NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CE/CZ4041 Machine Learning

Group Project: Dog Breed Identification

Group 23

Name	Matriculation Number
Hsiao Chia Yu	U1821839L
Lew Jian Chung Kenny	U1821042H
Li Wei-Ting	U1823750J
Lim Yong Xuan Isaac	U1821486G
Neo Shun Xian Nicholas	U1820539F

Tables of Content

Roles in this Project

Hsiao Chia Yu
Image Preprocessing, I/O Optimization, Feature Extraction & Ensemble Learning

Lew Jian Chung Kenny
Training of Models, Presentation slides, Video

Li Wei-Ting
Image Preprocessing, Research, Model Exploration

Lim Yong Xuan Isaac
Training of Models, Presentation Slides, Video

Neo Shun Xian Nicholas
Image Preprocessing, Feature Extraction & Ensemble Learning, Hyperparameter Tuning

Introduction

Problem Statement

There are about 7000 stray dogs in Singapore. Stray dogs in general are a threat as they may bite humans or spread diseases such as rabies. The main cause of stray dogs' problem is likely to be the dog owners abandoning their dog due to unfamiliarity with the dog's characteristics. To prevent such issues, we aimed to devise a dog breed identification app to guide users about a particular dog's characteristics for the soon-to-be dog owners. As such, when they encounter a dog of their liking, they can use our app to snap a photo of the dog to find out its breed and its characteristics. Hence, this provides soon-to-be dog owners with more information to assist them in making the right decision to pet that particular dog, thereby reducing the cases of abandoned dogs in the future. However, our app requires a dog breed classifier first before we could even deliver the app. As such, the objective for this project is to build a classifier to identify the dog breeds given the dog images.

The Dog Breed Dataset

The dog breed dataset consists of a training set with 10,222 images of 120 dog breeds. Each image has a filename that is its unique id. It also consists of a test set with 10,357 images which are unlabelled. The labels.csv provided in the dataset shows the filename id and the dog breed class of the training set.

Aim of our project

The aim of this project is to determine the dog breeds from the dog images with at least 90% accuracy on the validation set of the data.

Project Overview

The overview of the project pipeline is shown in Figure 0 below:

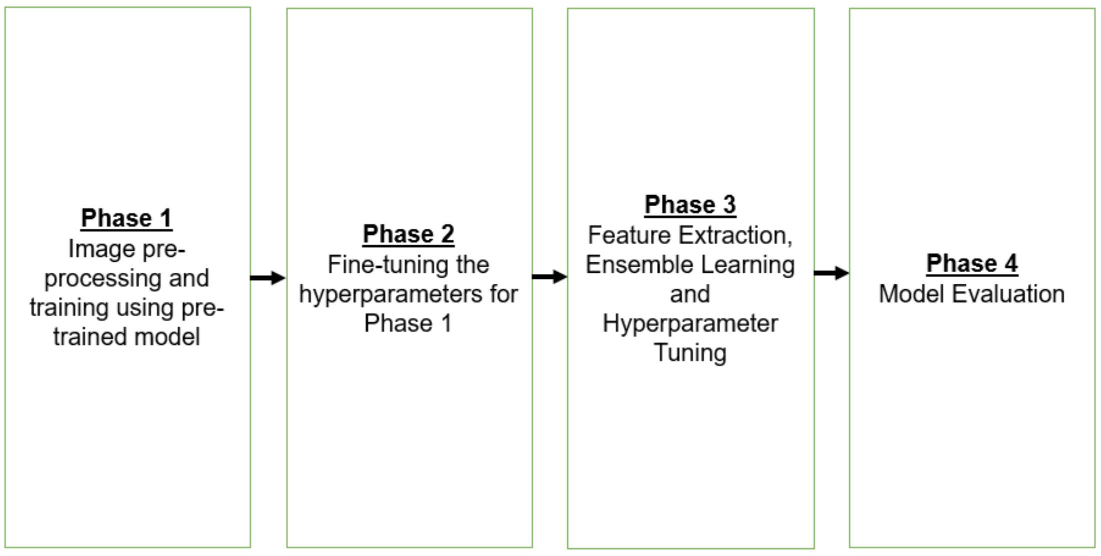
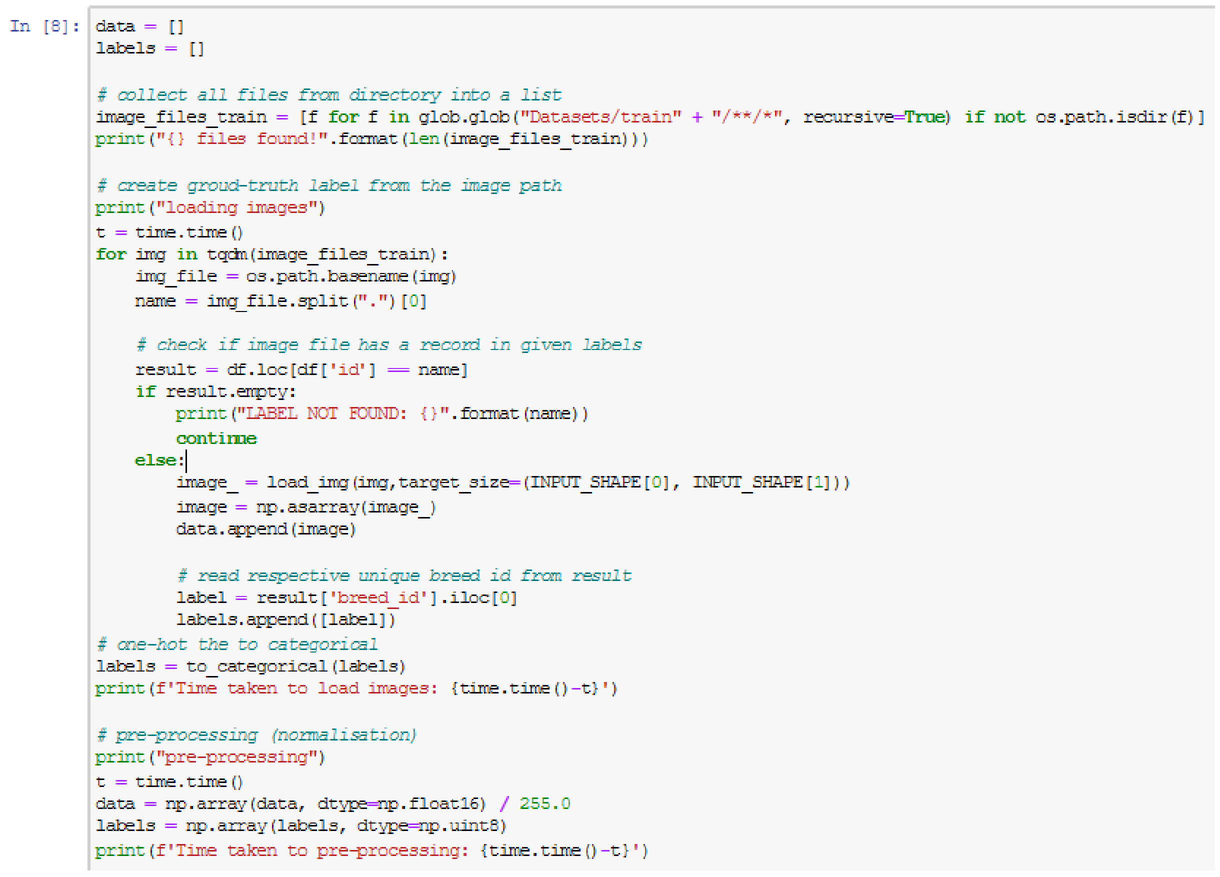

Phase 1
Image pre-processing and training using pre-trained model
Phase 2
Fine-tuning the hyperparameters for Phase 1
Phase 3
Feature Extraction, Ensemble Learning and Hyperparameter Tuning
Phase 4
Model Evaluation

