CIFAR100 Image Classification - Coloured

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. Debanga Raj Neog, P.D. (2020) Cutmix augmentation in python, Medium. Depurr.
    Available at: https://medium.com/depurr/cutmix-augmentation-in-python-bf099a97afac (Accessed: November 25, 2022).
11. 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).

12. 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="fine")
```

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

Item Labels

- 0 : apple
- 1 : aquarium_fish
- 2 : baby
- 3 : bear
- 4 : beaver
- 5 : bed
- 6 : bee
- 7 : beetle
- 8 : bicycle
- 9 : bottle
- 10 : bowl
- 11 : boy
- 12 : bridge
- 13 : bus

- 14 : butterfly
- 15 : camel
- 16 : can
- 17 : castle
- 18 : caterpillar
- 19 : cattle
- 20 : chair
- 21 : chimpanzee
- 22 : clock
- 23 : cloud
- 24 : cockroach
- 25 : couch
- 26 : crab
- 27 : crocodile
- 28 : cup
- 29 : dinosaur
- 30 : dolphin
- 31 : elephant
- 32 : flatfish
- 33 : forest
- 34 : fox
- 35 : girl
- 36 : hamster
- 37 : house
- 38 : kangaroo
- 39 : keyboard
- 40 : lamp
- 41 : lawn_mower
- 42 : leopard
- 43 : lion
- 44 : lizard
- 45 : lobster
- 46 : man
- 47 : maple_tree
- 48 : motorcycle
- 49 : mountain
- 50 : mouse
- 51 : mushroom
- 52 : oak_tree
- 53 : orange
- 54 : orchid
- 55 : otter
- 56 : palm_tree
- 57 : pear
- 58 : pickup_truck
- 59 : pine_tree
- 60 : plain
- 61 : plate

- 62 : poppy
- 63 : porcupine
- 64 : possum
- 65 : rabbit
- 66 : raccoon
- 67 : ray
- 68 : road
- 69 : rocket
- 70 : rose
- 71 : sea
- 72 : seal
- 73 : shark
- 74 : shrew
- 75 : skunk
- 76 : skyscraper
- 77 : snail
- 78 : snake
- 79 : spider
- 80 : squirrel
- 81 : streetcar
- 82 : sunflower
- 83 : sweet_pepper
- 84 : table
- 85 : tank
- 86 : telephone
- 87 : television
- 88 : tiger
- 89 : tractor
- 90 : train
- 91 : trout
- 92 : tulip
- 93 : turtle
- 94 : wardrobe
- 95 : whale
- 96 : willow_tree
- 97 : wolf
- 98 : woman
- 99 : worm

Superclass Labels

- 4, 30, 55, 72, 95 : aquatic mammals
- 1, 32, 67, 73, 91 : fish
- 54, 62, 70, 82, 92 : flowers
- 9, 10, 16, 28, 61 : food containers
- 0, 51, 53, 57, 83 : fruit and vegetables
- 22, 39, 40, 86, 87 : household electrical devices
- 5, 20, 25, 84, 94 : household furniture
- 6, 7, 14, 18, 24 : insects

- 3, 42, 43, 88, 97 : large carnivores
- 12, 17, 37, 68, 76 : large man-made outdoor things
- 23, 33, 49, 60, 71 : large natural outdoor scenes
- 15, 19, 21, 31, 38 : large omnivores and herbivores
- 34, 63, 64, 66, 75 : medium-sized mammals
- 26, 45, 77, 79, 99 : non-insect invertebrates
- 2, 11, 35, 46, 98 : people
- 27, 29, 44, 78, 93 : reptiles
- 36, 50, 65, 74, 80 : small mammals
- 47, 52, 56, 59, 96 : trees
- 8, 13, 48, 58, 90 : vehicles 1
- 41, 69, 81, 85, 89 : vehicles 2

In [ ]:

```python
# super class labels
super_class_labels = {
    "aquatic mammals": [4, 30, 55, 72, 95],
    "fish": [1, 32, 67, 73, 91],
    "flowers": [54, 62, 70, 82, 92],
    "food containers": [9, 10, 16, 28, 61],
    "fruit and vegetables": [0, 51, 53, 57, 83],
    "household electrical devices": [22, 39, 40, 86, 87],
    "household furniture": [5, 20, 25, 84, 94],
    "insects": [6, 7, 14, 18, 24],
    "large carnivores": [3, 42, 43, 88, 97],
    "large man-made outdoor things": [12, 17, 37, 68, 76],
    "large natural outdoor scenes": [23, 33, 49, 60, 71],
    "large omnivores and herbivores": [15, 19, 21, 31, 38],
    "medium-sized mammals": [34, 63, 64, 66, 75],
    "non-insect invertebrates": [26, 45, 77, 79, 99],
    "people": [2, 11, 35, 46, 98],
    "reptiles": [27, 29, 44, 78, 93],
    "small mammals": [36, 50, 65, 74, 80],
    "trees": [47, 52, 56, 59, 96],
    "vehicles 1": [8, 13, 48, 58, 90],
    "vehicles 2": [41, 69, 81, 85, 89],
}

# coarse labels
coarse_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",
}
```

```python
# class labels
class_labels = {
    0: "apple",
    1: "aquarium_fish",
    2: "baby",
    3: "bear",
    4: "beaver",
    5: "bed",
    6: "bee",
    7: "beetle",
    8: "bicycle",
    9: "bottle",
    10: "bowl",
    11: "boy",
    12: "bridge",
    13: "bus",
    14: "butterfly",
    15: "camel",
    16: "can",
    17: "castle",
    18: "caterpillar",
    19: "cattle",
    20: "chair",
    21: "chimpanzee",
    22: "clock",
    23: "cloud",
    24: "cockroach",
    25: "couch",
    26: "crab",
    27: "crocodile",
    28: "cup",
    29: "dinosaur",
    30: "dolphin",
    31: "elephant",
    32: "flatfish",
    33: "forest",
    34: "fox",
    35: "girl",
    36: "hamster",
    37: "house",
    38: "kangaroo",
    39: "keyboard",
    40: "lamp",
    41: "lawn_mower",
    42: "leopard",
    43: "lion",
    44: "lizard",
    45: "lobster",
    46: "man",
    47: "maple_tree",
    48: "motorcycle",
    49: "mountain",
    50: "mouse",
    51: "mushroom",
    52: "oak_tree",
    53: "orange",
    54: "orchid",
    55: "otter",
    56: "palm_tree",
    57: "pear",
    58: "pickup_truck",
    59: "pine_tree",
    60: "plain",
    61: "plate",
    62: "poppy",
    63: "porcupine",
```

