CS4287: Project 2

Natural Dataset

Erona Aliu - 18228402 - John O’Keeffe - 18249531

1. Data Exploration

1.1 Dataset - Natural Image Dataset (Kaggle)

The first steps of the project involved choosing a dataset that is best suited to our project requirements. The most interesting dataset we found was the Natural Images Dataset.[1] This dataset contains 8 classes: airplane, car, cat, dog, flower, fruit, motorbike and person, each containing 600+ images.

Through some manual inspection, we found that these images seemed reliable enough in terms of color, size and quality. Based on this evaluation, we found this was a good dataset to carry on with learning more about ResNeT, Transfer Learning and Machine Learning as a whole.

The images within these classes started off as follows prior to preprocessing as seen in *Figure 1.1.1*.

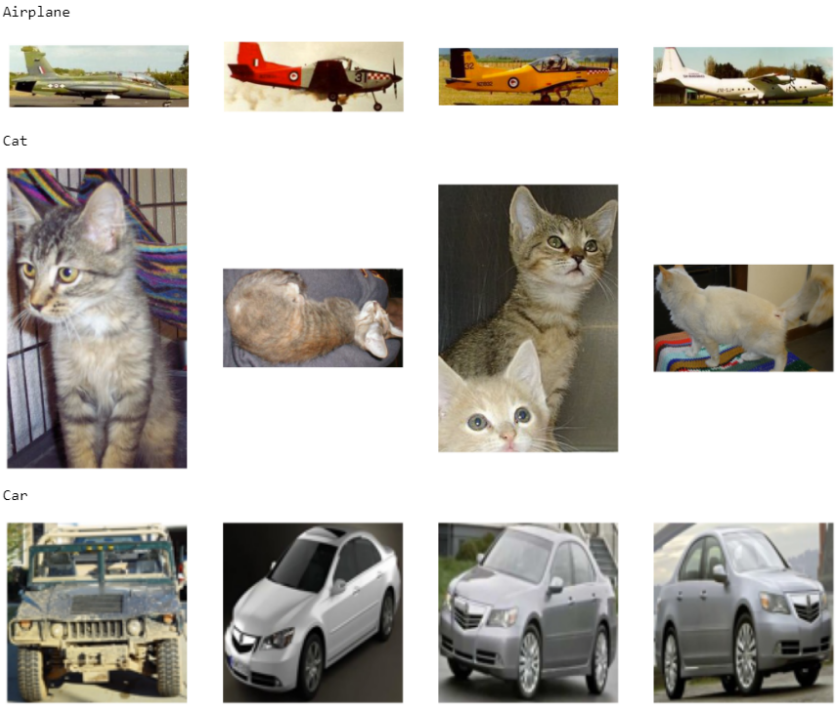


Figure 1.1.1

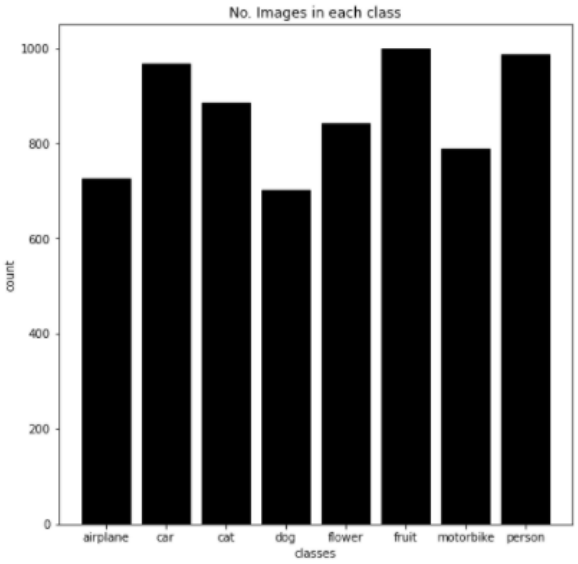


Figure 1.1.2

In *Figure 1.1.2* we have the number of images in each class represented by a bar chart. We can see that we have a substantial amount of images to work with in each class and that this dataset has enough images for us to train a model with.