Figure 0: Project Overview

Phase 1: Image pre-processing & training using a single pre-trained model

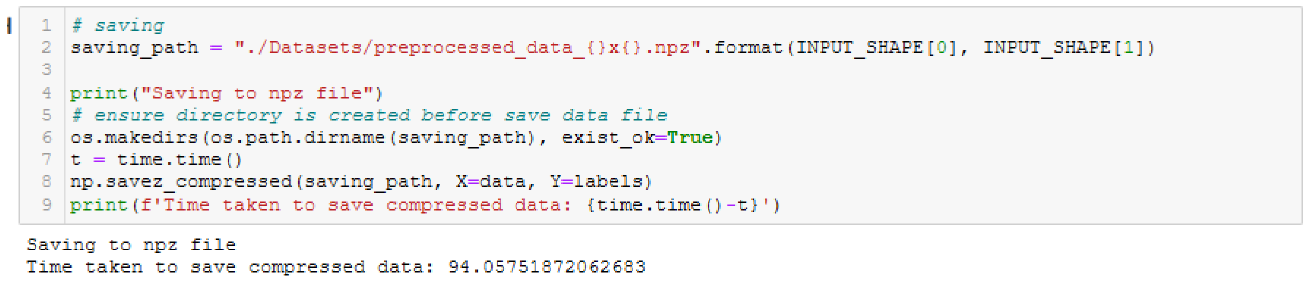
Image pre-processing

Loading the images individually from disk for training is relatively inefficient. It takes 50.6 seconds to completely load all the training images for each runtime. We will also need to resize our images into a standard size for training of the models. We will be resizing the images to 128 by 128 pixels in this phase. Resizing of images takes another 4.3 seconds. As such, we use the numpy savez_compressed library to compress all of our images into one single .npz file so that every time we call the images, we only need to call the .npz file instead of loading the images individually. Saving the .npz file takes 94.1 seconds but it is only run once to generate the .npz file. Subsequent loading of the .npz files only requires 4.63 seconds for each runtime. Figure 1.1 and Figure 1.2 below show code snippets of how we convert the individual images into a numpy array and compress them into a single .npz file respectively.



```
In [8]: data = []
        labels = []

        # collect all files from directory into a list
        image_files_train = [f for f in glob.glob("Datasets/train" + "/**/*", recursive=True) if not os.path.isdir(f)]
        print("{} files found!".format(len(image_files_train)))

        # create groud-truth label from the image path
        print("loading images")
        t = time.time()
        for img in tqdm(image_files_train):
            img_file = os.path.basename(img)
            name = img_file.split(".")[0]

            # check if image file has a record in given labels
            result = df.loc[df['id'] == name]
            if result.empty:
                print("LABEL NOT FOUND: {}".format(name))
                continue
            else:
                image_ = load_img(img,target_size=(INPUT_SHAPE[0], INPUT_SHAPE[1]))
                image = np.asarray(image_)
                data.append(image)

                # read respective unique breed id from result
                label = result['breed_id'].iloc[0]
                labels.append([label])
        # one-hot the to categorical
        labels = to_categorical(labels)
        print(f'Time taken to load images: {time.time()-t}')

        # pre-processing (normalisation)
        print("pre-processing")
        t = time.time()
        data = np.array(data, dtype=np.float16) / 255.0
        labels = np.array(labels, dtype=np.uint8)
        print(f'Time taken to pre-processing: {time.time()-t}')
```

Figure 1.1: Convert individual images into numpy array



```
# saving
saving_path = "./Datasets/preprocessed_data_{}x{}.npz".format(INPUT_SHAPE[0], INPUT_SHAPE[1])

print("Saving to npz file")
# ensure directory is created before save data file
os.makedirs(os.path.dirname(saving_path), exist_ok=True)
t = time.time()
np.savez_compressed(saving_path, X=data, Y=labels)
print(f'Time taken to save compressed data: {time.time()-t}')
```

```
Saving to npz file
Time taken to save compressed data: 94.05751872062683
```

Figure 1.2: Save the images into a compressed .npz file

Model Structure of Transfer Learning

Figure 1.3 shows the model structure that we will adopt in this phase.

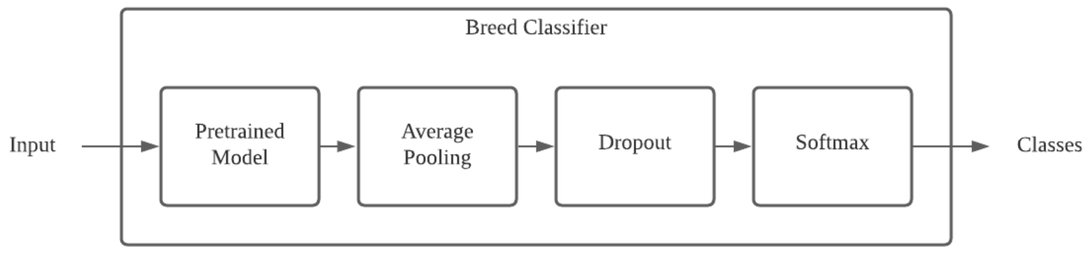
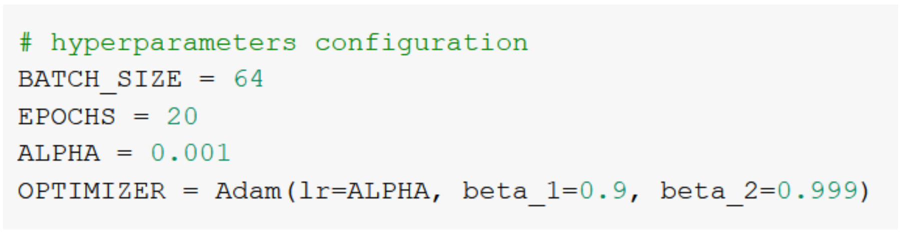


Figure 1.3: Model structure of phase 1

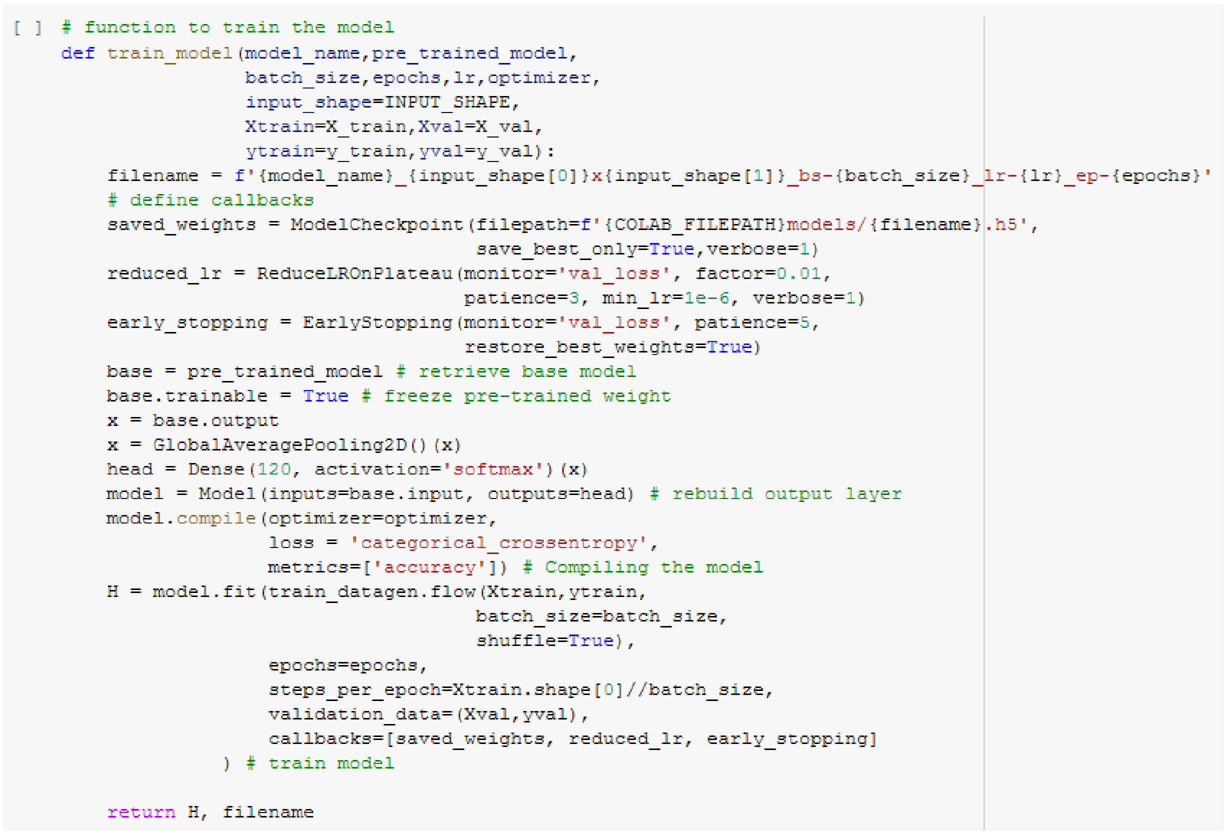
We first split our training data into training and validation sets, with a ratio of 80% training set and 20% validation set. We will then use 16 different pre-trained models to train our model and will then use the validation accuracy as a metric for comparison. We used pre-trained models to train our model because those models have been previously trained on large datasets. Hence, we can directly use the weights and architecture obtained and apply the learning to our problem. This is described as transfer learning. Transfer learning is beneficial because we do not need a lot of data to train the model and the performances of the trained models are generally better. We also use the ImageDataGenerator in the keras preprocessing library for image augmentation to expand the training dataset in order to improve the performance and the ability of the model to generalise since we have 120 classes but only 10222 training images. Moreover, we need to reserve 20% of the data for the validation set and hence, making our training set even smaller.

Figure 1.4 below shows the hyperparameters used in this phase. Figure 1.5 shows how our training function is being defined and Figure 1.6 shows an example of how the training function is being called on a pre-trained model, DenseNet121. Figure 1.7 shows the configurations we did for Image Augmentation.



