CS4287 NEURAL COMPUTING

Assignment 1: Convolutional Neural Network

Overview

For this project, we were tasked with implementing a Convolutional Neural Network using one of the popular CNN architectures. We made the decision to use ResNet to try and train our CNN to be able to recognize different species of monkey.

By: Luke O' Loughlin, 19231326 & Rachel O'Donoghue, 19274505

Table of Contents

The Data Set

Visualization of data

For this project, we decided that we would use the [10 Monkey Species data set from Kaggle.] Our goal is to accurately distinguish between the ten different monkey species using our Convolutional Neural Network (CNN). This dataset is divided into three directories, training, validation, and testing, with ten subfolders in each directory. These folders are labelled n0 through n9, with each number denoting a distinct species of monkey. The labels represent the following:

n0, mantled_howler
n1, patas_monkey
n2, bald_uakari
n3, japanese_macaque
n4, pygmy_marmoset
n5, white_headed_capuchin
n6, silvery_marmoset
n7, common_squirrel_monkey
n8, black_headed_night_monkey
n9, nilgiri_langur

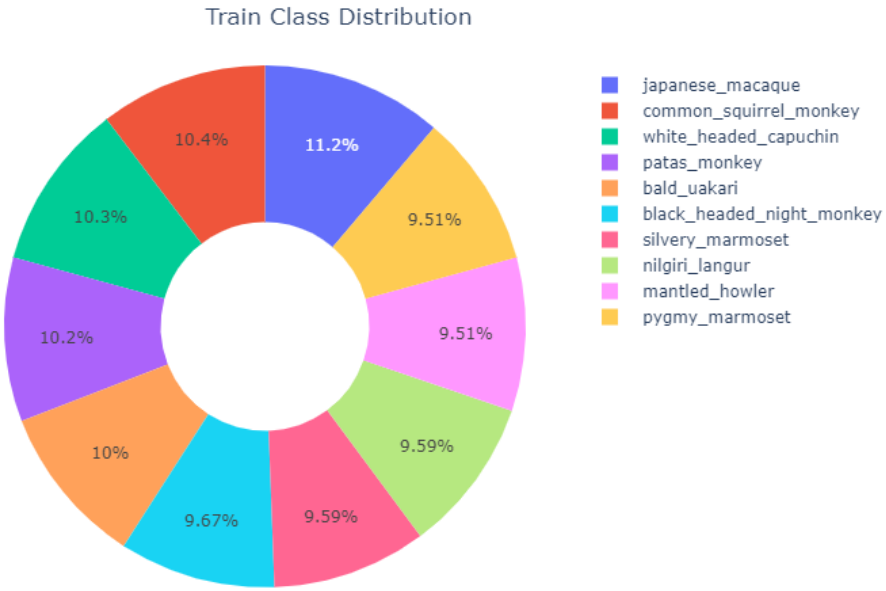


Figure 1.1: Visualisation of the percentage of images in each subfolder.

There are 1,370 total photos, of which 948 are in the training set, 222 are in the validation set, and 200 are in the test set. These images were only divided into sets for training and validation in the original dataset, however we choose to divide the data manually in order to create a test set.

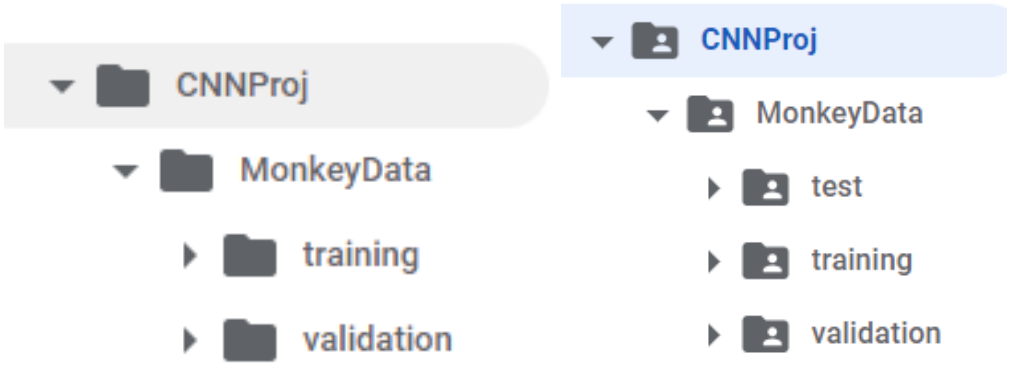


Figure 1.2: (Left) Original directory structure vs (Right) New directory structure.

	Label	Latin Name	Common Name	Training Images	Valdation Images	Test Images
0	n0	alouatta_palliata\t	mantled_howler	116	21	20
1	n1	erythrocebus_patas\t	patas_monkey	124	23	20
2	n2	cacajao_calvus\t	bald_uakari	122	22	20
3	n3	macaca_fuscata\t	japanese_macaque	137	25	20
4	n4	cebuella_pygmea\t	pygmy_marmoset	116	21	20
5	n5	cebus_capucinus\t	white_headed_capuchin	126	23	20
6	n6	mico_argentatus\t	silvery_marmoset	117	21	20
7	n7	saimiri_sciureus\t	common_squirrel_monkey	127	23	20
8	n8	aotus_nigriceps\t	black_headed_night_monkey	118	22	20
9	n9	trachypithecus_johnii	nilgiri_langur	117	21	20

Figure 1.3: Visualisation of the number of images in each directory.

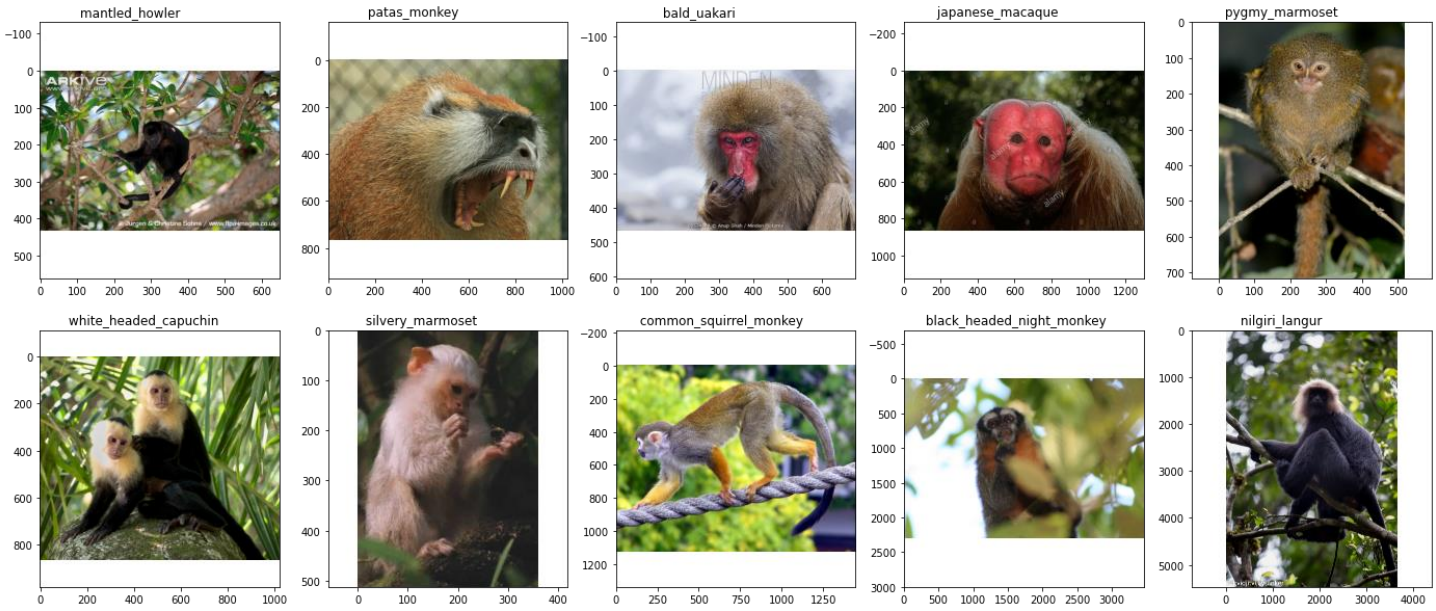


Figure 1.4: Randomly selected images from the dataset of each monkey species.