```
    64: "possum",
    65: "rabbit",
    66: "raccoon",
    67: "ray",
    68: "road",
    69: "rocket",
    70: "rose",
    71: "sea",
    72: "seal",
    73: "shark",
    74: "shrew",
    75: "skunk",
    76: "skyscraper",
    77: "snail",
    78: "snake",
    79: "spider",
    80: "squirrel",
    81: "streetcar",
    82: "sunflower",
    83: "sweet_pepper",
    84: "table",
    85: "tank",
    86: "telephone",
    87: "television",
    88: "tiger",
    89: "tractor",
    90: "train",
    91: "trout",
    92: "tulip",
    93: "turtle",
    94: "wardrobe",
    95: "whale",
    96: "willow_tree",
    97: "wolf",
    98: "woman",
    99: "worm",
}

NUM_CLASS = 100
```

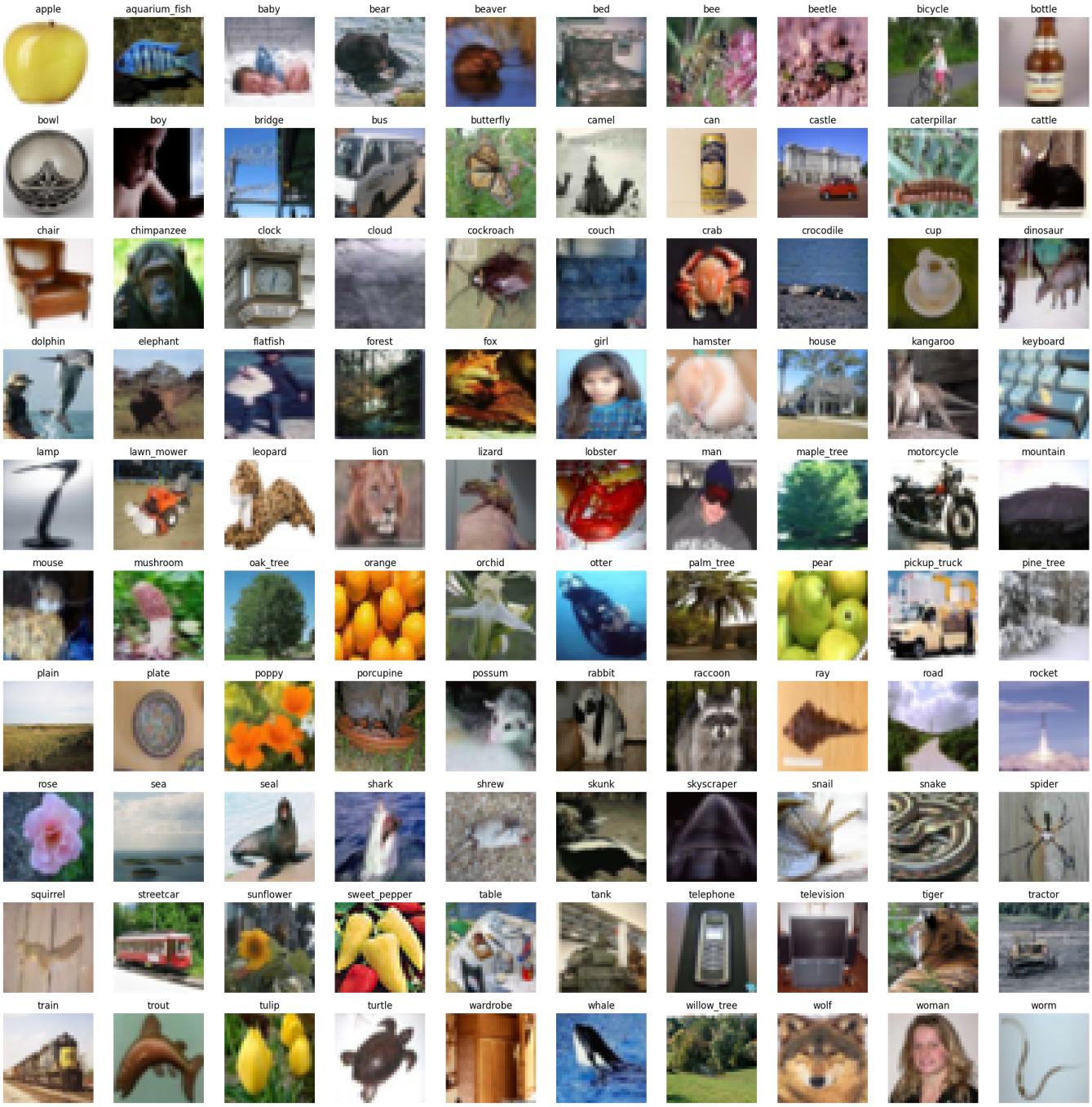
Each image is a 32x32 image as well as 3 color channel [RGB] (coloured image). Therefore, we can set the IMG_SIZE as a tuple (32, 32, 3)

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

Visualising the Dataset

Let's look at what the images look like.

In [ ]:
```
fig, ax = plt.subplots(10, 10, figsize=(20, 20), tight_layout=True)

for label, subplot in enumerate(ax.ravel()):
    subplot.axis("off")
    subplot.imshow(x_train[np.random.choice(np.where(y_train == label)[0])])
    subplot.set_title(class_labels[label])

