COMP 309 PROJECT
Image Classifier

Tyler Shamsuddoha
300428076

INTRODUCTION

Throughout the project, the aim is to classify the data in the unseen test set between three different categories. The categories were strawberries, cherries and tomatoes. The end goal for this project is to develop a classifier that is able to classify the unseen test data accurately into these three categories of fruit as well as compare this classifier to the performance of the baseline model, which is a multilayered perceptron. I approached this assignment by implementing a simple multilayer perceptron in Keras, following online guidance, than used it to obtain a baseline accuracy. This baseline accuracy would be used as a comparison to the CNN model that I develop in order to report on the differences between the results. I aim to achieve this by iteratively tuning my CNN model and examining the effect of different parameters and optimisers on the classification accuracy. I believe that subsequently I should be left with a solution that's close to optimal for the unseen test data set. However, there are a lot of factors that could hinder this which I will address as difficulties within this process.

PROBLEM INVESTIGATION

EDA (Exploratory Data Analysis) is the process of analysing data to become familiar with the details about the data set. It is needed in the classification process in order to understand the data thoroughly. Throughout the EDA process, we are able to look at different aspects of the data and bring different areas of the data into focus. In order to do this, we must observe the data from from many angles in order to summarise the data without making any assumptions.

First I opened the data folder and observed the data, I noted that there were three different classes within the train and the test data, each one representing the fruit that we were to classify. Before looking at images in detail, I selected all the images in the folder to see if the classes were balanced.

Class	Amount of Instances	Balanced
Cherry	1500	YES
Strawberry	1500	YES
Tomato	1500	YES

As each class contained 1500 instances, the three classes of fruit were balanced initially. Next I looked at the data in depth and scrolled through the images. Although the project handout states that all the images are of the size 300 x 300, I decided to double check by writing a python script that would identify any images that are not 300 x 300. Using the python script on the training folder I found no outlying images. However I decided the script would be handy in the later stages in the project when adding images to the training data.

An immediate problem that I discovered when observing the images inside the training set data, was that a lot of the images were not suitable for training. The images contained a variety of noise, some images in the document had a lot of noise which I immediately thought should be removed, whilst some images in the folder contained minimal noise which I initially thought would be better for training.

There were different types of noise in the images. The different types of noise that I was initially concerned about and found within the images is listed below:

- Images with a lot of filters applied, so that subject fruit is extremely unclear.
- Leaves in images that are not from the subject fruit, but another fruit tree.
- Multiple fruit, in which the subject fruit is not the subject fruit.
- Photos that do not contain the subject fruit.
- Photos in which the fruit is not the main subject of the photo.
- Distorted colours in photos.

Upon further thought, there are multiple ways to deal with this problem. One way could be to simply remove the images from the training data. Another way would be to edit the images in a way that solves the problem and keeps the image in the training set. This could be by cropping noise out of images, editing filters, focusing on the subject fruit etc. There are problems with both solutions. Removing the images means that we have to define under what criteria an image in the data is removed. This can be hard as since the test set is unseen by us, I am unsure about what images it will contain. Therefore removing images from the training set when similar images will be present in the test set will be detrimental to the model as it means the model will not be as robust when it sees the same images in the test set. This is because

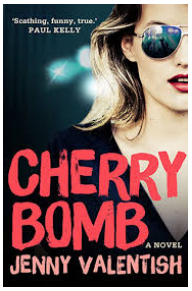
since the model has not learned what the noisy images that we have removed are classified as, if it experiences anything similar to these images, it will view them as an outlier and not be able to accurately classify them. Therefore there is a strong grey area when it comes to determining what images should be removed due to noise.

The solution of editing an image to correct it's noise for our training data, would mitigate this as the training set will be robust with images that could appear in the test set. However this is much more labour intensive and editing the images one by one is not worth this, as the edit might not be substantial to mitigate the noise. Therefore the solution that I decided on was a combination of the two, in which I would remove images from the training set based on certain conditions and then enrich the training set with more images to account for the images that I deleted. Removing an image is much less labour intensive than editing each image independently, however it could mean that the training data has less images which could lead to a classifier that is less trained and therefore less accurate at predicting the test set.

Below are some examples of changes that I made to the training data, in which I decided what alterations I should do to the dataset based on the noise and the context of the image.

