correct. The optimal input image size for a model with good speed/accuracy would be a low value, but may be higher for more complex classifications.

4.3 Neurons

Again, my prediction was that the more neurons the better. Having more processing available inside the model should make the model more efficient.

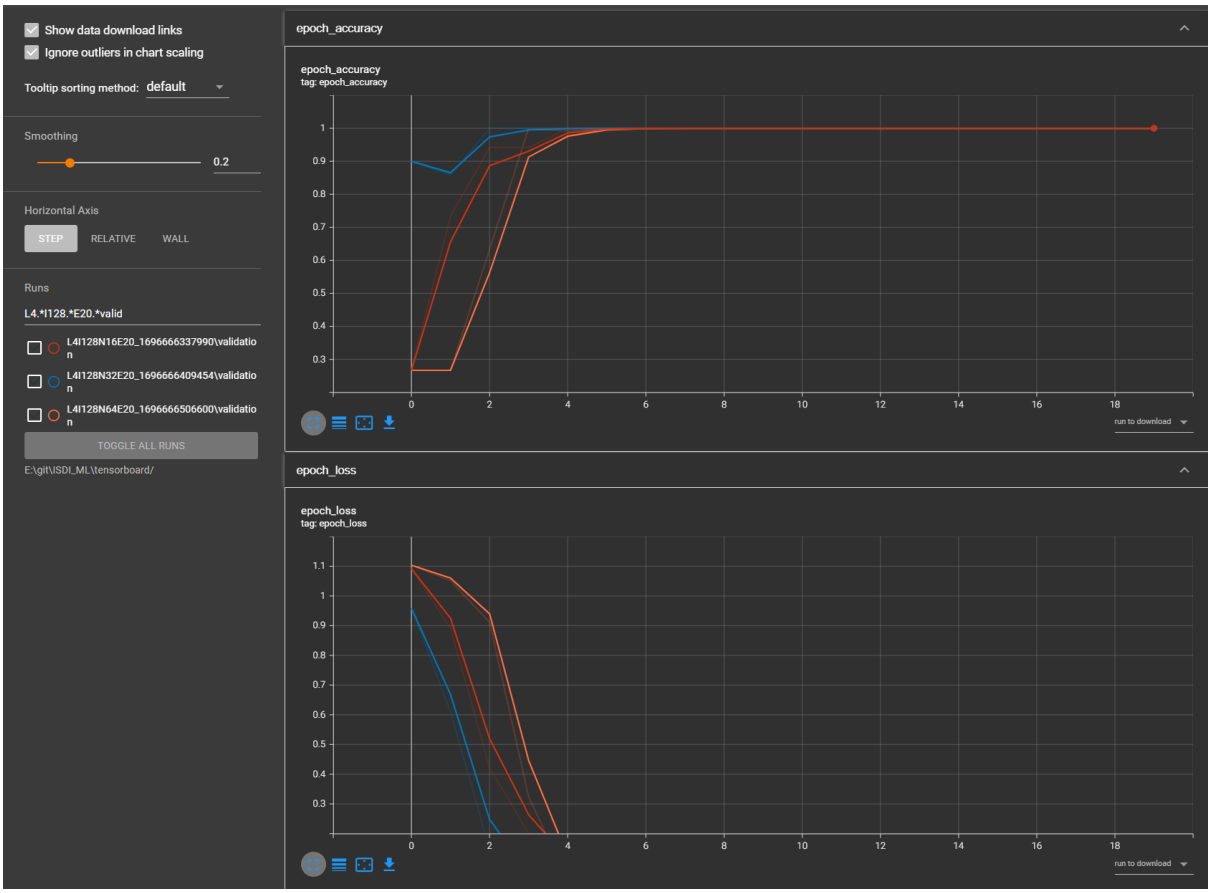


Figure 4.3 - Neuron count comparison graph

Red:	low	(16 neurons)
Blue:	mid	(32 neurons)
Orange:	high	(64 neurons)

All these models learn at a rate very close to each other. This means that more neurons only adds complexity to the calculations rather than making the model more effective. With this said, more complex images have more features to process and too low of a neuron count may not be able to return a valid result at all.

