[**1 Binary Classification**](#_gct0jdy101a1)3

[1.1 Data train/validation/test split (1a)](#_186iv5xwupva) 3

[1.1.1 Introduction](#_2i10dti86m3z) 3

[1.1.2 Rational](#_6p7mzz5akc9z) 3

[1.1.3 Design](#_1iqj8yqctxvv) 4

[1.1.4 Implementation](#_re0u2kz6bxqq) 4

[1.1.5 Testing & Results](#_ycty69ktd8a7) 5

[1.1.6 Discussion & Conclusion](#_lqn2wj5psnf4) 5

[1.1.7 References](#_ay2o39i8paaf) 6

[1.1.8 Appendix](#_pvkhyxrwjzus) 6

[1.2 Design and develop a baseline CNN (1b)](#_jfyvbk55fvrl) 10

[1.2.1 Introduction](#_om7yx8g5gvgg) 10

[1.2.2 Rational](#_6tw5kkyjyg9u) 10

[1.2.3 Design](#_lagni5v9egsu) 10

[1.2.4 Implementation](#_q9ra09p0rkkn) 11

[1.2.5 Testing & Results](#_szlj79xz3lpb) 13

[1.2.6 Discussion & Conclusion](#_vsc9789p3j9u) 16

[1.2.7 References](#_fwlk8yubcc7r) 16

[1.2.8 Appendix](#_f10l0oenzlqi) 17

[1.3 Improve the Baseline Model’s Performance (1c)](#_qdqeeixqj9jn) 21

[1.3.1 Introduction](#_bx2btvlyyysz) 21

[1.3.2 Rational](#_2xq1ay862b0w) 21

[1.3.3 Design](#_5is293kt1s8g) 21

[1.3.4 Implementation](#_8qup3kmbqyjf) 22

[1.3.5 Testing & Results](#_v6df76k838v2) 25

[1.3.6 Discussion & Conclusion](#_cpg0w0b4vyxz) 27

[1.3.7 References](#_8y14q4f0w74q) 27

[1.3.8 Appendix](#_zfx1r58askbu) 27

[1.3 Visualise Intermediate CNN Outputs (1d)](#_x5o4mn7sexve) 31

[1.4.1 Introduction](#_4z8fe3h9vupw) 31

[1.4.2 Implementation](#_dziqb4oelleg) 31

[1.4.3 Testing & Results](#_9mett1nh7cdj) 31

[1.4.4 Discussion & Conclusion](#_79yrznoepwd2) 31

[1.4.5 References](#_jec3crj5185l) 31

[1.4.6 Appendix](#_fvduyg5l97f) 31

[**2 Multi Classification - Transfer Learning**](#_qbi2exrouexv)32

[2.1 Data train/validation/test split (2a)](#_6iagm46yal33) 32

[2.1.1 Introduction](#_jzt926729vwm) 32

[2.1.2 Rational](#_n94oajcgcbg3) 32

[2.1.3 Design](#_wgjjs9ducpny) 32

[2.1.4 Implementation](#_ghim8264rw94) 32

[2.1.5 Testing & Results](#_sq206jc16ido) 33

[2.1.6 Discussion & Conclusion](#_30dkkyhctxj5) 33

[2.1.7 References](#_r26b116l26y8) 33

[2.1.8 Appendix](#_laeqobbgjmxl) 34

[2.2 InceptionV3 Fine-Tuning Based Transfer Learning (2b)](#_m5o6m54qk8zy) 37

[2.2.1 Introduction](#_eaajyytqoa78) 37

[2.2.2 Rational](#_yup2npf03plq) 37

[2.2.3 Design](#_mishxnffake6) 38

[2.2.4 Implementation](#_xer9ab2gsxhv) 39

[2.2.5 Testing & Results](#_joamxeehzhm1) 40

[2.2.6 Discussion & Conclusion](#_w95vjnm57tac) 42

[2.2.7 References](#_vdwvorio1xwo) 42

[2.2.8 Appendix](#_lvxeknbmmfdf) 42

[2.3 Test Network Performance with Unseen Data (2c)](#_pu8t89efn2sy) 47

[2.3.1 Introduction](#_l6op6jcuaxac) 47

[2.3.2 Implementation](#_banevdvjufje) 47

[2.3.3 Testing & Results](#_t6jlpajxbq1p) 47

[2.3.4 Discussion & Conclusion](#_suh758y9ylqu) 49

[2.3.5 References](#_jejhalrpzbfe) 49

[2.3.6 Appendix](#_tfwbxiwy79pc) 49

## 

# 1 Binary Classification

## 1.1 Data train/validation/test split (1a)

### 1.1.1 Introduction

The first step of training a convolutional neural network (CNN) is the gathering and preparation of the data. An evaluation protocol must be decided upon and the data divided into an appropriate train/validation/test split.

The objective of this exercise is to:

* Examine the data.
* Decide on an evaluation protocol.
* Determine the train/validation/test split.

The two classes selected out of the Imagnette dataset were n03028079 (cathedral) and n03445777 (golf ball).

### 1.1.2 Rational

There are a number of factors to be considered when deciding how to evaluate the model and split the training data.

In this case the major factors were:

1. The size of the dataset.
2. The computing resources available.

The dataset made up from the n03028079 and n03445777 classes is relatively small, with exactly 1350 samples per class.

The computing resources available were limited to free cloud platforms or trial credits for professional cloud platforms. Hence efficient use of computing resources was an important aspect of implementation.

Both the holdout method and k-fold cross validation were considered. The k-fold method was a good candidate for the dataset because with small datasets, k-fold reduces the variance in model performance estimates [1]. However this comes with a cost. It is expensive on computing resources. The value of k (number of folds) would likely be between 5 and 10, and since the model would have to be trained k times, it would vastly increase the resources used.

Additionally it adds complexity to the solution and more complexity gives greater opportunities for defects to appear in the model.

Ultimately the holdout method was selected because of its simplicity and low computing cost.

Defining the train/validate/test split is a balance. The training data has to have enough samples to train a model with sufficient complexity. While the validation data needs to have enough samples to accurately estimate the model performance for the purpose of tuning the hyperparameters. Finally the test set also requires an adequate number of samples to make an accurate final performance estimation.

A split of 70% training, 15% validation and 15% test seemed reasonable as it would allocate approximately 1900 samples to the train set and 400 to both the validation and test sets.

### 1.1.3 Design

The most effective way to split the data was to use a Python script. It was a much more time efficient and less error prone method than doing the split manually.

A high level algorithm for splitting the data was defined:

1. Create new directories with the class names and train, validate, test.
2. Copy all instances of the two chosen classes from the original Imagenette directory to the new folders. The images in the train and val should be combined into one directory that holds all instances of that class.
3. Split the images into the newly created train, validate, and test directories in accordance with the split stated in section 1.1.2

The details of implementation will be described in the following section.

### 1.1.4 Implementation

As shown in figure 1.1.8a in the appendix, it was first necessary to unzip the TAR file and copy the contents to a new ‘data’ directory in Google Drive.

Next the drive was mounted to allow the Python script to access the file system (see figure 1.1.8b). Additionally, the working directory and directories with original Imagenette data were added to the OS path.

Finally, the data splitting algorithm was implemented by writing three functions; init\_run, drop\_classes, split\_data.

A ‘run’ directory was created inside the Imagenette directory for separating the data to be used for the binary classification task.

The first function **init\_run** simply initialises the run directory, making sure all the required subdirectories exist and if not it creates them. The directory structure required for the Keras flow\_from\_directory method was used.

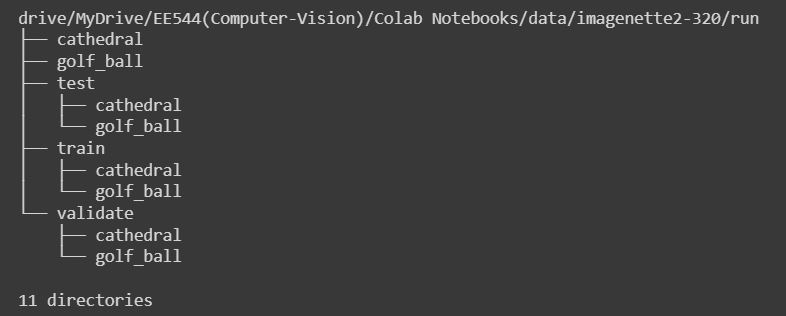


Figure 1.1.4a - Directory tree

**drop\_classes** drops the unused classes, only copying the used images into the run directory.

**split\_data** uses three for loops to copy the pre-defined portion of images of each class into the train, validate and test directories. For instance, the 70% training allocation translates to 945 samples of each class being copied to the train directory.

### 1.1.5 Testing & Results

A quick test was done to ensure that the correct number of samples was in each directory. The test code which can be seen in figure 1.1.8d produced the following output:

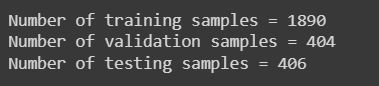


Figure 1.1.5a - Code cell with drive mounting operation

This shows that the data was successfully split as stated in section 1.1.2.

Note: the test directory has two more samples than validation. This is because 15% of 1350 amounted to 202.5. Hence the remainder was added to the test set.

### 1.1.6 Discussion & Conclusion

In a real life scenario, especially if dealing with a critical system (e.g. medical imaging or autonomous vehicle control), the use of k-fold cross evaluation would have been preferable. This is because every subsample is used exactly once at some point for validating the model [2]. K-fold cross evaluation would produce less variance in validation accuracy as rather than having a single validation set, there would be k sets, allowing a mean accuracy to be gathered during evaluation.

A preliminary test was done on the data to see how long it would take to train a basic model with the Imagenette data on Colab using the GPU accelerator. It took 547 seconds. This heavily influenced the decision to use the improved holdout method.

The split of 70% 15% 15% was a data oriented decision. If there were more samples available, a higher percentage could be allocated to the training set. The rationale behind the split was to give the minimum adequate number of samples to the test and validation sets while giving the rest to training.

