1. Introduction

In this report I will be exploring and showing my findings after implementing a deep neural network with an input layer, three hidden layers and an output layer to classify images from the CIFAR-10 dataset. A number of hyperparameters have been tweaked throughout experimentation to try and fit the model better.

## 2. Approach

In order to try and fit this particular dataset to a model, I opted to use a convolutional neural network as they are particularly good at and known for their ability with classification prediction problems and also with image data, this is due to image classification heavily relying on a models understanding of spatial relationships which is where a convolutional neural network shines. I am also more comfortable using a convolutional neural network than other neural networks, so it was a natural choice for me.

The layers I used were nothing out of the ordinary, the three hidden layers either 32 or 64 internal nodes or filters, then with a simple 3 x 3 matrix in order to fit the data after preprocessing.

I did pre process the data. After loading in the dataset, I began by normalizing the images to a number between 1 and 0. This is because the images RGB values are stored where each individual colour value for each pixel will be between 0 and 255 so to get the value between 1 and 0 the value had to be divided by 255.

Once the data had been normalized I got the class names of each label in the dataset and put them in order in array. I then made a class name conversion so that the label matched up with the correct number.

## 3. Methodology

To train and test my convolutional neural network I mainly used built in functions from the TensorFlow library. The data was loaded in as 50,000 training images and labels and 10,000 testing images and labels, it seemed like a good split to me so I left it as it was. Then after the preprocessing previously mentioned in the approach section I began the task of creating the convolutional neural network.

I used the Sequential method from the keras package in TensorFlow, this groups a linear stack of layers into a model, an input shape can be given however is not required as this method has automatic shape inference. I then added a Conv2D layer with 32 filters this is just due to the fact it was the best performing amount after a few trial and error attempts. Then I added a MaxPooling2D

layer to this layer just to remove the amount of pixels and unnecessary data. I then repeated this layer but with 64 filters and then I added a flattening layer as the information needs to be in a 1 dimensional vector in order to be processed.

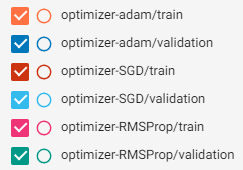
Lastly I added a dense layer of size 10 as there are 10 different labels that an image could be classified as.

The Hyper parameters I chose to select were Batch Size, Depth and Optimizer. I started with the optimizer as I would use the best performing optimizer for the rest of my experiments, I then chose to batch size to see if there were any notable improvements on a dataset this size. Lastly I wanted to test the depth as you would expect the more layers you have to give better more accurate results.

I also decided to record my test results using Tensorboard so that I had readily available graphs about my tests.

## 4. Results

The first parameters I chose to test were the different Optimizers. The three different optimizers I chose to test were Adam, SGD and RMSProp. I initially thought that Adam would be the best as that seems to commonly be the most popular and used optimizer, below are the following results from my tests with different optimizers.



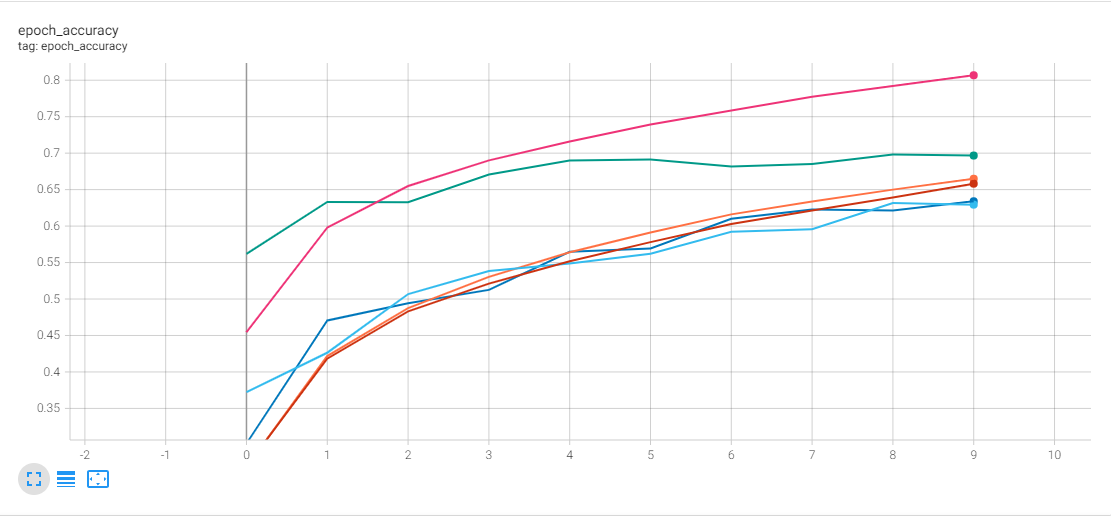


Figure 2. Optimizer Epoch Accuracy

Figure 1. Optimizer Key

## As can be seen in figure 2 the RMSProp seems to have highest accuracy after 10 epochs with both its validation and training results being higher than both of the other two optimizers with the SGD optimizer being the worst performing out of the three in this category. However the accuracy had stopped increasing by the 10th epoch or was at least seriously slowing indicated that the model was fitted and was around right before over fitting occurred. But in general all three optimizers had a steep climb early and then tailored off before stagnating pretty smoothly without any significant jumps in either direction.