Unlike the previous predictions, I got this one completely inverted. I over predicted the influence of neuron count because this model is simple enough to get away with lower neuron counts. The optimal number of neurons for a model with good speed/accuracy would be a low value, but may be higher for more complex classifications.

4.4 Epochs

My thoughts on the number of epochs required was that if the model was of a good enough quality, the model should reach 100% fairly quickly, and if it didn't, the model must not be up to scratch. The number of epochs to reach 100% itself would be a performance indicator of the model more so than a major contributing factor to the models performance.

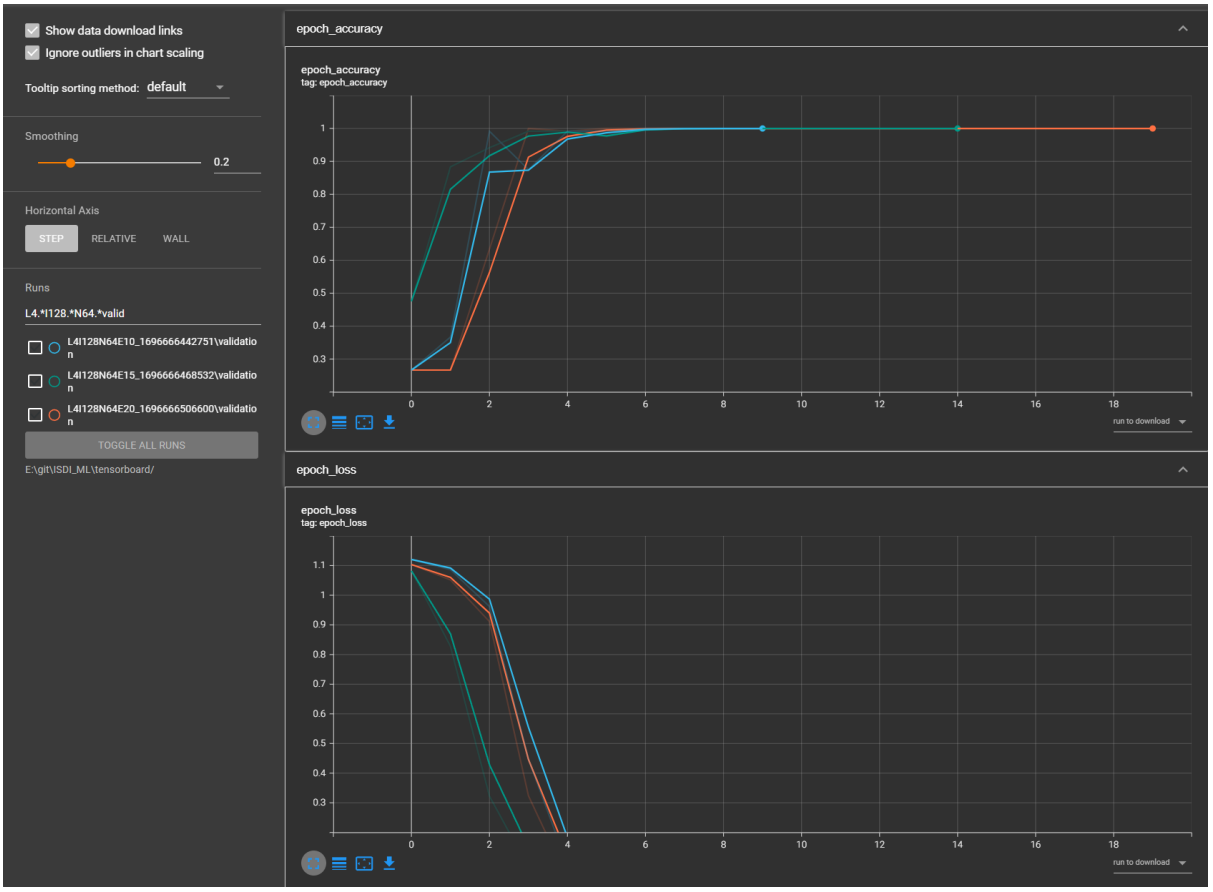


Figure 4.4A - Epoch (high) comparison graph

Blue:	low	(10 epochs)
Grey:	mid	(15 epochs)
Orange:	high	(20 epochs)

The key difference between this test and the aforementioned tests, is that changing the number of epochs only changes “how long” the model is allowed to train. All these models have identical parameters other than the number of epochs which means they are almost identical - such is the case in this graph.

