CZ4042 Neural Networks and Deep Learning

Car Make Recognition

Group Project Report

S/N	Team Member Name	Matriculation Number	Tutorial Group
1	Prem Adithya Suresh	U1820740B	CS4
2	Shearman Chua Wei Jie	U1820058D	
3	Sam Jian Shen	U1821296L	

1. Introduction

Automated car model analysis, especially fine-grained car categorization and verification, can be used for multitudinous purposes in intelligent transportation systems, including vehicle regulation, description, and indexing. For instance, fine-grained car categorization can be exploited to inexpensively automate and expedite paying tolls from the lanes, based on different rates for different types of vehicles.

For this project, we will be working on the Comprehensive Cars (CompCars) Dataset from http://mmlab.ie.cuhk.edu.hk/datasets/comp_cars/index.html for our car image categorization task. The Comprehensive Cars (CompCars) dataset contains data from two scenarios, including images from web-nature and surveillance-nature. The web-nature data contains 163 car makes with 1,716 car models. There is a total of 136,726 images capturing the entire cars and 27,618 images capturing the car parts. The full car images are labelled with bounding boxes and viewpoints. Each car model is labelled with five attributes, including maximum speed, displacement, number of doors, number of seats, and type of car. The surveillance-nature data contains 50,000 car images captured in the front view.

The goal of this project is to design a neural network to classify a given image into one of the 163 car makes. After which consider a multi-task learning framework that classifies not only car makes, but also car models and the corresponding attributes. Some of the other sub-tasks include varying the network depth, tuning the network parameters to improve validation accuracy and validation loss. We will then observe the effects of utilizing a more advanced loss function on the neural network, to see if it improves the overall classification accuracy of the neural network.

2. Review of Existing Techniques

In the paper “H. Liu, Y. Tian, Y. Wang, L. Pang, T. Huang, “Deep relative distance learning: tell the difference between similar vehicles,” in Computer Vision and Pattern Recognition (CVPR), 2016”, H. Liu and his colleagues proposed a Deep Relative Distance Learning (DRDL) method which exploits a two-branch deep convolutional network to project raw vehicle images into an Euclidean space where distance can be directly used to measure the similarity of arbitrary two vehicles.

DRDL is an end-to-end framework specifically designed for vehicle re-identification. It aims to learn a deep convolutional network that can project raw vehicle images into an Euclidean space where the L2 distance can thus be used directly to measure the similarity of arbitrary two vehicles. The basic idea of DRDL is to minimize the distances of the same vehicle images and maximize those of other vehicles.

Prior to the paper, many researches made use of the triplet loss deep convolutional network architecture for a multitude of recognition tasks. The architecture connects a deep convolutional network for feature extraction and a special triplet loss to achieve good performance in both person re-identification and face recognition problems. It is assumed that samples of the same identity should be closer from each other compared to samples of different identities. By optimizing the specifically designed triplet loss function, the network will gradually learn a harmonic embedding of each input image in Euclidean space that tends to maximize the relative distance between the matched pair and the mismatched pair. However, generating all possible triplets would result in numerous triplets and most of them are too easy to distinguish that would not make any contribution to the loss convergence in the training phase.

The paper proposed an end-to-end framework DRDL that is suited for both vehicle retrieval and vehicle re-identification tasks. Two different vehicles (with different license plates) could be almost the same regarding their appearance if they belong to the same model. The research aimed to capture both the inter-model difference and intra-model difference between different vehicles. In order to make the training phase more stable and accelerate the convergence speed, the paper proposed a new loss function to replace the triplet loss, namely, coupled clusters loss (CCL). Similarly, a convolutional network is used to extract features for each image here. But the triplet input is replaced by two different image sets: one positive set and one negative set.

The paper made use of a base VGG CNN M 1024 network structure, containing 5 convolutional layers and 2 fully connected layers, for their experiments. The dimension of the network's last fully connected layer "fc7" is 1024 neurons. The last fully connected layer "fc8" is a mixed feature of both the vehicle's model information and the feature representation learned from single triplet loss or coupled clusters loss. The dimension of "fc8" is set to 1024 in accordance with the output dimension of standard VGG CNN M 1024 network to eliminate the influence of feature dimensional difference when performing evaluation experiments. "fc7 2" in the mixed difference network is just the same as the output feature of a standard VGG CNN M 1024 network while "fc8" is an enhanced one suitable for both inter-model difference and intra-model difference metric. The networks are all fine-tuned on VGG CNN M 1024 which is pre-trained with the ImageNet dataset in ILSVRC-2012. They used a momentum of μ = 0.9 and weight decay λ = 2 × 10−4.

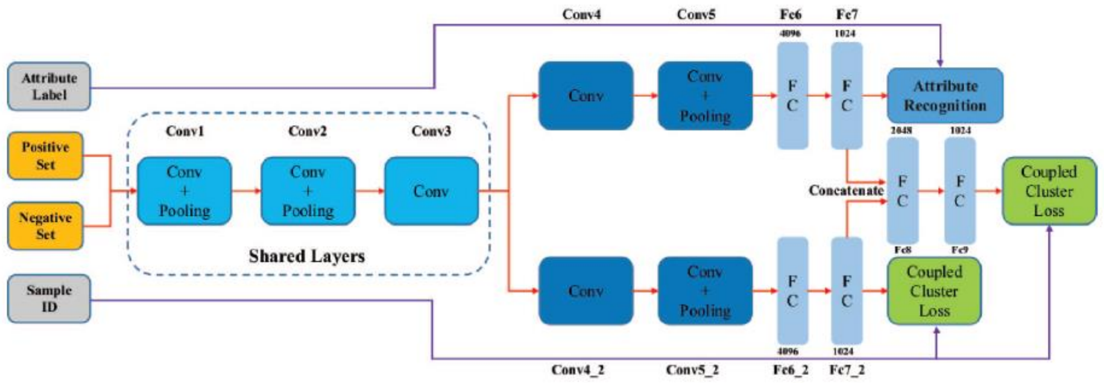


Fig.1 Mixed difference network structure based on VGG CNN M 1024 used in paper

2.1 Results

The network was trained on the “CompCars” dataset, using the Part-I subset(derived from L. Yang, P. Luo, C. C. Loy, X. Tang, “A large-scale car dataset for fine-grained categorization and verification,” in IEEE Computer Vision and Pattern Recognition (CVPR), 2015) which contains 431 car models with a total of 30955 images capturing the entire car and tested on vehicle verification task on the Part-III subset which contains 22236 images of 1145 models. The test data is organized into three sets, each of which has different difficulty, i. e. easy, medium, and hard. Each set contains 20,000 pairs of images, including 10,000 positive pairs and 10,000 negative pairs. Each image pair in the "easy set" is selected from the same viewpoint, while each pair in the "medium set" is selected from a pair of random viewpoints. Each negative pair in the "hard set" is chosen from the same car make.

Three other methods are introduced to perform the comparison experiments. The experimental results of the first two methods, “Deep Feature+SVM or Joint Bayesian”, are referred from “L. Yang, P. Luo, C. C. Loy, X. Tang, “A large-scale car dataset for fine-grained categorization and verification,” in IEEE Computer Vision and Pattern Recognition (CVPR), 2015”. They first utilize a deep convolutional network to train a vehicle model classification model on Part-I data of “CompCars”. Then, Joint Bayesian or SVM is applied to train a verification model on Part-II data with the classification network in step 1 as a feature extractor. The third algorithm, “VGG CNN M 1024” network with triplet loss function is trained with Part-I data of “CompCars” and the corresponding vehicle model labels. The results are given below:

Accuracy	Easy	Medium	Hard
GoogleNet+SVM	0.700	0.690	0.659
GoogleNet+Joint Bayesian	0.833	0.824	0.761
Mixed Diff+CCL	0.833	0.788	0.703

Fig.2 Results of Various Neural Networks in Paper

The “VGG CNN M 1024+Triplet Loss” got no results because its loss function failed to converge during the training phase.

3.Project Scope

3.1 Project Goal

The goal of this project is to classify a given image into one of the 163 car makes. After which consider designing a multi-task learning framework that classifies not only car makes, but also car models and the corresponding attributes.

In addition, try a more advanced loss function such as triplet loss for the neural network and compare the results of the model with the advanced loss function against that of the model with a baseline loss function (for example, SGD loss function).

3.2 Project Scope

For our team, we decided to narrow the scope of the project down to designing a multi-task learning framework that classifies a given image into one of the 163 car makes, as well as one of the car models.

Also, we will be varying the network depth, tuning the network parameters to try to improve the validation accuracy and validation loss of the neural network. We will then observe the effects of utilizing a more advanced loss function on the neural network, to see if it improves the overall classification accuracy as well as model loss of the neural network.

After discussing amongst our team, the finalized scope of our project is defined by:

1. Design a neural network that builds upon a previously published architecture for the purpose of car image categorization
2. Modify the hyperparameters of the network architecture by using a model selection search method (e.g. Grid Search,Random Search, Hyperband)
3. Design a multi-task learning framework that classifies not only car makes, but also car models
4. Try a more advanced loss function such as triplet loss for the neural network and compare the results with that of the neural network with a baseline loss function

4. Resources Used for Project

For this Project, our group will be building and training the neural network on the Google Colaboratory platform. The neural network will be coded in Python programming language. The reason behind this is that Python has many libraries that can be used to easily build neural networks. Also, by using the Google Colaboratory platform, we can utilize Tensor Processing Units (TPUs) developed by Google to allow our neural network to be trained faster.

4.1 Dataset

For our project, we will be using the Comprehensive Cars (CompCars) Dataset from http://mmlab.ie.cuhk.edu.hk/datasets/comp_cars/index.html for our car image categorization task. We will be splitting the dataset into training and validation datasets to be used to train and validate our neural network.

4.2 Python Libraries

For this project, we will be using the TensorFlow and Keras Python Libraries to build our neural network. As our team is utilizing the EfficientNetB5 architecture as the core of our neural network, we will need to import the **efficientnet.keras** library for the Keras (and TensorFlow Keras) reimplementation of EfficientNet, a lightweight convolutional neural network architecture achieving the state-of-the-art accuracy with an order of magnitude fewer parameters and FLOPS, on both ImageNet and five other commonly used transfer learning datasets.

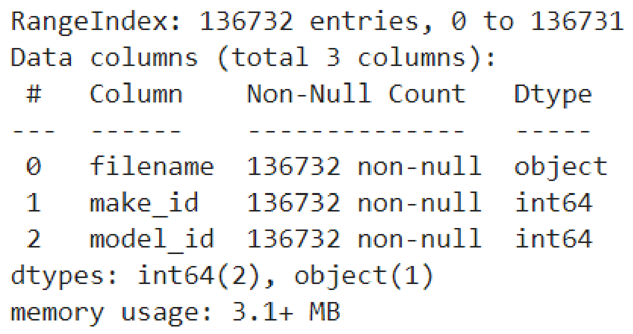
4.3 Tensor Processing Units (TPUs)

For our project, we will be using the Tensor Processing Units (TPUs) developed by Google when training our neural network. The reason for using the TPU accelerator for training our neural network is because by utilizing the TPU accelerator, we are able achieve a faster training time when training our neural network as compared to using a GPU accelerator or no accelerator at all.

In order to utilize the TPU accelerator in Google Colaboratory, we will be required to convert our images from the Comprehensive Cars (CompCars) Dataset as well as the labels into TFRecords, which is a format for storing a sequence of binary records.

5.Dataset Analysis and Preparation

5.1 Overview



```
RangeIndex: 136732 entries, 0 to 136731
Data columns (total 3 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   filename  136732 non-null  object
 1   make_id   136732 non-null  int64
 2   model_id  136732 non-null  int64
dtypes: int64(2), object(1)
memory usage: 3.1+ MB
```

Fig.3 Information regarding CompCars Dataset

From the figure above, we can observe that for the CompCars dataset that we have downloaded, there are 136,732 images in the dataset capturing the entire cars as well as their make_id labels and model_id that are available for us to use for the car image categorization task.

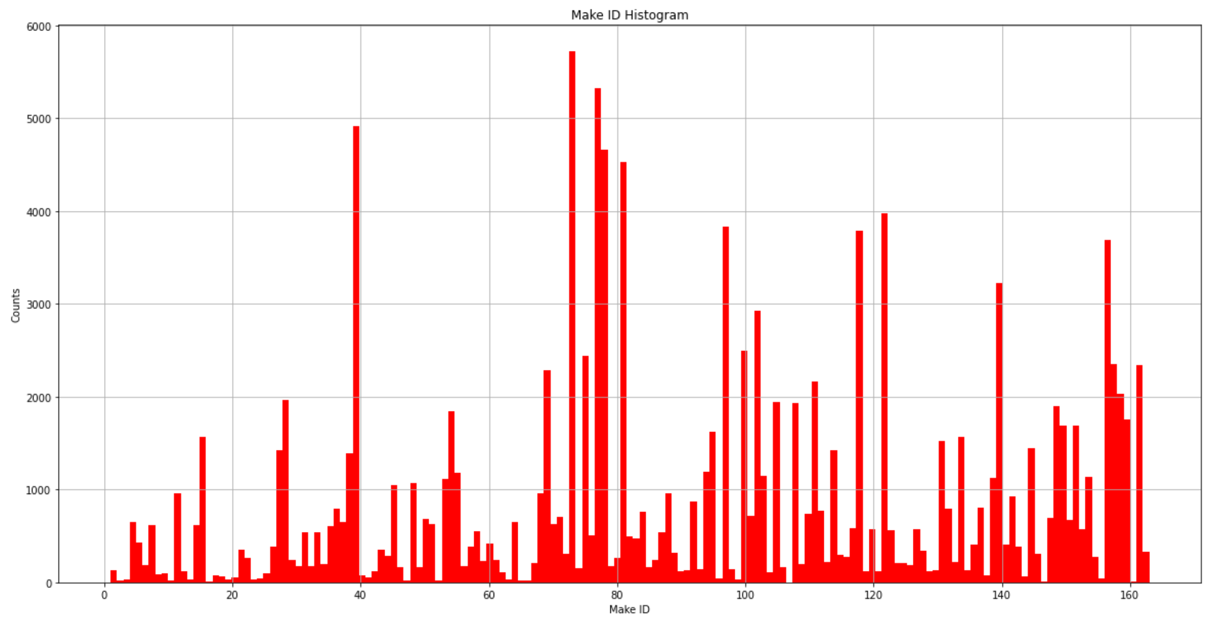


Fig.4 Histogram for Counts of Images for Each Make ID Category

In the figure above, we have plotted the histogram to help us visualize the number of images in the CompCars dataset for each of the car Make ID categories. The car makes in the CompCars dataset are labelled from 1 to 163, representing the 163 different car makes in the CompCar dataset. These are the following observations regarding the car makes in the CompCars dataset:

- We can see that all Make ID are ranging from 1 to 163 and there is a total of 163 unique IDs
- The counts for each Car Make are largely deviated from one another. Some of the Make IDs have counts as high as over 5000 images, and some as low as below 100 counts

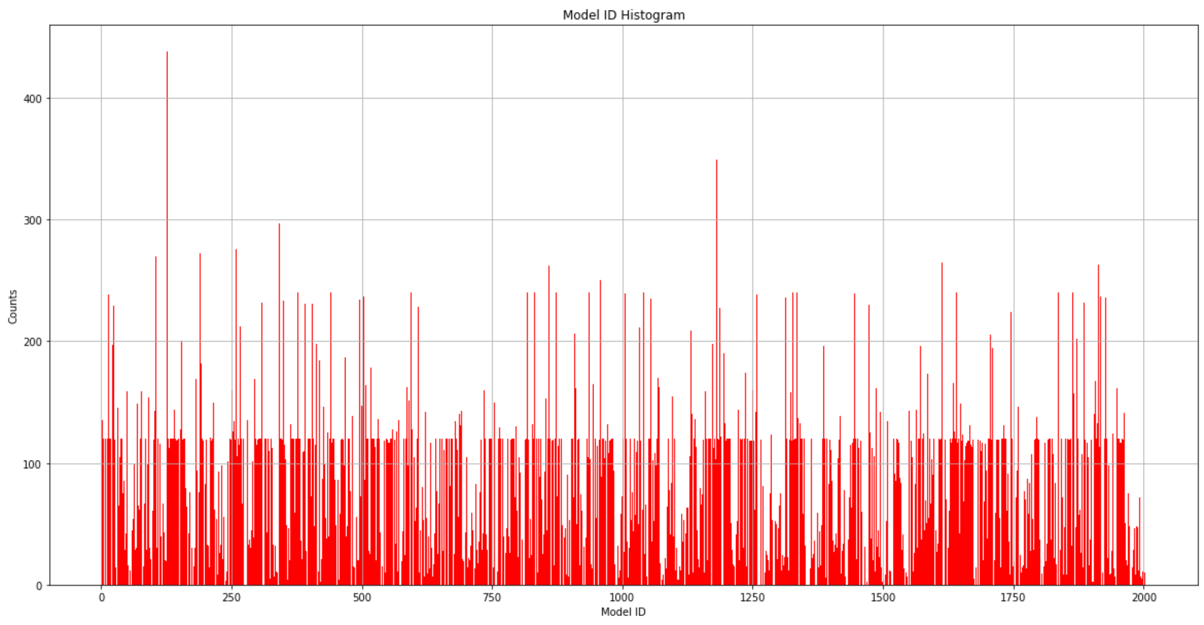


Fig.5 Histogram for Counts of Images for Each Model ID Category

In the figure above, we have plotted the histogram to help us visualize the number of images in the CompCars dataset for each of the car Model ID categories. There are 1716 unique car models in the CompCar dataset. These are the following observations regarding the car models in the CompCars dataset:

- Maximum Value of Model ID: 2004
- Minimum Value of Model ID: 1
- Total Number of Model Unique IDs: 1716
- Unlike Make ID, Model ID is not in running numerical order, there exist values between 1 to 2004 that do not belong in the Model ID labels.

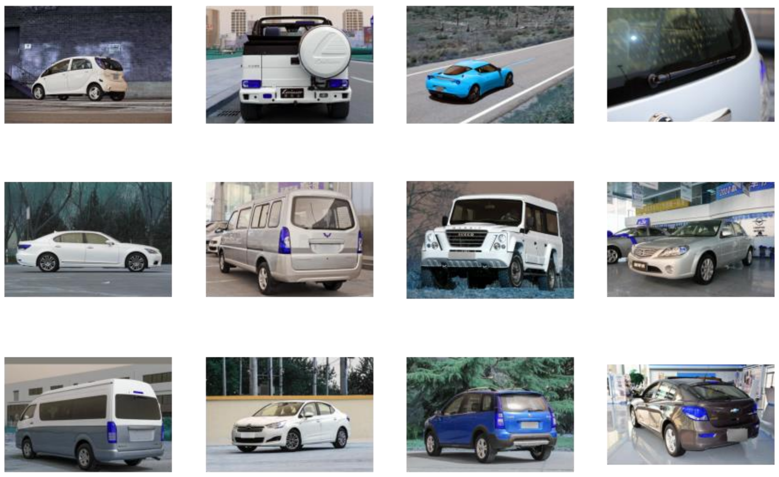


Fig.6 Example of Cars in CompCars Dataset

Above shows some of the sample images of cars that are in the CompCars dataset.

5.2 Dataset Preparation

5.2.1 Dropping Classes

As some of the classes are sparse, training the model to predict these classes may affect the overall accuracy of the model. Hence, sparse classes were dropped from the dataset resulting in 162 unique Make IDs and 1677 Model IDs down from 163 and 1716 respectively.

5.2.2 Splitting of Dataset

From the original CompCars dataset, we split the data into 3 portions for training, validating and testing of the neural network. The CompCars dataset was split into the 3 portions with the following ratio:

- Training Dataset - 67.5% of CompCars dataset
- Validation Dataset - 22.5% of CompCars dataset
- Test Dataset - 10% of CompCars dataset

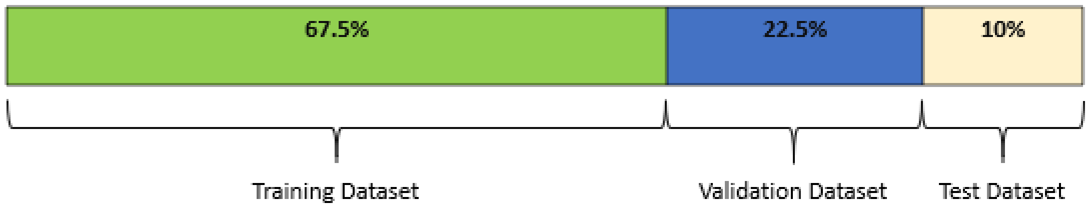
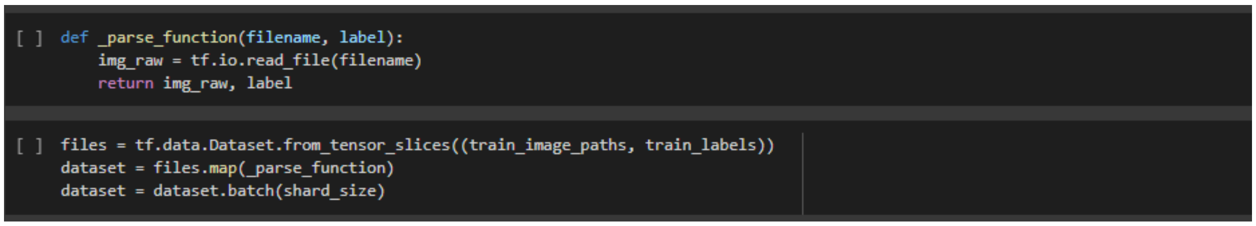


Fig.7 Training, Validation and Test Dataset Split

We split the dataset using **train_test_split** from **scikit-learn** to split the data into the 3 portions for training, validating and testing of the trained model. We split the data in a stratified fashion to ensure a balanced distribution of classes across the 3 potions where samples were shuffled before distribution.