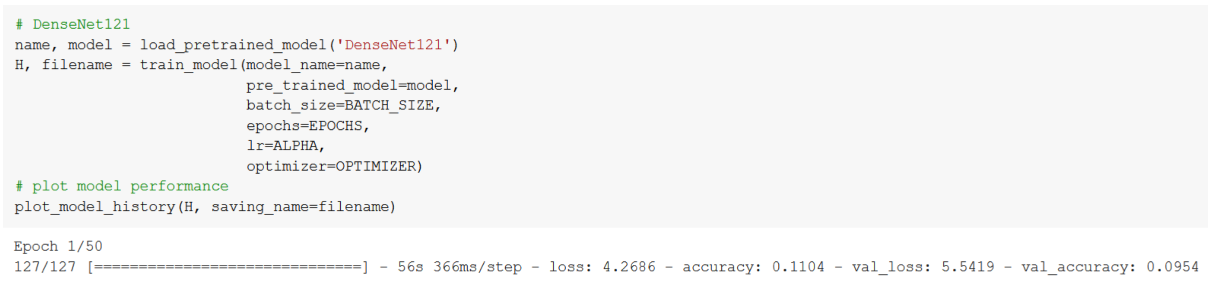
```python
# hyperparameters configuration
BATCH_SIZE = 64
EPOCHS = 20
ALPHA = 0.001
OPTIMIZER = Adam(lr=ALPHA, beta_1=0.9, beta_2=0.999)
```

Figure 1.4: Hyperparameters used for the training



```python
[ ]  # function to train the model
     def train_model(model_name,pre_trained_model,
                     batch_size,epochs,lr,optimizer,
                     input_shape=INPUT_SHAPE,
                     Xtrain=X_train,Xval=X_val,
                     ytrain=y_train,yval=y_val):
         filename = f'{model_name}_{input_shape[0]}x{input_shape[1]}_bs-{batch_size}_lr-{lr}_ep-{epochs}'
         # define callbacks
         saved_weights = ModelCheckpoint(filepath=f'{COLAB_FILEPATH}models/{filename}.h5',
                                         save_best_only=True,verbose=1)
         reduced_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.01,
                                        patience=3, min_lr=1e-6, verbose=1)
         early_stopping = EarlyStopping(monitor='val_loss', patience=5,
                                        restore_best_weights=True)
         base = pre_trained_model # retrieve base model
         base.trainable = True # freeze pre-trained weight
         x = base.output
         x = GlobalAveragePooling2D()(x)
         head = Dense(120, activation='softmax')(x)
         model = Model(inputs=base.input, outputs=head) # rebuild output layer
         model.compile(optimizer=optimizer,
                       loss = 'categorical_crossentropy',
                       metrics=['accuracy']) # Compiling the model
         H = model.fit(train_datagen.flow(Xtrain,ytrain,
                                          batch_size=batch_size,
                                          shuffle=True),
                       epochs=epochs,
                       steps_per_epoch=Xtrain.shape[0]//batch_size,
                       validation_data=(Xval,yval),
                       callbacks=[saved_weights, reduced_lr, early_stopping]
                  ) # train model

         return H, filename
```

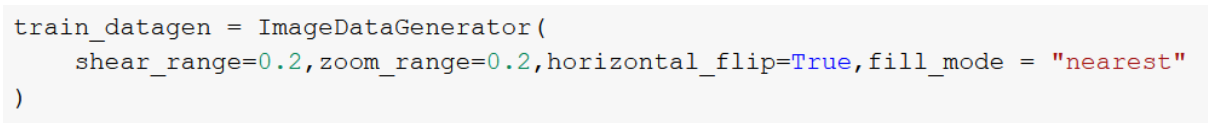
Figure 1.5: Training function



```python
# DenseNet121
name, model = load_pretrained_model('DenseNet121')
H, filename = train_model(model_name=name,
                          pre_trained_model=model,
                          batch_size=BATCH_SIZE,
                          epochs=EPOCHS,
                          lr=ALPHA,
                          optimizer=OPTIMIZER)
# plot model performance
plot_model_history(H, saving_name=filename)
```

```
Epoch 1/50
127/127 [==============================] - 56s 366ms/step - loss: 4.2686 - accuracy: 0.1104 - val_loss: 5.5419 - val_accuracy: 0.0954
```

Figure 1.6: Calling the training function to start training the model



```python
train_datagen = ImageDataGenerator(
    shear_range=0.2,zoom_range=0.2,horizontal_flip=True,fill_mode = "nearest"
)
```

Figure 1.7: Image Augmentation Configurations

Evaluation on single pre-trained models

Table 1.1 shows the performance of the 16 models based on its validation accuracy.

Pre-trained Model	Validation Loss	Validation Accuracy
DenseNet121	2.0054	0.5584
DenseNet169	2.0096	0.5403
DenseNet201	2.1014	0.5619
Xception	2.0886	0.5477
VGG16	4.7834	0.0098
VGG19	4.7835	0.0098
ResNet50	2.3343	0.4298
ResNet101	3.9437	0.1521
ResNet152	3.6561	0.1800
ResNet50V2	4.5995	0.0553
ResNet101V2	4.4987	0.0685
ResNet152V2	2.8603	0.3491
InceptionV3	2.3202	0.4166
InceptionResNetV2	2.2555	0.5462
MobileNet	4.6053	0.0621
MobileNetV2	4.2320	0.1076

Table 1.1: Validation Loss and Validation Accuracy for each of the pre-trained model

Conclusion from this phase

As seen from Table 1.1, the validation accuracy of the models are only between 50% to 60%, which is just slightly better than a random binary guess. To make the comparison fair, we used the same hyperparameters for all of our pre-trained models in this phase. This is to help us choose which pre-trained model has the most potential for fine-tuning, further training and also the steps we need to take next to improve the accuracy of our classifier which will be discussed in the next phase.

Phase 2: Fine-tuning the hyperparameters

In this phase, we will look at three attempts to improve the validation accuracy of the models:

1. Try out on different hyperparameters
    - Batch size
    - Learning Rate
2. Image size increase from 128x128 to 192x192 and changing dtype from float64 to float16
3. Change data type to uint8, add preprocessing after input layer and increase image size

<u>**First attempt: Try out on different hyperparameters**</u>

In this attempt, we will use the model from the previous phase with the highest validation accuracy (i.e **DenseNet201**) and tune the hyperparameters accordingly.

We tried out different values of learning rates and batch sizes, hoping to improve the models' performances. After the modification, the model indeed improved to a maximum of 63%.

Besides the fixed learning rate, we also attempted to dynamically adjust the learning rate with callbacks like LearningRateScheduler and ReduceLROnPlateau. With LearningRateScheduler, we are allowed to tune the learning rate in an exponentially decreasing manner as shown in Figure 2.1. On the other hand, the ReduceLROnPlateau allows the monitoring of validation loss or accuracy and performance reduction of learning rate when loss or accuracy is not improved as shown in Figure 2.2, showing a learning rate reduction to 1e-05 after 15 epochs and reduction to 1e-06 after 20 epochs.

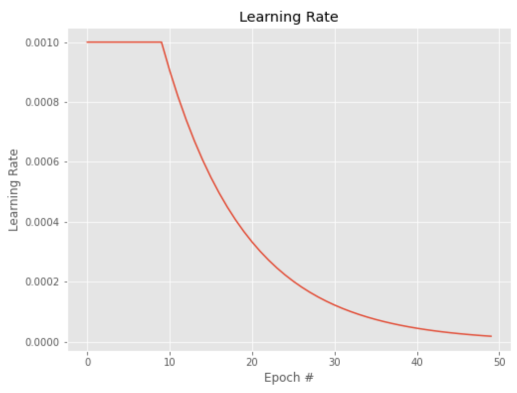


Figure 2.1: Learning Rate Scheduler

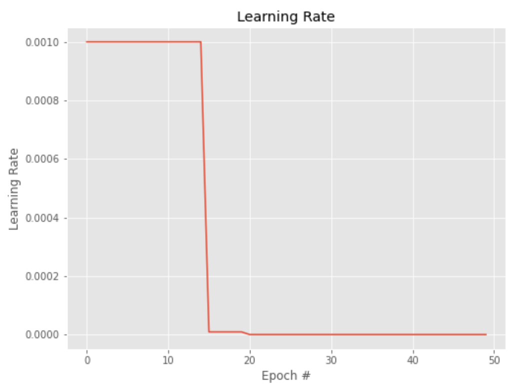


Figure 2.2: ReduceLROnPlateau

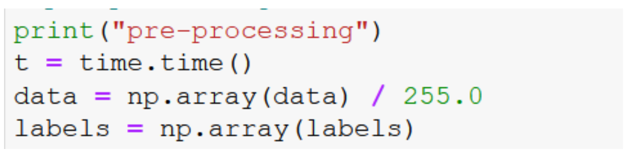
To prevent overfitting, we also adopted EarlyStopping and ModelCheckpoint callback. EarlyStopping allows early termination on the training model once it detects the validation loss plateaued. Whereas ModelCheckpoint allows us to save the best model weight wherever the validation loss decreases.

However, the validation accuracy is still way below our expectation. As such, we resorted to the second attempt as discussed below.

<u>**Second attempt: Image size increase from 128x128 to 192x192 and changing dtype from float64 to float16**</u>

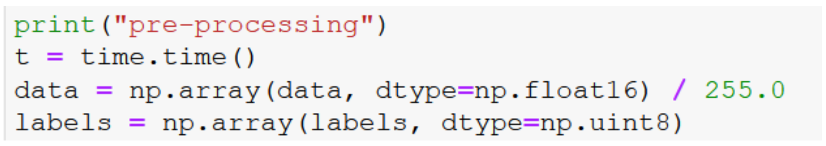
We are conscious that the original pre-trained models are trained with a larger image size, hence trying to provide larger image size might possibly get better performance.