### 1.1.7 References

[1]

F. Chollet, “4.2 Evaluating Machine Learning Models,” in *Deep Learning with Python*, 1st edition., Shelter Island, New York: Manning Publications Co., 2018.

[2]

P. F. Whelan, “Machine Learning for Computer Vision (1),” in *Computer Vision Course Notes – EE544*, Dublin City University, 2022.

### 1.1.8 Appendix

tar -xvf '/content/gdrive/MyDrive/Computer-Vision/Colab Notebooks/data/imagenette2-320.tgz' -C '/content/gdrive/MyDrive/Computer-Vision/Colab Notebooks/data/'

Figure 1.1.8a - Code cell with unzip command

# Mount google drive and define locations of original, unmodified imagenette data

from google.colab import drive, files

import os

drive.mount("/content/gdrive", force\_remount=True)

working\_dir = "/content/gdrive/MyDrive/Computer-Vision/Colab Notebooks/"

imagenette\_dir = os.path.join(working\_dir, "data/imagenette2-320")

original\_train\_dir = os.path.join(imagenette\_dir, "train")

original\_test\_dir = os.path.join(imagenette\_dir, "val")

Figure 1.1.8b - Code cell with drive mounting operation

from pathlib import Path

from shutil import copyfile

from distutils.dir\_util import copy\_tree # import necessary to circumvent issues with using shutils.copytree in Python v3.6

# run\_dir is where data the may be manipulated during runtime is put

run\_dir = os.path.join(imagenette\_dir, "run")

train\_dir = os.path.join(run\_dir, "train")

validate\_dir = os.path.join(run\_dir, "validate")

test\_dir = os.path.join(run\_dir, "test")

# Create 2 class folders that will contain all instances of their respective class

# Create 3 directories for train validate and test data

def init\_run():

Path(run\_dir+"/cathedral").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/golf\_ball").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/train").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/train/cathedral").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/train/golf\_ball").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/validate").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/validate/cathedral").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/validate/golf\_ball").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/test").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/test/cathedral").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/test/golf\_ball").mkdir(parents=True, exist\_ok=True)

# Only copy the classes to be used in the model to the run directory

def drop\_classes():

try:

# Copy all images in train/n03028079 and val/n03028079 to cathedral dir

copy\_tree(original\_train\_dir+"/n03028079", run\_dir+"/cathedral")

copy\_tree(original\_test\_dir+"/n03028079", run\_dir+"/cathedral")

# Copy all images in train/n03445777 and val/n03445777 to golf\_ball dir

copy\_tree(original\_train\_dir+"/n03445777", run\_dir+"/golf\_ball")

copy\_tree(original\_test\_dir+"/n03445777", run\_dir+"/golf\_ball")

except Exception as e:

print(e)

# Get a portion of data from each of the classes and put it the the appropriate dir

# Prior testing shows that the classes are perfectly balanced meaning that we can allocate the same number of each class to train validate and test dirs

def split\_data():

train\_split = 945

validate\_split = train\_split + 202

test\_split = validate\_split + 203

c1\_samples = os.listdir(run\_dir+"/cathedral")

c2\_samples = os.listdir(run\_dir+"/golf\_ball")

c1\_n\_samples = len(c1\_samples)

c2\_n\_samples = len(c2\_samples)

print("{} class 1 samples".format(c1\_samples))

print("{} class 2 samples".format(c2\_samples))

for i in range(0, train\_split):

copyfile(run\_dir+"/cathedral/"+c1\_samples[i], run\_dir+"/train/cathedral/"+c1\_samples[i])

copyfile(run\_dir+"/golf\_ball/"+c2\_samples[i], run\_dir+"/train/golf\_ball/"+c2\_samples[i])

for i in range(train\_split, validate\_split):

copyfile(run\_dir+"/cathedral/"+c1\_samples[i], run\_dir+"/validate/cathedral/"+c1\_samples[i])

copyfile(run\_dir+"/golf\_ball/"+c2\_samples[i], run\_dir+"/validate/golf\_ball/"+c2\_samples[i])

for i in range(validate\_split, test\_split):

copyfile(run\_dir+"/cathedral/"+c1\_samples[i], run\_dir+"/test/cathedral/"+c1\_samples[i])

copyfile(run\_dir+"/golf\_ball/"+c2\_samples[i], run\_dir+"/test/golf\_ball/"+c2\_samples[i])

init\_run()

drop\_classes()

split\_data()

Figure 1.1.8c - Code cell with 3 functions defined for splitting the data

train\_samples = len(os.listdir(run\_dir+"/train/cathedral")) + len(os.listdir(run\_dir+"/train/golf\_ball"))

validate\_samples = len(os.listdir(run\_dir+"/validate/cathedral")) + len(os.listdir(run\_dir+"/validate/golf\_ball"))

test\_samples = len(os.listdir(run\_dir+"/test/cathedral")) + len(os.listdir(run\_dir+"/test/golf\_ball"))

print("Number of training samples = {}\nNumber of validation samples = {}\nNumber of testing samples = {}\n".format(train\_samples, validate\_samples, test\_samples))

Figure 1.1.8d - Code cell with samples split test

## 1.2 Design and develop a baseline CNN (1b)

### 1.2.1 Introduction

The objective of this activity was to develop a simple baseline convolution neural network for the binary classification task. This means creating a ‘simplest thing that works’ cnn with no significant tuning of hyperparameters.

### 1.2.2 Rational

Colab was used for the model design and training due to its free access to GPUs and intuitive development environment.

The approach taken to develop the baseline model was to follow the requirements in the project spec, and where possible, choose the simplest component that works for a particular aspect of the model.

### 1.2.3 Design

Before writing code some design decisions were made.

The architecture was defined in the specification as follows:

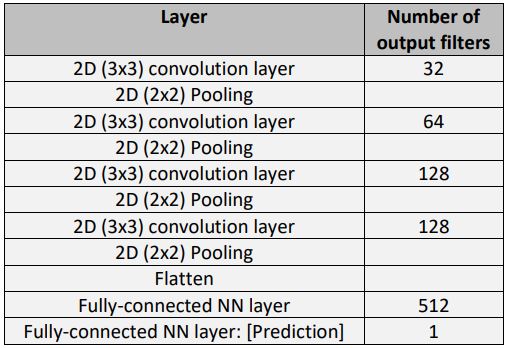


Figure 1.2.3a - binary classification CNN architecture table

Within this CNN the feature extraction is done at the convolution and pooling layers. The features extracted get progressively more advanced while moving through the layers [2]. The pooling layers bring two main advantages Firstly, they decrease the number of parameters by downsampling the images. That is, they reduce the information down by dividing the input into segments (e.g. 2x2 blocks) and combining the segments in the output by either picking the max value or the average of all values. Secondly the pooling layer makes the model better able to handle minor variance in the images, such as small translations or distortions. However the downside is that some information will be lost, depending on whether max or average pooling is used. [2]

The flatten layer is a reshaping layer that turns the 2D filters from the convolution layers into a 1D vector which can be fed into a fully connected layer.

Another requirement was that the convolution layers use the ReLu activation function. This adds non-linearity to the convolution stages which would otherwise be entirely linear. ReLu is a good choice over the hyperbolic tangent (tanh) activation function because it is fast to compute. A downside to this is that ReLu can cause dying neurons (neurons that only output 0) when the neuron weights are tweaked in such a way that the sum of its inputs are negative for all samples in the training set [1].

For the final layer, the Sigmoid activation function was chosen. This is because in a binary classification model, the output layer should output a value between 0 and 1.

The next consideration was the metric to be used for evaluating the performance of the model. Since this is a classification problem and the classes are balanced, accuracy - the percentage of correctly identified images, is a suitable metric.

The loss function determines how well the network is performing. Because this is a binary classification task, binary cross-entropy was selected as the loss function.

The optomiser is responsible for updating the network parameters depending on the output of the loss function. For the purpose of a baseline model, it made sense to select an optomiser that works well with minimal configuration and has an adaptive learning rate to help accelerate the training time. For this reason the ‘Adam’ optimisation algorithm was selected.

The general outline for building the baseline CNN model is as follows:

1. Prepare the data
2. Define the CNN architecture
3. Define the hyperparameters
4. Build and compile the model
5. Train the model

### 1.2.4 Implementation

Development of the baseline model was split up into components.

The first component was the preparation of the training and validation data for inputting to the CNN. The **stage\_train\_data** (figure 1.2.8a) function encapsulates this task. It uses the Keras ImageDataGenerator class and flow\_from\_directory method to extract the images from the directories created in section 1.1.4 and do some preprocessing. Images are rescaled to the [0, 1] range and resized to 150 x 150 pixels.

The function takes in the target size, batch size and class mode. It returns the train and validation data generators which are used later in the model fitting.

Next is the function that defines the architecture of the CNN; **baseline\_cnn\_model**. It takes in a dictionary of hyperparameters and returns a Keras sequential model object. As shown in the figure 1.2.8b, the layers are added in accordance with the specification. At this point a sanity test was performed to verify that the model laid out in baseline\_cnn\_model matches the requirements in the specification.

A test function **show\_model** (figure 1.2.8c) was defined which clears the Keras session (to reset figure labelling), uses the model summary method to get a text representation of the architecture, and finally uses the plot\_model method to export a PNG of the model structure. As shown in figure 1.2.4a, the model is structured accordingly.