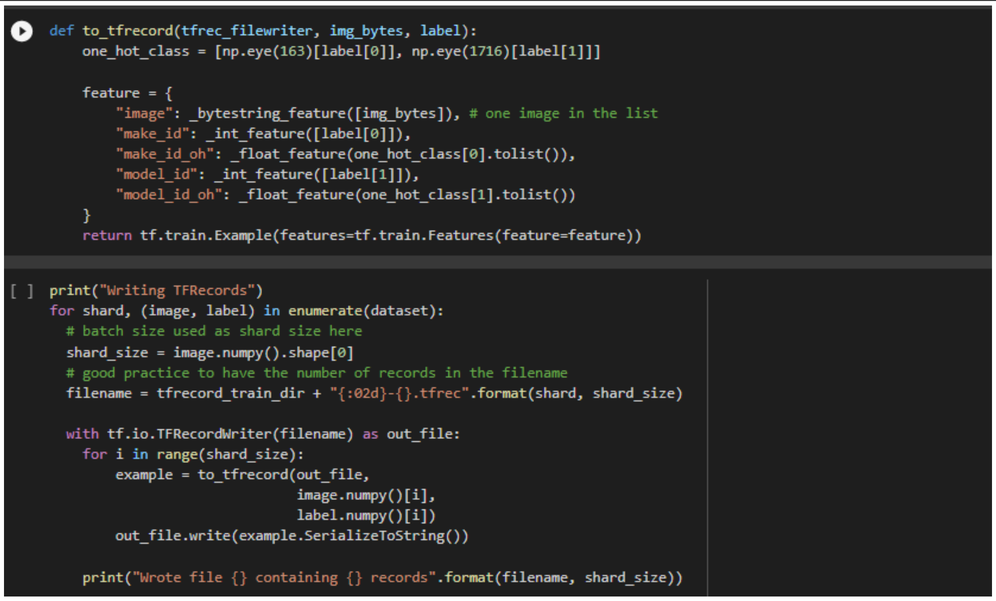
5.2.3 Converting Data to TFRecords

As mentioned, we will be using the TPU accelerator in Google Colaboratory when training the neural network. This will require us to convert the data and images we have into TFRecords format. We will be using the **tf.data** module for reading and writing data in TensorFlow.



```
def _parse_function(filename, label):
    img_raw = tf.io.read_file(filename)
    return img_raw, label
```

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



```
def to_tfrecord(tfrec_filewriter, img_bytes, label):
    one_hot_class = [np.eye(163)[label[0]], np.eye(1716)[label[1]]]

    feature = {
        "image": _bytestring_feature([img_bytes]), # one image in the list
        "make_id": _int_feature([label[0]]),
        "make_id_oh": _float_feature(one_hot_class[0].tolist()),
        "model_id": _int_feature([label[1]]),
        "model_id_oh": _float_feature(one_hot_class[1].tolist())
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))
```

```
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.8 Code to convert CompCars Dataset to TFRecords

In order to write CompCars dataset to TFRecords, we will be using the **tf.data.Dataset.from_tensor_slices()** method to create a dataset whose elements are slices of the given tensors which is stored as our variable **files**. Then, we will map the **files** variable to our **_parse_function()** function uses the **tf.io.read_file** method in tensorflow to read and output the entire contents of the input file names that contains the CompCars dataset images, in order for us to retrieve the images from our storage folder and store the results in the variable **dataset**. The dataset is then divided into batches to be stored as smaller **tfrec** files. We then iterate through each batch in the variable **dataset** and iterate through each image in the batch and encode the images using the to_tfrecord function that we have created, each image is then written out to their respective batch's **tfrec** files. Once we are done with this process, we will have:

- Training Dataset - 64 **tfrec** files
- Validation Dataset - 16 **tfrec** files
- Test Dataset - 8 **tfrec** files

6.Project Methods

6.1 Baseline Deep Neural Network

We first implemented a baseline Deep Neural Network to classify a given image into one of the 163 car makes as well as the 1716 car models. The Deep Neural Network will have a Multitask learning framework with 2 outputs, one for classifying the image into one of the car make and the other for classifying the image into one of the car models.

At the core of our network will be the EfficientNetB5 architecture which is a Convolutional Neural Network released in June 2019 by Google AI and is the new state-of-the-art on ImageNet. It introduces a systematic way to scale CNN (Convolutional Neural Networks) in a nearly optimal way. We then add a few more dense layers after the EfficientNetB5 architecture, followed by the 2 output layers to form our Neural Network to classify images into the car makes as well as the car models. For our baseline model, we will be using the SGD (Stochastic Gradient Descent) optimizer. Below shows the diagram of the Neural Network architecture:

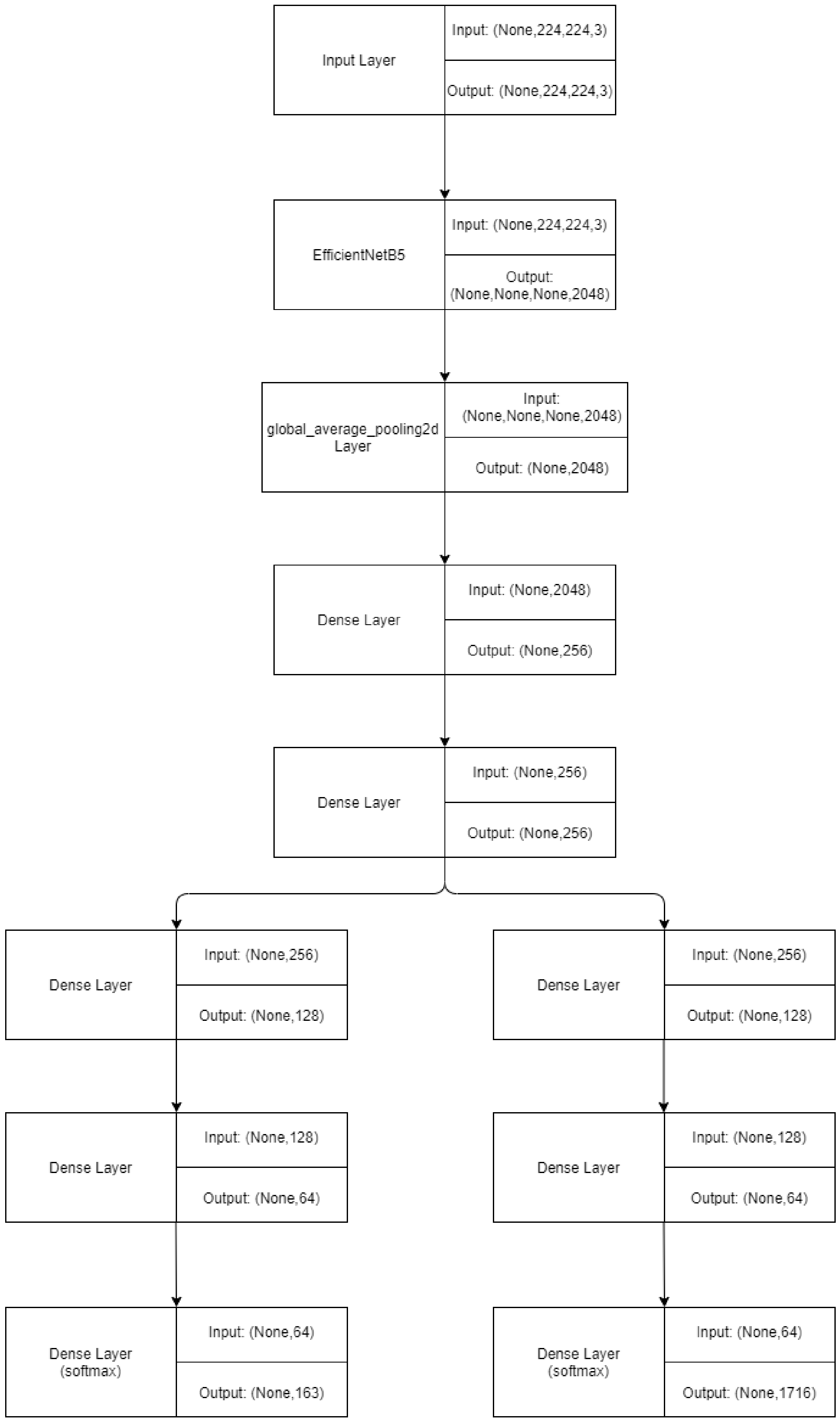


Fig.9 Neural Network Architecture

6.2 Deep Neural Network with Tuned Hyperparameters (Hyperband)

After implementing and training the baseline neural network for categorizing the car images into car makes and car models, to improve the validation accuracy and validation loss of the neural network, we will tune the hyperparameters of the neural network. Hyperparameters are points of choice or configuration that allow a machine learning model to be customized for a specific task or dataset.

For tuning of the hyperparameters for the neural network, our group has decided that we will be using the Hyperband tuning algorithm.

The Hyperband tuning algorithm is an optimized version of random search which uses early-stopping to speed up the hyperparameter tuning process. The Hyperband tuning algorithm uses adaptive resource allocation and early-stopping to quickly converge on a high-performing model. This is done using a sports championship style bracket. The algorithm trains many models for a few epochs and carries forward only the top-performing half of models to the next round. Hyperband determines the number of models to train in a bracket by computing 1 + $\log_{\text{factor}}$(max_epochs) and rounding it up to the nearest integer.

Before running the hyperparameter search, a callback is defined to clear the training outputs at the end of every training step when using the Hyperband tuning algorithm.

6.3 Deep Neural Network with Advanced Loss Function

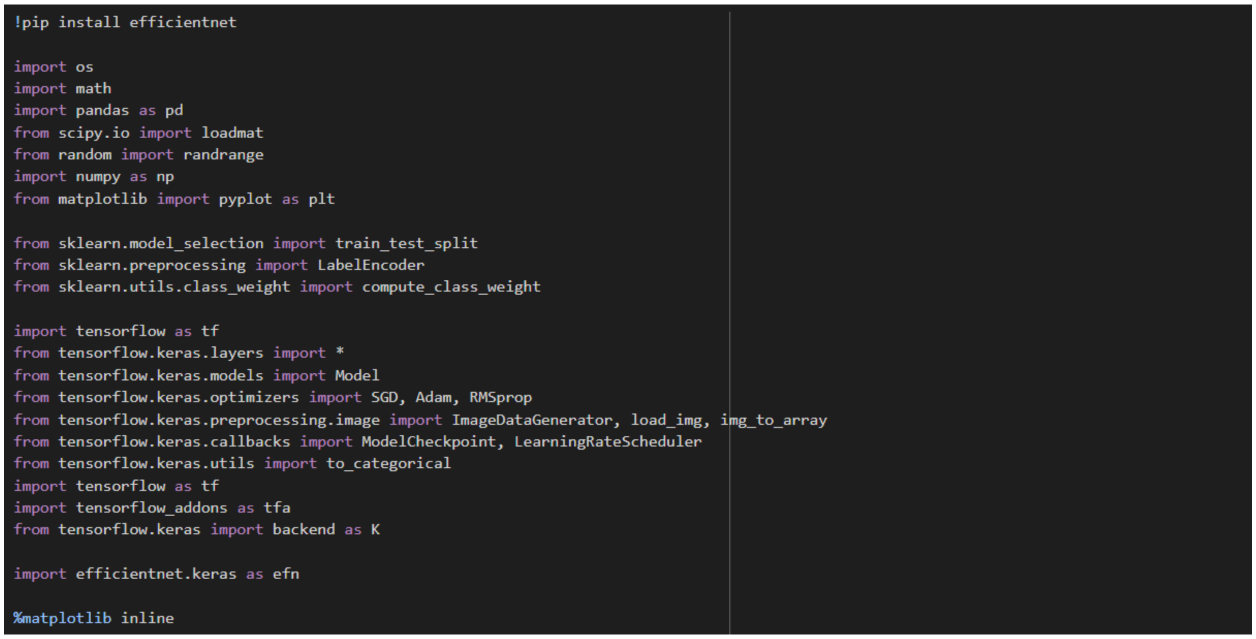
In addition, to fulfill our project scope, we will be trying a more advanced loss function for the neural network. For our project, we have decided to go with the Focal loss function, as the advanced loss function of choice for our neural network.
Focal loss was first introduced in the RetinaNet paper (https://arxiv.org/pdf/1708.02002.pdf). Focal loss is extremely useful for classification when you have highly imbalanced classes. It down-weights well-classified examples and focuses on hard examples. The loss value is much higher for a sample which is misclassified by the classifier as compared to the loss value corresponding to a well-classified example.
We will then compare the validation accuracy and loss of the neural network with the advanced loss function against the validation accuracy and loss of the neural network with the baseline **categorical_crossentropy** loss function.

7.Experiments and Results

7.1 Pre-Experiment Setup

Before we build and train our neural network, we will need to set up the Google Colaboratory notebook by importing the Python packages we require for our project, enable and test the TPU accelerator,as well as read and prepare the TFRecords that we have created for our car image categorization task.

7.1.1 Import Python Packages

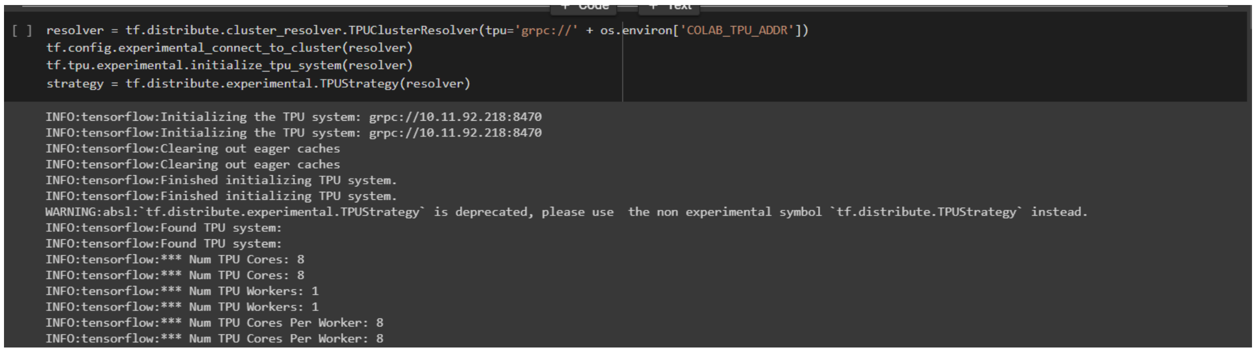


```
!pip install efficientnet

import os
import math
import pandas as pd
from scipy.io import loadmat
from random import randrange
import numpy as np
from matplotlib import pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.utils.class_weight import compute_class_weight

import tensorflow as tf
from tensorflow.keras.layers import *
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam, RMSprop
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateScheduler
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
import tensorflow_addons as tfa
from tensorflow.keras import backend as K

import efficientnet.keras as efn

%matplotlib inline
```

Fig.10 Imported Packages

As can be seen from the figure above, we will first have to install the **efficientnet** package for the EfficientNetB5 architecture that we will be using as the core of our neural network. We will also need to import the various **sklearn**, **TensorFlow** and **Keras** packages that we will be using for our project experiments.

7.1.2 Enabling TPU Accelerator



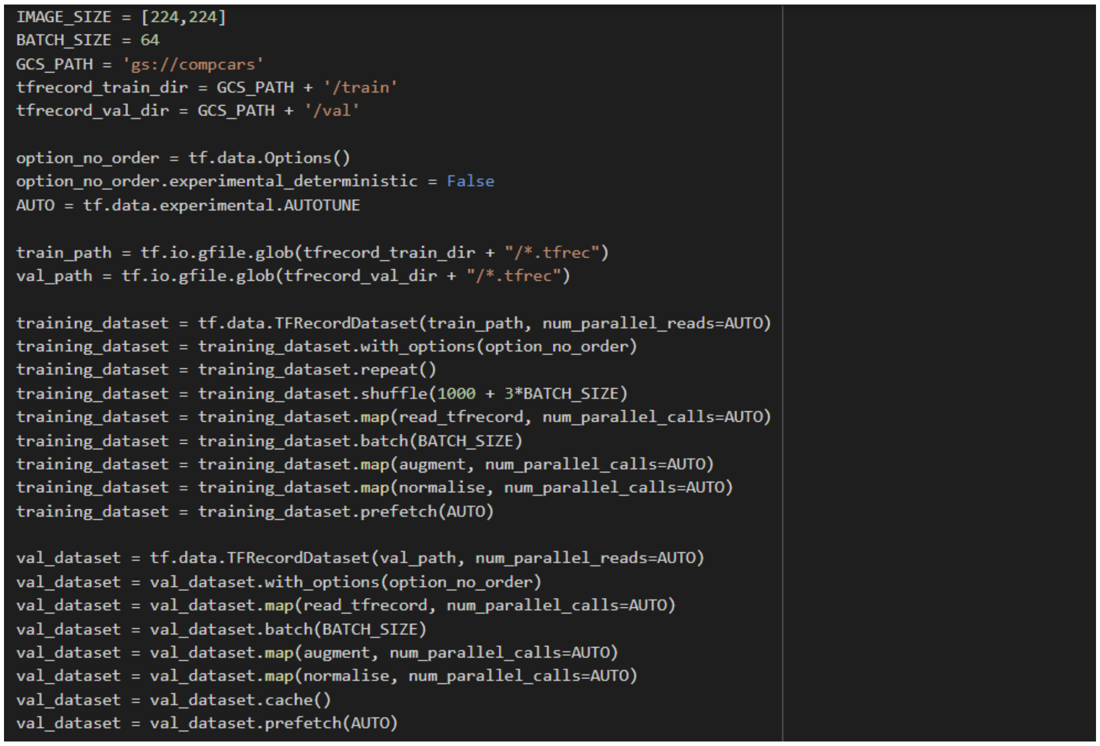
```
[ ] resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc://' + os.environ['COLAB_TPU_ADDR'])
    tf.config.experimental_connect_to_cluster(resolver)
    tf.tpu.experimental.initialize_tpu_system(resolver)
    strategy = tf.distribute.experimental.TPUStrategy(resolver)

    INFO:tensorflow:Initializing the TPU system: grpc://10.11.92.218:8470
    INFO:tensorflow:Initializing the TPU system: grpc://10.11.92.218:8470
    INFO:tensorflow:Clearing out eager caches
    INFO:tensorflow:Clearing out eager caches
    INFO:tensorflow:Finished initializing TPU system.
    INFO:tensorflow:Finished initializing TPU system.
    WARNING:absl:`tf.distribute.experimental.TPUStrategy` is deprecated, please use  the non experimental symbol `tf.distribute.TPUStrategy` instead.
    INFO:tensorflow:Found TPU system:
    INFO:tensorflow:Found TPU system:
    INFO:tensorflow:*** Num TPU Cores: 8
    INFO:tensorflow:*** Num TPU Cores: 8
    INFO:tensorflow:*** Num TPU Workers: 1
    INFO:tensorflow:*** Num TPU Workers: 1
    INFO:tensorflow:*** Num TPU Cores Per Worker: 8
    INFO:tensorflow:*** Num TPU Cores Per Worker: 8
```

Fig.11 Enabling TPU Accelerator

Next, we will need to enable Google's Cloud TPUs to allow us to make use of them for training our neural network, which will greatly reduce the training time needed to train our models.

7.1.3 Preparing Data for Neural Network

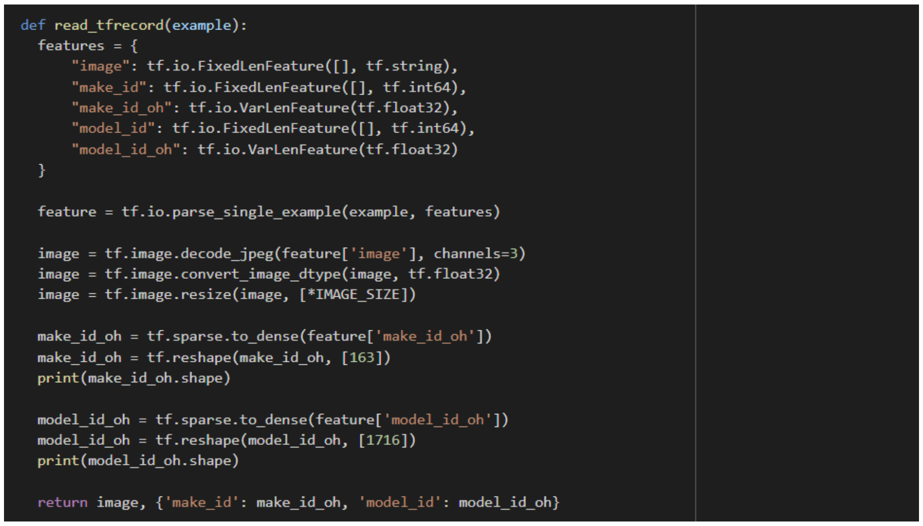
Before we can start to train our neural network, we will have to retrieve the TFRecords that we have prepared as well as to augment and normalize each image to help us achieve better validation accuracy during training.



```
IMAGE_SIZE = [224,224]
BATCH_SIZE = 64
GCS_PATH = 'gs://compcars'
tfrecord_train_dir = GCS_PATH + '/train'
tfrecord_val_dir = GCS_PATH + '/val'

option_no_order = tf.data.Options()
option_no_order.experimental_deterministic = False
AUTO = tf.data.experimental.AUTOTUNE

train_path = tf.io.gfile.glob(tfrecord_train_dir + "/*.tfrec")
val_path = tf.io.gfile.glob(tfrecord_val_dir + "/*.tfrec")

training_dataset = tf.data.TFRecordDataset(train_path, num_parallel_reads=AUTO)
training_dataset = training_dataset.with_options(option_no_order)
training_dataset = training_dataset.repeat()
training_dataset = training_dataset.shuffle(1000 + 3*BATCH_SIZE)
training_dataset = training_dataset.map(read_tfrecord, num_parallel_calls=AUTO)
training_dataset = training_dataset.batch(BATCH_SIZE)
training_dataset = training_dataset.map(augment, num_parallel_calls=AUTO)
training_dataset = training_dataset.map(normalise, num_parallel_calls=AUTO)
training_dataset = training_dataset.prefetch(AUTO)

val_dataset = tf.data.TFRecordDataset(val_path, num_parallel_reads=AUTO)
val_dataset = val_dataset.with_options(option_no_order)
val_dataset = val_dataset.map(read_tfrecord, num_parallel_calls=AUTO)
val_dataset = val_dataset.batch(BATCH_SIZE)
val_dataset = val_dataset.map(augment, num_parallel_calls=AUTO)
val_dataset = val_dataset.map(normalise, num_parallel_calls=AUTO)
val_dataset = val_dataset.cache()
val_dataset = val_dataset.prefetch(AUTO)
```

Fig.12 Code to Fetch TFRecords from Google Cloud Storage

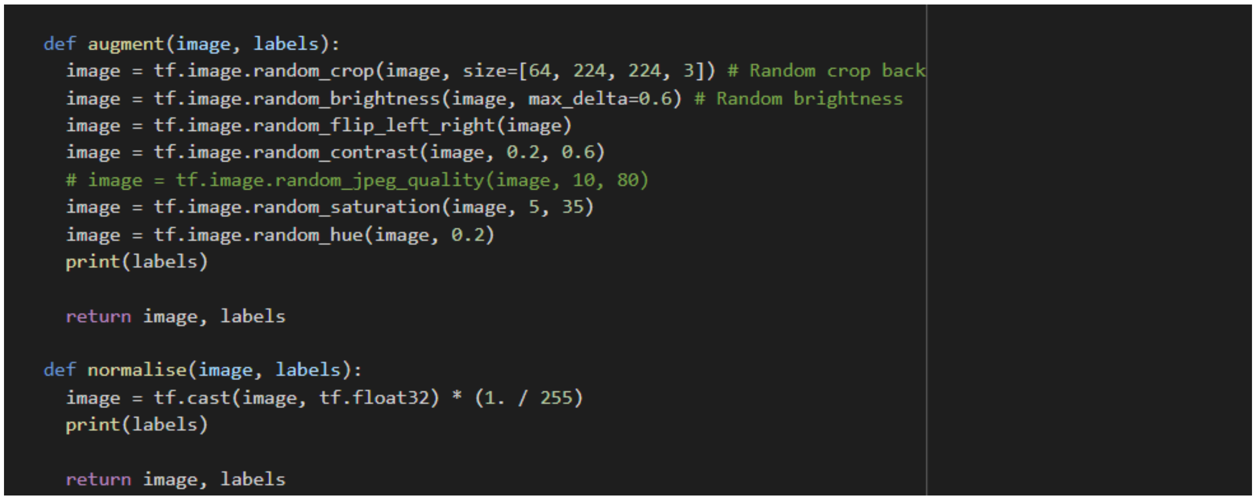
The figure above shows the code for fetching the training dataset and the validation dataset from our Google Cloud Storage. As the data is now stored as TFRecords, we will have to read and decode the TFRecords using the **read_tfrecord** function, after which, **augment** and **normalise** the image data using the augment and normalize functions respectively.



