CIFAR100 Image Classification - Coarse

Name: Soh Hong Yu

Admin Number: P2100775

Class: DAAA/FT/2B/01

Module Code: ST1504 Deep Learning

References (In Harvard format):

1. Krizhevsky, A., Nair, V. and Hinton, G. (2009) The CIFAR-10 dataset and CIFAR-100 dataset, CIFAR-10 and CIFAR-100 datasets.
   Available at: https://www.cs.toronto.edu/~kriz/cifar.html (Accessed: November 24, 2022).
2. User, D. (2022) An overview of state of the art (SOTA) deep neural networks (dnns), Deci.
   Available at: https://deci.ai/blog/sota-dnns-overview/ (Accessed: November 19, 2022).
3. Cox, S. (2021) The overlooked technique of image averaging, Photography Life.
   Available at: https://photographylife.com/image-averaging-technique (Accessed: November 19, 2022).
4. Gupta, A. et al. (2021) Adam vs. SGD: Closing the generalization gap on Image Classification, Adam vs. SGD: Closing the generalization gap on image classification.
   Available at: https://www.opt-ml.org/papers/2021/paper53.pdf (Accessed: November 19, 2022).
5. Nelson, J. (2020) Why and how to implement random crop data augmentation, Roboflow Blog. Roboflow Blog.
   Available at: https://blog.roboflow.com/why-and-how-to-implement-random-crop-data-augmentation (Accessed: November 19, 2022).
6. Zvornicanin, E. (2022) Convolutional Neural Network vs. Regular Neural Network, Baeldung on Computer Science.
   Available at: https://www.baeldung.com/cs/convolutional-vs-regular-nn (Accessed: November 19, 2022).
7. Baker, J. (2021) 8.2. networks using blocks (VGG)¶
   Available at: https://d2l.ai/chapter_convolutional-modern/vgg.html (Accessed: November 19, 2022).
8. Shinde, Y. (2021) How to code your resnet from scratch in tensorflow? , Analytics Vidhya.
   Available at: https://www.analyticsvidhya.com/blog/2021/08/how-to-code-your-resnet-from-scratch-in-tensorflow (Accessed: November 19, 2022).
9. Baker, J. (2021) 8.6. residual networks (ResNet)
   Available at: https://d2l.ai/chapter_convolutional-modern/resnet.html (Accessed: November 19, 2022).
10. Tan, M. and Le, Q.V. (2021) EFFICIENTNETV2: Smaller models and faster training, arXiv.org.
    Available at: https://arxiv.org/abs/2104.00298v3 (Accessed: November 25, 2022).
11. Tan, et (2019) Papers with code - efficientnet explained, Explained | Papers With Code.
    Available at: https://paperswithcode.com/method/efficientnet (Accessed: November 25, 2022).

Project Objective

Implement an image classifier using a deep learning network

Background Information

This dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs). Here is the list of classes in the CIFAR-100:

Initialising Libraries and Variables

In [ ]:
```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report
from keras.utils import to_categorical
import keras_tuner as kt
from keras.regularizers import l1, l2
from keras.layers import AveragePooling2D, ZeroPadding2D, BatchNormalization, Activation, Max
from keras.models import Sequential
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Normalization, Dense, Conv2D, Dropout, BatchNormalization, ReLU
from keras.models import Sequential
from keras.models import Model
from keras import Input
from keras.optimizers import *
from keras.callbacks import EarlyStopping
import visualkeras
from keras.layers import GlobalAveragePooling2D
```

Checking GPU

In [ ]:
```
# Check if Cuda GPU is available
tf.config.list_physical_devices('GPU')
```

Out[ ]:
```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Loading Datasets

In [ ]:
```
df = tf.keras.datasets.cifar100.load_data(label_mode="coarse")
```

In [ ]:
```
(x_train_val, y_train_val), (x_test, y_test) = df
```

As the training set will be used to train the model, we will need a set of data for model tuning, and the testing set will be used to evaluate the final model, ensuring the model is generalise and not overfit to the validation set due to model tuning.

To decide what size of the validation set, I have decided to split the data by 80:20 of the train set as the validation set.

Training set - 40000
Validation set - 10000
Testing set - 10000

In [ ]:
```
train_size = 40000
x_train, y_train = x_train_val[:train_size], y_train_val[:train_size]
x_val, y_val = x_train_val[train_size:], y_train_val[train_size:]
```

Exploratory Data Analysis

We will begin by conducting an exploratory data analysis of the data, to gain a better understanding of the characteristics of the dataset.

x_train: uint8 NumPy array of grayscale image data with shapes (50000, 32, 32, 3), containing the training data. Pixel values range from 0 to 255.

y_train: uint8 NumPy array of labels (integers in range 0-99) with shape (50000, 1) for the training data.

x_test: uint8 NumPy array of grayscale image data with shapes (10000, 32, 32, 3), containing the test data. Pixel values range from 0 to 255.

y_test: uint8 NumPy array of labels (integers in range 0-99) with shape (10000, 1) for the test data.

There are 20 different type of labels in the dataset. From the dataset, each value represent an item. The following list is the description of each value.

Labels

- 0 : aquatic mammals
- 1 : fish
- 2 : flowers
- 3 : food containers
- 4 : fruit and vegetables
- 5 : household electrical devices
- 6 : household furniture
- 7 : insects
- 8 : large carnivores
- 9 : large man-made outdoor things
- 10 : large natural outdoor scenes
- 11 : large omnivores and herbivores
- 12 : medium-sized mammals
- 13 : non-insect invertebrates
- 14 : people
- 15 : reptiles
- 16 : small mammals
- 17 : trees
- 18 : vehicles 1
- 19 : vehicles 2

In [ ]:
```
# coarse labels
class_labels = {
    0 : "aquatic mammals",
    1 : "fish",
    2 : "flowers",
    3 : "food containers",
    4 : "fruit and vegetables",
    5 : "household electrical devices",
    6 : "household furniture",
    7 : "insects",
    8 : "large carnivores",
    9 : "large man-made outdoor things",
    10 : "large natural outdoor scenes",
    11 : "large omnivores and herbivores",
    12 : "medium-sized mammals",
    13 : "non-insect invertebrates",
    14 : "people",
    15 : "reptiles",
    16 : "small mammals",
    17 : "trees",
    18 : "vehicles 1",
    19 : "vehicles 2",
}

NUM_CLASS = 20
```

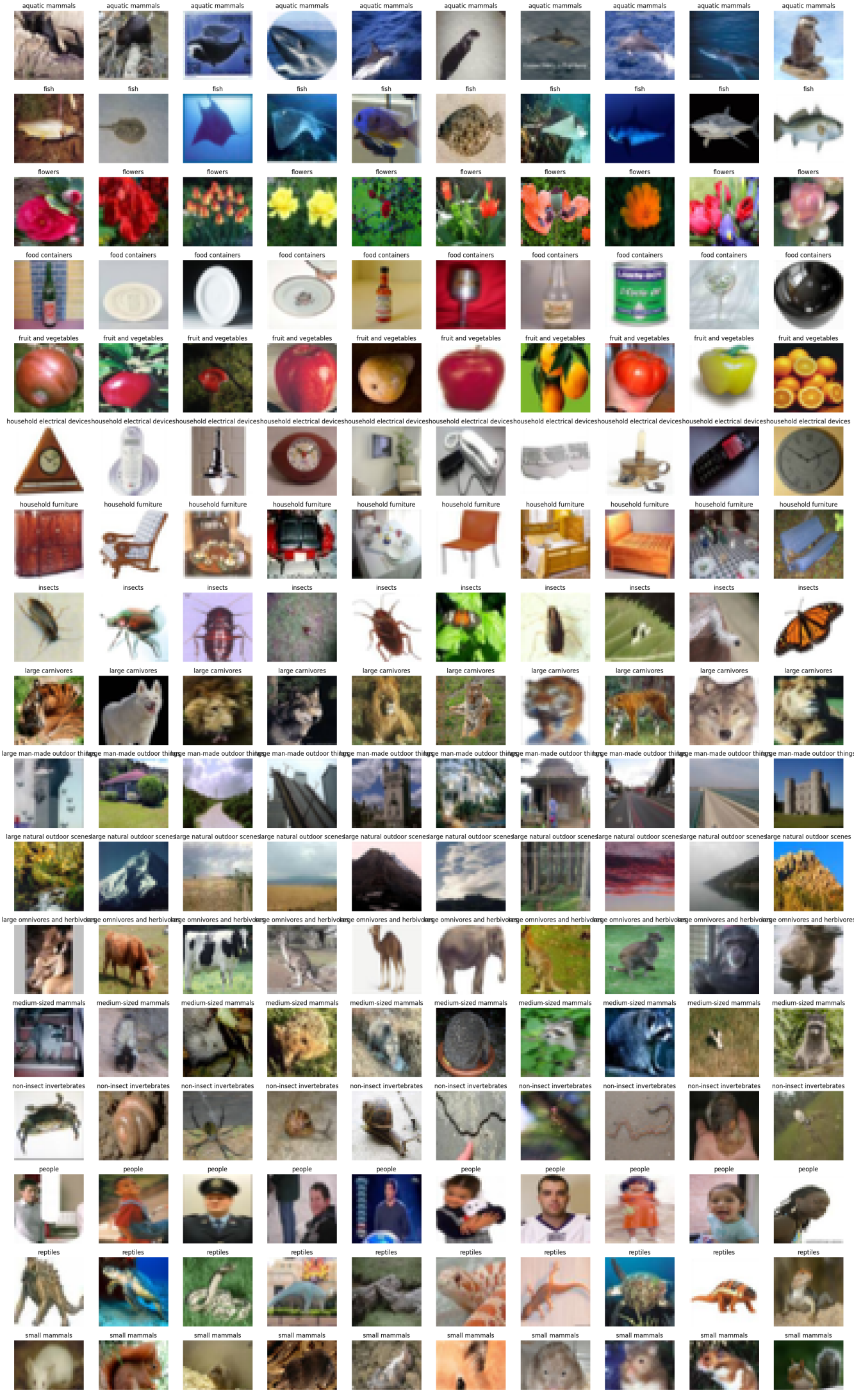
Each image is a 32x32 image as well as 3 color channel [RGB] (coloured image). Therefore, we can set the IMG_SIZE as a tuple (32, 32, 3)

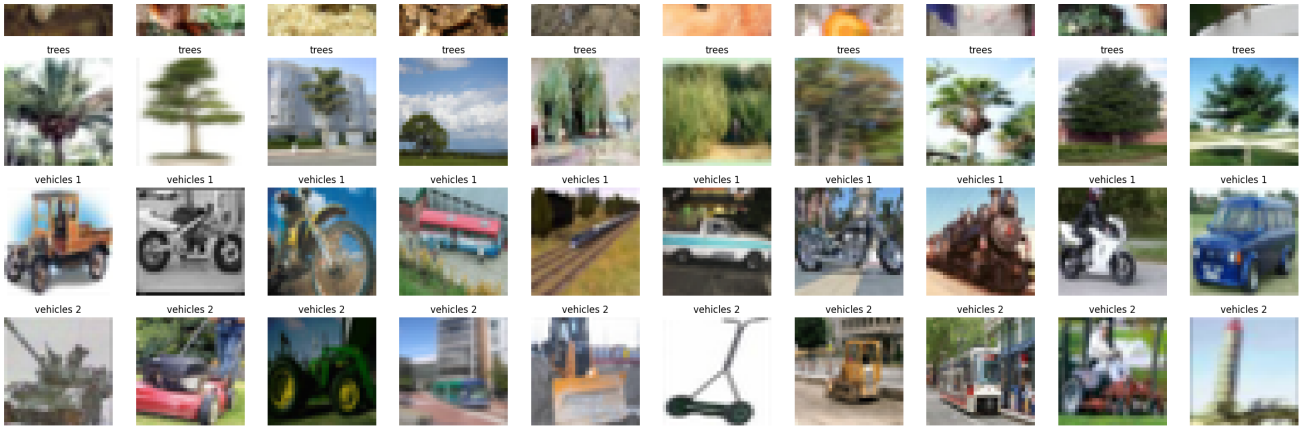
In [ ]:
```
IMG_SIZE = (32, 32, 3)
```

Visualising the Dataset

Let's look at what the images look like.

In [ ]:
```
fig, ax = plt.subplots(20, 10, figsize=(30, 60))
for i in range(20):
    images = x_train[np.squeeze(y_train == i)]
    random_index = np.random.choice(images.shape[0], 10, replace=False)
    images = images[random_index]
    label = class_labels[i]
    for j in range(10):
        subplot = ax[i, j]
        subplot.axis("off")
        subplot.imshow(images[j], cmap='Greys')
        subplot.set_title(label)

plt.show()
```


aquatic mammals
fish
flowers
food containers
fruit and vegetables
household electrical devices
household furniture
insects
large carnivores
large man-made outdoor things
large natural outdoor scenes
large omnivores and herbivores
medium-sized mammals
non-insect invertebrates
people
reptiles
small mammals



Observation

We note that there are certain images with other objects that is in the system as well. There is a wide variety of images and that the images are of different type of items too. For example, vehicle 2 has a lawn mower and rockets. This makes it hard for the model to generalise which might cause some issue later on.

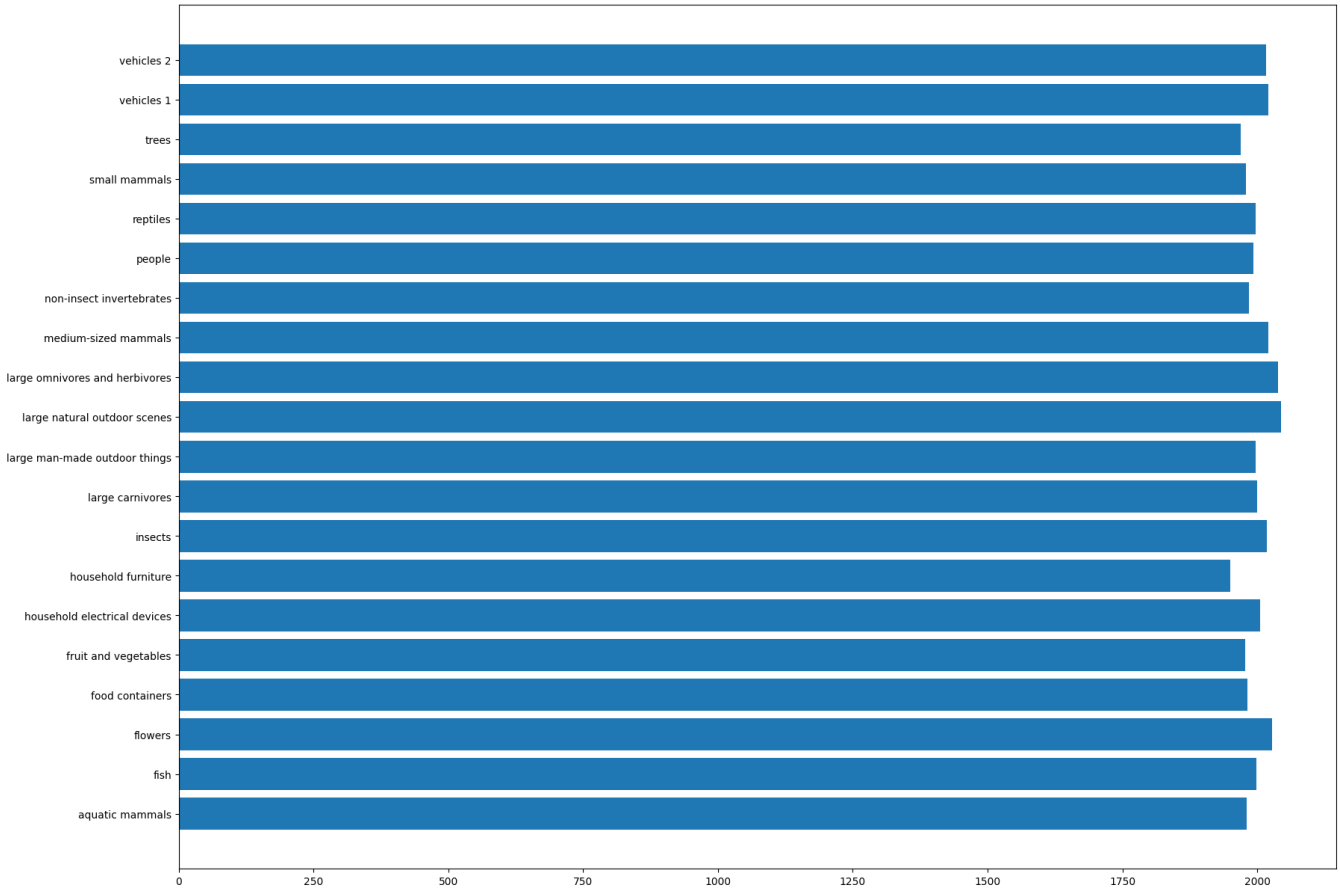
Class Distribution

When training a machine learning model, it is always important to check the distribution of the different classes in the dataset. This will inform us which metrics is the best to use and if anything is needed to balance the classes.

In [ ]:
```
labels, counts = np.unique(y_train, return_counts=True)
for label, count in zip(labels, counts):
    print(f"{class_labels[label]}: {count}")
```

```
aquatic mammals: 1980
fish: 1999
flowers: 2028
food containers: 1982
fruit and vegetables: 1978
household electrical devices: 2006
household furniture: 1950
insects: 2018
large carnivores: 2000
large man-made outdoor things: 1997
large natural outdoor scenes: 2044
large omnivores and herbivores: 2039
medium-sized mammals: 2020
non-insect invertebrates: 1985
people: 1993
reptiles: 1997
small mammals: 1979
trees: 1969
vehicles 1: 2020
vehicles 2: 2016
```

In [ ]:
```
plt.figure(figsize=(20, 15))
plt.barh(labels, counts, tick_label=list(class_labels.values()))
plt.show()
```



Observations

As we can see from the bar graph, the distribution of the images is even. This suggest that accuracy can be use as a primary metric.

Image Pixel Distribution

We need to know the pixel intensity and know the distribution of the pixels

In [ ]:
```
print("Max: ", np.max(x_train))
print("Min: ", np.min(x_train))
```

```
Max:  255
Min:  0
```

As expected, our pixels have values between 0 and 255.

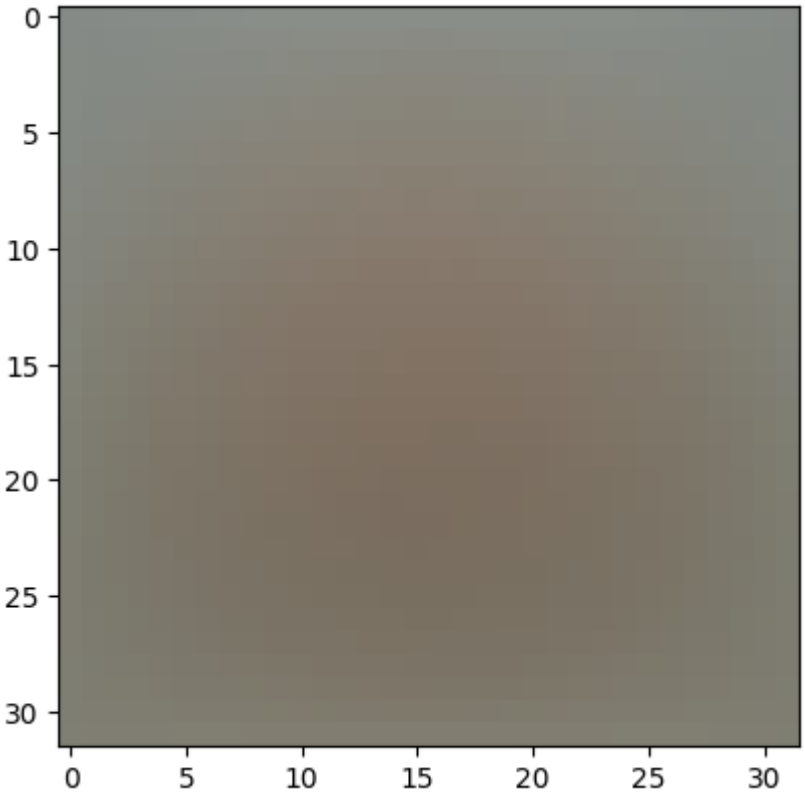
In [ ]:
```
mean, std = np.mean(x_train, axis=(0, 1, 2)), np.std(x_train, axis=(0, 1, 2))
print("Mean:", mean)
print("std:", std)
```

```
Mean: [129.26910793 124.11666553 112.55583118]
std: [68.11519598 65.3142698  70.31977601]
```

Image Averaging

Image Averaging involves stacking multiple photos on top of each other and averaging them together. The main purpose is to see the noise of the image adn therefore reducing it.

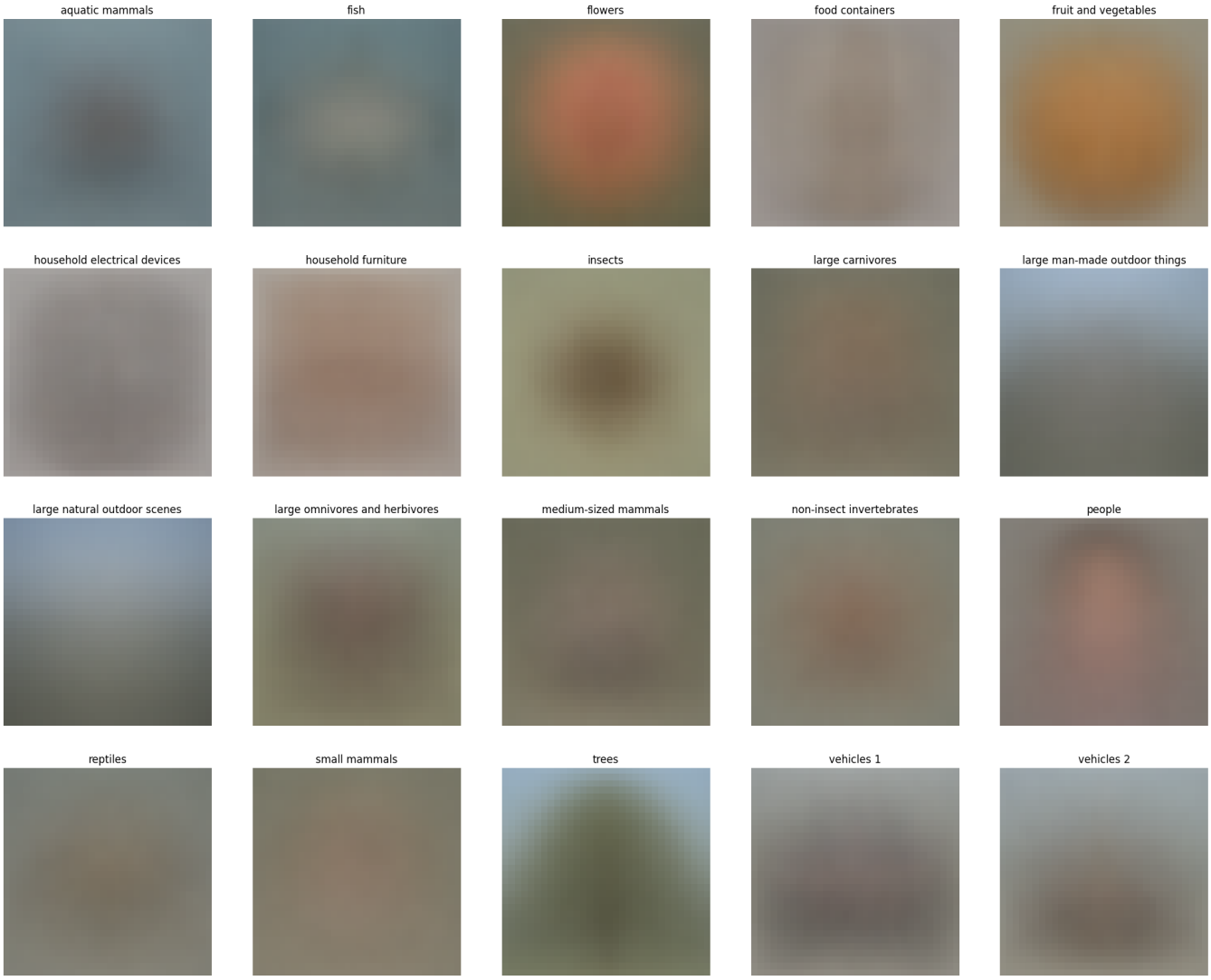
In [ ]:
```
plt.imshow(np.mean(x_train, axis=0) / 255, cmap='Greys')
plt.show()
```



Observation

We cannot see a single thing from the image. This is likely due to the color of the images overlaying each other giving this blur effect.

In [ ]:
```
fig, ax = plt.subplots(4, 5, figsize=(25, 20))

for idx, subplot in enumerate(ax.ravel()):
    avg_image = np.mean(x_train[np.squeeze(y_train == idx)], axis=0) / 255
    subplot.imshow(avg_image, cmap='Greys')
    subplot.set_title(f"{class_labels[idx]}")
    subplot.axis("off")
```



Observations

Although the average images is blurry, we can make out the images of a trees, flowers, fruits and vegetables etc. It is more difficult to make out the average image for the other classes, which might suggest that it is harder to predict these classes. We also note some of the average images like the flowers and fruits and vegetables have a similar shade and average image which might be a problem for the model.

Data Preprocessing

Before modelling, its is important to perform data preprocessing

One Hot Encoding

As they are, the current labels are encoded from 0-99, we will one hot encode the labels.

In [ ]:
```
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)
```

In [ ]:
```
print(y_train[0])
print("Label:", tf.argmax(y_train[0]))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
Label: tf.Tensor(11, shape=(), dtype=int64)
```

Normalizing the inputs

Image normalisation is done to the dataset.

Normalising the inputs means that we will calculate the mean and standard deviation of the training set, and then apply the formula below.

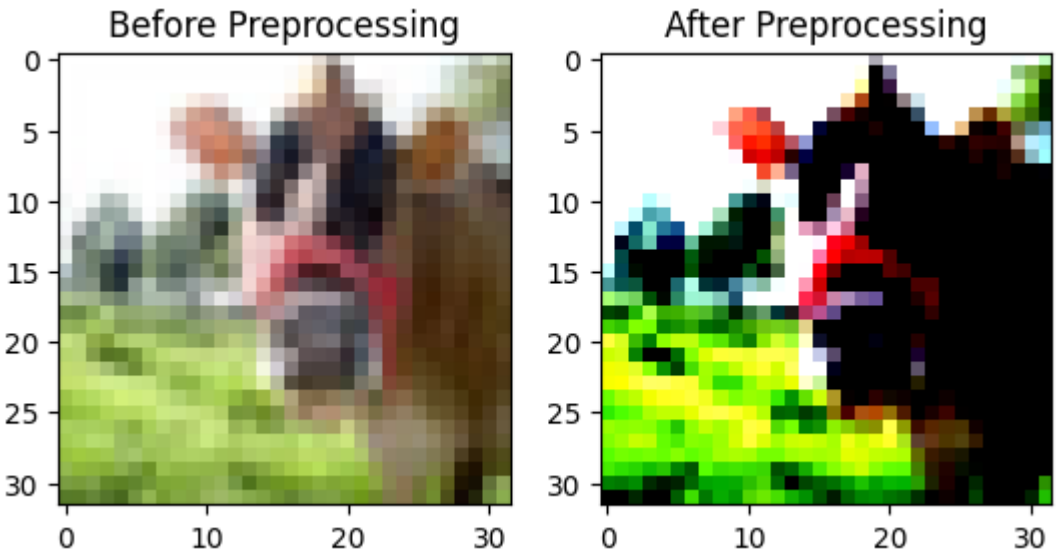
$$X = \frac{X - \mu}{\sigma}$$

Pixel values of each pixel are on similar scale, therefore normalisation can be used. This helps to optimize the algorithm to better converge during gradient descent.

In [ ]:
```
pre_processing_v1 = Normalization()
pre_processing_v1.adapt(x_train)
```

In [ ]:
```
fig, ax = plt.subplots(ncols=2)

ax[0].imshow(x_train[0])
ax[0].set_title('Before Preprocessing')
ax[1].imshow(tf.squeeze(pre_processing_v1(x_train[:1, :, :])))
ax[1].set_title('After Preprocessing')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..25
5] for integers).
```



Data Augmentation

To prevent overfitting of the model, we will apply data augmentation. Data augmentation is a method to reduce the variance of a model by imposing random transformations on the data for training.

Types of Image Data Augmentations

- Flipping
- Cropping
- Rotating
- Scaling
- Shearing
- Many more ...

For this case, we will be using only flipping, resizing and cropping. This is because as we seen during our exploratory data analysis. The images are all in the same orientation which means we can flip left and right to help make data augmentation better. To have more data points, we will resize and add more padding to the images, this will allow us to crop without cropping the object out of the image. Cropping the images also allows the model to generalise the data and identify features more easily.

Note: we will only be augmenting the training data as we do not want to edit the validation and test data as they will be used to evaluate the model's accuracy.

Batch Size

To help make the model to have a regularizing effect, we will choose the smaller batch sizes. We will choose a batch size of 64 as it allows the model to converge more easily.

In [ ]:
```
BATCH_SIZE = 64
```

Basic Augmentation

In [ ]:
```
def data_augmentation(x_train):
    imageArr = []
    for images in x_train:
        tf.convert_to_tensor(images)
        randomVal = np.random.randint(0,2)
        if randomVal == 1:
            image = tf.image.random_flip_left_right(images)
            image = tf.image.resize_with_crop_or_pad(
                image, IMG_SIZE[0] + 4, IMG_SIZE[1] + 4)
            image = tf.image.random_crop(
                image, size=IMG_SIZE
            )
            images = image
        imageArr.append(images)
    return np.array(imageArr)
```

In [ ]:
```
x_train_aug = np.copy(x_train)
```

In [ ]:
```
x_train_aug = data_augmentation(x_train_aug)
```

Let's see what happened to the data after we have augmented it.

In [ ]:
```
fig, ax = plt.subplots(2, 20, figsize=(100, 10))
for idx in range(40):
    subplot = ax.ravel()[idx]
    y_label = np.argmax(y_train, axis=1)
    if idx >= 20:
        subplot.set_title(f"A: {class_labels[idx % 20]}")
        subplot.imshow(x_train_aug[y_label == idx % 20][0])
    else:
        subplot.set_title(f"O: {class_labels[idx % 20]}")
        subplot.imshow(x_train[y_label == idx % 20][0])
    subplot.axis("off")
plt.show()
```

Observations

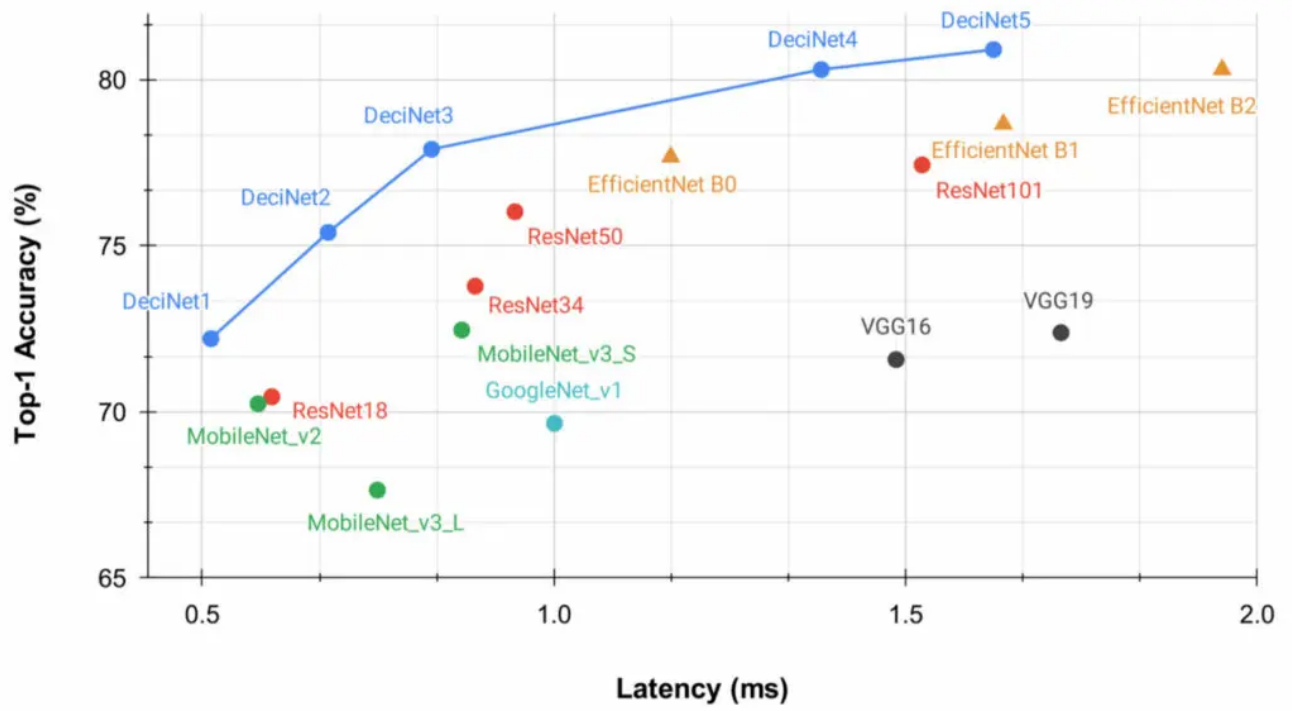
As we can see, some of the images have been shifted, rotated and cropped. This shows that the image augmentation works

Building Models

We will be building a few deep learning models to solve the image classification problem.

Model List:

1. Fully Connected Neural Network Model (Baseline)
2. Conv2D Neural Network Model
3. CustomVGG Model
4. CustomVGG16 Model
5. CustomResNet-10 Model
6. EfficientNetV2 Model



Overfitting

To prevent overfitting, we will be using Early Stopping. This will stop model training once it begins to overfit.

Optimizers

There are a lot of different types of optimizers offered by Tensorflow. The most common 2 are Adam and SGD optimizers.

Adam

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

SGD

SGD also known as Stochastic gradient descent is an iterative method for optimizing an objective

function with suitable smoothness.

Difference between Adam and SGD

Adam is faster compared to SGD, this is due to Adam using coordinate wise gradient clipping which tackle heavy-tailed noise. It also updates the learning rate for each network weight individually. However, SGD is known to perform better than SGD for image classification tasks. As Adam takes "shortcuts" as mentioned previously which is better for NLP and other purposes but for Image Classification, every detail is important to distinguish what the image is. Therefore for all the subsequent models, we will be using the SGD as our optimizer.

Utility Function

Before we begin building our models, we will first be building some functions that will help us to compare our models more easily.

In [ ]:
```
def plot_loss_curve(modelInfo):
    history = modelInfo.history
    history = pd.DataFrame(history)
    epochs = list(range(1, len(history) + 1))
    if np.max(history["val_loss"]) > 1 or np.max(history["loss"]) > 1:
        fig, ax = plt.subplots(1, 2, figsize=(20, 10))
        ax[0].set_title("Plot Loss Curve")
        ax[1].set_title("Plot Accuracy Curve")
        ax[0].scatter(epochs, history["loss"])
        ax[0].plot(epochs, history["loss"], label="Training Loss")
        ax[0].scatter(epochs, history["val_loss"])
        ax[0].plot(epochs, history["val_loss"], label="Validation Loss")
        ax[1].scatter(epochs, history["accuracy"])
        ax[1].plot(epochs, history["accuracy"], label="Training Accuracy")
        ax[1].scatter(epochs, history["val_accuracy"])
        ax[1].plot(epochs, history["val_accuracy"], label="Validation Accuracy")
        ax[0].set_ylabel("Accuracy")
        ax[0].set_xlabel("Epochs")
        ax[1].set_ylabel("Accuracy")
        ax[1].set_xlabel("Epochs")
        plt.legend()
    else:
        fig, ax = plt.subplots(1, 1, figsize=(20, 10))
        plt.title("Plot Loss Curve")
        plt.scatter(epochs, history["loss"])
        plt.plot(epochs, history["loss"], label="Training Loss")
        plt.scatter(epochs, history["val_loss"])
        plt.plot(epochs, history["val_loss"], label="Validation Loss")
        plt.scatter(epochs, history["accuracy"])
        plt.plot(epochs, history["accuracy"], label="Training Accuracy")
        plt.scatter(epochs, history["val_accuracy"])
        plt.plot(epochs, history["val_accuracy"], label="Validation Accuracy")
        plt.ylabel("Accuracy")
        plt.xlabel("Epochs")
        plt.legend()
    return fig
```

In [ ]:
```
allResults = pd.DataFrame()
```

In [ ]:
```
def storeResult(modelInfo):
    history = modelInfo.history
    global allResults
    best_val_idx = np.argmax(history["val_accuracy"])
    result = {}
    result["Model Name"] = modelInfo.model._name
```