Subject fruit is not present in the image

1) 

2) 

3) 

These three images are examples of images that I removed from the training data. I removed these images from the data folder. Eg. For image 1, 2 and 3 the subject fruit is not located in the picture. Therefore it is highly unlikely that the classifier will classify it correctly. It also means that the classifier will not be learning due to these inappropriate images. Therefore removing these images will guarantee an increase in accuracy compared to leaving them inside the data.

Altered images of the fruit/Levels of ripeness

4) 

5) 

6) 

I feel like it is important to address images in which the subject fruit is cropped, or not fully visible. This should not be accounted as noise but as variance. This is similar to what I can adjust in the tuning stage of the classification progress. Looking through the data folder it is evident that there are many photos that are already slightly distorted (Image 6), With different levels of brightness and contrast (Image 6), taken at different angles (Image 4) or zoomed in on (Image 5). These photos will be left in as they provide almost an extent of tuning to the image data. There are also a multitude of different levels of ripeness from the fruits. Since the test set is unseen, the best way to make the model would be to leave all different levels of ripeness and colours in the training data, in an attempt to not limit the classifier to classify one type of fruit or one colour of the type of fruit correctly.

Subject fruit is not the primary focus of image

7) 

8) 

9) 

For images 7, 8 and 9 we can see that the subject fruit in the image is not clear. Image 4 is a drawing in which the tomatoes are not only drawn in a cartoon style, but the tomatoes are hardly the focus of the image with them taking up such little space comparatively. Image 5 the tomato can hardly be viewed underneath the pasta in the image and finally for image 6 the tomato is amongst other fruit, including capsicum. It will classify the image based on the features from the capsicum. The target class is therefore not the main focus of these images, even though it is contained within the image. Images like this should be removed as well. The removal of images like this was essential in order to be confident about the quantity of noise that I had in the dataset. It is important to note that these types of images require some subjectiveness in order to decide which ones should be removed. Also that a certain level of noise is important within the dataset as to be accurate and realistic representation to what would be reflected in the real world and since the test dataset is unseen, it is safe to leave a small amount of noise in case images with a certain amount of noise are contained in the test set.

I decided to also resize the images to 64 x 64 when adding them to the ImageDataGenerator.. This is because the images were not able to run on my computer without the computer crashing, reducing the size was a slight concern at first as I thought that the model might yield some accuracy loss as it would be missing pixels in the images. However when I ran the images on the uni computers overnight with the full 300 x 300 size, I ended up getting the same results as the 64 x 64 pixel images. Therefore I decided the model should maintain having it's images stay at this size as it seemed to be fine and mitigated the crashing problem I faced.

Although removing the images did yield a slight improvement in accuracy, this is not 100% definitive of what the test accuracy will result in. This is because some of the removed images could actually be images could actually be the types of images that the CNN will encounter when it is trained with the test set.

Data Augmentation

Data augmentation plays an important role in the data science process. The training set provided has a limited amount of instances. Therefore data augmentation is crucial when training the model, in order to represent the possibility of images that the CNN can be tested against with greater accuracy. The idea behind data augmentation is that if there is increased variance in the training data, the model will be robust to the outside world and any unseen test data that it may encounter.

Initially, I thought of augmenting and adding images at the same time. This could be done following a simple tutorial that keras provide in which altered images are also saved in the training directory as their own image (training instance), effectively increasing the training size by the amount of images that have had data augmentation applied to them as well as ensuring that the training set is robust and increasingly correlated to what the real world data would represent. The idea was that if implemented correctly the model's classification accuracy would benefit from both these reasons, as well as save me the tedious process of manually saving and adding an extra 200+ images to the training set. However when implementing this I ran into the issue of not having enough memory to do this, and would frequently get a memory error when trying to save the images into the training directory. This means that I had to add images manually, by saving images online to enrich the training set and that the augmentation to the images would have to be executed at runtime creating a longer training training process.

I decided to implement a multitude of data augmentation procedures at run time by adding them as parameters in the ImageDataGenerator. Some of the ways that I augmented the data is shown below:

Rescaling

I rescaled the images to 1/255. (Pixels in an image can be between the range 0 - 255). This gave me a decimal number which I multiplied each pixel value by, therefore effectively normalising the pixels in each image within the training data. Normalisation is significant in this data set as it means that all the images have the same influence on the weights and therefore the loss of the function. This type of normalisation is easy to implement within our data as the pixel values themselves is the domain, 0 is white and 255 is black.