plt.show()
```



Observations

We can see that the images are not very consistent in the orientation. Data augmentation where images can be flipped/cropped could be possible to do better prediction. Based on the selection of images, There is no clear indication of missed identified items/wrongly augmented items from the dataset.

Class Distribution

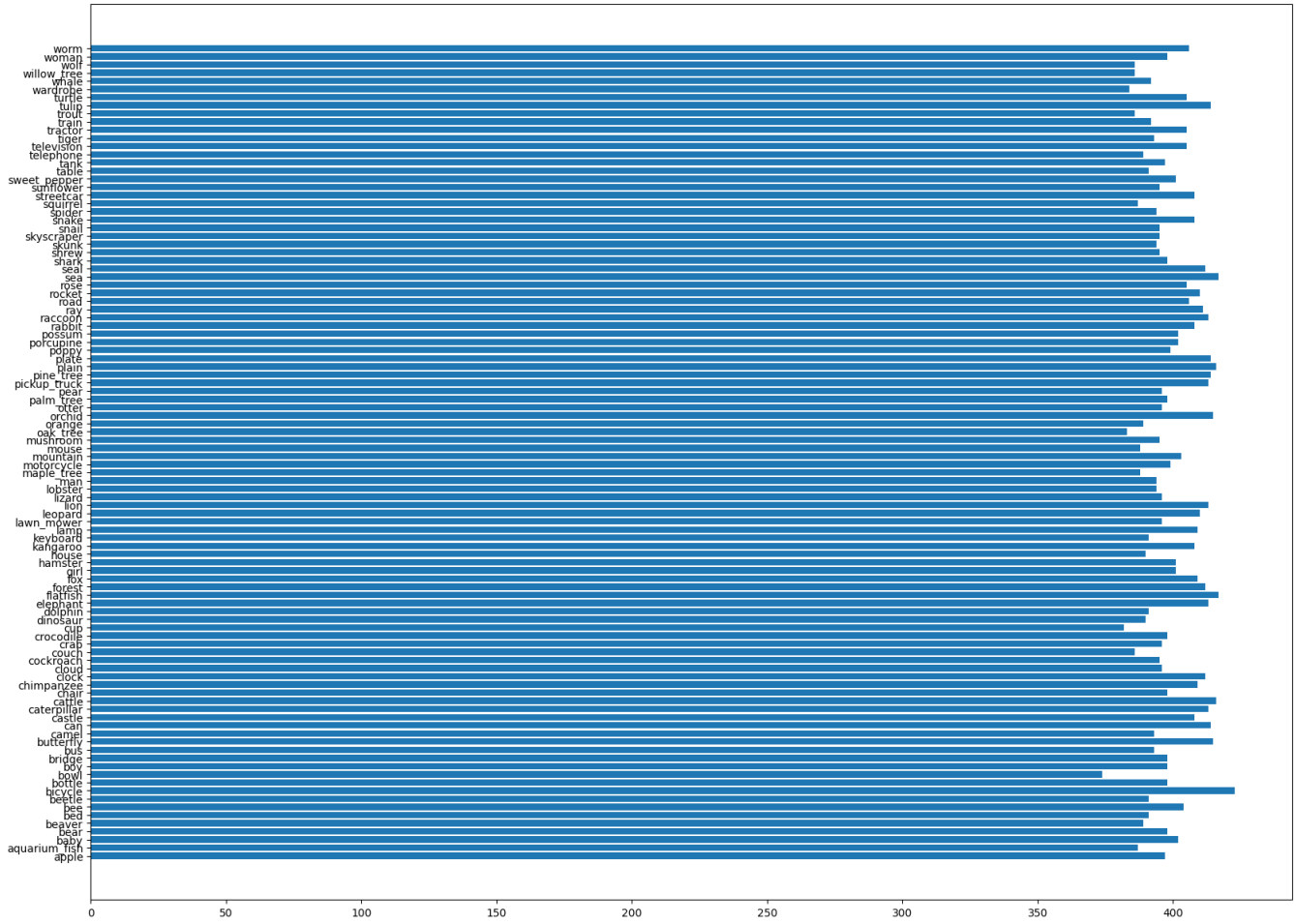
When training a machine learning model, it is always important to check the distribution of the different classes in the dataset. This will inform us which metrics is the best to use and if anything is needed to balance the classes.

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

apple: 397
aquarium_fish: 387
baby: 402
bear: 398
beaver: 389
bed: 391
bee: 404
beetle: 391
bicycle: 423
bottle: 398
bowl: 374
boy: 398
bridge: 398
bus: 393
butterfly: 415
camel: 393
can: 414
castle: 408
caterpillar: 413
cattle: 416
chair: 398
chimpanzee: 409
clock: 412
cloud: 396
cockroach: 395
couch: 386
crab: 396
crocodile: 398
cup: 382
dinosaur: 390
dolphin: 391
elephant: 413
flatfish: 417
forest: 412
fox: 409
girl: 401
hamster: 401
house: 390
kangaroo: 408
keyboard: 391
lamp: 409
lawn_mower: 396
leopard: 410
lion: 413
lizard: 396
lobster: 394
man: 394
maple_tree: 388
motorcycle: 399
mountain: 403
mouse: 388
mushroom: 395
oak_tree: 383
orange: 389
orchid: 415
otter: 396
palm_tree: 398
pear: 396
pickup_truck: 413
pine_tree: 414
plain: 416
plate: 414
poppy: 399
porcupine: 402
possum: 402
rabbit: 408

```
raccoon: 413
ray: 411
road: 406
rocket: 410
rose: 405
sea: 417
seal: 412
shark: 398
shrew: 395
skunk: 394
skyscraper: 395
snail: 395
snake: 408
spider: 394
squirrel: 387
streetcar: 408
sunflower: 395
sweet_pepper: 401
table: 391
tank: 397
telephone: 389
television: 405
tiger: 393
tractor: 405
train: 392
trout: 386
tulip: 414
turtle: 405
wardrobe: 384
whale: 392
willow_tree: 386
wolf: 386
woman: 398
worm: 406
```

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.

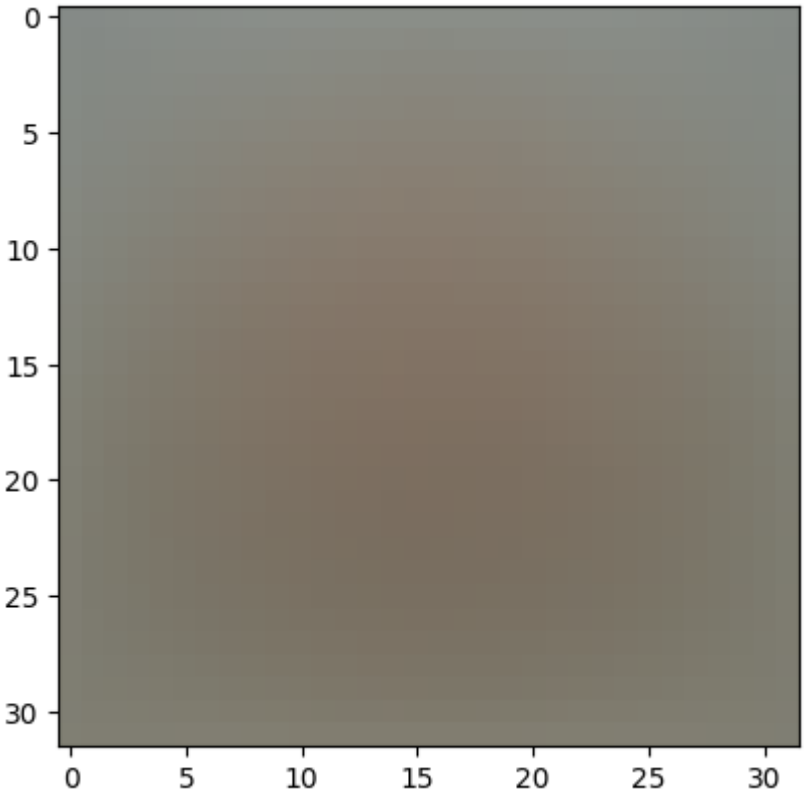
In [ ]:
```
mean, std = np.mean(x_train, axis=(0, 1, 2)), np.std(x_train, axis=(0, 1, 2))
print("Mean:", mean)
print("std:", std)
```

```
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.

