

**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

CZ4041 Machine Learning

Course Project Report

Kaggle Competition (Dog Breed Identification)

S/N	Team Member Name	Matriculation Number	Tutorial Group
1	Prem Adithya Suresh	U1820740B	CS4
2	Shearman Chua Wei Jie	U1820058D	
3	Poh Ying Xuan	U1821489B	
4	Hoo Bing Yuan	U1823670H	
5	Vivian Siow	U1823920J	

Contribution

Name	Work
Prem Adithya Suresh	Modelling, Additional Data Cleaning
Shearman Chua Wei Jie	Modelling, Dataset Creation
Poh Ying Xuan	Modelling, Additional Data Curation
Hoo Bing Yuan	Additional Data Curation, Additional Data Cleaning
Vivian Siow	Exploratory Data Analysis, Additional Data Cleaning

Contribution	1
1. Introduction	3
1.1 Problem Statement	3
2. Exploratory Data Analysis	3
3. Challenges of the Dog Breed Identification Prediction Kaggle Competition	6
4. Proposed Solution	7
4.1 Resources Used for Project	7
4.1.1 Python Libraries	7
4.1.2 Tensor Processing Unit	7
4.2 Data Preprocessing	8
4.2.1 Dataset Split	8
4.2.2 Dataset Creation using tf.data API	8
4.2.3 Efficient Data Pipeline Setup	12
4.2.4 Appending of Additional Data to Training Set	13
4.2.5 Image Augmentation	16
4.3 Project Methodologies	17
4.3.1 Learning Rate Scheduler	17
4.3.2 Dog Breed Identification Using EfficientNetB7 Convolutional Neural Network	18
4.3.2.1 Building of EfficientNetB7 Convolutional Neural Network	18
4.3.2.2 Results of EfficientNetB7 Convolutional Neural Network	19
4.3.3 Dog Breed Identification Using Multi-Modal Convolutional Neural Network	20
4.3.3.1 Choosing of CNN Models for Multi-Modal Convolutional Neural Network	20
4.3.3.2 Building of Multi-Modal Convolutional Neural Network	22
4.3.3.3 Results of Multi-Modal Convolutional Neural Network	24
5. Leaderboard Performance	26
5.1 Leaderboard Results of All Models	26
6. Conclusion	28

1. Introduction

Machine Learning algorithms have been used in a multitude of applications to solve complex tasks; one such application of Machine Learning algorithms is in the field of Computer Vision. Computer Vision is defined as a field of study that seeks to develop techniques to help computers “see” and interpret the content of digital images such as photographs and videos.

For our team’s Course Project for CZ4041 Machine Learning, we will be utilizing Machine Learning to solve a Computer Vision task in the form of a Kaggle Competition problem. The Kaggle Competition we have chosen to tackle is the Dog Breed Identification Prediction Competition.

1.1 Problem Statement

The Dog Breed Identification Prediction Kaggle Competition is a multiclass image classification problem which is a subset of the field of Computer Vision. In the competition, the dataset that we will be working on is the Stanford Dogs Dataset which is a strictly canine subset of the ImageNet dataset. The objective of the competition is to allow teams to practice fine-grained image categorization using various Machine Learning algorithms.

We are provided with a training set and a test set of images of dogs. Each image has a filename, that is its unique id. The dataset comprises 120 breeds of dogs. The goal of the competition is to create a classifier capable of determining a dog's breed from a photo. For the training dataset, the labels, or class of dog breed is provided for each image to allow us to use supervised learning methods to train a classifier to identify the various dogs’ breeds from the images. The test set consists of purely images without labels provided and is to be used by our trained classifier to make predictions on and predict the probability of each breed for each image in the test set.

The predicted probabilities of being in each dog breed class for each image are then saved into a CSV file to be uploaded onto the Dog Breed Identification Prediction Kaggle Competition leaderboard for evaluation. The evaluation score and ranked position of our prediction results will be based on the multi-class loss of the submission predictions, with 0.0 being the best possible evaluation score which means that our classifier is able to predict the dog breed for each image in the test set with a high level of confidence and accuracy.

2. Exploratory Data Analysis

The first step towards tackling the Dog Breed Identification Prediction Kaggle Competition is to first analyze the dataset that we are provided with to allow us to better understand the composition of the dataset, the amount of training data that we have to train the Classifier on, as well as to take note of the characteristics of the dataset such as any imbalance of the classes of the training data.

```
# PATHS TO IMAGES
PATH = '../input/dog-breed-identification/train/'
PATH2 = '../input/dog-breed-identification/test/'
IMGS = os.listdir(PATH); IMGS2 = os.listdir(PATH2)
print('There are %i train images and %i test images'%(len(IMGS),len(IMGS2)))
```

There are 10222 train images and 10357 test images

Fig. 1 Number of Images in Training and Test sets

From the Python notebook code snippet above, we can observe that we have 10222 images available in the training dataset for us to train our Dog Breed Classifier on. This gives us a sufficiently adequate amount of training data to train our Dog Breed Classifier on. We can also observe that there are 10357 images in the test set for us to make predictions on once we have trained the Dog Breed Classifier.

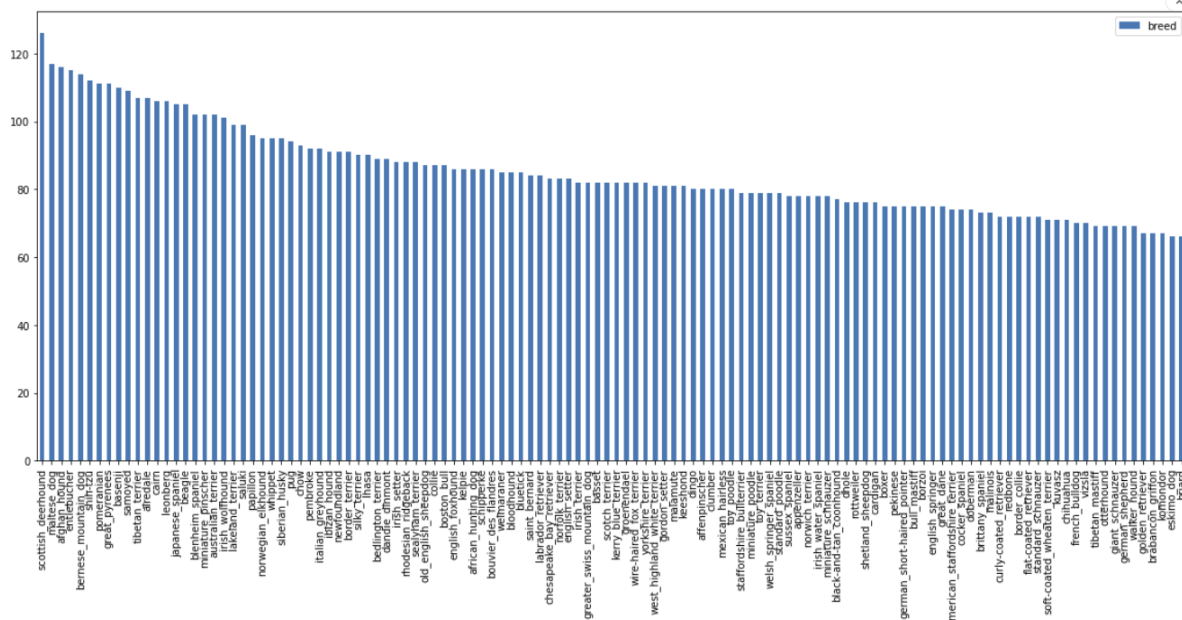


Fig.2 Number of Training Images for Each Dog Breed

We then made use of the “labels.csv” file containing the labels for each of the training images to help us analyze the breakdown of the number of training images available for each Dog Breed class, by plotting the counts of the number of labels in the CSV file for each unique Dog Breed class, for all the 120 Dog Breed classes, as a bar plot. From the bar plot, we can observe that although there are more training images for some of the classes, the training images are fairly well distributed for each of the classes without any heavily imbalanced classes for the training dataset.

```
# Visualize the top 20 breeds with the least data

breeds_all = labels_all["breed"]
breed_counts = breeds_all.value_counts()
plt.figure(figsize=(20,7))
g=sns.countplot(breeds_all, data= breeds_all, order=breeds_all.value_counts(ascending=True)[:20].index)
plt.title("Number of breed classes")
g.set_xticklabels(g.get_xticklabels(),rotation=90)
```

eskimo_dog	66	vizsla	70
briard	66	soft-coated_wheaten_terrier	71
komondor	67	chihuahua	71
golden_retriever	67	kuvasz	71
brabancon_griffon	67	redbone	72
giant_schnauzer	69	flat-coated_retriever	72
walker_hound	69	border_collie	72
otterhound	69	curly-coated_retriever	72
german_shepherd	69	standard_schnauzer	72
tibetan_mastiff	69		
french_bulldog	70		