Since we are using images for our data set, there is no real correlation to be found in our data. That would be more appropriately applied to data with potential correlating patterns, like with statistics.

Feature engineering

Feature engineering is analysing raw data and extrapolating new data features or improving existing ones. Due to the nature of our data being a verified image dataset, some techniques do not apply like handling outliers or dealing with missing values, but others do [1]. Since our data is categorical and thus means nothing to a computer, we must convert it to numerical data. Since monkey species have no kind of natural ordering to them, we will be using one-hot encoding. This means that our labels will be represented by binary variables. This is done in the pre-processing step

ImageDataGenerator.flow_from_directory, in which the class_mode parameter is set to categorical. This will automatically convert the data as it is fed in, into a 2D one-hot encoded label array.

ImageDataGenerator itself is a function which allows for a lot of preprocessing or data augmentation to take place. (Take about vals it takes in)

Pre-processing

We decided to perform some data augmentation for pre-processing. Data augmentation is 'a set of techniques to artificially increase the amount of data by generating new data point from existing data.' [2] This involves making minor variations to the data or creating new data points using deep learning models.

```
# Using ImageDataGenerator to preprocess our images
# All params are my selected preprocessing I want done to my images and will be randomly applied. Most are within ranges, so anything within that range
# Keeps our data unique everytime
train_datagen = ImageDataGenerator(
    rescale=1. / 255,           # The RGB vals in an image are [0, 255] by default. This scales down the vals to be [0, 1], normalising them
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True)
```

Figure 1.5: Code snippet of using ImageDataGenerator for our data augmentation.

We chose to use ImageDataGenerator for our data augmentation because we believed it would be an easy way to produce batches of augmented data. As shown in the code above, we use ImageDataGenerator to do a number of augmentations, such as:

1. 'rescale' is the rescaling factor. If value is None or 0, nothing happens, but if values are provided, it multiplies the data in the range of that value.
2. 'rotation_range' randomly rotates the image in the range of the value given.
3. 'width_shift_range' is a floating point number that is between the values 0.0 and 1.0. This defines the maximum amount by which the image will be randomly moved, either to the left or the right, as a percentage of the overall width.
4. 'height_shift_range' is much the same as 'width_shift_range', but it moves the image up or down randomly instead of left or right.
5. 'shear_range' is the shear intensity. It warps the image along an axis, in order to construct or correct the perception angles.
6. 'zoom_range' zooms in at a random point in the range given.
7. 'horizontal_flip' randomly flips the image horizontally.

ImageDataGenerator has many other features, such as 'vertical_flip', which randomly flips the image vertically, and 'brightness_range', which picks a brightness shift value from the range given, but the features listed above are the ones we decided to go with.

The Network Structure and Hyperparameters

For neural network (NN) model we chose to use the Keras API ResNet50V2, pre-trained on the ImageNet dataset. The other main model we were looking at was VGG19, but when comparing the two, ResNet50 was superior due to its (slightly) higher accuracy for image classification[3]. ResNet50, while being a much deeper network than VGG19, also uses far less memory.

It also does not suffer badly from the vanishing gradient problem. This problem occurs in deep networks because as we backpropagate from the final layer to the first layers passing weights, with every multiplication, the smaller the new weights become. This means in a deep network by the time the NN is multiplying the weights of the first layers, they are changing such a negligible amount, it doesn't matter. Applying the chain rule to the weights, one will see the gradient decreases rapidly and leads to slow or no learning. There is a similar problem to this known as the exploding gradient problem, in which instead of the results going towards zero, they grow exponentially by multiplying through layers that have a value greater than 1. ResNet manages to reduce these issues through a myriad of features. Explicit and intentional features to address them are having intermediate batch normalisation layers and using normalised initialisation. [4]

Batch normalisation (BN) standardises the input of a layer. It has two learnable parameters which allow it to adapt as needed. For NNs, it helps by restricting the distribution of input data, helping to keep the data from creating poor gradients. This can happen since all layers are stacked one after the other which can lead to weights of earlier layers varying slightly. It is in this way that by having multiple layers of BN throughout the ResNet50V2 structure, we can reduce the vanishing gradient problem.

With the use of BN, we will not be utilising other approaches to improve generalisation (reducing overfitting) like dropout or L2 regularisation (aka weight decay). BN offers a regularisation effect in it of itself and experiences no negative effects with the exclusion of dropout.[5] BN could actually be negatively affected by the use of both in the same network because dropping out can create noisy data, which BN will be taking in, and can harm the accuracy of results. L2 regularisation has no regularising effect when combined with BN, so it is pointless.[6]

Normalised initialisation, also known as Xavier initialisation, chooses from a random uniform bounded distribution to create the weights for a layer. This helps to maintain variation of activations and back-propagated gradients throughout the network (forward and backward), reducing the chance of the vanishing point problem. It should be noted that we will be using Keras's ResNet50V2 pretrained on the ImageNet dataset for our weight initialisation, meaning we will not be using normalised initialisation ourselves. That has most likely already been done when the pretrained model was initially being trained on the ImageNet dataset.

We mentioned ResNet used explicit features to reduce the vanishing/exploding gradients, but there are some whose inclusion, while having another main purpose, does significantly reduce the problem. This includes the use of the ReLU activation function and skip connections within residual learning. Their main purpose is to help resolve the problem of the degradation of the network the deeper down the layers it goes, with the saturation of accuracy as it converges.

Rectified Linear Unit(ReLU) is the most widely used activation function in CNNs. It has major advantages over sigmoid and tanh functions; it's very easy to calculate, since it is only a comparison between the input and 0(very computationally inexpensive and fast) and it has a derivative of 0 or 1, depending on if the input was positive or negative.[7] The second point is extremely important with regards to backpropagation. Since the computation is so simple, it is extremely quick to run. This is all the more important when being implemented in Deep NNs, like ResNet50V2. Due to this, ReLU stops the exponential growth of computation needed for the CNN. If the scale grows, time taken will scale linearly, as can be seen from its graph. This case of the derivative being absolute 0 or 1 means that backpropagation and learning can always be allowed to continue, even for high values (this is also how it reduces the vanishing gradient problem), something that would otherwise heavily affect a model like ResNet50V2. This is why ReLU works so well in the deep residual learning framework.

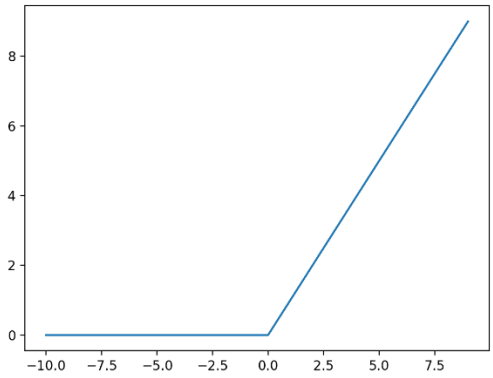


Figure 2.1: ReLU activation function

Deep residual learning is ResNet's defining feature, and what allowed it to win 1st place on the ILSVRC 2015 classification task on the ImageNet test set. The reason for its success was that it managed to significantly reduce the two main issues previously plaguing deep NNs; vanishing/exploding gradient and accuracy saturation. It did this by implementing "residual blocks".

Residual blocks are made up of something known as shortcut or skip connections. These skip connections work on the principle of identity mapping(ensuring the output of some layer is equal to the input). Normally we would pass the weight through each individual layer of our network in backpropagation, performing calculations along the way (causing the diminishing value of the gradient and saturation). Skip connection proposes we do the same calculation, but we also pass the the original weight and add it to the output before the second function is executed. This stops the degradation of results and doesn't negatively impact anything.