A better comparison would be for lower performing models such as the models with the lowest settings:

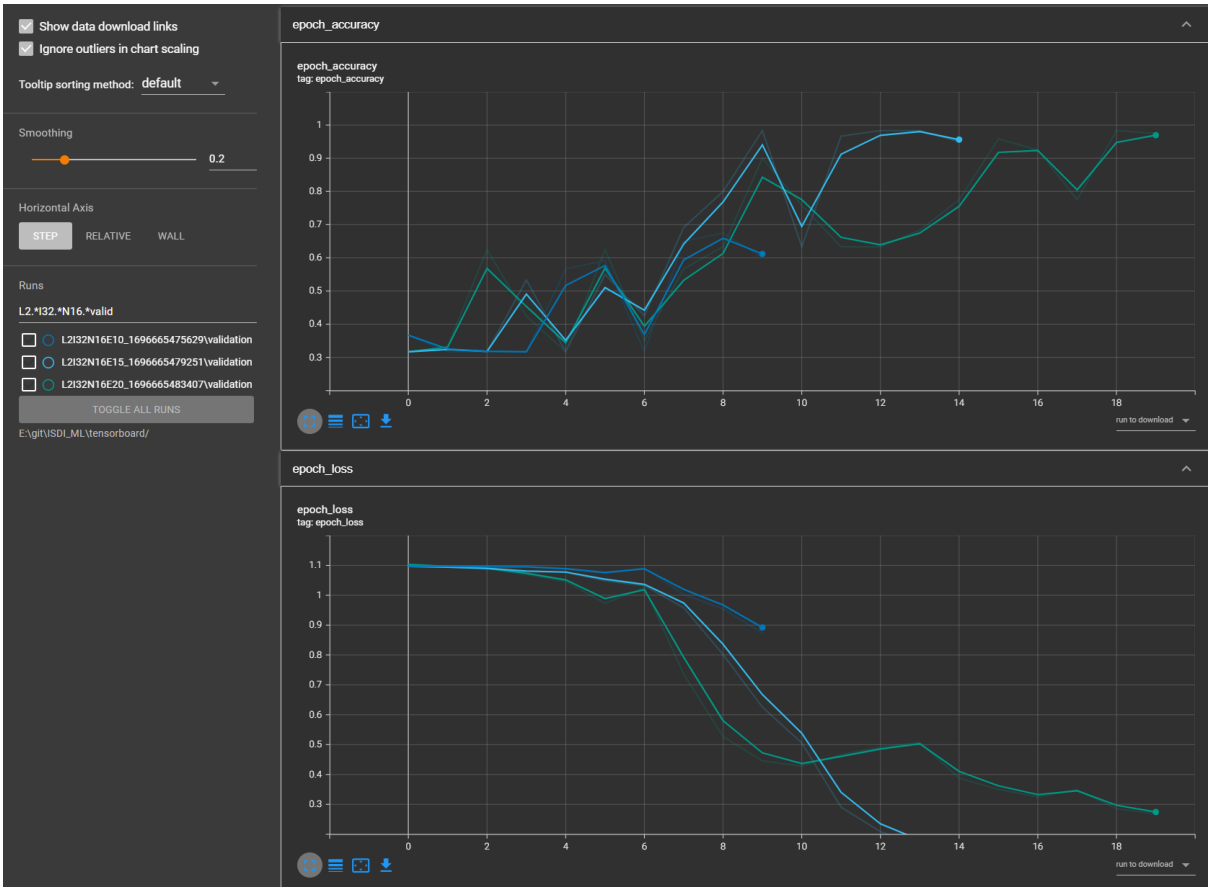


Figure 4.4B - Epoch (low) comparison graph

Dark Blue:	low	(10 epochs)
Light Blue:	mid	(15 epochs)
Grey:	high	(20 epochs)

Ah, much better! The effect of epoch number is far clearer here. The low model's training ends before it can even get to 70% accuracy, and the mid model only reaches the 90% range in its final epochs. The high model here shows a small drop in accuracy around the epoch 10-14 range, but looks like it may have made it to 100% accuracy in the late teens of epochs.