As such, we decided to increase the size of images to fit the minimum of the model's requirement. However, we faced the difficulty of increasing the image size due to GPU not being able to allocate enough memory to train the model. As we monitored the resource usage, we noticed a huge increase in the image data size from 364MB to 3.7GB when loading 128x128 resolution images from the disk into the memory. We were shocked with the increase in size for such downscaled images. The issue was caused by dtype after investigation. We use np.array to store the normalised color information which has a range of 0 to 1. Python uses the default of float64 for 64 bits computers, which caused a huge increase in data size as compared to uint8 encoded images. The extra bits for float64 provide greater precision level, however we do not require such precision, hence we explicitly changed the dtype to float16 allowing image size increased to 192x192 before hitting the hardware limit. Figure 2.3 and Figure 2.4 compare the difference when the dtype parameter is 'float' and 'float16' respectively. Table 2.1 shows the comparison of the amount of space used.



```
print("pre-processing")
t = time.time()
data = np.array(data) / 255.0
labels = np.array(labels)
```

Figure 2.3: Parameters in the np.array class -> default float64 type is declared



```
print("pre-processing")
t = time.time()
data = np.array(data, dtype=np.float16) / 255.0
labels = np.array(labels, dtype=np.uint8)
```

Figure 2.4: Explicit dtype parameters declared in the np.array class -> float16 (image pixels) and uint8 (labels) type are declared

	128x128	128x128	192x192
shape	(10222,128,128,3)	(10222,128,128,3)	(10222,192,192,3)
dtype	float64	float16	float16
size	3.74 GB	958 MB	2.11 GB

Table 2.1: Comparison on the amount of space used

Table 2.2 shows the performance of the 16 models based on its validation accuracy with the increase in image size from 128x128 to 192x192.

Pre-trained Model	Validation Loss	Validation Accuracy
DenseNet121	1.4369	0.6621
DenseNet169	1.3974	0.6641
DenseNet201	1.4147	0.6601
Xception	1.5211	0.6675
VGG16	4.7836	0.0098
VGG19	4.7836	0.0098
ResNet50	1.8648	0.5237
ResNet101	2.3000	0.4049
ResNet152	2.5054	0.4435
ResNet50V2	1.9919	0.5174
ResNet101V2	2.4084	0.4885
ResNet152V2	4.8114	0.0068
InceptionV3	1.8984	0.5990
InceptionResNetV2	1.6065	0.6484
MobileNet	1.5256	0.6367
MobileNetV2	1.5609	0.6337

Table 2.2: Validation Loss and Validation Accuracy for each of the pre-trained model

Generally, as seen from table 2.2, the validation accuracy of most of the models improve from the 50-60% range to the 60-70% range. This shows that there are slight improvements in the models trained when we increased the image size, at the expense of scaling down our data type to float16. However, the performance of the models are still below our expectation and hence we move on to our third attempt.

Third attempt: Change data type to uint8, adding preprocessing after input layer and increase image size to 224x224

We then attempt to experiment with another method, which is to compress all of our images into a .npz file, with dtype uint8, and increasing our image size. From Table 2.2, we see that the pre-trained model Xception performs the best among all other pre-trained models. Here, we only increase our image size to 224x224 due to resource limitations. We do not normalise the images first while saving the .npz file, because normalising the images during pre-processing is computationally expensive for a large dimension like 224x224. We normalise our images when we are building our model. The normalisation will be the first layer of the model and then we add Xception after that. Figure 2.6 shows the proposed model structure for this attempt.

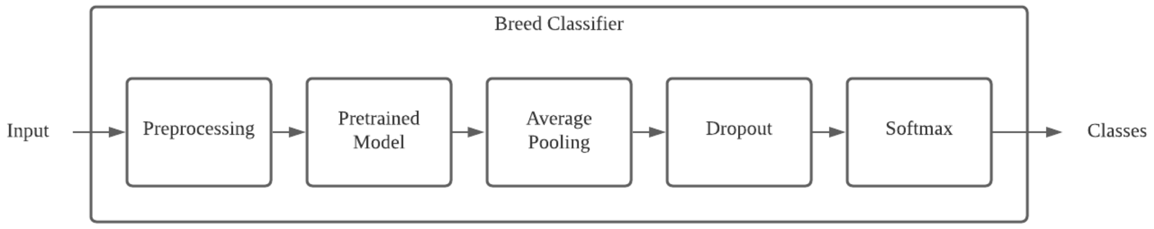
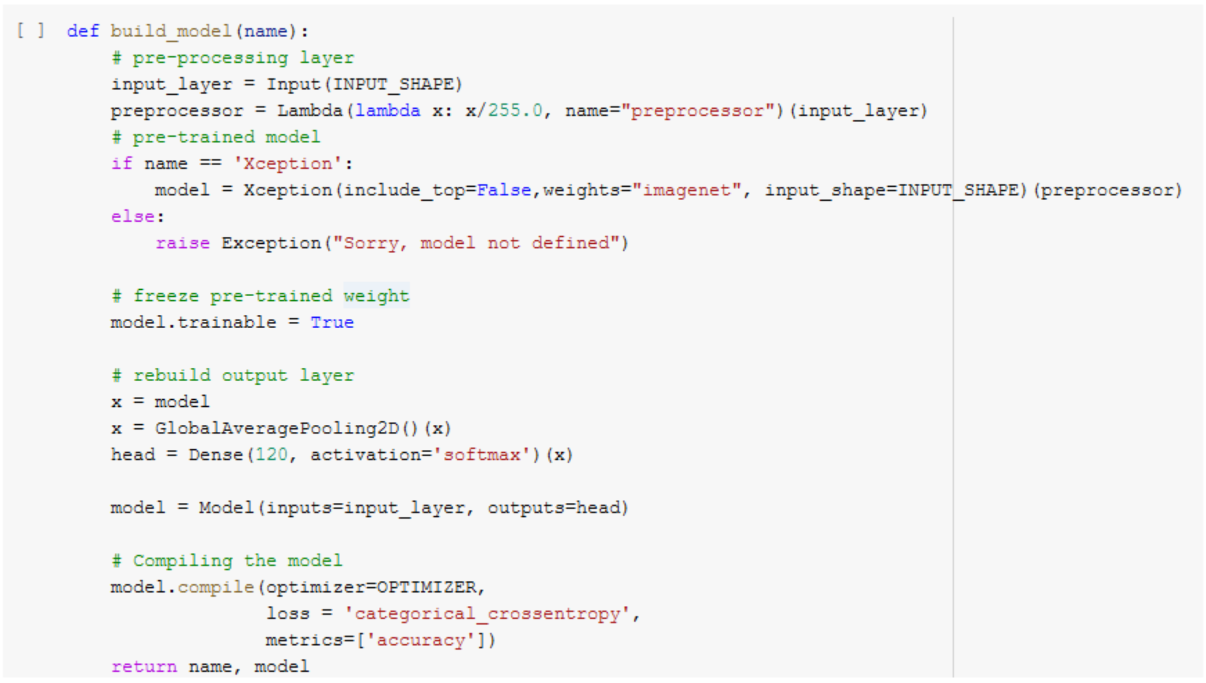


Figure 2.6: Model Structure for the third attempt

Figure 2.7 shows a code snippet of the model structure and also the normalisation.



```
[ ]  def build_model(name):
         # pre-processing layer
         input_layer = Input(INPUT_SHAPE)
         preprocessor = Lambda(lambda x: x/255.0, name="preprocessor")(input_layer)
         # pre-trained model
         if name == 'Xception':
             model = Xception(include_top=False,weights="imagenet", input_shape=INPUT_SHAPE)(preprocessor)
         else:
             raise Exception("Sorry, model not defined")

         # freeze pre-trained weight
         model.trainable = True

         # rebuild output layer
         x = model
         x = GlobalAveragePooling2D()(x)
         head = Dense(120, activation='softmax')(x)

         model = Model(inputs=input_layer, outputs=head)

         # Compiling the model
         model.compile(optimizer=OPTIMIZER,
                       loss = 'categorical_crossentropy',
                       metrics=['accuracy'])
         return name, model
```

Figure 2.7: Building the model with an additional normalisation pre-processing step

After training, the validation accuracy of the model improved slightly to 68.36%. Hence, the performance of the model improves with even larger image size. Figure 2.8 shows the loss and the accuracy plot of the trained model.

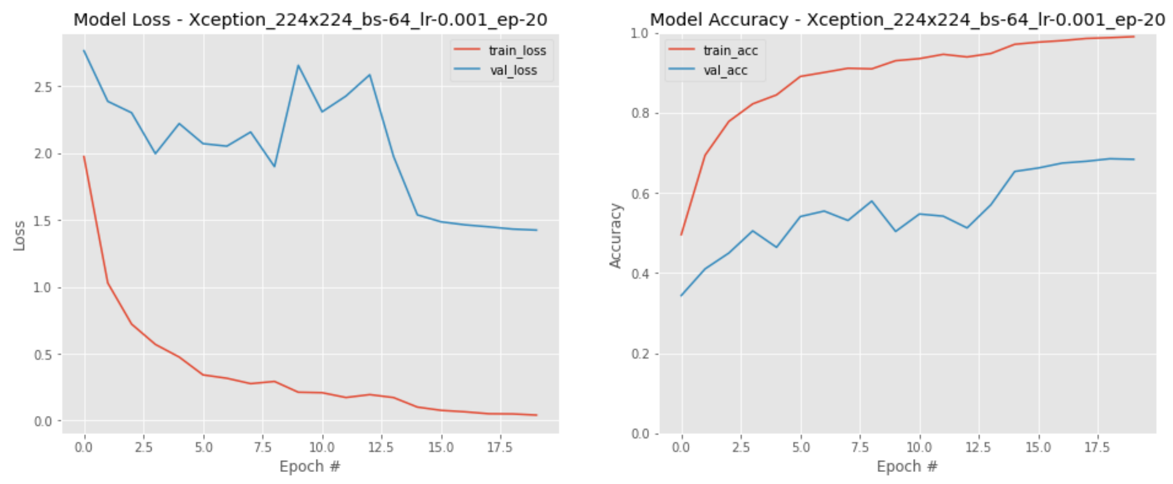


Figure 2.8: Accuracy and loss plot of the improved model

However, the model's performance is still far away from our aim as the validation accuracy is still not up to our expectation.