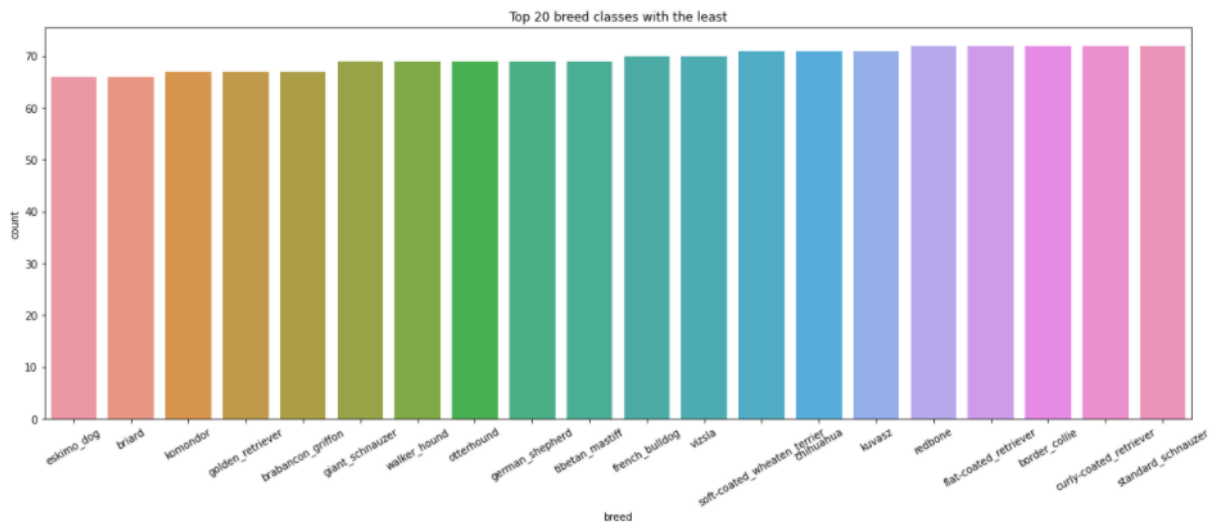


Fig.3 Top 20 breed classes with the least dataset

The countplot above shows the top 20 dog breed classes with the least number of training images. The Eskimo dog and Briard breeds are the Dog Breed classes with the least number of training images, with 66 training images for each class. Although the training images are fairly well distributed for each of the classes without any heavily imbalanced classes for the training dataset, more training images can be appended to the 20 Dog Breed classes with the least amount of data, in an attempt to improve the classification model's performance.

```
# Show image examples

IMG_GRID_ROWS = 5
IMG_GRID_COLUMNS = 4
num_of_images = IMG_GRID_ROWS * IMG_GRID_COLUMNS
fig = plt.figure(1, figsize=(IMG_GRID_COLUMNS * 4, IMG_GRID_ROWS * 4))
grid = ImageGrid(fig, 111, nrows_ncols=(IMG_GRID_ROWS, IMG_GRID_COLUMNS), axes_pad=0.05)

for i in range(num_of_images):
    ax = grid[i]
    ax.imshow(X_all[i, :, :, :])
    class_index = np.argmax(Y_all[i])
    ax.text(10, 200, ('LABEL: %s' % CLASS_NAMES[class_index]), backgroundcolor='w')
    ax.axis('off')
plt.show()
```

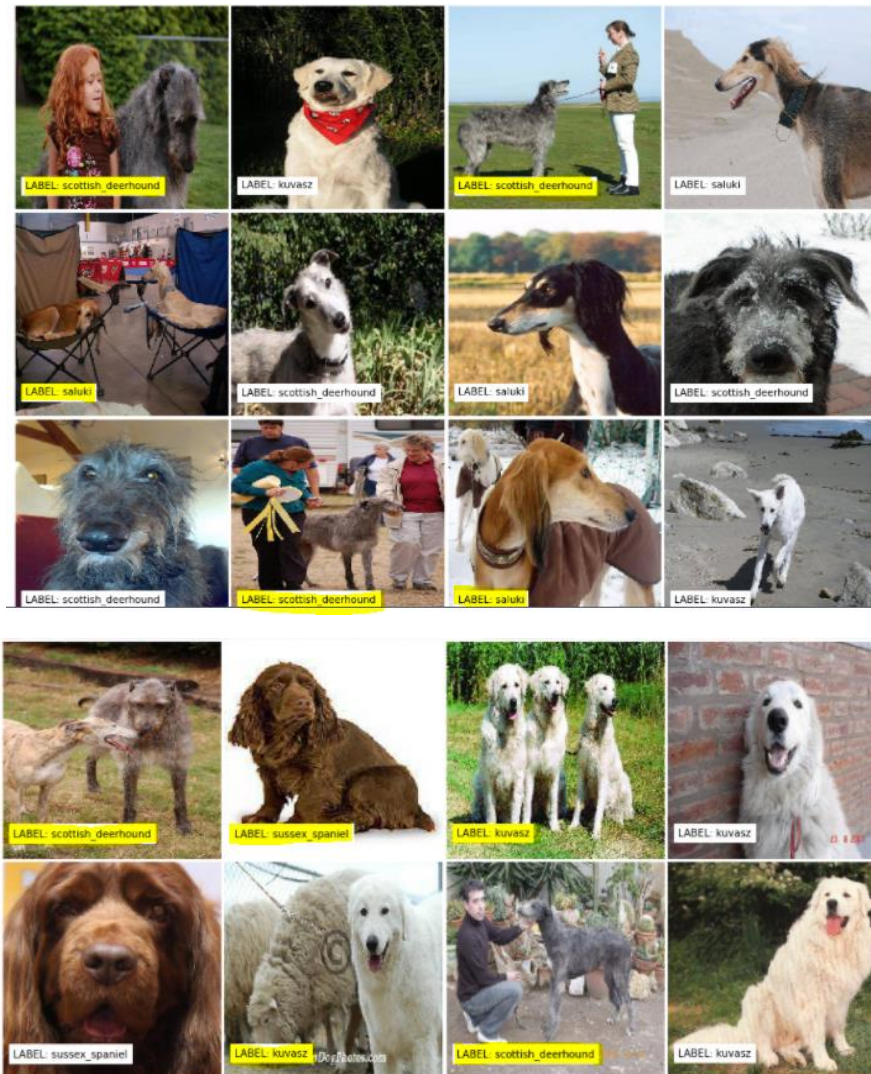


Fig.4 Show image examples of the selected breeds

Running the code snippet as shown above, we display images of a few selected dog breeds, namely, the Scottish Deerhound, Saluki, Sussex Spaniel and Kuvasz in a 5x4 grid. It can be observed that images with highlighted yellow labels contain noise in the data, in the form of the presence of other animals, humans or more than one dog in the image which can interfere with the training. This could affect the prediction accuracy and given that this is only the selected few, there will be more images of the 120 breed classes that require data cleaning.

3. Challenges of the Dog Breed Identification Prediction Kaggle Competition

From the Exploratory Data Analysis conducted in the above section, we can conclude that for this Kaggle Competition, we do not have to face the challenge of a highly imbalanced dataset when training the Classifier. However, there are still some challenges that have to be addressed when tackling this Kaggle Competition task.

Firstly, from the Exploratory Data Analysis conducted, we know that we have 10222 images available for us to train the Classifier. Although having more images will better aid the

Classifier when it tries to learn the multiclass prediction class during training, the large number of image files will take up a large amount of storage space as well as increase the time required to train the Classifier. Therefore, we will require an efficient way of storing the training data, as well as utilize Hardware Accelerators such as GPU and TPU to allow the training time of the Classifier to be sufficiently fast in order for us to be able to conduct more experiments and explore the use of different methods to create a Classifier that is able to give us the best possible evaluation score for the Kaggle Competition.

Secondly, as the Dog Breed Identification Prediction Kaggle Competition task is a significantly complex Machine Learning task, we will need to conduct research on the various State-of-the-Art (SOTA) machine learning algorithms in order to be able to solve the task with great accuracy as well as precision. Time will be required to understand these machine learning algorithms to be able to utilize them in the best possible way to solve our task at hand.

Thirdly, we also concluded that the dataset from Kaggle which consists of 120 Dog Breed classes lacked quality and there was a substantial amount of images that contained noise in the form of the presence of other animals, humans or more than one dog in the images. This could affect the classification model's performance and the overall accuracy of prediction. Hence, we will require an effective method to filter and remove large numbers of inconsistent training images.

4. Proposed Solution

4.1 Resources Used for Project

For this Project, our team has decided to go with the use of State-of-the-Art (SOTA) Convolutional Neural Networks in order to help us solve the Dog Breed Identification Kaggle Competition task. Therefore, our group will be building and training the Classifier on the Google Colaboratory platform. The Classifier will be coded in Python programming language. The reason behind this is that Python has many libraries that can be used to easily build the neural network architectures that we require for our task. Also, by using the Google Colaboratory platform, we can utilize Tensor Processing Units (TPUs) developed by Google to allow our Classifier to be trained faster.

4.1.1 Python Libraries

For this project, we will be using the TensorFlow and Keras Python Libraries to build our Dog Breed Identification Classifier.

4.1.2 Tensor Processing Unit

For our project, we will be using the Tensor Processing Unit (TPU) developed by Google when training our Classifier. The reason for using the TPU accelerator for training our Classifier is due to the fact that training Convolutional Neural Networks using image data is computationally intensive, and by utilizing the TPU accelerator, we will be able to achieve a significantly faster training time when training our Classifier, as compared to using a Graphics Processing Unit (GPU) accelerator or no hardware accelerator at all.

In order to utilize the TPU accelerator in Google Colaboratory, we will be required to convert our images from the Dog Breed Identification Kaggle Competition Dataset as well as the corresponding images labels, into tf.data format stored in TFRecords, which is a format for storing a sequence of binary records, and also the only data format the TPU accelerator is able to process.

4.2 Data Preprocessing

4.2.1 Dataset Split

When training the Classifier, the evaluation of a model's performance on the training dataset used to train the Classifier on would result in a biased score. Therefore, the model should be evaluated on a held-out sample (validation set) to give an unbiased estimate of model's performance. This is typically called a train-test split approach to algorithm evaluation.

Therefore, from the original training set that we are given for the Dog Breed Identification Kaggle Competition, we will have to further split the training set into a training set to be used to train the Classifier, as well as a validation set to be used during training to give an unbiased estimate of model performance.

4.2.2 Dataset Creation using tf.data API

As mentioned in the above sections, we will be utilizing the TPU hardware accelerator in Google Colaboratory when training the Dog Breed Identification Classifier. This will require us to convert the labels and images we have into TFRecords format, as it is impossible to use the Cloud TPUs unless you can feed them data quickly enough using the tf.data.Dataset API. We will be using the tf.data API for reading and writing data in TensorFlow.