The number of epochs required to reach 100% accuracy is roughly inversely proportional to the quality of the model in all other respects. This means that a great model may only need a few epochs, and a rather inefficient model may need training for a considerable length of time. The recommendation for number of epochs is rather a recommendation to upgrade the rest of the model first and then test different epochs as the model matures. I got this prediction bang on the money.

4.5 Lowest, Middlest & Highest Comparison

Out of interest, here's a graph that shows the model with all low, all mid, and all high values.

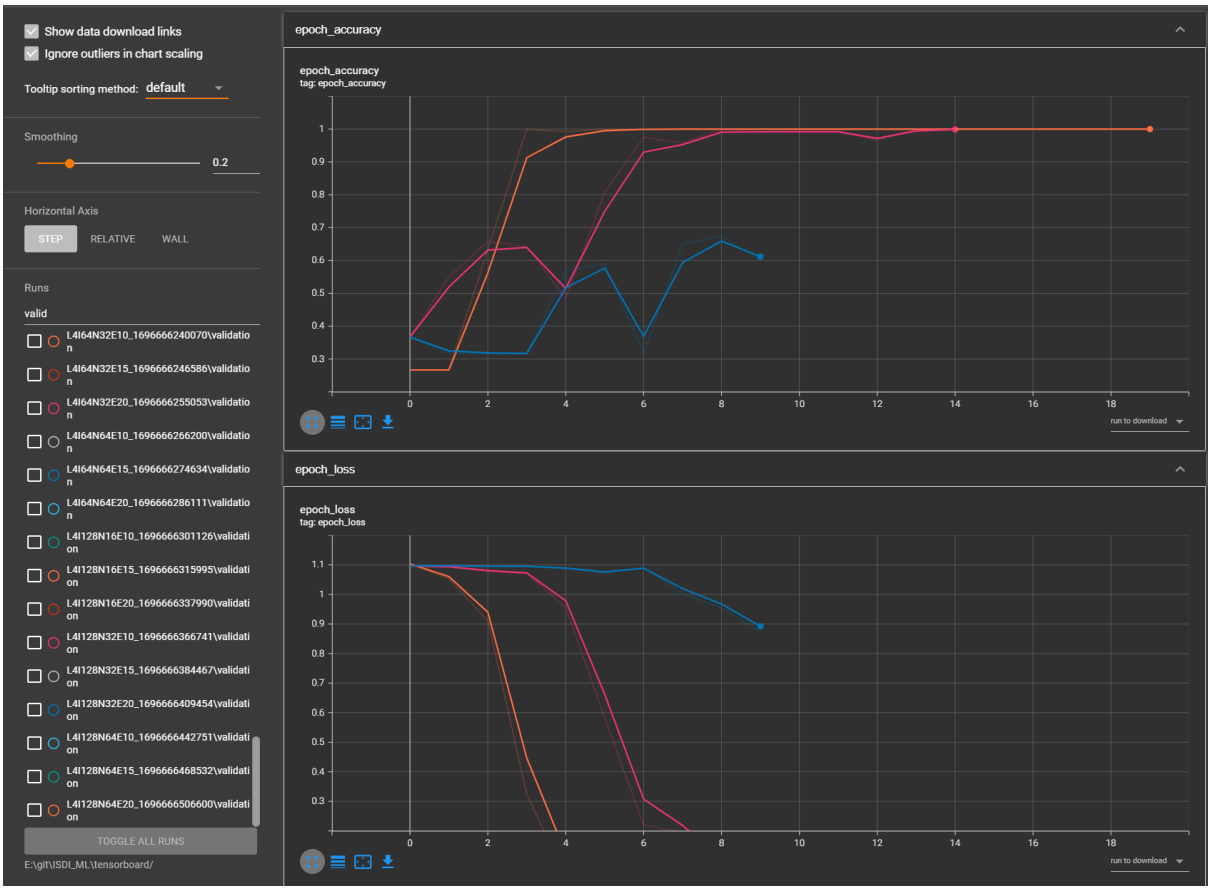


Figure 4.5 - Low, Mid & High model comparison graph

Blue:	low	(2 Layers, 32^2 pixels, 16 neurons, 10 epochs)
Pink:	mid	(3 Layers, 64^2 pixels, 32 neurons, 15 epochs)
Orange:	high	(4 Layers, 127^2 pixels, 64 neurons, 20 epochs)

To no surprise, the good model is the best, and the worst model is unusable.

4.6 Classifier Evaluation

Using the aforementioned knowledge from the Testing program and using some example code as a baseline, the classification program has been adapted and optimised to process all 600 example images.

```
100%|
Number of incorrect classifications: 0
Output directory:
E:\git\ISDI_ML\classified_images
Classification complete

End
```

Figure 4.6A - Classification program completion report

The completion report above shows that all 600 images were imported, categorised, stamped with their result, and exported to the output directory with an accuracy of 100%.