```python
def read_tfrecord(example):
  features = {
      "image": tf.io.FixedLenFeature([], tf.string),
      "make_id": tf.io.FixedLenFeature([], tf.int64),
      "make_id_oh": tf.io.VarLenFeature(tf.float32),
      "model_id": tf.io.FixedLenFeature([], tf.int64),
      "model_id_oh": tf.io.VarLenFeature(tf.float32)
  }

  feature = tf.io.parse_single_example(example, features)

  image = tf.image.decode_jpeg(feature['image'], channels=3)
  image = tf.image.convert_image_dtype(image, tf.float32)
  image = tf.image.resize(image, [*IMAGE_SIZE])

  make_id_oh = tf.sparse.to_dense(feature['make_id_oh'])
  make_id_oh = tf.reshape(make_id_oh, [163])
  print(make_id_oh.shape)

  model_id_oh = tf.sparse.to_dense(feature['model_id_oh'])
  model_id_oh = tf.reshape(model_id_oh, [1716])
  print(model_id_oh.shape)

  return image, {'make_id': make_id_oh, 'model_id': model_id_oh}
```

Fig.13 Code to read and decode TFRecords

In the figure above, we have the **read_tfrecord** function. For each record in the TFRecord file, the function first parses the record, and then decodes and resizes the image so that each image we feed into the neural network is of the same size, as well as extract the Car Make and Car Model labels for the image. The function then outputs the image decoded as well as its labels for each image record in the TFRecord file.

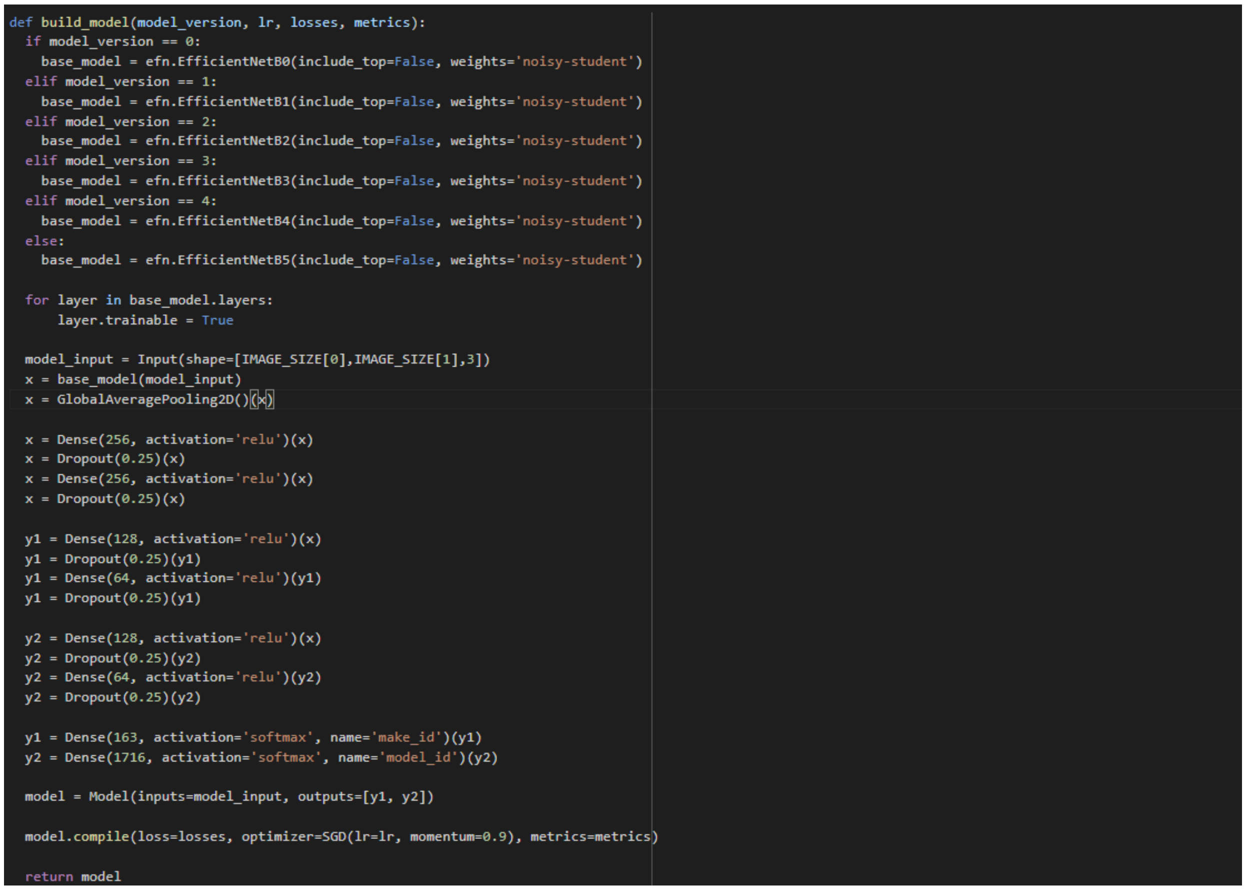


```python
def augment(image, labels):
  image = tf.image.random_crop(image, size=[64, 224, 224, 3]) # Random crop back
  image = tf.image.random_brightness(image, max_delta=0.6) # Random brightness
  image = tf.image.random_flip_left_right(image)
  image = tf.image.random_contrast(image, 0.2, 0.6)
  # image = tf.image.random_jpeg_quality(image, 10, 80)
  image = tf.image.random_saturation(image, 5, 35)
  image = tf.image.random_hue(image, 0.2)
  print(labels)

  return image, labels

def normalise(image, labels):
  image = tf.cast(image, tf.float32) * (1. / 255)
  print(labels)

  return image, labels
```

Fig.14 Code to augment and normalize images

After reading and decoding the TFRecords, we will have to augment and normalize each of the images in our dataset. The reason for augmenting the images is so that the neural network will not overfit on the training dataset images when we are training the images, by giving each image in both the training and validation dataset random cropping, brightness level, flipping the image, as well as changing the various features of the image, the neural network is able to learn better and improve the performance and ability of the model to generalize.
After which, we normalize each of the image data so that the images in our dataset will be more consistent and the scale of the data features will be of the same range.

7.2 Build Model Function and Learning Rate Curve

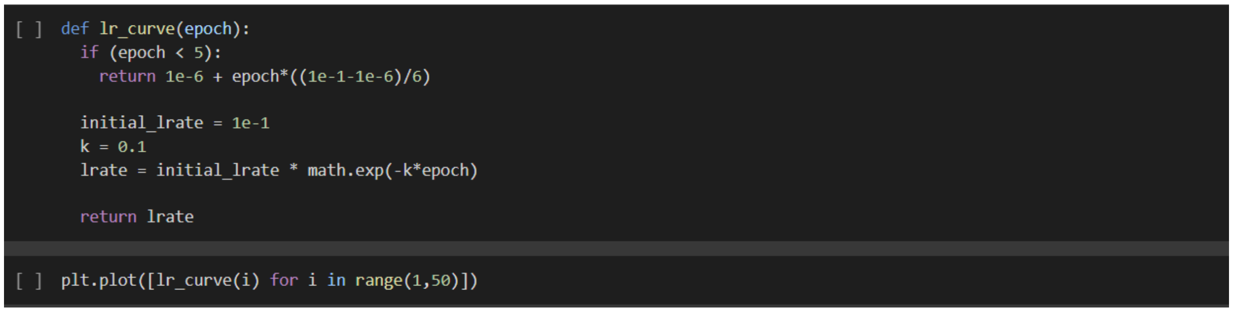


```python
def build_model(model_version, lr, losses, metrics):
  if model_version == 0:
    base_model = efn.EfficientNetB0(include_top=False, weights='noisy-student')
  elif model_version == 1:
    base_model = efn.EfficientNetB1(include_top=False, weights='noisy-student')
  elif model_version == 2:
    base_model = efn.EfficientNetB2(include_top=False, weights='noisy-student')
  elif model_version == 3:
    base_model = efn.EfficientNetB3(include_top=False, weights='noisy-student')
  elif model_version == 4:
    base_model = efn.EfficientNetB4(include_top=False, weights='noisy-student')
  else:
    base_model = efn.EfficientNetB5(include_top=False, weights='noisy-student')

  for layer in base_model.layers:
      layer.trainable = True

  model_input = Input(shape=[IMAGE_SIZE[0],IMAGE_SIZE[1],3])
  x = base_model(model_input)
  x = GlobalAveragePooling2D()(x)

  x = Dense(256, activation='relu')(x)
  x = Dropout(0.25)(x)
  x = Dense(256, activation='relu')(x)
  x = Dropout(0.25)(x)

  y1 = Dense(128, activation='relu')(x)
  y1 = Dropout(0.25)(y1)
  y1 = Dense(64, activation='relu')(y1)
  y1 = Dropout(0.25)(y1)

  y2 = Dense(128, activation='relu')(x)
  y2 = Dropout(0.25)(y2)
  y2 = Dense(64, activation='relu')(y2)
  y2 = Dropout(0.25)(y2)

  y1 = Dense(163, activation='softmax', name='make_id')(y1)
  y2 = Dense(1716, activation='softmax', name='model_id')(y2)

  model = Model(inputs=model_input, outputs=[y1, y2])

  model.compile(loss=losses, optimizer=SGD(lr=lr, momentum=0.9), metrics=metrics)

  return model
```

Fig.15 Code for building and compiling Neural Network Model

We first define a function called **build_model** which takes in the **model_version**, **lr**, **losses**, and **metrics** as arguments. The **model_version** is used to determine which version of the EfficientNet Convolutional Neural Network to use; for the baseline model, we will be using EfficientNetB5. The **lr** argument tells us the initial learning rate that we want to give the model. The losses argument tells us the **loss** function to use for the neural network and the **metrics** argument gives a function that is used to judge the performance of our model. We will also be using the **SGD** optimiser for the baseline model.

The **build_model** then sequential build our neural network, starting with the input layer which takes in an input of a 224 x 224 size image with 3 color channels, then the **base_model** which is the EfficientNetB5 Convolutional Neural Network, after that we have a Global Average Pooling layer, next we have 2 dense layers of 256 neurons and ‘relu’ activation function with their respective dropout layers with a dropout rate of 0.25. The model then split into 2 tails for a Multitask learning framework with 2 outputs, one for classifying the image into one of the car make and the other for classifying the image into one of the car models. The two tails of the model both have a dense layer of 128 neurons and ‘relu’ activation function, dropout layer with a dropout rate of 0.25, dense layer of 64 neurons and ‘relu’ activation function, another dropout layer with a dropout rate of 0.25. Then for the tail to classify the image into one of the car make, there is an output dense layer with ‘softmax’ activation and 163 neurons representing the 163 unique car makes, and for the other tail to classify the image into one of the car model, there is an output dense layer with ‘softmax’ activation and 1716 neurons representing the 1716 unique car models.



```
[ ]  def lr_curve(epoch):
       if (epoch < 5):
         return 1e-6 + epoch*((1e-1-1e-6)/6)

       initial_lrate = 1e-1
       k = 0.1
       lrate = initial_lrate * math.exp(-k*epoch)

       return lrate
```

```
[ ]  plt.plot([lr_curve(i) for i in range(1,50)])
```

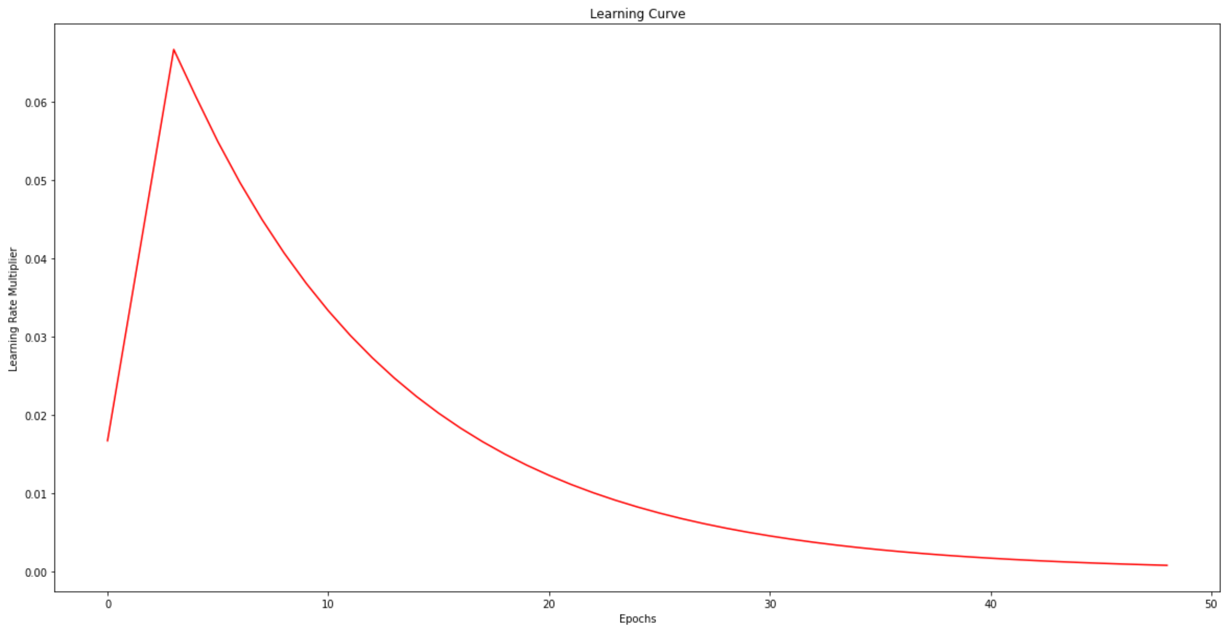


Fig.16 Code for Learning Rate Curve

In order to make our model more stable when learning the CompCars dataset, we will be using a Learning Rate Curve to vary the learning rate of the model while it is training. As can be seen from the learning rate curve plot in the figure, initially the learning rate is high, and it exponentially decreases with each training epoch. This ensures that smaller changes are made to the weights at each epoch when the model is training, in order to produce a more stable learning.

7.3 Baseline Deep Neural Network

7.3.1 Building the Baseline Deep Neural Network

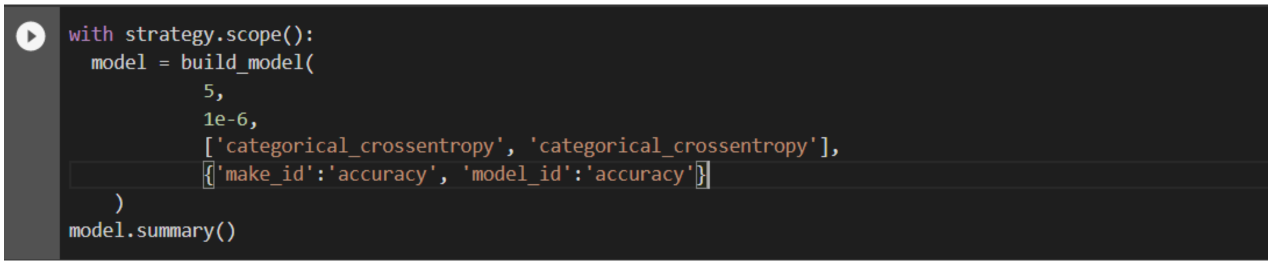

```
with strategy.scope():
  model = build_model(
            5,
            1e-6,
            ['categorical_crossentropy', 'categorical_crossentropy'],
            {'make_id':'accuracy', 'model_id':'accuracy'}
    )
model.summary()
```

Fig.17 Building Baseline Deep Neural Network

The figure above shows the code used to build the Baseline Deep Neural Network for the Car Categorization task. We build the model using **with strategy.scope()** to ensure that our neural network is using the TPU hardware accelerator when training. We call the **build_model** function with the parameters **5** for EfficientNetB5 Convolutional Neural Network to be the base of our model, **1e-6** as the initial learning rate, and the loss function for the model will be **categorical_crossentropy** and the 2 metrics to judge the performance of our model is the **accuracy** for classifying the make_id and model_id labels.

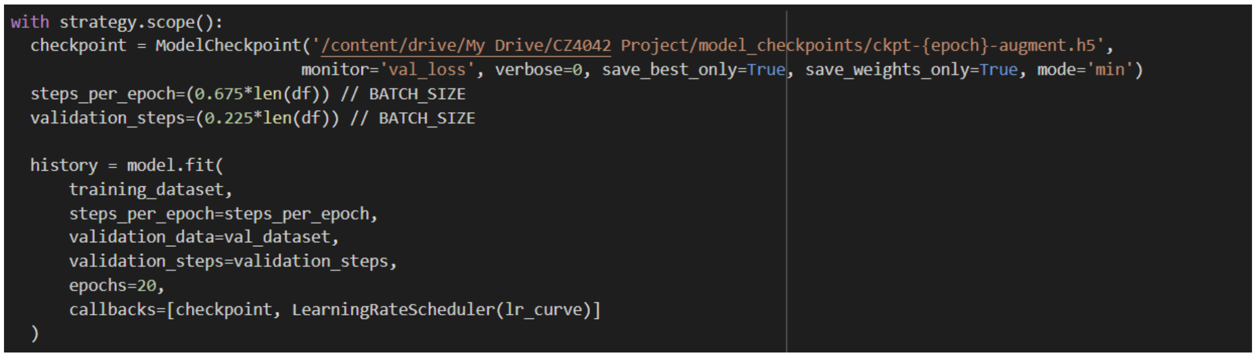

```
with strategy.scope():
  checkpoint = ModelCheckpoint('/content/drive/My Drive/CZ4042 Project/model_checkpoints/ckpt-{epoch}-augment.h5',
                              monitor='val_loss', verbose=0, save_best_only=True, save_weights_only=True, mode='min')
  steps_per_epoch=(0.675*len(df)) // BATCH_SIZE
  validation_steps=(0.225*len(df)) // BATCH_SIZE

  history = model.fit(
      training_dataset,
      steps_per_epoch=steps_per_epoch,
      validation_data=val_dataset,
      validation_steps=validation_steps,
      epochs=20,
      callbacks=[checkpoint, LearningRateScheduler(lr_curve)]
  )
```

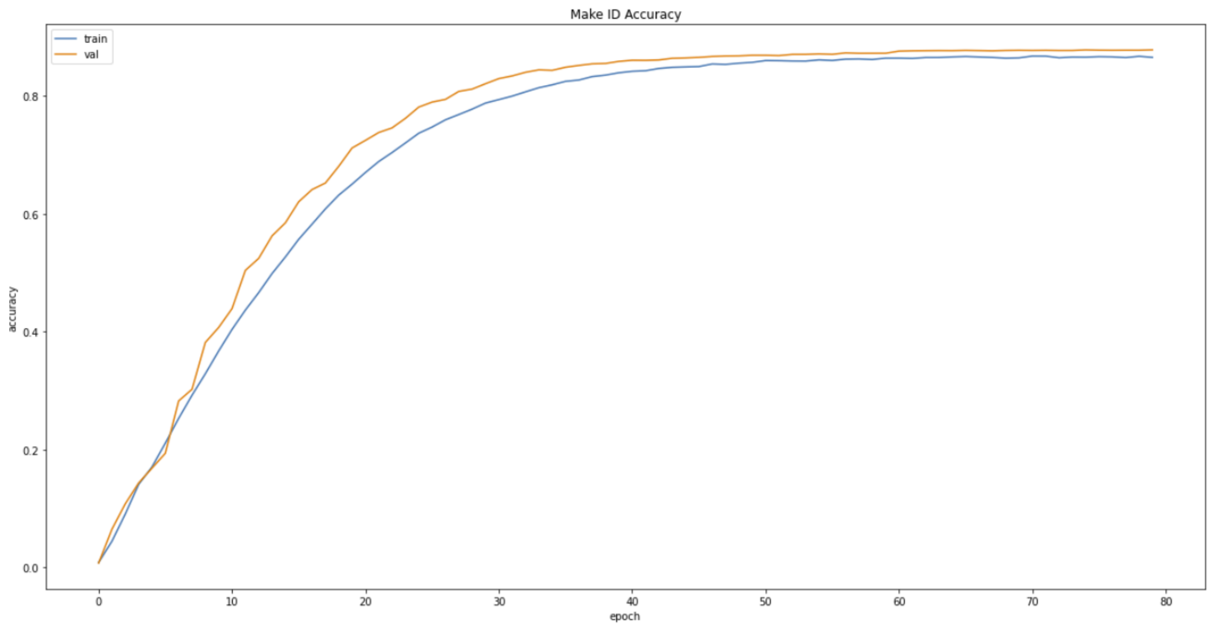
Fig.18 Fitting the Neural Network

Next, we train the model by calling the model.fit function. The model is trained using the training dataset that we have prepared and validated on the validation dataset that we have prepared. We also checkpoint the model if the model reduces its validation loss from the last best validation loss from previous epochs.

As the connection to the TPU in Google Colaboratory cuts off after roughly training 20 epochs, we train the model in batches of 20 epochs, using the saved model weights from the last batch until we have trained the model for a total of 80 epochs.

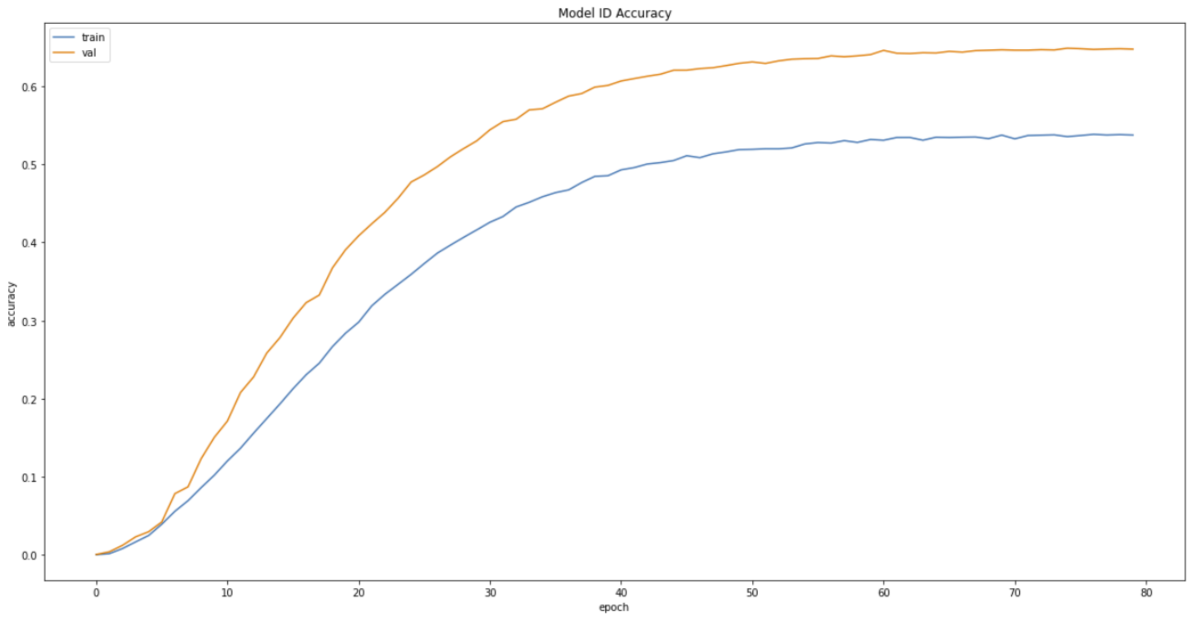
Also, since we are training the model batches of 20 epochs, we will save the model's training history in CSV files which will be combined together in one CSV file after running 80 epochs to allow us to plot the loss and accuracy graphs for the neural network.

7.3.2 Experiment Results



<u>Fig.19 Make ID Accuracy vs. Epochs</u>

From the figure above, we can see the plot of the training accuracy and validation accuracy vs. epochs for classifying the images into car make IDs. Our baseline model is performing relatively well, where we can achieve a 0.8783 validation accuracy and 0.8656 training accuracy at the last training epoch. From the graph we can also see that the model is not overfitting on the training data, with the validation accuracy higher than that of the training accuracy for all the 80 training epochs. The model converges at around 40 epochs for learning the classification of the images into car makes.



<u>Fig.20 Model ID Accuracy vs. Epochs</u>

From the figure above, we can see the plot of the training accuracy and validation accuracy vs. epochs for classifying the images into car model IDs. Our baseline model can achieve a 0.6468 validation accuracy and 0.5370 training accuracy at the last training epoch. From the graph we can also see that the model is not overfitting on the training data, with the validation accuracy higher than that of the training accuracy for all the 80 training epochs. The model converges at around 80 epochs for learning the classification of the images into car models.

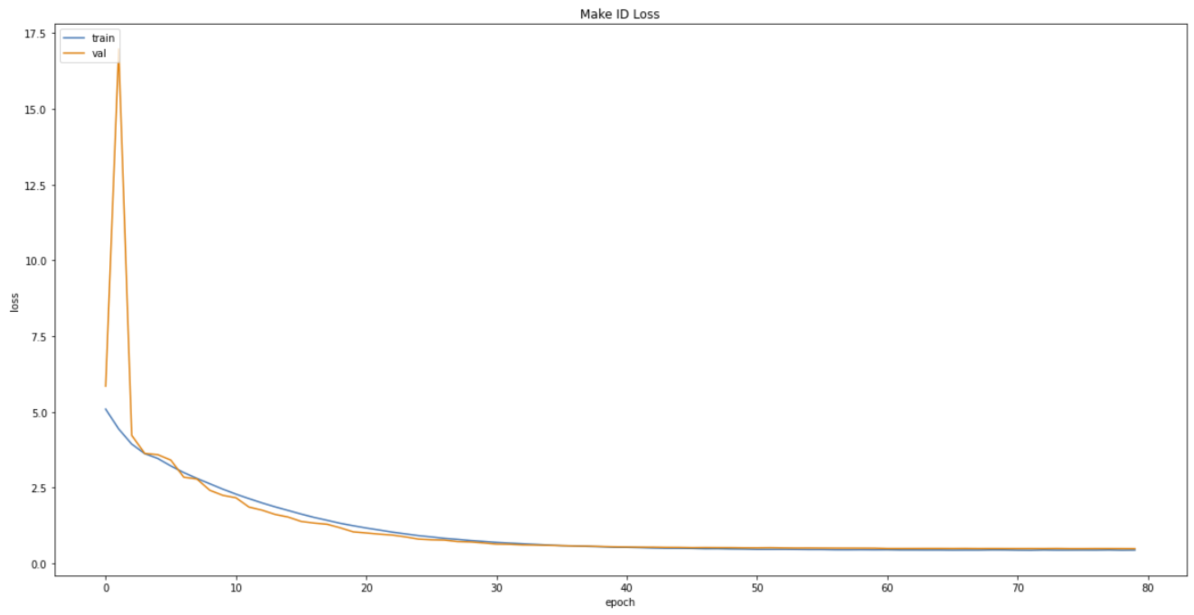


Fig.21 Make ID Loss vs. Epochs

The figure above shows the plot of the training loss and validation loss vs. epochs for classifying the images into car make IDs. From the plot, we can observe that the baseline model is learning the dataset well, with both the validation and training loss decreasing with each epoch and converging at around 50 epochs.

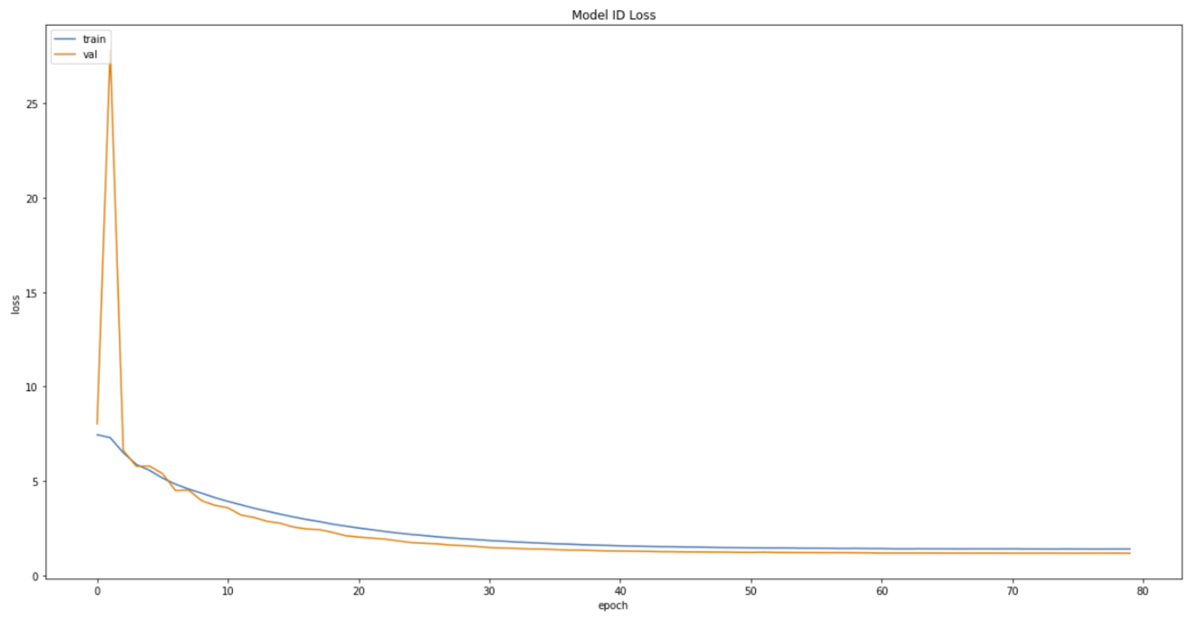
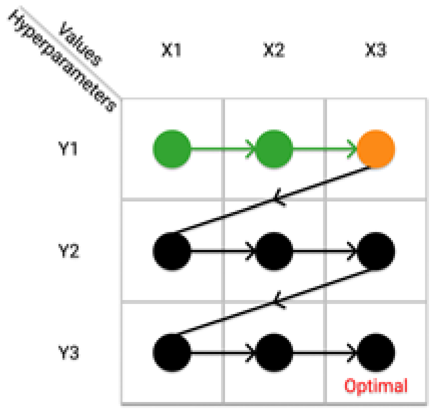


Fig.22 Model ID Loss vs. Epochs

The figure above shows the plot of the training loss and validation loss vs. epochs for classifying the images into car model IDs. From the plot, we can observe that the baseline model is learning well, with both the validation and training loss decreasing with each epoch, and the losses are still decreasing at 80 epochs of training, although the decrease is very minute due to the reduced learning rate with each increasing epoch.

7.4 Deep Neural Network Hyperparameters Tuning (Hyperband Tuning algorithm)

7.4.1 Why we choose Hyperband Tuning algorithm and how it works?



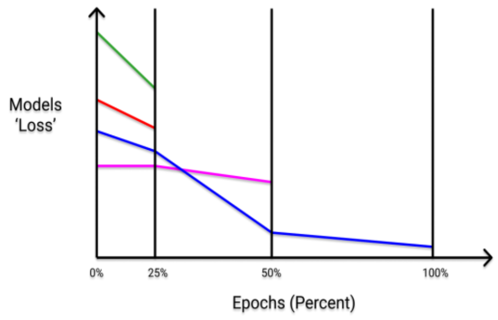


Fig 23. Visual Representation of Grid Search Technique **Fig 24. Visual Representation of Hyperband Technique**

The simplest implementation for neural network optimal hyperparameters search is the Grid Search Technique. This technique tries every possible combination for the given sets of hyperparameters values. The pattern illustrated in Fig 23 shows how it sequentially performs the search, where each set of hyperparameters is considered before moving to the next one. Once all combinations are evaluated, the model with the set of hyperparameters which yield the highest accuracy is the best.

However, the drawbacks outweigh its benefits. It suffers from exponential growth. In our context, our model may have up to **6** types of hyperparameters to consider with each set having **5** values which is **7776** combinations. This shows that the number of evaluations increases exponentially with each additional parameter, due to the curse of dimensionality. Thus, making this impractical technique.

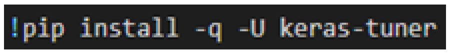
For our project, we aim to search for the optimal hyperparameters for our neural network using the shortest execution time possible. This motivated us to use the Hyperband Tuning algorithm for hyperparameters optimization.

Hyperband (HB) is an hyperparameters optimization strategy that makes use of cheap-to-evaluate approximations of the objective function on smaller budgets (lesser epochs). HB repeatedly calls Successive Halving (SH) to identify the best out of n randomly sampled configurations [1]. In a nutshell, SH evaluates these n number of configurations with a small budget, keeps the best half and doubles their budget.

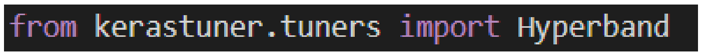
In other words, we give the HB algorithm a maximum number of epochs to run (max budget), and the HB algorithm first evaluates n randomly sampled configurations, with a small portion of the max number of epochs defined, keeps the best half and double their number of epochs to be run as a portion of the max number of epochs defined, and will iteratively do so until we run the best configurations using the max number of epochs defined. Fig 24, shows the visualization of the loss of the model against the percentage of maximum number of epochs defined, where firstly 25 percent of the max epochs are run with n configurations, followed by half of the n configurations running at 50 percent of the max epochs, and finally a quarter of the n configurations running at 100 percent of the max epochs.

7.4.2 Implementation of Hyperband Tuning algorithm

7.4.2.1 Setup



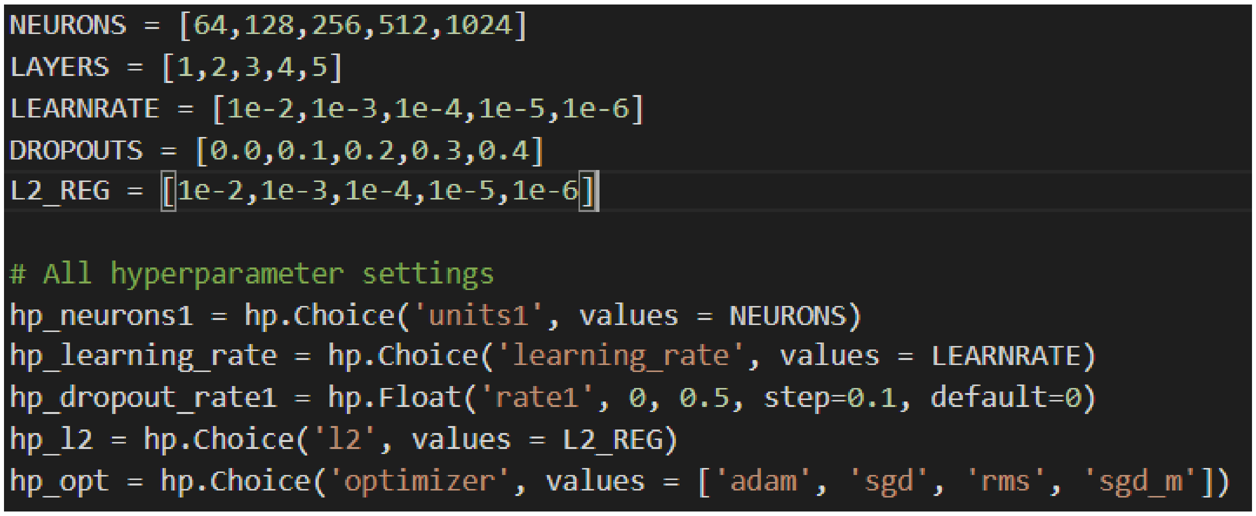
```
!pip install -q -U keras-tuner
```