1.2 Preprocessing

1.2.1 Resizing Images

We resized the images to 50x50 in order to maintain consistency. Since our network receives inputs of the same size, all images need to be resized to a fixed size before inputting them into the CNN. The larger the fixed size, the less shrinking required, thus less deformation of features and patterns inside the images. Also, models train faster on smaller images[2] so we kept them relatively small at 50x50 pixels. We used the OpenCV library in order to achieve this. OpenCV is a library for computer vision, machine learning and image processing.[3]

```
image_resized = cv2.resize(image, (50,50))
```

2. Network Structure and Architecture and other HyperParameters

2.1 CNN Architecture - ResNeT

In recent years, neural networks have become much deeper with networks going from a minimal number of layers (e.g. AlexNet, LeNet, VGGNet) to over a hundred layers (ResNet).
Residual Learning was introduced by He et al. from Microsoft Research in the paper titled “Deep Residual Learning for Image Recognition”.[5] From observing the existing neural networks at the time, they found that accuracy does not come with simply adding layers to the network in order to make it better at image recognition.

AlexNet → 8 Layers
VGGNet→ 16 Layers
GoogleLeNet→ 22 Layers

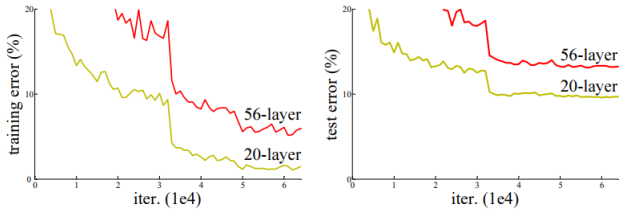
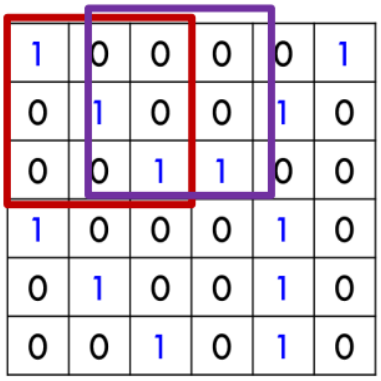


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Figure 2.1.1

If we were to use either of the networks above to train our model and we wanted to add an extra layer in order to increase the performance/accuracy of the model, we would run into what’s known as the “degradation problem” [6] *(Figure 2.1.1)* [5]. This is seen when we increase the depth of a network, it will lead to a decrease in performance on both test and training data.

The degradation problem introduces a situation where we have a set of feature maps in a CNN constantly being downsized which will eventually be too small to downsize further. However, you can still continue to add layers and perform convolutions while maintaining dimensionality by using padding with a stride of 1. *(Figure 2.1.2)*



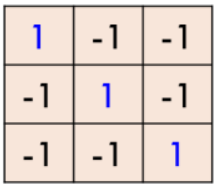


Figure 2.1.2

If we were to add another layer to an already sufficiently deep model, the next layer should aim to be an exact copy of the previous block which is known as identity mapping[7]. This is because the model has already calculated some strong features. However, *Figure 2.1.1* suggests that there is difficulty in learning this identity mapping. Thus, adding extra layers is deemed inefficient and we must opt for a new solution. This is where “Deep Residual Learning for Image Recognition” comes in.

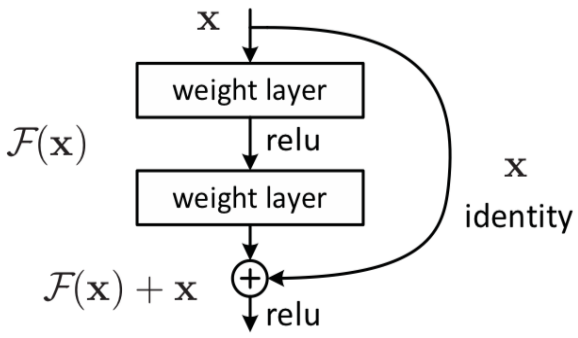


Figure 2.1.3

The researchers at Microsoft shine a light on designing the ability to simply skip this identity function learning instead and pass forward the information with a skipped connection *(Figure 2.1.3)*.[8] In order to keep learning, the application of a convolution to the feature maps is introduced, however this time we just add it's old values of residual information. The residual learning is the identity map and you can add any further learning features on top of that.[9]