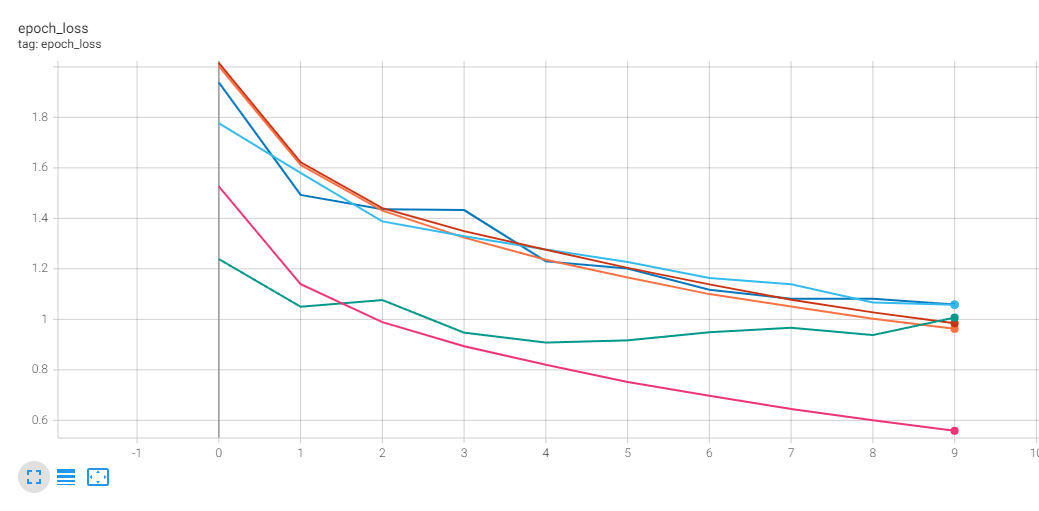


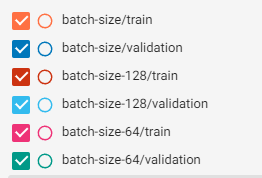
Figure 4. Epoch Accuracy

Figure 3. Optimizer Epoch Loss

Figure 3 shows the same general trends as figure 2 with SGD being the worst and then RMSProp being better, however RMSProps validation starts off then best slightly improves and then decreases with its loss going up above that of Adam which started off by far the worst. There could be a number of reasons for this but the most likely reason I see is that its an outlier.

By the looks of these results it would appear that RMSProp is the best optimizer to use for this scenario however I elected to use Adam for the rest of the experiments as it gave the second best results, was not too far off RMSProp performance wise and was more consistent to use with fewer jumps.

The next hyper parameter I chose to test was batch size, I knew previously that the main reason for using different batch sizes is when there are much larger datasets where trying to pass all the data through in one go would consume all the memory on a machine, so they are batched up to make processing it more manageable, however I wanted to see if the batch size effected the accuracy of a convolutional neural network.



## As can be seen I figure 4 the results are rather varied and do not seem to follow an particular pattern at all a batch size of 32 has both the best and worst accuracy respectively and the rest are all over the place, there are a few dips here and there however the overall pattern is that all of them improve by the final epoch significantly and rise fairly smoothly. The reason for these results may simply be that not enough tests were performed or that they are simply outliers however I believe the more believable reason to be that the batch size does not effect the accuracy at this scale.

Figure 4. Batch size key, orange and blue batch size 32

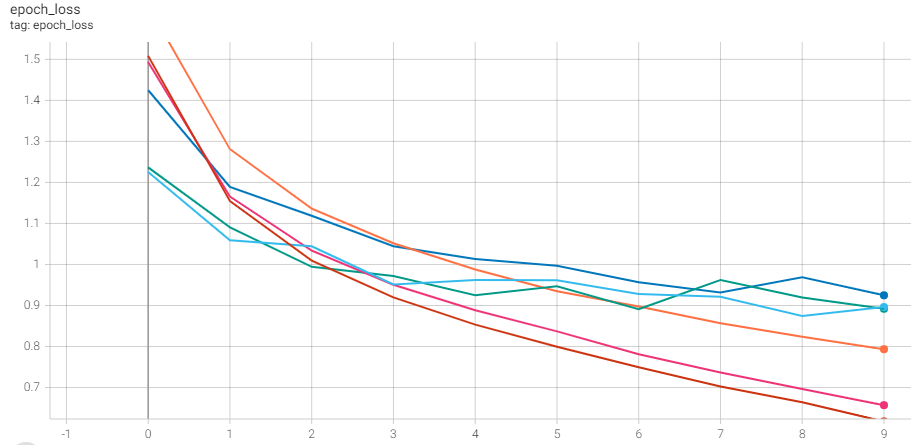
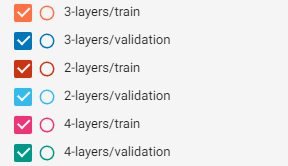
Figure 5 shows us a similar sort of pattern again, however this time all of the trains and all of the validations are grouped somewhat together. This time the ride is less smooth with more peaks and troughs. But the grouping and general nature of the pattern leads me to believe that there is no pattern or correlation between batch size and accuracy and loss.