<u>Motivation for the need to do feature extraction and ensemble learning</u>

Having tried out three attempts to enhance the performance of our model, we realised that the performance of our model did improve but not as much as what we had expected. Hence, we attempt to adopt another method, which is feature extraction and the use of ensemble learning to combine the pre-trained models, hoping to have a better performance.

Phase 3: Training of models using feature extraction and ensemble learning

The use of feature extraction helps to extract features from the dataset with the pretrained models. The collected features are then combined as feature maps and used as input for a multi-classes classifier. These feature maps are easier to process and help to reduce redundant data from the dataset.

Ensemble learning is a technique that creates multiple models and then combines them to produce improved results. In phase 2, we can see that most of our trained models' validation accuracies stagnate around 60-70% if you were to train the models individually. These machine learning models are considered as 'weak learners' as they failed to converge to the desired level.

Our Aim

We aimed to use ensemble learning to combine our 'weak learners', as seen in phase 2, to form an ensemble. This is an efficient approach because different models work differently on our dog breed dataset and hence, some models may perform better on some data and not so good on the others. After the combination of the models, the individual models will cancel out each other's weaknesses when the data is fed in, thus achieving a better model performance.

We also can see that from phase 2, the validation accuracy of the model increases with larger image size. With the help of feature extraction and on top of ensemble learning, we aimed to increase the size of our images to be larger than the image dimensions that the pre-trained models require for much better model performance.

Building the ensembles based on the individual models we have experimented in previous parts

Based on phase 2, we noted down the 'weak learners' that have better performances as compared to the other individual models. We also checked on all the six models and its default input image size to decide how much of an increase in the input image size we should take. Table 3.1 below shows the default input image size of the six best models for our classification task.

Pre-trained Model	Image Size
Xception	299x299
DenseNet169	224x224
InceptionResNetV2	299x299
MobileNet	224x224

InceptionV3	299x299
ResNet50	224x224

Table 3.1: Default size of each pre-trained model

From the table above, since the largest default input image size among the six models is 299x299, we decided to use that as our input image size for this phase.

Model Structure

Figure 3.1 shows our proposed model structure used in this phase.

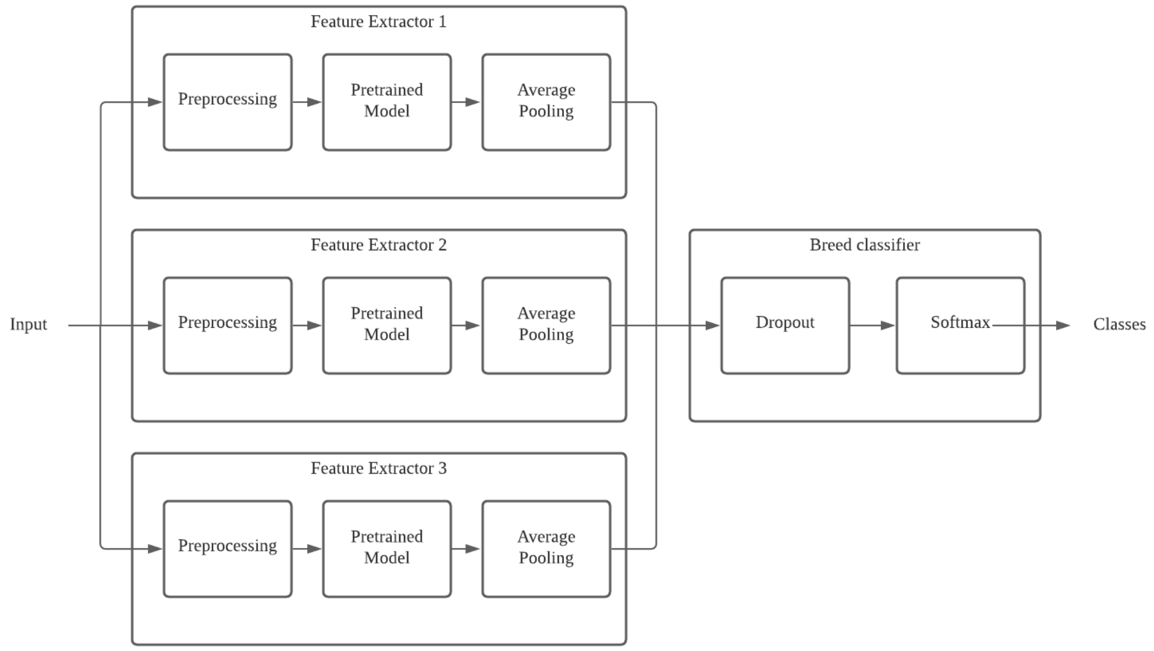


Figure 3.1: Proposed model for this phase

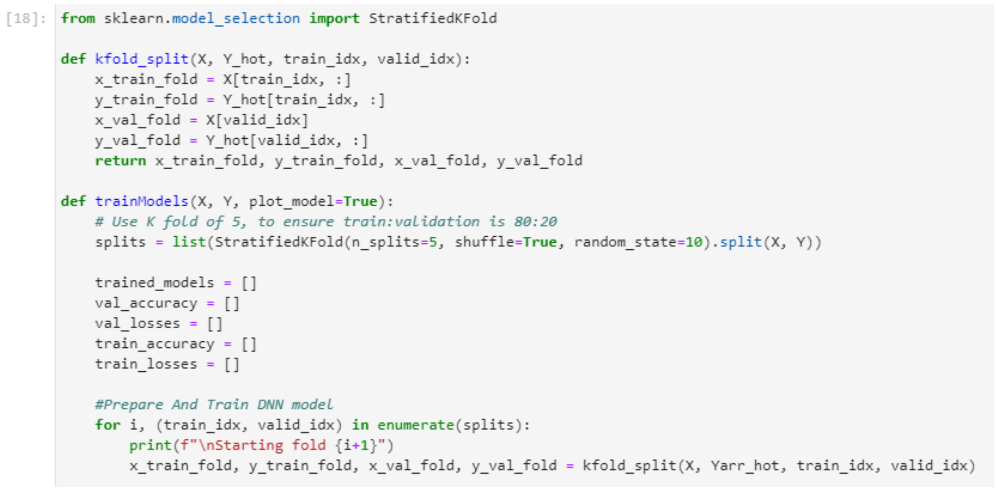
From Figure 3.1, we see from the proposed model that feature extraction is not part of our classifier anymore as compared to the model introduced in Phase 1 and 2. The benefits of this is that it allows us to increase our image size and allows us to save the extracted features so as to allow faster training of our model as our final model is now smaller and shallower.

The reason why we are able to increase our image size to 299x299 here is because the prediction step only requires forward propagation of the model to extract features. Whereas in the third attempt to improve the model in phase 2, where the image size is capped at 224x224, both the forward propagation and backpropagation of the model is needed to train the model.

The need to reserve the memory spaces for backpropagation is the cause that prevents us from training using larger image sizes in phase 2. Hence, this motivates us to separate the feature extraction from the classifier model. This will not be an issue when only forward propagation is needed in this phase.

K-Fold Cross Validation

We then use k-fold cross validation, where k=5, in our training to ensure that the evaluation is fair when we split our data into 5 sets and each set takes turn to be the validation set for each iteration. Figure 3.2 shows a code snippet of how k-fold cross validation is being implemented in training our model.



```
[18]: from sklearn.model_selection import StratifiedKFold

      def kfold_split(X, Y_hot, train_idx, valid_idx):
          x_train_fold = X[train_idx, :]
          y_train_fold = Y_hot[train_idx, :]
          x_val_fold = X[valid_idx]
          y_val_fold = Y_hot[valid_idx, :]
          return x_train_fold, y_train_fold, x_val_fold, y_val_fold

      def trainModels(X, Y, plot_model=True):
          # Use K fold of 5, to ensure train:validation is 80:20
          splits = list(StratifiedKFold(n_splits=5, shuffle=True, random_state=10).split(X, Y))

          trained_models = []
          val_accuracy = []
          val_losses = []
          train_accuracy = []
          train_losses = []

          #Prepare And Train DNN model
          for i, (train_idx, valid_idx) in enumerate(splits):
              print(f"\nStarting fold {i+1}")
              x_train_fold, y_train_fold, x_val_fold, y_val_fold = kfold_split(X, Yarr_hot, train_idx, valid_idx)
```

Figure 3.2: Implementation of k-fold cross validation

Choosing the Combination for the Ensemble Model

We then form five variations of the ensembles for comparison in their performances, with the best two to the best six models in phase 2 based on validation accuracy. We adopted this approach because we want to roughly know the minimum number of models we should include in our ensemble model such that the performance is good based on the validation accuracy. All the 6 models considered have accuracy higher than random guessing (1/120) and hence, the models can be used for ensemble training.