It is noted by the researchers, however, that in reality, "it is unlikely that identity mappings are optimal, but our reformulation may help to precondition the problem".[5]

In summary, the residual learning architecture allows us to create very deep learning models with over 150 layers without running into the degradation problem. This is accomplished through passing information forward via the addition of a skipped connection if previous layers are blocked from adding more layers. It is done through adding a Residual Block *(Figure 2.1.3)* which allows information to flow more easily from one layer to the next. The vanishing gradient problem is another one that is prone to occur as we increase layers, the gradient becomes too small to allow the network to evolve.

On an extra note, ResNet is originally trained on the ImageNet dataset and uses transfer learning, so it is possible to load pre-trained convolutional weights and train a classifier on top of our model.[10]

In conclusion to this section, we found that, based on our research and university learning, the ResNeT architecture was the one we want to move forward with.

2.2 Weight Initialisation

Weight initialisation is used to define the initial values for the parameters in a model prior to training on the dataset.[11] Since we are using transfer learning, we add predetermined imagenet weights to aid the ResNet model.

2.3 Batch Normalisation

This is a layer that is already included in the ResNet50 architecture so it was not something we had to add ourselves. Batch normalisation allows for every layer of the network to do learning more independently and it is used to normalize the output of the previous layers.[12] Basically, it is a technique that standardizes inputs to each layer for each mini-batch.

2.4 Transfer Learning

Transfer learning allows us to utilize the elements of a pre-trained model by reusing them in a new machine learning model *(Figure 2.4.1)*[13]. Since the ResNet Architecture is built on top of ImageNet, we can load pre-trained convolutional weights and train our model on top of it. Refer to CNN Architecture - ResNet above for more information about ResNet and Residual Learning.

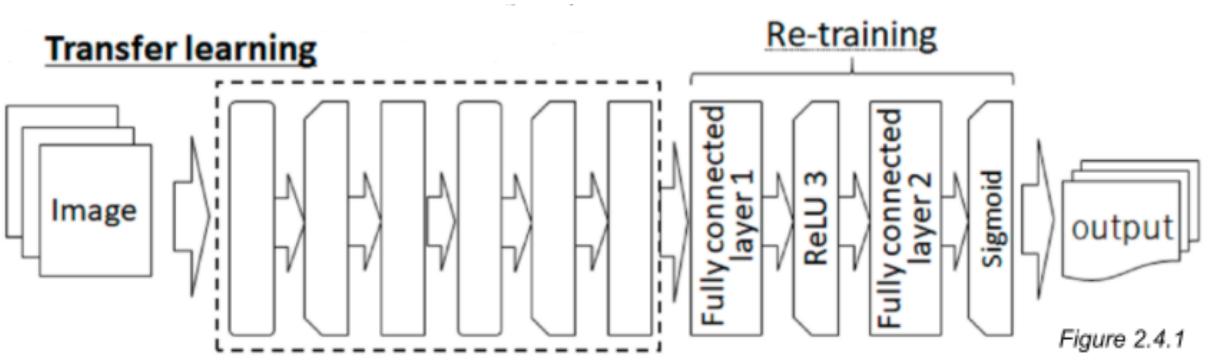


Figure 2.4.1

```
resnet = ResNet50(weights='imagenet', include_top=False, input_shape = (32, 32, 3) )
```

We load in the convolutional weights that are trained on the ImageNet data.