Zoom Range

I implemented the keras augmentation of zoom_range in order to make the model for the fruit dataset more robust to a variety of changes in the subject fruit size. Although there was a lot of different size fruits in the dataset already, there was no harm in implementing this to ensure an even greater variety of angles and viewpoints of the subject fruit, in an attempt to achieve a higher classification accuracy. This function also somewhat simulates the cropping of random features in an image, therefore also making the classifier increasingly robust to subject size changes, angles and noise. I set it to only 0.2 as if the zooming is too prominent, than it could create noise in the data as it might be zoomed into areas that are not the subject fruit. This therefore highlights the importance of preprocessing the images in which the subject fruit is not the main focus of the image, as it ensures that once the zoom augmentation is applied that the chance of it zooming onto a part of the subject fruit is increased.

Rotation Range

Rotation range was implemented in the ImageDataGenerator as well. Rotation range works by specifying an angle and the ImageDataGenerator rotates the image by a random angle between the negative and positive of the selected number. For example, for the parameter 'rotation_range = 35' the images will be rotated at a random angle that is between -35 and 35 degrees. The idea behind adding this parameter in was that it would make the model understand the subject fruits from an increased amount of angles. The trade off is however if the rotation is too apparent and subsequently results in the model not understanding the subject that it is looking at, or if rotating the image by a certain amount results in the subject fruit not becoming the focus of the entirety of the image. Therefore, like many of the parameters in the ImageDataGenerator there is an optimal amount that is needed for this parameter to be effective, based on the dataset. It is important to note that initially, I noticed that images in the dataset itself already represent the fruit from many angles, therefore this parameter might not be needed as much as the other ones. However adding a rotation_range that was not too strong did tend to increase the accuracy of my model slightly, furthermore highlighting how the training set still could benefit from an increased amount of image instances with the rotation applied. So I decided to keep it in for most of the training tests. I found that the optimal amount for increasing the accuracy was setting the rotation range to around 40, meaning that the ImageDataGenerator would rotate the images at a random angle between -40 and 40. Regardless rotation range is also important because it is important to ensure that the model is trained against a variety of images so that it accurately trains itself in correspondence to the noise the world would represent.

METHODOLOGY

Training and Test Splits + Loading Image Files

After cleaning the data using the framework explained above, I had a total of 3748 instances of training data (1257 Cherry + 1240 Strawberry + 1251 Tomato). I decided that I would use validation sets to monitor the loss and the accuracy of the training process in real time against data that the model has not seen before. In order to implement this I first randomly selected 15% of the new size of each image class to place in the validation data. This was to ensure that the proportion was the same throughout the validation data (222 Cherry, 219 Strawberry, 221 Tomato) when compared to the training.

```
print("Loading Images...")

validation_image_generator = ImageDataGenerator(rescale=1./255, rotation_range=40)
val_data_gen = validation_image_generator.flow_from_directory(batch_size=batch_size,
                                                              directory=validation_images_dir,
                                                              target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                              class_mode='categorical')
print("Loaded Validation Set Images Successfully\n")

train_image_generator = ImageDataGenerator(rescale=1./255, zoom_range=0.2, rotation_range=40)
train_data_gen = train_image_generator.flow_from_directory(batch_size=batch_size,
                                                           directory=train_dir,
                                                           # shuffle=True,
                                                           target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                           class_mode='categorical')
print("Loaded Training Images Successfully\n")
print("Starting training....\n")
```

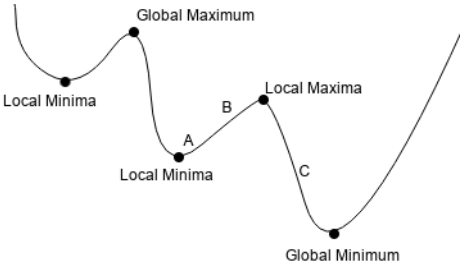
Although I originally loaded the data using the arrays in Bing's loadData() function, I eventually decided that using an ImageDataGenerator from Keras to import the training set and a seperate image data generator to import the validation set data was more efficient. This is because the generator provided easy inputtable parameters in order to provide the image manipulation which saved me the hassle of manually addressing the image data within the arrays inside python. Using the ImageDataGenerator also turned out to be beneficial as Keras provides a flow.from() function, which I used to specify the classes of fruit automatically based on the folder structure, which was already provided in the Template code handout. ImageDataGenerator by Keras provides many online resources and tutorials on how the augmentation works within the function which made it easier for me to manipulate image data in ways that would be of greater benefit towards the classifier model. It did this by providing simple parameters to pass through to the generator instance. These varied from flips, shuffles, zooms, crops as well as changing the brightness, contrast and so much more which I will go into detail below.