In order to avoid having to download the large Dog Breed Identification dataset from Kaggle, our team created a Python notebook within the Kaggle environment itself to convert the dataset into TFRecords to be stored in our Google Cloud Storage. The notebook is shared publicly and can be accessed via the following link:

<https://www.kaggle.com/shearmanchua/create-tfrecords-for-dog-breed-classification>

As mentioned in the previous section, we will be splitting the training set into a training set to be used to train the Classifier, as well as a validation set to be used during training to give an unbiased estimate of model performance. Therefore, we will be storing the TFRecords of the training set and validation set in 2 different Google Cloud Storage buckets.

Encoding Training Images Labels as Numeric Classes

```
label_encoder = LabelEncoder().fit(df.breed.astype(str))
train_df.breed = label_encoder.transform(train_df.breed.astype(str))
keys = label_encoder.classes_
values = label_encoder.transform(label_encoder.classes_)
dictionary = dict(zip(keys, values))
label_encoder = LabelEncoder().fit(df.breed.astype(str))
test_df.breed = label_encoder.transform(test_df.breed.astype(str))
```

Fig.5 Encoding String Labels of Images as Numeric Classes

Firstly, as the labels of the training set images are given as strings which are the respective dog breeds' names, we will first have to encode them using the **sklearn.preprocessing.LabelEncoder** function to encode the strings into Numeric classes from 0 to 119 for the 120 Dog Breeds.

```
SHARDS = 128
nb_images = len(train_df)
shard_size = math.ceil(1.0 * nb_images / SHARDS)
print("Pattern matches {} images which will be rewritten as {} .tfrec files containing {} images each.".format(nb_images, SHARDS, shard_size))
```

Pattern matches 8177 images which will be rewritten as 128 .tfrec files containing 64 images each.

```
def train_parse_function(filename, label):
    print(label)
    img_raw = tf.io.read_file('../input/dog-breed-identification/train/' + filename + '.jpg')
    return img_raw, label
```

+ Code

+ Markdown

```
files = tf.data.Dataset.from_tensor_slices((train_image_paths, train_labels))
dataset = files.map(train_parse_function)
dataset = dataset.batch(shard_size)
```

Fig.6 Building of tf.data.Dataset

Next, we define the number of TFRecord files we want to save the training set into. From the code snippet above, we can observe that there will be a total of 128 TFRecord files, each containing 64 images for the training set. We then define a function called **train_parse_function()** which takes in the filename of an image, as well as its label as the function's argument, and return the actual image from the Dog Breed Identification dataset using the file path, as well as the label that was parsed in.

In order to start converting the data into tf.data format, we first use the function **tf.data.Dataset.from_tensor_slices()** to instantiate the Tensorflow dataset by passing in the training images filenames and training images encoded labels that were obtained from the "labels.csv" file. Next, the instantiated **tf.data.Dataset** is then mapped to our **train_parse_function()** in order to obtain the actual image data from the filenames parsed into the **tf.data.Dataset**. After which, we then batch the **tf.data.Dataset** into the defined **shard_size** to batch the data into 128 batches to be stored as 128 TFRecord files.

```
def to_tfrecord(tfrec_filewriter, img_bytes, label):
    one_hot_class = [np.eye(120)[label[0]]]

    feature = {
        "image": _bytes_feature([img_bytes]), # one image in the list
        "breed": _int_feature([label[0]]),
        "breed_oh": _float_feature(one_hot_class[0].tolist())
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))
```

+ Code

+ Markdown

```
print("Writing TFRecords")
for shard, (image, label) in enumerate(dataset):
    # batch size used as shard size here
    shard_size = image.numpy().shape[0]
    # good practice to have the number of records in the filename
    filename = tfrecord_train_dir + "{:02d}-{}.tfrec".format(shard, shard_size)

    with tf.io.TFRecordWriter(filename) as out_file:
        for i in range(shard_size):
            example = to_tfrecord(out_file,
                                image.numpy()[i],
                                label.numpy()[i])
            out_file.write(example.SerializeToString())

    print("Wrote file {} containing {} records".format(filename, shard_size))
```

Fig.7 Writing of tf.data.Dataset into TFRecords

Next, in order to write the created **tf.data.Dataset** into TFRecords, we define a function, **to_tfrecord()**, to map each of the feature in the **tf.data.Dataset** into a feature dictionary to convert the **tf.data** into **tf.train.Example** to be written out to the TFRecords.

With the **to_tfrecord()** function defined, we then use a for-loop to loop through all of the batches in the **tf.data.Dataset** to convert the **tf.data.Dataset** into **tf.train.Example** format and written into the TFRecords.

```
SHARDS = 32
nb_images = len(test_df)
shard_size = math.ceil(1.0 * nb_images / SHARDS)
print("Pattern matches {} images which will be rewritten as {} .tfrec files containing {} images each.".format(nb_images, SHARDS, shard_size))
```

```
Pattern matches 2045 images which will be rewritten as 32 .tfrec files containing 64 images each.
```

Fig.8 Number of TFRecords for Validation Set

We then repeat the same procedure for the validation set in order to create the TFRecords for the validation set from the given Dog Breed Identification dataset.

Upload to GS Bucket

```
from google.cloud import storage

# For uploading to GCS buckets:
STORAGE_CLIENT = storage.Client.from_service_account_json('../input/cz4041/My Project 78884-3c1398ad9056.json')
```

```
def create_bucket(dataset_name):
    """Creates a new bucket. https://cloud.google.com/storage/docs/ """
    bucket = STORAGE_CLIENT.create_bucket(dataset_name)
    print('Bucket {} created'.format(bucket.name))
```

```
bucket_name = 'cz4041_train'
try:
    create_bucket(bucket_name)
except:
    pass
```

Fig.9 Functions to Create Google Cloud Storage Bucket

After the TFRecords for the training and validation sets are created, they are stored locally in the Kaggle environment runtime, so, the next task is to actually upload all these TFRecords onto Google Cloud Storage. From the code snippet above, we must first pass in the json file that contains the Google Cloud Storage credentials to the storage Client API in order to gain access to our Google Cloud Storage. Next, we define a function that helps us to create a new storage bucket in our Google Cloud Storage called **create_bucket()**. We then created our first bucket for our training set TFRecords called 'cz4041_train'.

```
def upload_blob(bucket_name, source_file_name, destination_blob_name):
    """Uploads a file to the bucket. https://cloud.google.com/storage/docs/ """
    bucket = STORAGE_CLIENT.get_bucket(bucket_name)
    blob = bucket.blob(destination_blob_name)
    blob.upload_from_filename(source_file_name)
    print('File {} uploaded to {}'.format(
        source_file_name,
        destination_blob_name))
```

+ Code

+ Markdown

```
train_files = os.listdir('./tfrecords/train')
print(train_files)
```

```
for file in train_files:
    local_data = './tfrecords/train/'+file
    file_name = file
    upload_blob(bucket_name, local_data, file_name)

print('\nData inside of the GCS Bucket ',bucket_name,':\n')
list_blobs(bucket_name)
```

Fig.10 Uploading of Training TFRecords onto Google Cloud Storage Bucket

After we have created the Google Cloud Storage Bucket for the training set TFRecords, we must then upload the training set TFRecords onto the Google Cloud Storage Bucket. This is done using a for loop to loop through all of the training set TFRecords stored locally in the Kaggle environment runtime and upload each of them onto the Google Cloud Storage Bucket using the **upload_blob()** function.

```
test_files = os.listdir('./tfrecords/val')
print(test_files)
```

```
bucket_name = 'cz4041_val'
try:
    create_bucket(bucket_name)
except:
    pass
```

```
for file in test_files:
    local_data = './tfrecords/val/'+file
    file_name = file
    upload_blob(bucket_name, local_data, file_name)

print('\nData inside of the GCS Bucket ',bucket_name,':\n')
list_blobs(bucket_name)
```

Fig.11 Uploading of Validation TFRecords onto Google Cloud Storage Bucket

The same process is done for the validation set TFRecords stored locally in the Kaggle environment runtime to upload them to Google Cloud Storage Bucket. A separate Google Cloud Storage Bucket is created for the validation set TFRecords called 'cz4041_val'.

4.2.3 Efficient Data Pipeline Setup

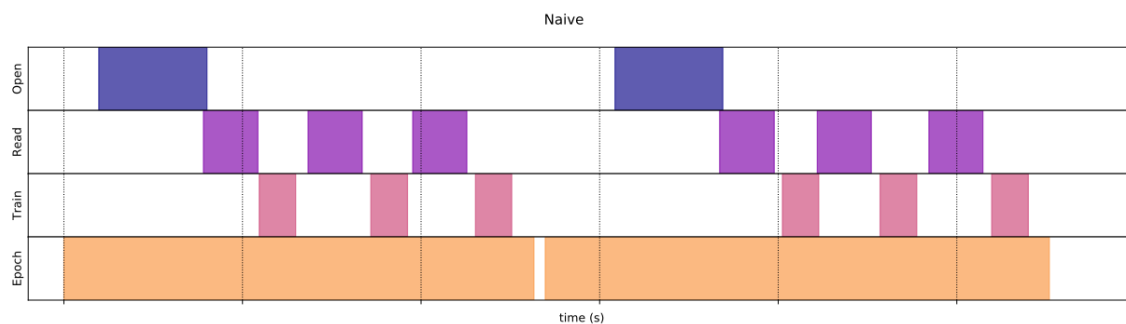


Fig.12 Naive Data Pipeline

To achieve peak performance the input pipeline has to deliver data for the next training step before the current step has finished. The `tf.data` API helps to build flexible and efficient input pipelines. Fig.9 shows the time taken per epoch for a naive data pipeline where training times are longer. Using the `tf.data` API, a more efficient data pipeline was built to optimize performance of the model by implementing prefetching, parallel mapping, and caching.