2.5 Layers

2.5.1 Layers

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_4 (Sequential)    (None, 50, 50, 3)         0
_________________________________________________________________
resnet50 (Functional)        (None, 2, 2, 2048)        23587712
_________________________________________________________________
flatten_2 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 65544
=================================================================
Total params: 23,653,256
Trainable params: 23,600,136
Non-trainable params: 53,120
_________________________________________________________________
```

Figure 2.5.1.1

```
resnet = ResNet50(weights='imagenet', include_top=False, input_shape = (50, 50, 3))
model = tf.keras.models.Sequential()
model.add(data_augmentation)
model.add(resnet)
model.add(tf.keras.layers.Flatten())
model.add(Dense(8, activation="softmax"))
```

Figure 2.5.1.2

2.5.2 The Sequential Layer

The sequential layer is the easiest way to build a model in keras and it allows us to build a model layer by layer.[14] We use the .add() function as seen in *Figure 2.5.1.2.*

2.5.3 The ResNet50 Layer

The ResNet50 layer is a CNN that is 50 layers deep. We used this as opposed to ResNet34 since it has a higher accuracy due to replacing each of the 2-layer blocks in ResNet34 with a 3-layer bottleneck block.[15] ResNet50 *(Figure 2.5.3.1)*[16] is also offered in keras which makes it very easy to train a model through a few simple steps.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 2.5.3.1

2.5.4 The Flatten Layer

This allows us to convert the data into a 1-dimensional array for inputting it to the next layer *(Figure 2.5.4.1)*[17]. It is connected to the fully-connected layer (dense layer).

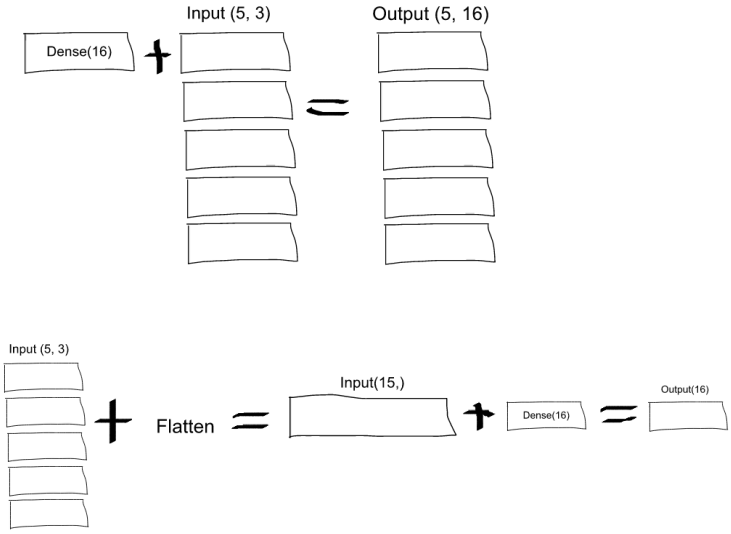


Figure 2.5.4.1

2.5.7 The Dense Layer

The dense layer, also known as the fully-connected layer, is the deeply connected neural network layer. From what we have seen, it is one of the most commonly and frequently used layers, and for good reason. The dense layer allows for the connection of the neurons inside it to connect to every neuron in the preceding layer. It is used to classify images based on output from the convolutional layers.[18]

2.6 Categorical Cross Entropy

Categorical cross entropy is a loss function that is used in multi-class classification tasks. These are tasks where an example can only belong to one out of many possible categories, and the model must decide which one.
We normalize the data by dividing it by 255 and then we use the label encoded to convert the classes to numerical data from 0 to 7 (y_encoded). *(Figure 2.6.1)*
The to_categorical function converts the class vector to a binary class matrix for use with categorical cross entropy.