```
    result["Epochs"] = len(history["loss"])
    result["Batch Size"] = BATCH_SIZE
    result["Train Loss"] = history["loss"][best_val_idx]
    result["Val Loss"] = history["val_loss"][best_val_idx]
    result["Train Acc"] = history["accuracy"][best_val_idx]
    result["Val Acc"] = history["val_accuracy"][best_val_idx]
    result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
    allResults = allResults.append(result, ignore_index=True)
    return result
```

Baseline Fully Connected Neural Network

As our baseline model, we will be using it to compare against our other models that we are trying to build. This model will be very simple Model using the Sequential class and 3 Hidden Layers. For each hidden layer, we will be using the ReLU activation function and for the final output layer we will be using softmax as there is multiple classes therefore sigmoid will not be usable. As there are multiple category that we are predicting, we will be using the categorical_crossentropy as our loss function. The optimizer will be SGD as mentioned previously and we will be using the metrics of accuracy as the classes are quite balanced.

Training baseline model without Data Augmentation

To train the baseline model, we will first use our unaugmented data to fit and train the model. Subsequently, we will use our augmented data to fit and train and compare the difference.

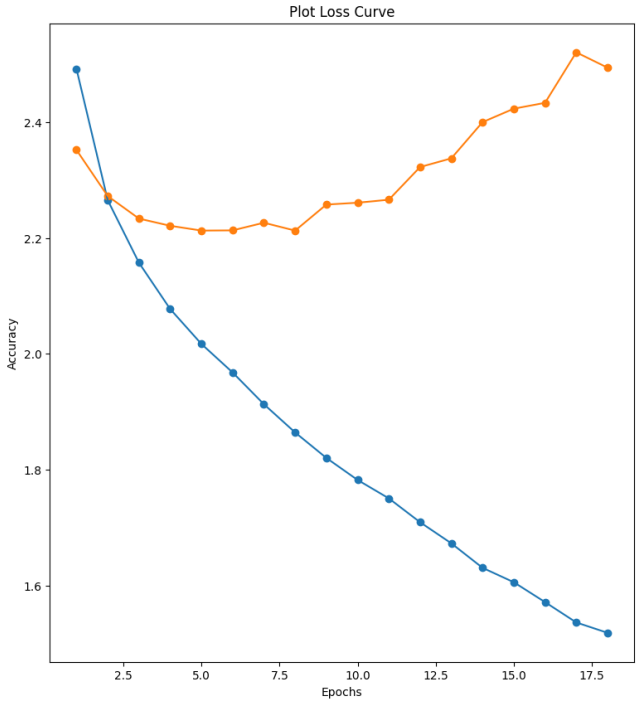
In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = Flatten()(x)
x = Dense(128, 'relu')(x)  # Hidden Layer 1
x = Dense(128, 'relu')(x)  # Hidden Layer 3
x = Dense(128, 'relu')(x)  # Hidden Layer 3
x = Dense(NUM_CLASS, 'softmax')(x)
baseModel = Model(inputs=inputs, outputs=x, name="baseline")
baseModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                  loss='categorical_crossentropy', metrics=['accuracy'])
```

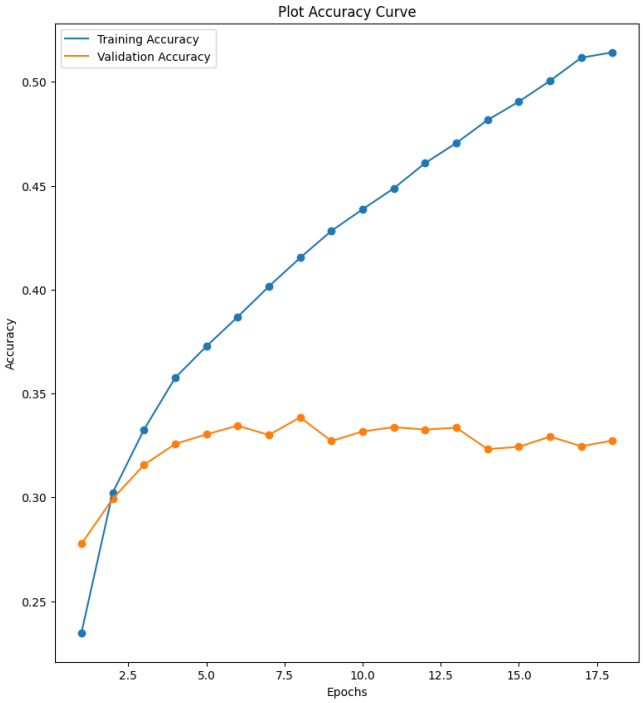
In [ ]:
```
baseModelHistory = baseModel.fit(x_train, y_train, epochs=100,
                                 validation_data=(x_val, y_val), batch_size=BATCH_SIZE, callb
```

```
Epoch 1/100
625/625 [==============================] - 5s 7ms/step - loss: 2.4924 - accuracy: 0.2348 - va
l_loss: 2.3523 - val_accuracy: 0.2776
Epoch 2/100
625/625 [==============================] - 4s 6ms/step - loss: 2.2656 - accuracy: 0.3022 - va
l_loss: 2.2726 - val_accuracy: 0.2993
Epoch 3/100
625/625 [==============================] - 3s 5ms/step - loss: 2.1574 - accuracy: 0.3325 - va
l_loss: 2.2337 - val_accuracy: 0.3156
Epoch 4/100
625/625 [==============================] - 3s 5ms/step - loss: 2.0774 - accuracy: 0.3575 - va
l_loss: 2.2212 - val_accuracy: 0.3257
Epoch 5/100
625/625 [==============================] - 3s 5ms/step - loss: 2.0171 - accuracy: 0.3727 - va
l_loss: 2.2129 - val_accuracy: 0.3303
Epoch 6/100
625/625 [==============================] - 3s 5ms/step - loss: 1.9677 - accuracy: 0.3868 - va
l_loss: 2.2133 - val_accuracy: 0.3345
Epoch 7/100
625/625 [==============================] - 3s 5ms/step - loss: 1.9129 - accuracy: 0.4015 - va
l_loss: 2.2265 - val_accuracy: 0.3300
Epoch 8/100
625/625 [==============================] - 3s 5ms/step - loss: 1.8642 - accuracy: 0.4153 - va
l_loss: 2.2129 - val_accuracy: 0.3385
Epoch 9/100
625/625 [==============================] - 3s 5ms/step - loss: 1.8201 - accuracy: 0.4282 - va
l_loss: 2.2579 - val_accuracy: 0.3271
Epoch 10/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7821 - accuracy: 0.4387 - va
l_loss: 2.2611 - val_accuracy: 0.3317
Epoch 11/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7503 - accuracy: 0.4487 - va
l_loss: 2.2662 - val_accuracy: 0.3338
Epoch 12/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7091 - accuracy: 0.4609 - va
l_loss: 2.3227 - val_accuracy: 0.3326
Epoch 13/100
625/625 [==============================] - 3s 5ms/step - loss: 1.6726 - accuracy: 0.4706 - va
l_loss: 2.3377 - val_accuracy: 0.3335
Epoch 14/100
625/625 [==============================] - 3s 5ms/step - loss: 1.6303 - accuracy: 0.4816 - va
l_loss: 2.4003 - val_accuracy: 0.3232
Epoch 15/100
625/625 [==============================] - 3s 5ms/step - loss: 1.6054 - accuracy: 0.4904 - va
l_loss: 2.4235 - val_accuracy: 0.3243
Epoch 16/100
625/625 [==============================] - 3s 5ms/step - loss: 1.5711 - accuracy: 0.5005 - va
l_loss: 2.4336 - val_accuracy: 0.3292
Epoch 17/100
625/625 [==============================] - 3s 5ms/step - loss: 1.5359 - accuracy: 0.5116 - va
l_loss: 2.5209 - val_accuracy: 0.3246
Epoch 18/100
625/625 [==============================] - 3s 5ms/step - loss: 1.5180 - accuracy: 0.5142 - va
l_loss: 2.4944 - val_accuracy: 0.3273
```

In [ ]:
```
print(storeResult(baseModelHistory))
plot_loss_curve(baseModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'baseline', 'Epochs': 18, 'Batch Size': 64, 'Train Loss': 1.8641740083694458,
'Val Loss': 2.212881326675415, 'Train Acc': 0.41530001163482666, 'Val Acc': 0.338499993085861
2, '[Train - Val] Acc': 0.07680001854896545}
```





Observations

From the loss curve, We can see that as the model increase in epochs, the model becomes more generalise and the loss functions starts decreasing too. However, the accuracy of both training and validation is very low at 41.5% and 33.8%. This means that the model is not very strong at predicting and it is similar to randomly choosing a label. Let's see if augmentations will improve the accuracy.

Training baseline model with Data Augmentation

As mentioned previously, we will train the baseline model with the dataset that was augmented.

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = Flatten()(x)
x = Dense(128, 'relu')(x)  # Hidden Layer 1
x = Dense(128, 'relu')(x)  # Hidden Layer 3
x = Dense(128, 'relu')(x)  # Hidden Layer 3
x = Dense(NUM_CLASS, 'softmax')(x)
baseAugModel = Model(inputs=inputs, outputs=x, name="baselineAug")
baseAugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                     loss='categorical_crossentropy', metrics=['accuracy'])
```

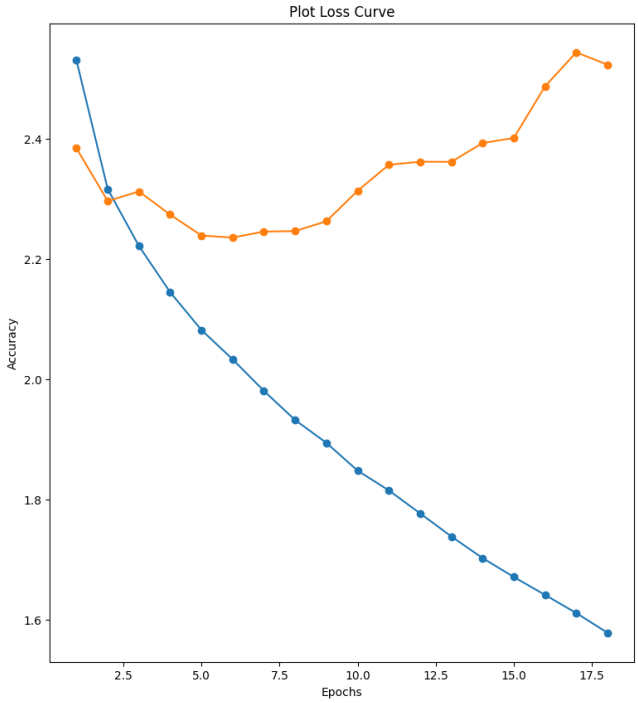
In [ ]:
```
baseAugModelHistory = baseAugModel.fit(x_train_aug, y_train, epochs=100,
                                       validation_data=(x_val, y_val), batch_size=BATCH_SIZE, callb
```

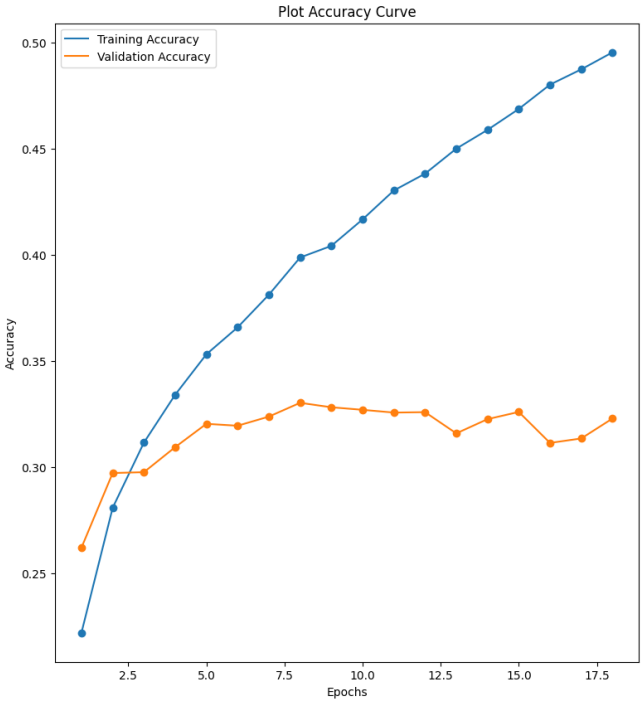
```
Epoch 1/100
625/625 [==============================] - 4s 6ms/step - loss: 2.5300 - accuracy: 0.2219 - va
l_loss: 2.3840 - val_accuracy: 0.2620
Epoch 2/100
625/625 [==============================] - 4s 6ms/step - loss: 2.3160 - accuracy: 0.2808 - va
l_loss: 2.2960 - val_accuracy: 0.2972
Epoch 3/100
625/625 [==============================] - 3s 5ms/step - loss: 2.2213 - accuracy: 0.3115 - va
l_loss: 2.3120 - val_accuracy: 0.2976
Epoch 4/100
625/625 [==============================] - 3s 5ms/step - loss: 2.1444 - accuracy: 0.3341 - va
l_loss: 2.2734 - val_accuracy: 0.3094
Epoch 5/100
625/625 [==============================] - 3s 5ms/step - loss: 2.0817 - accuracy: 0.3532 - va
l_loss: 2.2386 - val_accuracy: 0.3204
Epoch 6/100
625/625 [==============================] - 3s 5ms/step - loss: 2.0328 - accuracy: 0.3659 - va
l_loss: 2.2353 - val_accuracy: 0.3195
Epoch 7/100
625/625 [==============================] - 3s 5ms/step - loss: 1.9804 - accuracy: 0.3812 - va
l_loss: 2.2453 - val_accuracy: 0.3238
Epoch 8/100
625/625 [==============================] - 3s 5ms/step - loss: 1.9319 - accuracy: 0.3988 - va
l_loss: 2.2461 - val_accuracy: 0.3303
Epoch 9/100
625/625 [==============================] - 3s 5ms/step - loss: 1.8939 - accuracy: 0.4043 - va
l_loss: 2.2624 - val_accuracy: 0.3282
Epoch 10/100
625/625 [==============================] - 3s 5ms/step - loss: 1.8477 - accuracy: 0.4168 - va
l_loss: 2.3130 - val_accuracy: 0.3270
Epoch 11/100
625/625 [==============================] - 3s 5ms/step - loss: 1.8148 - accuracy: 0.4304 - va
l_loss: 2.3563 - val_accuracy: 0.3257
Epoch 12/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7767 - accuracy: 0.4383 - va
l_loss: 2.3614 - val_accuracy: 0.3259
Epoch 13/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7383 - accuracy: 0.4501 - va
l_loss: 2.3612 - val_accuracy: 0.3159
Epoch 14/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7023 - accuracy: 0.4590 - va
l_loss: 2.3925 - val_accuracy: 0.3226
Epoch 15/100
625/625 [==============================] - 3s 5ms/step - loss: 1.6708 - accuracy: 0.4688 - va
l_loss: 2.4009 - val_accuracy: 0.3260
Epoch 16/100
625/625 [==============================] - 3s 5ms/step - loss: 1.6411 - accuracy: 0.4803 - va
l_loss: 2.4868 - val_accuracy: 0.3114
Epoch 17/100
625/625 [==============================] - 3s 5ms/step - loss: 1.6108 - accuracy: 0.4875 - va
l_loss: 2.5433 - val_accuracy: 0.3135
Epoch 18/100
625/625 [==============================] - 3s 5ms/step - loss: 1.5780 - accuracy: 0.4955 - va
l_loss: 2.5228 - val_accuracy: 0.3229
```

In [ ]:
```
print(storeResult(baseAugModelHistory))
plot_loss_curve(baseAugModelHistory)
plt.show()
```

```
{'Model Name': 'baselineAug', 'Epochs': 18, 'Batch Size': 64, 'Train Loss': 1.931877970695495
6, 'Val Loss': 2.246060371398926, 'Train Acc': 0.39879998564720154, 'Val Acc': 0.330300003290
1764, '[Train - Val] Acc': 0.06849998235702515}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```





Observations

We can also see that by augmenting the data, we see that even though the accuracy for both training and validation decreased, there is a slight increase in validation loss which means that the model is becoming less generalise to fit to the dataset. However, as the accuracy is still very low, more improvements need to be made to make the model better.

Conv2D Neural Network Model

After creating our baseline model, we begin making more complex models. We will be building a simple convolutional neural network (CNN). We will be using tensorflow's Conv2D layers to build the models. The reason why we use a CNN architecture is because CNNs are well suited to solve the problem of image classification. This is because the convolution layers consider the context in the local neighbourhood of the input data and constructs features from the neighbourhood. CNNs also reduce the number of parameters in the network due to its sparse connections and weight sharing properties.

Training conv2D model without Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = Conv2D(32, (5, 5), input_shape=IMG_SIZE, activation='relu')(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.2)(x)
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
x = Dense(NUM_CLASS, 'softmax')(x)
conv2DModel = Model(inputs=inputs, outputs=x, name="conv2D")
conv2DModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                    loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
conv2DModel.summary()
```

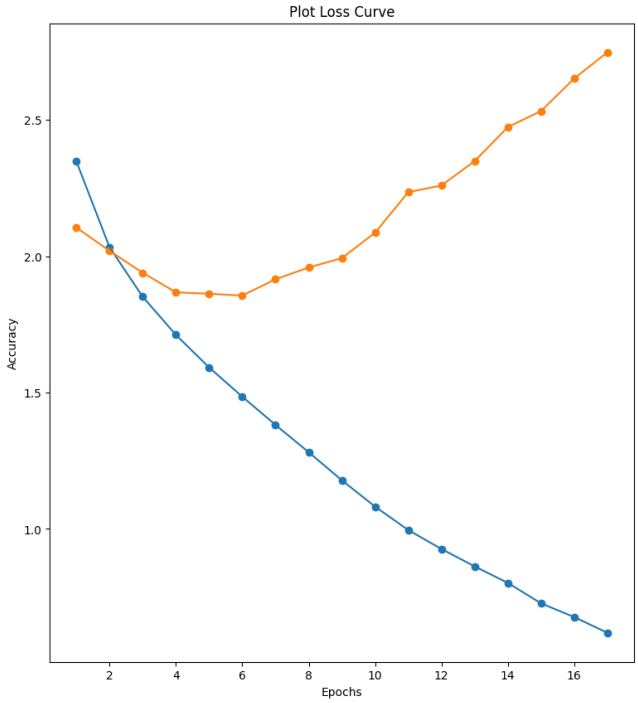
```
Model: "conv2D"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 normalization (Normalizatio  (None, 32, 32, 3)        7
 n)

 conv2d (Conv2D)             (None, 28, 28, 32)        2432

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 dropout (Dropout)           (None, 14, 14, 32)        0

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 128)               802944

 dense_1 (Dense)             (None, 20)                2580

=================================================================
Total params: 807,963
Trainable params: 807,956
Non-trainable params: 7
_________________________________________________________________
```

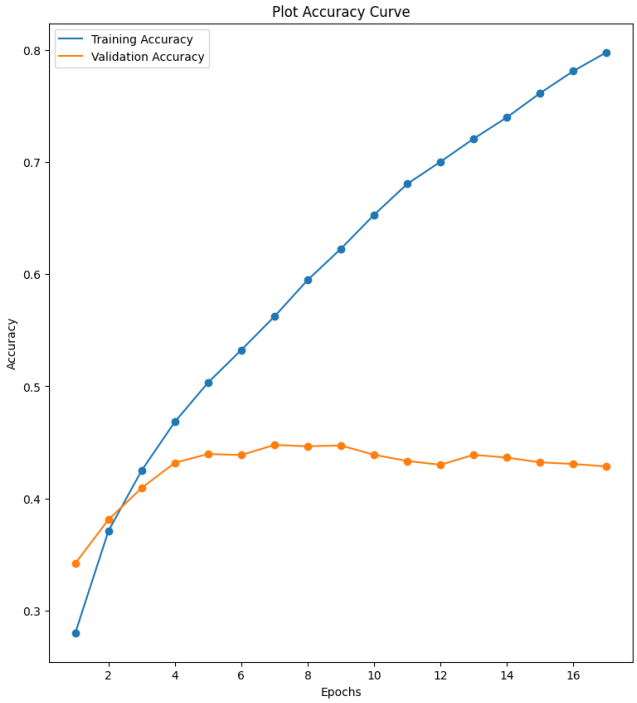
In [ ]:
```
conv2DModelHistory = conv2DModel.fit(x_train, y_train, epochs=100,
                                     validation_data=(x_val, y_val), batch_size=BATCH_SIZE, c
```

```
Epoch 1/100
625/625 [==============================] - 6s 7ms/step - loss: 2.3483 - accuracy: 0.2802 - va
l_loss: 2.1053 - val_accuracy: 0.3421
Epoch 2/100
625/625 [==============================] - 4s 6ms/step - loss: 2.0330 - accuracy: 0.3711 - va
l_loss: 2.0199 - val_accuracy: 0.3812
Epoch 3/100
625/625 [==============================] - 3s 5ms/step - loss: 1.8514 - accuracy: 0.4250 - va
l_loss: 1.9394 - val_accuracy: 0.4094
Epoch 4/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7116 - accuracy: 0.4683 - va
l_loss: 1.8677 - val_accuracy: 0.4318
Epoch 5/100
625/625 [==============================] - 3s 6ms/step - loss: 1.5923 - accuracy: 0.5033 - va
l_loss: 1.8625 - val_accuracy: 0.4396
Epoch 6/100
625/625 [==============================] - 4s 6ms/step - loss: 1.4849 - accuracy: 0.5324 - va
l_loss: 1.8559 - val_accuracy: 0.4387
Epoch 7/100
625/625 [==============================] - 4s 6ms/step - loss: 1.3815 - accuracy: 0.5623 - va
l_loss: 1.9157 - val_accuracy: 0.4477
Epoch 8/100
625/625 [==============================] - 3s 6ms/step - loss: 1.2813 - accuracy: 0.5948 - va
l_loss: 1.9589 - val_accuracy: 0.4465
Epoch 9/100
625/625 [==============================] - 3s 5ms/step - loss: 1.1784 - accuracy: 0.6226 - va
l_loss: 1.9932 - val_accuracy: 0.4472
Epoch 10/100
625/625 [==============================] - 4s 7ms/step - loss: 1.0816 - accuracy: 0.6528 - va
l_loss: 2.0866 - val_accuracy: 0.4389
Epoch 11/100
625/625 [==============================] - 3s 6ms/step - loss: 0.9959 - accuracy: 0.6805 - va
l_loss: 2.2351 - val_accuracy: 0.4334
Epoch 12/100
625/625 [==============================] - 4s 6ms/step - loss: 0.9263 - accuracy: 0.7002 - va
l_loss: 2.2595 - val_accuracy: 0.4300
Epoch 13/100
625/625 [==============================] - 3s 6ms/step - loss: 0.8625 - accuracy: 0.7207 - va
l_loss: 2.3490 - val_accuracy: 0.4389
Epoch 14/100
625/625 [==============================] - 3s 6ms/step - loss: 0.8019 - accuracy: 0.7397 - va
l_loss: 2.4740 - val_accuracy: 0.4364
Epoch 15/100
625/625 [==============================] - 3s 6ms/step - loss: 0.7270 - accuracy: 0.7613 - va
l_loss: 2.5326 - val_accuracy: 0.4322
Epoch 16/100
625/625 [==============================] - 3s 6ms/step - loss: 0.6767 - accuracy: 0.7810 - va
l_loss: 2.6522 - val_accuracy: 0.4307
Epoch 17/100
625/625 [==============================] - 3s 6ms/step - loss: 0.6189 - accuracy: 0.7978 - va
l_loss: 2.7480 - val_accuracy: 0.4285
```

In [ ]:
```
print(storeResult(conv2DModelHistory))
plot_loss_curve(conv2DModelHistory)
plt.show()
```

```
{'Model Name': 'conv2D', 'Epochs': 17, 'Batch Size': 64, 'Train Loss': 1.3815046548843384, 'V
al Loss': 1.9156562089920044, 'Train Acc': 0.562250018119812, 'Val Acc': 0.44769999384880066,
'[Train - Val] Acc': 0.11455002427101135}
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```





Observation

Comparing the Conv2D model with the baseline model, we can see that there is a significant difference in the training accuracy and validation accuracy. The loss functions decrease suggesting the model is performing better and generalising more

Training conv2D model with Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = Conv2D(32, (5, 5), input_shape=IMG_SIZE, activation='relu')(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.2)(x)
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
x = Dense(NUM_CLASS, 'softmax')(x)
conv2DAugModel = Model(inputs=inputs, outputs=x, name="conv2DAug")
conv2DAugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                       loss='categorical_crossentropy', metrics=['accuracy'])
```

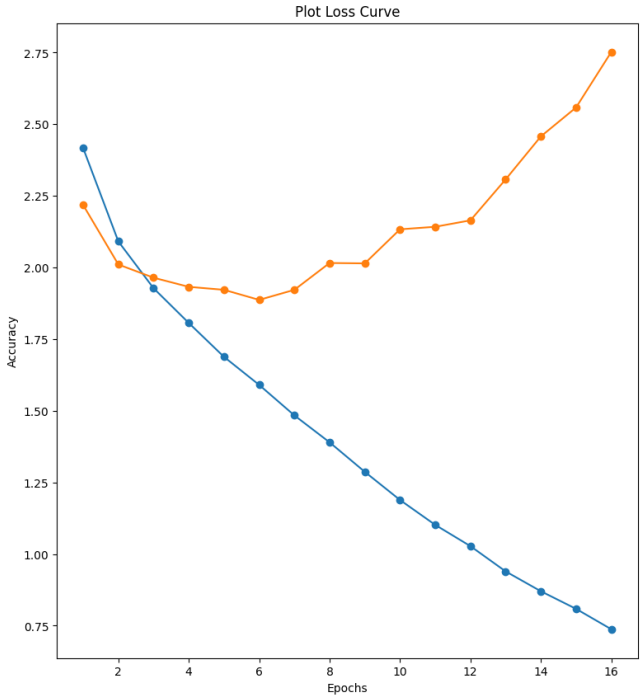
In [ ]:
```
conv2DAugModelHistory = conv2DAugModel.fit(x_train_aug, y_train, epochs=100,
                                           validation_data=(x_val, y_val), batch_size=BATCH_SIZE, c
```

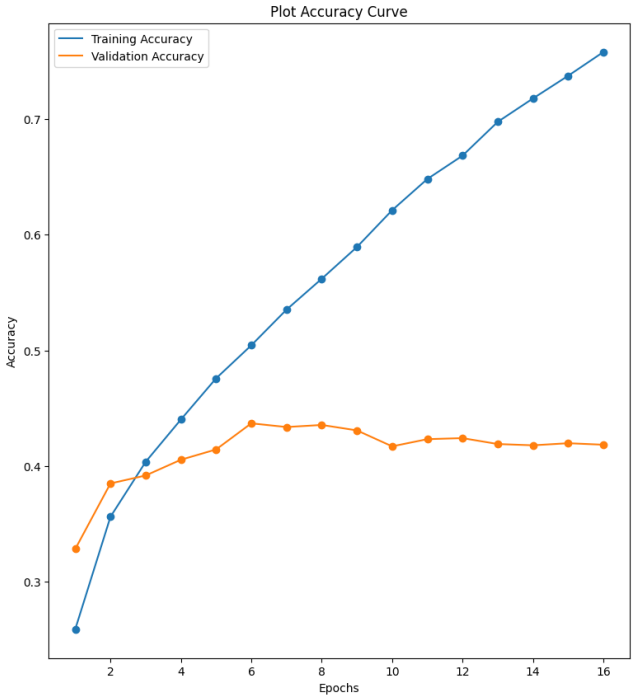
```
Epoch 1/100
625/625 [==============================] - 5s 7ms/step - loss: 2.4169 - accuracy: 0.2588 - va
l_loss: 2.2181 - val_accuracy: 0.3282
Epoch 2/100
625/625 [==============================] - 4s 6ms/step - loss: 2.0909 - accuracy: 0.3561 - va
l_loss: 2.0105 - val_accuracy: 0.3849
Epoch 3/100
625/625 [==============================] - 4s 6ms/step - loss: 1.9277 - accuracy: 0.4036 - va
l_loss: 1.9645 - val_accuracy: 0.3918
Epoch 4/100
625/625 [==============================] - 3s 6ms/step - loss: 1.8065 - accuracy: 0.4402 - va
l_loss: 1.9325 - val_accuracy: 0.4054
Epoch 5/100
625/625 [==============================] - 3s 6ms/step - loss: 1.6882 - accuracy: 0.4758 - va
l_loss: 1.9222 - val_accuracy: 0.4143
Epoch 6/100
625/625 [==============================] - 4s 6ms/step - loss: 1.5902 - accuracy: 0.5044 - va
l_loss: 1.8870 - val_accuracy: 0.4369
Epoch 7/100
625/625 [==============================] - 4s 6ms/step - loss: 1.4838 - accuracy: 0.5352 - va
l_loss: 1.9222 - val_accuracy: 0.4337
Epoch 8/100
625/625 [==============================] - 4s 6ms/step - loss: 1.3899 - accuracy: 0.5619 - va
l_loss: 2.0158 - val_accuracy: 0.4355
Epoch 9/100
625/625 [==============================] - 4s 6ms/step - loss: 1.2869 - accuracy: 0.5894 - va
l_loss: 2.0144 - val_accuracy: 0.4308
Epoch 10/100
625/625 [==============================] - 4s 6ms/step - loss: 1.1886 - accuracy: 0.6212 - va
l_loss: 2.1333 - val_accuracy: 0.4170
Epoch 11/100
625/625 [==============================] - 3s 6ms/step - loss: 1.1018 - accuracy: 0.6482 - va
l_loss: 2.1422 - val_accuracy: 0.4232
Epoch 12/100
625/625 [==============================] - 3s 6ms/step - loss: 1.0276 - accuracy: 0.6684 - va
l_loss: 2.1646 - val_accuracy: 0.4241
Epoch 13/100
625/625 [==============================] - 3s 6ms/step - loss: 0.9394 - accuracy: 0.6977 - va
l_loss: 2.3076 - val_accuracy: 0.4190
Epoch 14/100
625/625 [==============================] - 3s 5ms/step - loss: 0.8701 - accuracy: 0.7179 - va
l_loss: 2.4575 - val_accuracy: 0.4179
Epoch 15/100
625/625 [==============================] - 3s 5ms/step - loss: 0.8087 - accuracy: 0.7375 - va
l_loss: 2.5584 - val_accuracy: 0.4197
Epoch 16/100
625/625 [==============================] - 4s 6ms/step - loss: 0.7376 - accuracy: 0.7581 - va
l_loss: 2.7521 - val_accuracy: 0.4184
```

In [ ]:

```
print(storeResult(conv2DAugModelHistory))
plot_loss_curve(conv2DAugModelHistory)
plt.show()
```

```
{'Model Name': 'conv2DAug', 'Epochs': 16, 'Batch Size': 64, 'Train Loss': 1.590235948562622,
'Val Loss': 1.887040376663208, 'Train Acc': 0.5043749809265137, 'Val Acc': 0.4368999898433685
3, '[Train - Val] Acc': 0.06747499108314514}
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```



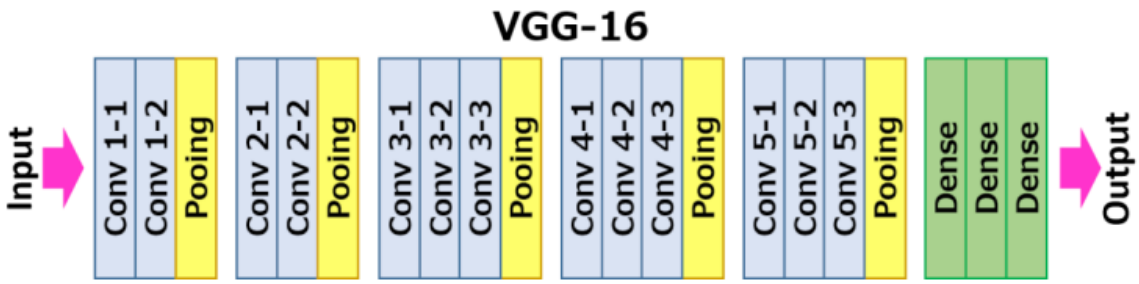


Observation

Comparing the augmented data trained under the conv2D model, we can see that although both training accuracy and validation accuracy decreased. But the model seem to be more generalise due to the loss functions decreasing. But more should be done.