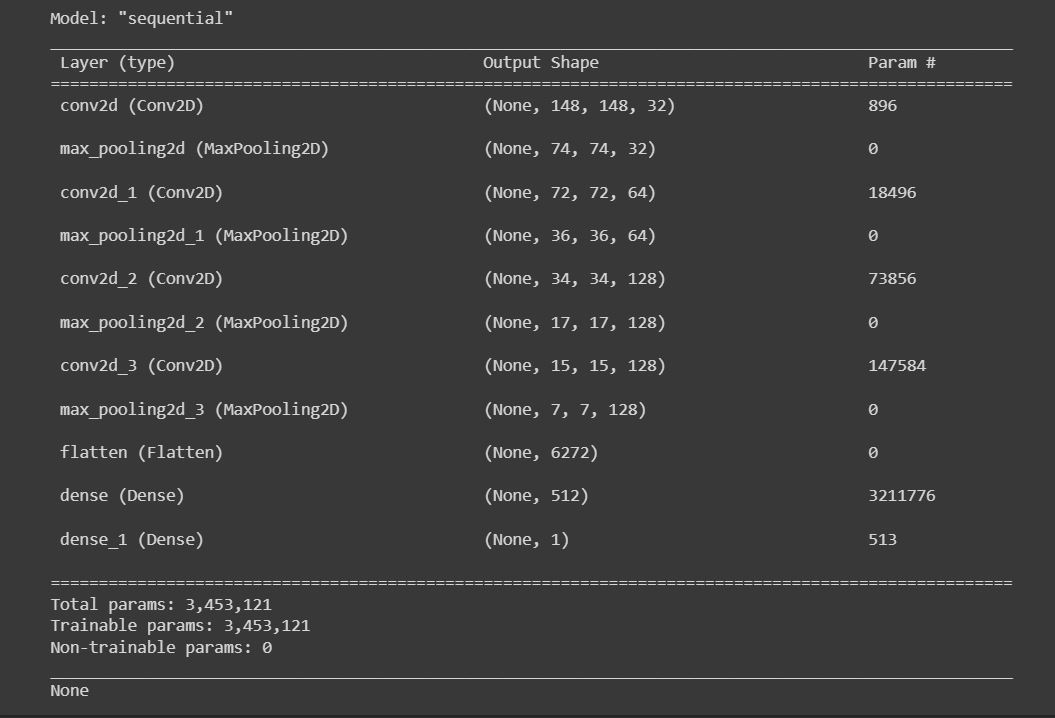


Figure 1.2.4a - output of model.summary() from show\_model function

Building and compiling the model is achieved by using the **build\_model** (figure 1.2.8d) function. It was designed to be reusable and will be extended later in the project. It takes in a model name and hyperparameters dictionary, and returns a compiled CNN model. If the name matches “baseline\_bin\_class” it will compile the model with the loss function set to ‘binary\_cross\_entropy’ and the metric set to ‘accuracy’. The optimizer is passed in through the hyperparameters dictionary.

The model training is done through the function; **train\_model**. As shown in figure 1.2.8e, It takes in a compiled model, train data generator, validate data generator, and hyperparameters dictionary. It uses the Keras fit method to initiate training of the model. The training history is saved and returned from the function. The computational cost of the training is measured in seconds. This is achieved by using the Python native time module. The system time is recorded the instant before training commences and then subtracted from the system time at completion of training.

The final component of the model training code is the **run\_baseline** (figure 1.2.8f) function. It sequentially calls all the functions previously discussed and returns the model and model history. Before running it, the hyperparameters must be stated in a dictionary. For the baseline model, the network structure hyperparameters were the dilation rate and stride. The training hyperparameters were the number of epochs and the optimizer. The number of epochs was set to 20 for this experiment. This should give the model adequate training while giving the opportunity to see if and at what point overfitting occurs. The amount of hyperparameters is likely to be increased later in the project for fine tuning the network. See output below:

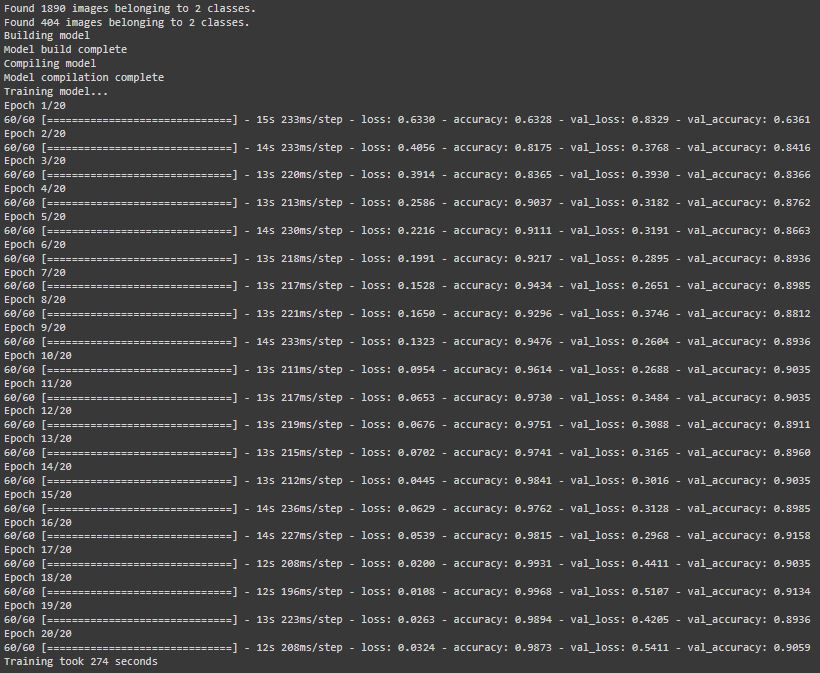


Figure 1.2.4b - output for run\_baseline

### 1.2.5 Testing & Results

The experiment yielded a baseline model with surprisingly high accuracy.

Final validation accuracy was 90.59% while the final validation loss was 0.5411.

The model performance was displayed by creating a **plot\_metrics** (figure 1.2.8g) function which outputs the final validation accuracy and loss as text and plots an accuracy graph and loss graph.



Figure 1.2.5a - validation accuracy and loss output

Another sanity test was performed to verify that there were no data leaks in the training, validation, or test data. This was done by creating a function; **check\_data\_leaks** (figure 1.2.8h) that uses Python sets. If an intersection were to be found, it would mean that there was a defect in the data splitting process. After running the test, no intersections were detected meaning that the training process had not been compromised.

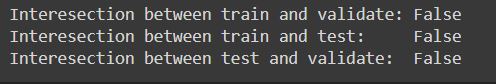


Figure 1.2.5b - output from data leak sanity test

As shown in figure 1.2.5c the train accuracy increases constantly throughout training and approaches 100%. The validation accuracy increases to 90% and then gets stuck at approximately 10 epochs. This shows that something stalled the improvement of the network performance. One of the most common causes of this deficiency is overfitting - when the network starts to remember patterns in the training set that are not necessarily reflective of the real life data. Overfitting is said to occur when there are too many parameters for the number of training samples [3]. This is likely to be the case considering there are 3,453,121 parameters in this model and only 1890 training samples.

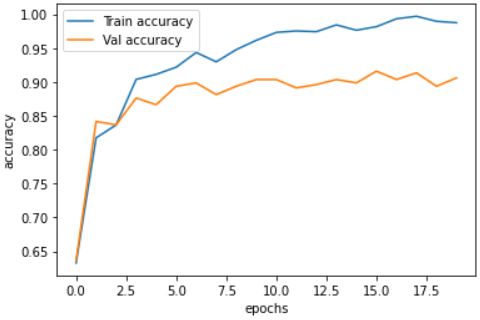


Figure 1.2.5c - train and validation accuracy graph

In the loss graph displayed in figure 1.2.5d, the training loss converges towards 0 while the validation loss decreases until around 8 epochs and then begins to increase. This is a strong indication of overfitting.

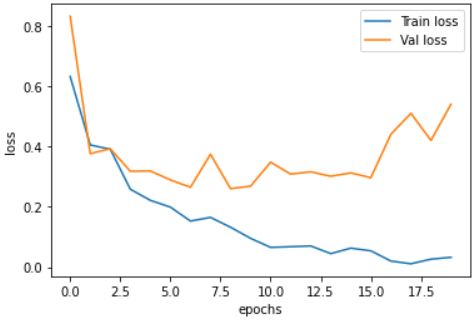


Figure 1.2.5d - train and validation loss graph

The final test accuracy was 91.87% while the test loss at 4.0, was relatively high compared to the training loss. Again, This high loss supports the theory that overfitting is the primary cause for the plateau in validation accuracy. The code for this in figure 1.2.8i.

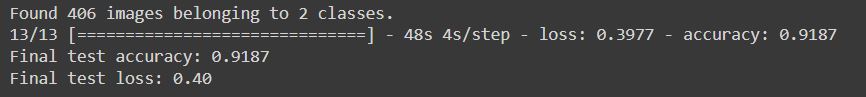


Figure 1.2.5e - final test evaluation metrics

The cost of training the network was measured in seconds and amounted to 274 seconds

### 1.2.6 Discussion & Conclusion

Initially after training the model and measuring the accuracy at 90%+ for both validation and test sets, it was presumed that there was an error in the data or evaluation to cause it to be so high. After some testing and further thought, the conclusion was made that making classification predictions on whether an image is either a cathedral or golf ball isn’t a very complicated problem. If there were a human performing the binary classification, a prediction accuracy of almost 100% would be expected.

There are a number of regularisation methods that can be used to improve the model and reduce overfitting.

Such methods include altering the structure of the CNN, implementing early stopping in the training, and artificially increasing and diversifying the training data. These will be investigated in the next section of this report.

### 1.2.7 References

[1]

A. Géron, “Chapter 11: Training Deep Neural Networks,” in *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*, Second edition., Sebastopol, California: O’Reilly Media, Inc., 2017.

[2]

P. F. Whelan, “Convolutional Neural Networks (1),” in *Computer Vision Course Notes – EE544*, Dublin City University, 2022.