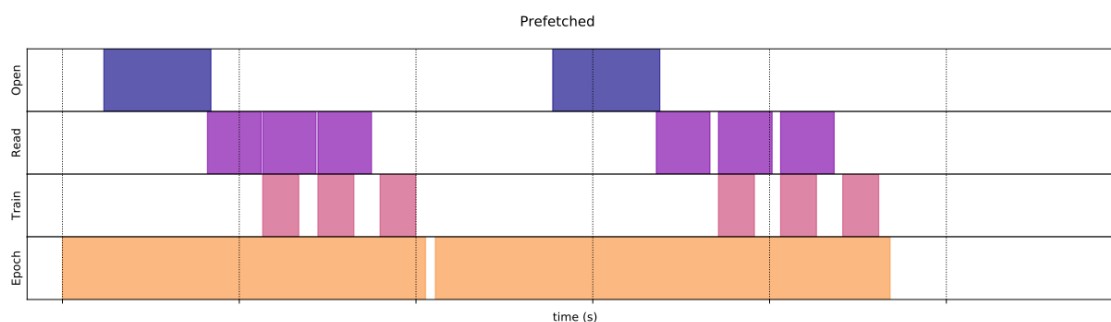


Fig.13 Data Pipeline with Prefetching

Prefetching: With prefetching, while the model is executing the current training step, the input pipeline reads the data required for the next training step. This reduces the waiting time for data to be read from the disk between train steps therefore improving the overall training time per epoch as shown in figure 3-7.

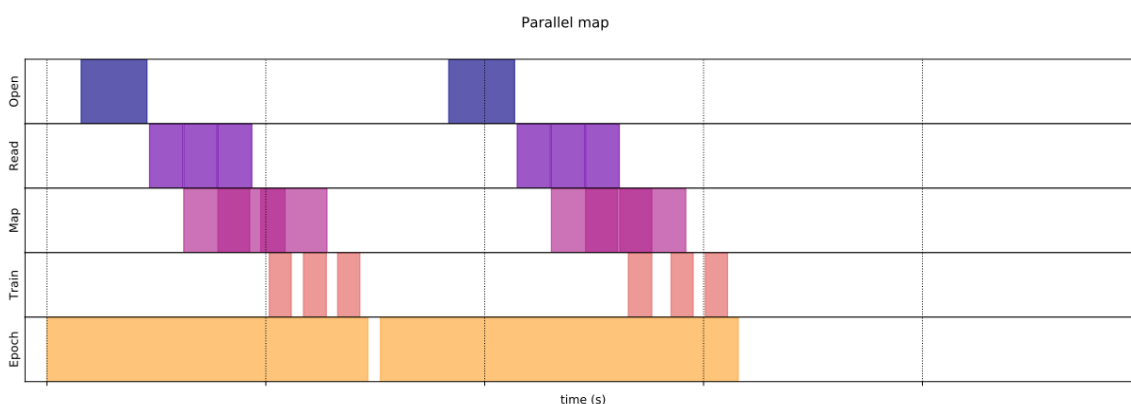


Fig.14 Parallel Data Pipeline

Parallel mapping: Often, data is pre-processed at train time and this may make training times even longer. The tf.data API offers map transformation which allows users to apply functions to each element of the input dataset which can be used to perform various kinds of preprocessing. Instead of sequential mapping, parallel mapping can be applied which results in shorter training times per epoch.

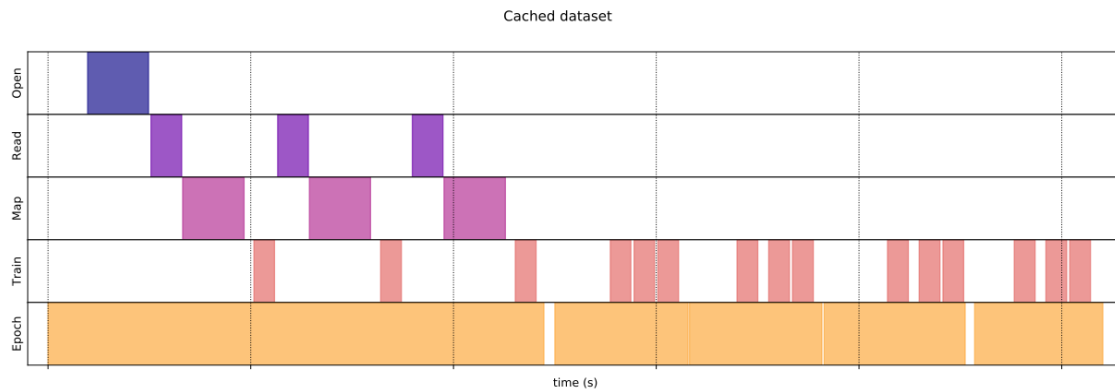


Fig.15 Data Pipeline with Caching

Caching: Pre-processing functions can be expensive and caching reduces the need for mapping to be done more than once. However, if the dataset is large, caching will not improve epoch training times as the entire dataset will not fit into memory and data will have to be read on demand from the disk. Generally, validation datasets are small enough to be cached and hence caching can be enabled for validation datasets only.

4.2.4 Appending of Additional Data to Training Set

Aside from experimenting with different classification models, we also decided to augment our training dataset with more images, in an attempt to improve classification performance. There were multiple approaches of sourcing for data, namely other Dog Breed datasets or APIs provided by image hosting platforms. Other datasets include but are not limited to “Tsinghua Dogs” and “California dog by parts” datasets containing 70248 and 8351 images respectively. Unfortunately, the intersection of similar dog breeds between these datasets and the Kaggle training dataset were limited to relatively popular dog breeds and missing much of the 120 classes required for the augmentation to be effective.

Since data augmentation using other Dog Breed datasets was not viable, we turned towards image hosting platforms. The alternative and arguably superior approach would have been using the Google Image API. However, Google limits API calls to 100 queries a day, with additional queries behind a paywall. By process of elimination, we decided to use the API provided by **Flickr**, to utilize the sheer volume of data the image hosting platform has.

Downloading Flickr Images

In the **search_images** function, a JSON string containing an array of image metadata is retrieved through the Flickr API using the **flickr.photos.search** method. The image metadata is then fed into **download_image** and **save_image** function to save the image to its predefined path name.

```
def search_images(keyword, page):
    params = {'api_key': API_KEY,
              'safe_search': '1',
              'media': 'photos',
              'content_type': '1',
              'privacy_filter': '1',
              'license': '1,2,4,5',
              'page': page,
              'sort': 'relevance',
              'method': 'flickr.photos.search',
              'format': 'json'}
    query_dict = {'text': keyword}
    response = requests.get(REST_ENDPOINT, params=dict(params, **query_dict))
    return json.loads(unjsonify(response.text))
```

```
def download_image(photo, image_filename):
    image_dir = os.path.dirname(image_filename)
    if not os.path.isdir(image_dir):
        os.makedirs(image_dir)
    image_url = IMAGE_URL % (photo['farm'], photo['server'], photo['id'], photo['secret'])
    return save_image(image_url, image_filename)
```

```
def save_image(url, image_filename):
    r = requests.get(url, stream=True)
    with open(image_filename, 'wb') as f:
        for chunk in r.iter_content(chunk_size=1024):
            f.write(chunk)
    return True
return False
```

Fig.16 Functions to download Flickr images

Cleaning Additional Data

Although this approach yielded quantity in data, it lacked quality and it was clear that there was a substantial amount of non-dog images that might instead be detrimental to the model's performance. Thus, we ran the images against a pre-trained resnet50 model from Keras, trained on the ImageNet dataset with 118 classes related to dogs while downloading from **Flickr**. By checking if the maximum image probability is within the 118 classes related to dogs, we were able to predict if the image is a dog as seen in Fig.10.

```
import numpy as np

from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from tqdm import tqdm
from keras.applications.resnet50 import preprocess_input, decode_predictions

ResNet50_model = ResNet50()

def path_to_tensor(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    x = image.img_to_array(img)
    return np.expand_dims(x, axis=0)

def ResNet50_predict_labels(img_path):
    img = preprocess_input(path_to_tensor(img_path))
    return np.argmax(ResNet50_model.predict(img))

def dog_detector(img_path):
    prediction = ResNet50_predict_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

```
import glob
import matplotlib.image as mpimg

keyword = 'chow'
search_and_download_images(keyword)
plt.figure(figsize=(10, 10))
images = glob.glob("/content/" + keyword + '/*.jpg')
count = 0