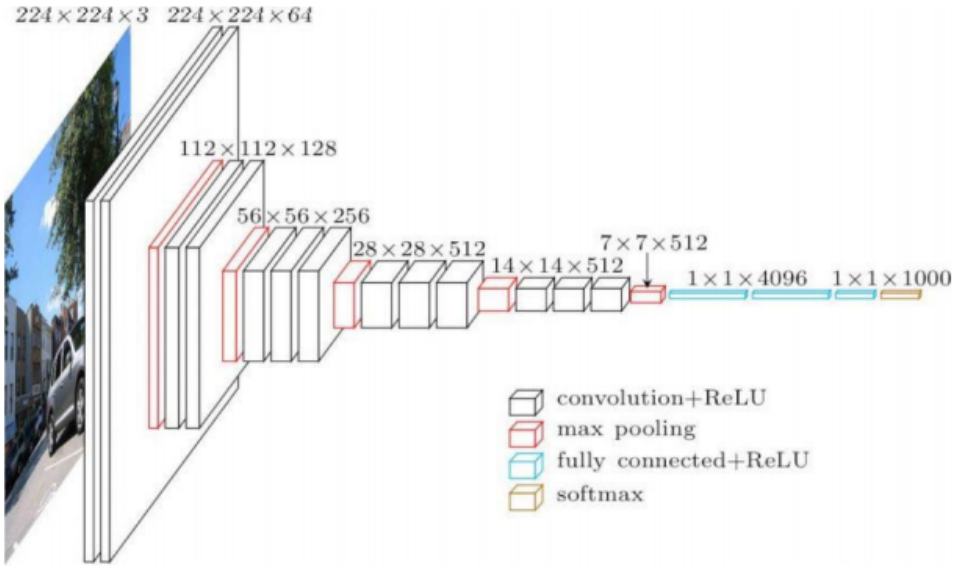
CustomVGG Model

VGG-16 is a convolutional neural network that is 16 layers deep.



The Architecture

The architecture depicted below is VGG16.



Building the Custom VGG model

From the main VGG16 model, we can see that the VGG network is build based on blocks. Each block contains 2/3 layers of Conv2D and a MaxPooling2D layer. We will build it based on the [https://d2l.ai/chapter_convolutional-modern/vgg.html#]. After the main VGG block has been created, there is a flatten layer followed by 2 fully connected neural networks [relu] which helps the model reach the output layer [softmax].

In [ ]:
```
def vgg_block(num_convs, num_channels):
    blk = Sequential()
    for _ in range(num_convs):
        blk.add(
            Conv2D(num_channels, kernel_size=3,
                   padding='same', activation='relu'))
        blk.add(
            BatchNormalization())
    blk.add(MaxPool2D(pool_size=2, strides=2))
    return blk
```

Training CustomVGG model without Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = vgg_block(2, 32)(x)  # we are using less filters compared to VGG16
x = vgg_block(2, 64)(x)
x = vgg_block(3, 128)(x)
x = vgg_block(3, 256)(x)
x = GlobalAveragePooling2D()(x)
x = Dense(256, 'relu')(x)
```

```
x = Dropout(0.3)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGGModel = Model(inputs=inputs, outputs=x, name="CustomVGG")
customVGGModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                       loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:

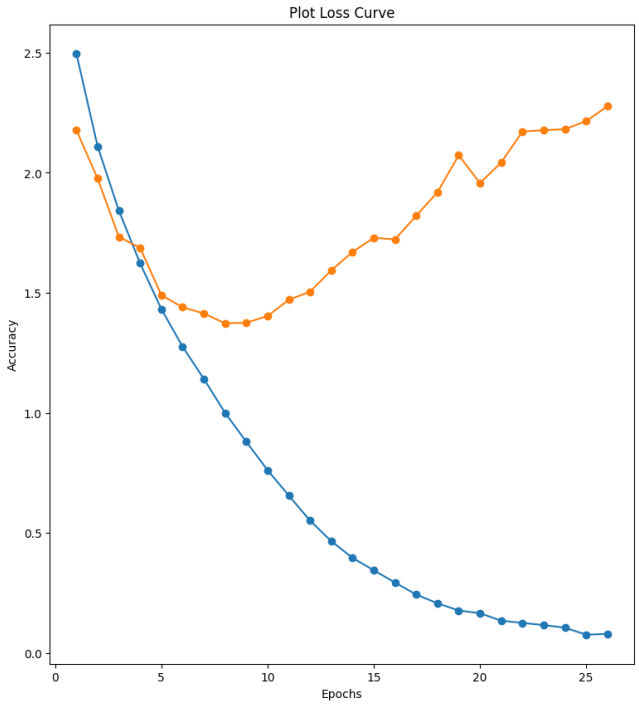
```
customVGGModelHistory = customVGGModel.fit(x_train, y_train, epochs=50,
                                           validation_data=(x_val, y_val), batch_size=BATCH_S
```

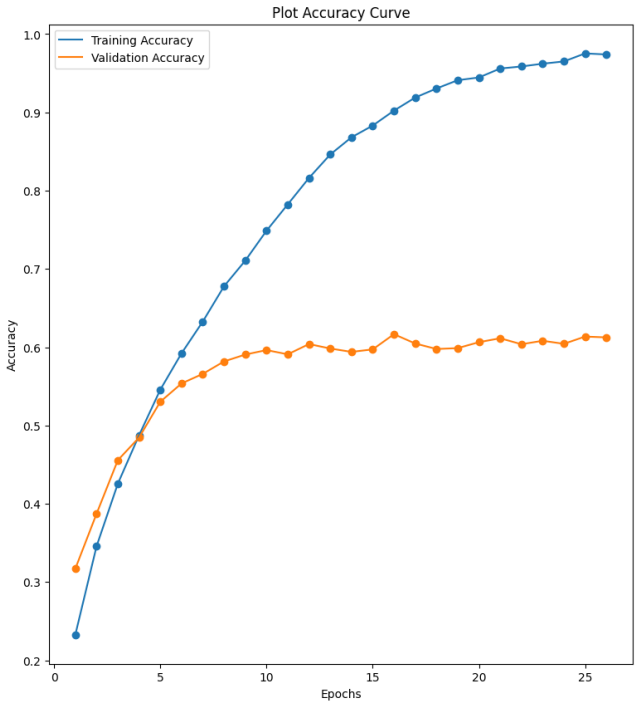
```
Epoch 1/50
625/625 [==============================] - 15s 21ms/step - loss: 2.4969 - accuracy: 0.2327 -
val_loss: 2.1778 - val_accuracy: 0.3171
Epoch 2/50
625/625 [==============================] - 12s 19ms/step - loss: 2.1101 - accuracy: 0.3458 -
val_loss: 1.9761 - val_accuracy: 0.3873
Epoch 3/50
625/625 [==============================] - 12s 19ms/step - loss: 1.8423 - accuracy: 0.4256 -
val_loss: 1.7330 - val_accuracy: 0.4556
Epoch 4/50
625/625 [==============================] - 12s 19ms/step - loss: 1.6233 - accuracy: 0.4875 -
val_loss: 1.6876 - val_accuracy: 0.4846
Epoch 5/50
625/625 [==============================] - 12s 20ms/step - loss: 1.4325 - accuracy: 0.5457 -
val_loss: 1.4907 - val_accuracy: 0.5303
Epoch 6/50
625/625 [==============================] - 12s 20ms/step - loss: 1.2759 - accuracy: 0.5922 -
val_loss: 1.4395 - val_accuracy: 0.5537
Epoch 7/50
625/625 [==============================] - 12s 20ms/step - loss: 1.1419 - accuracy: 0.6327 -
val_loss: 1.4141 - val_accuracy: 0.5659
Epoch 8/50
625/625 [==============================] - 13s 20ms/step - loss: 1.0002 - accuracy: 0.6778 -
val_loss: 1.3731 - val_accuracy: 0.5818
Epoch 9/50
625/625 [==============================] - 12s 20ms/step - loss: 0.8799 - accuracy: 0.7106 -
val_loss: 1.3755 - val_accuracy: 0.5906
Epoch 10/50
625/625 [==============================] - 13s 20ms/step - loss: 0.7616 - accuracy: 0.7488 -
val_loss: 1.4030 - val_accuracy: 0.5963
Epoch 11/50
625/625 [==============================] - 14s 22ms/step - loss: 0.6558 - accuracy: 0.7824 -
val_loss: 1.4709 - val_accuracy: 0.5909
Epoch 12/50
625/625 [==============================] - 13s 21ms/step - loss: 0.5513 - accuracy: 0.8162 -
val_loss: 1.5043 - val_accuracy: 0.6040
Epoch 13/50
625/625 [==============================] - 15s 23ms/step - loss: 0.4660 - accuracy: 0.8461 -
val_loss: 1.5934 - val_accuracy: 0.5983
Epoch 14/50
625/625 [==============================] - 13s 21ms/step - loss: 0.3962 - accuracy: 0.8681 -
val_loss: 1.6703 - val_accuracy: 0.5940
Epoch 15/50
625/625 [==============================] - 13s 21ms/step - loss: 0.3446 - accuracy: 0.8831 -
val_loss: 1.7289 - val_accuracy: 0.5972
Epoch 16/50
625/625 [==============================] - 13s 21ms/step - loss: 0.2934 - accuracy: 0.9022 -
val_loss: 1.7225 - val_accuracy: 0.6166
Epoch 17/50
625/625 [==============================] - 13s 21ms/step - loss: 0.2434 - accuracy: 0.9189 -
val_loss: 1.8208 - val_accuracy: 0.6047
Epoch 18/50
625/625 [==============================] - 13s 21ms/step - loss: 0.2070 - accuracy: 0.9305 -
val_loss: 1.9187 - val_accuracy: 0.5977
Epoch 19/50
625/625 [==============================] - 13s 21ms/step - loss: 0.1770 - accuracy: 0.9410 -
val_loss: 2.0729 - val_accuracy: 0.5988
Epoch 20/50
625/625 [==============================] - 13s 21ms/step - loss: 0.1658 - accuracy: 0.9445 -
val_loss: 1.9572 - val_accuracy: 0.6064
Epoch 21/50
625/625 [==============================] - 13s 21ms/step - loss: 0.1345 - accuracy: 0.9560 -
val_loss: 2.0423 - val_accuracy: 0.6114
Epoch 22/50
625/625 [==============================] - 13s 22ms/step - loss: 0.1253 - accuracy: 0.9586 -
val_loss: 2.1718 - val_accuracy: 0.6038
```

```
Epoch 23/50
625/625 [==============================] - 13s 21ms/step - loss: 0.1163 - accuracy: 0.9621 -
val_loss: 2.1771 - val_accuracy: 0.6082
Epoch 24/50
625/625 [==============================] - 13s 21ms/step - loss: 0.1055 - accuracy: 0.9650 -
val_loss: 2.1814 - val_accuracy: 0.6043
Epoch 25/50
625/625 [==============================] - 13s 21ms/step - loss: 0.0763 - accuracy: 0.9753 -
val_loss: 2.2147 - val_accuracy: 0.6136
Epoch 26/50
625/625 [==============================] - 13s 22ms/step - loss: 0.0795 - accuracy: 0.9740 -
val_loss: 2.2769 - val_accuracy: 0.6125
```

In [ ]:
```
print(storeResult(customVGGModelHistory))
plot_loss_curve(customVGGModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'CustomVGG', 'Epochs': 26, 'Batch Size': 64, 'Train Loss': 0.2934300899505615,
'Val Loss': 1.722504734992981, 'Train Acc': 0.9021999835968018, 'Val Acc': 0.616599977016449,
'[Train - Val] Acc': 0.2856000065803528}
```





Observations

Comparing our baseline model and customVGG model, we can see that the customVGG model is very overfitted as the validation loss is super high while training loss is super low. We need to do data augmentation etc to reduce overfitting.

Training CustomVGG model without Data Augmentation and using L1 Lasso Regularisation

In [ ]:
```
def vgg_block_l1(num_convs, num_channels, weight_decay=0.0005):
    blk = Sequential()
    for _ in range(num_convs):
        blk.add(
            Conv2D(num_channels, kernel_size=3,
                   padding='same', activation='relu', kernel_regularizer=l1(weight_decay)))
        blk.add(
            BatchNormalization())
```

```
    blk.add(MaxPool2D(pool_size=2, strides=2))
    return blk
```

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = vgg_block_l1(2, 32)(x)  # we are using less filters compared to VGG16
x = vgg_block_l1(2, 64)(x)
x = vgg_block_l1(3, 128)(x)
x = vgg_block_l1(3, 256)(x)
x = GlobalAveragePooling2D()(x)
x = Dense(256, 'relu')(x)
x = Dropout(0.3)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGGL1Model = Model(inputs=inputs, outputs=x, name="CustomVGG_L1")
customVGGL1Model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                         loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
customVGGL1ModelHistory = customVGGL1Model.fit(x_train, y_train, epochs=50,
                                               validation_data=(x_val, y_val), batch_size=BAT
```

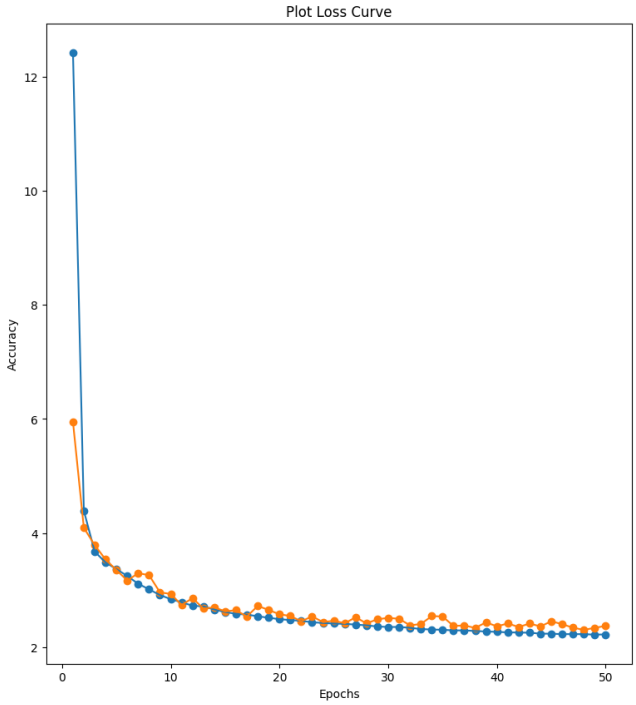
```
Epoch 1/50
625/625 [==============================] - 19s 27ms/step - loss: 12.4283 - accuracy: 0.2304 -
val_loss: 5.9503 - val_accuracy: 0.1721
Epoch 2/50
625/625 [==============================] - 15s 24ms/step - loss: 4.3914 - accuracy: 0.2458 -
val_loss: 4.0914 - val_accuracy: 0.2101
Epoch 3/50
625/625 [==============================] - 15s 24ms/step - loss: 3.6748 - accuracy: 0.2575 -
val_loss: 3.7883 - val_accuracy: 0.2306
Epoch 4/50
625/625 [==============================] - 15s 24ms/step - loss: 3.4840 - accuracy: 0.2903 -
val_loss: 3.5357 - val_accuracy: 0.2879
Epoch 5/50
625/625 [==============================] - 15s 24ms/step - loss: 3.3663 - accuracy: 0.3163 -
val_loss: 3.3503 - val_accuracy: 0.3140
Epoch 6/50
625/625 [==============================] - 15s 24ms/step - loss: 3.2469 - accuracy: 0.3437 -
val_loss: 3.1672 - val_accuracy: 0.3465
Epoch 7/50
625/625 [==============================] - 15s 24ms/step - loss: 3.1094 - accuracy: 0.3655 -
val_loss: 3.2955 - val_accuracy: 0.3214
Epoch 8/50
625/625 [==============================] - 15s 24ms/step - loss: 3.0131 - accuracy: 0.3832 -
val_loss: 3.2615 - val_accuracy: 0.3207
Epoch 9/50
625/625 [==============================] - 16s 25ms/step - loss: 2.9117 - accuracy: 0.3948 -
val_loss: 2.9603 - val_accuracy: 0.3799
Epoch 10/50
625/625 [==============================] - 15s 24ms/step - loss: 2.8403 - accuracy: 0.4106 -
val_loss: 2.9304 - val_accuracy: 0.3839
Epoch 11/50
625/625 [==============================] - 16s 25ms/step - loss: 2.7846 - accuracy: 0.4253 -
val_loss: 2.7365 - val_accuracy: 0.4345
Epoch 12/50
625/625 [==============================] - 16s 25ms/step - loss: 2.7256 - accuracy: 0.4364 -
val_loss: 2.8596 - val_accuracy: 0.4007
Epoch 13/50
625/625 [==============================] - 15s 25ms/step - loss: 2.7127 - accuracy: 0.4433 -
val_loss: 2.6744 - val_accuracy: 0.4333
Epoch 14/50
625/625 [==============================] - 15s 24ms/step - loss: 2.6577 - accuracy: 0.4446 -
val_loss: 2.6963 - val_accuracy: 0.4333
Epoch 15/50
625/625 [==============================] - 15s 24ms/step - loss: 2.6142 - accuracy: 0.4582 -
val_loss: 2.6257 - val_accuracy: 0.4674
Epoch 16/50
625/625 [==============================] - 15s 24ms/step - loss: 2.5836 - accuracy: 0.4635 -
val_loss: 2.6461 - val_accuracy: 0.4494
Epoch 17/50
625/625 [==============================] - 15s 24ms/step - loss: 2.5693 - accuracy: 0.4692 -
val_loss: 2.5357 - val_accuracy: 0.4757
Epoch 18/50
625/625 [==============================] - 15s 24ms/step - loss: 2.5402 - accuracy: 0.4759 -
val_loss: 2.7222 - val_accuracy: 0.4365
Epoch 19/50
625/625 [==============================] - 15s 24ms/step - loss: 2.5141 - accuracy: 0.4808 -
val_loss: 2.6503 - val_accuracy: 0.4477
Epoch 20/50
625/625 [==============================] - 15s 24ms/step - loss: 2.4886 - accuracy: 0.4875 -
val_loss: 2.5747 - val_accuracy: 0.4643
Epoch 21/50
625/625 [==============================] - 15s 24ms/step - loss: 2.4742 - accuracy: 0.4923 -
val_loss: 2.5445 - val_accuracy: 0.4745
Epoch 22/50
625/625 [==============================] - 15s 24ms/step - loss: 2.4600 - accuracy: 0.4938 -
val_loss: 2.4518 - val_accuracy: 0.4906
```

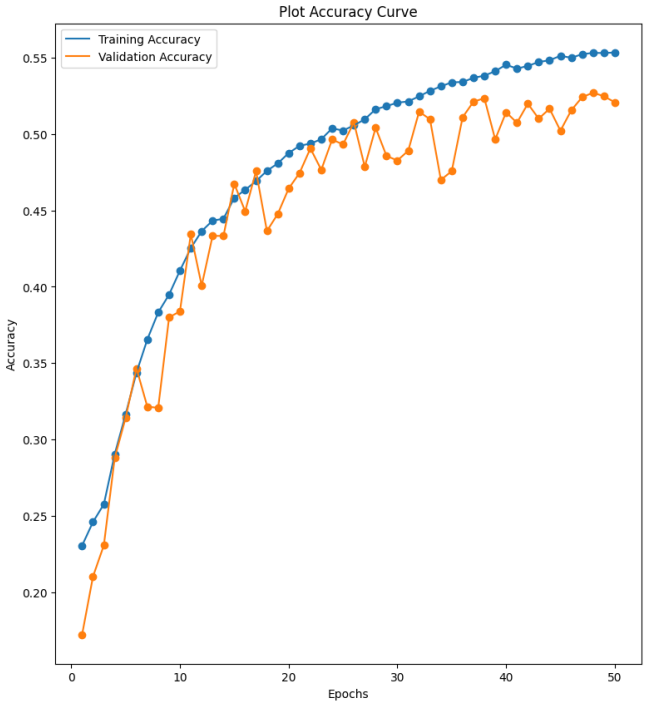
```
Epoch 23/50
625/625 [==============================] - 15s 25ms/step - loss: 2.4355 - accuracy: 0.4967 -
val_loss: 2.5381 - val_accuracy: 0.4766
Epoch 24/50
625/625 [==============================] - 15s 24ms/step - loss: 2.4223 - accuracy: 0.5038 -
val_loss: 2.4371 - val_accuracy: 0.4964
Epoch 25/50
625/625 [==============================] - 15s 24ms/step - loss: 2.4116 - accuracy: 0.5023 -
val_loss: 2.4625 - val_accuracy: 0.4932
Epoch 26/50
625/625 [==============================] - 15s 25ms/step - loss: 2.4069 - accuracy: 0.5057 -
val_loss: 2.4167 - val_accuracy: 0.5077
Epoch 27/50
625/625 [==============================] - 16s 25ms/step - loss: 2.3944 - accuracy: 0.5097 -
val_loss: 2.5228 - val_accuracy: 0.4786
Epoch 28/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3806 - accuracy: 0.5163 -
val_loss: 2.4154 - val_accuracy: 0.5041
Epoch 29/50
625/625 [==============================] - 16s 26ms/step - loss: 2.3615 - accuracy: 0.5183 -
val_loss: 2.4868 - val_accuracy: 0.4857
Epoch 30/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3519 - accuracy: 0.5206 -
val_loss: 2.5119 - val_accuracy: 0.4827
Epoch 31/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3528 - accuracy: 0.5214 -
val_loss: 2.4952 - val_accuracy: 0.4891
Epoch 32/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3376 - accuracy: 0.5247 -
val_loss: 2.3821 - val_accuracy: 0.5147
Epoch 33/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3161 - accuracy: 0.5283 -
val_loss: 2.4035 - val_accuracy: 0.5097
Epoch 34/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3079 - accuracy: 0.5312 -
val_loss: 2.5429 - val_accuracy: 0.4700
Epoch 35/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3036 - accuracy: 0.5339 -
val_loss: 2.5365 - val_accuracy: 0.4757
Epoch 36/50
625/625 [==============================] - 15s 24ms/step - loss: 2.2914 - accuracy: 0.5341 -
val_loss: 2.3704 - val_accuracy: 0.5110
Epoch 37/50
625/625 [==============================] - 15s 24ms/step - loss: 2.2927 - accuracy: 0.5368 -
val_loss: 2.3781 - val_accuracy: 0.5212
Epoch 38/50
625/625 [==============================] - 15s 24ms/step - loss: 2.2815 - accuracy: 0.5383 -
val_loss: 2.3360 - val_accuracy: 0.5235
Epoch 39/50
625/625 [==============================] - 15s 24ms/step - loss: 2.2735 - accuracy: 0.5412 -
val_loss: 2.4338 - val_accuracy: 0.4965
Epoch 40/50
625/625 [==============================] - 18s 28ms/step - loss: 2.2664 - accuracy: 0.5454 -
val_loss: 2.3625 - val_accuracy: 0.5143
Epoch 41/50
625/625 [==============================] - 22s 35ms/step - loss: 2.2581 - accuracy: 0.5429 -
val_loss: 2.4147 - val_accuracy: 0.5074
Epoch 42/50
625/625 [==============================] - 17s 26ms/step - loss: 2.2553 - accuracy: 0.5447 -
val_loss: 2.3509 - val_accuracy: 0.5199
Epoch 43/50
625/625 [==============================] - 17s 28ms/step - loss: 2.2516 - accuracy: 0.5472 -
val_loss: 2.4125 - val_accuracy: 0.5101
Epoch 44/50
625/625 [==============================] - 15s 23ms/step - loss: 2.2350 - accuracy: 0.5483 -
val_loss: 2.3619 - val_accuracy: 0.5166
```

```
Epoch 45/50
625/625 [==============================] - 14s 22ms/step - loss: 2.2341 - accuracy: 0.5513 -
val_loss: 2.4521 - val_accuracy: 0.5022
Epoch 46/50
625/625 [==============================] - 14s 22ms/step - loss: 2.2266 - accuracy: 0.5500 -
val_loss: 2.4017 - val_accuracy: 0.5156
Epoch 47/50
625/625 [==============================] - 15s 24ms/step - loss: 2.2276 - accuracy: 0.5523 -
val_loss: 2.3393 - val_accuracy: 0.5242
Epoch 48/50
625/625 [==============================] - 20s 32ms/step - loss: 2.2262 - accuracy: 0.5532 -
val_loss: 2.3015 - val_accuracy: 0.5271
Epoch 49/50
625/625 [==============================] - 16s 26ms/step - loss: 2.2187 - accuracy: 0.5531 -
val_loss: 2.3374 - val_accuracy: 0.5248
Epoch 50/50
625/625 [==============================] - 15s 24ms/step - loss: 2.2218 - accuracy: 0.5534 -
val_loss: 2.3708 - val_accuracy: 0.5208
```

In [ ]:
```
print(storeResult(customVGGL1ModelHistory))
plot_loss_curve(customVGGL1ModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'CustomVGG_L1', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 2.22622323036193
85, 'Val Loss': 2.30151104927063, 'Train Acc': 0.5532000064849854, 'Val Acc': 0.5271000266075
134, '[Train - Val] Acc': 0.026099979877471924}
```





Observations

Even though by applying the L1 Lasso Regularisation, the model becomes generalise as both loss functions decreased. The decrease is consistent but the accuracy of both training and validation is slightly better than baseline and is worst that the normal conv2D model. This suggest the L1 Lasso Regularisation method is not very strong at improving the accuracy of the models.

Training CustomVGG model without Data Augmentation and using L2 Ridge Regularisation

In [ ]:
```
def vgg_block_l2(num_convs, num_channels, weight_decay=0.0005):
    blk = Sequential()
```

```
    for _ in range(num_convs):
        blk.add(
            Conv2D(num_channels, kernel_size=3,
                   padding='same', activation='relu', kernel_regularizer=l2(weight_decay)))
        blk.add(
            BatchNormalization())
    blk.add(MaxPool2D(pool_size=2, strides=2))
    return blk
```

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = vgg_block_l2(2, 32)(x)  # we are using less filters compared to VGG16
x = vgg_block_l2(2, 64)(x)
x = vgg_block_l2(3, 128)(x)
x = vgg_block_l2(3, 256)(x)
x = GlobalAveragePooling2D()(x)
x = Dense(256, 'relu')(x)
x = Dropout(0.3)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGGL2Model = Model(inputs=inputs, outputs=x, name="CustomVGG_L2")
customVGGL2Model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                         loss='categorical_crossentropy', metrics=['accuracy'])
```

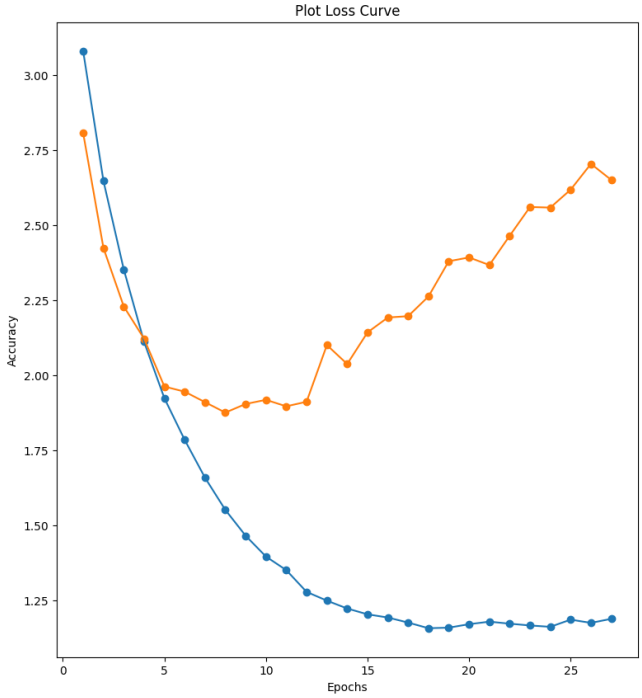
In [ ]:
```
customVGGL2ModelHistory = customVGGL2Model.fit(x_train, y_train, epochs=50,
                                               validation_data=(x_val, y_val), batch_size=BAT
```

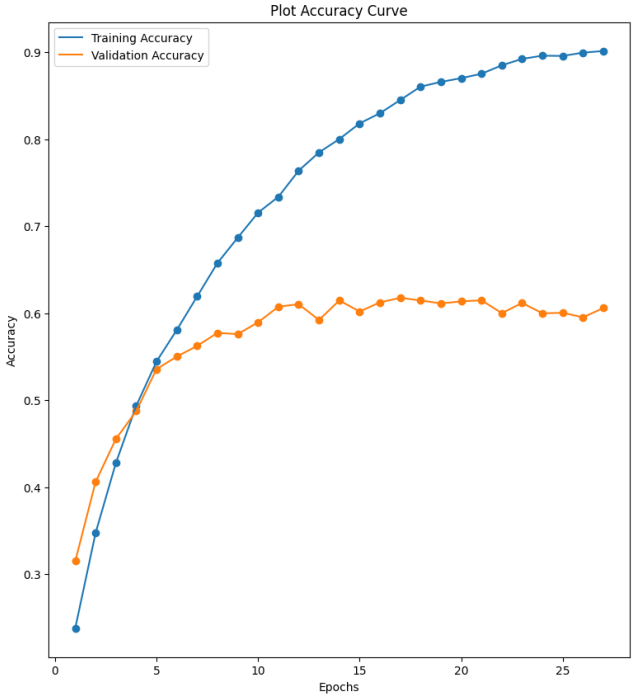
```
Epoch 1/50
625/625 [==============================] - 19s 26ms/step - loss: 3.0814 - accuracy: 0.2377 -
val_loss: 2.8082 - val_accuracy: 0.3148
Epoch 2/50
625/625 [==============================] - 15s 24ms/step - loss: 2.6482 - accuracy: 0.3473 -
val_loss: 2.4245 - val_accuracy: 0.4059
Epoch 3/50
625/625 [==============================] - 15s 23ms/step - loss: 2.3533 - accuracy: 0.4278 -
val_loss: 2.2297 - val_accuracy: 0.4554
Epoch 4/50
625/625 [==============================] - 15s 25ms/step - loss: 2.1116 - accuracy: 0.4933 -
val_loss: 2.1222 - val_accuracy: 0.4874
Epoch 5/50
625/625 [==============================] - 16s 26ms/step - loss: 1.9238 - accuracy: 0.5444 -
val_loss: 1.9627 - val_accuracy: 0.5357
Epoch 6/50
625/625 [==============================] - 17s 28ms/step - loss: 1.7848 - accuracy: 0.5808 -
val_loss: 1.9456 - val_accuracy: 0.5502
Epoch 7/50
625/625 [==============================] - 16s 25ms/step - loss: 1.6588 - accuracy: 0.6193 -
val_loss: 1.9107 - val_accuracy: 0.5624
Epoch 8/50
625/625 [==============================] - 18s 28ms/step - loss: 1.5515 - accuracy: 0.6577 -
val_loss: 1.8759 - val_accuracy: 0.5774
Epoch 9/50
625/625 [==============================] - 16s 26ms/step - loss: 1.4654 - accuracy: 0.6870 -
val_loss: 1.9046 - val_accuracy: 0.5760
Epoch 10/50
625/625 [==============================] - 16s 25ms/step - loss: 1.3959 - accuracy: 0.7157 -
val_loss: 1.9185 - val_accuracy: 0.5896
Epoch 11/50
625/625 [==============================] - 15s 24ms/step - loss: 1.3507 - accuracy: 0.7339 -
val_loss: 1.8968 - val_accuracy: 0.6075
Epoch 12/50
625/625 [==============================] - 15s 24ms/step - loss: 1.2781 - accuracy: 0.7640 -
val_loss: 1.9121 - val_accuracy: 0.6104
Epoch 13/50
625/625 [==============================] - 16s 25ms/step - loss: 1.2490 - accuracy: 0.7850 -
val_loss: 2.1011 - val_accuracy: 0.5922
Epoch 14/50
625/625 [==============================] - 16s 26ms/step - loss: 1.2227 - accuracy: 0.8003 -
val_loss: 2.0378 - val_accuracy: 0.6147
Epoch 15/50
625/625 [==============================] - 16s 25ms/step - loss: 1.2033 - accuracy: 0.8183 -
val_loss: 2.1432 - val_accuracy: 0.6020
Epoch 16/50
625/625 [==============================] - 16s 25ms/step - loss: 1.1927 - accuracy: 0.8300 -
val_loss: 2.1929 - val_accuracy: 0.6125
Epoch 17/50
625/625 [==============================] - 16s 26ms/step - loss: 1.1758 - accuracy: 0.8452 -
val_loss: 2.1974 - val_accuracy: 0.6177
Epoch 18/50
625/625 [==============================] - 18s 28ms/step - loss: 1.1570 - accuracy: 0.8606 -
val_loss: 2.2629 - val_accuracy: 0.6147
Epoch 19/50
625/625 [==============================] - 17s 28ms/step - loss: 1.1589 - accuracy: 0.8661 -
val_loss: 2.3805 - val_accuracy: 0.6113
Epoch 20/50
625/625 [==============================] - 17s 28ms/step - loss: 1.1704 - accuracy: 0.8704 -
val_loss: 2.3932 - val_accuracy: 0.6136
Epoch 21/50
625/625 [==============================] - 19s 30ms/step - loss: 1.1788 - accuracy: 0.8754 -
val_loss: 2.3678 - val_accuracy: 0.6149
Epoch 22/50
625/625 [==============================] - 14s 22ms/step - loss: 1.1722 - accuracy: 0.8851 -
val_loss: 2.4655 - val_accuracy: 0.6001
```

```
Epoch 23/50
625/625 [==============================] - 12s 19ms/step - loss: 1.1662 - accuracy: 0.8925 -
val_loss: 2.5613 - val_accuracy: 0.6118
Epoch 24/50
625/625 [==============================] - 13s 21ms/step - loss: 1.1617 - accuracy: 0.8962 -
val_loss: 2.5594 - val_accuracy: 0.5999
Epoch 25/50
625/625 [==============================] - 20s 32ms/step - loss: 1.1856 - accuracy: 0.8959 -
val_loss: 2.6191 - val_accuracy: 0.6005
Epoch 26/50
625/625 [==============================] - 21s 33ms/step - loss: 1.1748 - accuracy: 0.8997 -
val_loss: 2.7052 - val_accuracy: 0.5953
Epoch 27/50
625/625 [==============================] - 17s 27ms/step - loss: 1.1882 - accuracy: 0.9016 -
val_loss: 2.6516 - val_accuracy: 0.6060
```

In [ ]:
```
print(storeResult(customVGGL2ModelHistory))
plot_loss_curve(customVGGL2ModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'CustomVGG_L2', 'Epochs': 27, 'Batch Size': 64, 'Train Loss': 1.17581009864807
13, 'Val Loss': 2.197413682937622, 'Train Acc': 0.8452000021934509, 'Val Acc': 0.617699980735
7788, '[Train - Val] Acc': 0.22750002145767212}
```





Observations

Comparing the CustomVGGL2 model and CustomVGG model, the accuracy of the validation data increased but there is an increase in training loss and validation loss. This suggest that the model has become less generalised even though a regularisation is suppose to help make the model more generalise and reduce overfitting.