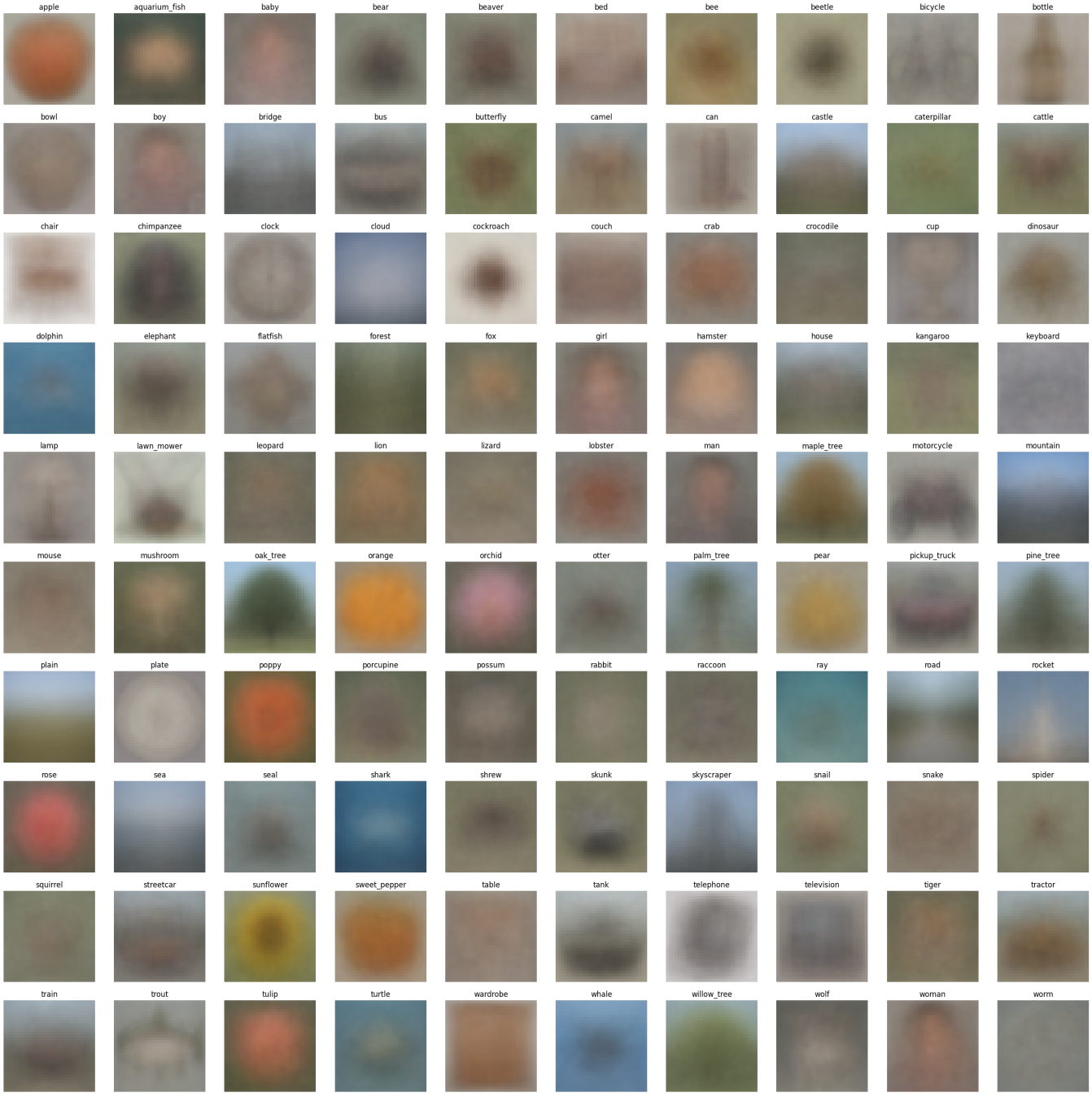
In [ ]:
```
plt.imshow(np.mean(x_train, axis=0) / 255, cmap='Greys')
plt.show()
```



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(10, 10, figsize=(32, 32))

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 trout, sunflower, apple, can, orange, bottle 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 orange and sweet pepper 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. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]
Label: tf.Tensor(19, 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.