Here are the combinations we considered:

1. Best 2: Xception and DenseNet169
2. Best 3: Xception, DenseNet169, InceptionResNetV2
3. Best 4: Xception, DenseNet169, InceptionResNetV2, MobileNet
4. Best 5: Xception, DenseNet169, InceptionResNetV2, MobileNet, InceptionV3
5. Best 6: Xception, DenseNet169, InceptionResNetV2, MobileNet, InceptionV3, ResNet50

models	Train Accuracy		Train Loss		Validation Accuracy		Validation Loss	
	avg	max	avg	min	avg	max	avg	min
Best 2	0.9413	0.9565	0.2095	0.1568	0.8796	0.8850	0.4057	0.3741
Best 3	0.9535	0.9615	0.1401	0.1114	0.9052	**0.9120**	0.3251	0.3003
Best 4	0.9623	0.9697	0.1131	0.0884	0.9040	**0.9100**	0.3379	0.3106
Best 5	0.9691	0.9784	0.0891	0.0626	0.9068	**0.9105**	0.3439	0.3158
Best 6	0.9795	0.9905	0.0601	0.0256	0.9050	**0.9105**	0.3938	0.3722

Table 3.2: Performances of the ensemble models

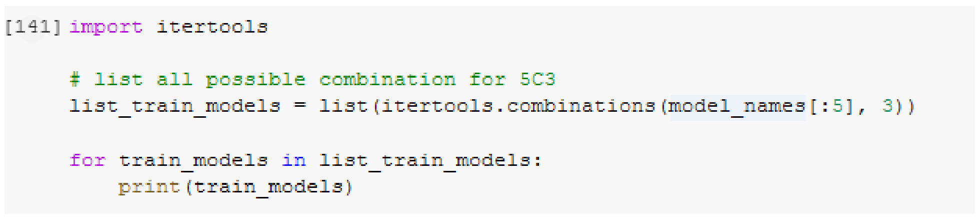
Table 3.2 shows the ensemble models performances based on the above combinations we have considered. From the results, we can see that the max validation accuracy of the ensemble models stagnates around 90% when three or more models are combined together. By Occam’s razor, we favor a simpler version of the ensemble model as compared to a more complex one. As a result, we will adopt a combination of three models in our ensemble model.

<u>Experiment with different combinations of ensemble models</u>

To look at which combination of three models makes the best ensemble model, we chose the best five models that were trained in phase 2 (Table 2.2) based on validation accuracy, and made all possible combinations of three models. In total, we will have 10 (5 choose 3) ensemble models to analyse on their performances. The five best performing models are:

1. Xception
2. DenseNet169
3. InceptionResNetV2
4. MobileNet
5. InceptionV3

Figure 3.3 shows a code snippet to obtain all the possible combinations.



```
[141] import itertools

      # list all possible combination for 5C3
      list_train_models = list(itertools.combinations(model_names[:5], 3))

      for train_models in list_train_models:
          print(train_models)
```

Figure 3.3: All possible combinations for the ensemble model

The combinations are as follow:

C1. 'Xception', 'DenseNet169', 'InceptionResNetV2'
C2. 'Xception', 'DenseNet169', 'MobileNet'
C3. 'Xception', 'DenseNet169', 'InceptionV3'
C4. 'Xception', 'InceptionResNetV2', 'MobileNet'
C5. 'Xception', 'InceptionResNetV2', 'InceptionV3'
C6. 'Xception', 'MobileNet', 'InceptionV3'
C7. 'DenseNet169', 'InceptionResNetV2', 'MobileNet'
C8. 'DenseNet169', 'InceptionResNetV2', 'InceptionV3'
C9. 'DenseNet169', 'MobileNet', 'InceptionV3'
C10. 'InceptionResNetV2', 'MobileNet', 'InceptionV3'

Result - 3-model ensemble combination

Models	Train Accuracy		Train Loss		Validation Accuracy		Validation Loss	
	avg	max	avg	min	avg	max	avg	min
C1	0.94563	0.95524	0.16915	0.14057	0.90256	0.90758	0.32262	0.29229
C2	0.95855	0.96710	0.13738	0.09977	0.87859	0.88460	0.40307	0.38960
C3	0.96065	0.97371	0.12707	0.07739	0.89591	0.90166	0.34541	0.32629
C4	0.95378	0.96894	0.14101	0.09433	0.90080	0.90807	0.32707	0.30371
C5	0.95823	0.96710	0.12296	0.09310	0.91078	**0.91389**	0.30685	0.29105
C6	0.95111	0.96466	0.15646	0.10268	0.89190	0.89487	0.35257	0.33268
C7	0.95246	0.95891	0.13933	0.12085	0.88828	0.89584	0.38324	0.34981
C8	0.94786	0.95500	0.15610	0.13558	0.90295	0.91002	0.31699	0.29355
C9	0.95292	0.97212	0.14855	0.08709	0.87634	0.88264	0.41603	0.39132
C10	0.95630	0.96515	0.12926	0.10483	0.90207	0.90753	0.34584	0.32888

Table 3.3: Performances of the 3-model ensemble combinations

From Table 3.3, we can see that the best 3-model combination is C5, which is Xception, InceptionResNetV2 and InceptionV3, with the highest validation accuracy of 91.389%. We will now move on to hyperparameter tuning of the best ensemble model to further enhance the performance of our model based on its validation accuracy.

Hyperparameter Tuning of the best ensemble model

From the best basic model, we tune the:

1. Batch size (32, 64, 128, 256)
2. Dropout (0.7, 0.8, 0.9)
3. Learning rate (1e-03, 1e-04, 1e-05)

In total, we will have 36 combinations of the best ensemble model via hyperparameter tuning.

Evaluation of the best few models for prediction

The combinations are shown below in Table 3.4:

Combination	Batch Size	Dropout rate	Learning Rate
Default	128	0.9	1e-03
H1	32	0.7	1e-03
H2			1e-04
H3			1e-05
H4		0.8	1e-03
H5			1e-04
H6			1e-05
H7		0.9	1e-03
H8			1e-04
H9			1e-05
H10	64	0.7	1e-03
H11			1e-04
H12			1e-05
H13		0.8	1e-03
H14			1e-04
H15			1e-05
H16		0.9	1e-03
H17			1e-04
H18			1e-05
H19	128	0.7	1e-03
H20			1e-04
H21			1e-05
H22		0.8	1e-03

H23			1e-04
H24			1e-05
H25		0.9	1e-04
H26			1e-05
H27	256	0.7	1e-03
H28			1e-04
H29			1e-05
H30		0.8	1e-03
H31			1e-04
H32			1e-05
H33		0.9	1e-03
H34			1e-04
H35			1e-05

Table 3.4: Hyperparameter tuning model combinations

Result - Hyperparameter Tuning of the best ensemble model

Table 3.5 below shows the performance of the 36 tuned models (including the default one).

Models	**Train Accuracy**		**Train Loss**		**Validation Accuracy**		**Validation Loss**	
	avg	**max**	**avg**	**min**	**avg**	**max**	**avg**	**min**
Default	0.95564	0.97285	0.13327	0.07549	0.90921	0.91638	0.31606	0.26947
H1	0.94732	0.96919	0.16546	0.09217	0.89933	0.90856	0.34472	0.32209
H2	0.95490	0.96332	0.17534	0.13723	0.91068	0.91540	0.30605	0.27016
H3	0.93365	0.94558	0.34942	0.21565	0.90364	0.91100	0.43628	0.32134
H4	0.95718	0.96453	0.12585	0.10203	0.90276	0.90705	0.34359	0.31921
H5	0.94959	0.96393	0.19229	0.13025	0.90921	0.91443	0.31557	0.28398
H6	0.92511	0.93605	0.45355	0.30957	0.90021	0.90465	0.53279	0.41774
H7	0.95084	0.95928	0.15283	0.12648	0.90550	0.91345	0.35405	0.31415
H8	0.94622	0.94987	0.18020	0.16252	0.91166	**0.91687**	0.29037	0.26618
H9	0.92369	0.92785	0.41204	0.32801	0.90051	0.90660	0.48297	0.40075
H10	0.97542	0.99169	0.08064	0.03366	0.90305	0.90753	0.32890	0.30563