[3]

P. F. Whelan, “Deep Learning: Training,” in *Computer Vision Course Notes – EE544*, Dublin City University, 2022.

### 1.2.8 Appendix

# Preprocess images to use in the CNN

def stage\_train\_data(target\_size, batch\_size, class\_mode):

train\_datagen = ImageDataGenerator(rescale=1./255)

validate\_datagen = ImageDataGenerator(rescale=1./255)

train\_generator = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=target\_size,

batch\_size=batch\_size,

class\_mode=class\_mode

)

validate\_generator = validate\_datagen.flow\_from\_directory(

validate\_dir,

target\_size=target\_size,

batch\_size=batch\_size,

class\_mode=class\_mode

)

return train\_generator, validate\_generator

Figure 1.2.8a - stage\_train\_data function

def baseline\_cnn\_model(hyperparameters):

model = Sequential()

# Input Layer

model.add(Conv2D(

32,

(3, 3),

activation="relu",

strides=hyperparameters["strides"],

dilation\_rate=hyperparameters["dialation\_rate"],

input\_shape=(150, 150, 3)

))

# Hidden Layers

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation="relu", strides=(1, 1), dilation\_rate=1,))

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(128, (3, 3), activation="relu", strides=(1, 1), dilation\_rate=1,))

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(128, (3, 3), activation="relu", strides=(1, 1), dilation\_rate=1,))

model.add(MaxPooling2D((2, 2)))

model.add(Flatten())

model.add(Dense(512, activation="relu"))

# Output Layer

model.add(Dense(1, activation="sigmoid"))

return model

Figure 1.2.8b - baseline\_cnn\_model function

# Show summary of model and plot model structure

def show\_model(model):

K.clear\_session()

print(model.summary(line\_length=100))

plot\_model(model, "baseline-model.png", show\_shapes=True, show\_layer\_names=True)

test\_hparams = {

"dialation\_rate": 1,

"strides": (1, 1)

}

test\_model = baseline\_cnn\_model(test\_hparams)

show\_model(test\_model)

Figure 1.2.8c - cnn architecture sanity test

# Build and compile the model

def build\_model(model\_name, hyperparameters):

print("Building model")

# Build baseline binary classification model

if model\_name == "baseline\_bin\_class":

model = baseline\_cnn\_model(hyperparameters)

print("Model build complete")

print("Compiling model")

model.compile(loss="binary\_crossentropy", optimizer=hyperparameters["optimizer"], metrics=["accuracy"])

print("Model compilation complete")

return model

Figure 1.2.8d - build\_model function

# Train the model

def train\_model(compiled\_model, train\_gen, validate\_gen, hyperparameters):

print("Training model...")

t0 = time()

history = compiled\_model.fit(

train\_gen,

validation\_data=validate\_gen,

epochs=hyperparameters["epochs"],

)

print("Training took {} seconds".format(int(time() - t0)))

return history

Figure 1.2.8e - train\_model function

# Define hyperparameters and build baseline CNN

hyperparameters = {

"epochs": 20,

"optimizer": "adam",

"dialation\_rate": 1,

"strides": (1, 1)

}

def run\_baseline(hyperparameters):

train\_gen, val\_gen = stage\_train\_data((150, 150), 64, "binary")

model = build\_model("baseline\_bin\_class", hyperparameters)

history = train\_model(model, train\_gen, val\_gen, hyperparameters)

return model, history

model, history = run\_baseline(hyperparameters)

Figure 1.2.8f - run\_baseline function

def plot\_metrics(history):

validation\_metrics = pd.DataFrame(history.history)

print(validation\_metrics)

print()

validation\_metrics[["accuracy", "val\_accuracy"]].plot(xlabel="epochs", ylabel="accuracy").legend(["Train accuracy", "Val accuracy"])

validation\_metrics[["loss", "val\_loss"]].plot(xlabel="epochs", ylabel="loss").legend(["Train loss", "Val loss"])

final\_val\_acc = validation\_metrics.at[19, "val\_accuracy"]

final\_val\_loss = validation\_metrics.at[19, "val\_loss"]

print("Final validation accuracy: {:.4f}".format(final\_val\_acc))

print("Final validation loss: {:.4f}".format(final\_val\_loss))

print()

plot\_metrics(history)

Figure 1.2.8g - plot\_metrics function

# This function ensures that there were no errors in the data splitting such that an image is more than 1 set

def build\_sample\_set(dir, s):

for sample in os.listdir(dir):

s.add(sample)

# Use set operations to make sure there are no intersections between datasets

def check\_data\_leaks():

train\_samples = set()

validate\_samples = set()

test\_samples = set()

build\_sample\_set(run\_dir+"/train/cathedral", train\_samples)

build\_sample\_set(run\_dir+"/train/golf\_ball", train\_samples)

build\_sample\_set(run\_dir+"/validate/cathedral", validate\_samples)

build\_sample\_set(run\_dir+"/validate/golf\_ball", validate\_samples)

build\_sample\_set(run\_dir+"/test/cathedral", test\_samples)

build\_sample\_set(run\_dir+"/test/golf\_ball", test\_samples)

print("Intersection between train and validate: {}".format(not train\_samples.isdisjoint(validate\_samples)))

print("Intersection between train and test: {}".format(not train\_samples.isdisjoint(test\_samples)))

print("Intersection between test and validate: {}".format(not test\_samples.isdisjoint(validate\_samples)))

# If all are false then there are no intersections/data leaks

check\_data\_leaks()

Figure 1.2.8h - check\_data\_leaks function

# Do final evaluation of the model using holdout test data

test\_datagen = ImageDataGenerator(rescale=1./255)

test\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(150, 150),

batch\_size=32,

class\_mode="binary"

)

test\_metrics = model.evaluate(test\_generator)

print("Final test accuracy: {:.4f}".format(test\_metrics[1]))

print("Final test loss: {:.2f}".format(test\_metrics[0]))

Figure 1.2.8i - final test evaluation

## 1.3 Improve the Baseline Model’s Performance (1c)

### 1.3.1 Introduction

According to the Chollet’s summary on machine learning fundamentals, the next stage after building a model that overfits is to regularise the model and tune its hyperparameters [1].

The objective of this exercise is to propose improvements to the baseline model, experiment by applying said improvements, and record the results.

A validation accuracy of 90.1% was achieved in the baseline. Reducing the validation and test loss and increasing the validation accuracy to 93%+ could be seen as a reasonable goal for this activity.

### 1.3.2 Rational

Adding convolutional or dense layers to the network is not permitted. However, that does not rule out adding other layers to the CNN such as a dropout layer.

Adding new data is another common way to solve overfitting but this is not allowed in this instance. Instead the training data can be artificially expanded by using data augmentation.

### 1.3.3 Design

After a regularisation technique is applied or a hyperparameter is tuned, the result will be recorded. This will allow an accurate assessment of how each change affects the network.

Data augmentation will enrich the training data by introducing new variants to the network. It does this by randomly transforming the original sample in some way within the bounds of user set parameters. It is important to choose transformations that are representative of real life data, otherwise it may make convergence more difficult [2]. For this task, the majority of real life data would likely come from people capturing images with their personal electronic devices (mobile phones or cameras). With this knowledge it is reasonable to assume that a variety of transformations might be present such as:

* Rotations
* Shifts
* Shears
* Zooms
* Brightness adjusts
* Horizontal flips

It is difficult to define suitable bounds for each transformation because with human error, a real life image will not necessarily conform to any standard. For example, an image could be zoomed to the point where it is unrecognisable, or it could be rotated 180 degrees. But for this experiment the assumption will be made that only images that are easily recognisable by humans will be presented to the network.

Adding dropout layers to the CNN structure is another way to reduce overfitting. Dropout layers randomly switch off neurons during the training process, making the network work harder to converge. One of the researchers who first proposed dropout; Geoffrey Hinton, likened it to how banks rotate tellers to prevent them from conspiring to defraud the bank [3]. In deep learning terms, this means that randomly switching off neurons will prevent them from making connections that predict ‘superstitions’ about the training data.

For instance implementing the following techniques is likely a good starting point:

1. Apply data augmentation
2. Add dropout layers to the network structure
3. Implement early stopping

### 1.3.4 Implementation

Data augmentation was implemented on the training data by using the Keras ImageDataGenerator class. This allows the transformations to be applied during runtime rather than permanently adding extra samples and using up more storage. The function **augment\_train\_data** was added which is a modification of the stage\_train\_data function used in 1.2.4. As well as rescaling the training images now random transformations are applied - rotation, translation, shear, zoom, horizontal flip & brightness adjust. To compensate for the now much larger training set, the number of epochs was increased to 40.

Some experimentation was done on the values for the ranges of the translations. Initially higher values of .33 for shifting, scaling & zoom and 45 degrees for rotation were used. It was found that while the training and validation accuracy and loss were more highly correlated, the model performed slightly worse and the validation loss was very unstable.

It should be noted that running the neural network for this task was done through the **run\_improved\_bin\_class** function (figure 1.3.8c).

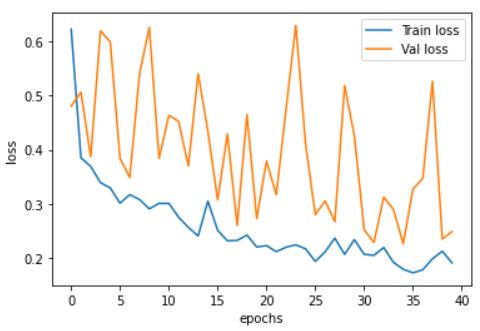
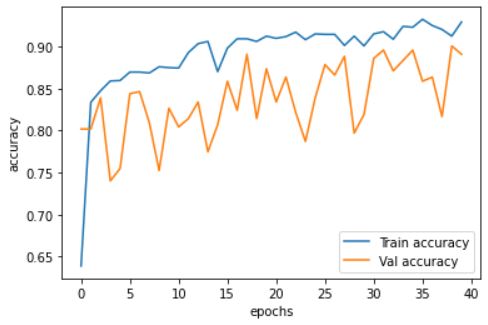


Figure 1.3.4a - results from data augmentation experiment 1

On further thought, the compounding effect of all of these translations with a higher range probably generated some images that pushed the boundaries of what an actual image would look like.