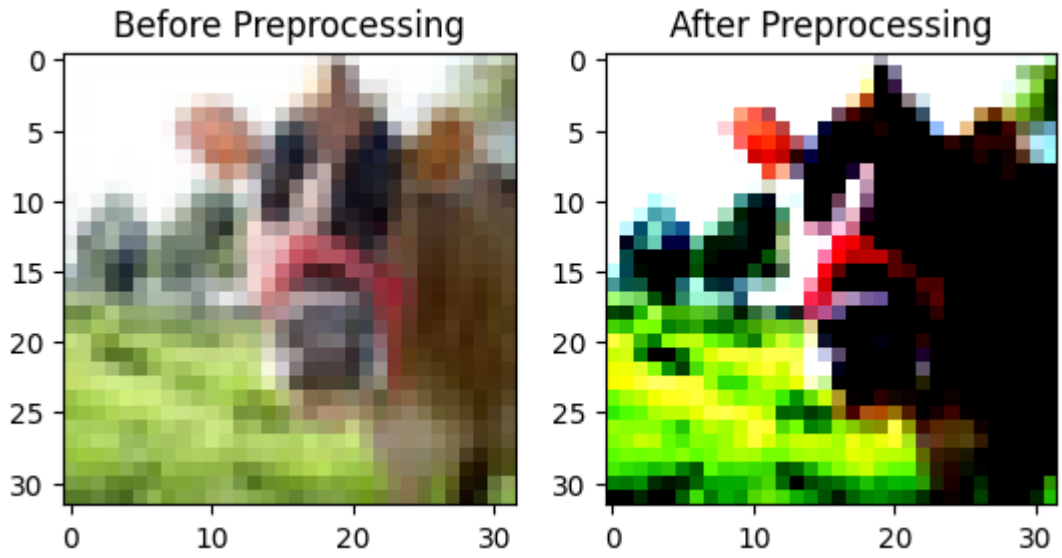
$$X = \frac{X - \mu}{\sigma}$$

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)
```

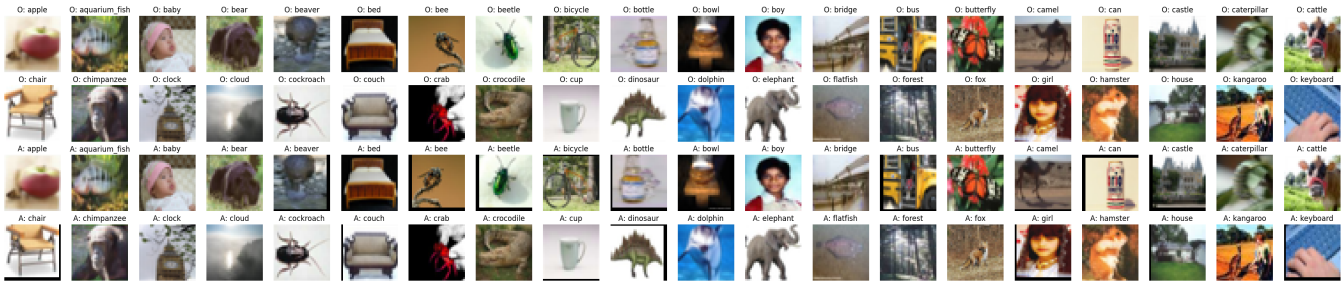
In [ ]:
```
x_train_aug = np.copy(x_train)
```

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(4, 20, figsize=(40, 8))
for idx in range(80):
    subplot = ax.ravel()[idx]
    y_label = np.argmax(y_train, axis=1)
    if idx >= 40:
```

```
        subplot.set_title(f"A: {class_labels[idx % 40]}")
        subplot.imshow(x_train_aug[y_label == idx % 40][0])
    else:
        subplot.set_title(f"O: {class_labels[idx % 40]}")
        subplot.imshow(x_train[y_label == idx % 40][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

CutMix Augmentation

This image augmentation method cuts patches and paste it among training images. This improves the model robustness against input corruptions and its out-of-distribution detection performances.

In [ ]:
```
AUTO = tf.data.AUTOTUNE
```

In [ ]:
```
x_train_cutmix = x_train.copy()
```

In [ ]:
```
def preprocess_image(image, label):
    image = tf.image.resize(image, (32, 32))
    image = tf.image.convert_image_dtype(image, tf.float32) / 255.0
    return image, label
```

In [ ]:
```
train_ds_one = (
    tf.data.Dataset.from_tensor_slices((x_train_cutmix, y_train))
    .shuffle(1024)
    .map(preprocess_image, num_parallel_calls=AUTO)
)
train_ds_two = (
    tf.data.Dataset.from_tensor_slices((x_train_cutmix, y_train))
    .shuffle(1024)
    .map(preprocess_image, num_parallel_calls=AUTO)
)

# Combine two shuffled datasets from the same training data.
train_ds = tf.data.Dataset.zip((train_ds_one, train_ds_two))

val_ds = tf.data.Dataset.from_tensor_slices((x_val, y_val))
val_ds = (
    val_ds.map(preprocess_image, num_parallel_calls=AUTO)
    .batch(BATCH_SIZE)
    .prefetch(AUTO)
)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [25], line 2
      1 train_ds_one = (
----> 2     tf.data.Dataset.from_tensor_slices((x_train_cutmix, y_train))
      3     .shuffle(1024)
      4     .map(preprocess_image, num_parallel_calls=AUTO)
      5 )
      6 train_ds_two = (
      7     tf.data.Dataset.from_tensor_slices((x_train_cutmix, y_train))
      8     .shuffle(1024)
      9     .map(preprocess_image, num_parallel_calls=AUTO)
     10 )
     12 # Combine two shuffled datasets from the same training data.

File c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\tensorflow\python\data\ops
\dataset_ops.py:814, in DatasetV2.from_tensor_slices(tensors, name)
    736 @staticmethod
    737 def from_tensor_slices(tensors, name=None):
    738   """Creates a `Dataset` whose elements are slices of the given tensors.
    739
    740   The given tensors are sliced along their first dimension. This operation
   (...)
    812     Dataset: A `Dataset`.
    813   """
--> 814   return TensorSliceDataset(tensors, name=name)

File c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\tensorflow\python\data\ops
\dataset_ops.py:4720, in TensorSliceDataset.__init__(self, element, is_files, name)
   4717 batch_dim = tensor_shape.Dimension(
   4718     tensor_shape.dimension_value(self._tensors[0].get_shape()[0]))
   4719 for t in self._tensors[1:]:
-> 4720   batch_dim.assert_is_compatible_with(
   4721       tensor_shape.Dimension(
   4722           tensor_shape.dimension_value(t.get_shape()[0])))
   4724 variant_tensor = gen_dataset_ops.tensor_slice_dataset(
   4725     self._tensors,
   4726     output_shapes=structure.get_flat_tensor_shapes(self._structure),
   4727     is_files=is_files,
   4728     metadata=self._metadata.SerializeToString())
   4729 super(TensorSliceDataset, self).__init__(variant_tensor)

File c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\tensorflow\python\framewor
k\tensor_shape.py:299, in Dimension.assert_is_compatible_with(self, other)
    289 """Raises an exception if `other` is not compatible with this Dimension.
    290
    291 Args:
   (...)
    296     is_compatible_with).
    297 """
    298 if not self.is_compatible_with(other):
--> 299   raise ValueError("Dimensions %s and %s are not compatible" %
    300                    (self, other))

ValueError: Dimensions 32 and 40000 are not compatible
```

In [ ]:

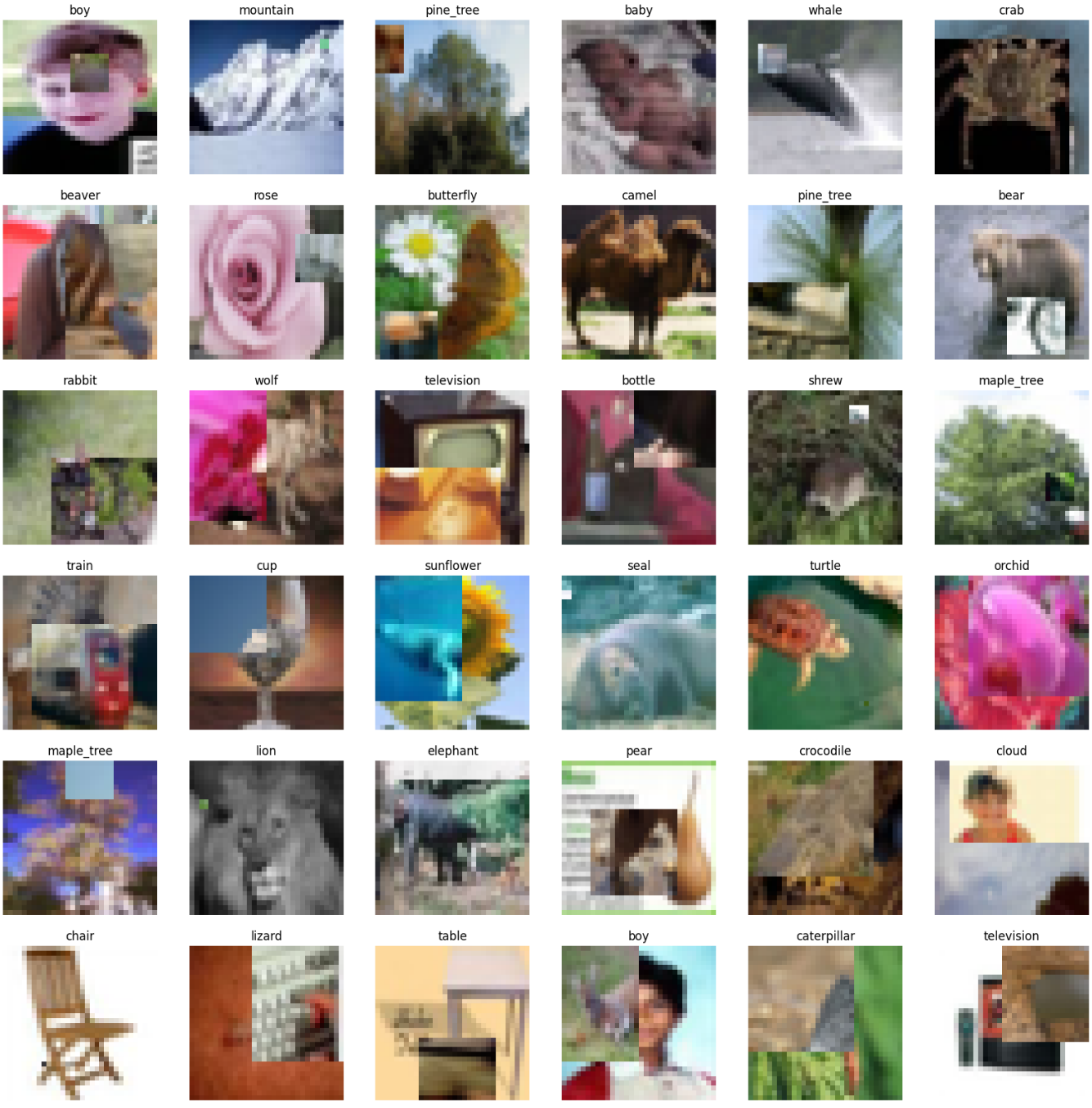
```python
def sample_beta_distribution(size, concentration_0=0.2, concentration_1=0.2):
    gamma_1_sample = tf.random.gamma(shape=[size], alpha=concentration_1)
    gamma_2_sample = tf.random.gamma(shape=[size], alpha=concentration_0)
    return gamma_1_sample / (gamma_1_sample + gamma_2_sample)