The Loss Function

Loss functions are used in data science to fully understand and improve machine learning algorithms. It works by mapping a number to one or many variables, therefore representing the cost at different stages in the training process, or the estimated value against the actual value. When implemented inside the CNN, the loss function is crucial as it is used to quantify the quality of the prediction model. I decided to choose the loss function of Categorical_Crossentropy, or also known as 'Softmax Loss'. This is because it combined Softmax activation with the Cross Entropy loss. I will talk more about the softmax activation function below, but it is important to note that the loss function of categorical cross entropy is heavily reliant on Softmax as it and cross entropy are used in unison with each other for this loss function. It is also important to note that while there are other loss functions such as MSE, MAE, etc these are mainly used for regression problems. Since we are doing a classification task with >2 classes, it is appropriate to ensure that the classifier for the loss function is switched from binary to categorical, thus justifying my selection of categorical_crossentropy.

Optimisation Method

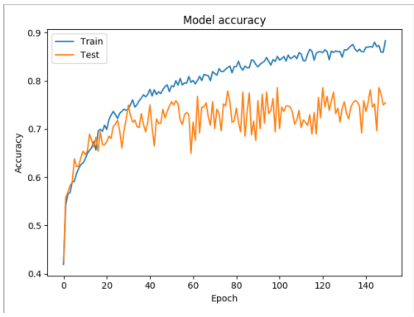
For the optimizer in the model I decided to try a lot before I settled with one. The first optimizer is the one that we were taught in class, as well as used in the previous assignment. This was the Stochastic Gradient Descent optimizer (SGD). My reasoning behind using SGD was because I was familiar with it from learning about it throughout machine learning and touching over it in the lectures. It was also the recommended classifier for the binary classification example that Keras provides in their website. I recall that while Batch Descent was able to be computationally faster than SGD, SGD mitigates being stuck at local minima by being able to move closer to a maxima, in order to eventually end up at a better minima. This is because of the random noise that the SGD algorithm introduces. However, the tradeoff for this is computation time since SGD is prone to overshooting the random jumps and missing the local minima. To help with this, we are able to decrease the learning rate over time. Therefore as the algorithm progresses,we end up eventually taking smaller steps and finally ending up at a global minimum or an area that has a very significantly low level of loss.



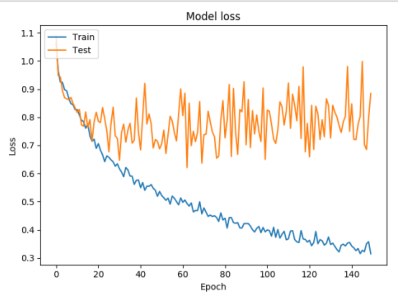
SGD allows up to take a step towards a worse loss, in order to reach a better minimum eventually. Eg. Starting at A and steping towards B (Worse loss than A) in order to reach C (Better loss than A and B)

It is important to note that when using SGD in Keras, you require an adaptive learning rate, otherwise it becomes much easier to miss a local minima. This is because as the learning rate is high, the steps that the model is taking are big, therefore meaning that it becomes easy to overshoot and miss the regions in which the loss is low. The way to configure this is with the Decay parameter when creating the optimizer. Decay ensures that the learning rate is not fixed and that it changes over time, as good practice we want to set it to the learning rate divided by the number of epochs. This therefore ensures that the learning rate is decreasing as the epochs rise and as a result of this, the model is able to take smaller steps and descend into areas with minimal loss (increasingly optimal) which would not have been possible with a large learning rate/step size as it would have a very high chance of missing these regions.