for i in images:
    img = mpimg.imread(i)
    img = cv2.resize(img, (300, 300))
    count += 1
    ax = plt.subplot(3, 3, count)
    ax.set_title(str(dog_detector(i)) + " for " + os.path.basename(i))
    plt.imshow(img)
    plt.axis("off")
```

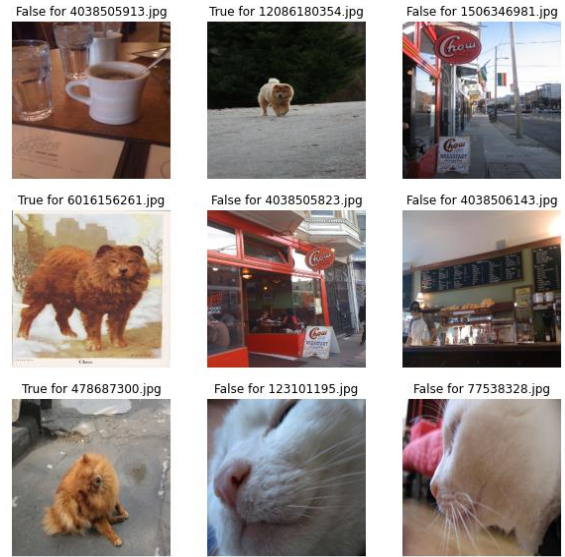


Fig.17 Filtering dog images from additional data

Hence, we were able to obtain an additional dataset containing 21,650 images comprising all the 120 Dog Breed classes. However, the cleaned data still contained some false positives of dog images which were cleaned further with a baseline model that we trained on the original Kaggle dataset. The cleaned data was passed through the baseline model to obtain the one-hot predictions on them. In the kernel density plot shown in fig.11, a large percentage of images were predicted with at least 0.9 probability. Hence, only those images were added to the original dataset resulting in an increase of 14,031 labelled images in the dataset.

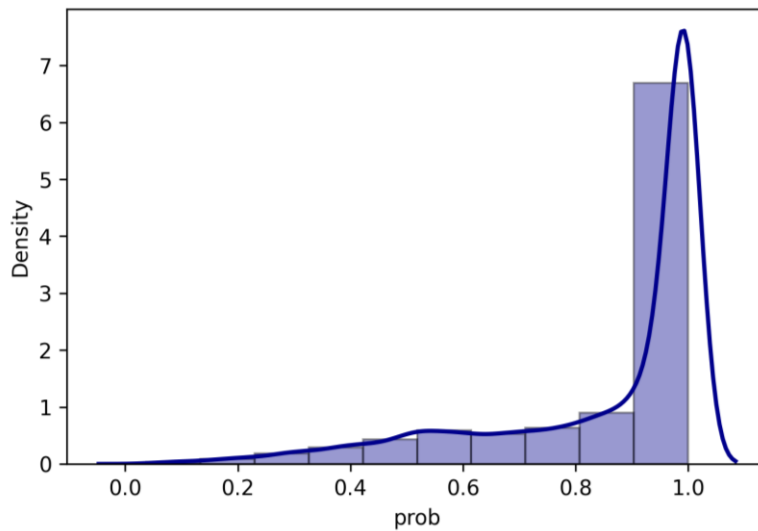


Fig.18 Kernel Density Plot of Probabilities

4.2.5 Image Augmentation



Fig.19 Various Image Transformations

We applied image augmentation to obtain images that were randomly transformed for every epoch as shown in the figure above. Through image augmentation, the size of the dataset is increased virtually as the model is exposed to slightly transformed “new” images every epoch.

Apart from helping the model generalize, image augmentation is also used in the following scenarios:

1. Sparse number of training samples
2. Heavy class imbalance
3. Heavy overfitting on training data
4. Lower test accuracy than validation accuracy

Image augmentation helps the model to generalize and become more robust hence achieving a higher accuracy on the test data. If the training loss of the model decreases faster than the validation loss or if the validation loss does not decrease while training loss decreases, the model can be taken as not generalizing well.

Augmentation was applied using TensorFlow’s image processing functions by carefully choosing the different transformations to be applied including the permissible range of values for those transformations.

We had to be careful to not overdo the augmentation as that could affect the model’s learning negatively. For example, it does not make sense to rotate the image upside down as images passed to the model for inference contain dogs which are right side up. Hence, exposing the model to upside down images of dogs causes the model to learn unnecessary information, albeit making it robust, resulting in a much slower convergence or requiring a lot more data

to achieve a high accuracy. Therefore, it is important to carefully consider the amount and type of augmentation performed based on the problem and data at hand.

4.3 Project Methodologies

In this section, we document the methodologies that our team utilized to tackle the task of Dog Breed Identification. This includes the various types of Convolutional Neural Network architectures that our team experimented with throughout the duration of the project, type of Learning Rate Scheduler utilized, in order to achieve better performance for the classification of dog breeds from previously suggested solutions on Kaggle.

Our team mainly focused on developing solutions that utilizes State-of-the-Art (SOTA) Convolutional Neural Network(CNN) models, such as EfficientNets, InceptionResNet, NASNet etc. The reason behind this is that these SOTA CNN models are pre-trained on the ImageNet dataset, and since the Stanford Dogs Dataset used for the Dog Breed Identification Prediction Kaggle Competition is a subset of the ImageNet dataset, by applying the methodology of transfer learning, we will be able to achieve good classification performance by utilizing these models which have weights pre-trained on the ImageNet dataset.

4.3.1 Learning Rate Scheduler

One common issue encountered when utilizing transfer learning methods for image classification, is the transfer learning model overfitting on the training dataset during training. This can be caused by a multitude of factors, and one of the most common factors that causes the transfer learning model to overfit is the use of a constant learning rate during the training of the model. As the model is being trained every epoch, the weights of the models are updated to improve the accuracy of the model, however, when the model reaches a high enough accuracy, we would ideally want the weights of the model to be updated on a smaller magnitude to prevent the model from overfitting.

Therefore, in order to ensure that as we train the model, the magnitude in which the weights of the models are updated decreases with each epoch of training, we used a Learning Rate Scheduler to reduce the learning rate of the model as it is being trained.

```
LR = 0.0001 # 0.0005
EPOCHS = 40
WARMUP = 10

def get_cosine_schedule_with_warmup(lr, num_warmup_steps, num_training_steps, num_cycles=0.5):
    """
    Modified the get_cosine_schedule_with_warmup from huggingface for tensorflow
    (https://huggingface.co/transformers/_modules/transformers/optimization.html#get_cosine_schedule_with_warmup)

    Create a schedule with a learning rate that decreases following the
    values of the cosine function between 0 and `pi * cycles` after a warmup
    period during which it increases linearly between 0 and 1.
    """

    def lr_fn(epoch):
        if epoch < num_warmup_steps:
            return float(epoch) / float(max(1, num_warmup_steps)) * lr
        progress = float(epoch - num_warmup_steps) / float(max(1, num_training_steps - num_warmup_steps))
        return max(0.0, 0.5 * (1.0 + math.cos(math.pi * float(num_cycles) * 2.0 * progress))) * lr

    return tf.keras.callbacks.LearningRateScheduler(lr_fn, verbose=True)