```
from kerastuner.tuners import Hyperband
```

Fig 25. Imported Libraries

We utilize a Keras library called **keras-tuner** developed by the Keras team. It is a library that allows users to easily perform optimal hyperparameter operations such as the Hyperband technique for TensorFlow neural network models. Before we begin, we installed the latest keras-tuner to our Google Collaboratory and imported its relevant methods such as **Hyperband**.

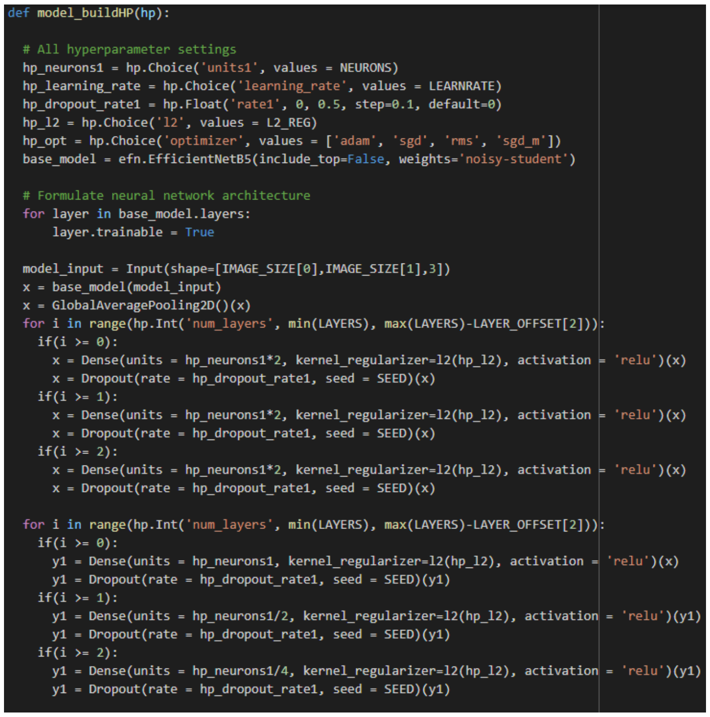


```
NEURONS = [64,128,256,512,1024]
LAYERS = [1,2,3,4,5]
LEARNRATE = [1e-2,1e-3,1e-4,1e-5,1e-6]
DROPOUTS = [0.0,0.1,0.2,0.3,0.4]
L2_REG = [1e-2,1e-3,1e-4,1e-5,1e-6]

# All hyperparameter settings
hp_neurons1 = hp.Choice('units1', values = NEURONS)
hp_learning_rate = hp.Choice('learning_rate', values = LEARNRATE)
hp_dropout_rate1 = hp.Float('rate1', 0, 0.5, step=0.1, default=0)
hp_l2 = hp.Choice('l2', values = L2_REG)
hp_opt = hp.Choice('optimizer', values = ['adam', 'sgd', 'rms', 'sgd_m'])
```

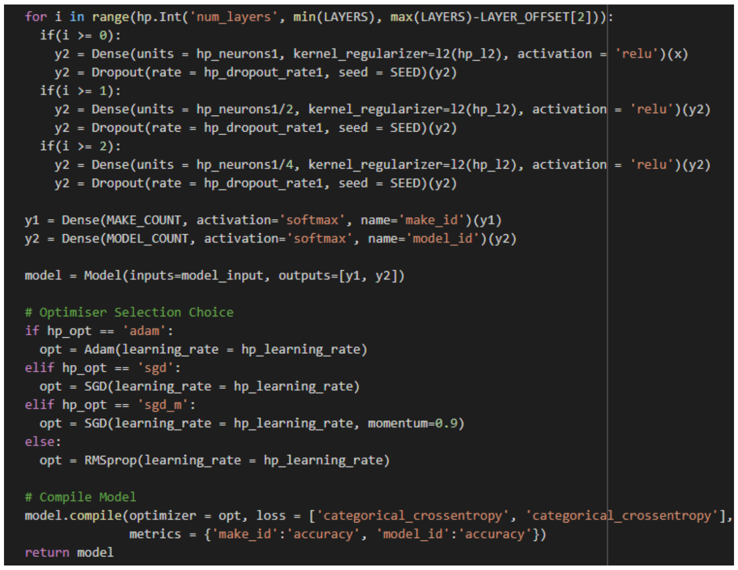
Fig 26. Parameters for Hyperparameters Search

We first define the hyperparameters that we want the Hyperband algorithm to perform the search on by putting each combination of a hyperparameter in a list, as shown in Fig 26. For our group, we will want the Hyperband algorithm to vary the number of neurons in the layers, the number of dense layers used in the model, the L2 weight decay of each Dense layer, dropout rate of each dense layer, the learning rate of the model, as well as the optimizer used by the model.

We then created a function that is like our baseline model's model **build_model** function which supports the optimal tuning process as shown below in Fig 27. Then, we build the model by executing a Hyperband function with the respective arguments as shown in Fig 28. It consists of the model built using the function in Fig 27, the objective metrics to optimize(our group chose validation loss), the number of maximum epochs that the Hyperband algorithm has as its budget, we then select a reduction factor **5** for our Hyperband run, strategy is passed into **distribution_strategy** so that the Hyperband algorithm will utilize the TPU when it is running, a **SEED** value is also passed in so that the results are reproducible, the directory to save the trails and lastly the folder name to store all these process data. Once built, we will have to search for the best hyperparameter configuration by running the **search** method with the respective argument via train function as shown in Fig 29, similar to the arguments used when we call the **model.fit()** method for training our baseline model.

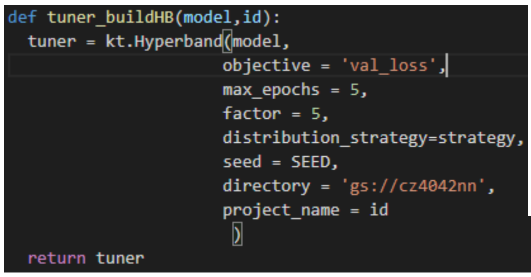


```
def model_buildHP(hp):

  # All hyperparameter settings
  hp_neurons1 = hp.Choice('units1', values = NEURONS)
  hp_learning_rate = hp.Choice('learning_rate', values = LEARNRATE)
  hp_dropout_rate1 = hp.Float('rate1', 0, 0.5, step=0.1, default=0)
  hp_l2 = hp.Choice('l2', values = L2_REG)
  hp_opt = hp.Choice('optimizer', values = ['adam', 'sgd', 'rms', 'sgd_m'])
  base_model = efn.EfficientNetB5(include_top=False, weights='noisy-student')

  # Formulate neural network architecture
  for layer in base_model.layers:
      layer.trainable = True

  model_input = Input(shape=[IMAGE_SIZE[0],IMAGE_SIZE[1],3])
  x = base_model(model_input)
  x = GlobalAveragePooling2D()(x)
  for i in range(hp.Int('num_layers', min(LAYERS), max(LAYERS)-LAYER_OFFSET[2])):
    if(i >= 0):
      x = Dense(units = hp_neurons1*2, kernel_regularizer=l2(hp_l2), activation = 'relu')(x)
      x = Dropout(rate = hp_dropout_rate1, seed = SEED)(x)
    if(i >= 1):
      x = Dense(units = hp_neurons1*2, kernel_regularizer=l2(hp_l2), activation = 'relu')(x)
      x = Dropout(rate = hp_dropout_rate1, seed = SEED)(x)
    if(i >= 2):
      x = Dense(units = hp_neurons1*2, kernel_regularizer=l2(hp_l2), activation = 'relu')(x)
      x = Dropout(rate = hp_dropout_rate1, seed = SEED)(x)

  for i in range(hp.Int('num_layers', min(LAYERS), max(LAYERS)-LAYER_OFFSET[2])):
    if(i >= 0):
      y1 = Dense(units = hp_neurons1, kernel_regularizer=l2(hp_l2), activation = 'relu')(x)
      y1 = Dropout(rate = hp_dropout_rate1, seed = SEED)(y1)
    if(i >= 1):
      y1 = Dense(units = hp_neurons1/2, kernel_regularizer=l2(hp_l2), activation = 'relu')(y1)
      y1 = Dropout(rate = hp_dropout_rate1, seed = SEED)(y1)
    if(i >= 2):
      y1 = Dense(units = hp_neurons1/4, kernel_regularizer=l2(hp_l2), activation = 'relu')(y1)
      y1 = Dropout(rate = hp_dropout_rate1, seed = SEED)(y1)
```

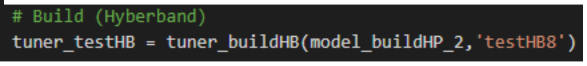


```
  for i in range(hp.Int('num_layers', min(LAYERS), max(LAYERS)-LAYER_OFFSET[2])):
    if(i >= 0):
      y2 = Dense(units = hp_neurons1, kernel_regularizer=l2(hp_l2), activation = 'relu')(x)
      y2 = Dropout(rate = hp_dropout_rate1, seed = SEED)(y2)
    if(i >= 1):
      y2 = Dense(units = hp_neurons1/2, kernel_regularizer=l2(hp_l2), activation = 'relu')(y2)
      y2 = Dropout(rate = hp_dropout_rate1, seed = SEED)(y2)
    if(i >= 2):
      y2 = Dense(units = hp_neurons1/4, kernel_regularizer=l2(hp_l2), activation = 'relu')(y2)
      y2 = Dropout(rate = hp_dropout_rate1, seed = SEED)(y2)

  y1 = Dense(MAKE_COUNT, activation='softmax', name='make_id')(y1)
  y2 = Dense(MODEL_COUNT, activation='softmax', name='model_id')(y2)

  model = Model(inputs=model_input, outputs=[y1, y2])

  # Optimiser Selection Choice
  if hp_opt == 'adam':
    opt = Adam(learning_rate = hp_learning_rate)
  elif hp_opt == 'sgd':
    opt = SGD(learning_rate = hp_learning_rate)
  elif hp_opt == 'sgd_m':
    opt = SGD(learning_rate = hp_learning_rate, momentum=0.9)
  else:
    opt = RMSprop(learning_rate = hp_learning_rate)

  # Compile Model
  model.compile(optimizer = opt, loss = ['categorical_crossentropy', 'categorical_crossentropy'],
                metrics = {'make_id':'accuracy', 'model_id':'accuracy'})
  return model
```

Fig 27. Optimal Hyperparameter Search Model Function

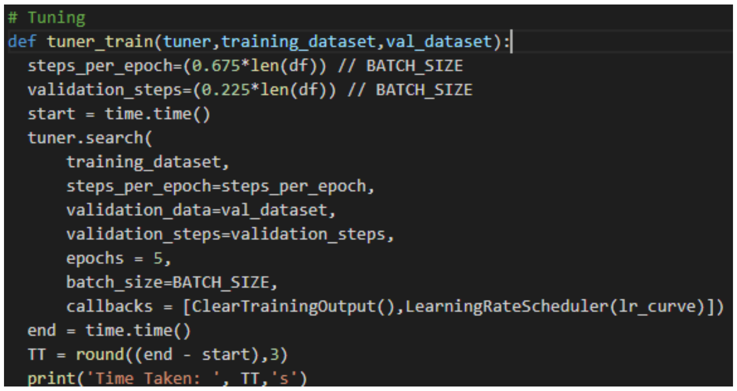


```
def tuner_buildHB(model,id):
  tuner = kt.Hyperband(model,
                       objective = 'val_loss',
                       max_epochs = 5,
                       factor = 5,
                       distribution_strategy=strategy,
                       seed = SEED,
                       directory = 'gs://cz4042nn',
                       project_name = id
                       )
  return tuner
```



```
# Build (Hyberband)
tuner_testHB = tuner_buildHB(model_buildHP_2,'testHB8')
```

Fig 28. Build Model Function with Random Search Techniques (Left), Execute the function (Right)



```
# Tuning
def tuner_train(tuner,training_dataset,val_dataset):
  steps_per_epoch=(0.675*len(df)) // BATCH_SIZE
  validation_steps=(0.225*len(df)) // BATCH_SIZE
  start = time.time()
  tuner.search(
      training_dataset,
      steps_per_epoch=steps_per_epoch,
      validation_data=val_dataset,
      validation_steps=validation_steps,
      epochs = 5,
      batch_size=BATCH_SIZE,
      callbacks = [ClearTrainingOutput(),LearningRateScheduler(lr_curve)])
  end = time.time()
  TT = round((end - start),3)
  print('Time Taken: ', TT,'s')
```

Fig 29. Train Model Function

7.4.3 Running of the Hyperband Tuning algorithm

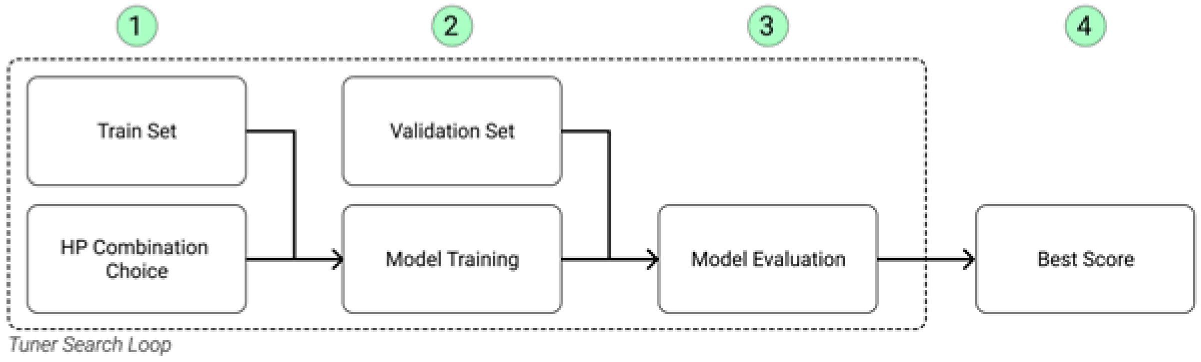
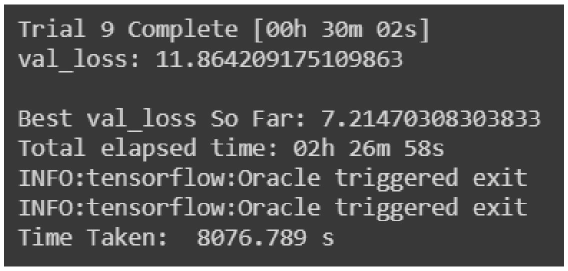


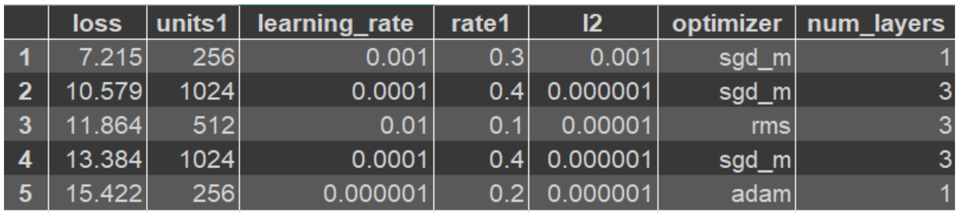
Fig 30. Hyperparameter tuning process with Keras Tuner

With the functions defined for the Hyperband algorithm, we then run the **tuner_train** method which will start the Hyperband algorithm. The algorithm will begin by running a training process for each of the Hyperparameter Combination Choice as shown in Fig 30.

The training process will generate a trial summary result as shown Fig 31 (top), after which, the training process is repeated by the algorithm until all searches are exhausted, then it will generate the top 5 trials as shown in Fig 31 (bottom).



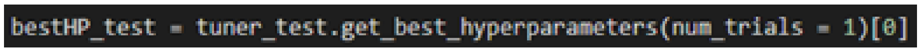
```
Trial 9 Complete [00h 30m 02s]
val_loss: 11.864209175109863

Best val_loss So Far: 7.21470308303833
Total elapsed time: 02h 26m 58s
INFO:tensorflow:Oracle triggered exit
INFO:tensorflow:Oracle triggered exit
Time Taken:  8076.789 s
```



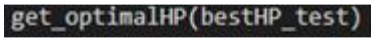
	loss	units1	learning_rate	rate1	l2	optimizer	num_layers
1	7.215	256	0.001	0.3	0.001	sgd_m	1
2	10.579	1024	0.0001	0.4	0.000001	sgd_m	3
3	11.864	512	0.01	0.1	0.00001	rms	3
4	13.384	1024	0.0001	0.4	0.000001	sgd_m	3
5	15.422	256	0.000001	0.2	0.000001	adam	1

Fig 31. A completed summary trials (top) and top list of hyperparameters (bottom)

Once the training is over, we re-run the Hyperband build function as shown in Fig 28 (Right), to retrieve all the trials summary. After which the best optimal hypermeter is determined by the best score amongst all the trials by using the code as shown in Fig 32 (Top). Lastly, display the output using this function in Fig 32 (Bottom).



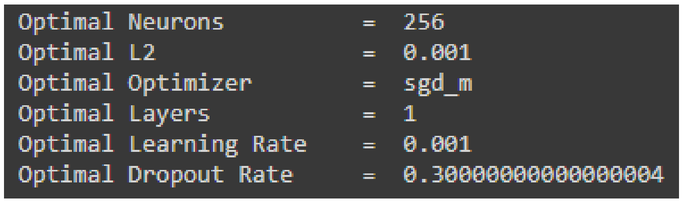
```
bestHP_test = tuner_test.get_best_hyperparameters(num_trials = 1)[0]
```



```
get_optimalHP(bestHP_test)
```

Fig 32. Retrieved (Top) and display (Bottom) optimal hyperparameter

7.4.3 Optimal Hyperparameter Result

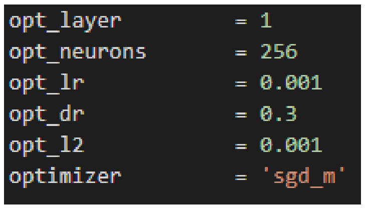