A summary on the experiments performed in order to find a more optimal set of data augmentation parameters:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exp No. | Rotation | Y-Shift | X-shift | X-flip | Shear | Zoom | Bright | Result |
| 1 | 45 | 0.33 | 0.33 | True | 0.33 | 0.33 | 0.6-1.4 | Significantly less overfitting than baseline. Reduction in both training and validation performance indicating now underfitting. |
| 2 | Off | Off | Off | True | 0.15 | 0.15 | Off | Performance roughly on par with baseline but with slightly less overfitting. |
| 3 | 25 | Off | Off | True | 0.2 | 0.2 | Off | Performance worse than baseline, no improvement in overfitting. |
| 4 | 25 | 0.15 | 0.15 | True | 0.2 | 0.2 | Off | Performance slightly better than baseline, overfitting reduced. |

Experiment 4 yielded the best results. To demonstrate the improvement relative to the baseline, the baseline model was trained again with the same number of epochs as the data augmentation experiment models.

The baseline model accuracy and loss plots:

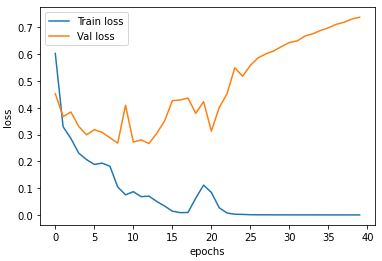
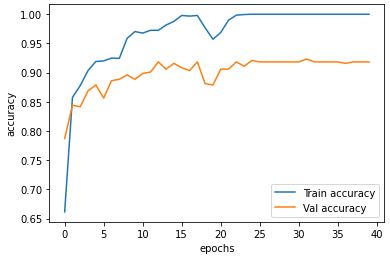


Figure 1.3.4b - results from baseline with 40 epochs

The data augmented model accuracy and loss plots:

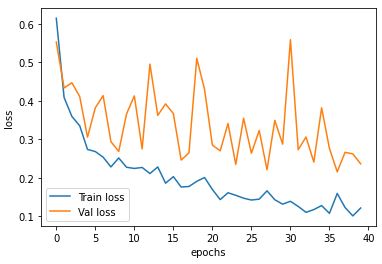
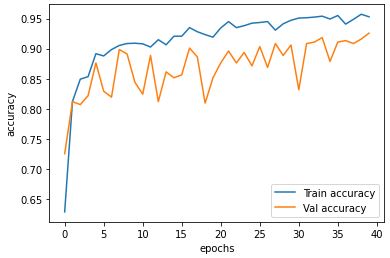


Figure 1.3.4c - results from data augmentation experiment 4

As can be seen by comparing figure 1.3.4b and 1.3.4c, the validation loss of the baseline model increases steadily after 20 epochs while in the model with data augmentation, the validation loss fluctuates but is also more highly correlated with the training loss. This indicates that overfitting has been reduced in the updated CNN.

The next technique used to try and improve the network was adding dropout layers.

A dropout layer was added between the flatten layer and the fully connected layer with a rate of 0.5. This was implemented in the **improved\_cnn\_model** function (figure 1.3.8b). The rate of 0.5 was chosen as an intermediate starting point. If the model started to take too long to converge it could be decreased. Conversely, if the model was still strongly overfitting it could be increased.

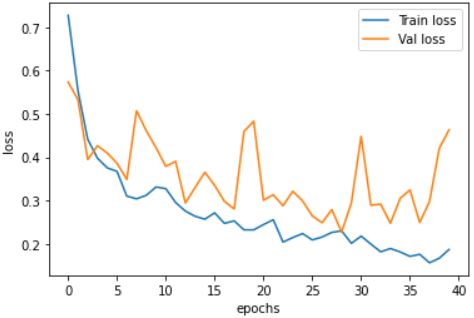
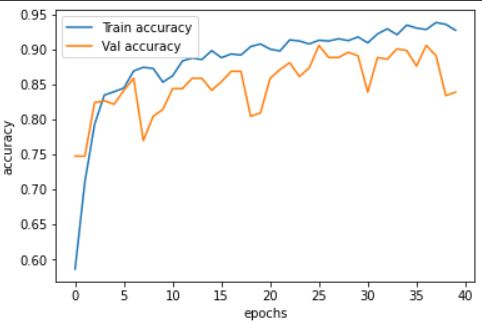


Figure 1.3.4d - results from adding dropout to network

As can be seen in figure 1.3.4d, adding dropout appears to have further reduced overfitting in the network. However it has also highlighted 2 problems:

1. The network looks like it could further reduce its loss meaning not enough epochs
2. The instability of the validation loss produced a set of final weights that have worse than the average performance of the weights throughout training, with a final validation accuracy of only 83%

Both of these problems can be alleviated by introducing early stopping. This is a simple concept - stop the training early if a specific metric fails to improve at a defined rate. It means that an arbitrarily large number of epochs can be defined and the network should stop training when loss stops decreasing. This can be implemented in Keras by using the EarlyStopping callback function. A useful metric to monitor is validation loss as it will stop the training if the model starts to overfit. Another useful feature of Keras early stopping is the ability to save the weights of the model at the best epoch. This was the final improvement made to the network.

### 1.3.5 Testing & Results

Training the model with early stopping and max epochs set to 100 yielded the following accuracy and loss graphs:

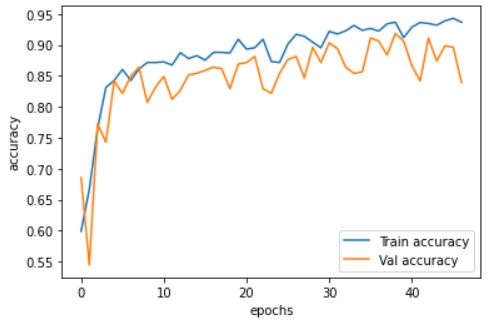


Figure 1.3.5a - final model accuracy plot

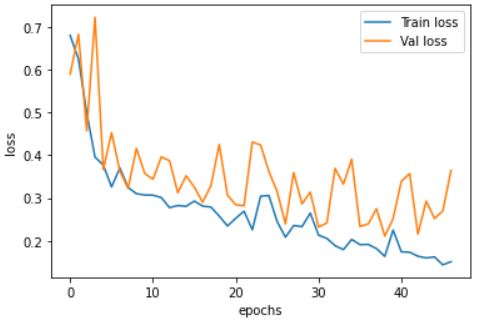


Figure 1.3.5b - final model loss plot

With early stopping implemented, the model stops training just before it seems to start overfitting. Because of the early stopping retore\_best\_weights being active, the final validation accuracy would be from the epoch with the lowest validation loss.

Hence, the final validation accuracy was 91.83% while the loss was 0.2113



Figure 1.3.5c - validation metrics at epoch 39

The model was evaluated with the holdout test data to get the final model performance evaluation. The final test accuracy was 92.12% while the test loss was 0.2. The code for the final evaluation can be found in figure 1.2.8i.

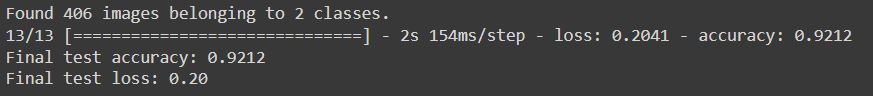


Figure 1.3.5d - final test evaluation

The cost of training was measured in seconds and totaled 1018.

### 1.3.6 Discussion & Conclusion

Through experimentation a marginal increase in performance from the baseline model was achieved. This was mostly by applying regularisation techniques to reduce overfitting. The performance was enhanced from the baseline test accuracy and loss of 91.87% and 4.0 to the improved model test accuracy and loss of 92.12% and 2.0. The main success was reducing the loss caused by overfitting.

However the computational cost was greatly increased, in this case by almost 300%. This shows that a small performance increase in the network can require a large increase in the training cost.

### 1.3.7 References

[1]

F. Chollet, “Chapter 4. Fundamentals of machine learning,” in *Deep Learning with Python*, 1st edition., Shelter Island, New York: Manning Publications Co., 2018.

[2]

P. F. Whelan, “Convolutional Neural Networks (2),” in *Computer Vision Course Notes – EE544*, Dublin City University, 2022.

[3]

neuralthreads, “Dropout — Regularization technique that clicked in Geoffrey Hinton’s mind at a bank,” *Medium.com*, 2021.<https://medium.com/@neuralthreads/dropout-regularization-technique-that-clicked-in-geoffrey-hintons-mind-at-a-bank-fa7fa8c5e1fb>

### 1.3.8 Appendix

# Preprocess images and apply data augmentation to the train data

# Transformations to be applied: rotation, shift, shear, zoom, horizontal flip

def augment\_train\_data(target\_size, batch\_size, class\_mode):

train\_datagen = ImageDataGenerator(

rescale=1./255,

rotation\_range=25,

height\_shift\_range=0.15,

width\_shift\_range=0.15,

horizontal\_flip=True,

shear\_range=0.2,

zoom\_range=0.2,

fill\_mode="nearest"

)

train\_gen = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=target\_size,

batch\_size=batch\_size,

class\_mode=class\_mode,

)

# No augmentation to be done to validation data

validate\_datagen = ImageDataGenerator(rescale=1./255)

validate\_gen = validate\_datagen.flow\_from\_directory(

validate\_dir,

target\_size=target\_size,

batch\_size=batch\_size,

class\_mode=class\_mode

)

return train\_gen, validate\_gen

Figure 1.3.8a - augment\_train\_data function

def improved\_cnn\_model(hyperparameters):

model = Sequential()

# Input Layer

model.add(Conv2D(

32,

(3, 3),

activation="relu",

strides=hyperparameters["strides"],

dilation\_rate=hyperparameters["dialation\_rate"],

padding="same",

input\_shape=(150, 150, 3)

))