Training CustomVGG model with Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = vgg_block(2, 32)(x)  # we are using less filters compared to VGG16
x = vgg_block(2, 64)(x)
```

```
x = vgg_block(3, 128)(x)
x = vgg_block(3, 256)(x)
x = GlobalAveragePooling2D()(x)
x = Dense(256, 'relu')(x)
x = Dropout(0.3)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGGAugModel = Model(inputs=inputs, outputs=x, name="CustomVGGAug")
customVGGAugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                          loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:

```
customVGGAugModelHistory = customVGGAugModel.fit(x_train_aug, y_train, epochs=50,
                                                 validation_data=(x_val, y_val), batch_size=BAT
```

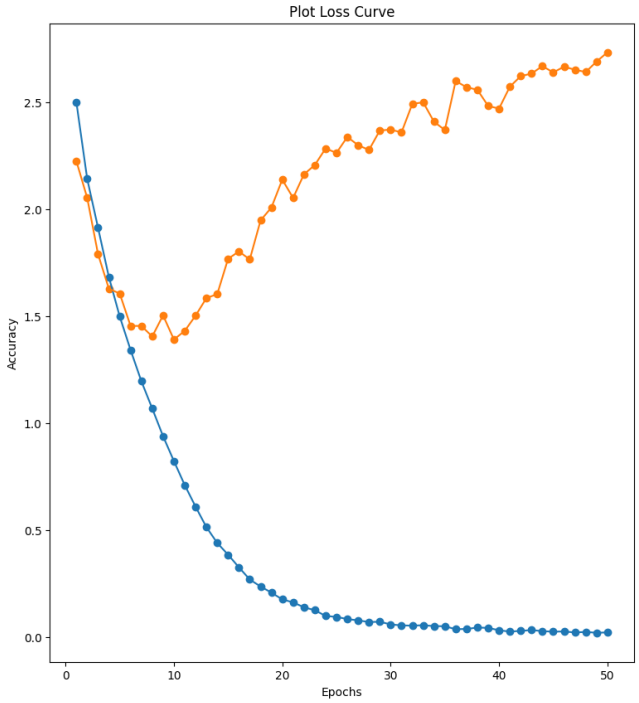
```
Epoch 1/50
625/625 [==============================] - 19s 26ms/step - loss: 2.5015 - accuracy: 0.2336 -
val_loss: 2.2258 - val_accuracy: 0.3125
Epoch 2/50
625/625 [==============================] - 14s 22ms/step - loss: 2.1427 - accuracy: 0.3391 -
val_loss: 2.0538 - val_accuracy: 0.3684
Epoch 3/50
625/625 [==============================] - 15s 24ms/step - loss: 1.9142 - accuracy: 0.4044 -
val_loss: 1.7907 - val_accuracy: 0.4447
Epoch 4/50
625/625 [==============================] - 15s 24ms/step - loss: 1.6840 - accuracy: 0.4721 -
val_loss: 1.6291 - val_accuracy: 0.4887
Epoch 5/50
625/625 [==============================] - 15s 24ms/step - loss: 1.4995 - accuracy: 0.5219 -
val_loss: 1.6067 - val_accuracy: 0.5001
Epoch 6/50
625/625 [==============================] - 16s 26ms/step - loss: 1.3432 - accuracy: 0.5695 -
val_loss: 1.4554 - val_accuracy: 0.5451
Epoch 7/50
625/625 [==============================] - 18s 28ms/step - loss: 1.1965 - accuracy: 0.6120 -
val_loss: 1.4553 - val_accuracy: 0.5565
Epoch 8/50
625/625 [==============================] - 16s 25ms/step - loss: 1.0696 - accuracy: 0.6540 -
val_loss: 1.4059 - val_accuracy: 0.5739
Epoch 9/50
625/625 [==============================] - 19s 30ms/step - loss: 0.9399 - accuracy: 0.6923 -
val_loss: 1.5048 - val_accuracy: 0.5593
Epoch 10/50
625/625 [==============================] - 21s 33ms/step - loss: 0.8237 - accuracy: 0.7294 -
val_loss: 1.3917 - val_accuracy: 0.5939
Epoch 11/50
625/625 [==============================] - 18s 29ms/step - loss: 0.7114 - accuracy: 0.7642 -
val_loss: 1.4320 - val_accuracy: 0.5955
Epoch 12/50
625/625 [==============================] - 14s 23ms/step - loss: 0.6105 - accuracy: 0.7983 -
val_loss: 1.5034 - val_accuracy: 0.5990
Epoch 13/50
625/625 [==============================] - 16s 25ms/step - loss: 0.5148 - accuracy: 0.8285 -
val_loss: 1.5854 - val_accuracy: 0.5974
Epoch 14/50
625/625 [==============================] - 14s 23ms/step - loss: 0.4409 - accuracy: 0.8526 -
val_loss: 1.6040 - val_accuracy: 0.5969
Epoch 15/50
625/625 [==============================] - 15s 24ms/step - loss: 0.3861 - accuracy: 0.8731 -
val_loss: 1.7689 - val_accuracy: 0.5835
Epoch 16/50
625/625 [==============================] - 15s 24ms/step - loss: 0.3270 - accuracy: 0.8892 -
val_loss: 1.8034 - val_accuracy: 0.5996
Epoch 17/50
625/625 [==============================] - 14s 22ms/step - loss: 0.2701 - accuracy: 0.9104 -
val_loss: 1.7659 - val_accuracy: 0.6071
Epoch 18/50
625/625 [==============================] - 17s 27ms/step - loss: 0.2372 - accuracy: 0.9205 -
val_loss: 1.9486 - val_accuracy: 0.5987
Epoch 19/50
625/625 [==============================] - 17s 28ms/step - loss: 0.2089 - accuracy: 0.9301 -
val_loss: 2.0087 - val_accuracy: 0.5935
Epoch 20/50
625/625 [==============================] - 16s 26ms/step - loss: 0.1770 - accuracy: 0.9401 -
val_loss: 2.1392 - val_accuracy: 0.5997
Epoch 21/50
625/625 [==============================] - 15s 24ms/step - loss: 0.1633 - accuracy: 0.9458 -
val_loss: 2.0535 - val_accuracy: 0.5993
Epoch 22/50
625/625 [==============================] - 16s 25ms/step - loss: 0.1402 - accuracy: 0.9544 -
val_loss: 2.1639 - val_accuracy: 0.5986
```

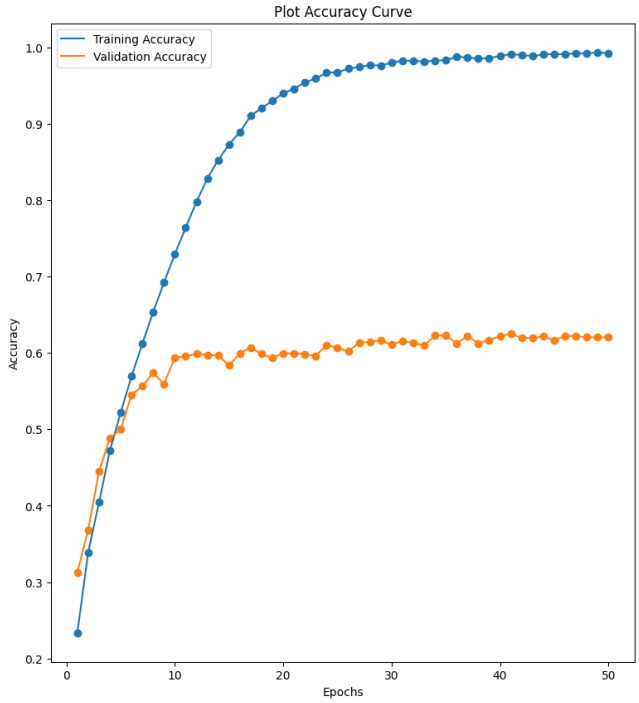
```
Epoch 23/50
625/625 [==============================] - 15s 24ms/step - loss: 0.1267 - accuracy: 0.9592 -
val_loss: 2.2070 - val_accuracy: 0.5960
Epoch 24/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0996 - accuracy: 0.9670 -
val_loss: 2.2848 - val_accuracy: 0.6104
Epoch 25/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0954 - accuracy: 0.9675 -
val_loss: 2.2639 - val_accuracy: 0.6066
Epoch 26/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0852 - accuracy: 0.9721 -
val_loss: 2.3383 - val_accuracy: 0.6020
Epoch 27/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0790 - accuracy: 0.9744 -
val_loss: 2.3001 - val_accuracy: 0.6137
Epoch 28/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0714 - accuracy: 0.9768 -
val_loss: 2.2775 - val_accuracy: 0.6144
Epoch 29/50
625/625 [==============================] - 15s 25ms/step - loss: 0.0726 - accuracy: 0.9764 -
val_loss: 2.3687 - val_accuracy: 0.6165
Epoch 30/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0601 - accuracy: 0.9800 -
val_loss: 2.3727 - val_accuracy: 0.6108
Epoch 31/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0556 - accuracy: 0.9825 -
val_loss: 2.3596 - val_accuracy: 0.6158
Epoch 32/50
625/625 [==============================] - 15s 23ms/step - loss: 0.0545 - accuracy: 0.9826 -
val_loss: 2.4929 - val_accuracy: 0.6128
Epoch 33/50
625/625 [==============================] - 15s 23ms/step - loss: 0.0564 - accuracy: 0.9814 -
val_loss: 2.5005 - val_accuracy: 0.6099
Epoch 34/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0525 - accuracy: 0.9828 -
val_loss: 2.4095 - val_accuracy: 0.6226
Epoch 35/50
625/625 [==============================] - 15s 23ms/step - loss: 0.0516 - accuracy: 0.9832 -
val_loss: 2.3708 - val_accuracy: 0.6233
Epoch 36/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0378 - accuracy: 0.9878 -
val_loss: 2.6000 - val_accuracy: 0.6125
Epoch 37/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0390 - accuracy: 0.9868 -
val_loss: 2.5709 - val_accuracy: 0.6219
Epoch 38/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0455 - accuracy: 0.9853 -
val_loss: 2.5589 - val_accuracy: 0.6127
Epoch 39/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0440 - accuracy: 0.9859 -
val_loss: 2.4833 - val_accuracy: 0.6169
Epoch 40/50
625/625 [==============================] - 18s 29ms/step - loss: 0.0328 - accuracy: 0.9890 -
val_loss: 2.4709 - val_accuracy: 0.6220
Epoch 41/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0267 - accuracy: 0.9911 -
val_loss: 2.5733 - val_accuracy: 0.6252
Epoch 42/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0300 - accuracy: 0.9900 -
val_loss: 2.6227 - val_accuracy: 0.6197
Epoch 43/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0338 - accuracy: 0.9887 -
val_loss: 2.6345 - val_accuracy: 0.6194
Epoch 44/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0288 - accuracy: 0.9907 -
val_loss: 2.6696 - val_accuracy: 0.6218
```

```
Epoch 45/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0262 - accuracy: 0.9913 -
val_loss: 2.6401 - val_accuracy: 0.6164
Epoch 46/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0275 - accuracy: 0.9911 -
val_loss: 2.6669 - val_accuracy: 0.6223
Epoch 47/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0225 - accuracy: 0.9926 -
val_loss: 2.6526 - val_accuracy: 0.6223
Epoch 48/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0251 - accuracy: 0.9922 -
val_loss: 2.6419 - val_accuracy: 0.6205
Epoch 49/50
625/625 [==============================] - 15s 23ms/step - loss: 0.0210 - accuracy: 0.9936 -
val_loss: 2.6908 - val_accuracy: 0.6205
Epoch 50/50
625/625 [==============================] - 17s 27ms/step - loss: 0.0228 - accuracy: 0.9924 -
val_loss: 2.7339 - val_accuracy: 0.6205
```

In [ ]:
```
print(storeResult(customVGGAugModelHistory))
plot_loss_curve(customVGGAugModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'CustomVGGAug', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 0.02670162357389
927, 'Val Loss': 2.573303461074829, 'Train Acc': 0.9911249876022339, 'Val Acc': 0.62519997358
32214, '[Train - Val] Acc': 0.36592501401901245}
```





Observations

Comparing customVGG model without Data Augmentation to the customVGG model with Data Augmentation, we can see that by applying augmentation, the validation accuracy increase as well as the training accuracy almost being a 1. This suggest that augmentation is beneficial for a VGG model

CustomVGG16

To reduce overfitting, we will be increasing the number of dropout layers. But to keep the model's performance, we will be increasing the number of layers to match the VGG16 model.

CustomVGG-16 model without Data Augmentation

In [ ]:
```
def vgg_block_16(num_convs, num_channels, weight_decay=0.0005, dropout=[]):
    blk = Sequential()
    while num_convs - len(dropout) > 0:
        dropout.append(0)
    for idx in range(num_convs):
        blk.add(
            Conv2D(num_channels, kernel_size=3,
                   padding='same', activation='relu', kernel_regularizer=l2(weight_decay)))
        blk.add(
            BatchNormalization())
        blk.add(Dropout(dropout[idx]))
    blk.add(MaxPool2D(pool_size=2, strides=2))
    return blk
```

In [ ]:
```
tf.keras.backend.clear_session()
weight_decay = 0.0005
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = vgg_block_16(2, 64,dropout=[0.3])(x)
x = vgg_block_16(2, 128,dropout=[0.4])(x)
x = vgg_block_16(3, 256,dropout=[0.4,0.4])(x)
x = vgg_block_16(3, 512,dropout=[0.4,0.4])(x)
x = vgg_block_16(3, 512, dropout=[0.4, 0.4])(x)
x = Dropout(0.5)(x)
x = GlobalAveragePooling2D()(x)
x = Dense(512, 'relu',kernel_regularizer=l2(weight_decay))(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGG16Model = Model(inputs=inputs, outputs=x, name="CustomVGG16")
customVGG16Model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                         loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
customVGG16Model.summary()
```

```
Model: "CustomVGG16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 normalization (Normalizatio  (None, 32, 32, 3)        7
 n)

 sequential (Sequential)     (None, 16, 16, 64)        39232

 sequential_1 (Sequential)   (None, 8, 8, 128)         222464

 sequential_2 (Sequential)   (None, 4, 4, 256)         1478400

 sequential_3 (Sequential)   (None, 2, 2, 512)         5905920

 sequential_4 (Sequential)   (None, 1, 1, 512)         7085568

 dropout_13 (Dropout)        (None, 1, 1, 512)         0

 global_average_pooling2d (G  (None, 512)              0
 lobalAveragePooling2D)

 dense (Dense)               (None, 512)               262656

 batch_normalization_13 (Bat  (None, 512)              2048
 chNormalization)

 dropout_14 (Dropout)        (None, 512)               0

 dense_1 (Dense)             (None, 20)                10260

=================================================================
Total params: 15,006,555
Trainable params: 14,997,076
Non-trainable params: 9,479
_________________________________________________________________
```

In [ ]:
```
customVGG16ModelHistory = customVGG16Model.fit(x_train, y_train, epochs=50,
                                               validation_data=(x_val, y_val), batch_size=BAT
```

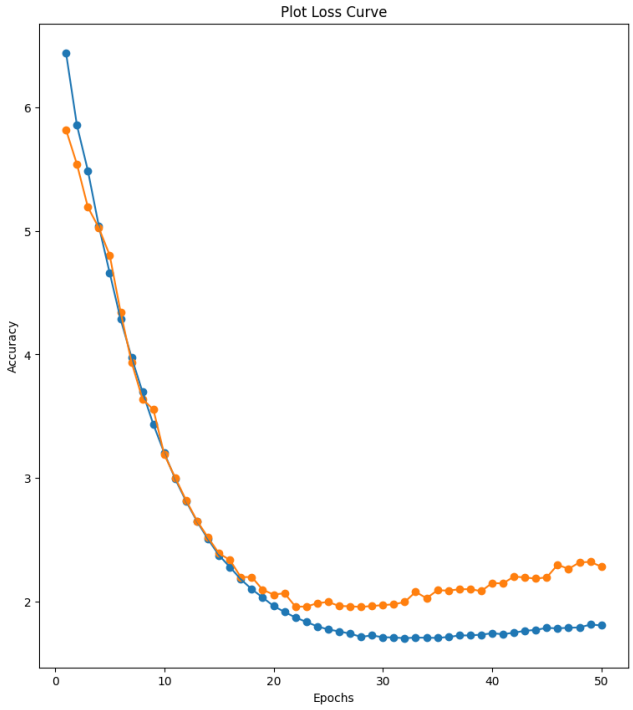
```
Epoch 1/50
625/625 [==============================] - 48s 69ms/step - loss: 6.4399 - accuracy: 0.1032 -
val_loss: 5.8177 - val_accuracy: 0.1154
Epoch 2/50
625/625 [==============================] - 40s 65ms/step - loss: 5.8534 - accuracy: 0.1480 -
val_loss: 5.5385 - val_accuracy: 0.1189
Epoch 3/50
625/625 [==============================] - 41s 66ms/step - loss: 5.4876 - accuracy: 0.1663 -
val_loss: 5.1920 - val_accuracy: 0.1636
Epoch 4/50
625/625 [==============================] - 42s 68ms/step - loss: 5.0403 - accuracy: 0.1979 -
val_loss: 5.0249 - val_accuracy: 0.1731
Epoch 5/50
625/625 [==============================] - 44s 70ms/step - loss: 4.6590 - accuracy: 0.2346 -
val_loss: 4.8025 - val_accuracy: 0.1954
Epoch 6/50
625/625 [==============================] - 40s 64ms/step - loss: 4.2845 - accuracy: 0.2650 -
val_loss: 4.3416 - val_accuracy: 0.2214
Epoch 7/50
625/625 [==============================] - 40s 63ms/step - loss: 3.9764 - accuracy: 0.2900 -
val_loss: 3.9370 - val_accuracy: 0.2771
Epoch 8/50
625/625 [==============================] - 40s 64ms/step - loss: 3.6973 - accuracy: 0.3212 -
val_loss: 3.6403 - val_accuracy: 0.3170
Epoch 9/50
625/625 [==============================] - 40s 64ms/step - loss: 3.4332 - accuracy: 0.3501 -
val_loss: 3.5537 - val_accuracy: 0.3329
Epoch 10/50
625/625 [==============================] - 40s 64ms/step - loss: 3.2020 - accuracy: 0.3818 -
val_loss: 3.1894 - val_accuracy: 0.3723
Epoch 11/50
625/625 [==============================] - 40s 64ms/step - loss: 2.9922 - accuracy: 0.4184 -
val_loss: 3.0032 - val_accuracy: 0.4038
Epoch 12/50
625/625 [==============================] - 40s 64ms/step - loss: 2.8114 - accuracy: 0.4445 -
val_loss: 2.8192 - val_accuracy: 0.4349
Epoch 13/50
625/625 [==============================] - 40s 64ms/step - loss: 2.6487 - accuracy: 0.4719 -
val_loss: 2.6539 - val_accuracy: 0.4710
Epoch 14/50
625/625 [==============================] - 40s 64ms/step - loss: 2.5053 - accuracy: 0.4960 -
val_loss: 2.5195 - val_accuracy: 0.4892
Epoch 15/50
625/625 [==============================] - 40s 64ms/step - loss: 2.3760 - accuracy: 0.5198 -
val_loss: 2.3915 - val_accuracy: 0.5136
Epoch 16/50
625/625 [==============================] - 40s 64ms/step - loss: 2.2807 - accuracy: 0.5352 -
val_loss: 2.3379 - val_accuracy: 0.5310
Epoch 17/50
625/625 [==============================] - 40s 64ms/step - loss: 2.1817 - accuracy: 0.5580 -
val_loss: 2.1992 - val_accuracy: 0.5538
Epoch 18/50
625/625 [==============================] - 40s 64ms/step - loss: 2.1016 - accuracy: 0.5790 -
val_loss: 2.2006 - val_accuracy: 0.5530
Epoch 19/50
625/625 [==============================] - 40s 64ms/step - loss: 2.0345 - accuracy: 0.5895 -
val_loss: 2.0970 - val_accuracy: 0.5757
Epoch 20/50
625/625 [==============================] - 40s 64ms/step - loss: 1.9654 - accuracy: 0.6092 -
val_loss: 2.0583 - val_accuracy: 0.5869
Epoch 21/50
625/625 [==============================] - 40s 64ms/step - loss: 1.9179 - accuracy: 0.6234 -
val_loss: 2.0686 - val_accuracy: 0.5832
Epoch 22/50
625/625 [==============================] - 40s 63ms/step - loss: 1.8728 - accuracy: 0.6364 -
val_loss: 1.9608 - val_accuracy: 0.6096
```

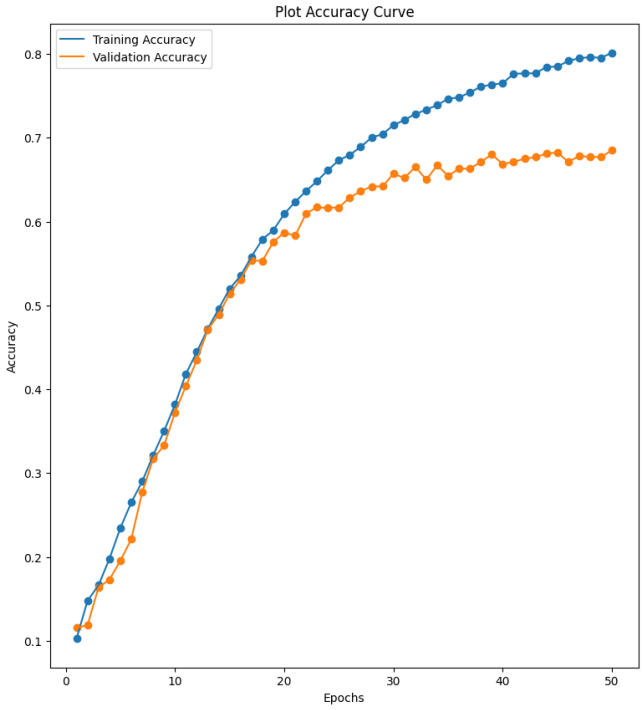
```
Epoch 23/50
625/625 [==============================] - 40s 65ms/step - loss: 1.8371 - accuracy: 0.6482 -
val_loss: 1.9608 - val_accuracy: 0.6170
Epoch 24/50
625/625 [==============================] - 40s 64ms/step - loss: 1.8007 - accuracy: 0.6613 -
val_loss: 1.9884 - val_accuracy: 0.6162
Epoch 25/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7762 - accuracy: 0.6731 -
val_loss: 1.9986 - val_accuracy: 0.6169
Epoch 26/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7603 - accuracy: 0.6795 -
val_loss: 1.9657 - val_accuracy: 0.6285
Epoch 27/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7422 - accuracy: 0.6891 -
val_loss: 1.9634 - val_accuracy: 0.6363
Epoch 28/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7165 - accuracy: 0.7001 -
val_loss: 1.9581 - val_accuracy: 0.6418
Epoch 29/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7276 - accuracy: 0.7042 -
val_loss: 1.9657 - val_accuracy: 0.6419
Epoch 30/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7133 - accuracy: 0.7150 -
val_loss: 1.9732 - val_accuracy: 0.6570
Epoch 31/50
625/625 [==============================] - 40s 63ms/step - loss: 1.7111 - accuracy: 0.7210 -
val_loss: 1.9794 - val_accuracy: 0.6519
Epoch 32/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7055 - accuracy: 0.7283 -
val_loss: 1.9981 - val_accuracy: 0.6653
Epoch 33/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7104 - accuracy: 0.7333 -
val_loss: 2.0801 - val_accuracy: 0.6499
Epoch 34/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7091 - accuracy: 0.7389 -
val_loss: 2.0286 - val_accuracy: 0.6670
Epoch 35/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7078 - accuracy: 0.7463 -
val_loss: 2.0931 - val_accuracy: 0.6546
Epoch 36/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7150 - accuracy: 0.7481 -
val_loss: 2.0906 - val_accuracy: 0.6630
Epoch 37/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7274 - accuracy: 0.7539 -
val_loss: 2.1007 - val_accuracy: 0.6631
Epoch 38/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7281 - accuracy: 0.7609 -
val_loss: 2.1024 - val_accuracy: 0.6711
Epoch 39/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7338 - accuracy: 0.7633 -
val_loss: 2.0858 - val_accuracy: 0.6800
Epoch 40/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7454 - accuracy: 0.7649 -
val_loss: 2.1511 - val_accuracy: 0.6685
Epoch 41/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7375 - accuracy: 0.7762 -
val_loss: 2.1484 - val_accuracy: 0.6713
Epoch 42/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7503 - accuracy: 0.7767 -
val_loss: 2.2040 - val_accuracy: 0.6749
Epoch 43/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7648 - accuracy: 0.7768 -
val_loss: 2.1966 - val_accuracy: 0.6769
Epoch 44/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7728 - accuracy: 0.7844 -
val_loss: 2.1873 - val_accuracy: 0.6809
```

```
Epoch 45/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7885 - accuracy: 0.7848 -
val_loss: 2.1976 - val_accuracy: 0.6825
Epoch 46/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7852 - accuracy: 0.7915 -
val_loss: 2.2993 - val_accuracy: 0.6711
Epoch 47/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7897 - accuracy: 0.7948 -
val_loss: 2.2658 - val_accuracy: 0.6782
Epoch 48/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7924 - accuracy: 0.7960 -
val_loss: 2.3188 - val_accuracy: 0.6769
Epoch 49/50
625/625 [==============================] - 40s 64ms/step - loss: 1.8156 - accuracy: 0.7953 -
val_loss: 2.3227 - val_accuracy: 0.6770
Epoch 50/50
625/625 [==============================] - 40s 64ms/step - loss: 1.8090 - accuracy: 0.8012 -
val_loss: 2.2840 - val_accuracy: 0.6850
```

In [ ]:
```
print(storeResult(customVGG16ModelHistory))
plot_loss_curve(customVGG16ModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'CustomVGG16', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 1.808951139450073
2, 'Val Loss': 2.2840487957000732, 'Train Acc': 0.8012250065803528, 'Val Acc': 0.685000002384
1858, '[Train - Val] Acc': 0.11622500419616699}
```





Observations

We can see that after we have added more dropout layers, the model has become more generalised and that the both loss has been reduced significantly compared to the other models previously. This suggest that we should add more layers and increase the dropout to make model more generalise and prevent overfitting at the same time.