```
Optimal Neurons          =  256
Optimal L2               =  0.001
Optimal Optimizer        =  sgd_m
Optimal Layers           =  1
Optimal Learning Rate    =  0.001
Optimal Dropout Rate     =  0.30000000000000004
```

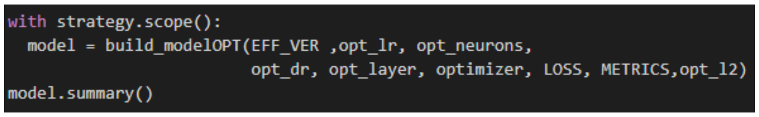
Fig 33. optimal hyperparameters respective values

After running the Hyperband algorithm, we obtain the optimal hyperparameters for our car image categorisation neural network as shown in Fig 33 above.

7.4.4 Building the Neural Network with Optimal Parameters

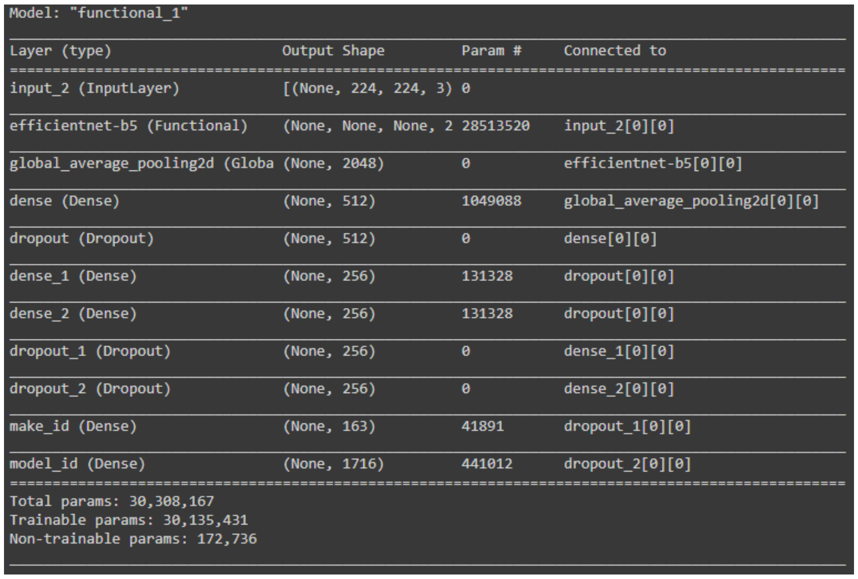


```
opt_layer          = 1
opt_neurons        = 256
opt_lr             = 0.001
opt_dr             = 0.3
opt_l2             = 0.001
optimizer          = 'sgd_m'
```



```
with strategy.scope():
  model = build_modelOPT(EFF_VER ,opt_lr, opt_neurons,
                         opt_dr, opt_layer, optimizer, LOSS, METRICS,opt_l2)
model.summary()
```

<u>Fig 34. New declaration of hyperparameter (left) and re-run the baseline model (right)</u>

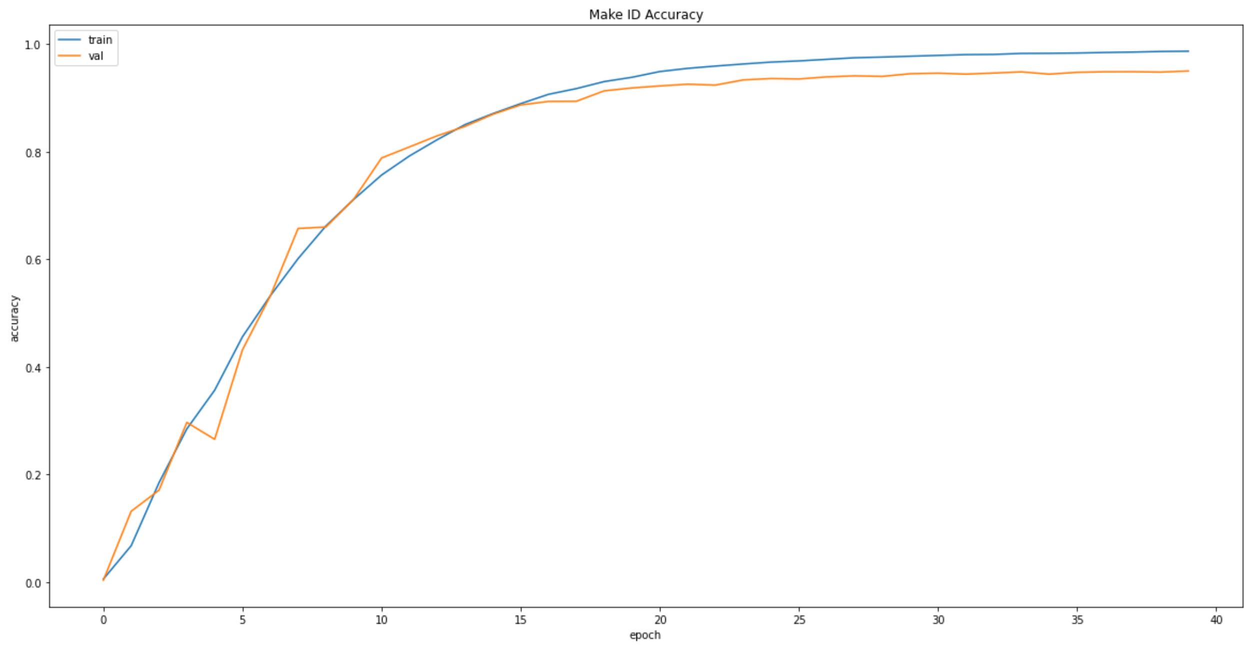


```
Model: "functional_1"
________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================================
input_2 (InputLayer)            [(None, 224, 224, 3) 0
________________________________________________________________________________________________
efficientnet-b5 (Functional)    (None, None, None, 2 28513520    input_2[0][0]
________________________________________________________________________________________________
global_average_pooling2d (Globa (None, 2048)         0           efficientnet-b5[0][0]
________________________________________________________________________________________________
dense (Dense)                   (None, 512)          1049088     global_average_pooling2d[0][0]
________________________________________________________________________________________________
dropout (Dropout)               (None, 512)          0           dense[0][0]
________________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          131328      dropout[0][0]
________________________________________________________________________________________________
dense_2 (Dense)                 (None, 256)          131328      dropout[0][0]
________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 256)          0           dense_1[0][0]
________________________________________________________________________________________________
dropout_2 (Dropout)             (None, 256)          0           dense_2[0][0]
________________________________________________________________________________________________
make_id (Dense)                 (None, 163)          41891       dropout_1[0][0]
________________________________________________________________________________________________
model_id (Dense)                (None, 1716)         441012      dropout_2[0][0]
================================================================================================
Total params: 30,308,167
Trainable params: 30,135,431
Non-trainable params: 172,736
________________________________________________________________________________________________
```

<u>Fig 35. Summary of the new baseline model with optimal hyperparameters</u>.

After finding the optimal hyperparameters using the Hyperband algorithm, we change the hyperparameters for our baseline model based on the hyperparameter values shown in Fig 33. Re-declare and re-run the model as shown in Fig 34. The summary of the new baseline model as shown in Fig 35. We then run the training for the new model with the optimal parameters, to observe the effects it has on the validation loss and accuracy of the model.

7.4.5 Baseline Model with Optimal Hyperparameter Evaluation



<u>Figure 36. The optimal baseline model Train and Validation 'Make' Accuracy graph</u>

From the figure above, we can see the plot of the training accuracy and validation accuracy vs. epochs for classifying the images into car make IDs. Our model with Optimal Hyperparameter is performing impressively well, where we can achieve a 0.9493 validation accuracy and 0.9861 training accuracy at the last epoch. From the graph we can also see that the model is slightly overfitting on the training data towards the last 25 epochs, with the validation accuracy lower than that of the training accuracy. The model converges at around 35 epochs for learning the classification of the images into car makes.

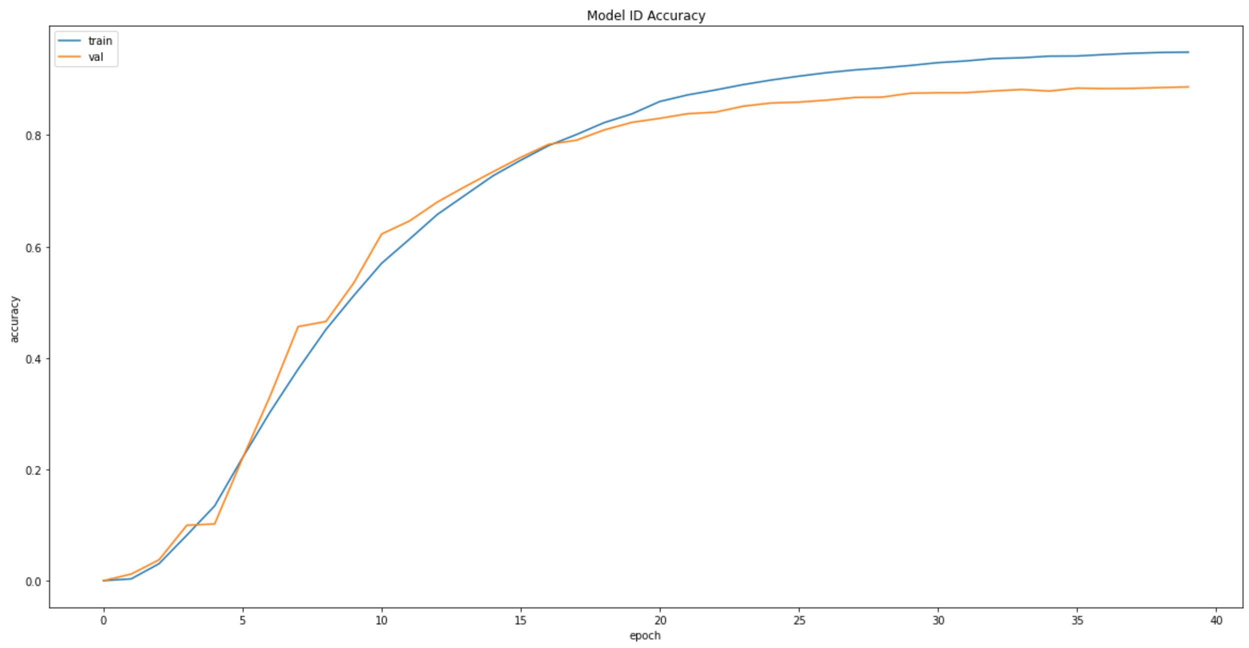


Figure 37. The optimal baseline model Train and Validation ‘Model’ Accuracy graph

From the figure above, we can see the plot of the training accuracy and validation accuracy vs. epochs for classifying the images into car model IDs. Our model with Optimal Hyperparameter is performing decently, where we are able to achieve a 0.8863 validation accuracy and 0.9486 training accuracy at the last epoch. From the graph we can also see that the model is slightly overfitting on the training data towards the last 20 epochs, with the validation accuracy lower than that of the training accuracy. The model converges at around 30 epochs for learning the classification of the images into car models.

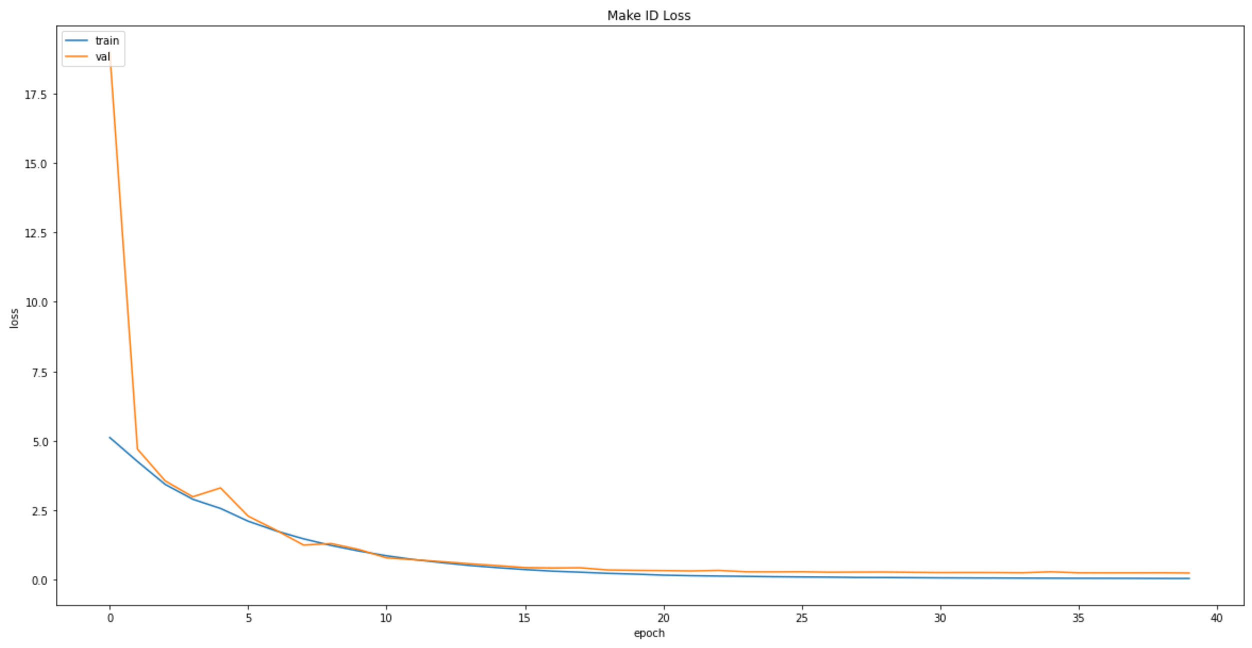


Figure 38. The optimal baseline model Train and Validation ‘Make’ Loss graph

From the figure above, we can see the plot of the training loss and validation loss vs. epochs for classifying the images into car make IDs. Our model with Optimal Hyperparameter is performing impressively well, where we can achieve a 0.2473 validation loss and 0.0493 training loss at the last epoch. The model converges at around 35 epochs for learning the classification of the images into car makes.

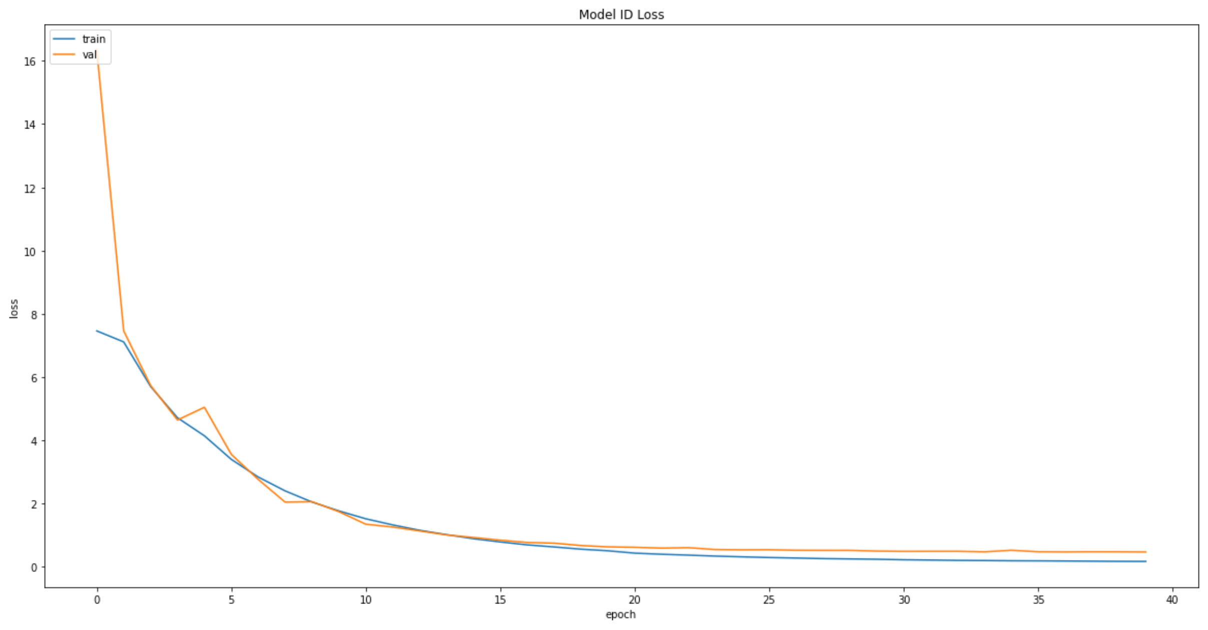
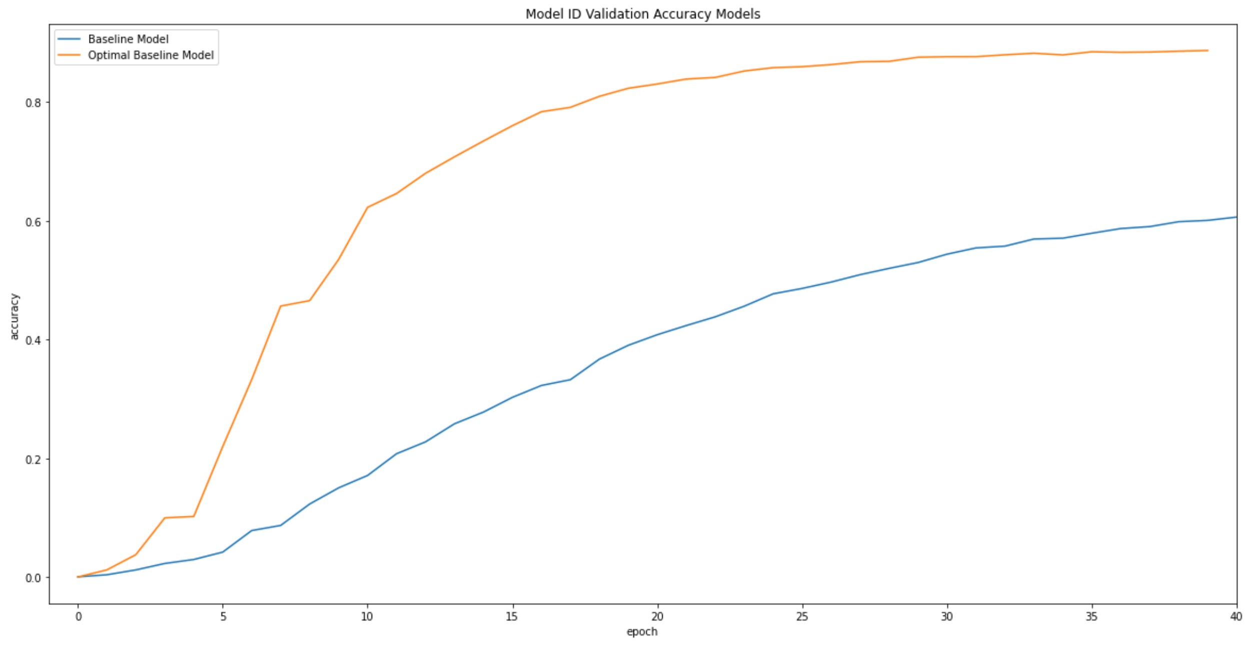
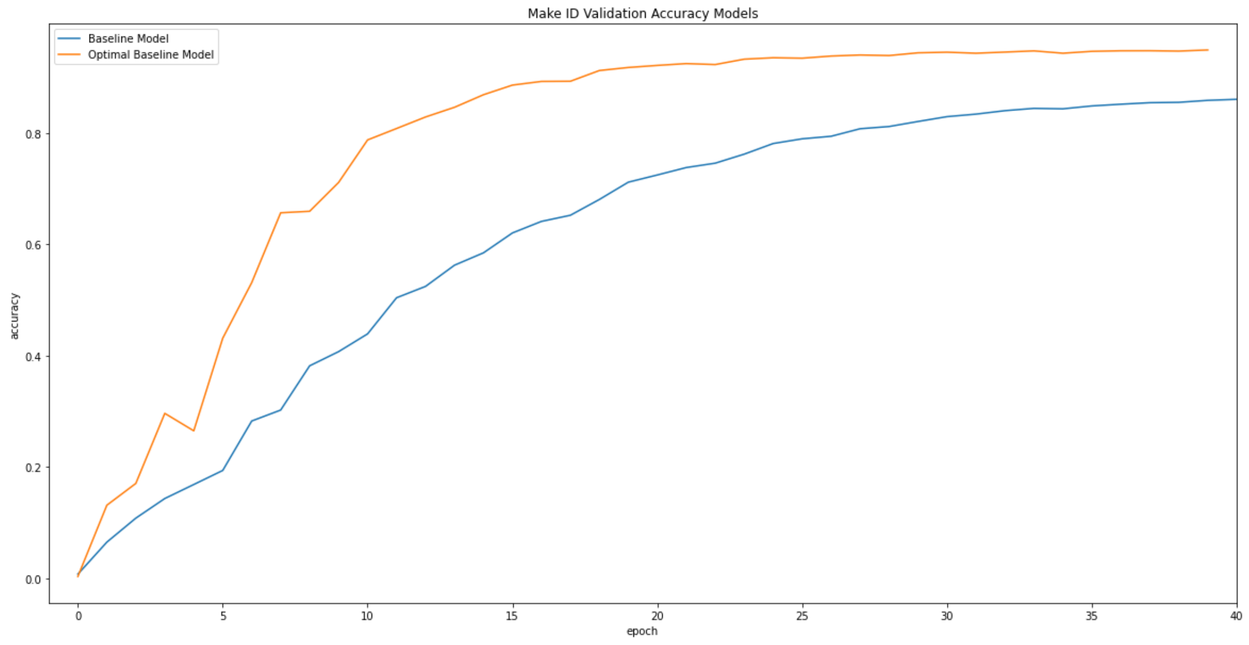


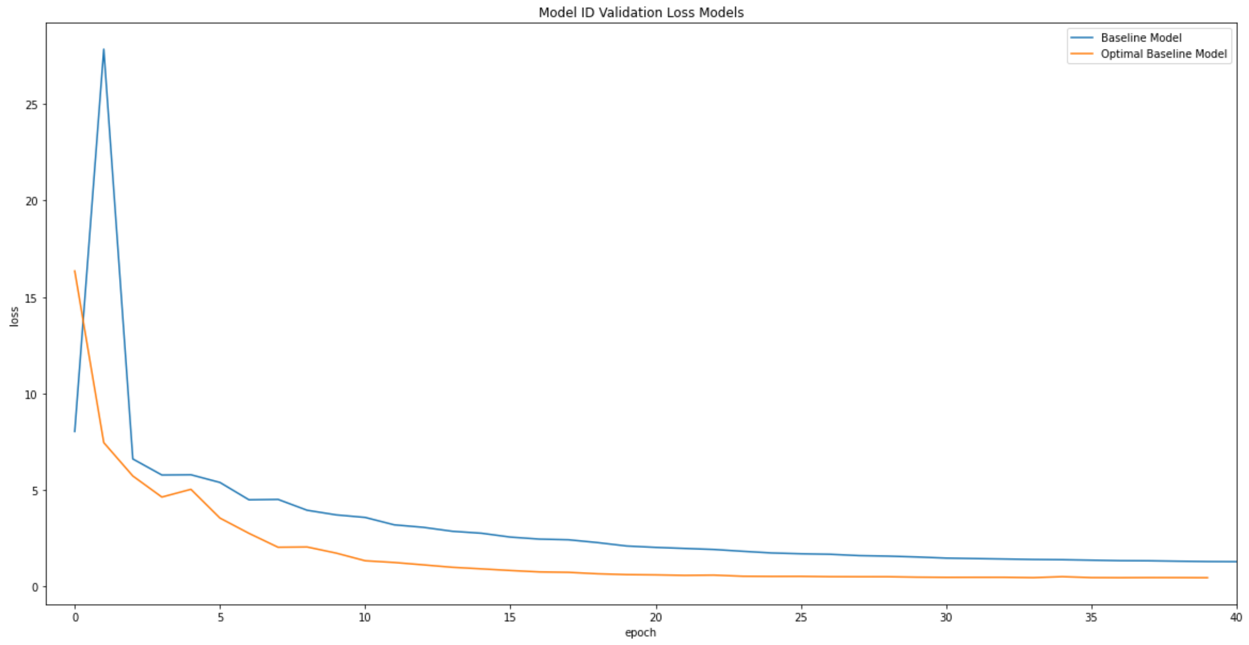
Figure 39. The optimal baseline model Train and Validation ‘Model’ Loss graph

From the figure above, we can see the plot of the training loss and validation loss vs. epochs for classifying the images into car model IDs. Our model with Optimal Hyperparameter is performing decently, where we are able to achieve a 0.4584 validation loss and 0.1595 training loss at the last epoch. The model converges at around 30 epochs for learning the classification of the images into car models.

7.4.6 Baseline Model vs Baseline Model with Optimal Hyperparameters







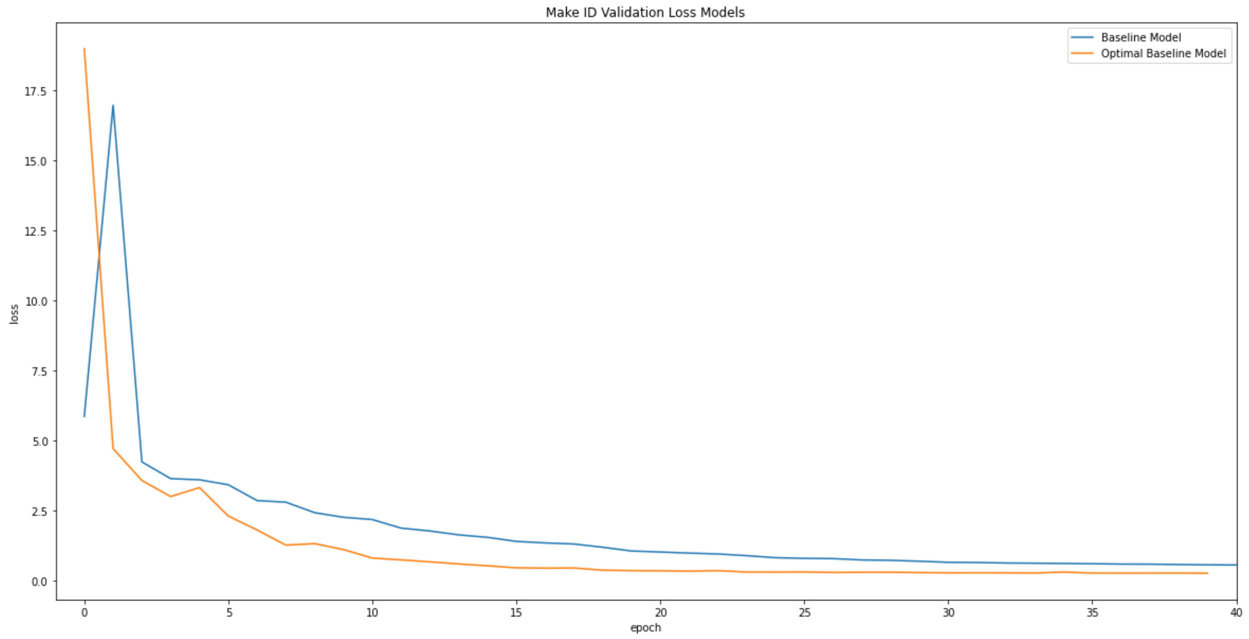


Fig 40. Comparison between different Baseline Model's accuracy and loss vs Baseline Model with Optimal Hyperparameter graph.

We can evidently see that in Fig 40. The baseline model with optimal hyperparameter has significantly improved in performance. For example, the baseline model with optimal hyperparameter manages to reach around 0.8588 makeIDs' validation accuracy in approximately 15th epoch as compared to baseline model 40th epoch. Besides that, the baseline model with optimal hyperparameter has managed to achieve higher validation accuracy and lower validation loss as compared to baseline model. This can be shown in the table below.

S/N	Model	Train					Validation				
		Loss			Accuracy		Loss			Accuracy	
		Overall	Make	Model	Make	Model	Overall	Make	Model	Make	Model
1	**BL**	2.1224	0.5344	1.5881	0.8395	0.4850	1.8434	0.5479	1.2960	0.8588	0.6005
2	**BLO**	0.3735	0.0493	0.1595	0.9861	0.9486	0.8693	0.2473	0.4584	0.9493	0.8863

Legend: BL (Baseline), BLO (Baseline with Optimal Hyperparameter)

Fig 41. Table of metrics values at the end of 40th epoch

This goes to show that by utilizing an optimal hyperparameter combination, a neural network is capable of achieving better accuracy and efficiency in learning the dataset. This also tells us that the Hyperband tuning algorithm that we used is capable of selecting optimal hyperparameters as intended, in the shortest time possible without having to try all the combinations of hyperparameters.

7.5 Deep Neural Network with Optimal Parameters and Focal Loss Function

7.5.1 Using Focal Loss Advanced Loss Function

After obtaining the optimal parameters for our Deep Neural Network for categorization of car images, we then went on to try a more advanced loss function for the neural network. For our project, we have decided to go with the Focal loss function, as the advanced loss function for our neural network.

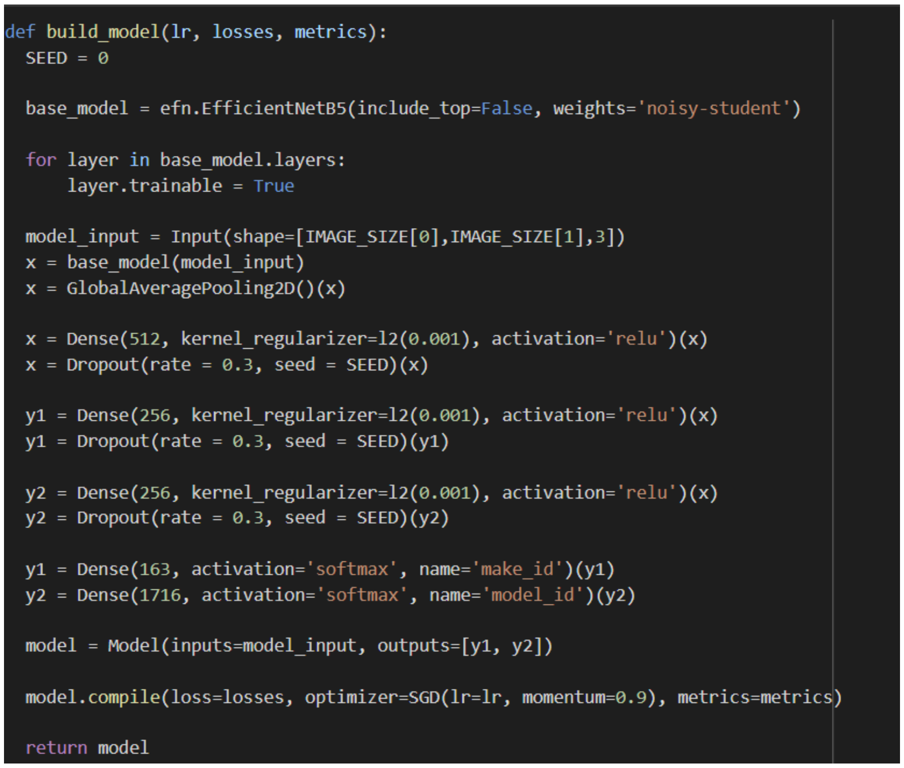
The reason for utilizing the Focal Loss advanced loss function is due to the fact that we were able to get a decent validation accuracy for the classification of car makes using our neural network with the **categorical_crossentropy** loss function, however, we want to find out if the validation accuracy for the classification of car models can be improved further.

The Focal Loss function is extremely useful for classification when you have highly imbalanced classes. It down-weights well-classified examples and focuses on hard examples. The loss value is much higher for a sample which is misclassified by the classifier as compared to the loss value corresponding to a well-classified example. Therefore, for our project, we would like to determine if the Focal Loss function is able to increase the validation accuracy for the classification of car models, where the classes for car model ID are highly imbalanced.

7.5.2 Building Model with Function Focal Loss Advanced Loss Function

For the neural network with the Focal Loss function, we will be using the optimal parameters derived from the Hyperband algorithm when building the model. The model will consist of:

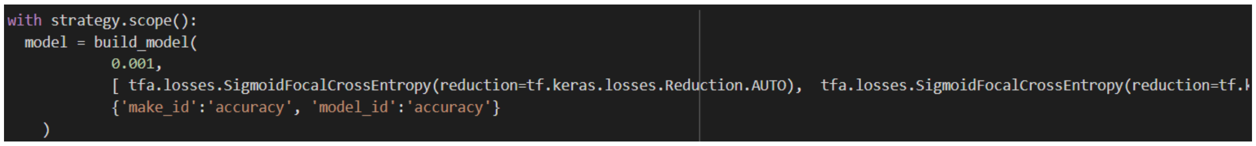
- Base EfficientNetB5 Convolutional Neural Network
- Global Average Pooling Layer
- Common Dense Layer with 512 neurons, L2 regularization of 0.001, ‘relu’ activation
- Dropout layer of 0.3 dropout rate
- Car Make Classification Tail: Dense Layer with 256 neurons, L2 regularization of 0.001, ‘relu’ activation, Dropout layer of 0.3 dropout rate, output Dense layer with 163 neurons and ‘softmax’ activation
- Car Model Classification Tail: Dense Layer with 256 neurons, L2 regularization of 0.001, ‘relu’ activation, Dropout layer of 0.3 dropout rate, output Dense layer with 1716 neurons and ‘softmax’ activation