```
# standardizing the input data
x_data = x_data.astype('float32')/255
```

```
# converting the y_data into categorical:
y_encoded = LabelEncoder().fit_transform(y_data)
y_categorical = to_categorical(y_encoded)
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 0., 0., 1.]], dtype=float32)
```

Figure 2.6.1

2.6 The Optimizer

Optimizers are used to update weights and biases. In more detailed words: the internal parameters of a model to reduce the error. We used Adam as we could visibly see from our experimentation that it produced a high accuracy output and our model converges comfortably at 95.83% after running 20 epochs.

3. Results

To conclude our results, we get an accuracy value above 0.9 each time we run the model on the data. We used categorical cross entropy as the loss function.

3.1 Top K Accuracy

The top K accuracy gives our model's top K highest probability answers that match with the expected answer *(Figure 3.1.1).* It considers classification correct if any of the K predictions match with the expected accuracy. In order to calculate this, keras provides this metric. *(Figure 3.1.2)*.

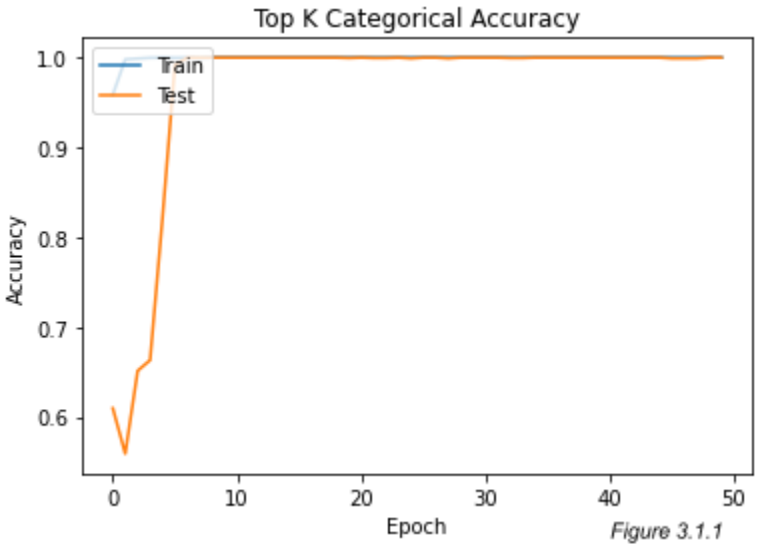


Figure 3.1.1

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005),
              loss='categorical_crossentropy',
              metrics=['accuracy', keras.metrics.Precision(name='precision'), keras.metrics.Recall(name='recall'),
                       tf.keras.metrics.TopKCategoricalAccuracy(name='top_k_categorical_accuracy')]
             )
```

Figure 3.1.2

3.2 Precision & Recall

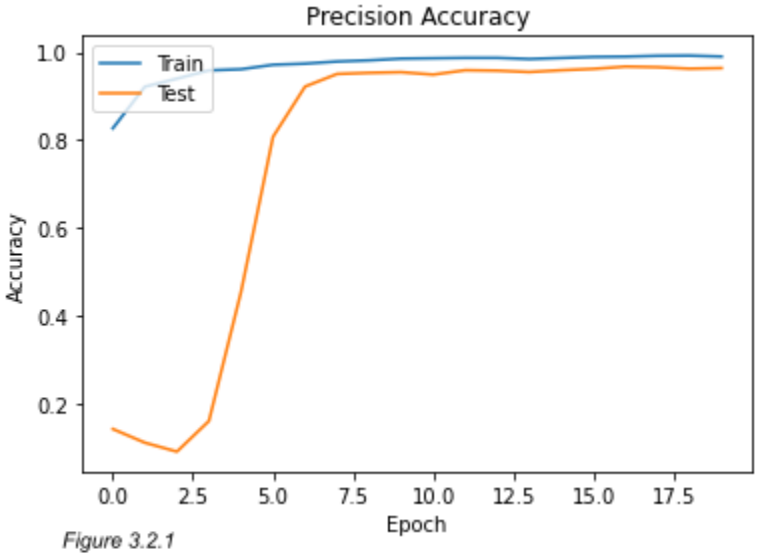


Figure 3.2.1

Precision quantifies the number of positive class predictions that actually belong to the positive class.