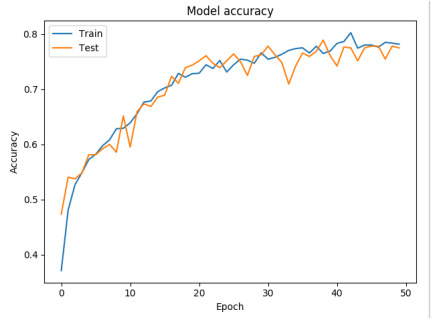
Adam Accuracy



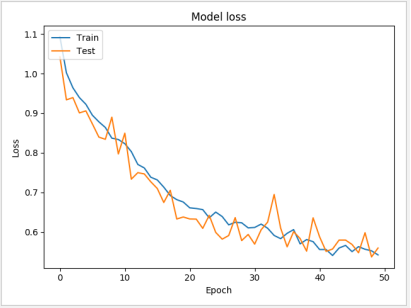
Adam Loss



Nadam Accuracy



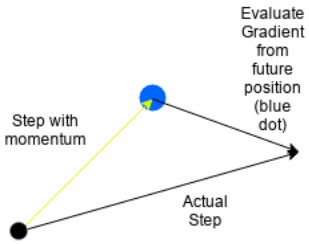
Nadam Loss



Although initially I was training all my models using SGD due to knowing how it works, I ended up trying out multiple optimisers. Two models that increased my accuracy significantly was Adam and Nadam. Looking into both these optimisers, I found out that Adam and Nadam are very similar. It is interesting to see that Adam is a combination of both SGD and RMSProp. Overall, Adaptive Moment Estimation (Adam) is an adaptive learning rate optimizer. It proved to be very powerful and managed to lower the loss significantly due to its ability to update weights based on the training data or parameters taken in. However, I will focus on the latter optimisation method since it yielded better accuracy. Nesterov-accelerated Adaptive Moment Estimation (Nadam) is essentially Adam combined with the Nestrov Accelerated Gradient (NAG). The NAG addresses the issue of a ball rolling down a slope, and gaining too much momentum (and not knowing to slow down) that it passes the minima and misses it completely. It solves this by starting with a big step, than after each step it re-calculates the gradient and based on the gradient, updates its parameters to increase the chance that the minima is not missed as shown in the diagram below.

In this diagram, the current position of the model is the black dot. We know that our current momentum will carry us to the blue dot. So while we are at the black dot, we evaluate the gradient from the blue dot and update the parameters to find our actual stepping size (the black arrow).

NAG

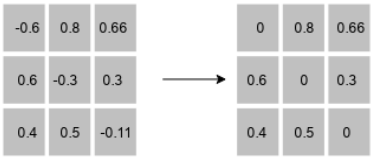


Therefore, Nadam being the combination of NAG and Adam, is a very adaptive optimisation method and significantly increased accuracy when trained with a lot of epochs as shown in the graph above.

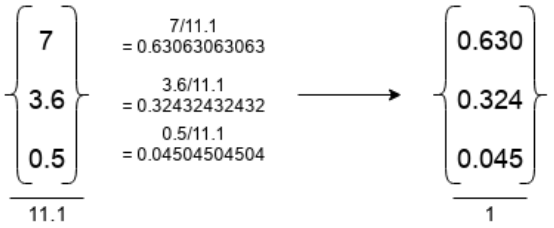
Activation Function

I used Relu for the hidden layers and Softmax for the output layer. I will be explaining Softmax in depth later on, but will touch on Relu now. The relu activation function essentially takes negative numbers and changes them to the value of 0. This is important as it means that the computations that the model has to go undergo is significantly reduced, as we do not require a lot of complex computations if the outputted value is 0.

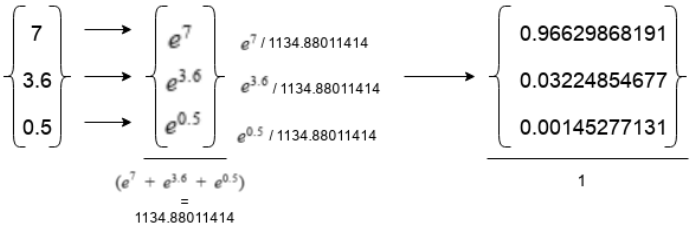
Relu activation function



To fully understand Softmax, it is easier to look at the outputs of the CNN and how we can normalise this data, as this is essentially what softmax ia achieving however with added benefits. An an example if our CNN outputted the data 0.8, 5.3, 9.2 than we could normalise this by taking the sum of these outputs, and dividing each individual output by this sum, in order to achieve the new value of of all the numbers being less than one and the sum of all the new values equaling to one. An example of this is shown below:



This can be compared to the Softmax activation function, which does a similar thing. However Softmax uses the natural logarithm number and calculated the value of e^x for each of the numbers in the output layer, where x is equal to the number in the output layer. An example of this is shown below:



As we can see by the Softmax example above, the differences between the output values after the Softmax function is applied is much larger, therefore increasing accuracy when compared to the first method. Also, using the first method (normalizing the vector) does not account for potential negative values. Negative values when normalised by this procedure can have a value that is less than 0. Another example is that if you have an output layer that contains negative values and sums upto 0, eg. (-5, -2, 4, 3) this makes normalising the vector impossible since you can not divide by the sum in this case due to it being the value of 0. Softmax efficiently solves this as since the output of the exponent function will always be positive since we are using the mathematical constant e, therefore we will always be able to divide by the sum of ($e^x + e^y + e^z + \ldots$). This is why I used Softmax for the output layer of my CNN model.