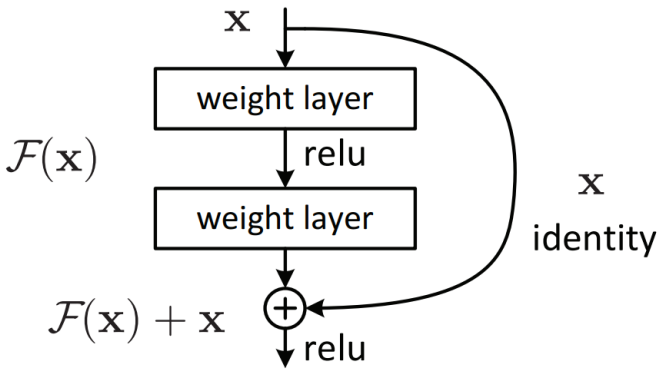


Figure 2.2: Residual block [4]

ResNet consists of two types of residual block; the Identity block and Convolutional block. [8]

The identity block is used when we our input and output have the same dimensionality. (In ResNet50 this is achieved by using padding to maintain the dimensionality beforehand)

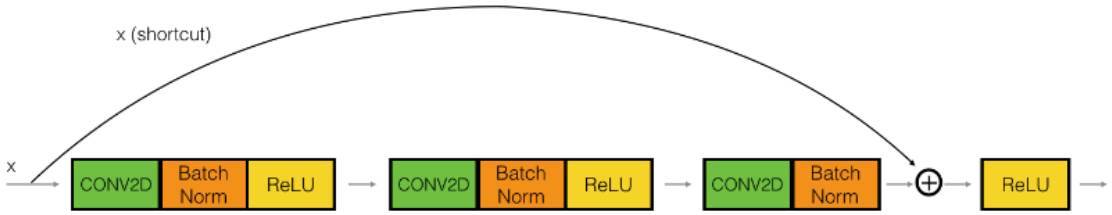


Figure 2.3: Identity Residual Block

The convolutional block is used when we our input and output have the different dimensionality and the output must be resized. This is achieved through the use of kernels like 1x1 Conv2D, padding and strides. (Conv2D and Batch Normalisation in the case of ResNet)

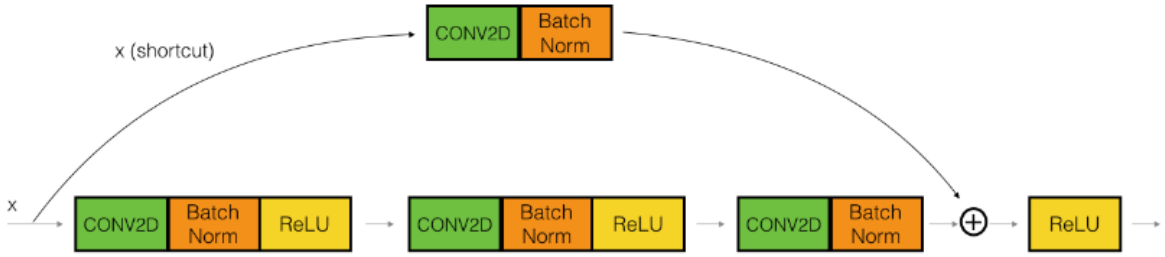


Figure 2.4: Convolutional Residual Block

Aside from the extra shortcut step, the layers withing the blocks are the same. It is first a 1x1 Conv2D, then 3x3 Conv2D and finally a 1x1 Conv2D. This is due to ResNet50 using the bottleneck design. The 1x1 convolutions decrease and the increase the dimensionality, so the 3x3 has a much smaller input and output without losing any detail. The shortcuts are very important to this design, as without them the time complexity and model are doubled. [4]

It is in terms of shortcuts that we see the difference between ResNet50V1 and ResNet50V2. Looking at figure 2.5, we can see that V1 uses post-activation, taking in a weight that has been calculated and only doing 1 more activation before addition. V2 takes in a pre-activation weight, doing the calculation within the shortcut. It also applies the BN and activation before its addition for the convolution operation.[9]

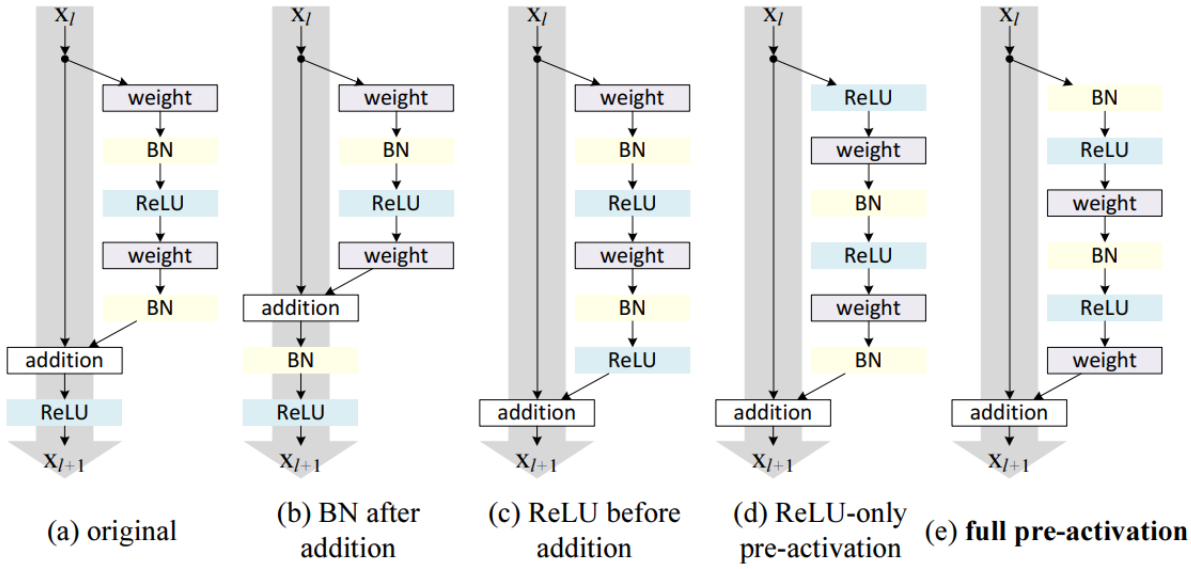


Figure 2.5: (a) ResNet50V1, post-activation. (e) ResNet50V2, pre-activation [10]

The other activation function we will see used in our ResNet model is Softmax. This is the final layer of the model, as it scales all values down to probabilities (between 0 and 1). We use this, as opposed to sigmoid, as we have multiple classes which can be the result. It is this function which the use of one hot encoding, mentioned in feature engineering, comes in as crucial. It needs values of this type to be able to output a result. Softmax performs Softmax normalisation on the data passed in. This defers from standard normalisation in the way it handles larger and smaller changes differently. It also ties in very well with our loss function: Categorical Cross-Entropy.

As mentioned, we are using the Keras ResNet50V2 model, pre-trained on the ImageNet dataset. We decided to go with the pre-trained model as we felt due to the small size of our dataset and with our choice of monkey species that it would be ideal in giving us a higher chance of better identifying our data. We found that ImageNet not only contains general monkey images, it specifically has a few of the breeds our dataset includes too, like patas, macaque and howler monkeys.[11] This guaranteed too us that this would be the perfect dataset for our model to be pretrained on. This method of using a pre-trained model is known as transfer learning.

"Transfer learning and domain adaptation refer to the situation where what has been learned in one setting … is exploited to improve generalisation in another setting" [12] The idea of transfer learning is when one takes a model that has been trained on some data and uses it as a starting off point for another model that is tangentially related. Our dataset of monkeys is the perfect example. Our ResNet is using the ImageNet dataset. This is a collection of quality controlled, human-annotated images divided into 1000 classes. This means the model will be easily able to identify a monkey, and the patterns and features that make up a monkey. However, our dataset is more explicit than "monkey", requiring our model to find even more precise patterns. Top quality models can take days to train on this data. It is these models which we will be taking advantage of, specifically ResNet50V2. Due to the sheer scale of the images being processed, the models become extremely good at quickly identifying key features and making accurate predictions.

This comes with many potential advantages including a higher start, higher slope and higher asymptote. Our ResNet50V2 model will definitely benefit from a higher start, being able to identify the entity it is seeing at the very least. We would hope that the improvement of ability during training and where we would converge would be better, due to having more previous data about

features and the like but that will remain to be seen in training. This also leads to being to implement something on from this; network fine-tuning.