CustomVGG-16 model with Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
weight_decay = 0.0005
```

```
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = vgg_block_16(2, 64,dropout=[0.3])(x)
x = vgg_block_16(2, 128,dropout=[0.4])(x)
x = vgg_block_16(3, 256,dropout=[0.4,0.4])(x)
x = vgg_block_16(3, 512,dropout=[0.4,0.4])(x)
x = vgg_block_16(3, 512, dropout=[0.4, 0.4])(x)
x = Dropout(0.5)(x)
x = GlobalAveragePooling2D()(x)
x = Dense(512, 'relu',kernel_regularizer=l2(weight_decay))(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGG16AugModel = Model(inputs=inputs, outputs=x, name="CustomVGG16Aug")
customVGG16AugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                      loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:

```
customVGG16AugModelHistory = customVGG16AugModel.fit(x_train_aug, y_train, epochs=50,
                                                     validation_data=(x_val, y_val), batch_size=BAT
```

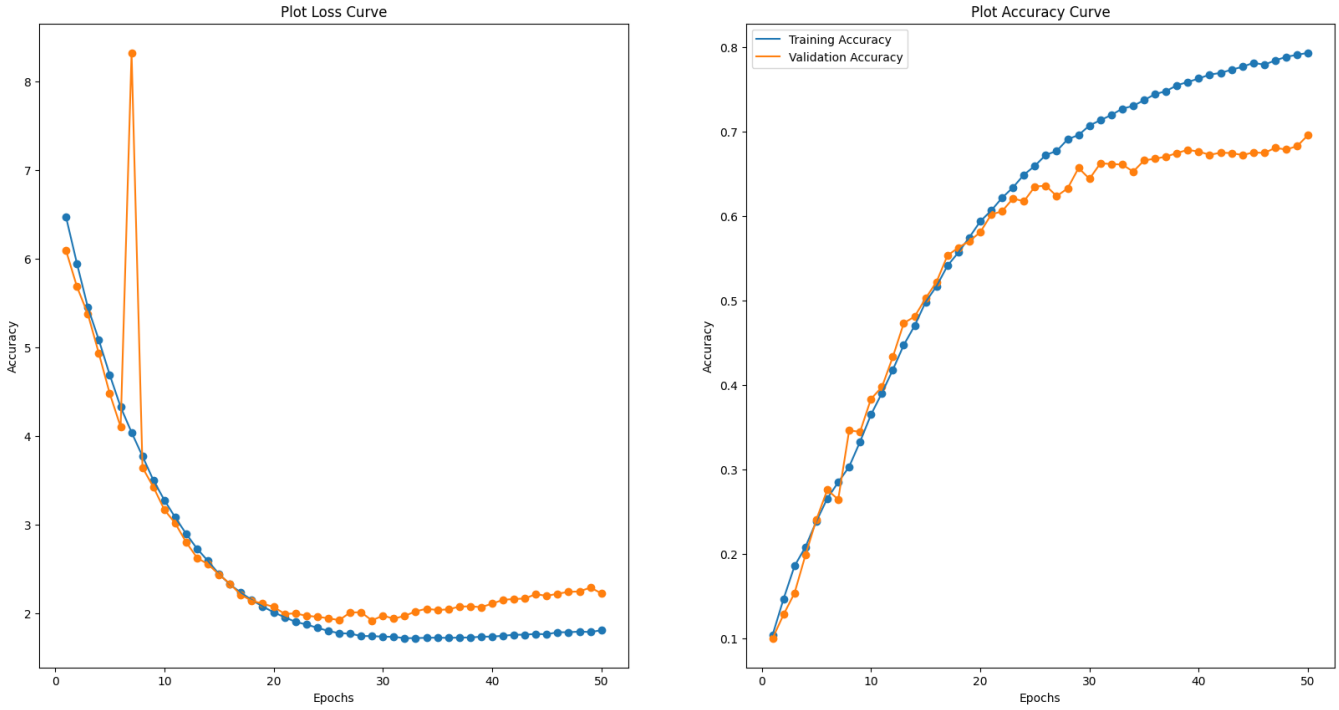
```
Epoch 1/50
625/625 [==============================] - 46s 68ms/step - loss: 6.4733 - accuracy: 0.1045 -
val_loss: 6.0955 - val_accuracy: 0.1004
Epoch 2/50
625/625 [==============================] - 41s 65ms/step - loss: 5.9481 - accuracy: 0.1470 -
val_loss: 5.6866 - val_accuracy: 0.1284
Epoch 3/50
625/625 [==============================] - 41s 65ms/step - loss: 5.4527 - accuracy: 0.1857 -
val_loss: 5.3773 - val_accuracy: 0.1537
Epoch 4/50
625/625 [==============================] - 41s 65ms/step - loss: 5.0869 - accuracy: 0.2083 -
val_loss: 4.9366 - val_accuracy: 0.1986
Epoch 5/50
625/625 [==============================] - 41s 65ms/step - loss: 4.6927 - accuracy: 0.2386 -
val_loss: 4.4823 - val_accuracy: 0.2407
Epoch 6/50
625/625 [==============================] - 41s 65ms/step - loss: 4.3295 - accuracy: 0.2657 -
val_loss: 4.1047 - val_accuracy: 0.2759
Epoch 7/50
625/625 [==============================] - 41s 65ms/step - loss: 4.0436 - accuracy: 0.2849 -
val_loss: 8.3244 - val_accuracy: 0.2645
Epoch 8/50
625/625 [==============================] - 41s 65ms/step - loss: 3.7743 - accuracy: 0.3033 -
val_loss: 3.6462 - val_accuracy: 0.3462
Epoch 9/50
625/625 [==============================] - 41s 65ms/step - loss: 3.5027 - accuracy: 0.3329 -
val_loss: 3.4263 - val_accuracy: 0.3446
Epoch 10/50
625/625 [==============================] - 41s 65ms/step - loss: 3.2796 - accuracy: 0.3648 -
val_loss: 3.1713 - val_accuracy: 0.3835
Epoch 11/50
625/625 [==============================] - 41s 65ms/step - loss: 3.0832 - accuracy: 0.3905 -
val_loss: 3.0171 - val_accuracy: 0.3975
Epoch 12/50
625/625 [==============================] - 41s 65ms/step - loss: 2.8989 - accuracy: 0.4180 -
val_loss: 2.8035 - val_accuracy: 0.4332
Epoch 13/50
625/625 [==============================] - 41s 65ms/step - loss: 2.7301 - accuracy: 0.4473 -
val_loss: 2.6280 - val_accuracy: 0.4731
Epoch 14/50
625/625 [==============================] - 40s 65ms/step - loss: 2.5922 - accuracy: 0.4702 -
val_loss: 2.5556 - val_accuracy: 0.4810
Epoch 15/50
625/625 [==============================] - 41s 66ms/step - loss: 2.4461 - accuracy: 0.4981 -
val_loss: 2.4405 - val_accuracy: 0.5025
Epoch 16/50
625/625 [==============================] - 41s 65ms/step - loss: 2.3291 - accuracy: 0.5168 -
val_loss: 2.3358 - val_accuracy: 0.5220
Epoch 17/50
625/625 [==============================] - 40s 64ms/step - loss: 2.2353 - accuracy: 0.5411 -
val_loss: 2.2116 - val_accuracy: 0.5531
Epoch 18/50
625/625 [==============================] - 41s 65ms/step - loss: 2.1532 - accuracy: 0.5568 -
val_loss: 2.1438 - val_accuracy: 0.5625
Epoch 19/50
625/625 [==============================] - 41s 66ms/step - loss: 2.0794 - accuracy: 0.5745 -
val_loss: 2.1110 - val_accuracy: 0.5702
Epoch 20/50
625/625 [==============================] - 41s 65ms/step - loss: 2.0144 - accuracy: 0.5935 -
val_loss: 2.0751 - val_accuracy: 0.5807
Epoch 21/50
625/625 [==============================] - 41s 65ms/step - loss: 1.9589 - accuracy: 0.6063 -
val_loss: 1.9968 - val_accuracy: 0.6015
Epoch 22/50
625/625 [==============================] - 41s 65ms/step - loss: 1.9065 - accuracy: 0.6214 -
val_loss: 2.0005 - val_accuracy: 0.6056
```

```
Epoch 23/50
625/625 [==============================] - 41s 65ms/step - loss: 1.8763 - accuracy: 0.6338 -
val_loss: 1.9750 - val_accuracy: 0.6206
Epoch 24/50
625/625 [==============================] - 41s 65ms/step - loss: 1.8394 - accuracy: 0.6488 -
val_loss: 1.9629 - val_accuracy: 0.6173
Epoch 25/50
625/625 [==============================] - 41s 65ms/step - loss: 1.8038 - accuracy: 0.6593 -
val_loss: 1.9474 - val_accuracy: 0.6347
Epoch 26/50
625/625 [==============================] - 41s 66ms/step - loss: 1.7781 - accuracy: 0.6722 -
val_loss: 1.9248 - val_accuracy: 0.6358
Epoch 27/50
625/625 [==============================] - 40s 65ms/step - loss: 1.7738 - accuracy: 0.6769 -
val_loss: 2.0110 - val_accuracy: 0.6234
Epoch 28/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7463 - accuracy: 0.6906 -
val_loss: 2.0149 - val_accuracy: 0.6326
Epoch 29/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7457 - accuracy: 0.6957 -
val_loss: 1.9219 - val_accuracy: 0.6567
Epoch 30/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7391 - accuracy: 0.7070 -
val_loss: 1.9761 - val_accuracy: 0.6439
Epoch 31/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7371 - accuracy: 0.7133 -
val_loss: 1.9424 - val_accuracy: 0.6623
Epoch 32/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7220 - accuracy: 0.7194 -
val_loss: 1.9735 - val_accuracy: 0.6614
Epoch 33/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7230 - accuracy: 0.7269 -
val_loss: 2.0245 - val_accuracy: 0.6608
Epoch 34/50
625/625 [==============================] - 41s 66ms/step - loss: 1.7262 - accuracy: 0.7302 -
val_loss: 2.0540 - val_accuracy: 0.6524
Epoch 35/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7280 - accuracy: 0.7370 -
val_loss: 2.0392 - val_accuracy: 0.6657
Epoch 36/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7251 - accuracy: 0.7441 -
val_loss: 2.0471 - val_accuracy: 0.6677
Epoch 37/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7289 - accuracy: 0.7476 -
val_loss: 2.0774 - val_accuracy: 0.6701
Epoch 38/50
625/625 [==============================] - 42s 67ms/step - loss: 1.7296 - accuracy: 0.7546 -
val_loss: 2.0813 - val_accuracy: 0.6744
Epoch 39/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7387 - accuracy: 0.7584 -
val_loss: 2.0720 - val_accuracy: 0.6782
Epoch 40/50
625/625 [==============================] - 40s 65ms/step - loss: 1.7397 - accuracy: 0.7626 -
val_loss: 2.1112 - val_accuracy: 0.6761
Epoch 41/50
625/625 [==============================] - 41s 66ms/step - loss: 1.7491 - accuracy: 0.7672 -
val_loss: 2.1544 - val_accuracy: 0.6723
Epoch 42/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7605 - accuracy: 0.7692 -
val_loss: 2.1638 - val_accuracy: 0.6748
Epoch 43/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7615 - accuracy: 0.7729 -
val_loss: 2.1691 - val_accuracy: 0.6744
Epoch 44/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7702 - accuracy: 0.7764 -
val_loss: 2.2188 - val_accuracy: 0.6720
```

```
Epoch 45/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7657 - accuracy: 0.7811 -
val_loss: 2.1991 - val_accuracy: 0.6751
Epoch 46/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7895 - accuracy: 0.7790 -
val_loss: 2.2237 - val_accuracy: 0.6745
Epoch 47/50
625/625 [==============================] - 41s 65ms/step - loss: 1.7886 - accuracy: 0.7839 -
val_loss: 2.2439 - val_accuracy: 0.6805
Epoch 48/50
625/625 [==============================] - 39s 63ms/step - loss: 1.7963 - accuracy: 0.7883 -
val_loss: 2.2492 - val_accuracy: 0.6786
Epoch 49/50
625/625 [==============================] - 40s 64ms/step - loss: 1.7913 - accuracy: 0.7905 -
val_loss: 2.2930 - val_accuracy: 0.6825
Epoch 50/50
625/625 [==============================] - 40s 63ms/step - loss: 1.8114 - accuracy: 0.7930 -
val_loss: 2.2277 - val_accuracy: 0.6956
```

In [ ]:
```
print(storeResult(customVGG16AugModelHistory))
plot_loss_curve(customVGG16AugModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'CustomVGG16Aug', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 1.811427354812
622, 'Val Loss': 2.227701187133789, 'Train Acc': 0.7929999828338623, 'Val Acc': 0.69559997320
17517, '[Train - Val] Acc': 0.0974000096321106}
```



Observations

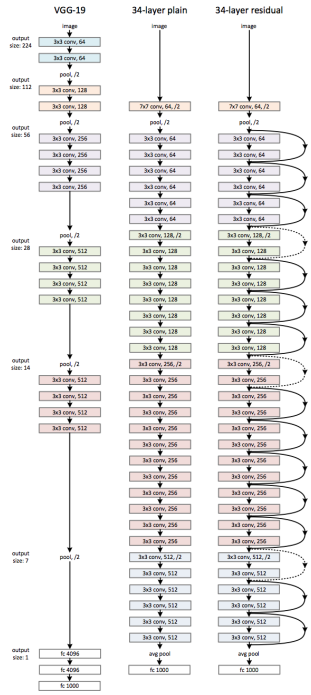
We note that the model reduced a lot of overfitting and become very generalise and performs better on the augmented dataset compared to the model that was trained without data augmentation. This means data augmentation for this model is very good.

CustomResNet Model

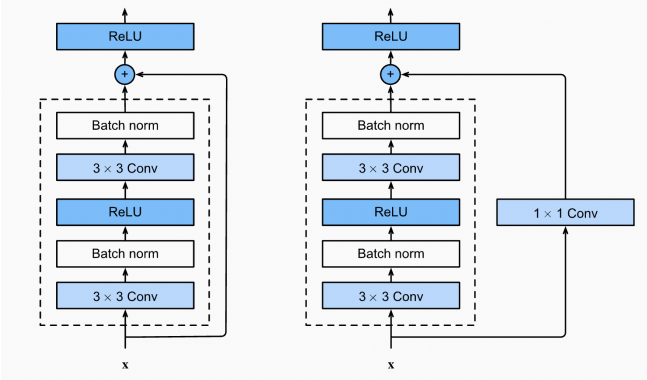
ResNets are called Residual Networks. ResNet is a special type of convolutional neural network (CNN). It was first introduced in 2015 by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their paper –

"Deep Residual Learning for Image Recognition".

A ResNet model can be called an upgraded version of the VGG architecture with some differences. The ResNet model will skip connections. The following image shows the difference between the ResNet and VGG model as well as a basic conv2D neural network.



As we can see from the diagram, we can see that the ResNet model has skip connections and it jumps the gun between it's layers. So what is the purpose? There are issues with classic neural networks called the vanishing gradient problem. With more layers being added to a neural network, the performance starts dropping due to the aforementioned vanishing gradient problem. To solve this issue, skipping connections [skipping layers] allows us to avoid the vanishing gradient problem.



As we can see from the image above, there are 2 types of skip connections, an Identity block [left side] and a Bottleneck / Convolutional block [right side]. The difference is that the Identity block directly adds the residue to the output whereas, the Convolutional block performs a convolution followed by Batch Normalisation on the residue before adding it to the output.

As there are many iterations of the ResNet model, and we found out the main features of ResNet network. We will be coding a small custom ResNet-10 [Number represents number of layers not inclusive of the convolutional blocks [Skip Connection Conv2D]] model based on [https://d2l.ai/chapter_convolutional-modern/resnet.html]

In [ ]:
```
def identity_block(x, filter, weight_decay=0.005):
    x_skip = x
```

```
    x = Conv2D(filter, (3, 3), padding='same',
               kernel_regularizer=l2(weight_decay))(x)
    x = BatchNormalization(axis=3)(x)
    x = Activation('relu')(x)
    x = Conv2D(filter, (3, 3), padding='same',
               kernel_regularizer=l2(weight_decay))(x)
    x = BatchNormalization(axis=3)(x)
    x = Add()([x, x_skip])
    x = Activation('relu')(x)
    return x
```

In [ ]:
```
def convolutional_block(x, filter, weight_decay=0.005):
    x_skip = x
    x = Conv2D(filter, (3, 3), padding='same', strides=(
        2, 2), kernel_regularizer=l2(weight_decay))(x)
    x = BatchNormalization(axis=3)(x)
    x = Activation('relu')(x)
    x = Conv2D(filter, (3, 3), padding='same')(x)
    x = BatchNormalization(axis=3)(x)
    x_skip = Conv2D(filter, (1, 1), strides=(2, 2),
                    kernel_regularizer=l2(weight_decay))(x_skip)
    x = Add()([x, x_skip])
    x = Activation('relu')(x)
    return x
```

Training CustomResNet model without Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
weight_decay=0.005
x = pre_processing_v1(inputs)
x = ZeroPadding2D((3, 3))(x)
x = Conv2D(64, kernel_size=7, strides=2, padding='same', kernel_regularizer=l2(weight_decay))
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = MaxPool2D(pool_size=3, strides=2, padding='same')(x)
block_layers = [1, 1, 1, 1]
filter_size = 64
for i in range(4):
    if i == 0:
        # For sub-block 1 Residual/Convolutional block not needed
        for j in range(block_layers[i]):
            x = identity_block(x, filter_size)
    else:
        # One Residual/Convolutional Block followed by Identity blocks
        # The filter size will go on increasing by a factor of 2
        filter_size = filter_size*2
        x = convolutional_block(x, filter_size)
        for j in range(block_layers[i] - 1):
            x = identity_block(x, filter_size)
x = AveragePooling2D((2, 2), padding='same')(x)
x = Flatten()(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customResNetModel = Model(inputs=inputs, outputs=x, name="CustomResNet")
customResNetModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                          loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
customResNetModelHistory = customResNetModel.fit(x_train, y_train, epochs=50,
                                                  validation_data=(x_val, y_val), batch_size=BAT
```

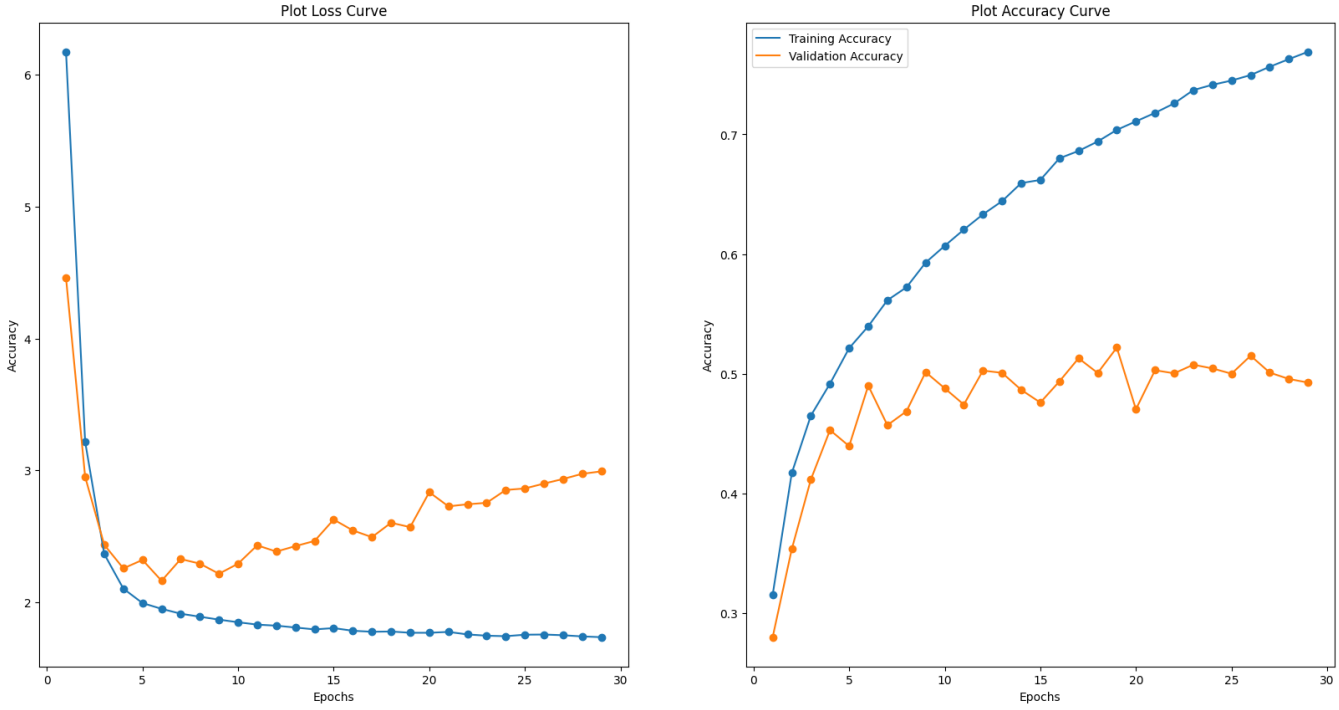
```
Epoch 1/50
625/625 [==============================] - 21s 29ms/step - loss: 6.1738 - accuracy: 0.3154 -
val_loss: 4.4602 - val_accuracy: 0.2800
Epoch 2/50
625/625 [==============================] - 16s 26ms/step - loss: 3.2171 - accuracy: 0.4171 -
val_loss: 2.9511 - val_accuracy: 0.3536
Epoch 3/50
625/625 [==============================] - 17s 27ms/step - loss: 2.3653 - accuracy: 0.4651 -
val_loss: 2.4351 - val_accuracy: 0.4118
Epoch 4/50
625/625 [==============================] - 16s 26ms/step - loss: 2.1029 - accuracy: 0.4918 -
val_loss: 2.2565 - val_accuracy: 0.4530
Epoch 5/50
625/625 [==============================] - 16s 26ms/step - loss: 1.9935 - accuracy: 0.5214 -
val_loss: 2.3207 - val_accuracy: 0.4396
Epoch 6/50
625/625 [==============================] - 16s 25ms/step - loss: 1.9496 - accuracy: 0.5397 -
val_loss: 2.1622 - val_accuracy: 0.4901
Epoch 7/50
625/625 [==============================] - 16s 26ms/step - loss: 1.9128 - accuracy: 0.5615 -
val_loss: 2.3272 - val_accuracy: 0.4571
Epoch 8/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8901 - accuracy: 0.5723 -
val_loss: 2.2925 - val_accuracy: 0.4687
Epoch 9/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8678 - accuracy: 0.5929 -
val_loss: 2.2157 - val_accuracy: 0.5014
Epoch 10/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8481 - accuracy: 0.6069 -
val_loss: 2.2923 - val_accuracy: 0.4879
Epoch 11/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8301 - accuracy: 0.6205 -
val_loss: 2.4313 - val_accuracy: 0.4743
Epoch 12/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8221 - accuracy: 0.6332 -
val_loss: 2.3840 - val_accuracy: 0.5027
Epoch 13/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8081 - accuracy: 0.6442 -
val_loss: 2.4254 - val_accuracy: 0.5008
Epoch 14/50
625/625 [==============================] - 16s 25ms/step - loss: 1.7938 - accuracy: 0.6593 -
val_loss: 2.4645 - val_accuracy: 0.4866
Epoch 15/50
625/625 [==============================] - 16s 25ms/step - loss: 1.8036 - accuracy: 0.6619 -
val_loss: 2.6267 - val_accuracy: 0.4760
Epoch 16/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7836 - accuracy: 0.6801 -
val_loss: 2.5445 - val_accuracy: 0.4937
Epoch 17/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7759 - accuracy: 0.6862 -
val_loss: 2.4933 - val_accuracy: 0.5130
Epoch 18/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7779 - accuracy: 0.6941 -
val_loss: 2.6023 - val_accuracy: 0.5007
Epoch 19/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7683 - accuracy: 0.7038 -
val_loss: 2.5694 - val_accuracy: 0.5219
Epoch 20/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7682 - accuracy: 0.7109 -
val_loss: 2.8342 - val_accuracy: 0.4704
Epoch 21/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7748 - accuracy: 0.7180 -
val_loss: 2.7270 - val_accuracy: 0.5030
Epoch 22/50
625/625 [==============================] - 17s 27ms/step - loss: 1.7556 - accuracy: 0.7259 -
val_loss: 2.7426 - val_accuracy: 0.5004
```

```
Epoch 23/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7462 - accuracy: 0.7370 -
val_loss: 2.7537 - val_accuracy: 0.5076
Epoch 24/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7423 - accuracy: 0.7414 -
val_loss: 2.8511 - val_accuracy: 0.5045
Epoch 25/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7536 - accuracy: 0.7450 -
val_loss: 2.8625 - val_accuracy: 0.5001
Epoch 26/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7550 - accuracy: 0.7496 -
val_loss: 2.8995 - val_accuracy: 0.5151
Epoch 27/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7496 - accuracy: 0.7565 -
val_loss: 2.9345 - val_accuracy: 0.5010
Epoch 28/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7407 - accuracy: 0.7629 -
val_loss: 2.9737 - val_accuracy: 0.4957
Epoch 29/50
625/625 [==============================] - 16s 26ms/step - loss: 1.7348 - accuracy: 0.7690 -
val_loss: 2.9920 - val_accuracy: 0.4927
```

In [ ]:
```
print(storeResult(customResNetModelHistory))
plot_loss_curve(customResNetModelHistory)
plt.show()
```

```
{'Model Name': 'CustomResNet', 'Epochs': 29, 'Batch Size': 64, 'Train Loss': 1.76825976371765
14, 'Val Loss': 2.5694122314453125, 'Train Acc': 0.7037749886512756, 'Val Acc': 0.52189999818
80188, '[Train - Val] Acc': 0.18187499046325684}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```



Observations

Comparing the customResNet model to the baseline model, we can see that both train and validation accuracy increased. Train loss is decreased as well which means that the model is generalise to the training data. However, more tuning will be need to change and modify to reduce the loss.

Training CustomResNet model with Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
weight_decay=0.005
x = pre_processing_v1(inputs)
x = ZeroPadding2D((3, 3))(x)
x = Conv2D(64, kernel_size=7, strides=2, padding='same', kernel_regularizer=l2(weight_decay))
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = MaxPool2D(pool_size=3, strides=2, padding='same')(x)
block_layers = [1, 1, 1, 1]
filter_size = 64
for i in range(4):
    if i == 0:
        # For sub-block 1 Residual/Convolutional block not needed
        for j in range(block_layers[i]):
            x = identity_block(x, filter_size)
    else:
        # One Residual/Convolutional Block followed by Identity blocks
        # The filter size will go on increasing by a factor of 2
        filter_size = filter_size*2
        x = convolutional_block(x, filter_size)
        for j in range(block_layers[i] - 1):
            x = identity_block(x, filter_size)
x = AveragePooling2D((2, 2), padding='same')(x)
x = Flatten()(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customResNetAugModel = Model(inputs=inputs, outputs=x, name="CustomResNetAug")
customResNetAugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                             loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
customResNetAugModelHistory = customResNetAugModel.fit(x_train_aug, y_train, epochs=50,
                                                        validation_data=(x_val, y_val), batch_size=BAT
```

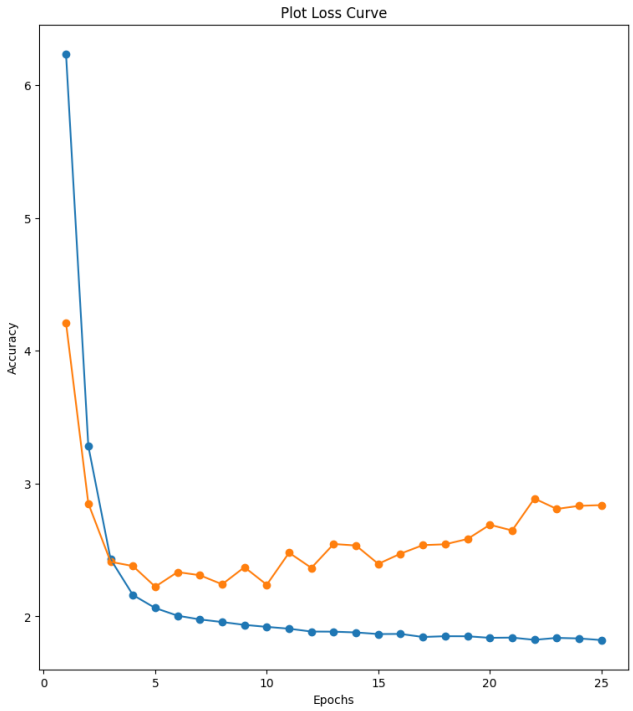
```
Epoch 1/50
625/625 [==============================] - 20s 29ms/step - loss: 6.2349 - accuracy: 0.2952 -
val_loss: 4.2101 - val_accuracy: 0.3454
Epoch 2/50
625/625 [==============================] - 16s 26ms/step - loss: 3.2823 - accuracy: 0.3865 -
val_loss: 2.8508 - val_accuracy: 0.3695
Epoch 3/50
625/625 [==============================] - 16s 26ms/step - loss: 2.4298 - accuracy: 0.4358 -
val_loss: 2.4110 - val_accuracy: 0.4050
Epoch 4/50
625/625 [==============================] - 16s 26ms/step - loss: 2.1612 - accuracy: 0.4682 -
val_loss: 2.3793 - val_accuracy: 0.4139
Epoch 5/50
625/625 [==============================] - 16s 26ms/step - loss: 2.0632 - accuracy: 0.4911 -
val_loss: 2.2238 - val_accuracy: 0.4499
Epoch 6/50
625/625 [==============================] - 16s 26ms/step - loss: 2.0057 - accuracy: 0.5139 -
val_loss: 2.3339 - val_accuracy: 0.4176
Epoch 7/50
625/625 [==============================] - 16s 26ms/step - loss: 1.9773 - accuracy: 0.5335 -
val_loss: 2.3101 - val_accuracy: 0.4535
Epoch 8/50
625/625 [==============================] - 17s 26ms/step - loss: 1.9577 - accuracy: 0.5497 -
val_loss: 2.2419 - val_accuracy: 0.4795
Epoch 9/50
625/625 [==============================] - 16s 26ms/step - loss: 1.9363 - accuracy: 0.5652 -
val_loss: 2.3705 - val_accuracy: 0.4578
Epoch 10/50
625/625 [==============================] - 16s 26ms/step - loss: 1.9214 - accuracy: 0.5804 -
val_loss: 2.2384 - val_accuracy: 0.5010
Epoch 11/50
625/625 [==============================] - 16s 26ms/step - loss: 1.9076 - accuracy: 0.5932 -
val_loss: 2.4796 - val_accuracy: 0.4677
Epoch 12/50
625/625 [==============================] - 17s 27ms/step - loss: 1.8860 - accuracy: 0.6052 -
val_loss: 2.3654 - val_accuracy: 0.4855
Epoch 13/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8852 - accuracy: 0.6147 -
val_loss: 2.5454 - val_accuracy: 0.4653
Epoch 14/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8795 - accuracy: 0.6274 -
val_loss: 2.5338 - val_accuracy: 0.4794
Epoch 15/50
625/625 [==============================] - 17s 26ms/step - loss: 1.8672 - accuracy: 0.6365 -
val_loss: 2.3956 - val_accuracy: 0.5138
Epoch 16/50
625/625 [==============================] - 17s 26ms/step - loss: 1.8682 - accuracy: 0.6479 -
val_loss: 2.4726 - val_accuracy: 0.4987
Epoch 17/50
625/625 [==============================] - 17s 27ms/step - loss: 1.8451 - accuracy: 0.6591 -
val_loss: 2.5367 - val_accuracy: 0.5005
Epoch 18/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8510 - accuracy: 0.6663 -
val_loss: 2.5432 - val_accuracy: 0.5063
Epoch 19/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8505 - accuracy: 0.6770 -
val_loss: 2.5828 - val_accuracy: 0.5021
Epoch 20/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8390 - accuracy: 0.6880 -
val_loss: 2.6899 - val_accuracy: 0.4906
Epoch 21/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8406 - accuracy: 0.6950 -
val_loss: 2.6470 - val_accuracy: 0.5055
Epoch 22/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8236 - accuracy: 0.7083 -
val_loss: 2.8875 - val_accuracy: 0.4835
```

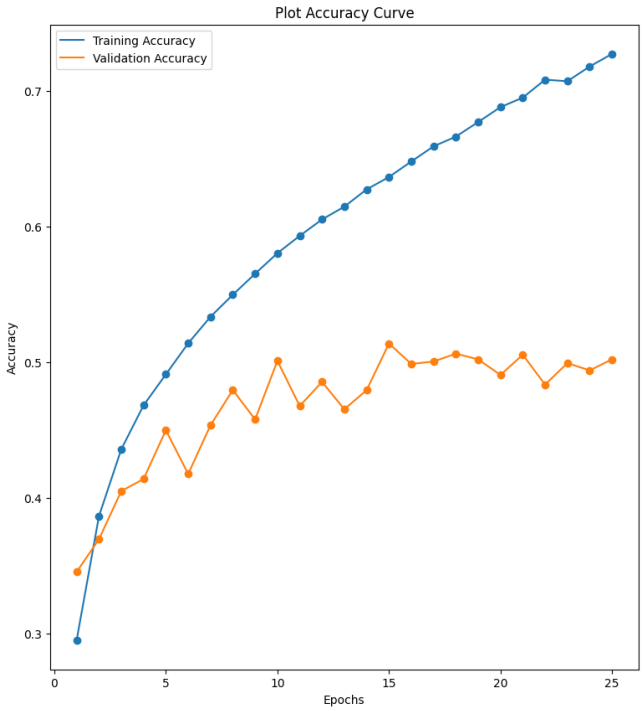
```
Epoch 23/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8384 - accuracy: 0.7071 -
val_loss: 2.8091 - val_accuracy: 0.4993
Epoch 24/50
625/625 [==============================] - 17s 26ms/step - loss: 1.8342 - accuracy: 0.7180 -
val_loss: 2.8323 - val_accuracy: 0.4940
Epoch 25/50
625/625 [==============================] - 16s 26ms/step - loss: 1.8219 - accuracy: 0.7273 -
val_loss: 2.8379 - val_accuracy: 0.5021
```

In [ ]:
```
print(storeResult(customResNetAugModelHistory))
plot_loss_curve(customResNetAugModelHistory)
plt.show()
```

```
{'Model Name': 'CustomResNetAug', 'Epochs': 25, 'Batch Size': 64, 'Train Loss': 1.86721193790
4358, 'Val Loss': 2.3955774307250977, 'Train Acc': 0.6365000009536743, 'Val Acc': 0.513800024
9862671, '[Train - Val] Acc': 0.12269997596740723}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```





CustomResNet model with Dropout Layers

To help reduce the overfitting and reduce the loss of the model, we will be using dropout layers. Dropout layers will randomly sets input units to 0 during the training period. This is means that weights and biases that might affect the model and cause the model to overfit might be ignore (disconnected)

