COMP 309
Machine Learning Tools and Techniques

Project: Image Classification

Author: Daniel Marshall
ID: 300440531

1: Introduction

This project is about building a model that can classify images provided into one of three categories/classes: ‘cherry’, ‘strawberry’, and ‘tomato’. This point is to build and save a model which does just that, and gain experience using the deep learning libraries in Python such as Keras and Tensorflow whilst doing this, along with a basic understanding of convolutional neural networks. The final model was then run against a test set of unseen images to determine the accuracy of the model.

2: Problem Investigation

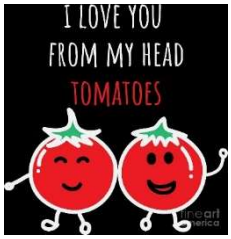
2.1 Exploratory Data Analysis

This section is where I will talk about the exploratory data analysis that I performed on this dataset. Given to us in the dataset were 4500 images, 1500 of which represented cherries, 1500 represented tomatoes, and the final 1500 represented strawberries. The first thing I did was go through and look at the images, and several stood out to me as noisy and unsuitable for the task. There were several main issues with these images, the first of which was that some did not contain an actual image of the fruit.

Take for example the image ‘cherry_0873’, which is just an image of two cherry trees. This is very unhelpful for an image classifier, as this is not a cherry. A human could easily see that this is a cherry tree, but this is not relevant to the class cherry, as the actual cherries on the tree are far too hard to make out for such a small image.

‘cherry_0873’

The next issue I found with the dataset was that several images were not actual photographs of the specified fruit, but rather they were artistic renditions of the fruits. These ranged from fruit-shaped handbags to t-shirts with fruit on them, to cartoons, to simple drawings of the fruit. One example, ‘tomato_0147’, is shown below, which I believed to be unhelpful to my classifier.



2.2 Pre-processing Techniques

The first technique I applied when pre-processing my data was to remove images like the ones I had discussed above, as I decided these would not be helpful to the classifier, as they are typically oversimplified versions of the fruit. Or they are deformed in some way as to make them noisy, whether it be the fact that some of the fruit in these images had eyes and mouths or they had arms and legs.

I loaded my images using the ImageGenerator function from the Keras library, which allowed me to apply several other pre-processing steps to the data. The first one I took was to set the zoom range to 0.2, which allows Keras to scale the images between 0.8-1.2 times the original size. This helps deal with issues of scale by randomising and varying the scale of the images.

The second and third parameters I altered were the horizontal_flip parameter and the vertical_flip parameter. These allowed the images to occasionally be flipped either vertically or randomly, which increases the variability of the input.

I also set the rotation_range to 40, allowing the images to be occasionally rotated by as much as 40 degrees, to increase input randomness.

I also rescaled the images by setting the rescale parameter equal to 1./255. This effectively normalises the data input, as the numbers were previously between 0 and 255, and then rescaling was done as computers perform better with normalised data in general.

I also experimented with epoch sizes, noticing that my classification accuracy tended to start low, increase and then gradually flatten out. For this reason, I set my epoch to 60, but added a call back with early stopping to my fit function, so that if the validation accuracy wasn't increasing to stop running the algorithm, so normally my runs ended up taking around 30-40 epochs to achieve the best results.

By the end of my pre-processing, I ended up with an array of 300x300x3, as the image sizes were left as 300 by 300, and I decided to leave the inputs as coloured as opposed to greyscale, so have 3 channels instead of one. The pre-processing steps did end up improving the accuracy of my model from around 50% to 67%, a significant improvement.

I then tested my model using an image size of 64x64 and didn't see a very large drop in accuracy, so I decided to keep it at this size as the time taken dropped dramatically from this reduction in image size. For my final model, I switched to 128x128 as this raised my accuracy slightly

3 Methodology

3.1 How I used the given images

The images provided to us were 4500 images, 1500 of each cherry, strawberry, and tomato. I removed around 50 images from each as part of my pre-processing. I then experimented with different ways of splitting the images into training and validation sets. I tried out three different splits for training to validation: 70:30, 80:20, and 90:10, and without changing any other model parameters these were my results. The parameter I was most interested in optimising for was validation accuracy, as this is what the final model would be tested on.