Network fine tuning in our model will be setting all the pre-trained layers of our ResNet50V2 model to be false, freezing them. This will mean we will not have to retrain the entire layer (layers in our case), making the amount of data required for training quite small compared to what it otherwise would be. It also means since far fewer parameters need to be updated a huge amount of time is saved when training. Without freezing ResNet50V2 layers would have around the full 24 million params as trainable, as can be seen below in figure 2.6. This all works on the premise of the CNN's initial layers being more general, being able to pick out broad stroke features (edges, outlines), and deeper down the layers, the more specific it gets with identification (eyes, mouth, tail, etc). In terms of our ResNet50V2, our "early" layers are all pre-set with weights, ideal for the task at hand. Freezing them at the beginning is ideal, allowing us to only have our final layers learning. [13]

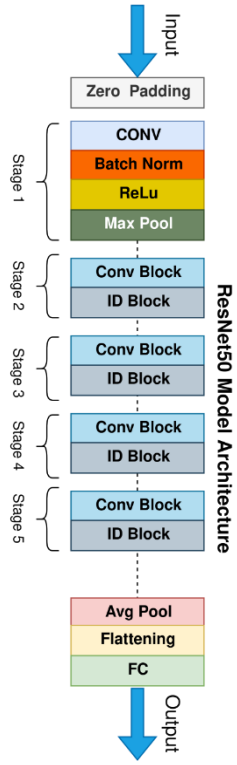
Here is an overview of some of our hyperparameters and there initial values

- Epochs: number of times the algorithm runs through the dataset. We chose 50 epochs
- Batch size: size of batches fed to ResNet50V2 model. We chose batch size of 32
- Image size: size of images fed to model. We chose image size of 224, as ResNet requires it
- Patience: used in early stopping to determine the tolerance before stopping. We chose 4

There are others like our optimiser, activation function, loss function, regularisation, pooling type which have or will be discussed elsewhere more in depth.

```
conv5_block3_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_1_bn[0][0]']
n)

conv5_block3_2_pad (ZeroPaddin  (None, 9, 9, 512)   0           ['conv5_block3_1_relu[0][0]']
g2D)

conv5_block3_2_conv (Conv2D)   (None, 7, 7, 512)    2359296     ['conv5_block3_2_pad[0][0]']

conv5_block3_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block3_2_conv[0][0]']
ization)

conv5_block3_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_2_bn[0][0]']
n)

conv5_block3_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block3_2_relu[0][0]']

conv5_block3_out (Add)         (None, 7, 7, 2048)   0           ['conv5_block2_out[0][0]',
                                                                 'conv5_block3_3_conv[0][0]']

post_bn (BatchNormalization)   (None, 7, 7, 2048)   8192        ['conv5_block3_out[0][0]']

post_relu (Activation)         (None, 7, 7, 2048)   0           ['post_bn[0][0]']

avg_pool (GlobalAveragePooling  (None, 2048)        0           ['post_relu[0][0]']
2D)

flatten (Flatten)              (None, 2048)         0           ['avg_pool[0][0]']

dense (Dense)                  (None, 256)          524544      ['flatten[0][0]']

dense_1 (Dense)                (None, 128)          32896       ['dense[0][0]']

dense_2 (Dense)                (None, 10)           1290        ['dense_1[0][0]']

==================================================================================================
Total params: 24,123,530
Trainable params: 558,730
Non-trainable params: 23,564,800
```

Figure 2.6: Last layers of my ResNet model and parameter count



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 2.7: (Left) Simplified view of ResNet50V2 architecture[9]. (Right) Table showing the structure of ResNet[4]

Our model is a ResNet50V2 with a single dense layer at the end containing Softmax. When initialising ResNet50V2, we set two important parameters; include_top and pooling. We set include_top to be false since we will be passing in our own input data (or else it would use the ImageNet data). We set our pooling as average, meaning we will be taking the average of the area our kernel is applied to and will shrink down the size of the image drastically. This is used over something like max (much more common) as it applies much better for ResNet, a deep NN, due to wanting to collapse the representation.

The overall structure of of ResNets are all very similar, as can be seen in figure 2.7 on the right, but here I will be specifically referring to ResNet50V2.

"Stage 0": We take the input data and perform zero padding on it. This is adding zeroes around the edge of the image to stop it from losing its dimensionality

Stage 1: We perform a 7x7 convolution, BN, ReLU activation function and max pooling.

Stages 2 - 5: From here we start getting into the residual blocks. You can see in the figure 2.7x the two block types, discussed earlier, in each and the number of layers and blocks there are per stage. Every stage has only 1 3x3 convolutional residual block, all the rest are identity residual blocks. Remember, each block consists of Conv2D, BN, activation (ReLU) and zero padding layers.

"Stage 6": After going through the final activation (the last of the 50 layers), we apply Global Average Pooling(GAP) to the data. This creates one feature map for every corresponding category of the classification task in the last multilayer perceptron convolutional layer[14][15]. Due to this, there is no need for the use of additional layers which would be traditionally used at the end of CNNs like flattening and dense layers (creating a fully connected multiplayer perceptron). The GAP even manages to reduce overfitting since there are no adjustable parameters in this layer, unlike dense

layers. Lastly, the values are passed into the final layer, a dense Softmax activation function which will return the likely it believes every class to be the image it was given.

The Cost/Loss/Error/Objective Function

For this part of our project, we analyzed a few different loss functions to determine which would best fit our CNN. A loss function analyses how effectively a neural network models the training data by comparing the target and predicted output values. During training, our aim is to reduce the loss between the predicted and desired outputs.

Initially, we compared MSE and categorical cross-entropy, two separate types of loss functions. Given the size of a classification task's decision boundary, cross-entropy is a superior measure than MSE. Although MSE does not adequately penalize incorrect classifications, it is the appropriate loss for regression when there is little difference between two values that may be predicted. [16] Additionally, if your network's output layer exhibits sigmoid linearity or Softmax nonlinearity and you want to increase the possibility that the input data will be correctly classified, cross-entropy emerges as the appropriate cost function to utilize. Due to this, we decided to go with categorical cross-entropy.

Cross-entropy is defined as:

$$L_{\text{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

Figure 3.0.1: Mathematical definition of Cross-Entropy [17]

Categorical cross-entropy is used when, for example, 'true labels are one-hot encoded, for example, we have the following true values for 3-class classification problem [1,0,0], [0,1,0] and [0,0,1].' [17]

The Optimizer

Optimizers are algorithms that are used to change the attributes, such as weights and learning rates in a neural network in order to reduce the amount of losses. When choosing one for our project, we looked through a number of different optimizers before having to decide between two: SGD (Stochastic Gradient Descent) and Adam. The SGD algorithm extends the Gradient Descent and fixes various issues that the GD algorithm had, and the derivative is calculated taking one point at a time. [18] The Adam algorithm is a stochastic gradient descent method that is based on the adaptive estimate of first and second-order moments. [19] We decided to go with Adam, as we thought it would be a better match for our CNN, as when comparing Adam and SGD, Adam is more efficient.

The Adam Optimization Algorithm, also known as the Adaptive Moment Estimation, is a combination of the 'RMSP' (Root Mean Square Propagation) algorithm and the 'gradient descent with momentum' algorithm and is an extension to the stochastic gradient descent. [20] Adam combines

the positive attributes of the RMSP and Momentum algorithms and builds on top of them, in order to give a more optimized gradient descent. In order to determine unique learning rates for each parameter, the algorithms make use of the strength of adaptive learning rates approaches. It scales the learning rate using squared gradients, similar to RMSP, and leverages momentum by using the gradient's moving average rather than the gradient itself, similar to SGD with momentum.

Adam was ultimately selected since it is effective and requires fewer adjustments for tuning.

Cross Fold Validation

When we first started doing our CNN, we originally used the holdout method. Holdout is when you divide your dataset into a 'train' and 'test' set. The test set is used to evaluate how well the model works on data that has not yet been seen, whereas the training set is used to train the model. [21] However, in order to avoid overfitting, we ended up implementing K-fold cross validation.