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
weight_decay = 0.005
x = pre_processing_v1(inputs)
x = ZeroPadding2D((3, 3))(x)
x = Conv2D(64, kernel_size=7, strides=2, padding='same',
           kernel_regularizer=l2(weight_decay))(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = MaxPool2D(pool_size=3, strides=2, padding='same')(x)
block_layers = [3, 4, 6, 3]
filter_size = 64
for i in range(4):
    if i == 0:
```

```
        # For sub-block 1 Residual/Convolutional block not needed
        for j in range(block_layers[i]):
            x = identity_block(x, filter_size)
    else:
        # One Residual/Convolutional Block followed by Identity blocks
        # The filter size will go on increasing by a factor of 2
        filter_size = filter_size*2
        x = convolutional_block(x, filter_size)
        for j in range(block_layers[i] - 1):
            x = identity_block(x, filter_size)
    x = Dropout(0.3)(x)
x = AveragePooling2D((2, 2), padding='same')(x)
x = Flatten()(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customResNetDropModel = Model(
    inputs=inputs, outputs=x, name="CustomResNetDropV2")
customResNetDropModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                              loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
customResNetDropModel.summary()
```

```
Model: "CustomResNetDropV2"
_____
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0         []

 normalization (Normalization)  (None, 32, 32, 3)      7         ['input_1[0][0]']

 zero_padding2d (ZeroPadding2D)  (None, 38, 38, 3)     0         ['normalization[17][0]']

 conv2d (Conv2D)             (None, 19, 19, 64)        9472      ['zero_padding2d[0][0]']

 batch_normalization (BatchNorm  (None, 19, 19, 64)    256       ['conv2d[0][0]']
 alization)

 activation (Activation)     (None, 19, 19, 64)        0         ['batch_normalization[0]
[0]']

 max_pooling2d (MaxPooling2D)   (None, 10, 10, 64)     0         ['activation[0][0]']

 conv2d_1 (Conv2D)           (None, 10, 10, 64)        36928     ['max_pooling2d[0][0]']

 batch_normalization_1 (BatchNo  (None, 10, 10, 64)    256       ['conv2d_1[0][0]']
 rmalization)

 activation_1 (Activation)   (None, 10, 10, 64)        0         ['batch_normalization_1[0]
[0]']

 conv2d_2 (Conv2D)           (None, 10, 10, 64)        36928     ['activation_1[0][0]']

 batch_normalization_2 (BatchNo  (None, 10, 10, 64)    256       ['conv2d_2[0][0]']
 rmalization)

 add (Add)                   (None, 10, 10, 64)        0         ['batch_normalization_2[0]
[0]',
                                                                  'max_pooling2d[0][0]']

 activation_2 (Activation)   (None, 10, 10, 64)        0         ['add[0][0]']

 conv2d_3 (Conv2D)           (None, 10, 10, 64)        36928     ['activation_2[0][0]']

 batch_normalization_3 (BatchNo  (None, 10, 10, 64)    256       ['conv2d_3[0][0]']
 rmalization)

 activation_3 (Activation)   (None, 10, 10, 64)        0         ['batch_normalization_3[0]
[0]']

 conv2d_4 (Conv2D)           (None, 10, 10, 64)        36928     ['activation_3[0][0]']

 batch_normalization_4 (BatchNo  (None, 10, 10, 64)    256       ['conv2d_4[0][0]']
 rmalization)

 add_1 (Add)                 (None, 10, 10, 64)        0         ['batch_normalization_4[0]
[0]',
                                                                  'activation_2[0][0]']

 activation_4 (Activation)   (None, 10, 10, 64)        0         ['add_1[0][0]']

 conv2d_5 (Conv2D)           (None, 10, 10, 64)        36928     ['activation_4[0][0]']

 batch_normalization_5 (BatchNo  (None, 10, 10, 64)    256       ['conv2d_5[0][0]']
 rmalization)

 activation_5 (Activation)   (None, 10, 10, 64)        0         ['batch_normalization_5[0]
```

```
[0]']

 conv2d_6 (Conv2D)              (None, 10, 10, 64)   36928       ['activation_5[0][0]']

 batch_normalization_6 (BatchNo  (None, 10, 10, 64)  256         ['conv2d_6[0][0]']
 rmalization)

 add_2 (Add)                    (None, 10, 10, 64)   0           ['batch_normalization_6[0]
[0]',
                                                                  'activation_4[0][0]']

 activation_6 (Activation)      (None, 10, 10, 64)   0           ['add_2[0][0]']

 dropout (Dropout)              (None, 10, 10, 64)   0           ['activation_6[0][0]']

 conv2d_7 (Conv2D)              (None, 5, 5, 128)    73856       ['dropout[0][0]']

 batch_normalization_7 (BatchNo  (None, 5, 5, 128)   512         ['conv2d_7[0][0]']
 rmalization)

 activation_7 (Activation)      (None, 5, 5, 128)    0           ['batch_normalization_7[0]
[0]']

 conv2d_8 (Conv2D)              (None, 5, 5, 128)    147584      ['activation_7[0][0]']

 batch_normalization_8 (BatchNo  (None, 5, 5, 128)   512         ['conv2d_8[0][0]']
 rmalization)

 conv2d_9 (Conv2D)              (None, 5, 5, 128)    8320        ['dropout[0][0]']

 add_3 (Add)                    (None, 5, 5, 128)    0           ['batch_normalization_8[0]
[0]',
                                                                  'conv2d_9[0][0]']

 activation_8 (Activation)      (None, 5, 5, 128)    0           ['add_3[0][0]']

 conv2d_10 (Conv2D)             (None, 5, 5, 128)    147584      ['activation_8[0][0]']

 batch_normalization_9 (BatchNo  (None, 5, 5, 128)   512         ['conv2d_10[0][0]']
 rmalization)

 activation_9 (Activation)      (None, 5, 5, 128)    0           ['batch_normalization_9[0]
[0]']

 conv2d_11 (Conv2D)             (None, 5, 5, 128)    147584      ['activation_9[0][0]']

 batch_normalization_10 (BatchN  (None, 5, 5, 128)   512         ['conv2d_11[0][0]']
 ormalization)

 add_4 (Add)                    (None, 5, 5, 128)    0           ['batch_normalization_10[0]
[0]',
                                                                  'activation_8[0][0]']

 activation_10 (Activation)     (None, 5, 5, 128)    0           ['add_4[0][0]']

 conv2d_12 (Conv2D)             (None, 5, 5, 128)    147584      ['activation_10[0][0]']

 batch_normalization_11 (BatchN  (None, 5, 5, 128)   512         ['conv2d_12[0][0]']
 ormalization)

 activation_11 (Activation)     (None, 5, 5, 128)    0           ['batch_normalization_11[0]
[0]']

 conv2d_13 (Conv2D)             (None, 5, 5, 128)    147584      ['activation_11[0][0]']

 batch_normalization_12 (BatchN  (None, 5, 5, 128)   512         ['conv2d_13[0][0]']
```

```
 ormalization)

 add_5 (Add)                    (None, 5, 5, 128)    0           ['batch_normalization_12[0]
[0]',
                                                                  'activation_10[0][0]']

 activation_12 (Activation)     (None, 5, 5, 128)    0           ['add_5[0][0]']

 conv2d_14 (Conv2D)             (None, 5, 5, 128)    147584      ['activation_12[0][0]']

 batch_normalization_13 (BatchN  (None, 5, 5, 128)   512         ['conv2d_14[0][0]']
 ormalization)

 activation_13 (Activation)     (None, 5, 5, 128)    0           ['batch_normalization_13[0]
[0]']

 conv2d_15 (Conv2D)             (None, 5, 5, 128)    147584      ['activation_13[0][0]']

 batch_normalization_14 (BatchN  (None, 5, 5, 128)   512         ['conv2d_15[0][0]']
 ormalization)

 add_6 (Add)                    (None, 5, 5, 128)    0           ['batch_normalization_14[0]
[0]',
                                                                  'activation_12[0][0]']

 activation_14 (Activation)     (None, 5, 5, 128)    0           ['add_6[0][0]']

 dropout_1 (Dropout)            (None, 5, 5, 128)    0           ['activation_14[0][0]']

 conv2d_16 (Conv2D)             (None, 3, 3, 256)    295168      ['dropout_1[0][0]']

 batch_normalization_15 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_16[0][0]']
 ormalization)

 activation_15 (Activation)     (None, 3, 3, 256)    0           ['batch_normalization_15[0]
[0]']

 conv2d_17 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_15[0][0]']

 batch_normalization_16 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_17[0][0]']
 ormalization)

 conv2d_18 (Conv2D)             (None, 3, 3, 256)    33024       ['dropout_1[0][0]']

 add_7 (Add)                    (None, 3, 3, 256)    0           ['batch_normalization_16[0]
[0]',
                                                                  'conv2d_18[0][0]']

 activation_16 (Activation)     (None, 3, 3, 256)    0           ['add_7[0][0]']

 conv2d_19 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_16[0][0]']

 batch_normalization_17 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_19[0][0]']
 ormalization)

 activation_17 (Activation)     (None, 3, 3, 256)    0           ['batch_normalization_17[0]
[0]']

 conv2d_20 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_17[0][0]']

 batch_normalization_18 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_20[0][0]']
 ormalization)

 add_8 (Add)                    (None, 3, 3, 256)    0           ['batch_normalization_18[0]
[0]',
                                                                  'activation_16[0][0]']
```

```
 activation_18 (Activation)     (None, 3, 3, 256)    0           ['add_8[0][0]']

 conv2d_21 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_18[0][0]']

 batch_normalization_19 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_21[0][0]']
 ormalization)

 activation_19 (Activation)     (None, 3, 3, 256)    0           ['batch_normalization_19[0]
[0]']

 conv2d_22 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_19[0][0]']

 batch_normalization_20 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_22[0][0]']
 ormalization)

 add_9 (Add)                    (None, 3, 3, 256)    0           ['batch_normalization_20[0]
[0]',
                                                                  'activation_18[0][0]']

 activation_20 (Activation)     (None, 3, 3, 256)    0           ['add_9[0][0]']

 conv2d_23 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_20[0][0]']

 batch_normalization_21 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_23[0][0]']
 ormalization)

 activation_21 (Activation)     (None, 3, 3, 256)    0           ['batch_normalization_21[0]
[0]']

 conv2d_24 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_21[0][0]']

 batch_normalization_22 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_24[0][0]']
 ormalization)

 add_10 (Add)                   (None, 3, 3, 256)    0           ['batch_normalization_22[0]
[0]',
                                                                  'activation_20[0][0]']

 activation_22 (Activation)     (None, 3, 3, 256)    0           ['add_10[0][0]']

 conv2d_25 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_22[0][0]']

 batch_normalization_23 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_25[0][0]']
 ormalization)

 activation_23 (Activation)     (None, 3, 3, 256)    0           ['batch_normalization_23[0]
[0]']

 conv2d_26 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_23[0][0]']

 batch_normalization_24 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_26[0][0]']
 ormalization)

 add_11 (Add)                   (None, 3, 3, 256)    0           ['batch_normalization_24[0]
[0]',
                                                                  'activation_22[0][0]']

 activation_24 (Activation)     (None, 3, 3, 256)    0           ['add_11[0][0]']

 conv2d_27 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_24[0][0]']

 batch_normalization_25 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_27[0][0]']
 ormalization)

 activation_25 (Activation)     (None, 3, 3, 256)    0           ['batch_normalization_25[0]
```

```
[0]']

 conv2d_28 (Conv2D)             (None, 3, 3, 256)    590080      ['activation_25[0][0]']

 batch_normalization_26 (BatchN  (None, 3, 3, 256)   1024        ['conv2d_28[0][0]']
 ormalization)

 add_12 (Add)                   (None, 3, 3, 256)    0           ['batch_normalization_26[0]
[0]',
                                                                  'activation_24[0][0]']

 activation_26 (Activation)     (None, 3, 3, 256)    0           ['add_12[0][0]']

 dropout_2 (Dropout)            (None, 3, 3, 256)    0           ['activation_26[0][0]']

 conv2d_29 (Conv2D)             (None, 2, 2, 512)    1180160     ['dropout_2[0][0]']

 batch_normalization_27 (BatchN  (None, 2, 2, 512)   2048        ['conv2d_29[0][0]']
 ormalization)

 activation_27 (Activation)     (None, 2, 2, 512)    0           ['batch_normalization_27[0]
[0]']

 conv2d_30 (Conv2D)             (None, 2, 2, 512)    2359808     ['activation_27[0][0]']

 batch_normalization_28 (BatchN  (None, 2, 2, 512)   2048        ['conv2d_30[0][0]']
 ormalization)

 conv2d_31 (Conv2D)             (None, 2, 2, 512)    131584      ['dropout_2[0][0]']

 add_13 (Add)                   (None, 2, 2, 512)    0           ['batch_normalization_28[0]
[0]',
                                                                  'conv2d_31[0][0]']

 activation_28 (Activation)     (None, 2, 2, 512)    0           ['add_13[0][0]']

 conv2d_32 (Conv2D)             (None, 2, 2, 512)    2359808     ['activation_28[0][0]']

 batch_normalization_29 (BatchN  (None, 2, 2, 512)   2048        ['conv2d_32[0][0]']
 ormalization)

 activation_29 (Activation)     (None, 2, 2, 512)    0           ['batch_normalization_29[0]
[0]']

 conv2d_33 (Conv2D)             (None, 2, 2, 512)    2359808     ['activation_29[0][0]']

 batch_normalization_30 (BatchN  (None, 2, 2, 512)   2048        ['conv2d_33[0][0]']
 ormalization)

 add_14 (Add)                   (None, 2, 2, 512)    0           ['batch_normalization_30[0]
[0]',
                                                                  'activation_28[0][0]']

 activation_30 (Activation)     (None, 2, 2, 512)    0           ['add_14[0][0]']

 conv2d_34 (Conv2D)             (None, 2, 2, 512)    2359808     ['activation_30[0][0]']

 batch_normalization_31 (BatchN  (None, 2, 2, 512)   2048        ['conv2d_34[0][0]']
 ormalization)

 activation_31 (Activation)     (None, 2, 2, 512)    0           ['batch_normalization_31[0]
[0]']

 conv2d_35 (Conv2D)             (None, 2, 2, 512)    2359808     ['activation_31[0][0]']

 batch_normalization_32 (BatchN  (None, 2, 2, 512)   2048        ['conv2d_35[0][0]']
```

```
 ormalization)

 add_15 (Add)                   (None, 2, 2, 512)    0           ['batch_normalization_32[0]
[0]',
                                                                  'activation_30[0][0]']

 activation_32 (Activation)     (None, 2, 2, 512)    0           ['add_15[0][0]']

 dropout_3 (Dropout)            (None, 2, 2, 512)    0           ['activation_32[0][0]']

 average_pooling2d (AveragePool  (None, 1, 1, 512)   0           ['dropout_3[0][0]']
 ing2D)

 flatten (Flatten)              (None, 512)          0           ['average_pooling2d[0][0]']

 dense (Dense)                  (None, 20)           10260       ['flatten[0][0]']

==================================================================================================
=====
Total params: 21,316,891
Trainable params: 21,301,652
Non-trainable params: 15,239
__________________________________________________________________________________________________
_____
```

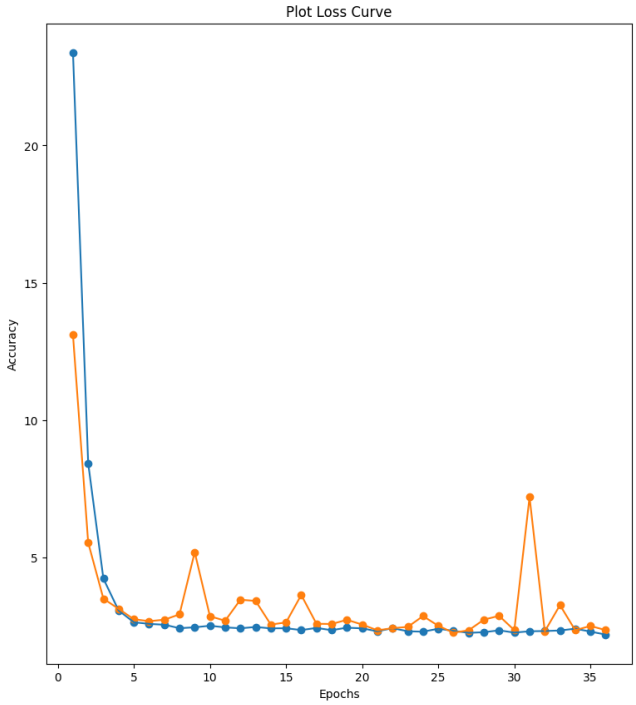
In [ ]:
```
customResNetDropModelHistory = customResNetDropModel.fit(x_train, y_train, epochs=50,
                                                validation_data=(x_val, y_val), batch_size=BAT
```

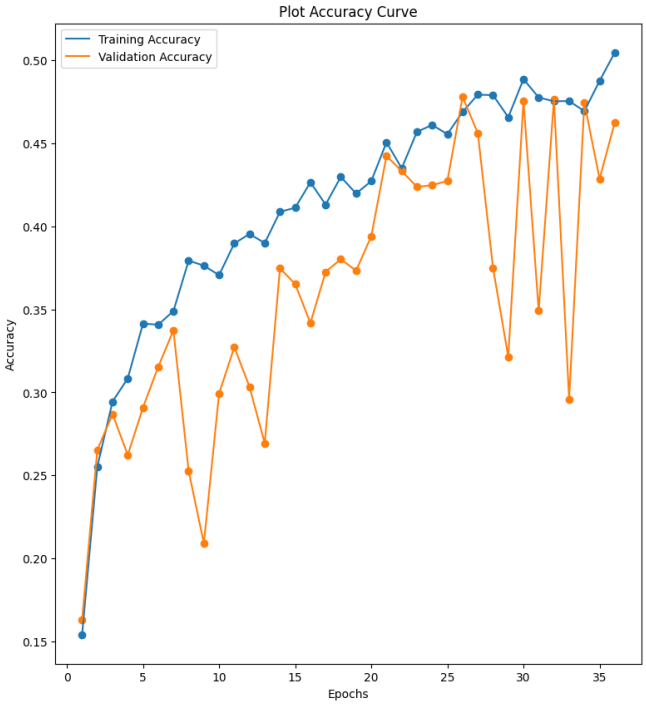
```
Epoch 1/50
625/625 [==============================] - 57s 83ms/step - loss: 23.3866 - accuracy: 0.1541 -
val_loss: 13.0995 - val_accuracy: 0.1630
Epoch 2/50
625/625 [==============================] - 50s 80ms/step - loss: 8.4237 - accuracy: 0.2553 -
val_loss: 5.5296 - val_accuracy: 0.2654
Epoch 3/50
625/625 [==============================] - 50s 81ms/step - loss: 4.2153 - accuracy: 0.2945 -
val_loss: 3.4863 - val_accuracy: 0.2869
Epoch 4/50
625/625 [==============================] - 50s 80ms/step - loss: 3.0559 - accuracy: 0.3085 -
val_loss: 3.1155 - val_accuracy: 0.2623
Epoch 5/50
625/625 [==============================] - 50s 80ms/step - loss: 2.6269 - accuracy: 0.3415 -
val_loss: 2.7369 - val_accuracy: 0.2909
Epoch 6/50
625/625 [==============================] - 50s 80ms/step - loss: 2.5684 - accuracy: 0.3408 -
val_loss: 2.6660 - val_accuracy: 0.3154
Epoch 7/50
625/625 [==============================] - 50s 81ms/step - loss: 2.5362 - accuracy: 0.3490 -
val_loss: 2.7191 - val_accuracy: 0.3374
Epoch 8/50
625/625 [==============================] - 50s 80ms/step - loss: 2.4108 - accuracy: 0.3794 -
val_loss: 2.9136 - val_accuracy: 0.2527
Epoch 9/50
625/625 [==============================] - 51s 81ms/step - loss: 2.4431 - accuracy: 0.3764 -
val_loss: 5.1783 - val_accuracy: 0.2090
Epoch 10/50
625/625 [==============================] - 52s 84ms/step - loss: 2.4996 - accuracy: 0.3706 -
val_loss: 2.8454 - val_accuracy: 0.2993
Epoch 11/50
625/625 [==============================] - 52s 83ms/step - loss: 2.4401 - accuracy: 0.3898 -
val_loss: 2.6824 - val_accuracy: 0.3273
Epoch 12/50
625/625 [==============================] - 51s 82ms/step - loss: 2.4074 - accuracy: 0.3954 -
val_loss: 3.4469 - val_accuracy: 0.3033
Epoch 13/50
625/625 [==============================] - 51s 82ms/step - loss: 2.4548 - accuracy: 0.3900 -
val_loss: 3.4048 - val_accuracy: 0.2693
Epoch 14/50
625/625 [==============================] - 51s 82ms/step - loss: 2.4046 - accuracy: 0.4088 -
val_loss: 2.5426 - val_accuracy: 0.3747
Epoch 15/50
625/625 [==============================] - 53s 84ms/step - loss: 2.4073 - accuracy: 0.4112 -
val_loss: 2.6205 - val_accuracy: 0.3653
Epoch 16/50
625/625 [==============================] - 52s 82ms/step - loss: 2.3433 - accuracy: 0.4266 -
val_loss: 3.6208 - val_accuracy: 0.3418
Epoch 17/50
625/625 [==============================] - 51s 82ms/step - loss: 2.4184 - accuracy: 0.4132 -
val_loss: 2.5710 - val_accuracy: 0.3726
Epoch 18/50
625/625 [==============================] - 52s 83ms/step - loss: 2.3390 - accuracy: 0.4299 -
val_loss: 2.5635 - val_accuracy: 0.3803
Epoch 19/50
625/625 [==============================] - 52s 82ms/step - loss: 2.4282 - accuracy: 0.4198 -
val_loss: 2.7160 - val_accuracy: 0.3733
Epoch 20/50
625/625 [==============================] - 52s 83ms/step - loss: 2.4056 - accuracy: 0.4273 -
val_loss: 2.5435 - val_accuracy: 0.3941
Epoch 21/50
625/625 [==============================] - 53s 85ms/step - loss: 2.2967 - accuracy: 0.4507 -
val_loss: 2.3336 - val_accuracy: 0.4424
Epoch 22/50
625/625 [==============================] - 52s 83ms/step - loss: 2.4111 - accuracy: 0.4350 -
val_loss: 2.4109 - val_accuracy: 0.4334
```

```
Epoch 23/50
625/625 [==============================] - 52s 83ms/step - loss: 2.2986 - accuracy: 0.4570 -
val_loss: 2.4579 - val_accuracy: 0.4238
Epoch 24/50
625/625 [==============================] - 52s 82ms/step - loss: 2.2823 - accuracy: 0.4611 -
val_loss: 2.8520 - val_accuracy: 0.4249
Epoch 25/50
625/625 [==============================] - 52s 83ms/step - loss: 2.3914 - accuracy: 0.4556 -
val_loss: 2.4987 - val_accuracy: 0.4274
Epoch 26/50
625/625 [==============================] - 52s 84ms/step - loss: 2.3126 - accuracy: 0.4690 -
val_loss: 2.2460 - val_accuracy: 0.4781
Epoch 27/50
625/625 [==============================] - 52s 84ms/step - loss: 2.2412 - accuracy: 0.4794 -
val_loss: 2.3347 - val_accuracy: 0.4562
Epoch 28/50
625/625 [==============================] - 52s 83ms/step - loss: 2.2642 - accuracy: 0.4791 -
val_loss: 2.7222 - val_accuracy: 0.3749
Epoch 29/50
625/625 [==============================] - 52s 82ms/step - loss: 2.3324 - accuracy: 0.4658 -
val_loss: 2.8608 - val_accuracy: 0.3214
Epoch 30/50
625/625 [==============================] - 52s 83ms/step - loss: 2.2501 - accuracy: 0.4888 -
val_loss: 2.3378 - val_accuracy: 0.4758
Epoch 31/50
625/625 [==============================] - 52s 82ms/step - loss: 2.2983 - accuracy: 0.4777 -
val_loss: 7.1896 - val_accuracy: 0.3492
Epoch 32/50
625/625 [==============================] - 52s 83ms/step - loss: 2.3067 - accuracy: 0.4754 -
val_loss: 2.2961 - val_accuracy: 0.4764
Epoch 33/50
625/625 [==============================] - 53s 85ms/step - loss: 2.3269 - accuracy: 0.4755 -
val_loss: 3.2514 - val_accuracy: 0.2958
Epoch 34/50
625/625 [==============================] - 51s 82ms/step - loss: 2.3936 - accuracy: 0.4694 -
val_loss: 2.3371 - val_accuracy: 0.4745
Epoch 35/50
625/625 [==============================] - 52s 83ms/step - loss: 2.2865 - accuracy: 0.4877 -
val_loss: 2.4920 - val_accuracy: 0.4287
Epoch 36/50
625/625 [==============================] - 52s 83ms/step - loss: 2.1773 - accuracy: 0.5048 -
val_loss: 2.3407 - val_accuracy: 0.4627
```

In [ ]:
```
print(storeResult(customResNetDropModelHistory))
plot_loss_curve(customResNetDropModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'CustomResNetDropV2', 'Epochs': 36, 'Batch Size': 64, 'Train Loss': 2.31256437
30163574, 'Val Loss': 2.2459845542907715, 'Train Acc': 0.4689500033855438, 'Val Acc': 0.47810
00018119812, '[Train - Val] Acc': -0.009149998426437378}
```





Observations

We can see that the model has a worst performance after applying dropout. This is likely because of the ResNet's batch normalization layers. When the two layers are put together, there will be a disharmony created. This is due too Batch Normalization layer having a normalization process which use the batch's mean and standard deviation. But the dropout layer will randomly drop the weights and bias in the network. This causes the model to lose the important weights [After the weights have been normalised] that is essential in the model's performance.

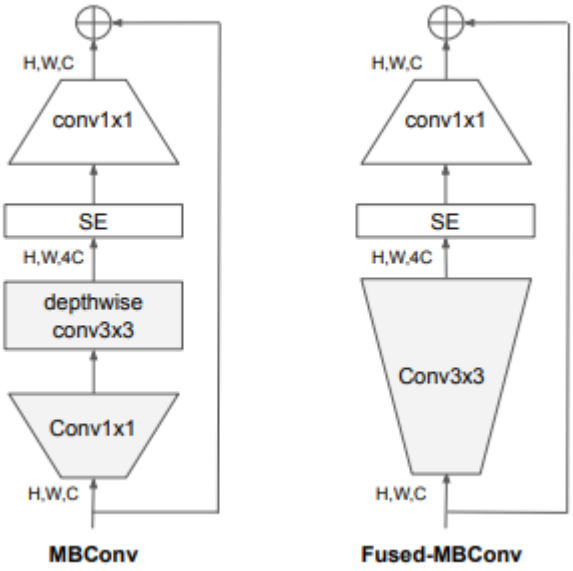
EfficientNetV2 Model

EfficientNet is a convolutional neural network architecture and scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient. Unlike conventional practice that arbitrary scales these factors, the EfficientNet scaling method uniformly scales network width, depth, and resolution with a set of fixed scaling coefficients.

However, the EfficientNet model is quite huge and comprehensive, we will be building a smaller scaled down version introduced by Ming Xing Tan called the EfficientNetV2 model.

Deep dive into the EfficientNetV2 network

The EfficientNetV2 model uses extensively the MBConv layer and the fused-MBConv layer as shown below. It also uses a smaller kernel size but adds more layers to compensate for the smaller kernel size.



Here is how the architecture looks like.

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

For this model, due to the higher complexity, we will be using a pre build model and removing all the weights and biases to train our own model using transfer learning

EfficientNetV2B0 model without Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = tf.keras.applications.efficientnet_v2.EfficientNetV2B0(
    include_top=False,
    weights='imagenet',
    input_shape=IMG_SIZE,
    pooling="max",
    include_preprocessing=False
)(x)
x = Flatten()(x)
x = Dense(NUM_CLASS, 'softmax')(x)
efficientNetModel = Model(
    inputs=inputs, outputs=x, name="efficientNetV2")
efficientNetModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                                loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
efficientNetModel.summary()
```

```
Model: "efficientNetV2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 normalization (Normalizatio  (None, 32, 32, 3)        7
 n)

 efficientnetv2-b0 (Function  (None, 1280)             5919312
 al)

 flatten (Flatten)           (None, 1280)              0

 dense (Dense)               (None, 20)                25620

=================================================================
Total params: 5,944,939
Trainable params: 5,884,324
Non-trainable params: 60,615
_________________________________________________________________
```

In [ ]:
```
efficientNetModelHistory = efficientNetModel.fit(x_train, y_train, epochs=50,
                                                 validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/50
625/625 [==============================] - 81s 109ms/step - loss: 2.0214 - accuracy: 0.3828 -
val_loss: 1.4868 - val_accuracy: 0.5292
Epoch 2/50
625/625 [==============================] - 67s 106ms/step - loss: 1.3683 - accuracy: 0.5642 -
val_loss: 1.2626 - val_accuracy: 0.5982
Epoch 3/50
625/625 [==============================] - 65s 104ms/step - loss: 1.1025 - accuracy: 0.6457 -
val_loss: 1.1320 - val_accuracy: 0.6400
Epoch 4/50
625/625 [==============================] - 65s 104ms/step - loss: 0.9173 - accuracy: 0.7020 -
val_loss: 1.1043 - val_accuracy: 0.6515
Epoch 5/50
625/625 [==============================] - 66s 105ms/step - loss: 0.7716 - accuracy: 0.7467 -
val_loss: 1.1132 - val_accuracy: 0.6594
Epoch 6/50
625/625 [==============================] - 66s 106ms/step - loss: 0.6573 - accuracy: 0.7846 -
val_loss: 1.1359 - val_accuracy: 0.6615
Epoch 7/50
625/625 [==============================] - 66s 106ms/step - loss: 0.5619 - accuracy: 0.8130 -
val_loss: 1.1492 - val_accuracy: 0.6699
Epoch 8/50
625/625 [==============================] - 65s 104ms/step - loss: 0.4794 - accuracy: 0.8394 -
val_loss: 1.2062 - val_accuracy: 0.6736
Epoch 9/50
625/625 [==============================] - 65s 103ms/step - loss: 0.4075 - accuracy: 0.8618 -
val_loss: 1.2239 - val_accuracy: 0.6731
Epoch 10/50
625/625 [==============================] - 65s 104ms/step - loss: 0.3480 - accuracy: 0.8826 -
val_loss: 1.2984 - val_accuracy: 0.6688
Epoch 11/50
625/625 [==============================] - 66s 106ms/step - loss: 0.3051 - accuracy: 0.8977 -
val_loss: 1.3233 - val_accuracy: 0.6675
Epoch 12/50
625/625 [==============================] - 65s 104ms/step - loss: 0.2685 - accuracy: 0.9091 -
val_loss: 1.3758 - val_accuracy: 0.6665
Epoch 13/50
625/625 [==============================] - 65s 104ms/step - loss: 0.2415 - accuracy: 0.9183 -
val_loss: 1.4177 - val_accuracy: 0.6736
Epoch 14/50
625/625 [==============================] - 65s 103ms/step - loss: 0.2110 - accuracy: 0.9294 -
val_loss: 1.5014 - val_accuracy: 0.6694
Epoch 15/50
625/625 [==============================] - 66s 106ms/step - loss: 0.1888 - accuracy: 0.9372 -
val_loss: 1.4581 - val_accuracy: 0.6784
Epoch 16/50
625/625 [==============================] - 67s 108ms/step - loss: 0.1743 - accuracy: 0.9430 -
val_loss: 1.5523 - val_accuracy: 0.6792
Epoch 17/50
625/625 [==============================] - 66s 105ms/step - loss: 0.1645 - accuracy: 0.9453 -
val_loss: 1.5362 - val_accuracy: 0.6793
Epoch 18/50
625/625 [==============================] - 65s 105ms/step - loss: 0.1452 - accuracy: 0.9513 -
val_loss: 1.5400 - val_accuracy: 0.6746
Epoch 19/50
625/625 [==============================] - 65s 104ms/step - loss: 0.1418 - accuracy: 0.9531 -
val_loss: 1.5848 - val_accuracy: 0.6753
Epoch 20/50
625/625 [==============================] - 65s 104ms/step - loss: 0.1333 - accuracy: 0.9557 -
val_loss: 1.5878 - val_accuracy: 0.6753
Epoch 21/50
625/625 [==============================] - 66s 105ms/step - loss: 0.1246 - accuracy: 0.9584 -
val_loss: 1.5747 - val_accuracy: 0.6824
Epoch 22/50
625/625 [==============================] - 65s 103ms/step - loss: 0.1099 - accuracy: 0.9639 -
val_loss: 1.6270 - val_accuracy: 0.6778
```