@tf.function
def get_box(lambda_value):
    cut_rat = tf.math.sqrt(1.0 - lambda_value)
```

```
    cut_w = 32 * cut_rat  # rw
    cut_w = tf.cast(cut_w, tf.int32)

    cut_h = 32 * cut_rat  # rh
    cut_h = tf.cast(cut_h, tf.int32)

    cut_x = tf.random.uniform((1,), minval=0, maxval=32, dtype=tf.int32)  # rx
    cut_y = tf.random.uniform((1,), minval=0, maxval=32, dtype=tf.int32)  # ry

    boundaryx1 = tf.clip_by_value(cut_x[0] - cut_w // 2, 0, 32)
    boundaryy1 = tf.clip_by_value(cut_y[0] - cut_h // 2, 0, 32)
    bbx2 = tf.clip_by_value(cut_x[0] + cut_w // 2, 0, 32)
    bby2 = tf.clip_by_value(cut_y[0] + cut_h // 2, 0, 32)

    target_h = bby2 - boundaryy1
    if target_h == 0:
        target_h += 1

    target_w = bbx2 - boundaryx1
    if target_w == 0:
        target_w += 1

    return boundaryx1, boundaryy1, target_h, target_w


@tf.function
def cutmix(train_ds_one, train_ds_two):
    (image1, label1), (image2, label2) = train_ds_one, train_ds_two

    alpha = [0.25]
    beta = [0.25]

    # Get a sample from the Beta distribution
    lambda_value = sample_beta_distribution(1, alpha, beta)

    # Define Lambda
    lambda_value = lambda_value[0][0]

    # Get the bounding box offsets, heights and widths
    boundaryx1, boundaryy1, target_h, target_w = get_box(lambda_value)

    # Get a patch from the second image (`image2`)
    crop2 = tf.image.crop_to_bounding_box(
        image2, boundaryy1, boundaryx1, target_h, target_w
    )
    # Pad the `image2` patch (`crop2`) with the same offset
    image2 = tf.image.pad_to_bounding_box(
        crop2, boundaryy1, boundaryx1, 32, 32
    )
    # Get a patch from the first image (`image1`)
    crop1 = tf.image.crop_to_bounding_box(
        image1, boundaryy1, boundaryx1, target_h, target_w
    )
    # Pad the `image1` patch (`crop1`) with the same offset
    img1 = tf.image.pad_to_bounding_box(
        crop1, boundaryy1, boundaryx1, 32, 32
    )

    # Modify the first image by subtracting the patch from `image1`
    # (before applying the `image2` patch)
    image1 = image1 - img1
    # Add the modified `image1` and `image2`  together to get the CutMix image
    image = image1 + image2

    # Adjust Lambda in accordance to the pixel ration
    lambda_value = 1 - (target_w * target_h) / (32 * 32)
```

```
    lambda_value = tf.cast(lambda_value, tf.float32)

    # Combine the labels of both images
    label = lambda_value * label1 + (1 - lambda_value) * label2
    return image, label
```

In [ ]:

```
# Create the new dataset using our `cutmix` utility
train_ds_cutmix = (
    train_ds.shuffle(1024)
    .map(cutmix, num_parallel_calls=AUTO)
    .batch(BATCH_SIZE)
    .prefetch(AUTO)
)

# Let's preview 36 samples from the dataset
x_train_cutmix, y_train_cutmix = next(iter(train_ds_cutmix))
plt.figure(figsize=(20, 20))
for i in range(36):
    ax = plt.subplot(6, 6, i + 1)
    plt.title(class_labels[np.argmax(y_train_cutmix[i])])
    plt.imshow(x_train_cutmix[i])
    plt.axis("off")
```



Observations

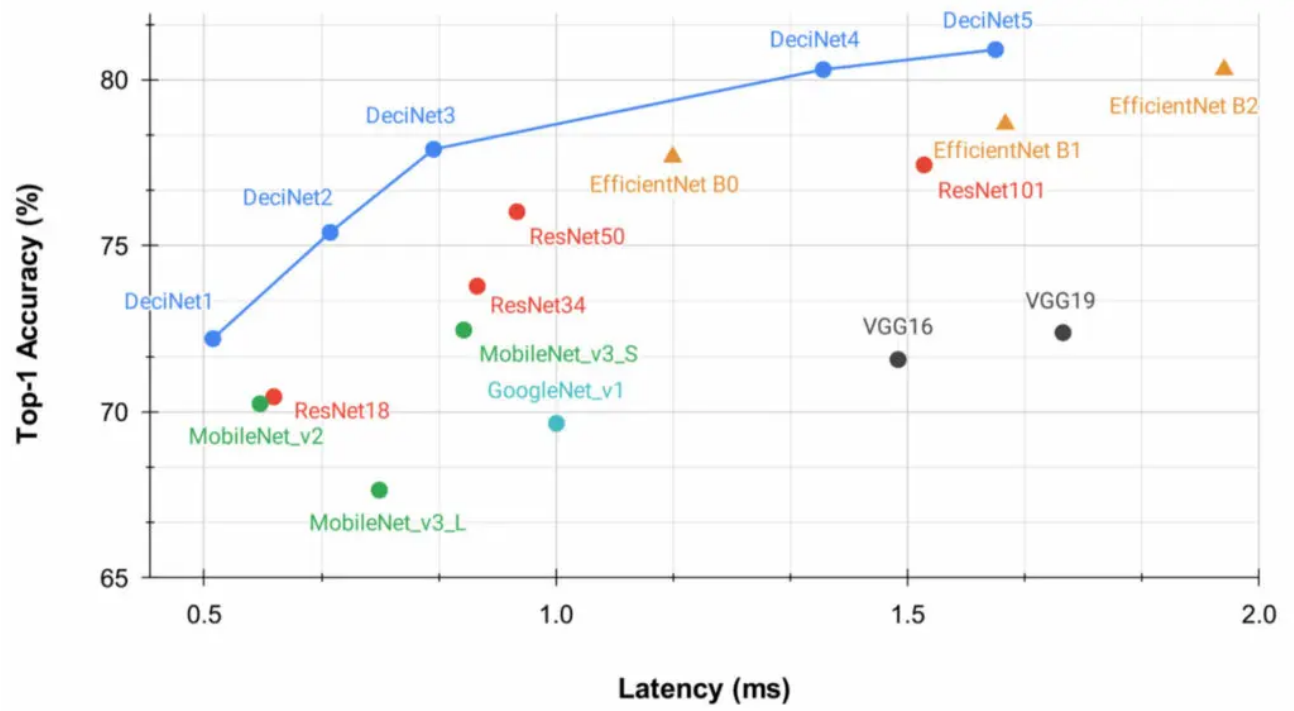
As we can see, the images have been cut and pasted and mixed together. This shows that the image augmentation works

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'])
```