K-fold cross-validation is a method for dividing data that is used to gauge how effectively a model works with unknown data. This method is used to tune the hyperparameters so that the model with the best hyperparameter value can be trained.[22] It is a non-substitutional resampling technique – it uses each example for training and validation exactly once. When compared to another validation method like the holdout, it produces a lower-variance estimate of the model performance.[22] When attempting to prevent overfitting, which can happen when a model is trained using all of the data, K-fold cross-validation is a fantastic option. [22]

K-fold cross-validation can be applied using Python and the scikit learn (Sklearn) library. This approach has 10 steps, which are as follows: [23]

1. Training and test datasets are created from the dataset.
2. After that, the training dataset is divided into K-folds.
3. Out of the K-folds, (K-1) fold is used for training.
4. 1 fold is used for validation.
5. The model with a given set of hyperparameters is trained using validation data as 1 fold and training data (K-1 folds). The model's performance is documented.
6. Steps 3, 4, and 5 are repeated until each k-fold gets used for validation purpose.
7. By adding up all of the model scores determined in step 5 for each of the K models, the mean and standard deviation of the model performance are calculated.
8. Repeat steps 3 through 7 for various hyperparameter values.
9. The hyperparameters that produce the best mean and standard deviation for the model scores are chosen.
10. The model is then trained using the training data set (step 2) and the model performance is computed on the test data set (step 1). [23]

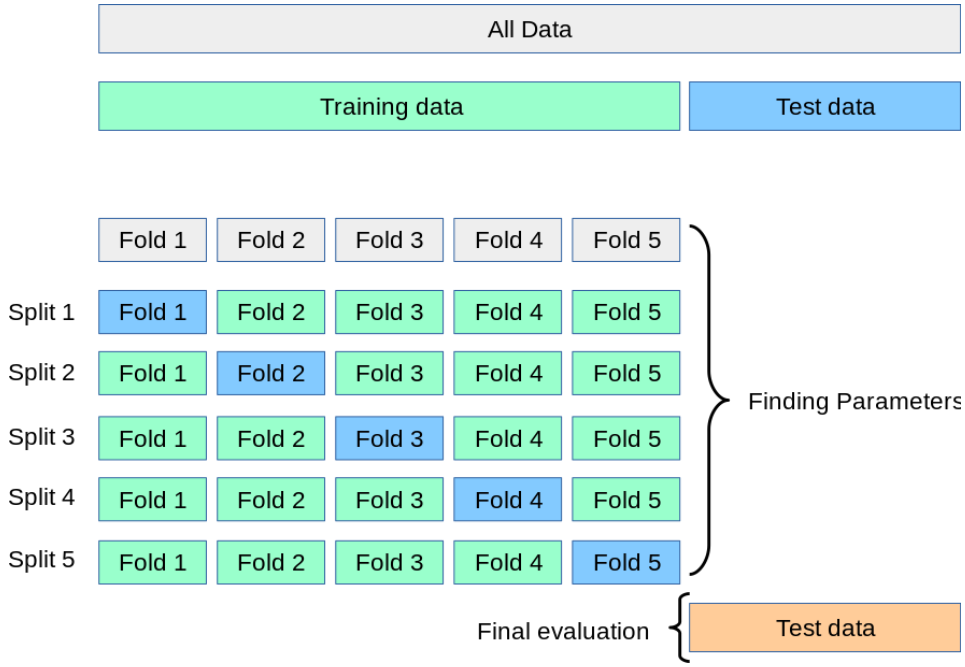


Figure 5.1: K-fold cross-validation diagram. [23]

Unfortunately, we did not manage to create any working version of cross fold validation using StratifiedKFold.

Results and Evaluation

Initial Results and Evaluation

We trained our model on Colab as it has access to GPUs which we can take advantage of for training. Each epoch took around either 35s or 200s, depending on if we had a GPU available to us to use. This is one advantage of a small dataset, as testing and retesting is very quick, relatively speaking.

```
Epoch 1/50
30/30 [==============================] - 48s 1s/step - loss: 1.5347 - acc: 0.5158 - val_loss: 0.4948 - val_acc: 0.9009
Epoch 2/50
30/30 [==============================] - 37s 1s/step - loss: 0.4028 - acc: 0.8966 - val_loss: 0.2172 - val_acc: 0.9595
Epoch 3/50
30/30 [==============================] - 38s 1s/step - loss: 0.2546 - acc: 0.9335 - val_loss: 0.1712 - val_acc: 0.9640
Epoch 4/50
30/30 [==============================] - 38s 1s/step - loss: 0.2053 - acc: 0.9525 - val_loss: 0.1410 - val_acc: 0.9595
Epoch 5/50
30/30 [==============================] - 38s 1s/step - loss: 0.1761 - acc: 0.9515 - val_loss: 0.1238 - val_acc: 0.9685
Epoch 6/50
30/30 [==============================] - 37s 1s/step - loss: 0.1434 - acc: 0.9610 - val_loss: 0.1124 - val_acc: 0.9685
Epoch 7/50
30/30 [==============================] - 37s 1s/step - loss: 0.1388 - acc: 0.9631 - val_loss: 0.0971 - val_acc: 0.9730
Epoch 8/50
30/30 [==============================] - 37s 1s/step - loss: 0.1145 - acc: 0.9705 - val_loss: 0.1004 - val_acc: 0.9730
Epoch 9/50
30/30 [==============================] - 38s 1s/step - loss: 0.1053 - acc: 0.9736 - val_loss: 0.0955 - val_acc: 0.9775
Epoch 10/50
30/30 [==============================] - 38s 1s/step - loss: 0.1036 - acc: 0.9778 - val_loss: 0.0890 - val_acc: 0.9685
Epoch 11/50
30/30 [==============================] - 37s 1s/step - loss: 0.1078 - acc: 0.9789 - val_loss: 0.0898 - val_acc: 0.9775
Epoch 12/50
30/30 [==============================] - 37s 1s/step - loss: 0.0738 - acc: 0.9842 - val_loss: 0.0857 - val_acc: 0.9775
Epoch 13/50
30/30 [==============================] - 37s 1s/step - loss: 0.0781 - acc: 0.9821 - val_loss: 0.0797 - val_acc: 0.9775
Epoch 14/50
30/30 [==============================] - 37s 1s/step - loss: 0.0849 - acc: 0.9789 - val_loss: 0.0811 - val_acc: 0.9775
Epoch 15/50
30/30 [==============================] - 39s 1s/step - loss: 0.0939 - acc: 0.9736 - val_loss: 0.0702 - val_acc: 0.9820
Epoch 16/50
30/30 [==============================] - 37s 1s/step - loss: 0.0687 - acc: 0.9810 - val_loss: 0.0589 - val_acc: 0.9865
Epoch 17/50
30/30 [==============================] - 36s 1s/step - loss: 0.0683 - acc: 0.9884 - val_loss: 0.0624 - val_acc: 0.9775
Epoch 18/50
30/30 [==============================] - 37s 1s/step - loss: 0.0402 - acc: 0.9926 - val_loss: 0.0587 - val_acc: 0.9775
Epoch 19/50
30/30 [==============================] - 37s 1s/step - loss: 0.0529 - acc: 0.9873 - val_loss: 0.0609 - val_acc: 0.9775
Epoch 20/50
30/30 [==============================] - 39s 1s/step - loss: 0.0768 - acc: 0.9778 - val_loss: 0.0573 - val_acc: 0.9775
Epoch 21/50
30/30 [==============================] - 36s 1s/step - loss: 0.0585 - acc: 0.9842 - val_loss: 0.0593 - val_acc: 0.9865
Epoch 22/50
30/30 [==============================] - 36s 1s/step - loss: 0.0614 - acc: 0.9800 - val_loss: 0.0486 - val_acc: 0.9820
Epoch 23/50
30/30 [==============================] - 37s 1s/step - loss: 0.0565 - acc: 0.9810 - val_loss: 0.0473 - val_acc: 0.9820
Epoch 24/50
30/30 [==============================] - 36s 1s/step - loss: 0.0479 - acc: 0.9873 - val_loss: 0.0592 - val_acc: 0.9730
Epoch 25/50
30/30 [==============================] - 39s 1s/step - loss: 0.0455 - acc: 0.9926 - val_loss: 0.0506 - val_acc: 0.9865
Epoch 26/50
30/30 [==============================] - 37s 1s/step - loss: 0.0566 - acc: 0.9863 - val_loss: 0.0534 - val_acc: 0.9865
Epoch 27/50
30/30 [==============================] - 37s 1s/step - loss: 0.0456 - acc: 0.9884 - val_loss: 0.0559 - val_acc: 0.9775
```

Figure 6.1: Model fit results showing accuracy and loss

These results are very promising. We believe these very accurate and quick results came as a result of taking advantage of ResNet50V2 being pretrained on ImageNet. As mentioned before, it contains multiple breeds of monkeys, some of which our dataset also contain. This contributed to the model being extremely well fitted to our data and we can clearly see the potential advantages of using a pretrained it. The starting training accuracy of 51.58% is much higher than it would be without being pretrained. The slope increases very quickly from the starting accuracy to 95.25% by epoch 4. The

asymptote also levels off at quite a level, around 98%. The same applies to the loss, and can be clearly seen in the graphs below.