# Hidden Layers

model.add(MaxPooling2D((2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation="relu", strides=(1, 1), dilation\_rate=1, padding="same"))

model.add(MaxPooling2D((2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), activation="relu", strides=(1, 1), dilation\_rate=1, padding="same"))

model.add(MaxPooling2D((2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), activation="relu", strides=(1, 1), dilation\_rate=1, padding="same"))

model.add(MaxPooling2D((2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dropout(0.5))

model.add(Dense(512, activation="relu"))

# Output Layer

model.add(Dense(1, activation="sigmoid"))

return model

Figure 1.3.8b - improved\_cnn\_model function implements dropout

# Define hyperparameters and build baseline CNN

hyperparameters = {

"epochs": 40,

"optimizer": "adam",

"dialation\_rate": 1,

"strides": (1, 1)

}

def run\_baseline(hyperparameters):

train\_gen, val\_gen = stage\_train\_data((150, 150), 32, "binary")

model = build\_model("baseline\_bin\_class", hyperparameters)

history = train\_model(model, train\_gen, val\_gen, hyperparameters)

return model, history

def run\_improved\_bin\_class(hyperparameters):

train\_gen, val\_gen = augment\_train\_data((150, 150), 32, "binary")

model = build\_model("improved\_bin\_class", hyperparameters)

history = train\_model(model, train\_gen, val\_gen, hyperparameters)

return model, history

#model, history = run\_baseline(hyperparameters)

model1, history1 = run\_improved\_bin\_class(hyperparameters)

Figure 1.3.8c - run\_improved\_bin\_class function

## 1.3 Visualise Intermediate CNN Outputs (1d)

### 1.4.1 Introduction

The purpose of this activity is to visualise the intermediate outputs of the convolutional layers in the improved model designed in section 1.2.

Regrettably this section was left incomplete due to time constraints.

### 1.4.2 Implementation

### 1.4.3 Testing & Results

### 1.4.4 Discussion & Conclusion

### 1.4.5 References

### 1.4.6 Appendix

# 2 Multi Classification - Transfer Learning

## 2.1 Data train/validation/test split (2a)

### 2.1.1 Introduction

The ImageWoof dataset has 10 dog classes with 1350 samples each.

The goal of this activity is to:

* Select any 5 classes.
* Divide each class into an appropriate train/validate/test split.

### 2.1.2 Rational

This is a multi classification problem. Considering that transfer learning will be used rather than training the model from scratch, a higher percentage of validation and test images may be possible. A researcher claimed to have obtained an accuracy of 85% on a medical imaging CNN using transfer learning with a training sample size of 400 (100 per class) [1].

The improved holdout method will be used for cross validation. One of the justifications for its use in the binary classification problem was that it is significantly less computationally demanding than k-fold or random subsampling. In this multi-classification task there is 2.5 times more data so efficient use of computing resources is still a concern.

A split of 60%, 20%, 20% for training validation and test samples was chosen. This would give 810 samples per class for the training set, and 270 samples per class for both the validation and test sets. A higher number of validation samples will mean lower validation variance - this is helpful for tuning hyperparameters.

### 2.1.3 Design

Using a Python script in Colab to split the data into directories on Google Drive remains the most sensible option.

Some improvements will be made on the script used in section 1 to make it possible to use with N different classes rather than just 2.

### 2.1.4 Implementation

Mounting Google Drive was the first step, giving Colab access to the drive file system.

The 3 functions (figure 2.1.8b) used for data splitting in section 1.1 were adapted. Rather than hard coding the chosen classes, a Python dictionary is defined with the 5 classes to be used in the model. The key is the original ImageWoof class name and the value is the new name assigned to it. To summarise the purpose of the functions:

* **init\_run** - Creates the train, validate and test directories in the run directory. Then creates a subdirectory for each class in each of the 3 to facilitate Keras flow\_from directory.
* **drop\_classes** - Copies all samples of the chosen classes into the run directory.
* **split\_data** - Copies the samples obtained from drop\_classes into the correct class subdirectory (which was created in init\_run). The number of samples that go in train, validate or test are defined as ranges in for loops.

### 2.1.5 Testing & Results

Some tests were performed to ensure that there were no errors in the data splitting. The function **get\_n\_samples** checks and displays the number of samples in each of the train, validate and test directories. Using a split of 60% 20% 20% should yield 4050, 1350 and 1350 samples.

To verify that each sample appears in exactly 1 of either the train validate or test datasets, the **check\_data\_leaks** function was written. It uses Python sets to check for intersections between the 3 datasets. If ‘False’ is printed for all 3 possible intersections, then it can be said that there are no data leaks between the sets.

The output of running these tests was the following:

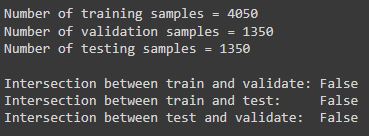


Figure 2.1.5a - data split tests output

This was the expected output and gave confirmation that incorrect data splitting would not be a factor in the model training.

### 2.1.6 Discussion & Conclusion

The chosen data split of 60/20/20, unlike the 70/15/15 split used for the binary classification, gives fewer training samples per class. The tradeoff was made because having less training samples. It was noted how in section 1 using data augmentation was successful at reducing overfitting, and this can be used again to enrich a training set with fewer samples. This is not the case with the validation and testing datasets which must not be altered via augmentation. So for this experiment, a higher percentage of validation and test samples will mean more accurate model evaluation.

### 2.1.7 References

[1]

R. Reif, “Impact of Sample Size on Transfer Learning,” *thisismetis.com*, 2018.<https://www.thisismetis.com/blog/impact-of-sample-size-on-transfer-learning>

### 2.1.8 Appendix

# Mount google drive and define locations of original, unmodified imagewoof data

from google.colab import drive, files

import os

drive.mount("/content/drive", force\_remount=True)

working\_dir = "/content/drive/MyDrive/Colab Notebooks/"

imagewoof\_dir = os.path.join(working\_dir, "data/imagewoof-320")

original\_train\_dir = os.path.join(imagewoof\_dir, "train")

original\_test\_dir = os.path.join(imagewoof\_dir, "val")

Figure 2.1.8a - mounting the drive

from pathlib import Path

from shutil import copyfile

from distutils.dir\_util import copy\_tree # import necessary to circumvent issues with using shutils.copytree in Python v3.6

# "n02088364": beagle

# "n02093754": border\_terrier

# "n02099601": golden\_retriever

# "n02111889": samoyed

# "n02115641": dingo

chosen\_classes = {

"n02088364": "beagle",

"n02093754": "border\_terrier",

"n02099601": "golden\_retriever",

"n02111889": "samoyed",

"n02115641": "dingo"

}

# Create 2 class folders that will contain all instances of their respective class

# Create 3 directories for train validate and test data

def init\_run():

Path(run\_dir+"/train").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/validate").mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/test").mkdir(parents=True, exist\_ok=True)

for c in chosen\_classes.keys():

Path(run\_dir+"/"+chosen\_classes[c]).mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/train/"+chosen\_classes[c]).mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/validate/"+chosen\_classes[c]).mkdir(parents=True, exist\_ok=True)

Path(run\_dir+"/test/"+chosen\_classes[c]).mkdir(parents=True, exist\_ok=True)

# Only copy the classes to be used in the model to the run directory

def drop\_classes():

try:

# Copy all images in train/class\_name and val/class\_name to cathedral dir

for c in chosen\_classes.keys():

copy\_tree(original\_train\_dir+"/"+c, run\_dir+"/"+chosen\_classes[c])

copy\_tree(original\_test\_dir+"/"+c, run\_dir+"/"+chosen\_classes[c])

except Exception as e:

print(e)

# Get a portion of data from each of the classes and put it the the appropriate dir

# Prior testing shows that the classes are perfectly balanced meaning that we can allocate the same number of each class to train validate and test dirs

def split\_data():

train\_split = 810

validate\_split = train\_split + 270

test\_split = validate\_split + 270

class\_samples = {}

for c in chosen\_classes.keys():

class\_name = chosen\_classes[c]

class\_samples[class\_name] = os.listdir(run\_dir+"/"+class\_name)

print("{} class {} samples".format(len(class\_samples[class\_name]), class\_name))

for i in range(0, train\_split):

copyfile(run\_dir+"/"+class\_name+"/"+class\_samples[class\_name][i], run\_dir+"/train/"+class\_name+"/"+class\_samples[class\_name][i])

for i in range(train\_split, validate\_split):

copyfile(run\_dir+"/"+class\_name+"/"+class\_samples[class\_name][i], run\_dir+"/validate/"+class\_name+"/"+class\_samples[class\_name][i])

for i in range(validate\_split, test\_split):

copyfile(run\_dir+"/"+class\_name+"/"+class\_samples[class\_name][i], run\_dir+"/test/"+class\_name+"/"+class\_samples[class\_name][i])

# run\_dir is where data the may be manipulated during runtime is put

run\_dir = os.path.join(imagewoof\_dir, "run")

init\_run()

drop\_classes()

split\_data()

train\_dir = os.path.join(run\_dir, "train")

validate\_dir = os.path.join(run\_dir, "validate")

test\_dir = os.path.join(run\_dir, "test")

Figure 2.1.8b - init\_run, drop\_classes and split\_data functions

## 2.2 InceptionV3 Fine-Tuning Based Transfer Learning (2b)

### 2.2.1 Introduction

Transfer learning is the idea of reusing a pre-trained network on a new (but similar) problem. A simple version of transfer learning is to remove the top layers of the pre-trained base network - which are responsible for the more application specific feature extraction, and replace them with a custom classifier trained on the new dataset [2]. The weights of the base network are not modified during training

A more elegant transfer learning method is called ‘fine tuning’. It involves following the aforementioned process, and then doing an additional training round - this time allowing some of the base network layers to be modified through backpropagation along with the custom classifier. This is the approach that will be adopted.

The pre-trained network to be used is Google’s Inception v3. Inception v3 is a CNN that uses convolutional, fully connected, max pooling, average pooling, concatenation, and dropout layers. It is capable of achieving an accuracy of 78.1% on Imagenet [1].