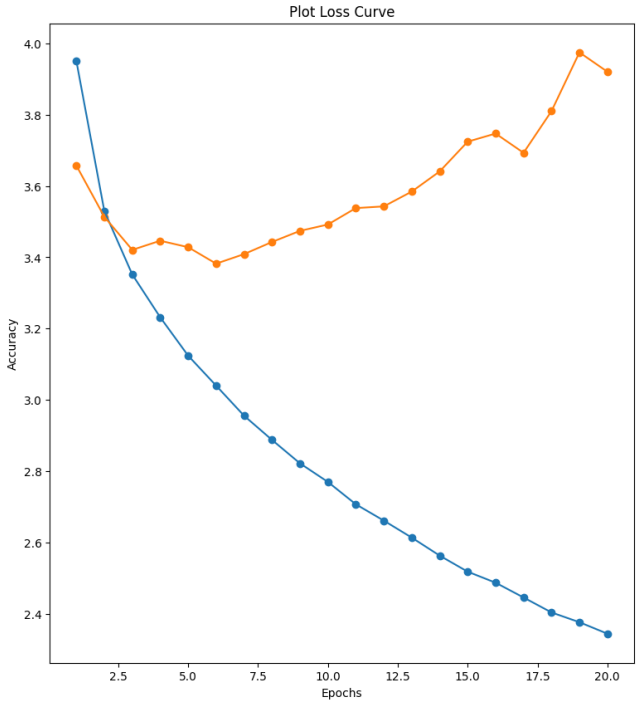
In [ ]:
```
baseModelHistory = baseModel.fit(x_train, y_train, epochs=100,
                                 validation_data=(x_val, y_val), batch_size=BATCH_SIZE, callb
```

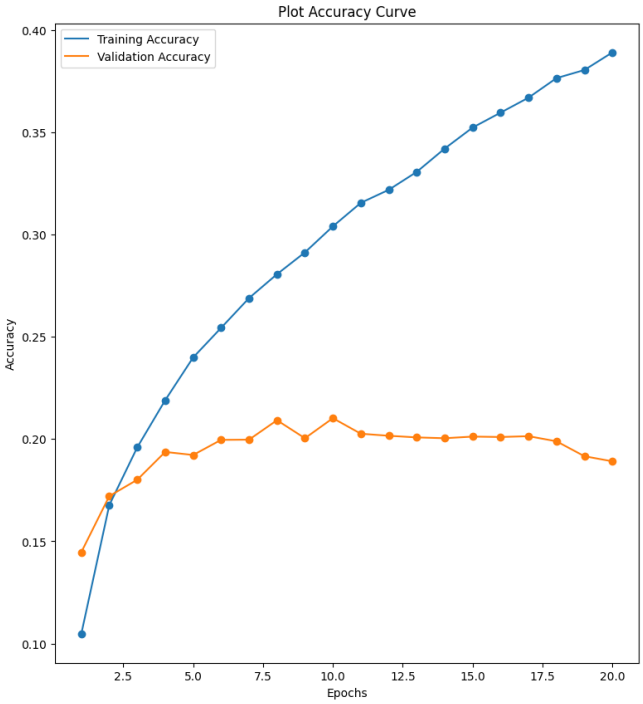
```
Epoch 1/100
625/625 [==============================] - 4s 6ms/step - loss: 3.9509 - accuracy: 0.1048 - va
l_loss: 3.6575 - val_accuracy: 0.1447
Epoch 2/100
625/625 [==============================] - 3s 6ms/step - loss: 3.5302 - accuracy: 0.1677 - va
l_loss: 3.5138 - val_accuracy: 0.1721
Epoch 3/100
625/625 [==============================] - 4s 6ms/step - loss: 3.3519 - accuracy: 0.1959 - va
l_loss: 3.4208 - val_accuracy: 0.1801
Epoch 4/100
625/625 [==============================] - 3s 6ms/step - loss: 3.2317 - accuracy: 0.2190 - va
l_loss: 3.4465 - val_accuracy: 0.1937
Epoch 5/100
625/625 [==============================] - 3s 6ms/step - loss: 3.1240 - accuracy: 0.2399 - va
l_loss: 3.4284 - val_accuracy: 0.1922
Epoch 6/100
625/625 [==============================] - 3s 6ms/step - loss: 3.0400 - accuracy: 0.2544 - va
l_loss: 3.3822 - val_accuracy: 0.1996
Epoch 7/100
625/625 [==============================] - 4s 6ms/step - loss: 2.9556 - accuracy: 0.2689 - va
l_loss: 3.4089 - val_accuracy: 0.1997
Epoch 8/100
625/625 [==============================] - 3s 6ms/step - loss: 2.8877 - accuracy: 0.2806 - va
l_loss: 3.4430 - val_accuracy: 0.2092
Epoch 9/100
625/625 [==============================] - 4s 6ms/step - loss: 2.8220 - accuracy: 0.2913 - va
l_loss: 3.4741 - val_accuracy: 0.2004
Epoch 10/100
625/625 [==============================] - 4s 6ms/step - loss: 2.7697 - accuracy: 0.3040 - va
l_loss: 3.4920 - val_accuracy: 0.2102
Epoch 11/100
625/625 [==============================] - 4s 6ms/step - loss: 2.7070 - accuracy: 0.3155 - va
l_loss: 3.5381 - val_accuracy: 0.2026
Epoch 12/100
625/625 [==============================] - 4s 6ms/step - loss: 2.6613 - accuracy: 0.3219 - va
l_loss: 3.5428 - val_accuracy: 0.2016
Epoch 13/100
625/625 [==============================] - 4s 6ms/step - loss: 2.6136 - accuracy: 0.3306 - va
l_loss: 3.5844 - val_accuracy: 0.2008
Epoch 14/100
625/625 [==============================] - 4s 6ms/step - loss: 2.5630 - accuracy: 0.3420 - va
l_loss: 3.6410 - val_accuracy: 0.2004
Epoch 15/100
625/625 [==============================] - 4s 6ms/step - loss: 2.5180 - accuracy: 0.3523 - va
l_loss: 3.7244 - val_accuracy: 0.2012
Epoch 16/100
625/625 [==============================] - 4s 6ms/step - loss: 2.4870 - accuracy: 0.3596 - va
l_loss: 3.7472 - val_accuracy: 0.2010
Epoch 17/100
625/625 [==============================] - 3s 5ms/step - loss: 2.4456 - accuracy: 0.3669 - va
l_loss: 3.6932 - val_accuracy: 0.2014
Epoch 18/100
625/625 [==============================] - 3s 5ms/step - loss: 2.4036 - accuracy: 0.3765 - va
l_loss: 3.8101 - val_accuracy: 0.1989
Epoch 19/100
625/625 [==============================] - 3s 6ms/step - loss: 2.3762 - accuracy: 0.3805 - va
l_loss: 3.9748 - val_accuracy: 0.1916
Epoch 20/100
625/625 [==============================] - 3s 6ms/step - loss: 2.3440 - accuracy: 0.3891 - va
l_loss: 3.9206 - val_accuracy: 0.1891
```