The validation set initially performed better than the training data in both accuracy and score. We believe this is because the images in the training data set are modified using the pre-processing effects already discussed, whereas the validation data set is unaltered, making it easier for the CNN to distinguish key features more quickly.

Before running the model.fit function, we first set first set up early stopping. This is a call back which we used as were unsure of what the optimal number of epochs to run was. The idea is to ensure not too few or too many epochs are run, as that could lead to under and overfitting respectively. This is done by having it monitor a value, in our case the validation loss and setting the patience for it. This is the number of epochs of plateauing/backward progression allowed to happen before stopping the training. Our value for this was three and we can see it clearly worked as we stopped 27 epochs in.

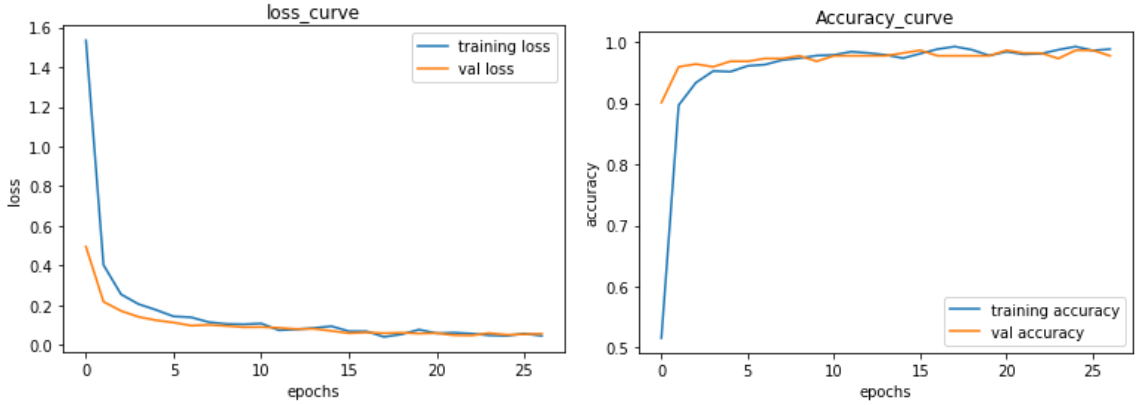


Figure 6.2: (Left) Loss curve and (Right) Accuracy curve for model

The graphs demonstrate how well the data performed. We can see the high starting point, quick learning rate and high asympotote.

We then moved to seeing how the model would do on our test set, completely new data for it.

We first used the classification_report function from the scikit learning software library, specifically within metrics. We pass it in our predicted results and true results and a report is built which contains the main classification metrics; precision, recall and f1-score for our model. Precision answers the question of "What proportion of positive identifications was actually correct?". Recall answers the question of "What proportion of actual positives was identified correctly?"[24] To properly evaluate our model, we must look at both these results. However, the issue is they are often in opposition to each other. I.e. increasing precision leads to lower recall and the inverse is true. This leaves us in a dilemma of wanting the best of both but not knowing how to get it. That is where F1-score comes into play. It is a harmonic mean of precision and recall. The F1-score ranges from 0-1 like precision and recall and it accounts for both. The closer it is to 1, the better the model.[25]

```
7/7 [==============================] - 6s 659ms/step
                                precision    recall  f1-score   support

 mantled_howler                      0.90      0.95      0.93        20
 patas_monkey                        1.00      1.00      1.00        20
 bald_uakari                         1.00      1.00      1.00        20
 japanese_macaque                    1.00      1.00      1.00        20
 pygmy_marmoset                      1.00      0.95      0.97        20
 white_headed_capuchin               1.00      0.95      0.97        20
 silvery_marmoset                    1.00      0.95      0.97        20
 common_squirrel_monkey              0.95      1.00      0.98        20
 black_headed_night_monkey           0.95      1.00      0.98        20
 nilgiri_langur                      0.95      0.95      0.95        20

                    accuracy                           0.97       200
                   macro avg        0.98      0.97      0.98       200
                weighted avg        0.98      0.97      0.98       200
```

Figure 6.3: Table showing precision, recall, F1-score and number of images for each monkey species

This was the first test our model ran on unseen data, and as you can see, it did quite well. This is the first Both precision and recall are quite good on recognising all species, meaning the F1-score is good too.

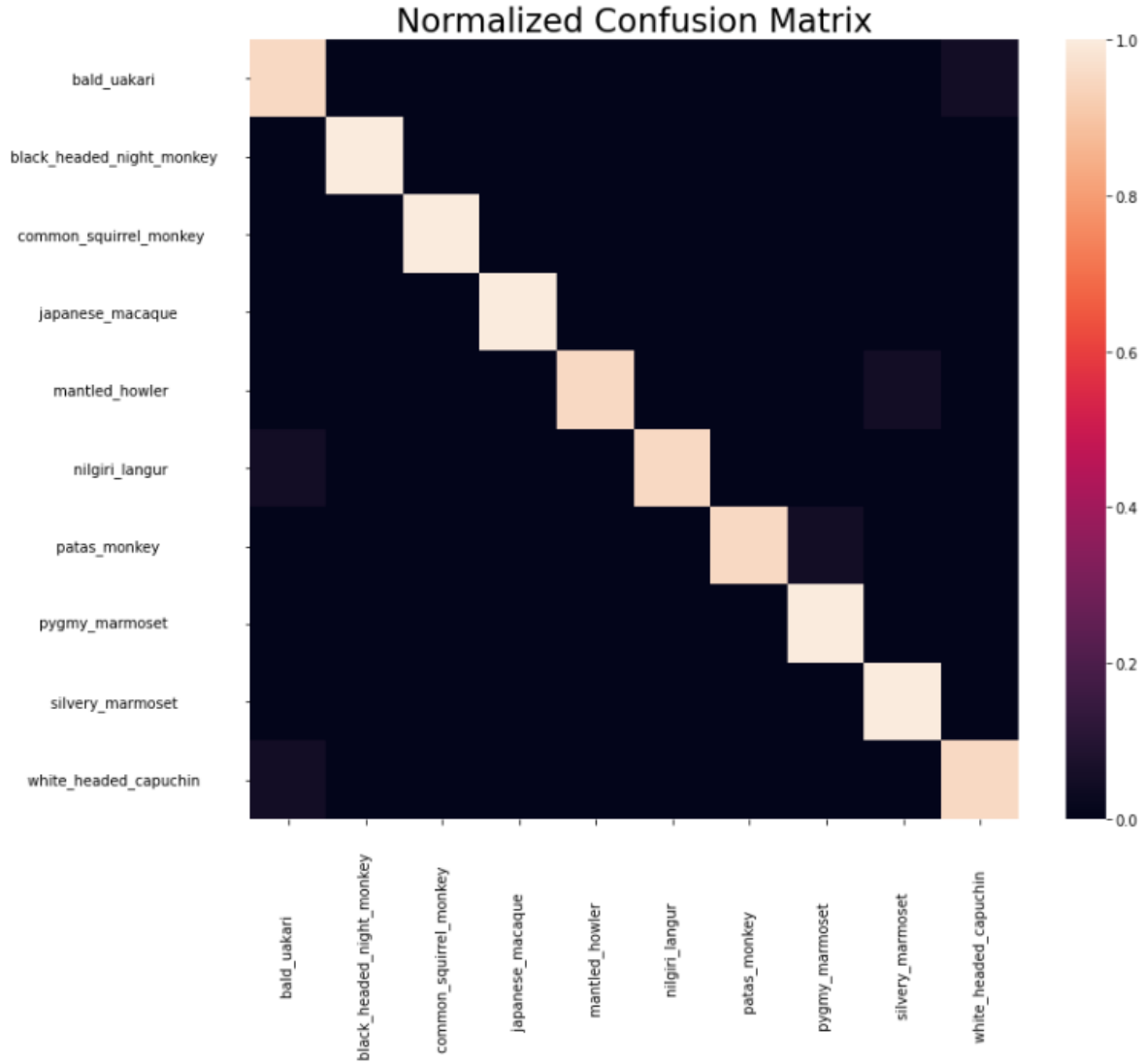


Figure 6.4: Confusion matrix showing correctness of guesses made by our CNN

Next we used a confusion matrix to show a summary of prediction results. We break down the results down into correct and incorrect predictions and show them with respect to each other. This creates a table, or matrix, of the number of classes, by number of classes (10x10 for us). It shows where the model is confused on predications.