```python
def build_model(lr, losses, metrics):
  SEED = 0

  base_model = efn.EfficientNetB5(include_top=False, weights='noisy-student')

  for layer in base_model.layers:
      layer.trainable = True

  model_input = Input(shape=[IMAGE_SIZE[0],IMAGE_SIZE[1],3])
  x = base_model(model_input)
  x = GlobalAveragePooling2D()(x)

  x = Dense(512, kernel_regularizer=l2(0.001), activation='relu')(x)
  x = Dropout(rate = 0.3, seed = SEED)(x)

  y1 = Dense(256, kernel_regularizer=l2(0.001), activation='relu')(x)
  y1 = Dropout(rate = 0.3, seed = SEED)(y1)

  y2 = Dense(256, kernel_regularizer=l2(0.001), activation='relu')(x)
  y2 = Dropout(rate = 0.3, seed = SEED)(y2)

  y1 = Dense(163, activation='softmax', name='make_id')(y1)
  y2 = Dense(1716, activation='softmax', name='model_id')(y2)

  model = Model(inputs=model_input, outputs=[y1, y2])

  model.compile(loss=losses, optimizer=SGD(lr=lr, momentum=0.9), metrics=metrics)

  return model
```

Fig 42. Code for building and compiling Neural Network Model

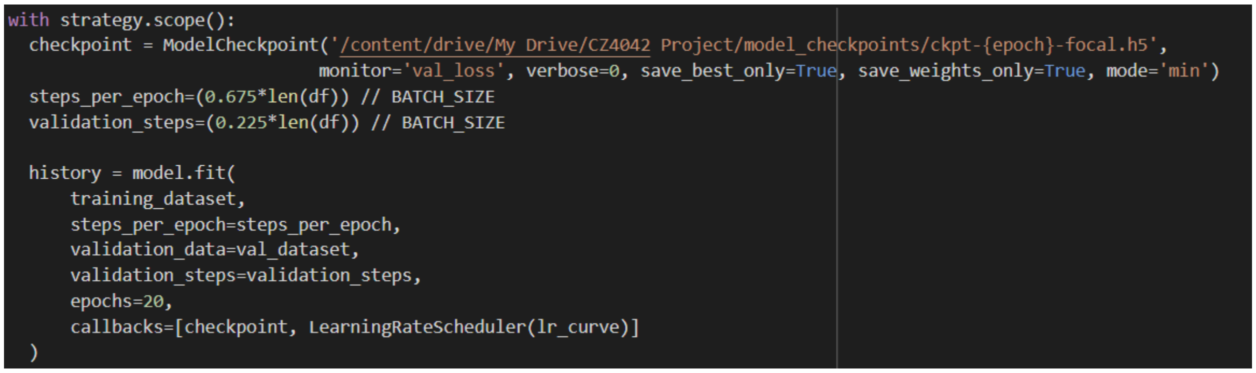
The figure above shows the function that is used to build the model that we have defined with the optimal parameters.



```python
with strategy.scope():
  model = build_model(
            0.001,
            [ tfa.losses.SigmoidFocalCrossEntropy(reduction=tf.keras.losses.Reduction.AUTO),  tfa.losses.SigmoidFocalCrossEntropy(reduction=tf.k
            {'make_id':'accuracy', 'model_id':'accuracy'}
    )
```

Fig 43. Building Deep Neural Network with Focal Loss

The figure above shows the code used to build the Deep Neural Network with the Focal Loss advanced loss function for the Car Categorization task. We build the model using **with strategy.scope()** to ensure that our neural network is using the TPU hardware accelerator when training. We call the **build_model** function with the parameters **0.001** as the initial learning rate, and the loss function for the model will be **tfa.losses.SigmoidFocalCrossEntropy** and the 2 metrics to judge the performance of our model is the **accuracy** for classifying the make_id and model_id labels.



```
with strategy.scope():
  checkpoint = ModelCheckpoint('/content/drive/My Drive/CZ4042 Project/model_checkpoints/ckpt-{epoch}-focal.h5',
                               monitor='val_loss', verbose=0, save_best_only=True, save_weights_only=True, mode='min')
  steps_per_epoch=(0.675*len(df)) // BATCH_SIZE
  validation_steps=(0.225*len(df)) // BATCH_SIZE