lr_schedule= get_cosine_schedule_with_warmup(lr=LR, num_warmup_steps=WARMUP, num_training_steps=EPOCHS)
```

Fig.20 Code Snippet of Custom Learning Rate Scheduler

As can be seen from the figure above, our team made use of a custom Learning Rate Scheduler, which first slowly increases the learning rate of the model from 0 to the targeted learning rate($1e-4$ for our project) from the first epoch till the predefined warm-up epoch(predefined as 10 epochs for our project), after which, it slowly decreases the learning rate of the model with each epoch, based on the curve of half a cosine cycle, until the last epoch(epoch 40) of training. By utilizing this custom Learning Rate Scheduler, our team is able to enable the model weights to be updated on a larger magnitude for the first 10 epochs of training, before reducing the magnitude in which the weights of the models are updated for the last 30 epochs of training, at a rate defined by half a cosine curve.

4.3.2 Dog Breed Identification Using EfficientNetB7 Convolutional Neural Network

The EfficientNets consists of a continuous family of models, namely B0 to B7, created by varying scaling factors of resolutions and depth and width. These factors are specifically chosen for each of these models to ensure the best optimal performance and implemented within the pre-trained models.

For our first model, we used EfficientNetB7 as our base model. The EfficientNetB7 showed the best results in the papers and we wanted to replicate these results onto our datasets. However, for this to work, we had to convert our images to 600 by 600 resolutions. As mentioned earlier, each model within the EfficientNets model has fixed factors that are specifically chosen for the best performance.

4.3.2.1 Building of EfficientNetB7 Convolutional Neural Network

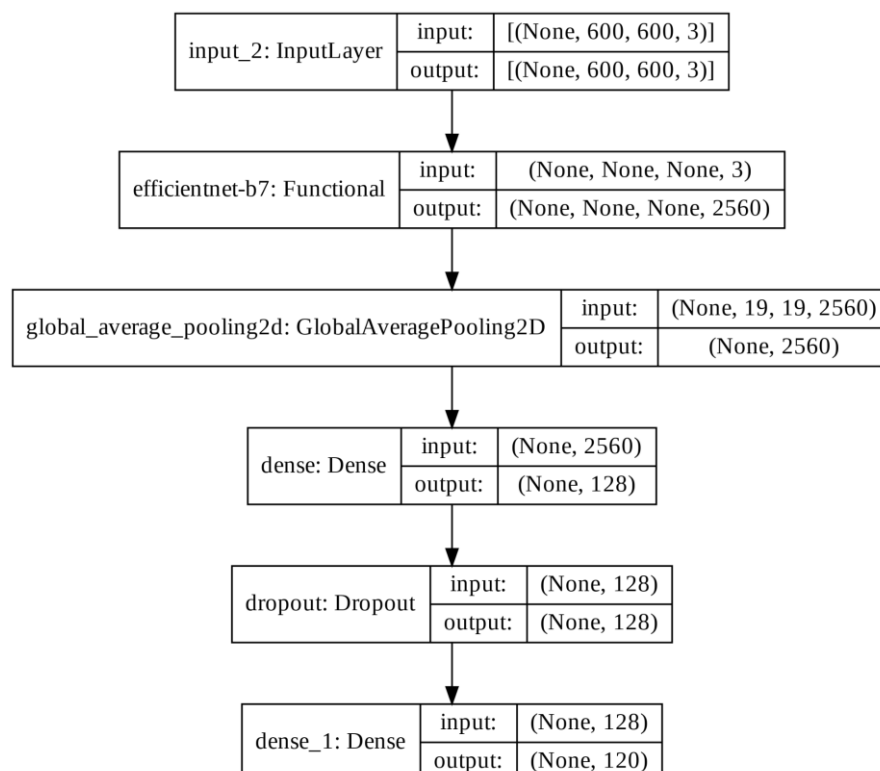


Fig.21 EfficientNetB7 Model Architecture

As shown in the figure above, we first fit the input image layers through the pre-trained EfficientNetB7 and then pooled using GlobalAveragePooling2D() layer before they are classified into 1 of the 120 dog breeds by the output Dense() layer with 120 neurons and “softmax” activation function.

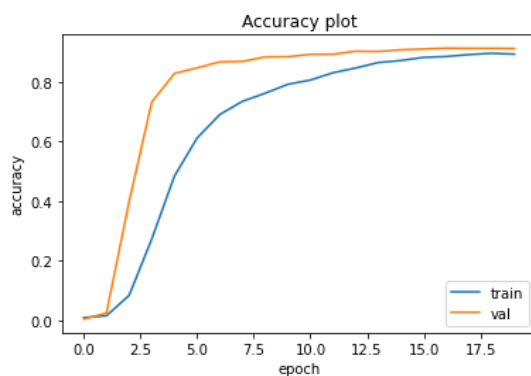
4.3.2.2 Results of EfficientNetB7 Convolutional Neural Network

For the EfficientNetB7 CNN architectures defined above, we trained them with both the original dataset from the Dog Breed Identification Prediction Kaggle Competition, as well as the dataset consisting of data from the Dog Breed Identification Prediction Kaggle Competition appended with additional data using the method detailed in section 4.2.4. After training the models, we recorded the models’ last epoch validation loss and accuracy, and also plotted the Loss vs. Epoch and Accuracy vs. Epoch graph.

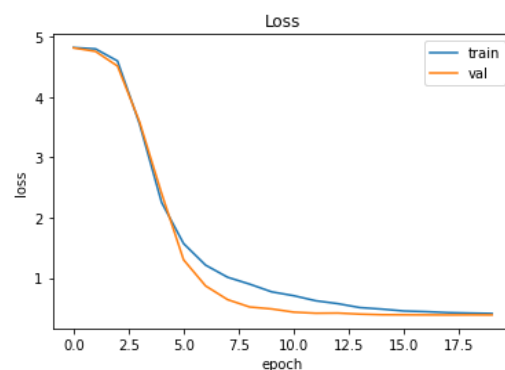
EfficientNetB7 Results:

Training Data	Validation Accuracy	Validation Loss
Original Dataset	0.9123	0.3879
Dataset with Appended Data	0.8410	0.4169

Original Dataset:

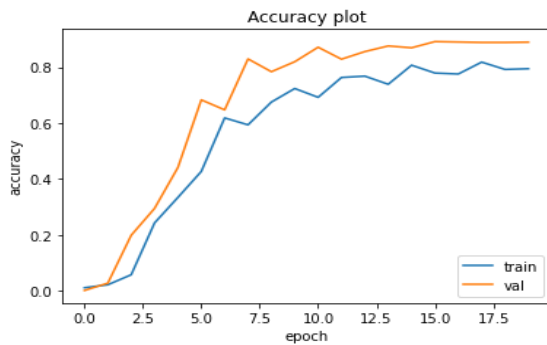


Accuracy vs Epoch

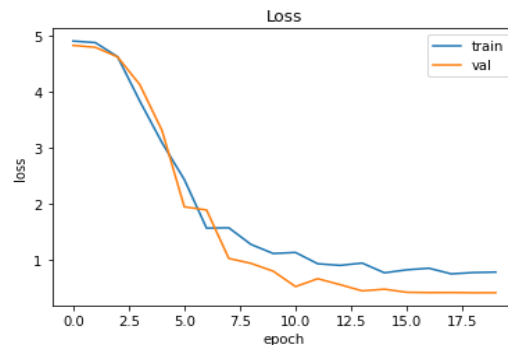


Loss vs Epoch

Dataset with Appended Data:



Accuracy vs Epoch



Loss vs Epoch

From the graph above, we noticed the EfficientNetB7 CNN model could classify the different dog breeds decently, yielding a validation loss of 3.87 and 0.416 for the original datasets and appended dataset, respectively. Unexpectedly, with the additional data, the model did not improve its validation loss and accuracy. One possible reason is that the new images that were introduced also incorporated more noises to the model as well. Hence, the model did not improve but stagnate in the overall learning.

4.3.3 Dog Breed Identification Using Multi-Modal Convolutional Neural Network

After our team developed the first classification model, making use of the EfficientNetB7 model as the base, our team realized that even after tuning the hyperparameters of the model, the model accuracy, as well as model loss will reach a plateau and no longer improve any further. One possible reason could be that a single CNN model may not be sufficient to map the features for the Dog Breed Identification task. Therefore, our team went on to further investigate the use of a multimodal approach for solving the task of Dog Breed Identification.

4.3.3.1 Choosing of CNN Models for Multi-Modal Convolutional Neural Network

In order to create a multi-modal CNN, we will first have to decide on the SOTA CNN models that we want to use to build the multi-modal CNN. Therefore, to help us decide on the SOTA CNN models to use, our team first created a code that will loop through all the SOTA CNN models available in Tensorflow, train each of them for 10 epochs and at the end of the loop, rank the models based on their final epoch validation loss, ranked from the lowest loss to the highest loss.

```

model_dictionary = {m[0]:m[1] for m in inspect.getmembers(tf.keras.applications, inspect.isfunction)}
model_benchmarks = {'model_name': [], 'num_model_params': [], 'val_loss': []}
effnets = ['EfficientNetB3','EfficientNetB7','EfficientNetB6','EfficientNetB2','EfficientNetB4','EfficientNetB0','EfficientNetB1']
castnets = ['VGG16','VGG19','MobileNetV3Small','MobileNetV3Large','ResNet152','ResNet101','ResNet50']

with strategy.scope():
    for model_name, model in tqdm(model_dictionary.items()):
        try:
            print('\n')
            print(model_name)
            if model_name == 'DenseNet121' or model_name == 'DenseNet169' or model_name == 'DenseNet201':
                base_model = model(include_top=False, weights='imagenet', classes = 120)
            else:
                base_model = model(include_top=False, weights='imagenet', classes=df['label'].nunique())
            base_model.trainable = False

            model_input = Input(shape=[IMAGE_SIZE[0],IMAGE_SIZE[1],3])

            if model_name in effnets:
                x = tf.keras.applications.efficientnet.preprocess_input(model_input)
            elif model_name == 'VGG16':
                x = tf.keras.applications.vgg16.preprocess_input(model_input)
            elif model_name == 'VGG19':
                x = tf.keras.applications.vgg19.preprocess_input(model_input)
            elif model_name == 'MobileNetV3Small' or model_name == 'MobileNetV3Large':
                x = tf.keras.applications.mobilenet_v3.preprocess_input(model_input)
            elif model_name == 'ResNet152' or model_name == 'ResNet101' or model_name == 'ResNet50':
                x = tf.keras.applications.resnet.preprocess_input(model_input)
            elif model_name == 'ResNet152V2' or model_name == 'ResNet101V2' or model_name == 'ResNet50V2':
                x = tf.keras.applications.resnet_v2.preprocess_input(model_input)
            elif model_name == 'DenseNet121' or model_name == 'DenseNet169' or model_name == 'DenseNet201':
                x = tf.keras.applications.densenet.preprocess_input(model_input)
            elif model_name == 'NASNetLarge' or model_name == 'NASNetMobile':
                x = tf.keras.applications.nasnet.preprocess_input(model_input)
            elif model_name == 'InceptionResNetV2':
                x = tf.keras.applications.inception_resnet_v2.preprocess_input(model_input)
            elif model_name == 'InceptionV3':
                x = tf.keras.applications.inception_v3.preprocess_input(model_input)
            elif model_name == 'MobileNetV2':
                x = tf.keras.applications.mobilenet_v2.preprocess_input(model_input)
            elif model_name == 'MobileNet':
                x = tf.keras.applications.mobilenet.preprocess_input(model_input)
            elif model_name == 'Xception':
                x = tf.keras.applications.xception.preprocess_input(model_input)

```

```

x = base_model(x)
x = GlobalAveragePooling2D()(x)
x = Dense(df['label'].nunique(), activation='softmax')(x)
model = Model(inputs=model_input, outputs=x)
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer=Adam(1e-3))

model.summary()

steps_per_epoch= (8177) // BATCH_SIZE
validation_steps= (2045) // BATCH_SIZE

history = model.fit(alt_training_dataset,
                    steps_per_epoch=steps_per_epoch,
                    validation_data=alt_val_dataset,
                    validation_steps=validation_steps,
                    epochs=10,
                    class_weight=d_class_weights,
                    verbose=2
                    )