The ideal form for the matrix is a straight diagonal line from top left to bottom right (meaning it has guessed correctly). As can be seen in figure 6.4, our matrix looks very well. There are some discrepancies, which match up with the classification report data, but we are overall very pleased with the results.

Manufacturing overfitting

Techniques to increase overfitting

Our model performed very well on both seen training data and unseen test data. However, this was due to multiple measures taken to reduce overfitting. This is when a model is over trained on the training dataset and becomes overly good at recognising the training data to the detriment of new data. Figure 7.1 communicates the idea well

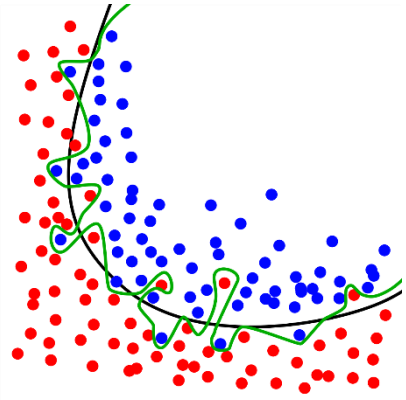


Figure 7.1: Graph showing good fit (black line) and overfitting (green line) [26]

As said, we took measures to prevent overfitting in our model. In an attempt to demonstrate an understanding of overfitting, I will discuss them and how I altered them, added new features and varied hyperparameters to encourage overfitting. [27]

Holdout data

This is was our test set; data you hold out until the end to show the model to get a true gauge of its ability. I did not remove this as I wanted a reference to see how well an overfitted model would work on it.

Cross-validation data

Our initial test results did not include cross validation (already discussed), so that was not an issue.

Data augmentation

In terms of this, we were doing much pre-processing to our data, so we disabled this to give the CNN the most straightforward data possible. It also meant less variation in possible training data, further limiting the results.

```
# Using ImageDataGenerator to preprocess our images
# All params are my selected preprocessing I want done to my images and will be randomly applied. Most are within ranges, so anything within that range
# Keeps our data unique everytime
# train_datagen = ImageDataGenerator(
#     rescale=1. / 255,          # The RGB vals in an image are [0, 255] by default. This scales down the vals to be [0, 1], normalising them
#     rotation_range=40,
#     width_shift_range=0.2,
#     height_shift_range=0.2,
#     shear_range=0.2,
#     zoom_range=0.2,
#     horizontal_flip=True)

train_datagen = ImageDataGenerator(rescale=1. / 255)
```

Figure 7.2: Showing disabled pre-processing (Only doing rescale as is necessary for ResNet50V2 to function)

Regularisation

This one is arguably the most interesting factors in terms of overfitting. It is the most direct way to stop (or induce) overfitting and yet we cannot do much here. Keras's ResNet50V2 has BN inherently built into it. It is a fundamental part of it, and as mentioned before it, it acts as its own regulariser through the model. This may prove to be a major issue. Implementing L2 is pointless as is rendered useless by BN. Dropout can negatively affect BN, but I am unsure if that will matter when it is only applied at the end of the model. I will put it to my overfitting model with a high rate to see the effect, if any.

Layer Complexity

Removing and simplifying layers can help to reduce needless complexity which can reduce overfitting. I will instead add a needless flatten layer and multiple dense layers to try and overtrain the model.

```
headModel = pretrained_model.output
headModel = tf.keras.layers.Flatten()(headModel)
headModel = tf.keras.layers.Dense(1024, activation='relu')(headModel)
headModel = tf.keras.layers.Dense(1024, activation='relu')(headModel)
headModel = tf.keras.layers.Dropout(0.5)(headModel)
headModel = tf.keras.layers.Dense(512, activation='relu')(headModel)
headModel = tf.keras.layers.Dense(512, activation='relu')(headModel)
headModel = tf.keras.layers.Dense(256, activation='relu')(headModel)
headModel = tf.keras.layers.Dropout(0.5)(headModel)
headModel = tf.keras.layers.Dense(128, activation='relu')(headModel)
headModel = tf.keras.layers.Dense(10, activation='softmax')(headModel)
```

Figure 7.3: Additional layers added

Early Stopping

This is something we had in our initial run, so I disabled it here to let the model run its full amount of epochs.

Varying Hyperparameters

We don't have too many hyperparameters to alter. I did change the number of epochs to 60, as that seemed like a decent amount to run within a given timeframe. I upped the batch size which I hoped would negatively affect the accuracy of unknown data. I kept image size the same, as ResNetV250 needs that and the patience was irrelevant with the early stopping gone.

Other

It should be noted that the pretrained ResNet50V2 is already extremely good at picking out monkeys due to ImageNet training, meaning overtraining may be an issue as it is already confident in certain general features.

Increasing dataset size can help a lot with overfitting, but due to the already relatively small scale of our dataset, size reduction was not an option.

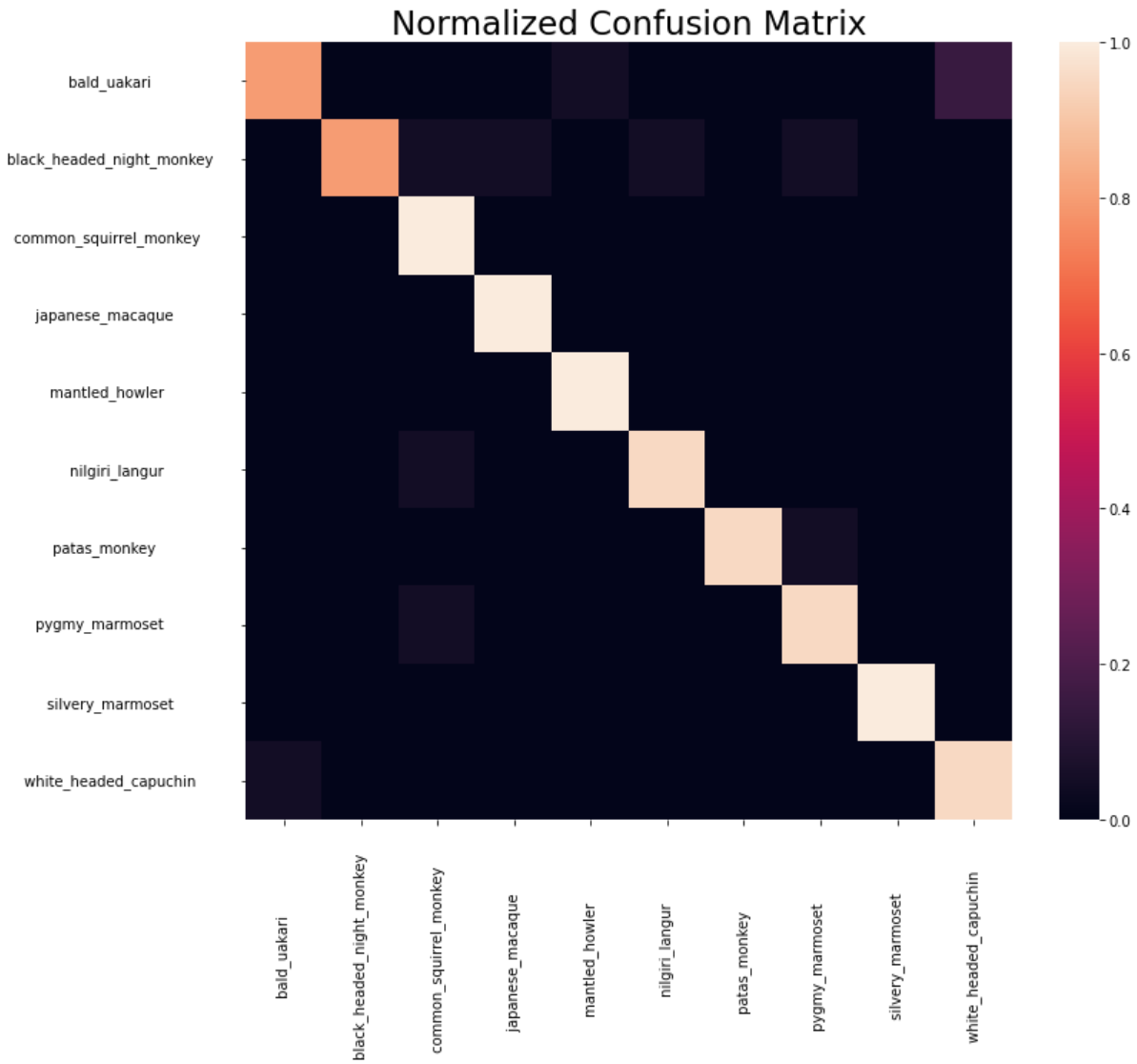
Results of attempted overfitting

```
Epoch 53/60
8/8 [==============================] - 29s 4s/step - loss: 0.0459 - acc: 0.9873 - val_loss: 0.1065 - val_acc: 0.9595
Epoch 54/60
8/8 [==============================] - 28s 4s/step - loss: 0.0152 - acc: 0.9958 - val_loss: 0.2277 - val_acc: 0.9685
Epoch 55/60
8/8 [==============================] - 29s 4s/step - loss: 0.0224 - acc: 0.9947 - val_loss: 0.1981 - val_acc: 0.9550
Epoch 56/60
8/8 [==============================] - 28s 4s/step - loss: 0.0309 - acc: 0.9916 - val_loss: 0.2866 - val_acc: 0.9595
Epoch 57/60
8/8 [==============================] - 28s 4s/step - loss: 0.0366 - acc: 0.9916 - val_loss: 0.2069 - val_acc: 0.9640
Epoch 58/60
8/8 [==============================] - 28s 4s/step - loss: 0.0548 - acc: 0.9916 - val_loss: 0.1805 - val_acc: 0.9595
Epoch 59/60
8/8 [==============================] - 28s 4s/step - loss: 0.0426 - acc: 0.9895 - val_loss: 0.2747 - val_acc: 0.9550
Epoch 60/60
8/8 [==============================] - 28s 3s/step - loss: 0.0233 - acc: 0.9926 - val_loss: 0.2362 - val_acc: 0.9640
```