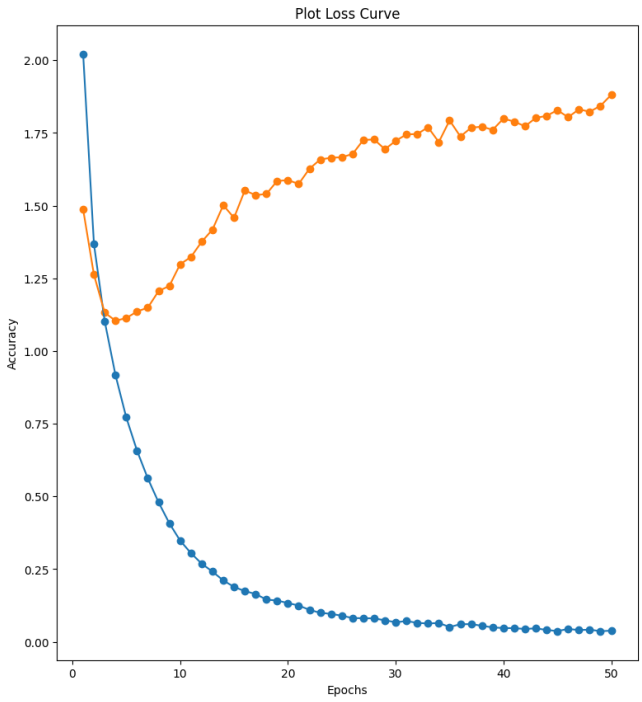
```
Epoch 23/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0995 - accuracy: 0.9674 -
val_loss: 1.6587 - val_accuracy: 0.6806
Epoch 24/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0961 - accuracy: 0.9683 -
val_loss: 1.6647 - val_accuracy: 0.6778
Epoch 25/50
625/625 [==============================] - 65s 105ms/step - loss: 0.0896 - accuracy: 0.9709 -
val_loss: 1.6665 - val_accuracy: 0.6806
Epoch 26/50
625/625 [==============================] - 66s 106ms/step - loss: 0.0817 - accuracy: 0.9732 -
val_loss: 1.6776 - val_accuracy: 0.6787
Epoch 27/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0809 - accuracy: 0.9735 -
val_loss: 1.7261 - val_accuracy: 0.6804
Epoch 28/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0806 - accuracy: 0.9734 -
val_loss: 1.7274 - val_accuracy: 0.6763
Epoch 29/50
625/625 [==============================] - 66s 105ms/step - loss: 0.0738 - accuracy: 0.9757 -
val_loss: 1.6939 - val_accuracy: 0.6834
Epoch 30/50
625/625 [==============================] - 66s 105ms/step - loss: 0.0673 - accuracy: 0.9786 -
val_loss: 1.7229 - val_accuracy: 0.6816
Epoch 31/50
625/625 [==============================] - 65s 103ms/step - loss: 0.0711 - accuracy: 0.9768 -
val_loss: 1.7451 - val_accuracy: 0.6799
Epoch 32/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0653 - accuracy: 0.9783 -
val_loss: 1.7462 - val_accuracy: 0.6825
Epoch 33/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0627 - accuracy: 0.9799 -
val_loss: 1.7696 - val_accuracy: 0.6855
Epoch 34/50
625/625 [==============================] - 64s 103ms/step - loss: 0.0649 - accuracy: 0.9793 -
val_loss: 1.7186 - val_accuracy: 0.6832
Epoch 35/50
625/625 [==============================] - 62s 100ms/step - loss: 0.0507 - accuracy: 0.9830 -
val_loss: 1.7931 - val_accuracy: 0.6851
Epoch 36/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0605 - accuracy: 0.9806 -
val_loss: 1.7379 - val_accuracy: 0.6833
Epoch 37/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0608 - accuracy: 0.9798 -
val_loss: 1.7680 - val_accuracy: 0.6833
Epoch 38/50
625/625 [==============================] - 62s 100ms/step - loss: 0.0550 - accuracy: 0.9824 -
val_loss: 1.7716 - val_accuracy: 0.6834
Epoch 39/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0495 - accuracy: 0.9837 -
val_loss: 1.7607 - val_accuracy: 0.6848
Epoch 40/50
625/625 [==============================] - 63s 100ms/step - loss: 0.0474 - accuracy: 0.9848 -
val_loss: 1.7995 - val_accuracy: 0.6842
Epoch 41/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0468 - accuracy: 0.9846 -
val_loss: 1.7888 - val_accuracy: 0.6825
Epoch 42/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0441 - accuracy: 0.9855 -
val_loss: 1.7729 - val_accuracy: 0.6910
Epoch 43/50
625/625 [==============================] - 62s 100ms/step - loss: 0.0462 - accuracy: 0.9849 -
val_loss: 1.8018 - val_accuracy: 0.6861
Epoch 44/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0408 - accuracy: 0.9874 -
val_loss: 1.8087 - val_accuracy: 0.6855
```

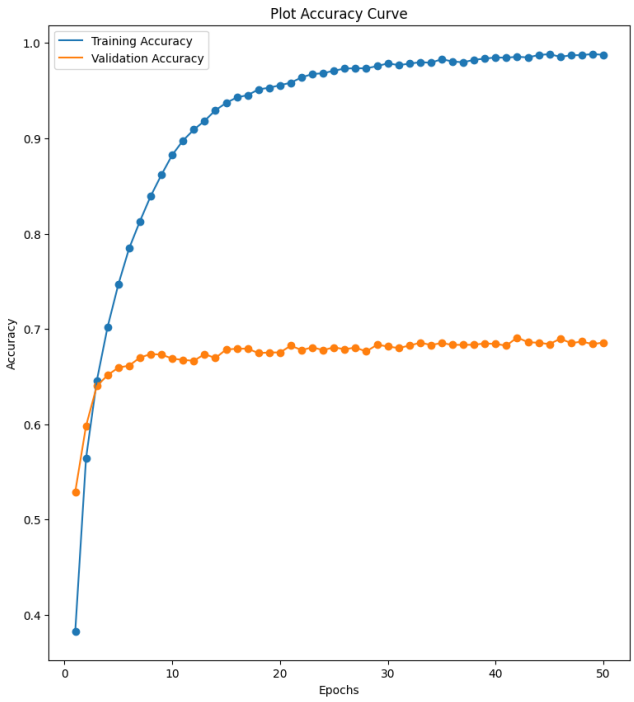
```
Epoch 45/50
625/625 [==============================] - 63s 101ms/step - loss: 0.0367 - accuracy: 0.9883 -
val_loss: 1.8283 - val_accuracy: 0.6840
Epoch 46/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0442 - accuracy: 0.9855 -
val_loss: 1.8041 - val_accuracy: 0.6898
Epoch 47/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0404 - accuracy: 0.9871 -
val_loss: 1.8313 - val_accuracy: 0.6851
Epoch 48/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0417 - accuracy: 0.9873 -
val_loss: 1.8230 - val_accuracy: 0.6867
Epoch 49/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0364 - accuracy: 0.9884 -
val_loss: 1.8433 - val_accuracy: 0.6841
Epoch 50/50
625/625 [==============================] - 62s 100ms/step - loss: 0.0383 - accuracy: 0.9876 -
val_loss: 1.8812 - val_accuracy: 0.6857
```

In [ ]:
```
print(storeResult(efficientNetModelHistory))
plot_loss_curve(efficientNetModelHistory)
plt.show()
```

```
{'Model Name': 'efficientNetV2', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 0.044109541922
807693, 'Val Loss': 1.7729359865188599, 'Train Acc': 0.9855499863624573, 'Val Acc': 0.6909999
847412109, '[Train - Val] Acc': 0.29455000162124634}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```





Observation

Comparing the EfficientNetV2 Model with the baseline model, we can see that the the model plateau and the model is able to generalise well. The accuracy of both training and validation improved which suggest that efficientNet is a good model.

EfficientNetV2B0 model with Data Augmentation

In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
```

```
x = tf.keras.applications.efficientnet_v2.EfficientNetV2B0(
    include_top=False,
    weights='imagenet',
    input_shape=IMG_SIZE,
    pooling="max",
    include_preprocessing=False
)(x)
x = Flatten()(x)
x = Dense(NUM_CLASS, 'softmax')(x)
efficientNetAugModel = Model(
    inputs=inputs, outputs=x, name="efficientNetV2Aug")
efficientNetAugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                             loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:

```
efficientNetAugModelHistory = efficientNetAugModel.fit(x_train_aug, y_train, epochs=50,
                                                       validation_data=(x_val, y_val), batch_size=BAT
```

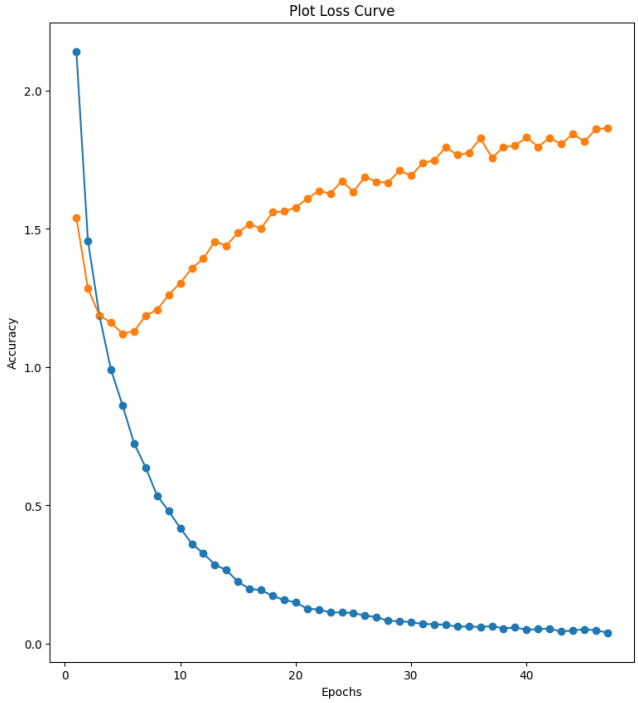
```
Epoch 1/50
625/625 [==============================] - 75s 103ms/step - loss: 2.1418 - accuracy: 0.3431 -
val_loss: 1.5413 - val_accuracy: 0.5078
Epoch 2/50
625/625 [==============================] - 63s 100ms/step - loss: 1.4578 - accuracy: 0.5383 -
val_loss: 1.2844 - val_accuracy: 0.5926
Epoch 3/50
625/625 [==============================] - 63s 101ms/step - loss: 1.1859 - accuracy: 0.6184 -
val_loss: 1.1871 - val_accuracy: 0.6241
Epoch 4/50
625/625 [==============================] - 63s 100ms/step - loss: 0.9914 - accuracy: 0.6780 -
val_loss: 1.1611 - val_accuracy: 0.6410
Epoch 5/50
625/625 [==============================] - 63s 101ms/step - loss: 0.8621 - accuracy: 0.7175 -
val_loss: 1.1214 - val_accuracy: 0.6609
Epoch 6/50
625/625 [==============================] - 62s 100ms/step - loss: 0.7247 - accuracy: 0.7599 -
val_loss: 1.1301 - val_accuracy: 0.6646
Epoch 7/50
625/625 [==============================] - 62s 99ms/step - loss: 0.6379 - accuracy: 0.7892 -
val_loss: 1.1858 - val_accuracy: 0.6607
Epoch 8/50
625/625 [==============================] - 62s 100ms/step - loss: 0.5351 - accuracy: 0.8212 -
val_loss: 1.2081 - val_accuracy: 0.6676
Epoch 9/50
625/625 [==============================] - 62s 99ms/step - loss: 0.4796 - accuracy: 0.8418 -
val_loss: 1.2620 - val_accuracy: 0.6665
Epoch 10/50
625/625 [==============================] - 63s 100ms/step - loss: 0.4182 - accuracy: 0.8587 -
val_loss: 1.3042 - val_accuracy: 0.6655
Epoch 11/50
625/625 [==============================] - 62s 98ms/step - loss: 0.3613 - accuracy: 0.8811 -
val_loss: 1.3578 - val_accuracy: 0.6639
Epoch 12/50
625/625 [==============================] - 63s 100ms/step - loss: 0.3268 - accuracy: 0.8905 -
val_loss: 1.3923 - val_accuracy: 0.6659
Epoch 13/50
625/625 [==============================] - 62s 100ms/step - loss: 0.2861 - accuracy: 0.9036 -
val_loss: 1.4547 - val_accuracy: 0.6659
Epoch 14/50
625/625 [==============================] - 62s 100ms/step - loss: 0.2660 - accuracy: 0.9108 -
val_loss: 1.4394 - val_accuracy: 0.6725
Epoch 15/50
625/625 [==============================] - 63s 101ms/step - loss: 0.2238 - accuracy: 0.9260 -
val_loss: 1.4868 - val_accuracy: 0.6716
Epoch 16/50
625/625 [==============================] - 62s 99ms/step - loss: 0.1985 - accuracy: 0.9336 -
val_loss: 1.5174 - val_accuracy: 0.6720
Epoch 17/50
625/625 [==============================] - 62s 99ms/step - loss: 0.1939 - accuracy: 0.9352 -
val_loss: 1.5018 - val_accuracy: 0.6741
Epoch 18/50
625/625 [==============================] - 62s 100ms/step - loss: 0.1732 - accuracy: 0.9431 -
val_loss: 1.5600 - val_accuracy: 0.6756
Epoch 19/50
625/625 [==============================] - 62s 99ms/step - loss: 0.1582 - accuracy: 0.9485 -
val_loss: 1.5632 - val_accuracy: 0.6781
Epoch 20/50
625/625 [==============================] - 63s 101ms/step - loss: 0.1495 - accuracy: 0.9506 -
val_loss: 1.5779 - val_accuracy: 0.6752
Epoch 21/50
625/625 [==============================] - 62s 99ms/step - loss: 0.1268 - accuracy: 0.9585 -
val_loss: 1.6092 - val_accuracy: 0.6686
Epoch 22/50
625/625 [==============================] - 63s 101ms/step - loss: 0.1237 - accuracy: 0.9597 -
val_loss: 1.6378 - val_accuracy: 0.6812
```

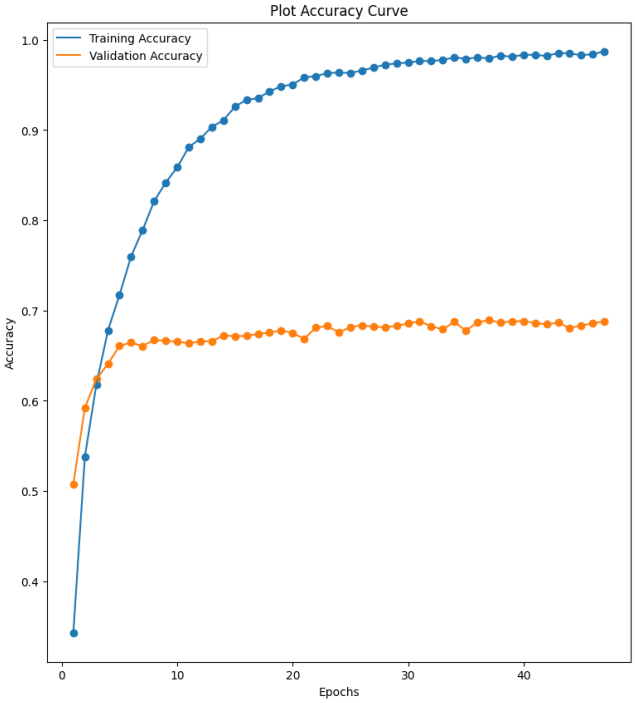
```
Epoch 23/50
625/625 [==============================] - 62s 99ms/step - loss: 0.1131 - accuracy: 0.9632 -
val_loss: 1.6279 - val_accuracy: 0.6829
Epoch 24/50
625/625 [==============================] - 62s 100ms/step - loss: 0.1131 - accuracy: 0.9637 -
val_loss: 1.6727 - val_accuracy: 0.6758
Epoch 25/50
625/625 [==============================] - 63s 101ms/step - loss: 0.1109 - accuracy: 0.9633 -
val_loss: 1.6348 - val_accuracy: 0.6818
Epoch 26/50
625/625 [==============================] - 62s 99ms/step - loss: 0.1018 - accuracy: 0.9661 -
val_loss: 1.6885 - val_accuracy: 0.6836
Epoch 27/50
625/625 [==============================] - 62s 100ms/step - loss: 0.0966 - accuracy: 0.9692 -
val_loss: 1.6699 - val_accuracy: 0.6822
Epoch 28/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0830 - accuracy: 0.9725 -
val_loss: 1.6678 - val_accuracy: 0.6815
Epoch 29/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0817 - accuracy: 0.9739 -
val_loss: 1.7116 - val_accuracy: 0.6832
Epoch 30/50
625/625 [==============================] - 64s 103ms/step - loss: 0.0778 - accuracy: 0.9748 -
val_loss: 1.6923 - val_accuracy: 0.6857
Epoch 31/50
625/625 [==============================] - 62s 100ms/step - loss: 0.0716 - accuracy: 0.9765 -
val_loss: 1.7383 - val_accuracy: 0.6881
Epoch 32/50
625/625 [==============================] - 62s 100ms/step - loss: 0.0704 - accuracy: 0.9766 -
val_loss: 1.7474 - val_accuracy: 0.6826
Epoch 33/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0687 - accuracy: 0.9777 -
val_loss: 1.7958 - val_accuracy: 0.6792
Epoch 34/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0616 - accuracy: 0.9805 -
val_loss: 1.7672 - val_accuracy: 0.6875
Epoch 35/50
625/625 [==============================] - 63s 101ms/step - loss: 0.0628 - accuracy: 0.9790 -
val_loss: 1.7753 - val_accuracy: 0.6777
Epoch 36/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0601 - accuracy: 0.9805 -
val_loss: 1.8280 - val_accuracy: 0.6869
Epoch 37/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0638 - accuracy: 0.9794 -
val_loss: 1.7568 - val_accuracy: 0.6895
Epoch 38/50
625/625 [==============================] - 61s 98ms/step - loss: 0.0554 - accuracy: 0.9826 -
val_loss: 1.7958 - val_accuracy: 0.6866
Epoch 39/50
625/625 [==============================] - 62s 99ms/step - loss: 0.0586 - accuracy: 0.9812 -
val_loss: 1.8007 - val_accuracy: 0.6882
Epoch 40/50
625/625 [==============================] - 63s 100ms/step - loss: 0.0507 - accuracy: 0.9836 -
val_loss: 1.8316 - val_accuracy: 0.6881
Epoch 41/50
625/625 [==============================] - 64s 102ms/step - loss: 0.0527 - accuracy: 0.9833 -
val_loss: 1.7967 - val_accuracy: 0.6864
Epoch 42/50
625/625 [==============================] - 66s 105ms/step - loss: 0.0550 - accuracy: 0.9822 -
val_loss: 1.8299 - val_accuracy: 0.6847
Epoch 43/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0438 - accuracy: 0.9854 -
val_loss: 1.8065 - val_accuracy: 0.6867
Epoch 44/50
625/625 [==============================] - 66s 106ms/step - loss: 0.0481 - accuracy: 0.9851 -
val_loss: 1.8432 - val_accuracy: 0.6805
```

```
Epoch 45/50
625/625 [==============================] - 65s 104ms/step - loss: 0.0525 - accuracy: 0.9828 -
val_loss: 1.8162 - val_accuracy: 0.6836
Epoch 46/50
625/625 [==============================] - 66s 106ms/step - loss: 0.0486 - accuracy: 0.9842 -
val_loss: 1.8604 - val_accuracy: 0.6857
Epoch 47/50
625/625 [==============================] - 66s 105ms/step - loss: 0.0399 - accuracy: 0.9872 -
val_loss: 1.8648 - val_accuracy: 0.6882
```

In [ ]:
```
print(storeResult(efficientNetAugModelHistory))
plot_loss_curve(efficientNetAugModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_20392\3161967465.py:14: FutureWarning: The
frame.append method is deprecated and will be removed from pandas in a future version. Use pa
ndas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
{'Model Name': 'efficientNetV2Aug', 'Epochs': 47, 'Batch Size': 64, 'Train Loss': 0.063754409
55162048, 'Val Loss': 1.7568185329437256, 'Train Acc': 0.9794250130653381, 'Val Acc': 0.68949
99742507935, '[Train - Val] Acc': 0.2899250388145447}
```





Model Selection

After running the different types of model, we need to decide on one of the model to be hyper tuned to be our final model

In [ ]:
```
allResults.sort_values(by=["Val Acc", "Train Acc"], ascending=False).style.apply(
    lambda x: [
        "background-color: red; color: white" if v else "" for v in x == x.min()]
).apply(
    lambda x: [
        "background-color: green; color: white" if v else "" for v in x == x.max()
    ]
)
```

Out[ ]:

	Model Name	Epochs	Batch Size	Train Loss	Val Loss	Train Acc	Val Acc	[Train - Val] Acc
12	CustomVGG16Aug	50	64	1.811427	2.227701	0.793000	0.695600	0.097400
18	efficientNetV2	50	64	0.044110	1.772936	0.985550	0.691000	0.294550
19	efficientNetV2Aug	47	64	0.063754	1.756819	0.979425	0.689500	0.289925
11	CustomVGG16	50	64	1.808951	2.284049	0.801225	0.685000	0.116225
9	CustomVGGAug	50	64	0.026702	2.573303	0.991125	0.625200	0.365925
8	CustomVGG_L2	27	64	1.175810	2.197414	0.845200	0.617700	0.227500
6	CustomVGG	26	64	0.293430	1.722505	0.902200	0.616600	0.285600
7	CustomVGG_L1	50	64	2.226223	2.301511	0.553200	0.527100	0.026100
14	CustomResNet	29	64	1.768260	2.569412	0.703775	0.521900	0.181875
15	CustomResNetAug	25	64	1.867212	2.395577	0.636500	0.513800	0.122700
17	CustomResNetDropV2	36	64	2.312564	2.245985	0.468950	0.478100	-0.009150
3	conv2D	17	64	1.381505	1.915656	0.562250	0.447700	0.114550
4	conv2DAug	16	64	1.590236	1.887040	0.504375	0.436900	0.067475
0	baseline	18	64	1.864174	2.212881	0.415300	0.338500	0.076800
1	baselineAug	18	64	1.931878	2.246060	0.398800	0.330300	0.068500

It seems like the CustomVGG16 Augmented model performed the best followed by the pretrained efficientNetV2 model. Therefore we will be making model improvements to the customVGG16 model.

Model Improvement - customVGG16

We will doing the following to tune the VGG models.

- Using the Cosine Annealing Learning Rate Scheduler
- Use Keras Tuner to do a search to fine

In [ ]:
```
steps_per_epoch = np.ceil(len(x_train) / BATCH_SIZE)
```

In [ ]:
```
def tune_vgg16_model(hp):
    weight_decay = hp.Float("weight_decay", min_value=3e-4,
                            max_value=1e-2, sampling="log")
    learning_rate = hp.Float(
        "learning_rate", min_value=1e-3, max_value=1e-1, sampling="log")
    scheduler = tf.keras.optimizers.schedules.CosineDecay(
        learning_rate, 50 * steps_per_epoch)
    optimizer = SGD(learning_rate=scheduler, momentum=0.9)
    inputs = Input(IMG_SIZE)
    x = pre_processing_v1(inputs)
    x = vgg_block_16(2, 64, dropout=[0.3])(x)
    x = vgg_block_16(2, 128, dropout=[0.4])(x)
    x = vgg_block_16(3, 256, dropout=[0.4, 0.4])(x)
    x = vgg_block_16(3, 512, dropout=[0.4, 0.4])(x)
    x = vgg_block_16(3, 512, dropout=[0.4, 0.4])(x)
    x = Dropout(0.5)(x)
    x = GlobalAveragePooling2D()(x)
    x = Dense(512, 'relu', kernel_regularizer=l2(weight_decay))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.5)(x)
    x = Dense(NUM_CLASS, 'softmax')(x)
```

```
    model = Model(inputs=inputs, outputs=x, name="tuneVGG16")
    model.compile(optimizer=optimizer,
                  loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

In [ ]:
```
VGG16Tuner = kt.RandomSearch(tune_vgg16_model, objective="val_accuracy", overwrite=True, proj
```

In [ ]:
```
VGG16Tuner.search(
    x_train_aug, y_train, validation_data=(x_val, y_val), epochs=60, batch_size=BATCH_SIZE, c
        EarlyStopping(monitor="val_accuracy", patience=10,
                      restore_best_weights=True)
    ]
)
VGG16Tuner.results_summary(num_trials=3)
```

```
Trial 3 Complete [01h 54m 28s]
val_accuracy: 0.7368000149726868

Best val_accuracy So Far: 0.7368000149726868
Total elapsed time: 03h 16m 42s
INFO:tensorflow:Oracle triggered exit
Results summary
Results in .\cifar20_vgg16
Showing 3 best trials
<keras_tuner.engine.objective.Objective object at 0x000002A88992AD90>
Trial summary
Hyperparameters:
weight_decay: 0.0028624153363848836
learning_rate: 0.050483592250525275
Score: 0.7368000149726868
Trial summary
Hyperparameters:
weight_decay: 0.0010914879067787544
learning_rate: 0.014859884666119429
Score: 0.7161999940872192
Trial summary
Hyperparameters:
weight_decay: 0.008132173395641411
learning_rate: 0.010873828384308268
Score: 0.7138000130653381
```

Tuned Model Selection

In [ ]:
```
vgg16_model = VGG16Tuner.get_best_models()[0]
```

Model Evaluation

Now it is time to evaluate my final model. To ensure it generalise well, We want to ensure the accuracy on the testing set consistent with that on the validation set.

Saving model

In [ ]:
```
vgg16_model.save('models/Cifar10CustomVGG16 - Final')
efficientNetModel.save('models/Cifar10EfficientNetV2')
```

```
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_con
volution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convol
ution_op while saving (showing 5 of 13). These functions will not be directly callable after
loading.
```

```
INFO:tensorflow:Assets written to: models/Cifar10CustomVGG16 - Final\assets
```


```
INFO:tensorflow:Assets written to: models/Cifar10CustomVGG16 - Final\assets
```


In [ ]:
```
vgg16_model.save('models/Cifar10CustomVGG16 - Final.h5')
efficientNetModel.save('models/Cifar10EfficientNetV2.h5')
```

Initiate model after tuning and saving

In [ ]:
```
final_model = tf.keras.models.load_model('models/Cifar10CustomVGG16 - Final')

final_model.summary()
```

```
Model: "tuneVGG16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 normalization (Normalizatio  (None, 32, 32, 3)        7
 n)

 sequential (Sequential)     (None, 16, 16, 64)        39232

 sequential_1 (Sequential)   (None, 8, 8, 128)         222464

 sequential_2 (Sequential)   (None, 4, 4, 256)         1478400

 sequential_3 (Sequential)   (None, 2, 2, 512)         5905920

 sequential_4 (Sequential)   (None, 1, 1, 512)         7085568

 dropout_13 (Dropout)        (None, 1, 1, 512)         0

 global_average_pooling2d (G  (None, 512)              0
 lobalAveragePooling2D)

 dense (Dense)               (None, 512)               262656

 batch_normalization_13 (Bat  (None, 512)              2048
 chNormalization)

 dropout_14 (Dropout)        (None, 512)               0

 dense_1 (Dense)             (None, 20)                10260

=================================================================
Total params: 15,006,555
Trainable params: 14,997,076
Non-trainable params: 9,479
_________________________________________________________________
```

Testing Set

After training our model, we need to use the test set to test the model accuracy of the model for unseen data.

In [ ]:
```
final_model.evaluate(x_test, y_test)
```

Out[ ]:
```
313/313 [==============================] - 17s 19ms/step - loss: 1.8333 - accuracy: 0.7337
[1.8333473205566406, 0.7336999773979187]
```

In [ ]:
```
y_pred = final_model.predict(x_test)
```

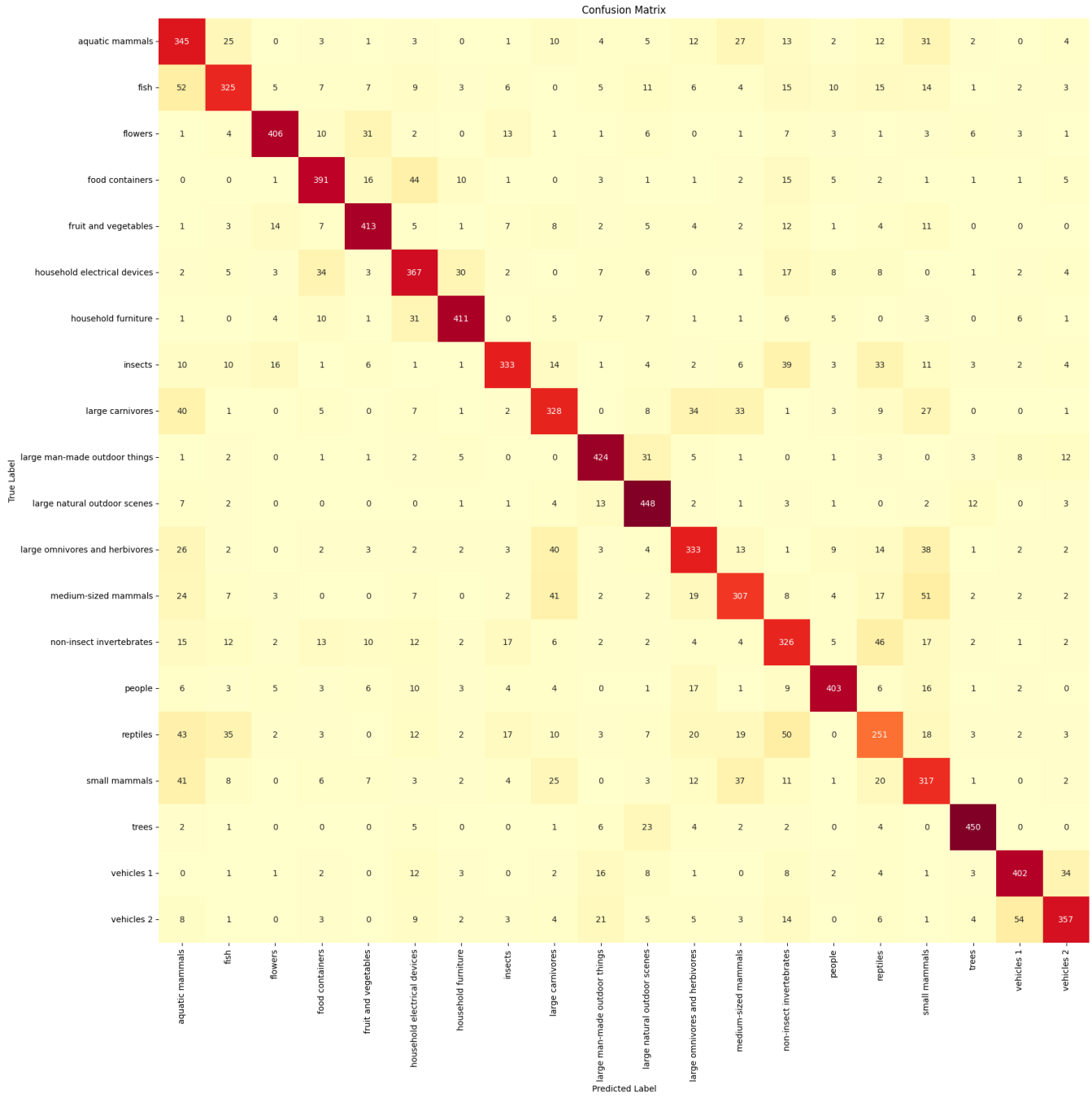
```
313/313 [==============================] - 14s 44ms/step
```

In [ ]:
```
report = classification_report(
    np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1), target_names=class_labels.values()
)
print(report)
```

```
                                precision    recall  f1-score   support

                aquatic mammals       0.55      0.69      0.61       500
                           fish       0.73      0.65      0.69       500
                        flowers       0.88      0.81      0.84       500
                food containers       0.78      0.78      0.78       500
          fruit and vegetables       0.82      0.83      0.82       500
  household electrical devices       0.68      0.73      0.70       500
           household furniture       0.86      0.82      0.84       500
                       insects       0.80      0.67      0.73       500
               large carnivores       0.65      0.66      0.65       500
 large man-made outdoor things       0.82      0.85      0.83       500
  large natural outdoor scenes       0.76      0.90      0.82       500
large omnivores and herbivores       0.69      0.67      0.68       500
          medium-sized mammals       0.66      0.61      0.64       500
      non-insect invertebrates       0.59      0.65      0.62       500
                        people       0.86      0.81      0.83       500
                      reptiles       0.55      0.50      0.53       500
                small mammals       0.56      0.63      0.60       500
                         trees       0.91      0.90      0.90       500
                    vehicles 1       0.82      0.80      0.81       500
                    vehicles 2       0.81      0.71      0.76       500

                      accuracy                           0.73     10000
                     macro avg       0.74      0.73      0.73     10000
                  weighted avg       0.74      0.73      0.73     10000
```

We can see from the classification report that the model is very good at identifying and differentiating trees, flowers and people but not as good as predicting aquatic mammals, reptiles, small mammals and non-insect invertebrates. Let's do a mini error analysis and find out why.