Train-Validation Split Analysis			
Train-Validation Split	Validation Accuracy	Validation Loss	Learn Time (s)
70:30	0.7393	0.6871	401
80:20	0.7444	0.6388	302
90:10	0.7887	0.6249	340

Having looked at the table, I then ranked each category, labelling green the best result, orange the runner up, and red the worst-performing split in that category. Note that this colouring convention is followed in later tables in this report. Surprisingly to me, the 90:10 split for train-validation produced the highest validation accuracy, by a considerable margin. I thought that 10% would be too small for a validation set, but this was the clear winner. It also had the smallest validation loss, another bonus. The 70:30 split came last in the two categories, as there was simply not enough data for the model to learn from, so it was the most inaccurate.

Following these results, I decided to use the 90:10 split for my final model.

3.2 Investigating Loss Functions

The second tuning step I carried out was investigating loss functions. Since this is a multi-class classification, many typical loss functions are not applicable for the task, such as MSE, MSLE, and MAE, as these are regression loss functions. I found three different loss functions that I believed could be applicable for the task, ‘categorical_crossentropy’, ‘sparse_categorical_crossentropy’, and ‘kullback_leibler_divergence’.

I ran into a problem when trying to use ‘sparse_categorical_crossentropy’, as it expects labels to be integers, but my labels are one-hot encoded, so they don’t work with this function. This just left the two other loss functions to try. I then ran my model using the

previously selected 90:10 split and swapped out the loss functions to determine which was the most successful, leading to the following table.

Loss Function Analysis			
Loss Function	Validation Accuracy	Validation Loss	Learn Time (s)
categorical_crossentropy	0.7840	0.5792	424
kullback_leibler_divergence	0.7723	0.6048	323

Looking at this table, I can see that categorical_crossentropy is the better loss function to use for this problem, so this is the one that I chose for my final model, although it took slightly longer than kullback_leibler_divergence. Interestingly, this is also the standard loss function for convolutional neural networks.

3.3 Investigating Optimisers

The next tuning step was to look at the different optimisers I could use for this problem. I selected five potential candidates: 'Adam', 'Adamax', 'Nadam', 'RMSprop', and 'SGD'. I selected these algorithms so that I had enough to be able to make a reasonable choice as to what the best optimiser for the given scenario was.

Optimiser Analysis			
Optimiser	Validation Accuracy	Validation Loss	Learn Time (s)
Adam	0.7300	0.6760	262
Adamax	0.8052	0.5205	428
Nadam	0.7864	0.5694	280
RMSprop	0.7441	0.6745	200
SGD	0.5845	0.9305	234

After running my algorithm for each optimiser, there was a clear winner, which was Adamax, as it managed to achieve a validation accuracy of over 80%, which was the personal goal I set for myself for this project, so that was very good to see. One downside to Adamax is that it did take significantly longer to run than the other four optimisers that I had picked for this section. SGD was the worst optimiser by a considerable margin, which was what I expected, as SGD is the simplest optimiser. This was the optimiser used to train the base MLP model, and also one of the optimisers that were used in the previous assignment. I was surprised that Adam was the second-worst algorithm in terms of

validation accuracy out of the 5, although it is worth noting that the top four all have reasonably good accuracies, and Adam is only 0.0141 behind RMSprop for accuracy.

3.4 Investigating the Regularisation Strategy

According to Shubham Jain of Analytics Vidhya, regularisation is a technique that makes slight modifications to the learning algorithm such that the model generalizes better [1]. What this means is that without any regularisation strategies, there is a high chance the trained model will be overfitted on the training data. Overfitting is where the model learns to model the training set very well but then fails to generalise enough to be able to adapt to new data, so will perform very badly on the test set.