  history = model.fit(
      training_dataset,
      steps_per_epoch=steps_per_epoch,
      validation_data=val_dataset,
      validation_steps=validation_steps,
      epochs=20,
      callbacks=[checkpoint, LearningRateScheduler(lr_curve)]
  )
```

Fig 44. Fitting the Neural Network

Next, we train the model by calling the **model.fit** function. The model is trained using the training dataset that we have prepared and validated on the validation dataset that we have prepared. We also checkpoint the model if the model reduces its validation loss from the last best validation loss from previous epochs.

As the connection to the TPU in Google Collaboratory cuts off after roughly training 20 epochs, we train the model in batches of 20 epochs, using the saved model weights from the last batch until we have trained the model for a total of 40 epochs.

Also, since we are training the model batches of 20 epochs, we will save the model's training history in CSV files which will be combined together in one CSV file after running 40 epochs to allow us to plot the loss and accuracy graphs for the neural network.

7.5.3 Experiment Results

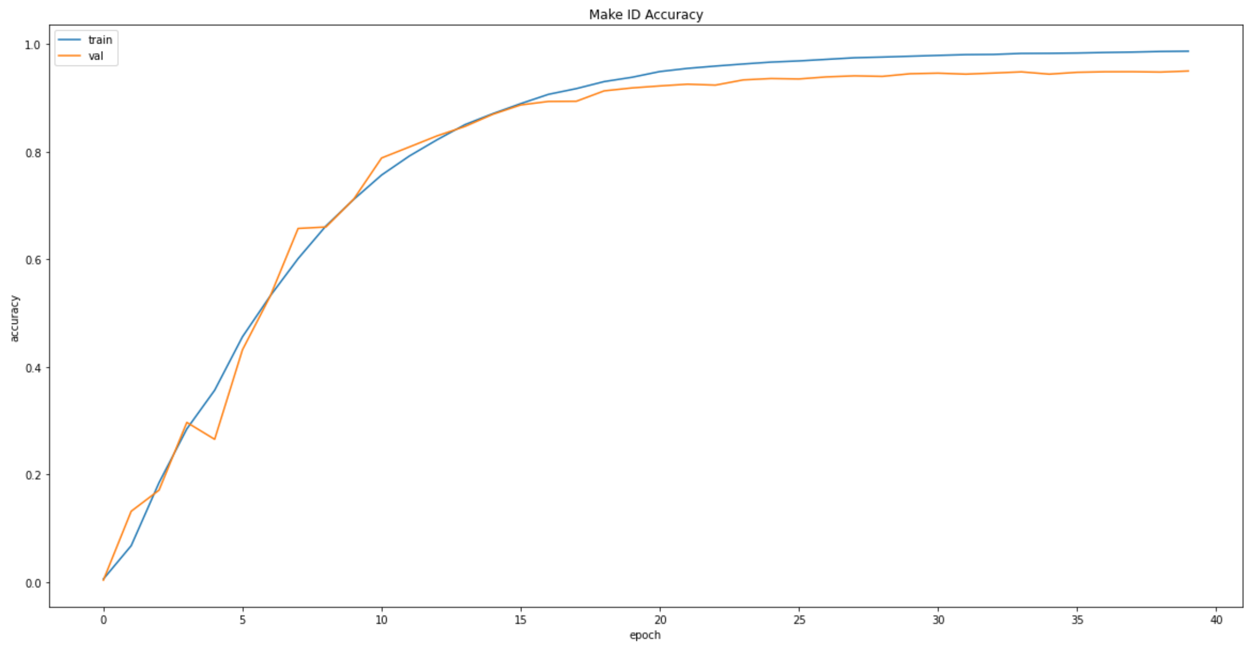


Fig 45. Make ID Accuracy vs. Epochs

From the figure above, we can see the plot of the training accuracy and validation accuracy vs. epochs for classifying the images into car make IDs. Our model with Focal Loss is performing decently, where we are able to achieve a 0.9152 validation accuracy and 0.9406 training accuracy at the last epoch. From the graph we can also see that the model is slightly overfitting on the training data towards the last 20 epochs, with the validation accuracy lower than that of the training accuracy. The model converges at around 30 epochs for learning the classification of the images into car makes. We can also see that the model with the Focal Loss function converges faster than the baseline model.

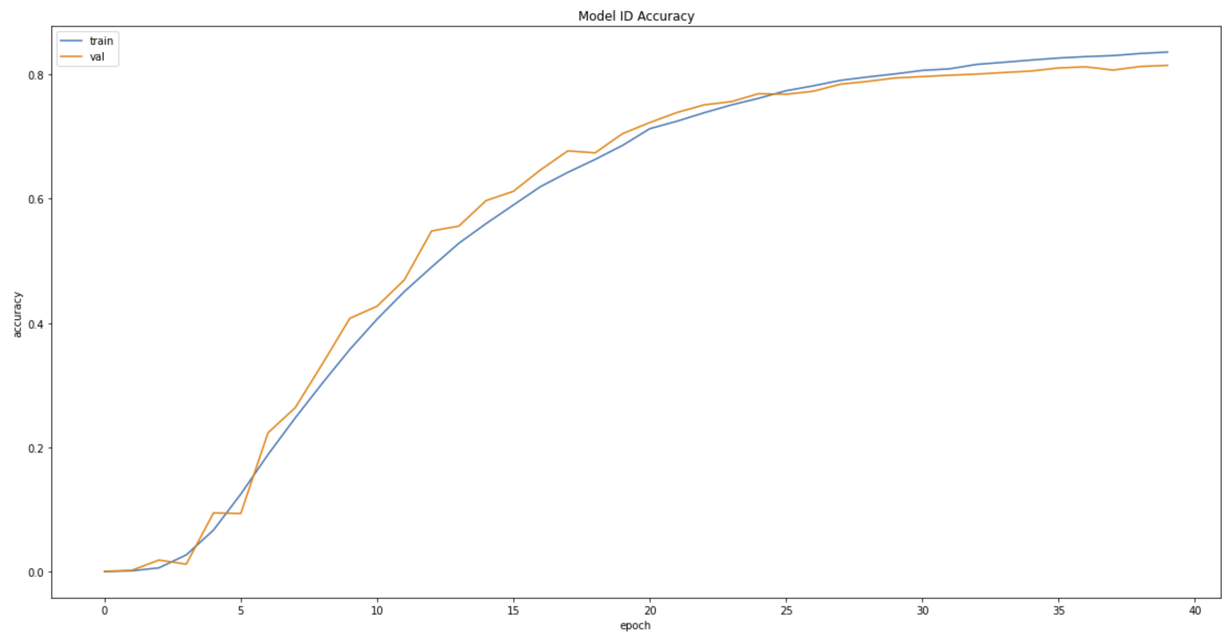


Fig 46. Model ID Accuracy vs. Epochs

From the figure above, we can see the plot of the training accuracy and validation accuracy vs. epochs for classifying the images into car model IDs. Our model with Focal Loss is performing well, where we are able to achieve a 0.8146 validation accuracy and 0.8360 training accuracy at the last epoch. From the graph we can also see that the model is slightly overfitting on the training data towards the last 15 epochs, with the validation accuracy lower than that of the training accuracy. The model starts to converge at around 40 epochs for learning the classification of the images into car models.

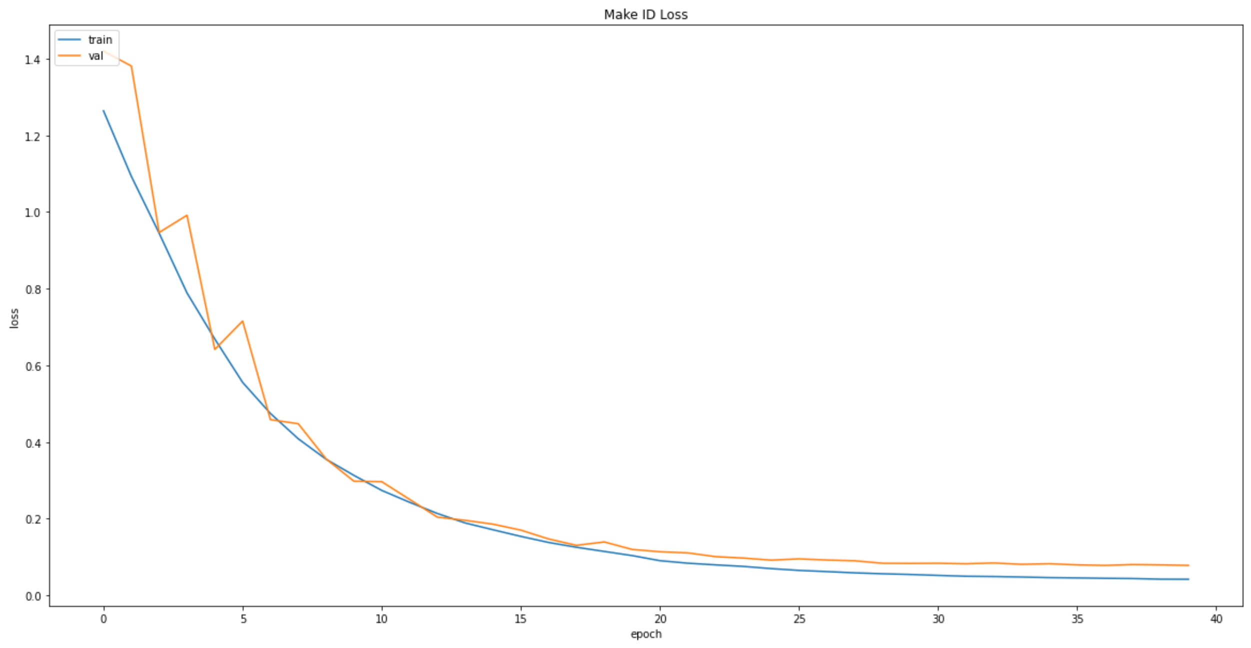


Fig 47. Make ID Loss vs. Epochs

The figure above shows the plot of the training loss and validation loss vs. epochs for classifying the images into car make IDs. From the plot, we can observe that the model is learning well with both the validation and training loss decreasing with each epoch and converging at around 30 epochs.

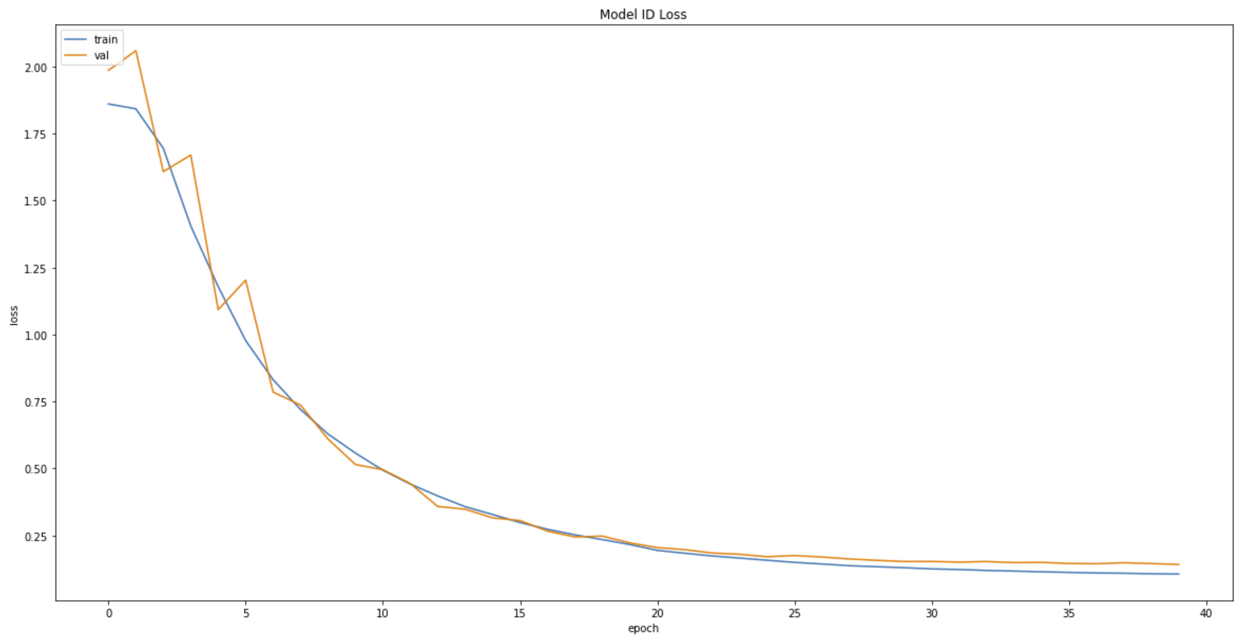


Fig 48. Model ID Loss vs. Epochs

The figure above shows the plot of the training loss and validation loss vs. epochs for classifying the images into car model IDs. From the plot, we can observe that the model is learning well with both the validation and training loss decreasing with each epoch and both losses converge at around 40 epochs.

8. Discussions

8.1 Comparing the Experimental Results of the Trained Models

After completing all the tasks defined in the project scope, we then wanted to compare the validation accuracies and losses of the 3 models that we have trained:

1. Baseline Model
2. Baseline Model with Optimal Hyperparameters
3. Model with Optimal Hyperparameters and Focal Loss Function

The table below shows the values of the various metrics of the 3 models at the end of 40 epoch of training.

Model	**Train**					**Validation**				
	Loss			**Accuracy**		**Loss**			**Accuracy**	
	Overall	Car Make	Car Model	Car Make	Car Mode	Overall	Car Make	Car Model	Car Make	Car Model
BL	2.1224	0.5344	1.5881	0.8395	0.4850	1.8434	0.5479	1.2960	0.8588	0.6005
BLO	0.3735	0.0493	0.1595	0.9861	0.9486	0.8693	0.2473	0.4584	0.9493	0.8863
FL	0.2259	0.0422	0.1061	0.9406	0.8360	0.2979	0.0784	0.1420	0.9152	0.8146

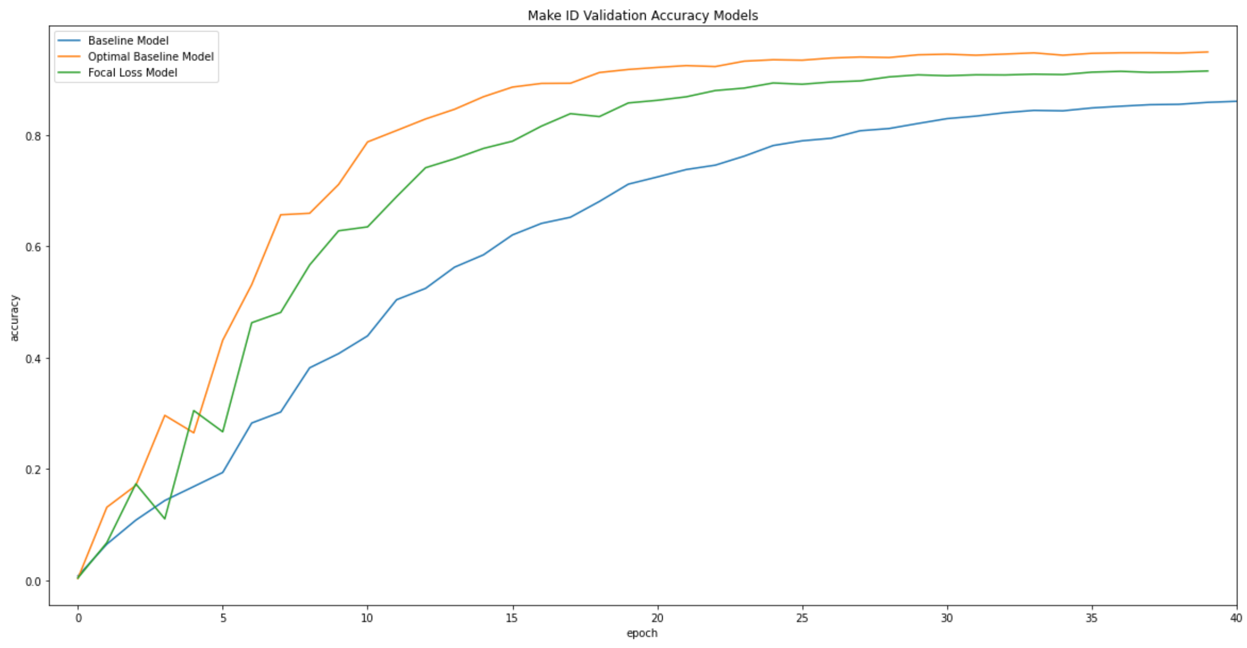
Legend: **BL** (Baseline), **BLO** (Baseline with Optimal Hyperparameters), **FL** (Focal Loss Function, Optimal Hyperparameters)

Fig 49. Table of metrics values at the end of 40th epoch for all the models

From Fig 49, we can see that the Baseline Model with Optimal Hyperparameters has the best validation accuracy for both classification of Car Make ID and Car Model ID among the 3 trained models, with a 94.93% validation accuracy for classification of Car Make ID and 88.63% accuracy for classification of Car Model ID. Also, the Model with Optimal Hyperparameters and Focal Loss Function has the best validation loss for both classification of Car Make ID and Car Model ID among the 3 trained models.

From the results that we have obtained, we can also observe that the model with the Focal Loss Advanced Loss Function did not fare better than the Baseline Model with Optimal Hyperparameters and SGD Loss Function in terms of validation accuracies for both classification of Car Make ID and Car Model ID.

In order to help us better visualize the experimental results that we have obtained, we also plotted the graphs of the various metrics against each training epoch. The graphs will help us better visualize how well each model learned during training of the dataset.



<u>Fig 51. Make ID Accuracy vs. Epochs</u>

In Fig 51 above, we have the plot of the validation accuracies of classifying Car Make ID for the 3 models that we have trained. From the plot, we can see that the Baseline Model with Optimal Hyperparameters has the highest validation accuracy throughout most of the epochs of training, amongst the 3 models, with the Model with Optimal Hyperparameters and Focal Loss Function having the second highest validation accuracy. We can also see from the graph that the Baseline Model with Optimal Hyperparameters, as well as the Model with Optimal Hyperparameters and Focal Loss Function learn faster than the baseline model, with their validation accuracies converging at around 25 epochs and the Baseline model validation accuracy starting to converge at around 40 epochs.

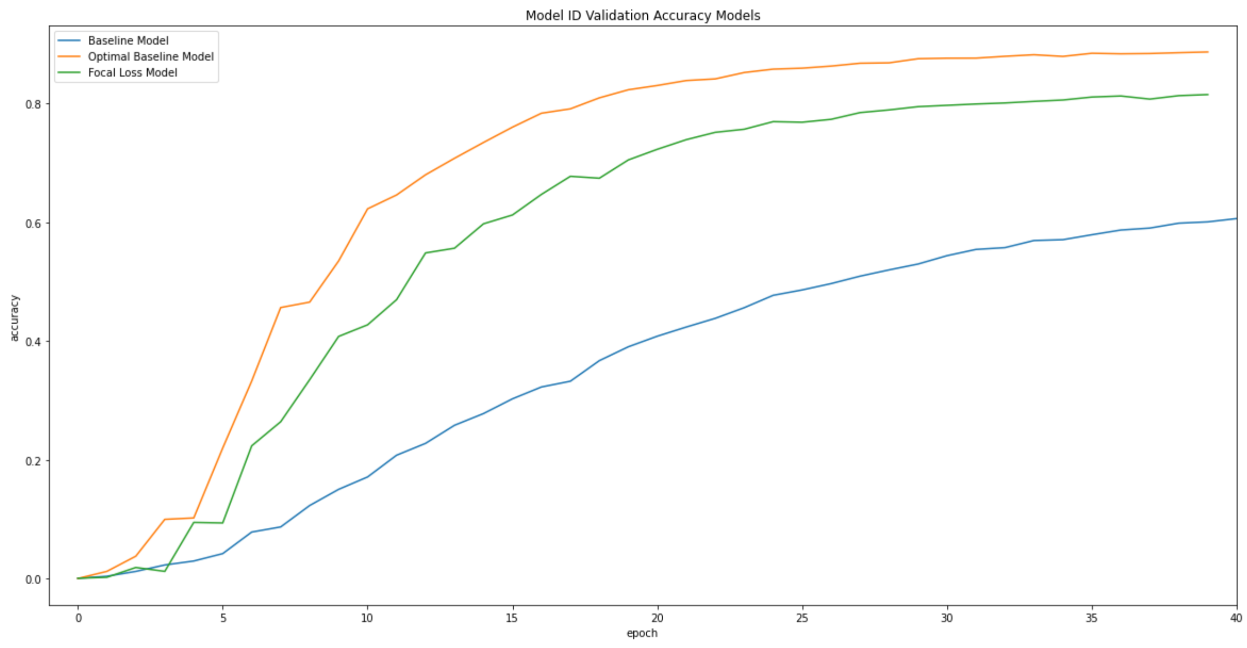
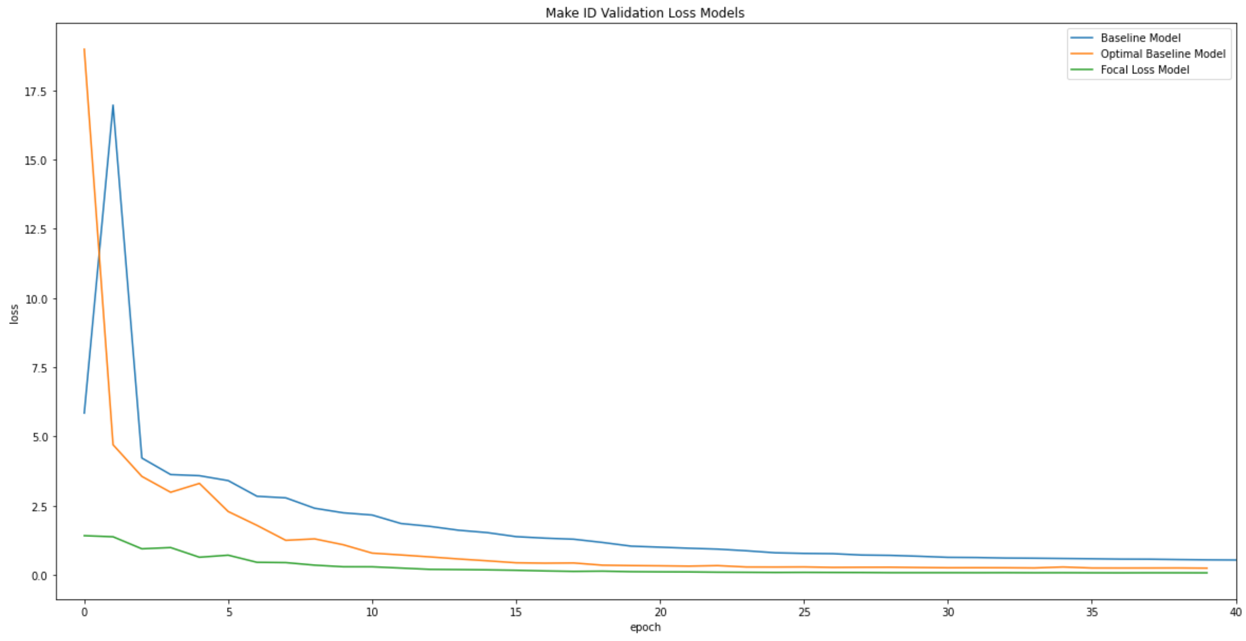


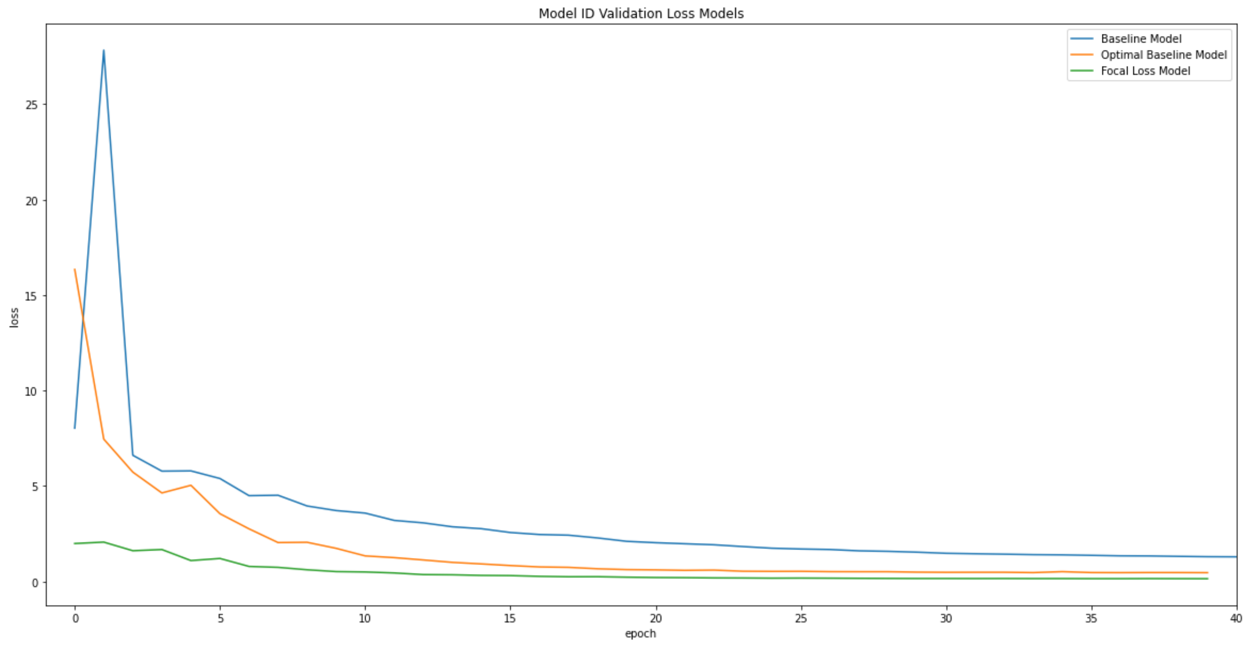
Fig 52. Model ID Accuracy vs. Epochs

In Fig 52 above, we have the plot of the validation accuracies of classifying Car Model ID for the 3 models that we have trained. From the plot, we can see that the Baseline Model with Optimal Hyperparameters has the highest validation accuracy throughout most of the epochs of training, amongst the 3 models, with the Model with Optimal Hyperparameters and Focal Loss Function having the second highest validation accuracy. We can also see from the graph that the Baseline Model with Optimal Hyperparameters, as well as the Model with Optimal Hyperparameters and Focal Loss Function learn faster than the baseline model, with their validation accuracies converging towards 40 epochs, while the Baseline model validation accuracy is still not converging at 40 epochs and is 20% than that of the other 2 models.



<u>Fig 53. Make ID Loss vs. Epochs</u>

In Fig 53 above, we have the plot of the validation losses of classifying Car Make ID for the 3 models that we have trained. From the plot, we can see that the Model with Optimal Hyperparameters and Focal Loss Function have the lowest validation loss amongst the 3 models for all the training epochs, with the Baseline Model with Optimal Hyperparameters having the second lowest validation loss. This goes to show that the Model with Optimal Hyperparameters and Focal Loss Function learns the dataset well from the start till the end of the training.

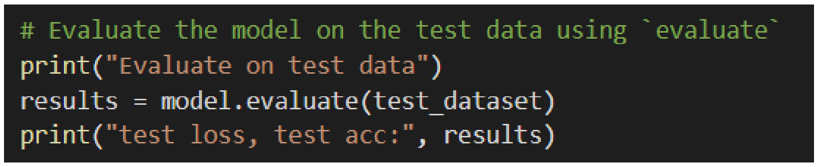


<u>Fig 54. Model ID Loss vs. Epochs</u>

In Fig 54 above, we have the plot of the validation losses of classifying Car Model ID for the 3 models that we have trained. From the plot, we can see that the Model with Optimal Hyperparameters and Focal Loss Function have the lowest validation loss amongst the 3 models for all the training epochs, with the Baseline Model with Optimal Hyperparameters having the second lowest validation loss. This goes to show that the Model with Optimal Hyperparameters and Focal Loss Function learns the dataset well from the start till the end of the training.

8.2 Comparing the Test Accuracy of the Trained Models

After we have trained the 3 models, we then assess the performance of the trained models using the test data. The test dataset will allow us to determine how well the trained models categorize car images from an unseen dataset after learning the model on the training dataset.



```
# Evaluate the model on the test data using `evaluate`
print("Evaluate on test data")
results = model.evaluate(test_dataset)
print("test loss, test acc:", results)
```

Fig 55. model.evaluate() function

As seen from the figure above, we used the **model.evaluate()** function and we input the test dataset to the 3 trained models to determine their test accuracies and test losses.

8.3 Prediction Results

S/N	**Model**	**Loss**			**Accuracy**	
		Overall	Make IDs	Model IDs	Make IDs	Model IDs
1	**Baseline**	1.9821	0.5410	1.4412	0.8730	0.6408
2	**Baseline w/ Optimal HP**	1.2533	0.3154	0.7742	**0.9451**	**0.8828**
3	**Focal Loss**	**0.3584**	**0.0957**	**0.1853**	0.9073	0.8039

Fig 56. Table of test metrics values for all the models

From the results in the figure above, we observe that the Baseline Model with Optimal Hyperparameters has the best test accuracy for both classification of Car Make ID and Car Model ID among the 3 trained models, with a 94.51% test accuracy for classification of Car Make ID and 88.28% accuracy for classification of Car Model ID. Also, the Model with Optimal Hyperparameters and Focal Loss Function has the best test loss for both classification of Car Make ID and Car Model ID among the 3 trained models.

Also, from the results, we can observe that the Baseline Model with Optimal Hyperparameters performed better than the models in past research papers in terms of classification test accuracy for the classification of Car Model ID when compared against the classification test accuracies from past research papers in Fig 2. This goes to show that the state-of-the-art EfficientNet Convolutional Neural Network is better than past architectures in terms of fine-grained car categorisation.

9. Conclusions

In conclusion, based on the results derived from the various experiments that we have performed, the state-of-the-art EfficientNet Convolutional Neural Network performs well for fine-grained car categorization tasks. With our baseline model able to achieve a decent 87.30% test accuracy for classification of Car Make ID and 64.08% accuracy for classification of Car Model ID with just 80 epochs of training.

Also, from our experimental results, we have proven that Hyperband is an efficient algorithm for finding the optimal parameters to use for a given neural network, allowing us to find the optimal parameters for our neural network for fine-grained car categorization. With the optimal parameters obtained from the Hyperband algorithm, the Baseline Model with Optimal Hyperparameters performed way better than the baseline model, with a 7.21% test accuracy increase for classification of Car Make ID and 24.2% accuracy increase for classification of Car Model ID with just 40 epochs of training.

We can also observe that the model with the Focal Loss Advanced Loss Function did not fare better than the Baseline Model with Optimal Hyperparameters and SGD Loss Function in terms of validation accuracies, as well as test accuracies for both classification of Car Make ID and Car Model ID. This is perhaps due to the fact that the optimal parameters that we used for the Baseline Model with Optimal Hyperparameters as well as models with the Focal Loss Advanced Loss Function are more suited for models with an SGD loss function as compared to a model with a Focal Loss function. One improvement that can be made to our project can be that we run the Hyperband algorithm to find the optimal parameters for a neural network utilizing the Focal Loss Advanced Loss Function. The results also go to tell us that the optimization of a model's Hyperparameters is more important than using a more advanced loss function when it comes to improving a model's validation and test accuracy.

Finally, we can conclude that with the optimal parameters, a neural network with the EfficientNet Convolutional Neural Network as the core of the architecture, performs better than the models in past research papers in terms of classification test accuracy for the classification of Car Model ID when compared against the classification test accuracies from past research papers in Fig 2. This goes to show that the state-of-the-art EfficientNet Convolutional Neural Network is better than past architectures in terms of fine-grained car categorisation.

10. Appendix

10.1 Project Resources

For our group's project, these are the resources that were created:
1. CZ4042 Neural Networks and Deep Learning Group Project Report (PDF format)
2. Source Codes used for the experiments, CSV files and README file for running of the Source Codes (.zip file format)
3. Google Drive containing trained models and Google Colaboratory notebook for loading saved models to evaluate on test dataset (https://drive.google.com/drive/folders/1jl7ZE9dXdwymx2dm-QLke46ZfbFXw5s9)

10.2 Running of Saved Models

For our group, after completing the project scope, we have 3 trained models saved in our Google Drive in .h5 format:
1. Baseline Model (cce_loss.h5)
2. Baseline Model with Optimal Hyperparameters (cce_loss_optimised.h5)
3. Model with Optimal Hyperparameters and Focal Loss Function (focal_loss.h5)

To run the saved models and evaluate them on the test dataset, we will need to use the **test.ipynb** notebook located in the same drive as the saved models.
Note: In order to be able to load the models from the Google Drive folder into the **test.ipynb** notebook, one must have a Google Drive account and add the folder into their Google Drive, as shown in the figure below:

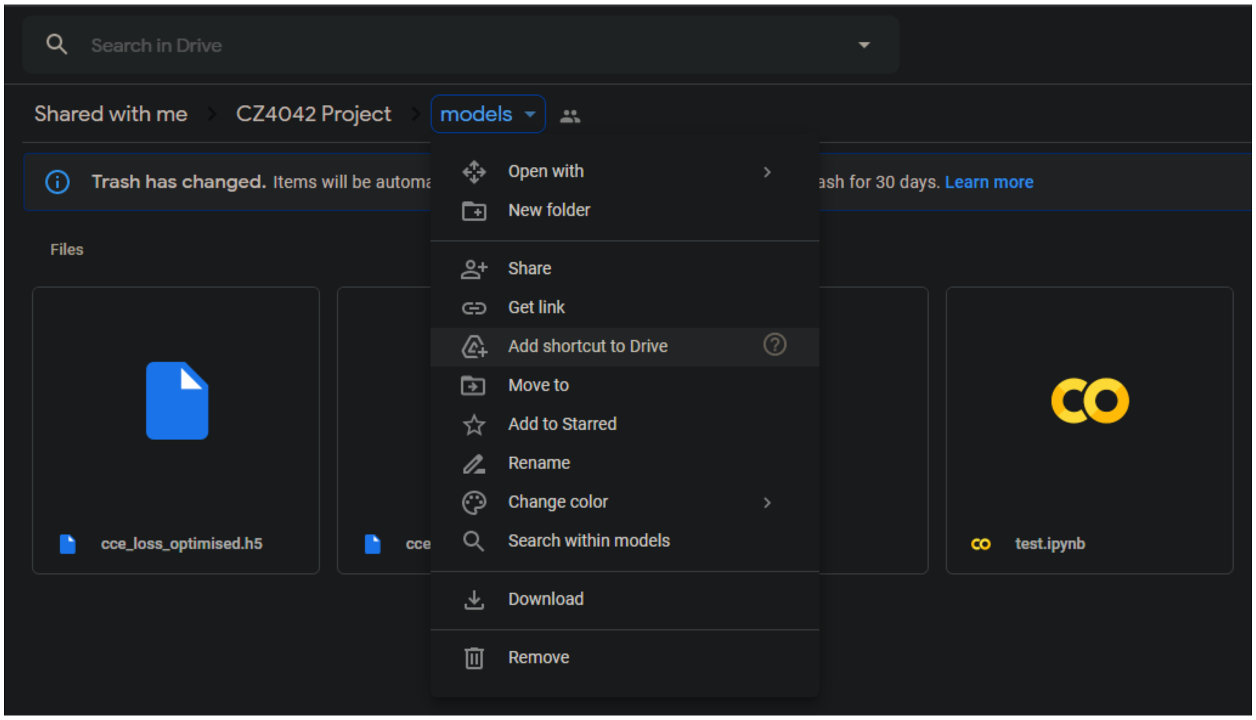
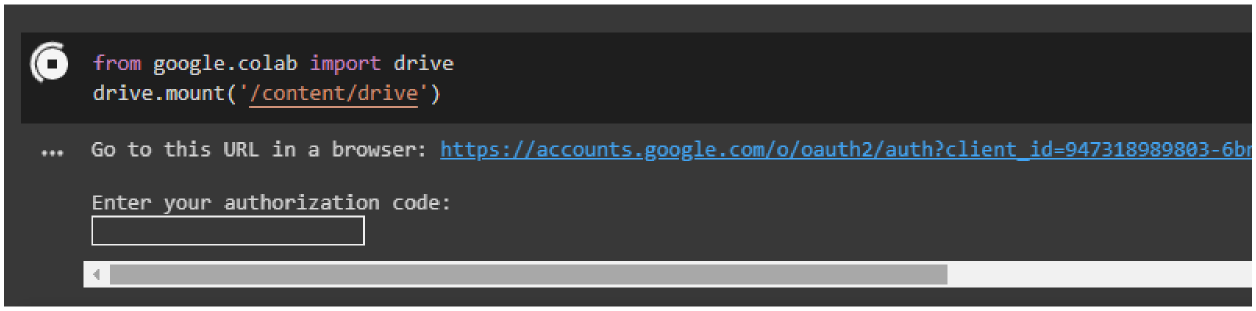


Fig 57. Adding models folder to Google Drive

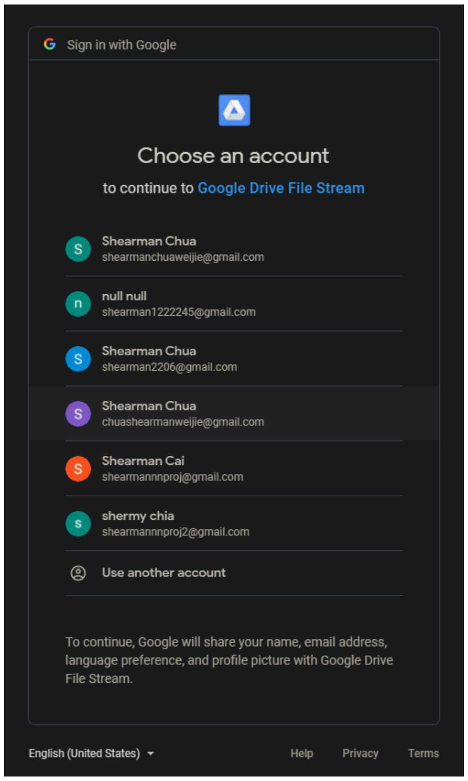
After adding the **models** folder into our Google Drive, we open the **test.ipynb** notebook to load and evaluate the saved models.