H11	0.94999	0.96515	0.21591	0.13017	0.91029	0.91540	0.33491	0.27635
H12	0.93086	0.94093	0.42773	0.27352	0.90393	0.91198	0.51055	0.36835
H13	0.96728	0.99095	0.09964	0.02756	0.90501	0.91096	0.33522	0.28686
H14	0.94661	0.95512	0.20695	0.15831	0.90882	0.91247	0.31947	0.28505
H15	0.92929	0.93298	0.36939	0.29692	0.90364	0.91100	0.45367	0.38317
H16	0.94742	0.96001	0.15857	0.11280	0.90716	0.91100	0.31242	0.28055
H17	0.94103	0.94913	0.20909	0.17173	0.90912	0.91438	0.30585	0.28802
H18	0.91616	0.91991	0.54917	0.50996	0.89826	0.90318	0.61910	0.56914
H19	0.96911	0.98838	0.10804	0.05321	0.90696	0.91149	0.30798	0.28007
H20	0.94764	0.95231	0.21453	0.18742	0.91107	0.91589	0.32380	0.30200
H21	0.92198	0.92858	0.59227	0.45248	0.89748	0.90856	0.66762	0.53572
H22	0.96300	0.98557	0.12130	0.05011	0.90667	0.91100	0.30637	0.29017
H23	0.94260	0.95170	0.23614	0.18129	0.90961	**0.91687**	0.33625	0.31040
H24	0.92032	0.92419	0.60106	0.49641	0.89836	0.90313	0.67624	0.56812
H25	0.93526	0.94533	0.25730	0.19607	0.90765	**0.91736**	0.34231	0.30109
H26	0.91447	0.91647	0.64984	0.59949	0.89219	0.89829	0.72127	0.66314
H27	0.96938	0.98557	0.11220	0.06178	0.90941	0.91394	0.29501	0.27907
H28	0.93668	0.94730	0.30760	0.21751	0.90785	0.91198	0.39582	0.34414
H29	0.91633	0.91880	0.72447	0.59955	0.89679	0.90122	0.79405	0.65013
H30	0.95918	0.96808	0.13877	0.11126	0.91039	0.91540	0.29644	0.27947
H31	0.93844	0.94534	0.27900	0.21282	0.91088	0.91536	0.36746	0.31120
H32	0.91535	0.91673	0.73288	0.71084	0.89258	0.90215	0.80494	0.77135
H33	0.95446	0.96283	0.14075	0.11628	0.91029	0.91394	0.29103	0.27166
H34	0.93194	0.93580	0.28574	0.25660	0.90667	0.91394	0.36438	0.34051
H35	0.90100	0.90487	1.10381	1.09701	0.87683	0.88258	1.16830	1.16346

Table 3.5: Results of the tuned model combinations

From Table 8, we choose three out of the 36 models based on the three highest validation accuracy as the models set for prediction for the test set. The three models are model H8, H23 and H25:

- H8: Batch Size = 32, Dropout = 0.9, Learning rate = 1e-04
- H23: Batch Size = 128, Dropout = 0.8, Learning rate = 1e-04
- H25: Batch Size = 128, Dropout = 0.9, Learning rate = 1e-04

Phase 4: Evaluation on the test set and the final submission result on kaggle

Metric used for the competition

From the competition website, the metric used was multi class log loss metric. The log loss metric takes into account the probabilities underlying the models, not just the final output of the classification. The bolder the probabilities, the better the model will be. The closer the log loss of a model towards zero, the better the model had performed.

Models we use to do prediction on the test set

As analysed in phase 3, the ensemble model that we will use is the Xception, InceptionResNetV2 and InceptionV3 combined model. We will be sending 3 different variations of the tuned ensemble models for submission.

Evaluation on the ensemble models on the test set

Table 4.1 shows the performances of our ensemble models on the test set, and also the private score ranking.

Ensemble Model: Xception, InceptionResNetV2 and InceptionV3

Model's Hyperparameter	Multi-class Log Loss Score	Private Score Ranking on Kaggle	Top N% out of 1280 teams on Kaggle (%)
Batch Size = 32, Dropout = 0.9, Learning Rate = 1e-04	**0.22694**	**348**	**27.188**
Batch Size = 128, Dropout = 0.8, Learning Rate = 1e-04	0.26547	427	33.359
Batch Size = 128, Dropout = 0.9, Learning Rate = 1e-04	0.24191	385	30.078

Table 4.1: Comparison of the tuned models on the test set

Conclusion from the result

From Table 4.1, we can see that the ensemble model with Batch Size 32, Dropout 0.9 and Learning Rate 1e-04 gives the best performance as compared to the other two tuned models, with the lowest multi-class log loss score of 0.22694, standing at the top 27.188%. Together with a validation accuracy of 91.687%, this model has great performance on our dog breed identification tasks. Screenshots of the submissions can be found in the appendix. We will submit the prediction with the lowest multi-class log loss score of 0.22694, the .csv file is named as ‘submission_bs32_dr09_lr1e4.csv’

Conclusion for this project

Having gone through three phases of analysis, from training our model using a pre-trained model, to different ways to increase image size, tuning hyperparameter of a single model to lastly experimenting on feature extraction and ensemble models, we conclude that the feature extraction and ensemble model method (phase 3) gives us the best performance in dealing with the dog breed identification task. We have also learnt that hyperparameter tuning can further enhance our ensemble model and thus, giving us greater performance in terms of high validation accuracy and also a low multi-class log loss score.

In conclusion, we have built a classifier with great performance on predicting the dog breeds of dogs belonging to one of the 120 dog breeds. We can now finally integrate our classifier into an app for the users. This can assist them in identifying a particular dog breed and also the characteristics of the dog based on its breed when they encounter a particular dog. Therefore, users are more well-informed about the dog they encounter and are more decisive in deciding whether they should pet it or not. As such, this reduces the cases of abandoned dogs in the future since decision making is much clearer now for the soon-to-be dog owners.

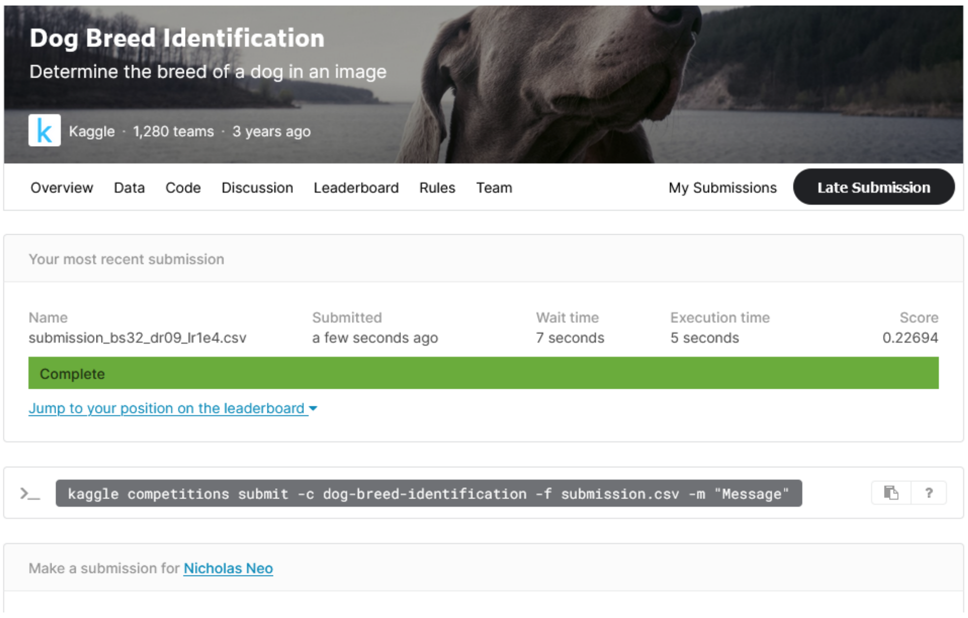
Challenges we faced in this project

In this project, the greatest challenge that we faced is the understanding of the data types (dtype) of our images when it is loaded into Jupyter Notebook or Google Colaboratory. If not configured, the default dtype of the loaded images as numpy arrays are float64, which will take up a huge amount of space on the RAM. We need to know that our data does not require such precision and therefore we should explicitly scale down our dtype to float16 or uint8 to optimise the space that we used. Another challenge that we faced is the RAM limitations of our computer or Google Colaboratory, there were initially many instances where the amount of RAM is insufficient for us to do this project. However, with consistent experimentation on how to optimise the process of loading our image data, a better understanding on the dtype that we used and a more efficient method which is feature extraction and ensemble learning, we manage to optimise our machine learning pipeline for this project and successfully train the models that has the performance we expected it to have.

Appendix

Screenshots of the submissions to kaggle

Batch size 32 + dropout rate 0.9 + learning rate 0.0001


Dog Breed Identification
Determine the breed of a dog in an image
Kaggle · 1,280 teams · 3 years ago
Overview Data Code Discussion Leaderboard Rules Team
My Submissions
Late Submission

Your most recent submission

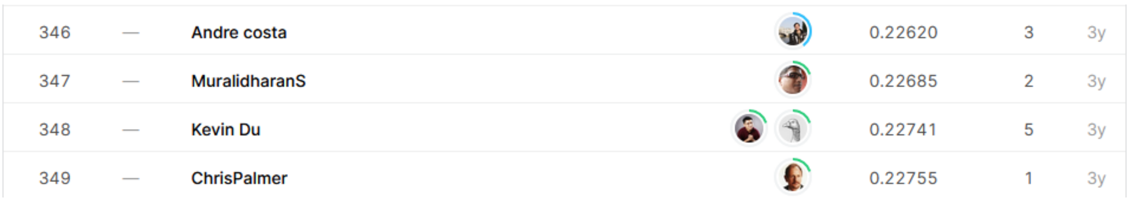
Name	Submitted	Wait time	Execution time	Score
submission_bs32_dr09_lr1e4.csv	a few seconds ago	7 seconds	5 seconds	0.22694