To avoid overfitting, we must employ regularisation techniques. The two that I implemented were adding dropout layers and using early stopping. Dropout layers randomly remove a set percentage of the nodes in a layer, and all connections going to and from them. This helps as it increases the randomness of the model, which is good to prevent overfitting.

The second regularisation technique I have implemented is early stopping. When I run my model for 50 epochs, what happens is the validation accuracy starts poor and increases, and then after several epochs, the validation accuracy reaches a maximum and starts to flatten out, or even decreases. When it decreases, this is because the model has started to overfit. To combat this, I have implemented an early stopping call back which stops training the model if the validation accuracy hasn't improved in 8 epochs, and saves the model, disregarding the last 8 epochs.

3.5 Investigating Activation Functions

Activation functions are mathematical equations that determine the output of a neural network. They are used to determine whether a specific neuron should fire or not, depending on whether the neuron is relevant to the classification. There are several available, and for this assignment, I decided to experiment with several non-linear activation functions, specifically 'relu', 'sigmoid', 'softmax', and 'tanh'. Since every single layer could theoretically have a different activation function, and I have four convolutional and 3 dense layers, there are 4^7 combinations of activation functions.

Since my model takes several minutes to run, I decided to simply try each one for every layer, to make this feasible. Therefore, I had four runs, each one having the same activation function for every layer. Note that this isn't quite true, as I never changed the output layer's activation function from 'softmax'. This is because softmax gives the probability of the input value being in a specific class, so is used for the output layer to produce the final classification. However, I did change every other activation function.

Activation Function Analysis			
Activation Function	Validation Accuracy	Validation Loss	Learn Time (s)
relu	0.8192	0.5095	393
sigmoid	0.3404	1.0985	72
softmax	0.3404	1.0985	83
tanh	0.6995	0.7619	225

After running my model for each activation function, relu was the clear winner, and sigmoid and softmax performed terribly, only slightly better than complete guessing. However, Relu did take significantly longer than the other activation functions. Tanh performed okay, but I ended up choosing relu for my final model simply because of the validation accuracy.

3.6 Investigating Hyper-parameters

I also briefly experimented with exploring hyper-parameters. Hyper-parameters are variable to do with determining the structure of a network, and how it is trained. These are things like the number of convolutional and dense layers, the number of epochs, and the batch size. I experimented with changing these options around, however, did not implement a grid search.

Firstly, I experimented with batch size. The default provided batch size is 32, so I tried 16, 64, and 128, but none of these batch sizes improved my performance over the default size of 32, which is the most common batch size anyway.

Next was the number of epochs. I originally started with 300 epochs, which took a very long time. I noticed that the model started overfitting after a certain number of epochs, and that the large amount was making it worse. I kept shortening it until eventually, I arrived at 60 epochs as he most suitable for my network. This was made slightly redundant by the early stopping that I implemented, as the model never reached the full number of epochs after implementing that.

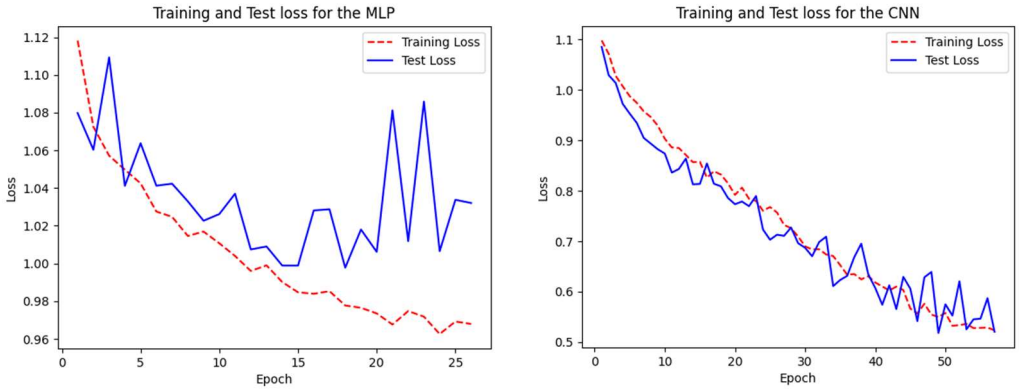
Lastly was the number of convolutional and dense layers. My base MLP contained just two dense layers. I added a third and performance increased. I then added convolutional layers, gradually increasing the number of them, and found four to be optimal, as after four the validation accuracy started decreasing.