Regularization Strategy

In order to improve the accuracy of the model, we need to ensure that it does not overfit. Overfitting is when the loss is large and the model performs well on the training set, but can not perform well on data that it has not seen before, due to it learning the training data too well, it views data that it has not seen before almost as outliers. Meaning that it will not get a good accuracy on validation sets and on the unseen test data. In order to mitigate this as much as possible, I used a Dropout layer in order to stop nodes in the network from running during training. This therefore reduced the models reliance on a couple nodes in the training stage of the process and as a result of this the loss slightly dropped. I should have experimented with early stopping, however I did add a model checkpoint in order to ensure that my model was not

saving if the loss was getting worse as this would be counterproductive. Therefore my model would only be overwritten after a fully epoch if and only if the loss value decreased. I used this instead of early stopping as I was not sure when/if my model would break out of the minimums that it would get stuck at. As a result of this I did not know what to set the patience parameter to, so I decided to use a model check point instead and leave it training for a long period of time.

Hyper-Parameter Settings

Some hyper parameters that I adjusted were:

- Batch Size
- Epochs
- Decay
- Steps per Epoch + Validation Steps

I also changed parameters within the Convolutional and Pooling layers of my CNN model.

I changed the batch size between 16, 32, 64 and 128. I noticed that the higher I changed the batch size to, the longer the computation time and as a result of this I was getting very long training sessions when the batch size was greater than 32. After testing all the batch sizes and reporting accuracy I decided I would stick to the batch size of 32 and this made me get a lot of different training sessions through my day with the other parameters being changed.

In terms of epoches, Initially my strategy was that I would train my model ignoring overfitting with a great amount of epochs and with a fixed seed. Then I would look throughout the output accuracies after each epoch to find the epoch number around where the overfitting started around and set the number of epochs to that number and train again. This would make it so that the outputted model would be saved after training and would be saved just before the model started to overfit. However, I did not take into account that the model loss might just be at a local minimum. Therefore to compensate for this I implemented ModelCheckPoints and greatly increased the number of epochs (100+) in order to save the best model regardless of how many epochs. This got rid of overfitting for my later training sessions and meant I needed to set a high amount of epochs to be certain that the model was not just stuck temporarily at a local minima. However, once I started using the classifiers that were increasingly adaptive, I did not use the Decay parameter anymore.

As touched on earlier, I only adjusted the parameter of decay when using the SGD model. This was to ensure that its learning rate would decrease. This is significant because if the learning rate did not decrease over time than the model might miss regions where the loss is significantly lower and ideal.

I adjusted the steps per epoch between high and low amounts, proportionally changing the batch size whenever I did to ensure that the training time was not obscenely long. After researching about what the ideal size would be and which tradeoff would be preferable in my situation, I found a tutorial by Keras recommending that the steps should be the train_data/batch_size. Using this seemed to be a good trade off and I did not change this after this implementation.

When building my model, I added four convolutional layers. These were easy to import from Keras. For the first layer (input layer) the most important parameter that I had to set was the input_shape. This is where I adjusted the image size to (64,64,3) so that the images that the CNN use are much smaller, therefore making the training the model with them, significantly faster. I implemented a simple model in which the numbers of filters increase as another Conv2D layer was added. This is shown with the input layer only having 16 filters, the second layer having double the amount as the first with 32 filters, the third layer having 64 and finally the final layer (output layer) having a total of 128 filters. The idea behind this was that as the patterns in the model get more complex, we need additional filters to be able to capture as much of them as we can. Therefore we need to increase the filter size as we move through the layers in the CNN for the model to understand more complex patterns.

I also added Max Pooling to my CNN model with the default values of 2 x 2 as the filter size and a stride of 2 meaning that it will access each pixel in the image and not miss any. This is because the width of the stride is the same width as the filter size. Since all the pixels are accounted for, max pooling will essentially progress throughout all the pixels in the image and select the max value from within the 2 x 2 pool, then add this to the output image. After the whole image is completed we should have an outputted image that is a new representation of the same image. This image will contain all the max values from the pools that we created by defining the pool width and height, as well as the stride length. This was very handy for when I was doing a lot of epochs as because the new representation of the image is of less size (since Max Pooling only selects the largest pixels) it takes less computation time to process these smaller images. It was also handy when I was testing if there was a correlation between training with larger image sizes and classification

accuracy, as this would slowly shrink the image throughout the process and therefore make it faster to train and prevent overfitting.