Complete

Jump to your position on the leaderboard

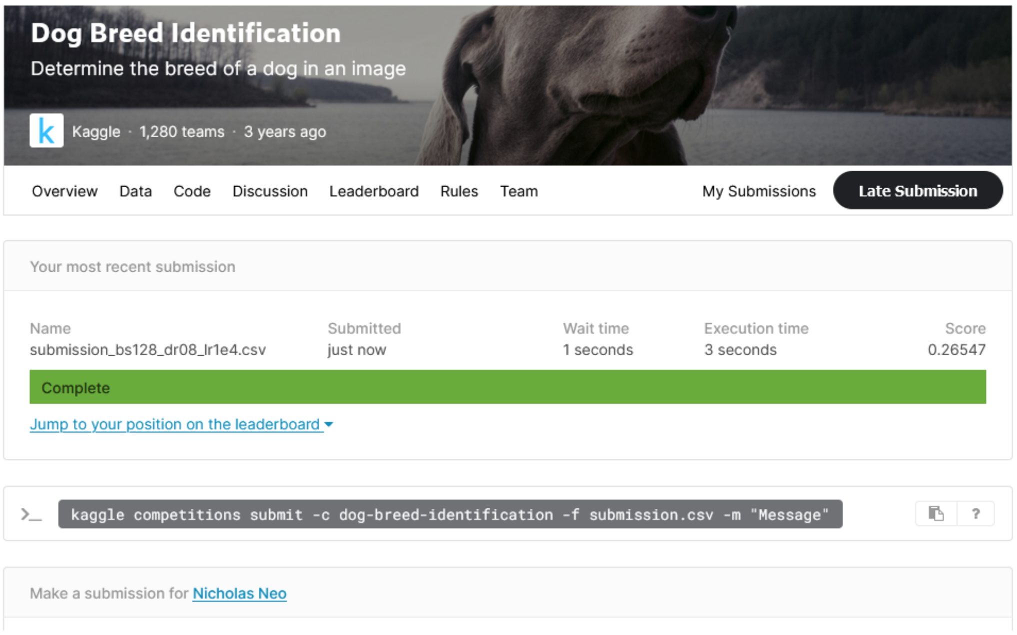
```
kaggle competitions submit -c dog-breed-identification -f submission.csv -m "Message"
```

Make a submission for Nicholas Neo



346	—	Andre costa	0.22620	3	3y
347	—	MuralidharanS	0.22685	2	3y
348	—	Kevin Du	0.22741	5	3y
349	—	ChrisPalmer	0.22755	1	3y

Batch size 128 + dropout rate 0.8 + learning rate 0.0001


Dog Breed Identification
Determine the breed of a dog in an image
Kaggle · 1,280 teams · 3 years ago
Overview Data Code Discussion Leaderboard Rules Team
My Submissions
Late Submission

Your most recent submission

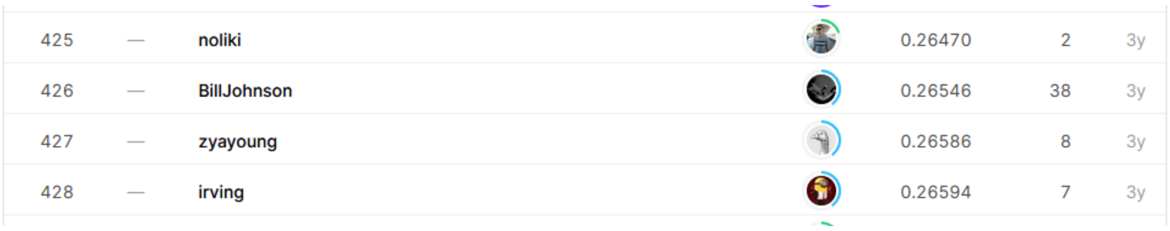
Name	Submitted	Wait time	Execution time	Score
submission_bs128_dr08_lr1e4.csv	just now	1 seconds	3 seconds	0.26547

Complete

Jump to your position on the leaderboard

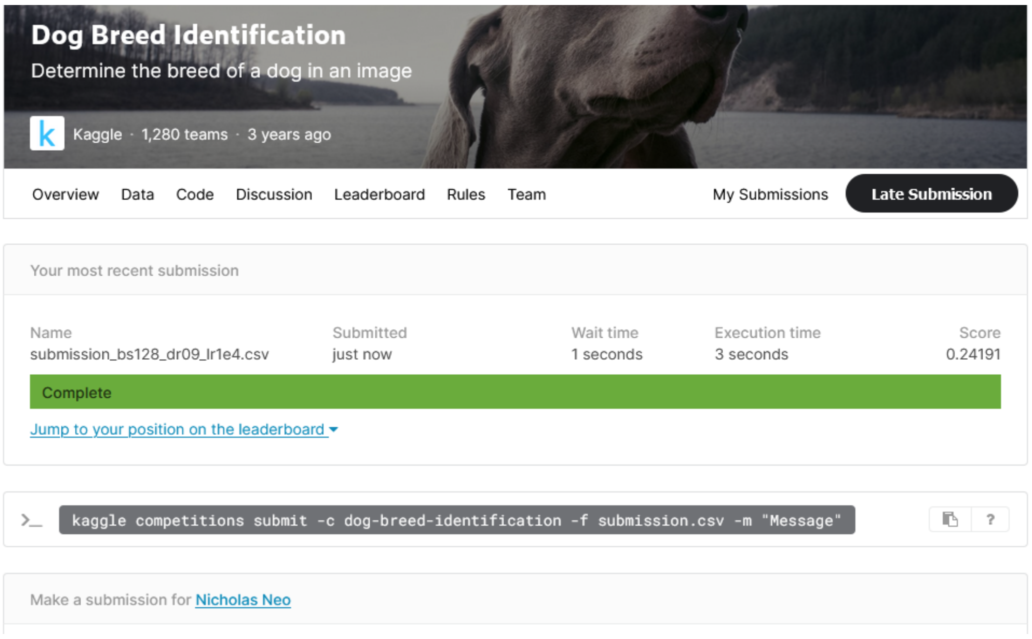
```
kaggle competitions submit -c dog-breed-identification -f submission.csv -m "Message"
```

Make a submission for Nicholas Neo



425	—	noliki	0.26470	2	3y
426	—	BillJohnson	0.26546	38	3y
427	—	zyayoung	0.26586	8	3y
428	—	irving	0.26594	7	3y

<u>Batch size 128 + dropout rate 0.9 + learning rate 0.0001</u>



Dog Breed Identification

Determine the breed of a dog in an image

Kaggle · 1,280 teams · 3 years ago

Your most recent submission

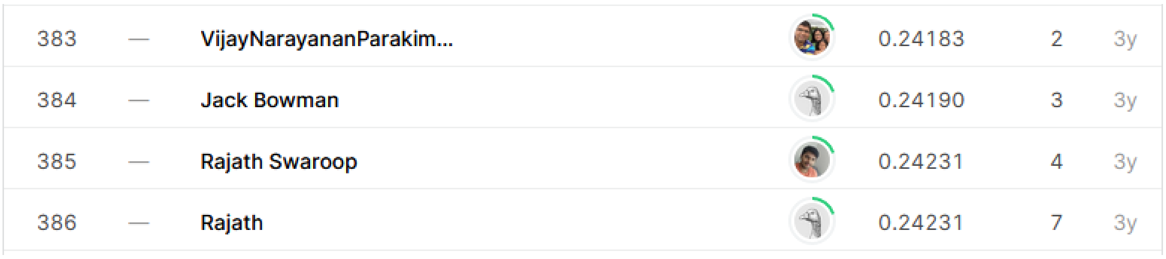
Name	Submitted	Wait time	Execution time	Score
submission_bs128_dr09_lr1e4.csv	just now	1 seconds	3 seconds	0.24191

Complete

Jump to your position on the leaderboard

```
kaggle competitions submit -c dog-breed-identification -f submission.csv -m "Message"
```

Make a submission for Nicholas Neo



383	—	VijayNarayananParakim...	0.24183	2	3y
384	—	Jack Bowman	0.24190	3	3y
385	—	Rajath Swaroop	0.24231	4	3y
386	—	Rajath	0.24231	7	3y

References

https://www.straitstimes.com/singapore/environment/joint-effort-to-manage-stray-dog-population-in-singapore

https://www.kaggle.com/c/dog-breed-identification/overview/evaluation

https://www.mygreatlearning.com/blog/feature-extraction-in-image-processing/

https://www.toptal.com/machine-learning/ensemble-methods-machine-learning

https://bdtechtalks.com/2020/11/12/what-is-ensemble-learning/

https://www.tensorflow.org/api_docs/python/tf/keras/applications

https://www.tensorflow.org/api_docs/python/tf/keras/applications/Xception

https://arxiv.org/pdf/1610.02357.pdf

https://www.tensorflow.org/api_docs/python/tf/keras/applications/DenseNet169

https://arxiv.org/pdf/1608.06993.pdf

https://www.tensorflow.org/api_docs/python/tf/keras/applications/InceptionResNetV2

https://arxiv.org/pdf/1602.07261v2.pdf

https://www.tensorflow.org/api_docs/python/tf/keras/applications/MobileNet

https://arxiv.org/pdf/1704.04861.pdf

https://www.tensorflow.org/api_docs/python/tf/keras/applications/InceptionV3

https://arxiv.org/pdf/1512.00567v3.pdf

https://www.tensorflow.org/api_docs/python/tf/keras/applications/ResNet50

https://arxiv.org/pdf/1512.03385.pdf

https://medium.com/@fzammito/whats-considered-a-good-log-loss-in-machine-learning-a529d400632d