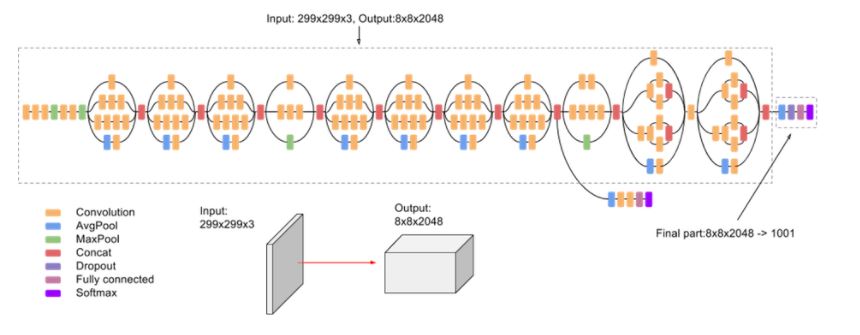


Figure 2.2.1a - Inception v3 source https://cloud.google.com/tpu/docs/inception-v3-advanced

The objective of this activity is to train a CNN model with transfer learning fine tuning based on Inception v3. A reasonable goal would be to train a network with an accuracy greater than 90%.

### 2.2.2 Rational

There were some considerations to be made before implementing the model. It was expected that training would be more computationally expensive than the binary classifier in section 1. The choice had to be made whether to stick to Colab or use a more powerful cloud computing platform such as AWS or Google Cloud. Ultimately a compromise was made - rather than migrating to an entirely different platform, the Colab Pro subscription was acquired which gives access to more powerful GPUs and more GPU time.

Another consideration was which dataset Inception v3 should be pre-trained on. Imagenet was the most obvious choice. This is because:

1. Imagenet is one of the largest and most comprehensive open-source computer vision datasets.
2. The dataset used for this experiment. ‘ImageWoof’ is a subset of Imagenet. This means that the Inception v3 base network is guaranteed to have been trained on data with sufficient resemblance to the new data.
3. This option is integrated with Keras and will require minimal setup.

### 2.2.3 Design

The only hard requirements for this activity is that:

* Fine tuning based transfer learning must be used.
* Inception v3 must be used as the convolutional base.
* Only the top 2 blocks of Inception are to be retrained as part of the fine tuning.
* The input image size must be 299 x 299 as per Inception v3 specification.

The architecture of the custom classifier will have 4 layers. The layer after the convolutional base will be a flatten layer. This will convert the filter outputted by the convolutional layer into a one dimensional feature array suitable for feeding into a dense layer. Between the flatten and fully connected layer there will be a dropout layer to regularise the model. The dense layer will feed into the output layer. The final layer will be fully connected with 5 neurons and will use the softmax activation function since the task is multi-classification, with mutually exclusive classes.

The general steps for doing the fine-tuning based transfer learning will be:

1. Initialise Inception v3 convolutional base with imagenet weights
2. Add the custom classifier to the top of the base
3. Freezing the convolutional base
4. Training the model
5. Unfreezing the top 2 blocks of the convolutional base
6. Fine-tuning the weights by retraining the network

The metric to be used for evaluating the model performance will be accuracy. Accuracy is a good choice because the dataset is balanced - there are the exact same number of each class. Adam was selected as the optimizer. This optimiser worked well on the Imagenette dataset which like ImageWoof is a subset of Imagenet.

### 2.2.4 Implementation

The main implementation function is called **fine\_tune\_training** (figure 2.2.8a). This is where the transfer learning algorithm is executed and all the other functions defined below are used.

The first step of implementing the transfer learning based CNN was to prepare the data generators for training and validation. The use of data augmentation regularisation in the binary classification task noticeably reduced overfitting. Following this, the decision was made to use it for the transfer learning model as well. The **augment\_train\_data** (figure 2.2.8b) function first used in section 1 was brought into the notebook. This takes in a target size tuple, batch size integer and class mode string. It outputs a training data generator and validation data generator The transformation values for augmentation which were discovered to be the most effective in section 1.3.4 were unchanged. Augmentation is not applied to the validation data.

The next step was to instantiate the convolutional base by importing InceptionV3 from Keras. It was initialised with weights to be the pretrained Imagenet weights. include\_top was set to False as the default classifier was to be replaced with a custom classifier. The new model with the Inception v3 base and custom classifier was created by using the **custom\_model** function (figure 2.2.8c). It instantiates a new Keras sequential model and adds the convolutional base along with

1. A flatten layer
2. A dropout layer with dropout frequency of 0.5
3. A dense layer with 256 neurons and ReLu activation
4. An output layer with 5 neurons and softmax activation

The convolutional base was frozen to facilitate the feature extraction stage. This was done by simply setting the trainable attribute of convolutional base to False.

The model was then compiled using the **compile\_model** (figure 2.2.8d) function with the categorical cross-entropy loss function since this is a multi-classification task. As stated in 2.2.3, the optimizer was set to adam and the metric was accuracy.

Training the model was done in the **train\_model** function (figure 2.2.8e). This was re-used from section 1. It takes in the compiled model and training parameters and returns the training history. It measures the training cost by printing the time taken in seconds. Early stopping was utilised. This concluded the feature extraction stage and marked the beginning of the fine tuning stage.

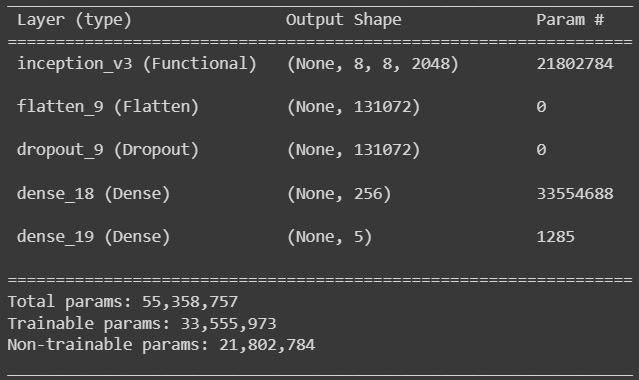


Figure 2.2.4a - summary of model trained at feature extraction stage

To make the top 2 Inception blocks trainable the convolutional base was unfrozen and the first 249 layers (0 to 248) were frozen. The model was recompiled to allow the changes to take place. The model was trained again to fine tune the upper convolutional layers of Inception v3.

The results of the training will be discussed in the next section.

### 2.2.5 Testing & Results

The training almost immediately started to show good results. At the feature extraction stage of training, the validation and test loss both fell rapidly and levelled out close to 0. The validation loss was lower than the training loss. This was because of the dropout layer which only affects training.

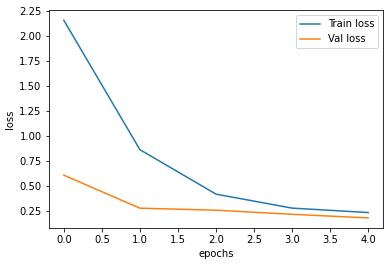
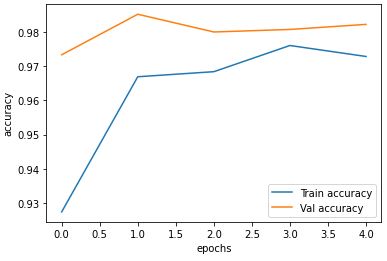


Figure 2.2.5a - feature extraction training and validation accuracy/loss plots

The final validation loss at the end of feature extraction was 0.1821, while the accuracy was 98.22%.

For the fine tuning stage it was difficult to improve upon the already high accuracy. Experimenting with hyperparameters was challenging due to the long training time. As can be seen in the plots in figure 1.2.5b, the early stop cut the training short at a validation accuracy of approximately 96%. Previous experiments with more lenient early stopping showed that validation accuracy would get to around 98% and then fluctuate without making any improvement past this point.

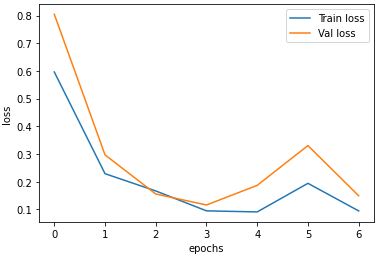
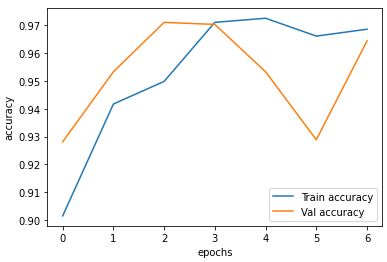


Figure 2.1.5b - feature extraction training and validation accuracy/loss plots

The confusion matrix (code in figure 2.2.8f) had a fairly even distribution. This meant that no one class was getting incorrectly classified as another more often.

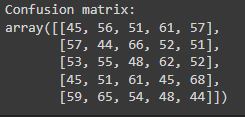


Figure 2.2.5c - confusion matrix

The training cost was measured in seconds. The feature extraction training time took 502 seconds while the fine-tuning training took 711 seconds, bringing the total training time to 1213 seconds. Since Colab does the GPU assignment in the background, the GPU used for training could have been either NVIDIA T4 or P100.

The final mode evaluation (code in figure 2.2.8e) yielded an accuracy of 97.26% and a loss of 0.1. This well exceeded the initial goal of getting a model with an accuracy of above 90%.

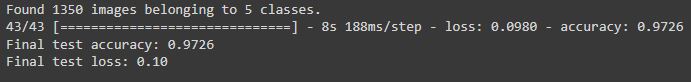


Figure 2.2.5d - final test output

### 2.2.6 Discussion & Conclusion

At the beginning of this exercise it was unclear as to how well the model would perform on the dataset. It was a multi-classification problem with highly correlated classes. This was a step up in difficulty from the binary classification task. Using the fine tuning method of transfer learning, a model with accuracy of over 97% was produced.