Transfer Learning - ImageNet

```
model = Sequential([
    conv_base,
    Conv2D(16, 3, padding='same', activation='relu', input_shape=(75, 75, 3)),
    MaxPooling2D(),
    Conv2D(32, 3, padding='same', activation='relu'),
    MaxPooling2D(),
    Conv2D(64, 3, padding='same', activation='relu'),
    MaxPooling2D(),
    Conv2D(128, 3, padding = 'same', activation ='relu'),
    Flatten(),
    Dropout(0,2),
    Dense(512, 'relu'),
    Dense(3, activation='softmax')
])
```

For transfer learning, I implemented the InceptionResNetV2. This is a CNN that has been trained with over 1 million images from the database ImageNet. The benefits of transfer learning is that comparatively you do not need as big of a training set. This is because the model already has been trained and the weights have already been pre-trained. When running this model, I found that it took much longer than my CNN model to train. After leaving it overnight and checking on it in the morning, I noticed that the accuracy was a little under my CNN. If I had the time and resources I would have liked to train this model again with some additional tuning to the parameters, however the computation load and the computation time for doing transfer learning was obscene and it was unrealistic to do this at this point. It was interesting to note that the transfer learning model I selected had the minimum requirement of having images being at least 75 x 75 pixels in size.

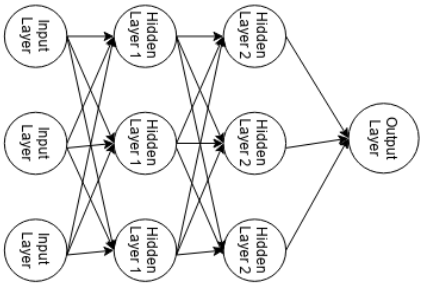
RESULT DISCUSSIONS (MLP vs CNN)

Model Type	Accuracy	Validation Accuracy	Validation Loss
CNN	**0.85**	**0.733**	**0.9**
MLP	**0.333**	**0.333**	**1.04**

Looking at the results between my MLP and my CNN, it is clear to see that the model that I built performs much better. An observation that I made when comparing the two models was that the CNN actually trained with an epoch much faster. Thinking about this, there are many reasons why this could be. One reason was because my CNN model had been tuned with the appropriate parameters. An important one could be the Max Pooling that I discussed earlier, in which the largest pixel values from the image are taken into another output, effective shrinking the image as well as representing the image differently. Since the image has smaller dimensions after Max Pooling this could get why it is faster than the MLP, as MLP can not use Max Pooling.

Another noticeable comparison is that the MLP has much less accuracy than the CNN model. Doing some research into this, I found that a reason why this could be is due to the nature of the MLP, being a fully connected model. This means that each neuron in the input layer is connected to every neuron in the next layer, and so on until we reach the output layer.

MLP, neurons are fully connected



The disadvantages of this I believe are reflected in the accuracy of the model. MLP being a fully connected model means that there is much more parameters for the model to process. The reason that this would lower the accuracy and make the model less efficient because some of the neural nodes might become redundant. From this I would assume that

it is very hard to get a model that only implements the MLP structure, that is very deep as it would likely be very inefficient. Another reason why the MLP is at a disadvantage when compared to my model is because the MLP is not Translation-Invariant. This means that the MLP is good at recognising images based on their location in the image. However, if the features are shifted around the image than the MLP struggles to identify them which is not good for the classification accuracy. In order to mitigate this, we would need to train multiple MLP models for different locations, or include a high amount of variance in the training set so that the MLP is increasingly trained on each potential location.

The CNN model has higher accuracy, and less loss. This makes sense, since the reasons why the MLP has lower accuracies and higher amount of overfitting is solved by the feedforward Convolutional Neural Networks. Since the model has the ability to have either fully connected layers such as Dense layers, or layers that are not fully connected as well. Therefore improving computation time as well as ensuring that the neurons in the layer do not become redundant. The way that the CNN works as well ensures that the model is looking for features within the whole image and not just a certain spot like the MLP. This means that it is Translation-Invariant.