Recall quantifies the number of positive class predictions made

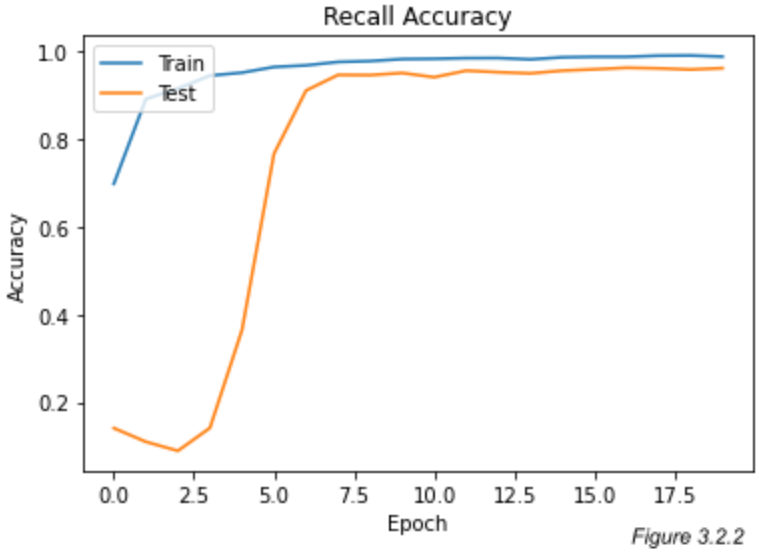


Figure 3.2.2

We can see from Figure 3.2.1 and Figure 3.2.2. that the accuracy in both retrospectives improves exponentially prior to levelling off.

4. Evaluation of Results

We used a confusion matrix in order to see how the model performs on the data *(Figure 4.1)*. We can see here that the model often confuses airplanes with flowers. We think this could be due to the colouring of the images.

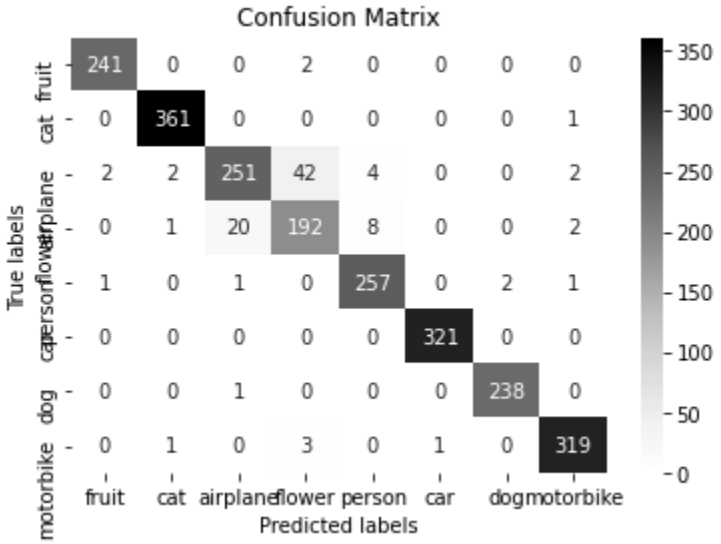


Figure 4.1

We used the Adam Optimizer as it provided the most accurate results as it is a stochastic gradient descent method. However, it is much faster than SGD and converges well.

After 20 epochs we noticed that the model converges comfortably at 95.83% accuracy which is very impressive.

```
loss, acc, prec, recall, topk = model.evaluate(X_test,  Y_test, verbose=2)
print("Trained model, accuracy: {:5.2f}%".format(100*acc))
```

```
72/72 - 1s - loss: 0.1752 - accuracy: 0.9583 - precision: 0.9595 - recall: 0.9578 - top_k_categorical_accuracy: 1.0000
Trained model, accuracy: 95.83%
```

5. Impact of Varying Hyperparameters

We noticed our graphs were very jagged *(Figure 5.1)*. The test accuracy trails along with the train accuracy with the exception of some extreme dips in the graph. Through some research online[19] we found that these dips are the cause of overfitting.