model_benchmarks['model_name'].append(model_name)
model_benchmarks['num_model_params'].append(base_model.count_params())
model_benchmarks['val_loss'].append(history.history['val_loss'][-1])
except Exception as e:
    print(e)
pass

```

Fig.22 Code Snippet for the Loop Used to Run the Training for All the SOTA CNN Models

From the figures above, we see the code snippet for the loop used to run the training for all the SOTA CNN models available in Tensorflow, where for each iteration, we build a new CNN model with a SOTA CNN model as the based, initialized with the preprocessing layer of the SOTA CNN model, the model is then compiled and trained for 10 epochs. The image size passed to each model is 331x331 pixels.

	model_name	num_model_params	val_loss
9	EfficientNetB7	64097687	0.211031
16	NASNetLarge	84916818	0.216893
10	InceptionResNetV2	54336736	0.237417
7	EfficientNetB4	17673823	0.243055
6	EfficientNetB3	10783535	0.250296
25	Xception	20861480	0.253399
11	InceptionV3	21802784	0.260942
8	EfficientNetB6	40960143	0.280778
5	EfficientNetB2	7768569	0.305236
4	EfficientNetB1	6575239	0.332661
2	DenseNet201	18321984	0.383018
20	ResNet152V2	58331648	0.398384
13	MobileNetV2	2257984	0.439196
18	ResNet101V2	42626560	0.443645
1	DenseNet169	12642880	0.449576
19	ResNet152	58370944	0.464144
3	EfficientNetB0	4049571	0.476771
17	ResNet101	42658176	0.488025
0	DenseNet121	7037504	0.526196
22	ResNet50V2	23564800	0.552592
21	ResNet50	23587712	0.601825
12	MobileNet	3228864	0.697078
14	MobileNetV3Large	4226432	0.726320
24	VGG19	20024384	1.199805
15	MobileNetV3Small	1529968	1.203225
23	VGG16	14714688	1.336465

Fig.23 Last Epoch Loss for All SOTA CNN Models

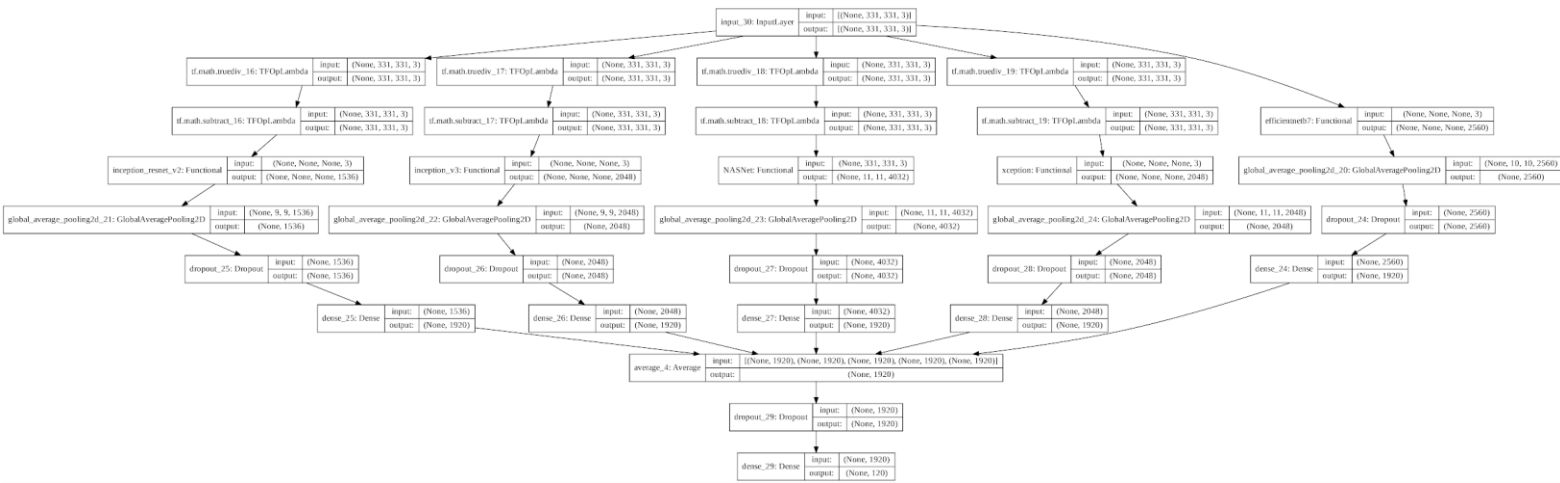
The figure above shows the SOTA CNN models ranked from the lowest final epoch validation loss to the highest final epoch validation loss. From the table, we can observe that amongst all the SOTA CNN models, the EfficientNetB7 model has the lowest last epoch validation loss.

From the results obtained, our team selected the top 5 SOTA CNN models from different families (top 5 models in the figure above consists of 3 EfficientNets from the same model family, only EfficientNetB7 will be used) of SOTA CNN models to be used to build our multi-modal CNN. The 5 models to be used are the EfficientNetB7, NASNetLarge, InceptionResNetV2, Xception, and InceptionV3 models.

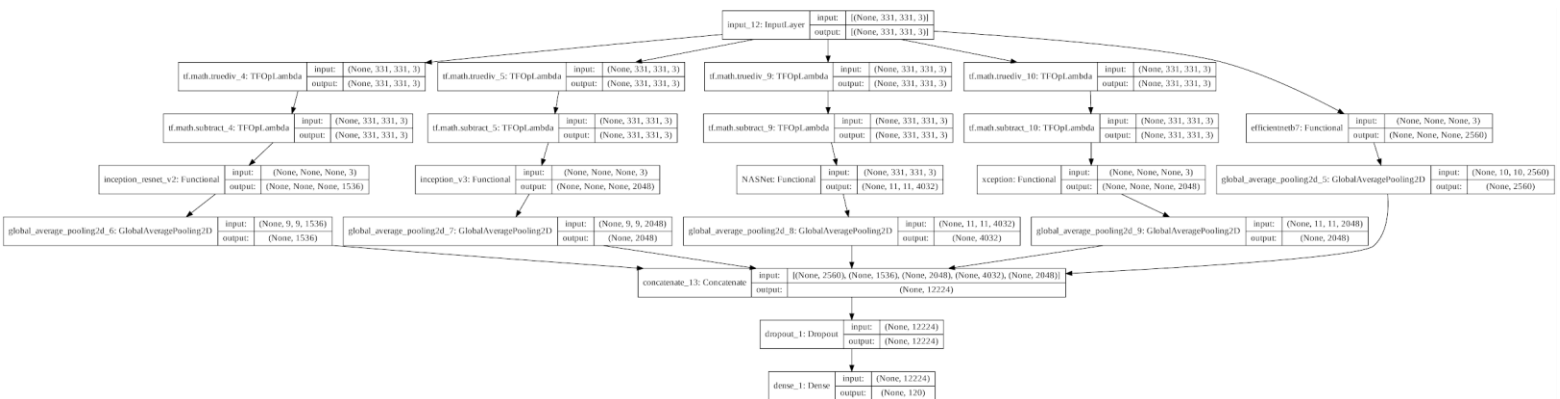
4.3.3.2 Building of Multi-Modal Convolutional Neural Network

In order to build our multi-modal CNN, we first have to create 5 base models using the 5 chosen SOTA CNN models. The features mapped by the 5 individual SOTA CNN base models are then pooled using a GlobalAveragePooling2D() layer before they are concatenated together to give a single feature map of the input, and then classified into 1 of the 120 dog breeds by the output Dense() layer with 120 neurons and “softmax” activation function. For concatenating the pooled feature maps of the 5 individual SOTA CNN base

models, we experimented with using both the Keras Concatenate() layer, as well as Average() layer, to see which gives us better performance for our multi-modal CNN.



In the figure above, we see the multi-modal CNN architecture that utilizes the Average() layer to concatenate the feature maps of the 5 individual SOTA CNN base models. From the figure, we are able to observe that that multi-modal CNN has one input layer which takes in a 331x331 image as the input, which is then passed to the 5 individual SOTA CNN base models' preprocessing layers before being passed to the 5 SOTA CNN base models to have the image features mapped. The mapped features by each individual model is then passed through a GlobalAveragePooling2D() to be pooled, followed by a Dropout() layer and Dense() layer of 1920 neurons with "relu" activation function to ensure that the features mapped by each individual model is of the same shape, before all of the feature maps of the 5 SOTA CNN base models are averaged together using the Average() layer and then passed into the output Dense() layer with 120 neurons and "softmax" activation function.



In the figure above, we see the multi-modal CNN architecture that utilizes the Concatenate() layer to concatenate the feature maps of the 5 individual SOTA CNN base models. From the figure, we are able to observe that that multi-modal CNN has one input layer which takes in a 331x331 image as the input, which is then passed to the 5 individual SOTA CNN base models' preprocessing layers before being passed to the 5 SOTA CNN base models to have the image features mapped. The mapped features by each individual model is then passed through a GlobalAveragePooling2D() to be pooled, then all of the feature maps of the 5 SOTA CNN base models are concatenated together into one feature map using the

Concatenate() layer before being passed into the output Dense() layer with 120 neurons and “softmax” activation function.

4.3.3.3 Results of Multi-Modal Convolutional Neural Network

For both the multi-modal CNN architectures defined above, we trained them with both the original dataset from the Dog Breed Identification Prediction Kaggle Competition, as well as the dataset consisting of data from the Dog Breed Identification Prediction Kaggle Competition appended with additional data using the method detailed in section 4.2.4. After training the models, we recorded the models’ last epoch validation loss and accuracy, and also plotted the Loss vs. Epoch and Accuracy vs. Epoch graphs for each model.

Multi-Modal CNN with Average() Layer Results:

Training Data	Validation Accuracy	Validation Loss
Original Dataset	0.9542	0.1561
Dataset with Appended Data	0.9511	0.1760

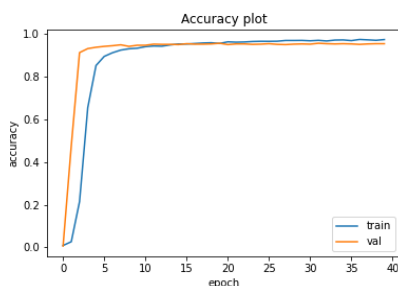
Multi-Modal CNN with Concatenate() Layer Results:

Training Data	Validation Accuracy	Validation Loss
Original Dataset	0.9510	0.1525
Dataset with Appended Data	0.9526	0.1622

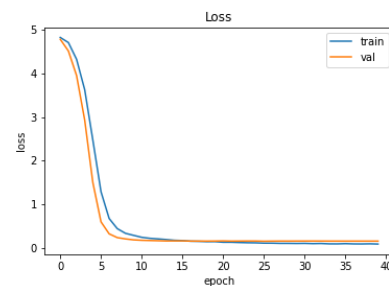
From the results above, we can observe that in terms of the model’s multiclass loss (which is the metric used for the Dog Breed Identification Prediction Kaggle Competition Leaderboard), the models that were trained using the original dataset from the Dog Breed Identification Prediction Kaggle Competition performed better than the models training using the dataset consisting of data from the Dog Breed Identification Prediction Kaggle Competition appended with additional data. Also, we are able to observe that the multi-modal CNN architecture that has the Concatenate() layer performs better than the multi-modal CNN architecture that has the Average() layer.

Multi-Modal CNN with Average() Layer Graphs:

Original Dataset:

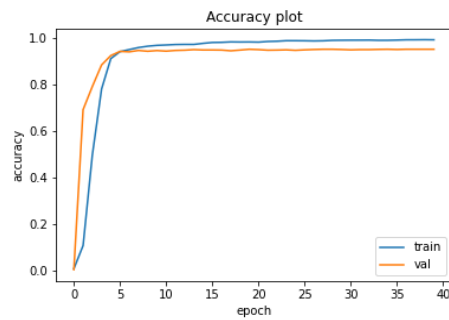


Accuracy vs Epoch

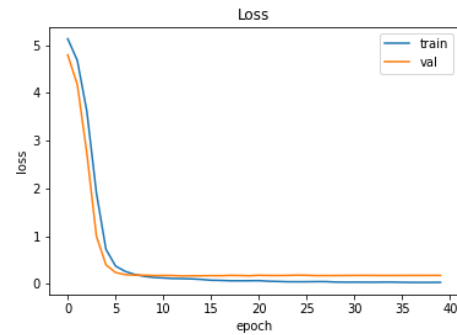


Loss vs Epoch

Dataset with Appended Data:



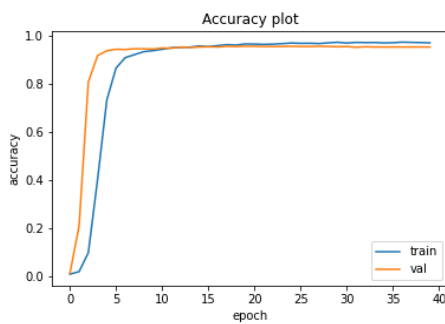
Accuracy vs Epoch



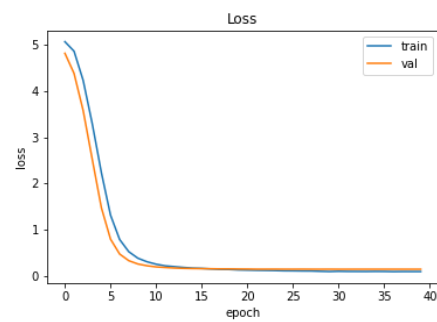
Loss vs Epoch

Multi-Modal CNN with Concatenate() Layer Graphs:

Original Dataset:

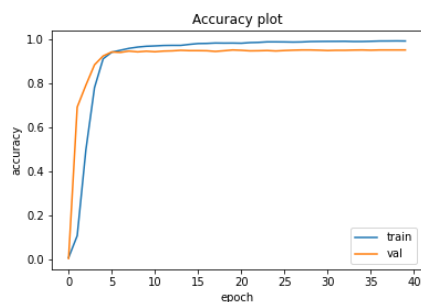


Accuracy vs Epoch

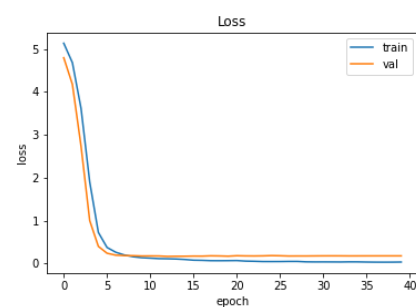


Loss vs Epoch

Dataset with Appended Data:



Accuracy vs Epoch



Loss vs Epoch

From the graphs above, we can see the advantages of utilizing transfer learning for the Dog Breed Identification Prediction Kaggle Competition. Using transfer learning, we are able to obtain high model accuracies with just a few epochs of training, after that, for the rest of the epochs, the accuracies and the losses of the models do not change much for the rest of the epochs, but the losses of the models are still able to be slightly improved with the rest of the epochs of training.

5. Leaderboard Performance

After training the models, we then used the models to make Dog Breed Identification predictions on the unseen and unlabeled test dog images dataset from the Dog Breed Identification Prediction Kaggle Competition, and for each image, the prediction will be an array of 120 values, which are the probabilities of the image for each class, the prediction probabilities of each test image is then put into a CSV file, which is then uploaded to the Dog Breed Identification Prediction Kaggle Competition page to be scored. The competition scores for each trained model are shown in the figures below.

5.1 Leaderboard Results of All Models

EfficientNetB7 CNN with Average() Layer Results:

Original Dataset:

Name	Submitted	Wait time	Execution time	Score
pred.csv	just now	1 seconds	2 seconds	0.35477

Complete

[Jump to your position on the leaderboard](#) ▼

Dataset with Appended Data:

Name	Submitted	Wait time	Execution time	Score
pred.csv	just now	1 seconds	2 seconds	0.36656

Complete

[Jump to your position on the leaderboard](#) ▼

Multi-Modal CNN with Average() Layer Results:

Original Dataset:

Name	Submitted	Wait time	Execution time	Score
pred.csv	just now	1 seconds	2 seconds	0.15502

Complete

[Jump to your position on the leaderboard](#) ▼

Dataset with Appended Data:

Name	Submitted	Wait time	Execution time	Score
pred.csv	just now	1 seconds	2 seconds	0.17542

Complete

[Jump to your position on the leaderboard](#) ▼

Multi-Modal CNN with Concatenate() Layer Results:

Original Dataset:

Name	Submitted	Wait time	Execution time	Score
pred (1).csv	just now	1 seconds	2 seconds	0.15290

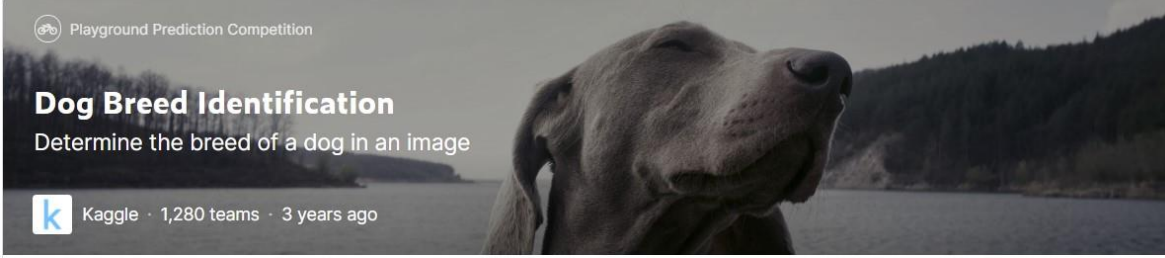
Complete

[Jump to your position on the leaderboard](#) ▼

Dataset with Appended Data:

Name	Submitted	Wait time	Execution time	Score
pred (2).csv	just now	1 seconds	2 seconds	0.16586
Complete				
Jump to your position on the leaderboard				

Best Leaderboard Performance:



Playground Prediction Competition

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
pred (1).csv	just now	1 seconds	2 seconds	0.15290
Complete				
Jump to your position on the leaderboard				

162	—	Shannon		0.14262	5	3y
163	—	treb1en		0.14273	52	3y
164	—	tweakers@4		0.14697	39	3y
165	—	VladimirGorea		0.14822	3	3y
166	—	Dave Coates		0.15267	24	3y
167	—	James Requa		0.15319	12	3y
168	—	fast.ai group 137		0.15321	27	3y
169	—	rikiya		0.15387	12	3y
170	—	Thiago		0.15421	25	3y
171	—	DEBattery		0.15707	11	3y
172	—	Seema Goel		0.15777	7	3y
173	—	agoila		0.15945	14	3y
174	—	fastai Igvaz		0.16052	34	3y
175	—	suvash		0.16059	6	3y

51-1280

[Load 1230 More](#)

From the figure above, we are able to observe that the best multi-class loss obtained by our team for the Dog Breed Identification Prediction Kaggle Competition is 0.15290 which is

obtained using the predictions made by the multi-modal CNN that has the Concatenate() layer for concatenating the feature maps of 5 individual SOTA CNN base models, trained on the original dataset from the Dog Breed Identification Prediction Kaggle Competition with no additional data appended. From the screenshot of the Dog Breed Identification Prediction Kaggle Competition Leaderboard, we are also able to observe that for the score of 0.15290, our team will take the 167th spot on the Leaderboard, out of 1280 places, which is equivalent to the top 13% of the Leaderboard.

6. Conclusion

The Dog Breed Identification classification problem has posed a fair share of challenges that has to be addressed before we could proceed to train our models for the task of Dog Breed Identification classification. The first challenge presented itself in the form of insufficient training samples for some Dog Breed classes, which we tried to counter by collecting more training images using the **Flickr** API and filtering the newly found images through a baseline model that we trained on the original Kaggle dataset to remove any false positive images. For the second challenge, the large training dataset of images caused training of the classification model to be computationally intensive, hence we utilized the Tensor Processing Unit(TPU) hardware accelerator to improve the overall training time for our classification task.

Once the challenges have been tackled, we proceed to use various State-of-the-Art(SOTA) machine learning algorithms for solving this complex machine learning task. Initially, we used the EfficientNetB7 CNN model for the Dog Breed Identification classification task. Even though the model was able to classify the different dog breeds decently, the performance of the model reached a threshold, even with the attempt to improve the accuracy by tuning the hyperparameters for the model. We realized that any individual SOTA CNN model is limited in their abilities to map all the features for the Dog Breed Identification task. The different SOTA CNN models from different families have different strengths in capturing the different features for the Dog Breed Identification task. Hence, we moved towards building a Multi-modal Convolutional Neural Network. For this, we ran tests for all SOTA CNN models to pick out the top five CNN models to be pooled together to form a Multi-modal Convolutional Neural Network, which is effective in mapping the different features for the Dog Breed Identification task. This model consists of the EfficientNetB7, NASNetLarge, InceptionResNetV2, Xception, and InceptionV3 models. This model yielded the best results out of all the models that we have tried. This sophisticated model has allowed us to obtain 167th spot on the leaderboard, translated to being in the top 13% on the Leaderboard.