4 Baseline Model (MLP) Comparison

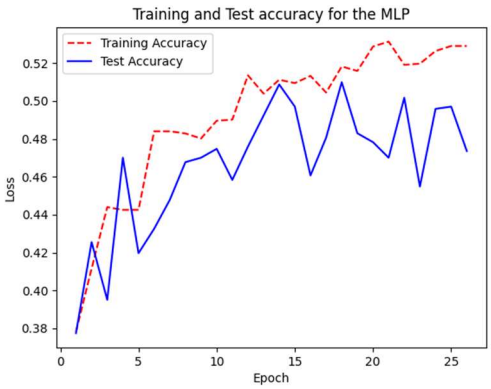
My baseline MLP consisted of two dense layers and no convolutional layers. It also used the relu activation function and the basic SGD optimiser. I used these settings to simulate what I believed to be a very simple neural network with multiple layers. I have plotted graphs for the training and validation loss and accuracy metrics and have done this for both the MLP and my CNN. I will compare the two pairs of graphs, and the time taken for each algorithm to compute to determine the differences between the two models.

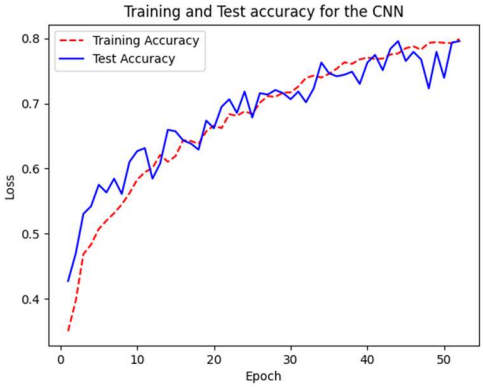
My basic MLP achieved an accuracy of 51% in an average time of 195 seconds. Compared to my CNN, which achieved an average accuracy of 80% in an average time of 423 seconds, the MLP is the worse of the two classification algorithms. Although it is worth noting that the MLP was more than two times as fast as my CNN, so if speed is more important than accuracy the MLP may be better.

The diagrams below compare the training and test losses for the MLP then CNN. Looking at the graphs, we can see that the CNN achieved a much lower loss, tapering out to a loss of around 0.52 for both training and test. In contrast, the MLP loss jumped around significantly, finishing up at a loss of around 1.04 for the test and 0.97 for the training data, showing overfitting.



The diagrams below compare the training and test accuracies for the MLP then CNN. Looking at the graphs, we can see that the CNN achieved a much higher accuracy, tapering out to an accuracy of around 0.80 for both training and test. In contrast, the MLP accuracy jumped around significantly, finishing up at an accuracy of around 0.57 for the test and 0.52 for the training data, showing overfitting.





5 Conclusions and Future Work

In conclusion, the CNN achieved much higher accuracy than my MLP model (84% vs 52%), and this tells me that the CNN is the far superior model for image classification. This increase in accuracy is a major pro of using my model over the MLP, however, one big con of my model is that it is much slower than the MLP model, simply because my model has several more layers than the MLP.

Looking ahead to potential future work, if I was to try to improve my accuracy further, I would download additional images to enrich my dataset. I would also investigate using pre-trained models, such as VGG16 and VGG19, as from my understanding they can lead to accuracies higher than the one I was able to achieve building my model. I would also consider automatically tuning my hyper-parameters, as the limited tests I did with them are by no means exhaustive, and I feel that this may lead to greater overall accuracy.

6 References

[1] - Shubham Jain, 2018, An Overview of Regularization Techniques in Deep Learning (with Python code) - https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/