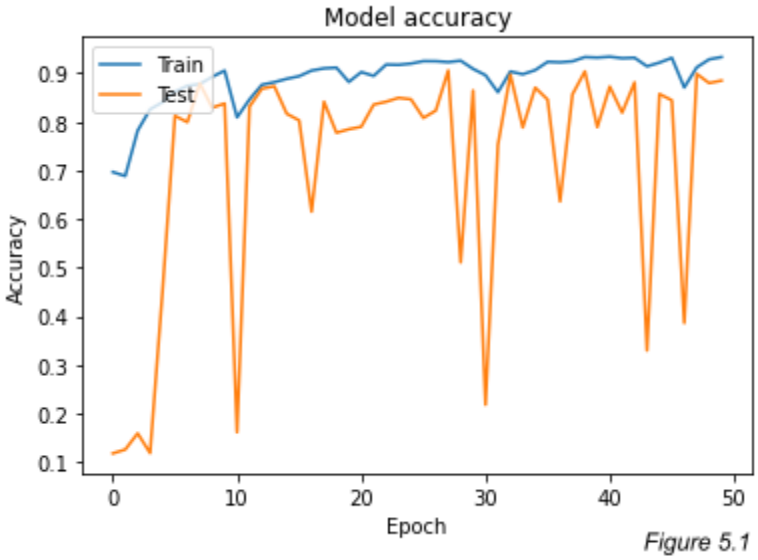


Figure 5.1

We confirmed our data was overfitting as the accuracy sprung up to 100% after instantiating a large number of epochs (e.g. 100). However, the test accuracy was also quite high but we still concluded that there was some overfitting going on. Because of this, we had to perform some data augmentation as well as lowering the learning rate.

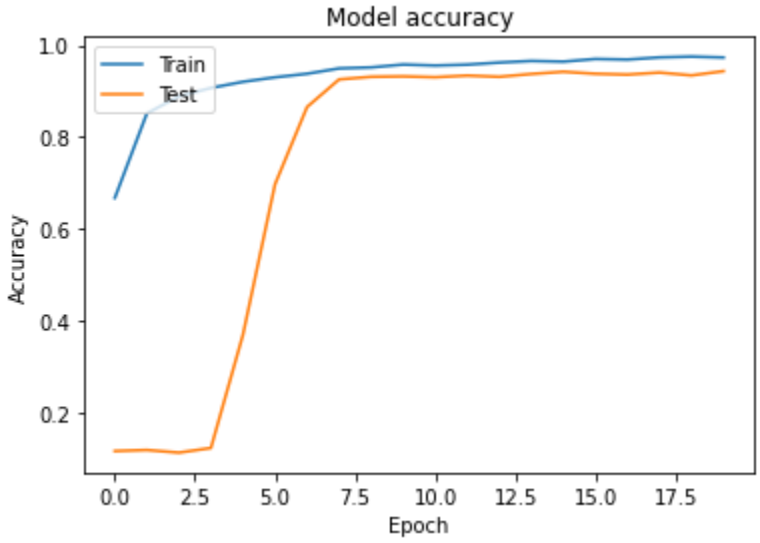


Figure 5.2

Augmentation is done through slightly altering the existing data. We add the augmentation as a layer in the model *(Figure 5.3)* so that it rotates the images horizontally and vertically. We also lowered the learning rate which improved the graph significantly as seen in *Figure 5.2*.

```
data_augmentation = tf.keras.Sequential([
  tf.keras.layers.RandomFlip("horizontal_and_vertical"),
  tf.keras.layers.RandomRotation(0.4),
])
```