```
from google.colab import drive
drive.mount('/content/drive')

Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6br

Enter your authorization code:
```

Fig 58. Mount Google Drive for Python Notebook

When running the **test.ipynb** notebook, there is a cell within the notebook to request the user to mount their Google Drive onto the Google Colaboratory platform. This is to allow the user to be able to load the saved models from the Google Drive folder. When prompted, click the URL to get the authorization code to key into the cell.



Sign in with Google

Choose an account

to continue to Google Drive File Stream

Shearman Chua
shearmanchuaweijie@gmail.com

null null
shearman1222245@gmail.com

Shearman Chua
shearman2206@gmail.com

Shearman Chua
chuashearmanweijie@gmail.com

Shearman Cai
shearmannnproj@gmail.com

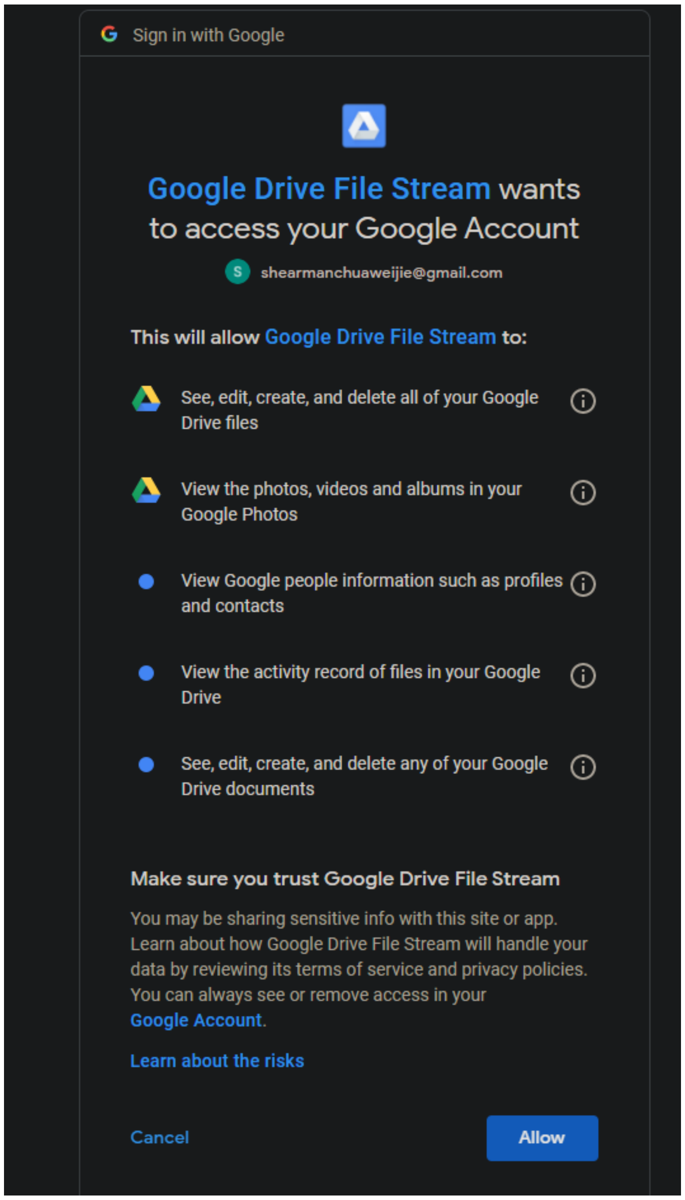
shermy chia
shearmannnproj2@gmail.com

Use another account

To continue, Google will share your name, email address, language preference, and profile picture with Google Drive File Stream.

Fig 59. Choose Drive to Mount

Next, when the URL is clicked, choose the Google Drive account with the **models** folder added.


Sign in with Google

Google Drive File Stream wants to access your Google Account

shearmanchuaweijie@gmail.com

This will allow Google Drive File Stream to:

- See, edit, create, and delete all of your Google Drive files
- View the photos, videos and albums in your Google Photos
- View Google people information such as profiles and contacts
- View the activity record of files in your Google Drive
- See, edit, create, and delete any of your Google Drive documents

Make sure you trust Google Drive File Stream

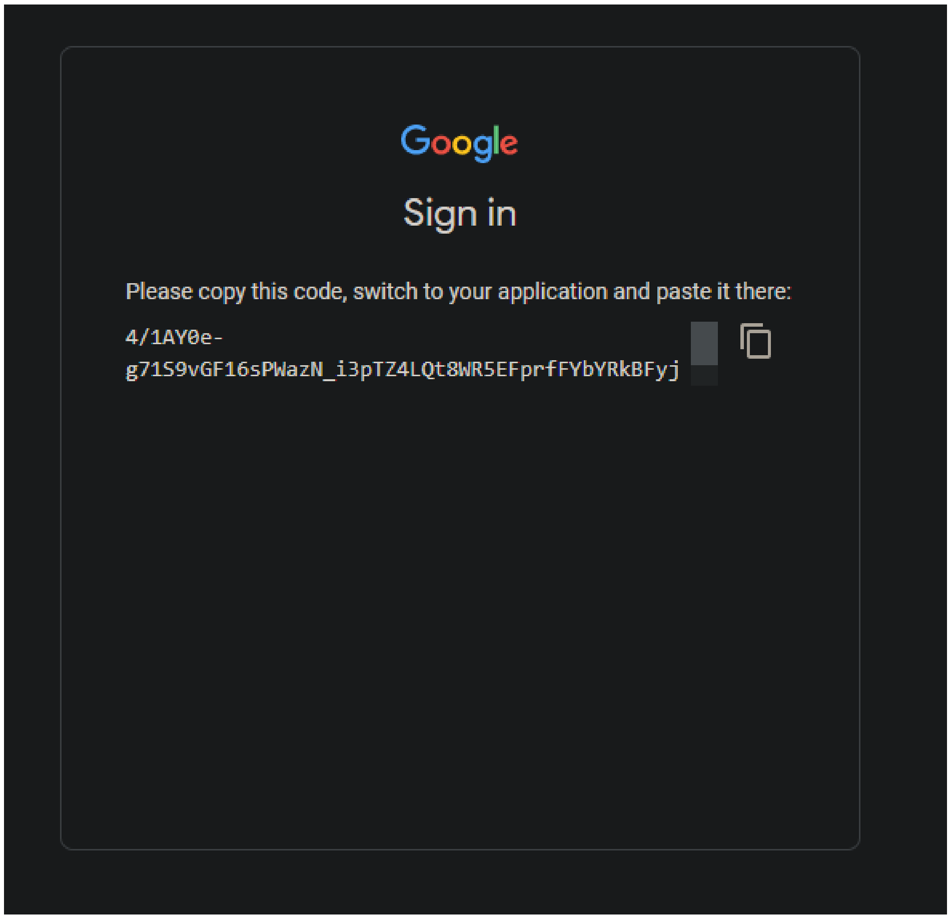
You may be sharing sensitive info with this site or app. Learn about how Google Drive File Stream will handle your data by reviewing its terms of service and privacy policies. You can always see or remove access in your Google Account.

Learn about the risks

Fig 60. Allow Drive to Mount

Next, you will be prompted to allow the Google Drive to be mounted onto Google Colaboratory.

Click on the **Allow** button.



Google

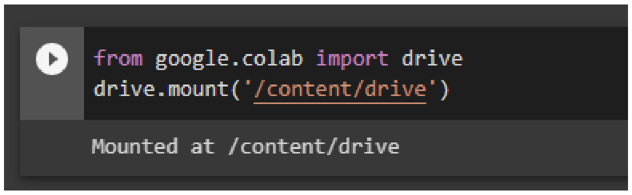
Sign in

Please copy this code, switch to your application and paste it there:

```
4/1AY0e-
g71S9vGF16sPWazN_i3pTZ4LQt8WR5EFprfFYbYRkBFyj
```

Fig 61. Copy Authorization Code

After allowing the Google Drive to be mounted, you will be given an authorization key to be pasted and entered into the **test.ipynb** notebook cell.

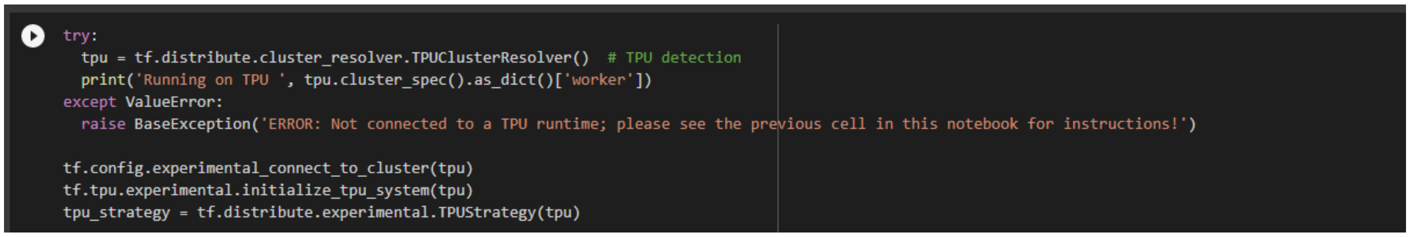


```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Fig 62. Google Drive Mounted

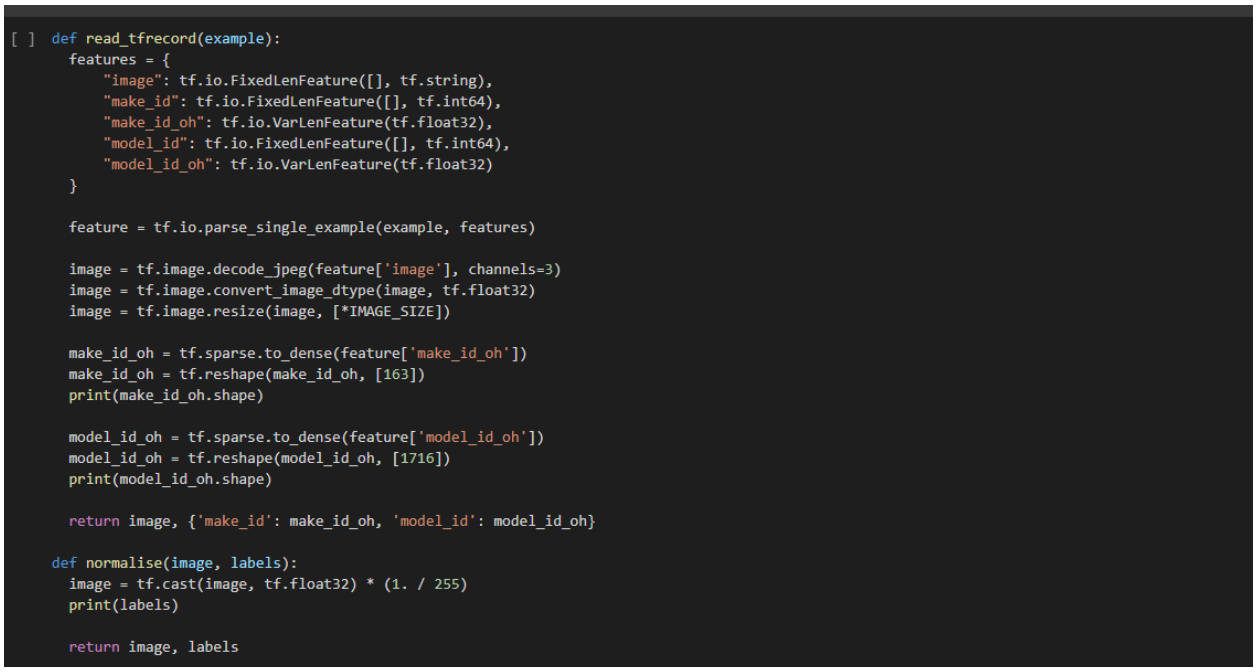
After pasting the authorization key clicking enter into the **test.ipynb** notebook cell, the Google Drive will be mounted.



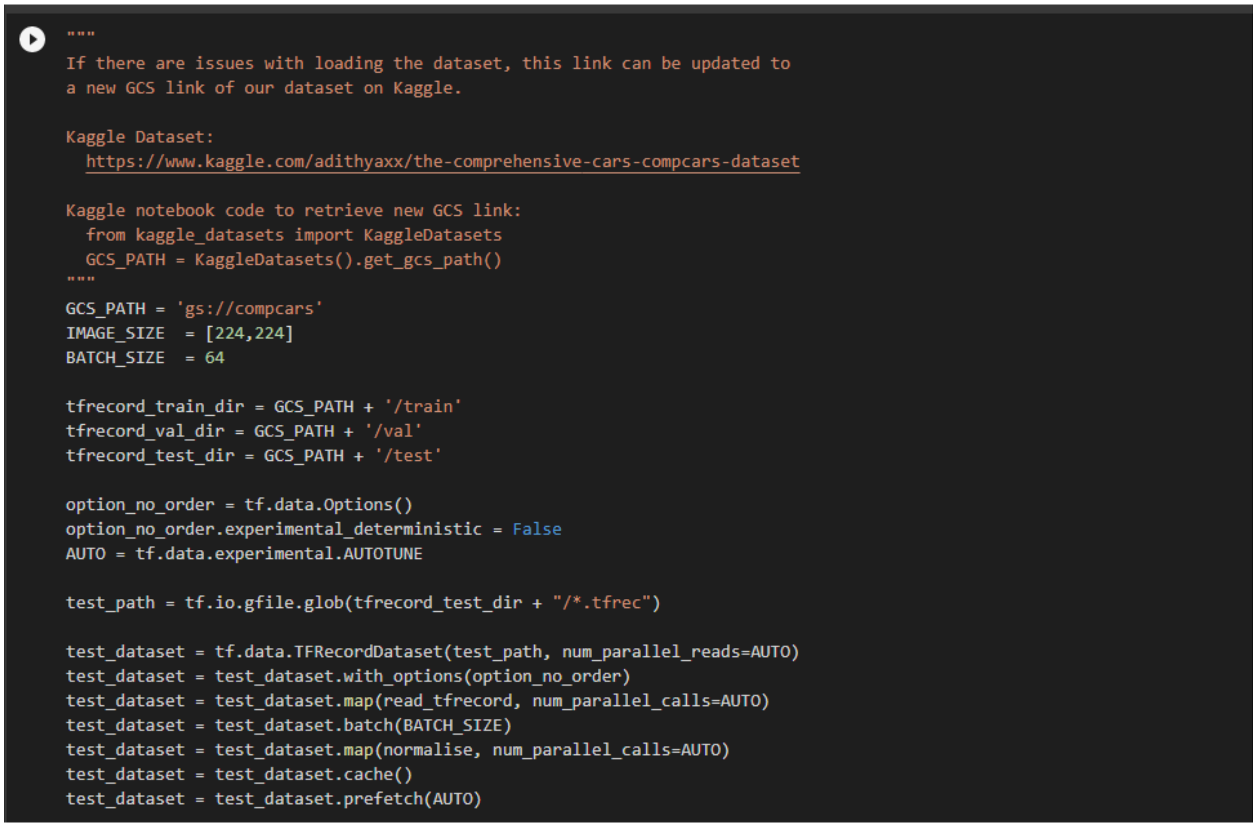
```
try:
  tpu = tf.distribute.cluster_resolver.TPUClusterResolver()  # TPU detection
  print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
except ValueError:
  raise BaseException('ERROR: Not connected to a TPU runtime; please see the previous cell in this notebook for instructions!')

tf.config.experimental_connect_to_cluster(tpu)
tf.tpu.experimental.initialize_tpu_system(tpu)
tpu_strategy = tf.distribute.experimental.TPUStrategy(tpu)
```

Fig 63. Google Drive Mounted

After mounting the Google Drive, we must run the cell as shown in the figure above to enable the use of TPU for running the models to be loaded and evaluated.



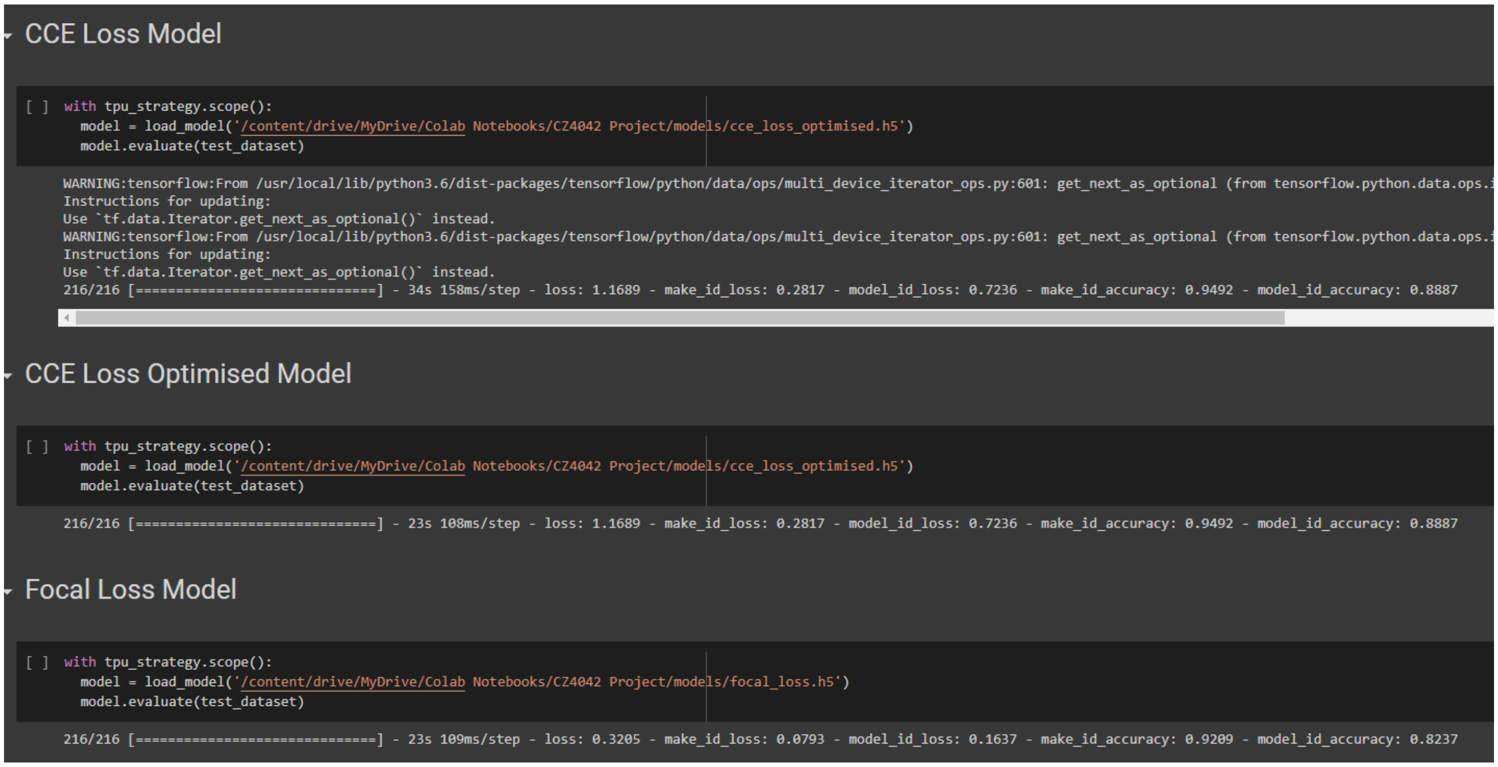
```
[ ]  def read_tfrecord(example):
       features = {
           "image": tf.io.FixedLenFeature([], tf.string),
           "make_id": tf.io.FixedLenFeature([], tf.int64),
           "make_id_oh": tf.io.VarLenFeature(tf.float32),
           "model_id": tf.io.FixedLenFeature([], tf.int64),
           "model_id_oh": tf.io.VarLenFeature(tf.float32)
       }

       feature = tf.io.parse_single_example(example, features)

       image = tf.image.decode_jpeg(feature['image'], channels=3)
       image = tf.image.convert_image_dtype(image, tf.float32)
       image = tf.image.resize(image, [*IMAGE_SIZE])

       make_id_oh = tf.sparse.to_dense(feature['make_id_oh'])
       make_id_oh = tf.reshape(make_id_oh, [163])
       print(make_id_oh.shape)

       model_id_oh = tf.sparse.to_dense(feature['model_id_oh'])
       model_id_oh = tf.reshape(model_id_oh, [1716])
       print(model_id_oh.shape)

       return image, {'make_id': make_id_oh, 'model_id': model_id_oh}

     def normalise(image, labels):
       image = tf.cast(image, tf.float32) * (1. / 255)
       print(labels)

       return image, labels
```



```
"""
If there are issues with loading the dataset, this link can be updated to
a new GCS link of our dataset on Kaggle.

Kaggle Dataset:
  https://www.kaggle.com/adithyaxx/the-comprehensive-cars-compcars-dataset

Kaggle notebook code to retrieve new GCS link:
  from kaggle_datasets import KaggleDatasets
  GCS_PATH = KaggleDatasets().get_gcs_path()
"""
GCS_PATH = 'gs://compcars'
IMAGE_SIZE  = [224,224]
BATCH_SIZE  = 64

tfrecord_train_dir = GCS_PATH + '/train'
tfrecord_val_dir = GCS_PATH + '/val'
tfrecord_test_dir = GCS_PATH + '/test'

option_no_order = tf.data.Options()
option_no_order.experimental_deterministic = False
AUTO = tf.data.experimental.AUTOTUNE

test_path = tf.io.gfile.glob(tfrecord_test_dir + "/*.tfrec")

test_dataset = tf.data.TFRecordDataset(test_path, num_parallel_reads=AUTO)
test_dataset = test_dataset.with_options(option_no_order)
test_dataset = test_dataset.map(read_tfrecord, num_parallel_calls=AUTO)
test_dataset = test_dataset.batch(BATCH_SIZE)
test_dataset = test_dataset.map(normalise, num_parallel_calls=AUTO)
test_dataset = test_dataset.cache()
test_dataset = test_dataset.prefetch(AUTO)
```

Fig 64. Functions to Prepare and Load Test dataset

Next, we will need to run the 2 functions in the figure above in order to load the test dataset TFRecords from the Google Storage Bucket to be used to evaluate the trained models on.



CCE Loss Model

```
[ ] with tpu_strategy.scope():
      model = load_model('/content/drive/MyDrive/Colab Notebooks/CZ4042 Project/models/cce_loss_optimised.h5')
      model.evaluate(test_dataset)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/data/ops/multi_device_iterator_ops.py:601: get_next_as_optional (from tensorflow.python.data.ops.i
Instructions for updating:
Use `tf.data.Iterator.get_next_as_optional()` instead.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/data/ops/multi_device_iterator_ops.py:601: get_next_as_optional (from tensorflow.python.data.ops.i
Instructions for updating:
Use `tf.data.Iterator.get_next_as_optional()` instead.
216/216 [==============================] - 34s 158ms/step - loss: 1.1689 - make_id_loss: 0.2817 - model_id_loss: 0.7236 - make_id_accuracy: 0.9492 - model_id_accuracy: 0.8887
```

CCE Loss Optimised Model

```
[ ] with tpu_strategy.scope():
      model = load_model('/content/drive/MyDrive/Colab Notebooks/CZ4042 Project/models/cce_loss_optimised.h5')
      model.evaluate(test_dataset)
```

```
216/216 [==============================] - 23s 108ms/step - loss: 1.1689 - make_id_loss: 0.2817 - model_id_loss: 0.7236 - make_id_accuracy: 0.9492 - model_id_accuracy: 0.8887
```

Focal Loss Model

```
[ ] with tpu_strategy.scope():
      model = load_model('/content/drive/MyDrive/Colab Notebooks/CZ4042 Project/models/focal_loss.h5')
      model.evaluate(test_dataset)
```

```
216/216 [==============================] - 23s 109ms/step - loss: 0.3205 - make_id_loss: 0.0793 - model_id_loss: 0.1637 - make_id_accuracy: 0.9209 - model_id_accuracy: 0.8237
```

Fig 65. Functions to Load the Trained Models and Evaluate on Test dataset

Finally, we run the 3 cells shown in the figure above to load the 3 trained models that we have in the Google Drive folder and we call the **model.evaluate()** function to evaluate the trained models on the test dataset.

11. References

[1]	L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization,” arXiv.org, 18-Jun-2018. [Online]. Available: https://arxiv.org/abs/1603.06560. [Accessed: 1-Nov-2020].

[2]	K. Pasupa, S. Vatathanavaro, and S. Tungjitnob, "Convolutional neural networks based focal loss for class imbalance problem: a case study of canine red blood cells morphology classification," *Journal of Ambient Intelligence and Humanized Computing,* 2020/02/15 2020, doi: 10.1007/s12652-020-01773-x.

[3]	L. Yang, P. Luo, C. C. Loy, X. Tang, “A large-scale car dataset for fine-grained categorization and verification,” in IEEE Computer Vision and Pattern Recognition (CVPR), 2015

[4]	H. Liu, Y. Tian, Y. Wang, L. Pang, T. Huang, “Deep relative distance learning: tell the difference between similar vehicles,” in Computer Vision and Pattern Recognition (CVPR), 2016