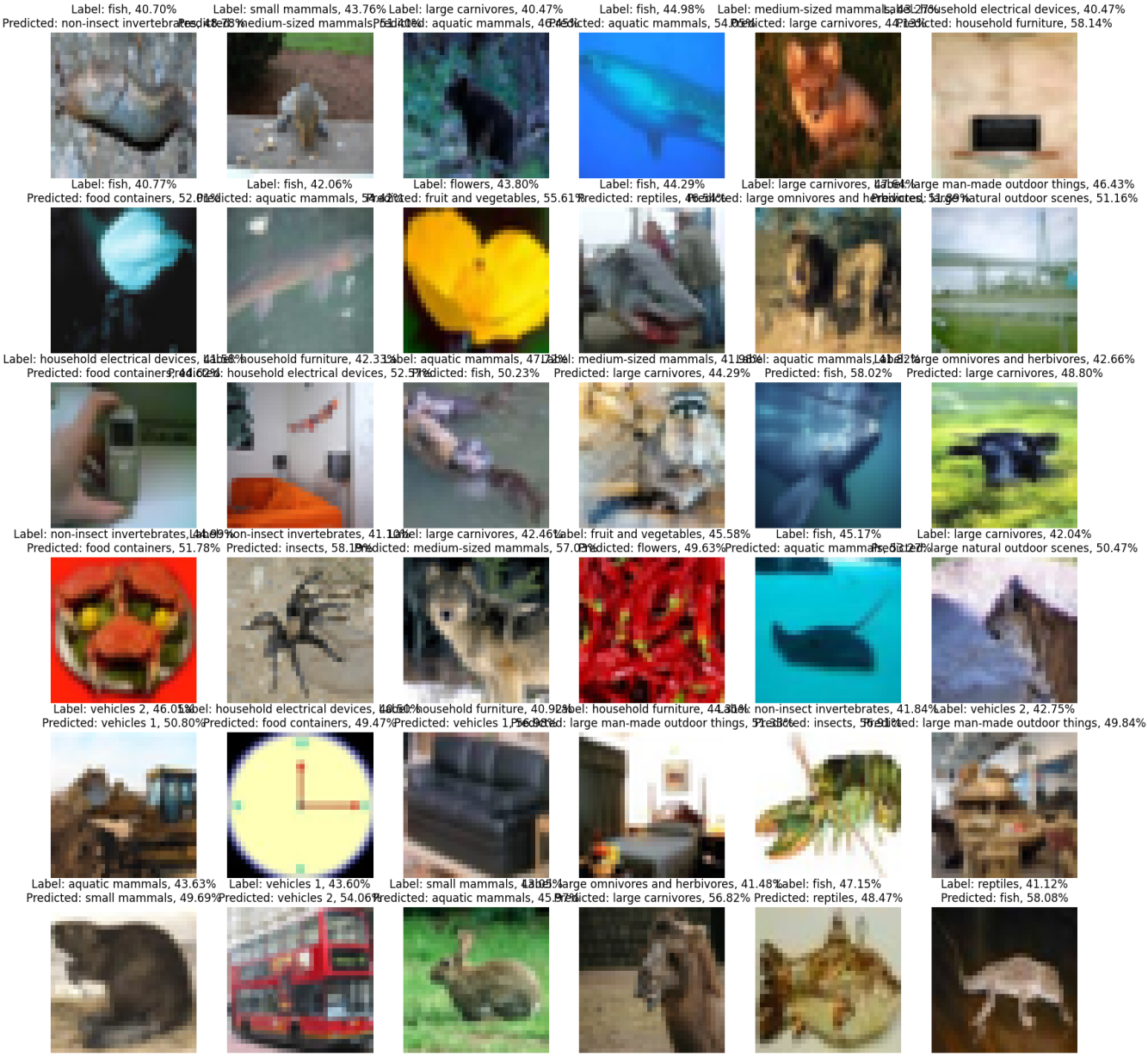
In [ ]:
```
plt.figure(1, figsize=(20, 20))
plt.title("Confusion Matrix")
sns.heatmap(tf.math.confusion_matrix(
    np.argmax(y_test, axis=1),
    np.argmax(y_pred, axis=1),
    num_classes=NUM_CLASS,
    dtype=tf.dtypes.int32,
    name=None
), annot=True, fmt="", cbar=False, cmap="YlOrRd", yticklabels=class_labels.values(), xticklab
plt.ylabel("True Label")
plt.xlabel("Predicted Label")
plt.show()
```



Error Analysis

In [ ]:
```
wrong = (np.argmax(y_test, axis=1) != np.argmax(y_pred, axis=1))
x_test_wrong = x_test[wrong]
y_test_wrong = np.argmax(y_test[wrong], axis=1)
y_pred_wrong = y_pred[wrong]
```

In [ ]:
```
fig, ax = plt.subplots(6, 6, figsize=(20, 20))
existArr = []
for subplot in ax.ravel():
    idx = np.random.choice(x_test_wrong.shape[0], 1, replace=False)
    while (y_pred_wrong[idx][0][int(y_test_wrong[idx])] * 100) <= 40 or idx in existArr:
        idx = np.random.choice(x_test_wrong.shape[0], 1, replace=False)
    pred = class_labels[np.argmax(y_pred_wrong[idx])]
    subplot.axis("off")
    actual = class_labels[int(y_test_wrong[idx])]
    subplot.imshow(x_test_wrong[idx].reshape(32, 32, 3))
    subplot.set_title(f"""Label: {actual}, {(y_pred_wrong[idx][0][int(y_test_wrong[idx])] * 1
Predicted: {pred}, {(np.max(y_pred_wrong[idx]) * 100):.2f}%""")
    existArr.append(idx)
```



Observations

When we look at the examples that the model made a wrong prediction, we can identify some reasons why it is the case.

1. Low pixel resolutions makes images hard to be distinguished.
    - Example: Row 2 Column 2
      That images looks a bunch of pixels [Due to low resolutions] and it is hard to distinguish the features.
2. Close Up
    - Example: Row 3 Column 4
      The model predicted the values of a medium sized mammals and large carnivores wrongly. This is likely due to image very close and making it seem like the animal is a carnivores based on the teeth in the image.

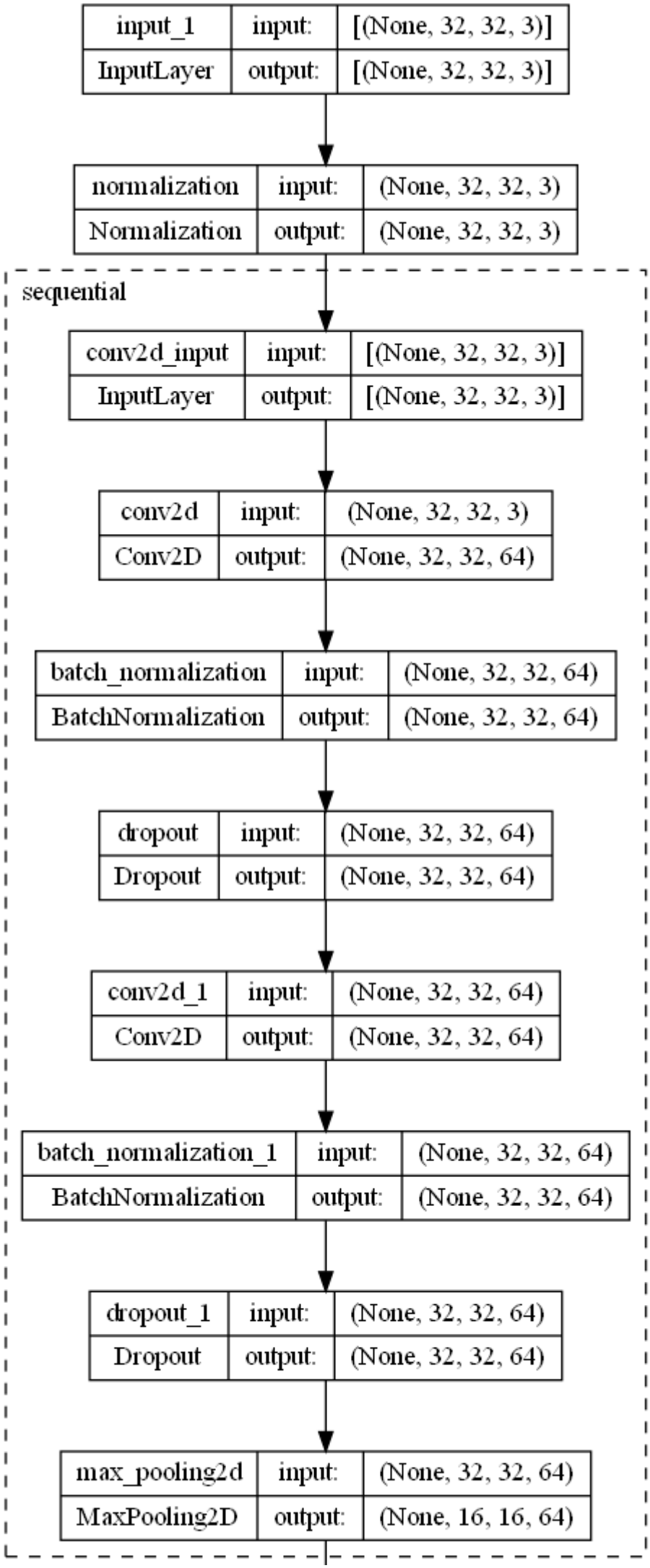
Therefore, it appears that the model's mistakes are reasonable.

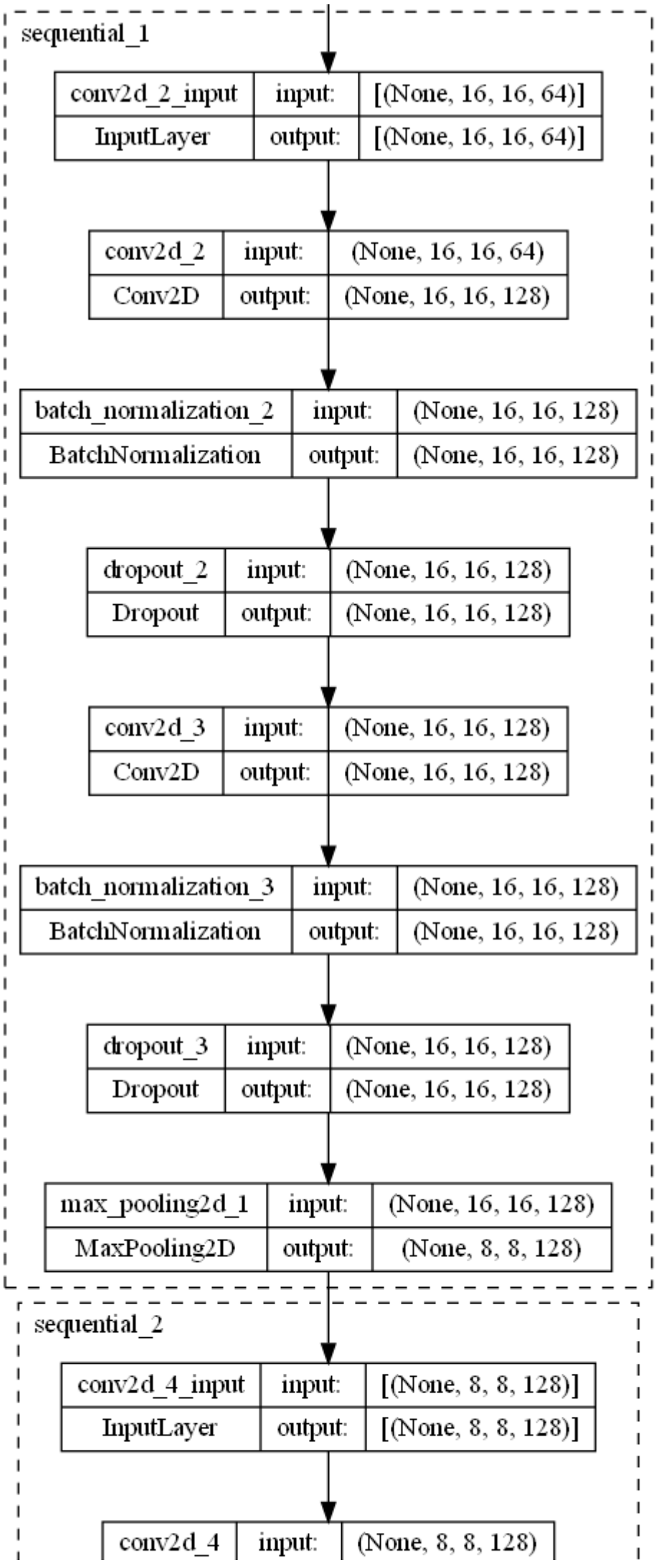
Model Visualisation

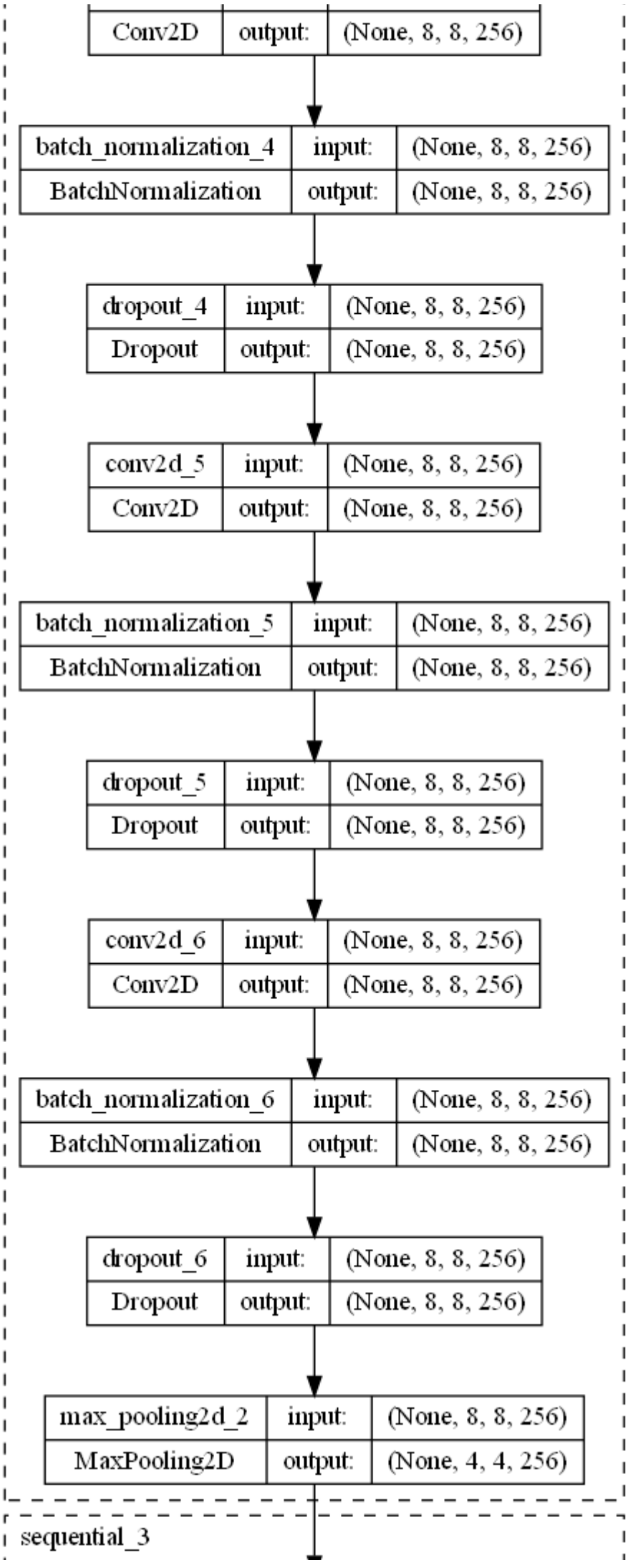
In [ ]:
```
tf.keras.utils.plot_model(final_model, show_shapes=True,
```

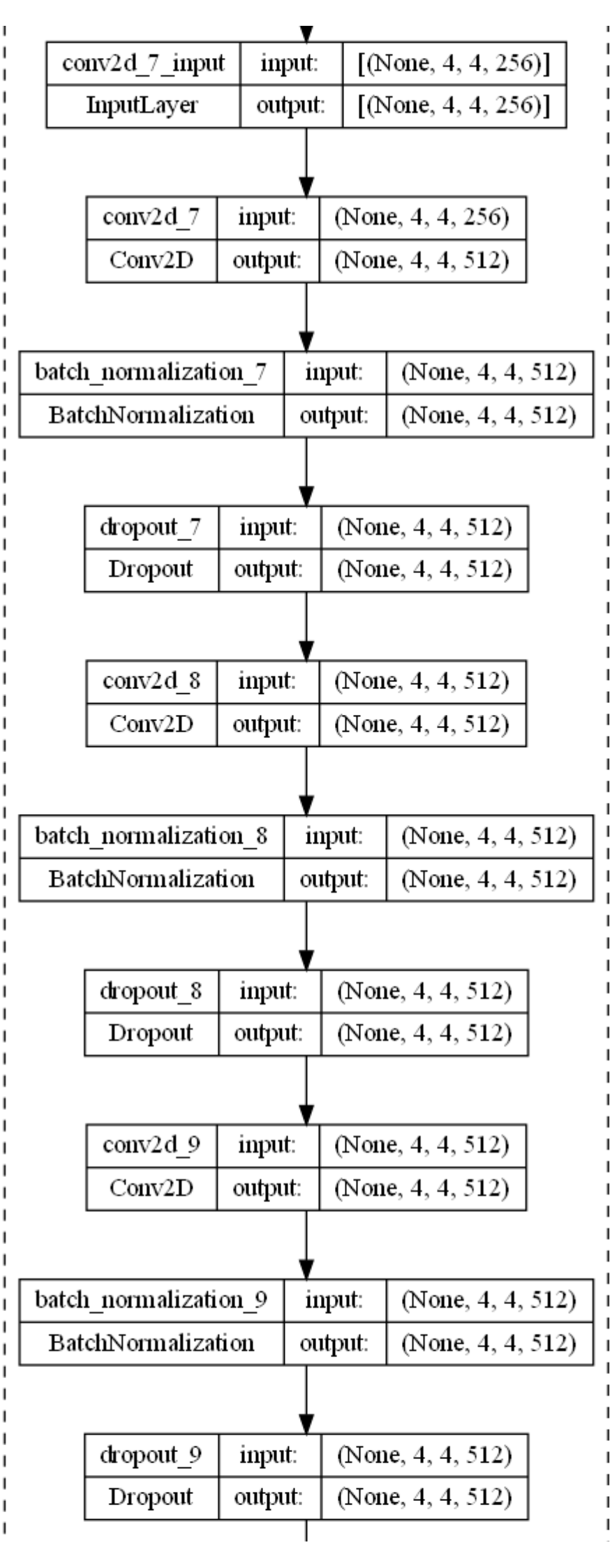
```
                                expand_nested=True, show_layer_activations=True)
```

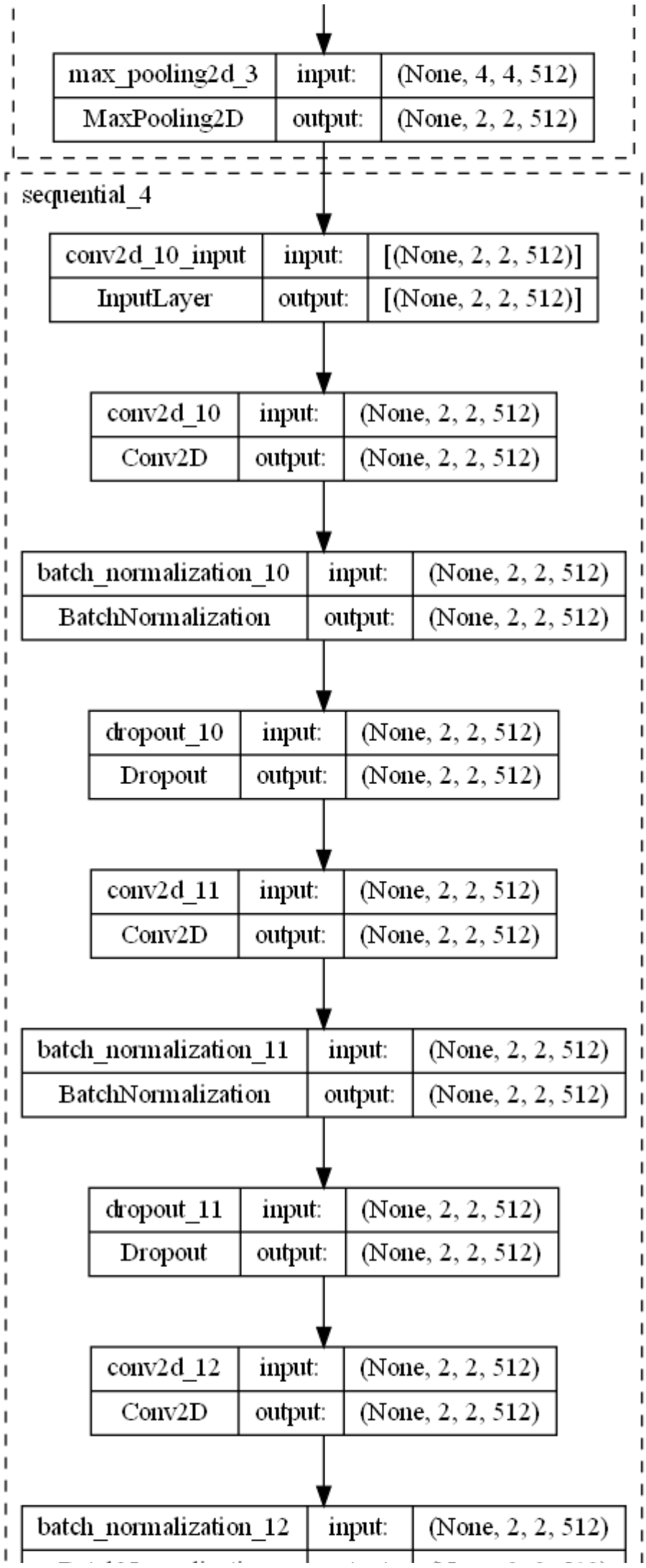
Out[ ]:

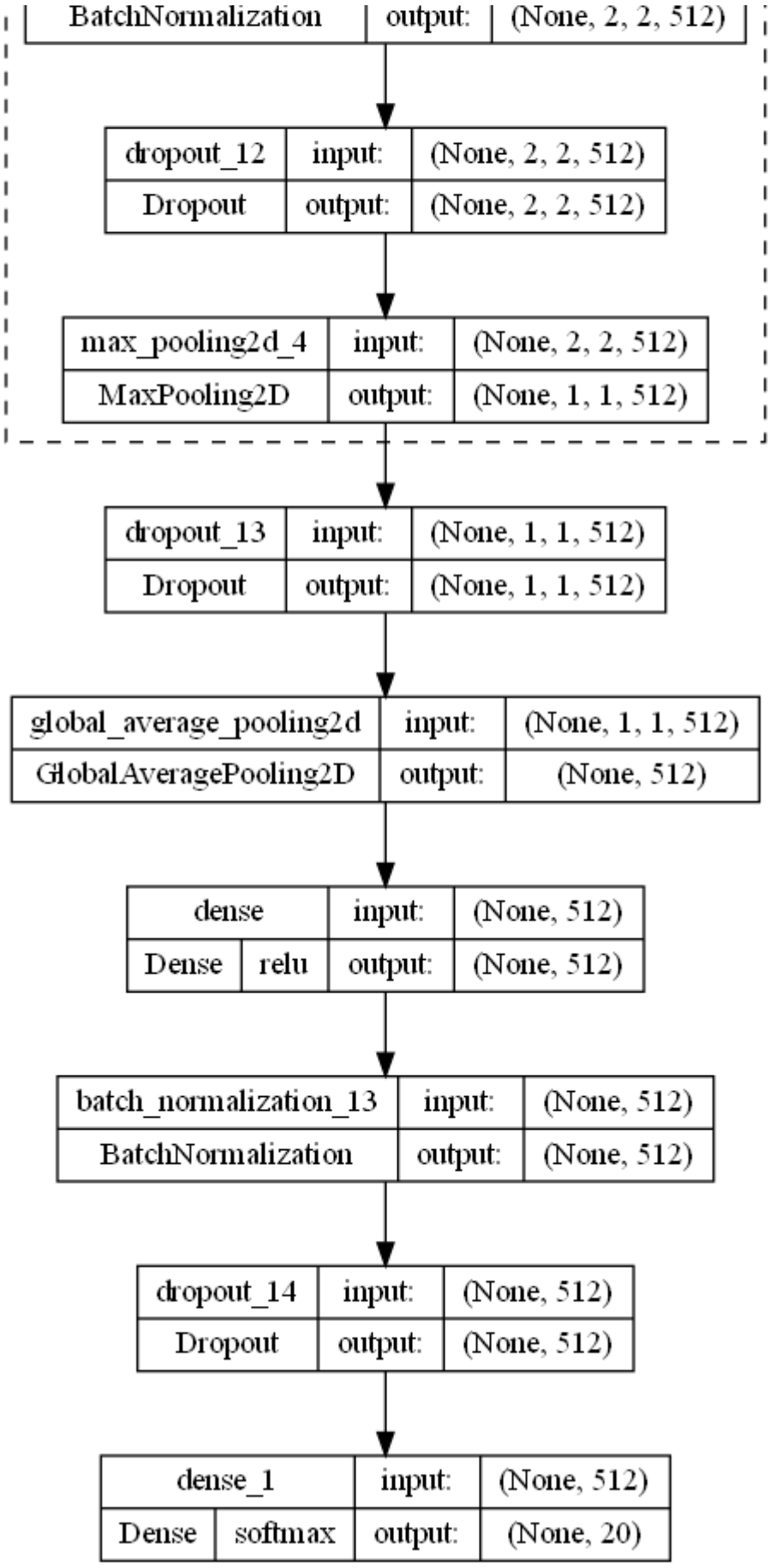

input_1 | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
normalization | input: | (None, 32, 32, 3)
Normalization | output: | (None, 32, 32, 3)
sequential
conv2d_input | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
conv2d | input: | (None, 32, 32, 3)
Conv2D | output: | (None, 32, 32, 64)
batch_normalization | input: | (None, 32, 32, 64)
BatchNormalization | output: | (None, 32, 32, 64)
dropout | input: | (None, 32, 32, 64)
Dropout | output: | (None, 32, 32, 64)
conv2d_1 | input: | (None, 32, 32, 64)
Conv2D | output: | (None, 32, 32, 64)
batch_normalization_1 | input: | (None, 32, 32, 64)
BatchNormalization | output: | (None, 32, 32, 64)
dropout_1 | input: | (None, 32, 32, 64)
Dropout | output: | (None, 32, 32, 64)
max_pooling2d | input: | (None, 32, 32, 64)
MaxPooling2D | output: | (None, 16, 16, 64)


sequential_1
conv2d_2_input | input: | [(None, 16, 16, 64)]
InputLayer | output: | [(None, 16, 16, 64)]
conv2d_2 | input: | (None, 16, 16, 64)
Conv2D | output: | (None, 16, 16, 128)
batch_normalization_2 | input: | (None, 16, 16, 128)
BatchNormalization | output: | (None, 16, 16, 128)
dropout_2 | input: | (None, 16, 16, 128)
Dropout | output: | (None, 16, 16, 128)
conv2d_3 | input: | (None, 16, 16, 128)
Conv2D | output: | (None, 16, 16, 128)
batch_normalization_3 | input: | (None, 16, 16, 128)
BatchNormalization | output: | (None, 16, 16, 128)
dropout_3 | input: | (None, 16, 16, 128)
Dropout | output: | (None, 16, 16, 128)
max_pooling2d_1 | input: | (None, 16, 16, 128)
MaxPooling2D | output: | (None, 8, 8, 128)
sequential_2
conv2d_4_input | input: | [(None, 8, 8, 128)]
InputLayer | output: | [(None, 8, 8, 128)]
conv2d_4 | input: | (None, 8, 8, 128)


Conv2D output: (None, 8, 8, 256)
batch_normalization_4 input: (None, 8, 8, 256)
BatchNormalization output: (None, 8, 8, 256)
dropout_4 input: (None, 8, 8, 256)
Dropout output: (None, 8, 8, 256)
conv2d_5 input: (None, 8, 8, 256)
Conv2D output: (None, 8, 8, 256)
batch_normalization_5 input: (None, 8, 8, 256)
BatchNormalization output: (None, 8, 8, 256)
dropout_5 input: (None, 8, 8, 256)
Dropout output: (None, 8, 8, 256)
conv2d_6 input: (None, 8, 8, 256)
Conv2D output: (None, 8, 8, 256)
batch_normalization_6 input: (None, 8, 8, 256)
BatchNormalization output: (None, 8, 8, 256)
dropout_6 input: (None, 8, 8, 256)
Dropout output: (None, 8, 8, 256)
max_pooling2d_2 input: (None, 8, 8, 256)
MaxPooling2D output: (None, 4, 4, 256)
sequential_3

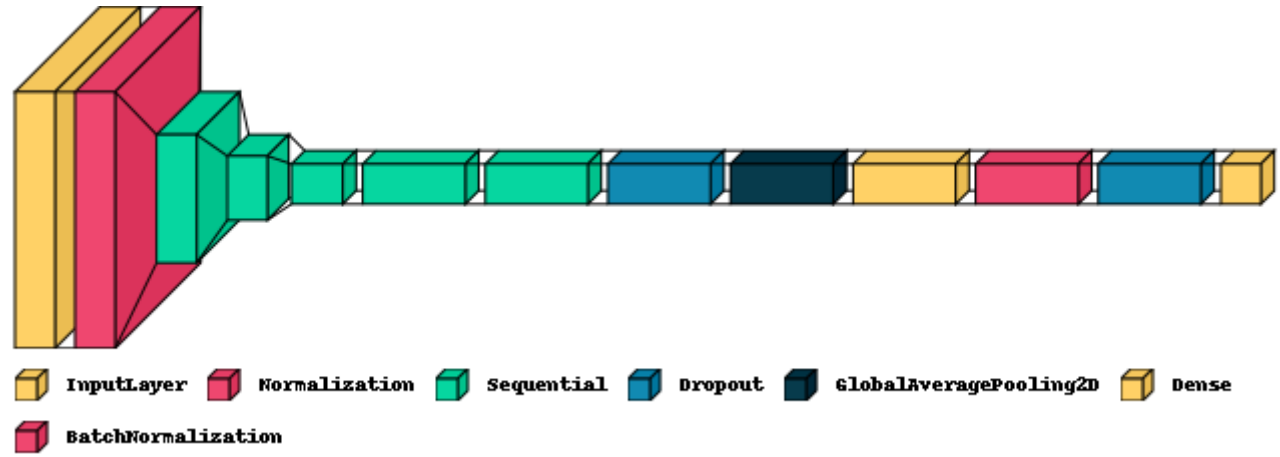

conv2d_7_input | input: | [(None, 4, 4, 256)]
InputLayer | output: | [(None, 4, 4, 256)]
conv2d_7 | input: | (None, 4, 4, 256)
Conv2D | output: | (None, 4, 4, 512)
batch_normalization_7 | input: | (None, 4, 4, 512)
BatchNormalization | output: | (None, 4, 4, 512)
dropout_7 | input: | (None, 4, 4, 512)
Dropout | output: | (None, 4, 4, 512)
conv2d_8 | input: | (None, 4, 4, 512)
Conv2D | output: | (None, 4, 4, 512)
batch_normalization_8 | input: | (None, 4, 4, 512)
BatchNormalization | output: | (None, 4, 4, 512)
dropout_8 | input: | (None, 4, 4, 512)
Dropout | output: | (None, 4, 4, 512)
conv2d_9 | input: | (None, 4, 4, 512)
Conv2D | output: | (None, 4, 4, 512)
batch_normalization_9 | input: | (None, 4, 4, 512)
BatchNormalization | output: | (None, 4, 4, 512)
dropout_9 | input: | (None, 4, 4, 512)
Dropout | output: | (None, 4, 4, 512)


max_pooling2d_3 | input: | (None, 4, 4, 512)
MaxPooling2D | output: | (None, 2, 2, 512)
sequential_4
conv2d_10_input | input: | [(None, 2, 2, 512)]
InputLayer | output: | [(None, 2, 2, 512)]
conv2d_10 | input: | (None, 2, 2, 512)
Conv2D | output: | (None, 2, 2, 512)
batch_normalization_10 | input: | (None, 2, 2, 512)
BatchNormalization | output: | (None, 2, 2, 512)
dropout_10 | input: | (None, 2, 2, 512)
Dropout | output: | (None, 2, 2, 512)
conv2d_11 | input: | (None, 2, 2, 512)
Conv2D | output: | (None, 2, 2, 512)
batch_normalization_11 | input: | (None, 2, 2, 512)
BatchNormalization | output: | (None, 2, 2, 512)
dropout_11 | input: | (None, 2, 2, 512)
Dropout | output: | (None, 2, 2, 512)
conv2d_12 | input: | (None, 2, 2, 512)
Conv2D | output: | (None, 2, 2, 512)
batch_normalization_12 | input: | (None, 2, 2, 512)



In [ ]:
```
visualkeras.layered_view(final_model,legend=True, to_file="vgg.png")
```

Out[ ]:



Summary

In summary, I experimented with various models using transfer learning and tried different image augmentation approach. More room for improvement can be made like tuning the efficientNetModel and using techniques like ensemble.