Figure 7.4: Model fit results showing accuracy and loss when attempting to overfit

```
2/2 [==============================] - 30s 26s/step
                              precision    recall  f1-score   support

 mantled_howler                    0.94      0.80      0.86        20
 patas_monkey                      1.00      0.80      0.89        20
 bald_uakari                       0.87      1.00      0.93        20
 japanese_macaque                  0.95      1.00      0.98        20
 pygmy_marmoset                    0.95      1.00      0.98        20
 white_headed_capuchin             0.95      0.95      0.95        20
 silvery_marmoset                  1.00      0.95      0.97        20
 common_squirrel_monkey            0.90      0.95      0.93        20
 black_headed_night_monkey         1.00      1.00      1.00        20
 nilgiri_langur                    0.86      0.95      0.90        20

                     accuracy                           0.94       200
                    macro avg      0.94      0.94      0.94       200
                 weighted avg      0.94      0.94      0.94       200
```

Figure 7.5: Shows precision, recall, F1-score and number of images for each monkey species when attempting to overfit



As can be seen from the results on the unseen test data, the model still performs reasonably well. We not believe this to be a severe form of overfitting, if overfit at all. I believe this is down to a few doubts I had earlier. The biggest is the inherent reduction of overfitting due to ResNet having BN, a regularises, built into the structure. Since it was also trained on ImageNet with numerous common monkey species, this means it is already quite adept at recognising them, meaning we are trying to change something quite deeply engrained (potentially in the layers which we froze, meaning a guarantee in no change).

References

[1] Patil, H. (2021) What is feature engineering — importance, tools and techniques for Machine Learning, Towards Data Science. Available at: https://towardsdatascience.com/what-is-feature-engineering-importance-tools-and-techniques-for-machine-learning-2080b0269f10

[2] Dilmegani, C. (2022) What is data augmentation? techniques, examples & benefits, AIMultiple. Available at: https://research.aimultiple.com/data-augmentation/

[3] S. Mascarenhas and M. Agarwal, "A comparison between VGG16, VGG19 and ResNet50 architecture frameworks for Image Classification," 2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON), 2021, pp. 96-99, doi: 10.1109/CENTCON52345.2021.9687944.

[4] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[5] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In ICML, 2015.

[6] Van Laarhoven, T. L2 Regularization versus Batch and Weight Normalization ArXiv abs/1706.05350 (2017)

[7] Brownlee, J. (2020) A gentle introduction to the rectified linear unit (ReLU), Machine Learning Mastery. Available at: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

[8] Oommen, P. (2020) Resnets — residual blocks & deep residual learning, Towards Data Science. Available at: https://towardsdatascience.com/resnets-residual-blocks-deep-residual-learning-a231a0ee73d2

[9] Sachan, A. (2019) Detailed guide to understand and implement ResNets, CV-Tricks. Available at: https://cv-tricks.com/keras/understand-implement-resnets/

[10] He, K., Zhang, X., Ren, S., Sun, J. (2016). Identity Mappings in Deep Residual Networks. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds) Computer Vision – ECCV 2016. ECCV 2016. Lecture Notes in Computer Science(), vol 9908. Springer, Cham. https://doi.org/10.1007/978-3-319-46493-0_38

[11] IMAGENET 1000 class list (no date) IMAGENET 1000 Class List - WekaDeeplearning4j. WekaDeeplearning4j. Available at: https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/

[12] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016.

[13] Mohan, S. (2020) Keras implementation of resnet-50 (residual networks) architecture from scratch, MLK - Machine Learning Knowledge. Available at: https://machinelearningknowledge.ai/keras-implementation-of-resnet-50-architecture-from-scratch/#ResNet-50_Keras_Implementation

[14] Explain pooling layers: Max Pooling, average pooling, global average pooling, and Global Max Pooling. (2021) Knowledge Transfer. Available at: https://androidkt.com/explain-pooling-layers-max-pooling-average-pooling-global-average-pooling-and-global-max-pooling/

[15] Lin, M., Chen, Q. & Yan, S. (2013). Network In Network (cite arxiv:1312.4400Comment: 10 pages, 4 figures, for iclr2014)

[16] Song, Y. (2017) What is the different between MSE error and cross-entropy error in NN, Cooking&Coding Girl. Available at: https://susanqq.github.io/tmp_post/2017-09-05-crossentropyvsmes/

[17] Koech, K.E. (2020) Cross-entropy loss function - towardsdatascience.com, Towards Data Science. Available at: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

[18] Chauhan, N.S. (2020) Optimization algorithms in Neural Networks, KDnuggets. Available at: https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html

[19] Keras Documentation: Adam (no date) Keras. Available at: https://keras.io/api/optimizers/adam/

[20] Intuition of adam optimizer (2020) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/intuition-of-adam-optimizer/

[21] Allibhai, E. (2018) Holdout vs. cross-validation in machine learning, Medium. Medium. Available at: https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f

[22] Kumar, A. (2022) K-fold cross validation - python example, Data Analytics. Available at: https://vitalflux.com/k-fold-cross-validation-python-example/

[23] 3.1. cross-validation: Evaluating estimator performance (no date) scikit. Available at: https://scikit-learn.org/stable/modules/cross_validation.html

[24] Classification: Precision and recall |machine learning | google developers (no date) Google. Google. Available at: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.

[25] LT, Z. (2021) Essential things you need to know about F1-score, Towards Data Science. Available at: https://towardsdatascience.com/essential-things-you-need-to-know-about-f1-score-dbd973bf1a3

[26] Overfitting (2022) Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Overfitting

[27] Lin, D.C.-E. (2020) 8 simple techniques to prevent overfitting | by David Chuan-en Lin ..., Towards Data Science. Available at: https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d