CONCLUSIONS

My model that I built fluctuated between the range of 70-75% accuracy when tested on the validation set. On the test images that we were given my model outputted an accuracy of 73.3% which I was happy with. I found that the MLP Baseline performed poorly compared to my CNN model as the accuracy for my CNN was better by 52%. The loss was better by in my model as well by 24.2% which meant my model was less prone to overfitting as expected, since it is using tools such as Max Pooling and Dropouts which the MLP can not use. Although the MLP is good, I think that it's implementation on its own is not that good. The accuracy was capped at 0.3 from the very start of running the MLP. I am happy with my CNN model as it is able to give me good accuracy and I know that I can take a lot of different steps when building the model to prevent overfitting.

FUTURE WORK

For future work, I think that I would find it interesting to build different MLP classifiers for the different location of the feature in the image. I would also like to explore the idea of linking these classifiers so that we can distinguish which MLP classifier to run based on the image. However in terms of further work for this project I would have liked to dabble in ensemble learning as learning about it in my previous assignment makes me think that it could be one of the increasingly powerful machine learning techniques. Although I am not sure how the combination process actually works, I know that Ensemble learning is the process of combining different classifiers to work on a joint problem. I would like to explore the idea of combining different MLP's as stated above using this which could either result in great results or terrible ones. I would also like to try playing around with the parameters for my MLP more as I do not think that it's accuracy represents the max that it could achieve.

The negatives of the way that I conducted this assignment was the lack of appropriate images that I added to the dataset. Although I did add images to the dataset I do not believe these images were good for the classifier, as if the model did experience any increase in accuracy it was very small that it was almost negligible. Next time I would like to add around 3000 instances to see how this actually changes the accuracy, as I did not get to see it this time. I also think that there could be an issue with exporting grayscale images into RGB values when using the Relu activation function, however I am not too sure about this and would like to investigate this further.

I think the pros of the way that I approached this project was the method that thoroughness that I explored parameters and tuned my model. However, if I was to do it again I would definitely further explore EDA as well because although I did add 200 images, I do not think that the amount of images I changed the classification accuracy at all. I would also look into the EDA step that I wanted to try but could not due to memory restrictions. I think that this could be a very interesting idea for preprocessing and could be a big deal in future machine learning. This would prevent a critical step in the machine learning process as it would mean that way less data would need to be gathered, as we can alter instances and save them as new ones. Exploring other models on ImageNet could also yield better results, however I would need a lot of time to do this as training with these models is very computationally expensive. Finally, the last area that would be interesting to explore would be automated optimisers, in which the tuning process would be much more streamlined. Looking online for these I have found the Talos optimizer which looks like it could be interesting as it supposedly automatically tunes the hyper-parameters within Keras to find the correct amount.

References

[1]"Multilayer Perceptron (MLP) vs Convolutional Neural Network in Deep Learning", Medium, 2019. [Online]. Available: https://medium.com/data-science-bootcamp/multilayer-perceptron-mlp-vs-convolutional-neural-network-in-deep-learning-c890f487a8f1. [Accessed: 22- Oct- 2019]

[2]"Understanding of Convolutional Neural Network (CNN) — Deep Learning", Medium, 2019. [Online]. Available: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148. [Accessed: 26- Oct- 2019]

[3]"7.9: TensorFlow.js Color Classifier: Softmax and Cross Entropy", YouTube, 2019. [Online]. Available: https://www.youtube.com/watch?v=r0QvaEra0og&t=577s. [Accessed: 26- Oct- 2019]

[4]V. Bushaev, "Adam — latest trends in deep learning optimization.", Medium, 2019. [Online]. Available: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c. [Accessed: 26- Oct- 2019]

[5]"tf.keras.preprocessing.image.ImageDataGenerator | TensorFlow Core r2.0", TensorFlow, 2019. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator. [Accessed: 28- Oct- 2019]

[6]D. Schmidt, "Understanding Nesterov Momentum (NAG)", Dominikschmidt.xyz, 2019. [Online]. Available: https://dominikschmidt.xyz/nesterov-momentum/. [Accessed: 28- Oct- 2019]

[7]J. Melville, "Nesterov Accelerated Gradient and Momentum", Jlmelville.github.io, 2019. [Online]. Available: https://jlmelville.github.io/mize/nesterov.html. [Accessed: 28- Oct- 2019]

[8]S. Ruder, "An overview of gradient descent optimization algorithms", Sebastian Ruder, 2019. [Online]. Available: https://ruder.io/optimizing-gradient-descent/. [Accessed: 29- Oct- 2019]

[9]J. Brownlee, "Understand the Impact of Learning Rate on Neural Network Performance", Machine Learning Mastery, 2019. [Online]. Available: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/. [Accessed: 28- Oct- 2019]