Figure 5.3

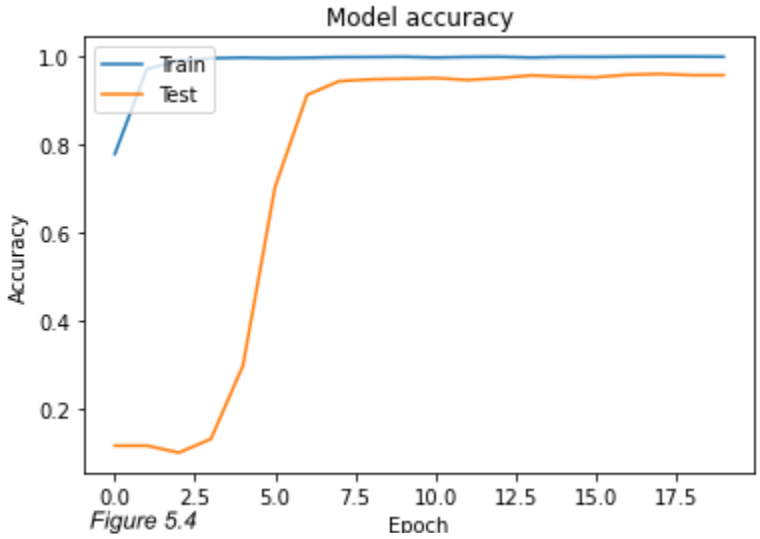


Figure 5.4

We noticed that when we lowered the learning rate and removed data augmentation (Figure 5.3), it did not change the graph much. We can conclude here that varying the learning rate was of much more convenience than adding data augmentation to deal with our overfitting issue.

References

[1] Kaggle Natural Dataset
https://www.kaggle.com/prasunroy/natural-images
[2] You Might Be Resizing Your Images Incorrectly
https://blog.roboflow.com/you-might-be-resizing-your-images-incorrectly/#:~:text=Principally%2C%20our%20machine%20learning%20models,and%20that%20time%20adds%20up.
[3] OpenCV resize image
https://pythonexamples.org/python-opencv-cv2-resize-image/
[4] How to Configure Image Data Augmentation
https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/
[5] “Deep Residual Learning for Neural Networks”
https://arxiv.org/pdf/1512.03385.pdf
[6] Degradation Problem
https://towardsdatascience.com/resnet-the-most-popular-network-in-computer-vision-era-973df3e92809
[7] Review of Identity Mappings in DRN
https://medium.com/deepreview/review-of-identity-mappings-in-deep-residual-networks-ad6533452f33
[8] Residual learning image
https://towardsdatascience.com/resnet-the-most-popular-network-in-computer-vision-era-973df3e92809
[9] ResNet Explained
https://www.youtube.com/watch?v=RQ4sMZiciuI
[10] How to use transfer learning when developing convolutional neural network models
https://machinelearningmastery.com/how-to-use-transfer-learning-when-developing-convolutional-neural-network-models/
[11] What is Weight Initialisation in Neural Networks
https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/#:~:text=Weight%20initialization%20is%20a%20procedure,of%20the%20neural%20network%20model.
[12] Everything you should know about dropouts and batch normalization
https://analyticsindiamag.com/everything-you-should-know-about-dropouts-and-batchnormalization-in-cnn/#:~:text=Batch%20normalization%20is%20a%20layer,the%20input%20layer%20in%20normalization.
[13] TF image
https://www.mdpi.com/2073-4441/12/1/96
[14] Sequential Layer
https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5
[15] ResNet50
https://viso.ai/deep-learning/resnet-residual-neural-network/
[16] ResNet18,34,50,101,152 Image
https://neurohive.io/en/popular-networks/resnet/
[17] Flatten Image
https://stackoverflow.com/questions/43237124/what-is-the-role-of-flatten-in-keras
[18] Dense Layer
https://machinelearningknowledge.ai/keras-dense-layer-explained-for-beginners/
[19] Stack overflow research for jaggedy graph

https://datascience.stackexchange.com/questions/75157/training-accuracy-is-97-but-validation-accuracy-is-stuck-at-40

Extra

Intuition Behind Residual Learning
https://towardsdatascience.com/intuition-behind-residual-neural-networks-fa5d2996b2c7
ResNET with Transfer Learning
https://medium.com/swlh/resnet-with-tensorflow-transfer-learning-13ff0773cf0c
Intro to Convolutional Neural Networks using TensorFlow
https://towardsdatascience.com/introduction-to-convolutional-neural-network-cnn-de73f69c5b83
Recall and Precision
https://machinelearningmastery.com/precision-recall-and-f-measure-for-imbalanced-classification/#:~:text=Precision%20quantifies%20the%20number%20of,positive%20examples%20in%20the%20dataset.