However the experiment didn’t go entirely as planned. The feature extraction phase of training produced a model with higher validation accuracy (98%) than that of the subsequent fine tuning model. It is possible that with more computing resources and using SGD with momentum and decay, the validation accuracy of the fine tuning model could have surpassed the feature extraction model accuracy.

### 2.2.7 References

[1]

Google, “Advanced Guide to Inception v3.” [Online]. Available:<https://cloud.google.com/tpu/docs/inception-v3-advanced#estimator_api>

[2]

P. F. Whelan, “Convolutional Neural Networks (2),” in *Computer Vision Course Notes – EE544*, Dublin City University, 2022.

### 2.2.8 Appendix

# Implement transfer learing by apply fine-tuning approach

def fine\_tune\_training(hyperparameters):

train\_gen, val\_gen = augment\_train\_data((299, 299), 32, "categorical")

# Initialise InceptionV3 network with weights trained on Imagenet. Don't include top as we will be adding our own

conv\_base = InceptionV3(weights="imagenet", include\_top=False, input\_shape=(299, 299, 3))

# print("Structure of Inception v3 convolutional base")

# conv\_base.summary()

# Add custom layers to the top of the base

model = custom\_model(conv\_base)

# Freeze the InceptionV3 convolutional layers

conv\_base.trainable = False

# Compile the model

model = compile\_model(model, hyperparameters)

# Check model structure

model.summary()

# Define early stopping callback

early\_stopping = EarlyStopping(monitor="val\_loss", patience=3, restore\_best\_weights=True)

# Train the custom classifier

history1 = train\_model(model, train\_gen, val\_gen, hyperparameters["feature\_ex\_epochs"], [early\_stopping])

plot\_metrics(history1)

# Since the last 2 blocks are being tuned, freeze the first 8 blocks. There are 31 layers per block so freeze up until 31\*8 = 248

conv\_base.trainable = True

for l in conv\_base.layers[0:249]:

l.trainable = False

# Compile the model again

model.compile(loss="categorical\_crossentropy", optimizer=hyperparameters["optimizer"], metrics=["accuracy"])

model.summary()

# Train the model again, fine tuning the Inceptionv3 conv layers

history2 = train\_model(model, train\_gen, val\_gen, hyperparameters["fine\_tuning\_epochs"], [early\_stopping])

plot\_metrics(history2)

return model, history1, history2

hyperparameters = {

"feature\_ex\_epochs": 5,

"fine\_tuning\_epochs": 40,

"optimizer": "adam",

}

model, h1, h2 = fine\_tune\_training(hyperparameters)

Figure 2.2.8a - fine\_tune\_training function

# Preprocess images and apply data augmentation to the train data

# Transformations to be applied: rotation, shift, shear, zoom, horizontal flip

def augment\_train\_data(target\_size, batch\_size, class\_mode):

train\_datagen = ImageDataGenerator(

rescale=1./255,

rotation\_range=25,

height\_shift\_range=0.15,

width\_shift\_range=0.15,

horizontal\_flip=True,

shear\_range=0.2,

zoom\_range=0.2,

fill\_mode="nearest"

)

train\_gen = train\_datagen.flow\_from\_directory(

train\_dir,

target\_size=target\_size,

batch\_size=batch\_size,

class\_mode=class\_mode,

)

# No augmentation to be done to validation data

validate\_datagen = ImageDataGenerator(rescale=1./255)

validate\_gen = validate\_datagen.flow\_from\_directory(

validate\_dir,

target\_size=target\_size,

batch\_size=batch\_size,

class\_mode=class\_mode

)

return train\_gen, validate\_gen

Figure 2.2.8b - augment\_train\_data function

# Takes in the convolutional base of a pretrained network, in this case InceptionV3

def custom\_model(conv\_base):

model = Sequential()

model.add(conv\_base)

model.add(Flatten())

model.add(Dropout(0.5))

model.add(Dense(256, activation="relu"))

# Output layer uses softmax activation function because it is a multi-classification task

model.add(Dense(5, activation="softmax"))

return model

Figure 2.2.8c - custom\_model function

# compile the model

def compile\_model(model, hyperparameters):

print("Compiling model")

model.compile(loss="categorical\_crossentropy", optimizer=hyperparameters["optimizer"], metrics=["accuracy"])

print("Model compilation complete")

return model

Figure 2.2.8d - compile\_model function

# Do final evaluation of the model using holdout test data

test\_datagen = ImageDataGenerator(rescale=1./255)

test\_generator = test\_datagen.flow\_from\_directory(

test\_dir,

target\_size=(299, 299),

batch\_size=32,

class\_mode="categorical"

)

test\_metrics = model.evaluate(test\_generator)

print("Final test accuracy: {:.4f}".format(test\_metrics[1]))

print("Final test loss: {:.2f}".format(test\_metrics[0]))

Figure 2.2.8e - final evaluation code

# Get confusion matrix

from sklearn.metrics import confusion\_matrix, classification\_report

pred = model.predict(test\_generator)

predictions = np.argmax(pred, axis=1)

print("Confusion matrix: ")

confusion\_matrix(test\_generator.classes, predictions)

Figure 2.2.8f - confusion matrix

## 2.3 Test Network Performance with Unseen Data (2c)

### 2.3.1 Introduction

The performance of a model in a controlled test environment and in the outside world can vary greatly. The goal of this exercise is to procure some ‘in the wild’ data for the dog breed classifier and test it to see how well it performs. At least one image of each class needs to be tested.

### 2.3.2 Implementation

Getting the images from sources other than the internet proved difficult. The next best thing was to use images that were highly unlikely to have been used in Imagenet (or its subset Imagewoof). The public Facebook page ‘Donegal Dog Barber’ was selected. Here 3 of the classes ‘samoyed’, ‘golden retriever’ and ‘border terrier’ were found. No instances of the ‘beagle’ class were found on the page and it was not likely that the ‘dingo’ class could be found in Donegal as a whole. Images of these 2 classes were sourced from Wikipedia.



Figure 2.3.2a - retriever-3 image sourced from Donegal Dog Barber

The function **classify\_image** (figure 2.3.6a) was written. It takes in a trained model and image path. It converts the image to a numpy array and transforms it such that it can be inputted to the model. The model makes a prediction and prints the result.

### 2.3.3 Testing & Results

A total of 11 images were tested. The class breakdown was 3 border terriers, 3 golden retrievers, 3 samoyeds, 1 beagle and 1 dingo. Running the code shown in figure 2.3.6a produced the following output:

|  |
| --- |
| Classifying image /content/drive/MyDrive/Colab Notebooks/data/samoyed-1.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: samoyed  Classifying image /content/drive/MyDrive/Colab Notebooks/data/retreiver-1.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: golden\_retriever  Classifying image /content/drive/MyDrive/Colab Notebooks/data/dingo-1.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: dingo  Classifying image /content/drive/MyDrive/Colab Notebooks/data/Beagle\_1.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: beagle  Classifying image /content/drive/MyDrive/Colab Notebooks/data/terrier-1.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: border\_terrier  Classifying image /content/drive/MyDrive/Colab Notebooks/data/samoyed-2.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: samoyed  Classifying image /content/drive/MyDrive/Colab Notebooks/data/terrier-2.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: border\_terrier  Classifying image /content/drive/MyDrive/Colab Notebooks/data/retreiver-2.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: golden\_retriever  Classifying image /content/drive/MyDrive/Colab Notebooks/data/samoyed-3.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: samoyed  Classifying image /content/drive/MyDrive/Colab Notebooks/data/terrier-3.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: border\_terrier  Classifying image /content/drive/MyDrive/Colab Notebooks/data/retreiver-3.jpg  Shape: (299, 299, 3)  Expanded Shape: (1, 299, 299, 3)  Predicition: golden\_retriever |

### 2.3.4 Discussion & Conclusion

The fine tuning based classifier performed very well on real life data as all the predictions were correct. This was in the expected range because the final test accuracy was above 97%, so for the 11 images we would expect no more than 1 incorrect prediction.

### 2.3.5 References

“Donegal Dog Barber” used for testing images

### 2.3.6 Appendix

from sklearn import preprocessing

from tensorflow.keras.models import load\_model

from tensorflow.keras.preprocessing import image

# Classify image located in image\_path with given model

def classify\_image(model, image\_path):

print("Classifying image {}".format(image\_path))

img = image.load\_img(image\_path, target\_size=(299, 299))

# Convert image to numpy array

img\_array = image.img\_to\_array(img)

print("Shape: {}".format(img\_array.shape))

# Expand and rescale image array

img\_array = np.expand\_dims(img\_array, axis=0)

img\_array = np.array(img\_array).astype('float32')/255

print("Expanded Shape: {}".format(img\_array.shape))

# Make prediciton and display class

class\_names = ["beagle", "border\_terrier", "dingo", "golden\_retriever", "samoyed", ]

predicted\_class = model.predict(img\_array)

print("Predicition: {}".format(class\_names[np.argmax(predicted\_class)]))

data\_dir = os.path.join(working\_dir, "data")

# model = load\_model(data\_dir+"/2b-dog\_breed\_fine\_tuning.h5")

classify\_image(model, data\_dir+"/samoyed-1.jpg")

print()

classify\_image(model, data\_dir+"/retreiver-1.jpg")

print()

classify\_image(model, data\_dir+"/dingo-1.jpg")

print()

classify\_image(model, data\_dir+"/Beagle\_1.jpg")

print()

classify\_image(model, data\_dir+"/terrier-1.jpg")

print()

classify\_image(model, data\_dir+"/samoyed-2.jpg")

print()

classify\_image(model, data\_dir+"/terrier-2.jpg")

print()

classify\_image(model, data\_dir+"/retreiver-2.jpg")

print()

classify\_image(model, data\_dir+"/samoyed-3.jpg")

print()

classify\_image(model, data\_dir+"/terrier-3.jpg")

print()

classify\_image(model, data\_dir+"/retreiver-3.jpg")

Figure 2.3.6a - classify\_image function and usage