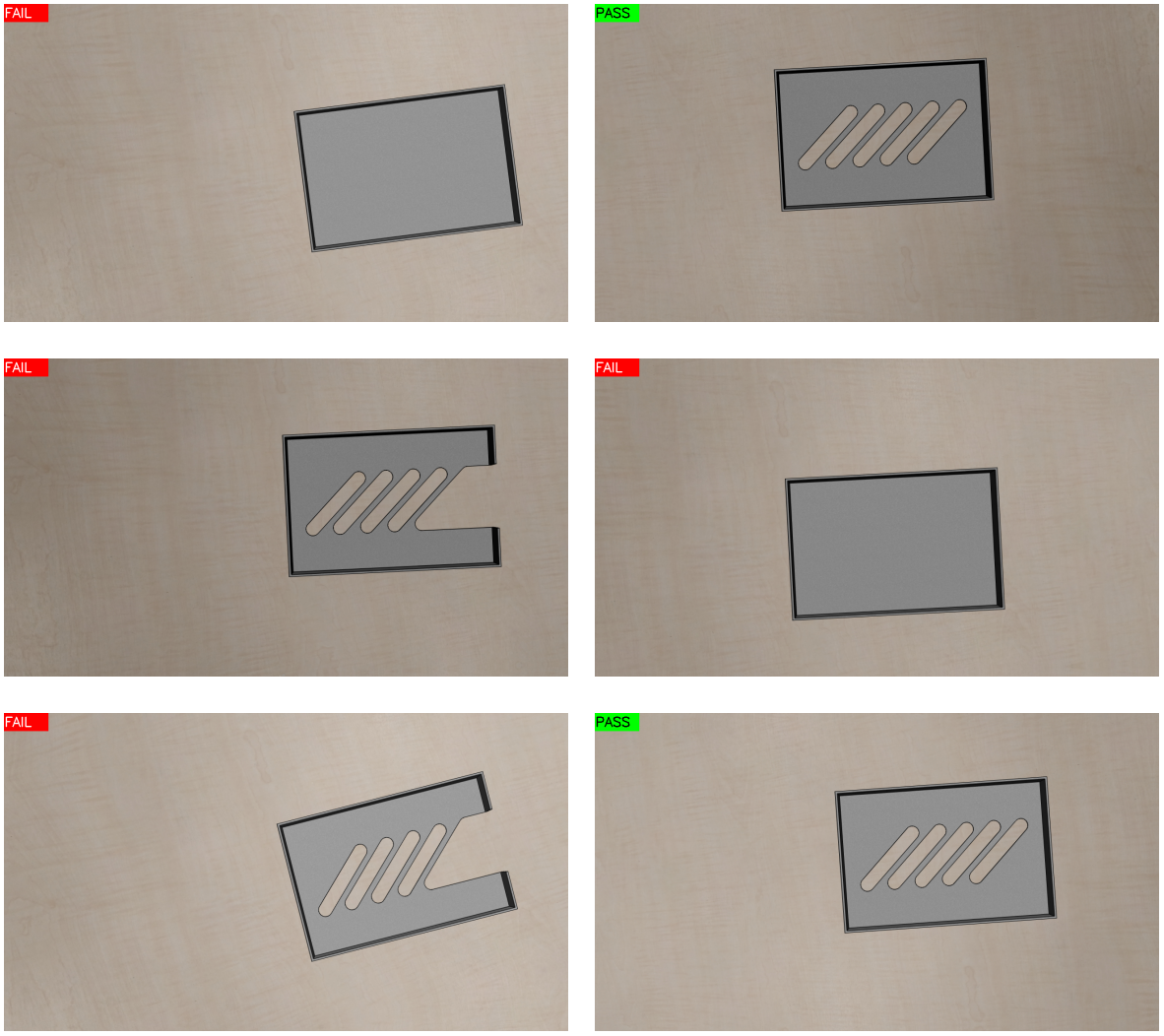


Figure 4.6B - Example classified images

4.7 Further Development

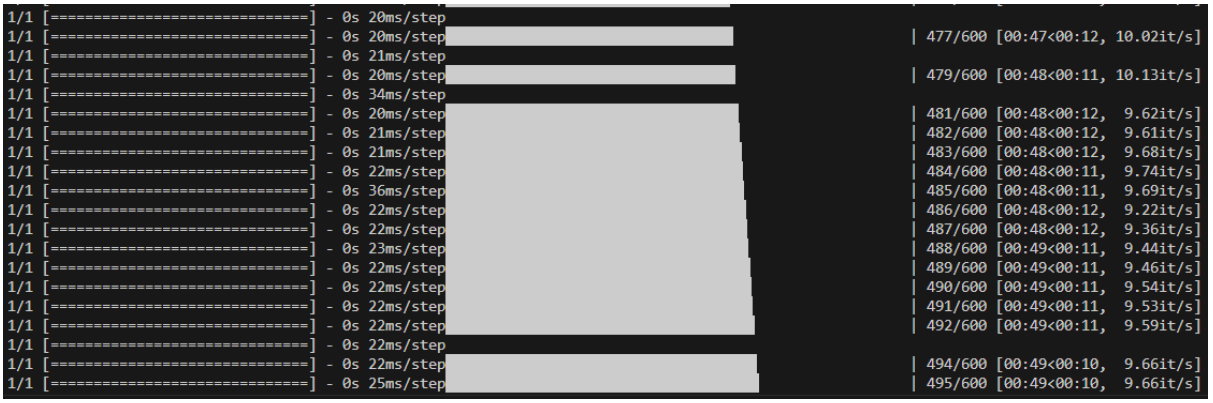


Figure 4.7 - Processing terminal output

The Classifier can turn around on average nine and a half images per second with about three quarters of that being the import and export of the images themselves. This could be sped up further with multicore processing, utilisation of GPUs or further refinement of the model to make it lighter.

One feature that is missing from this version of the Classifier is the distinction of Class_A and Class_B failures. The testing program did use *categorical_crossentropy* compilation protocol, but it did not make it into this version of the Classifier due to project constraints.

5. Conclusion & Reflection

The brief for this project stated the need for an image classification model that determines whether a manufactured part passes visual inspection. The algorithm I have created can fulfil this requirement with 100% accuracy (the brief only requires 80%) in under a tenth of a second per image. There are opportunities for further development such as further refinement and categorising failures further, but these are out of the scope of this project.

This project has been a fantastic experience to learn about machine learning and image classification. I hope that in the future I'll have the chance to further my knowledge in this field and apply these techniques to solve real world problems.