Figure 5. Epoch loss

The last metric I experimented with was depth. I tested on a 2, 3 and 4 layered convolutional neural network my general thinking going into this was that the more layers the better performance the cnn would have.



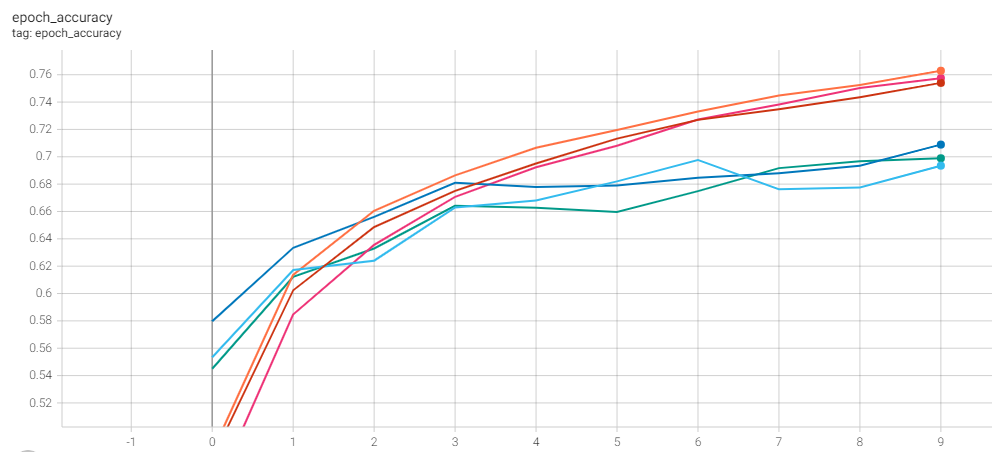


Figure 6. Depth key

in figure 7 you can see that the training and loss results are very close to each other and tightly grouped. There is hardly any difference in results between the cnn with 2 layers and the cnn with 4 layers. This could mean a few things, it could be that there is not too much correlation between more layers and accuracy, it could be an outlier. But the most likely problem here is that the layers I added and took out were not different enough to cause a big enough change in accuracy and should be improved upon next time. Other than that there was not too much movement and nearly all the lines follow a smooth patter where they start tailing off near the end.

Figure 7. Epoch Accuracy

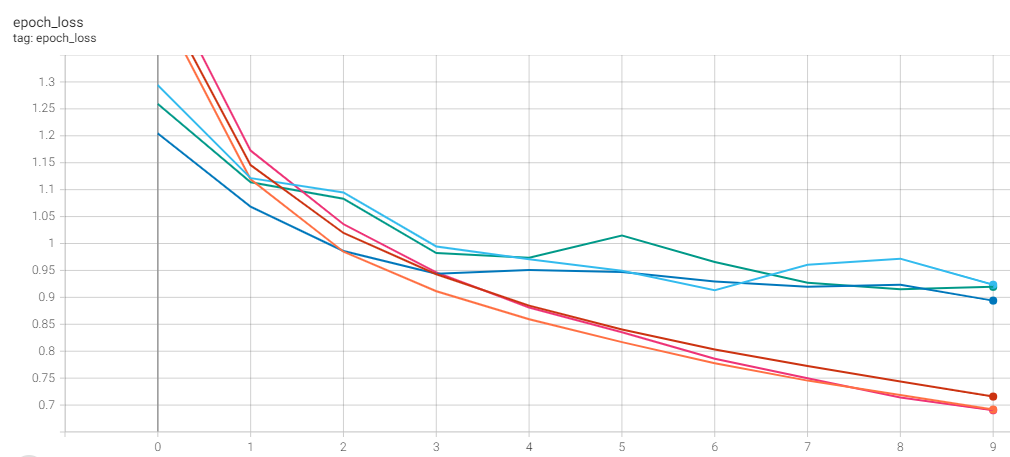


Figure 8. Epoch Loss

In figure 8 we again have the same pattern of the train and validation lines sticking together and grouping very close in an almost random order again showing that there is either not enough difference in the layers or that it does not affect the loss rates.

In conclusion I think there were ways to improve the work I did. I should have spent more time analyzing the testing data. I would also change the layers more when adding or removing them so that I may be able to see a more extreme change, I would also not pick batch size to use again as it was neither interesting or useful for this dataset and panning out more ahead next time would allow me to pick better hyper parameters to use and give me better results with more analysis