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

```
{'Model Name': 'baseline', 'Epochs': 20, 'Batch Size': 64, 'Train Loss': 2.769735097885132,
'Val Loss': 3.4920077323913574, 'Train Acc': 0.30399999022483826, 'Val Acc': 0.21019999682903
29, '[Train - Val] Acc': 0.09379999339580536}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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

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 30.3% and 21.1%. 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'])
```

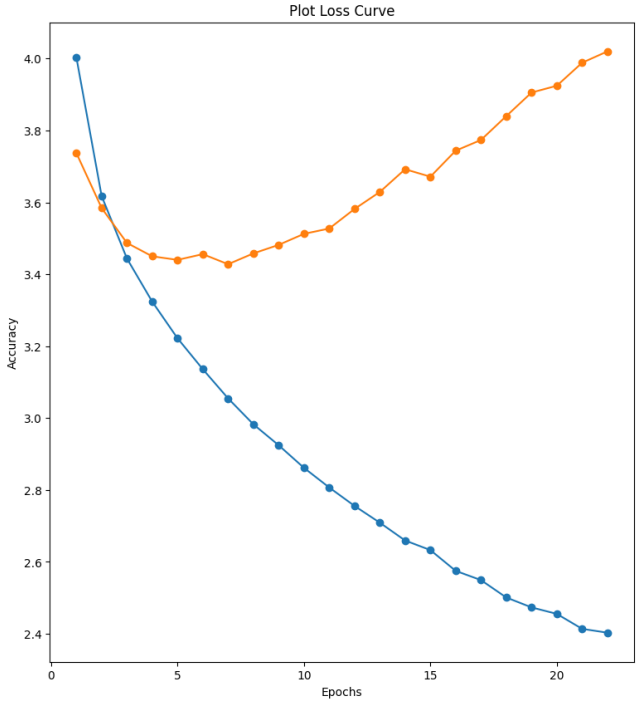
In [ ]:
```
baseAugModelHistory = baseAugModel.fit(x_train_aug, y_train, epochs=100,
                                       validation_data=(x_val, y_val), batch_size=BATCH_SIZE, callb
```

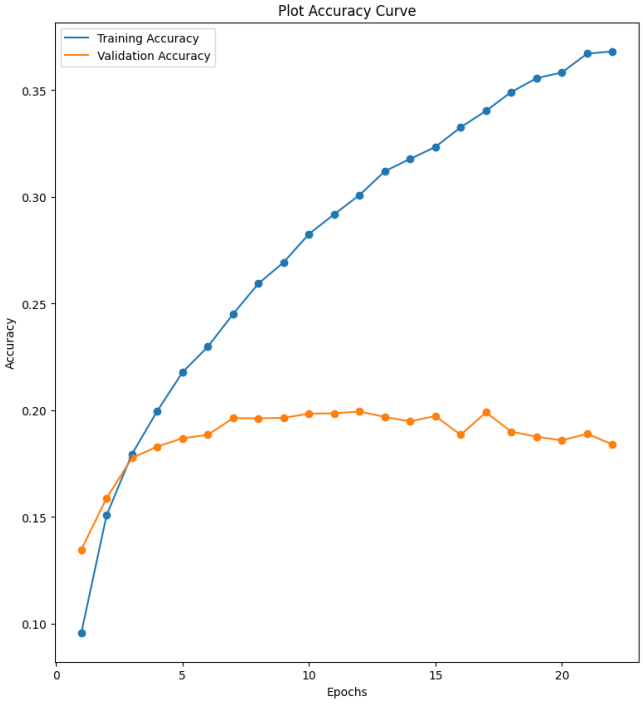
```
Epoch 1/100
625/625 [==============================] - 4s 5ms/step - loss: 4.0036 - accuracy: 0.0958 - va
l_loss: 3.7374 - val_accuracy: 0.1347
Epoch 2/100
625/625 [==============================] - 3s 6ms/step - loss: 3.6176 - accuracy: 0.1508 - va
l_loss: 3.5842 - val_accuracy: 0.1589
Epoch 3/100
625/625 [==============================] - 3s 5ms/step - loss: 3.4439 - accuracy: 0.1794 - va
l_loss: 3.4872 - val_accuracy: 0.1776
Epoch 4/100
625/625 [==============================] - 3s 5ms/step - loss: 3.3238 - accuracy: 0.1996 - va
l_loss: 3.4499 - val_accuracy: 0.1830
Epoch 5/100
625/625 [==============================] - 3s 6ms/step - loss: 3.2220 - accuracy: 0.2178 - va
l_loss: 3.4401 - val_accuracy: 0.1869
Epoch 6/100
625/625 [==============================] - 3s 5ms/step - loss: 3.1359 - accuracy: 0.2298 - va
l_loss: 3.4560 - val_accuracy: 0.1885
Epoch 7/100
625/625 [==============================] - 3s 6ms/step - loss: 3.0548 - accuracy: 0.2451 - va
l_loss: 3.4280 - val_accuracy: 0.1963
Epoch 8/100
625/625 [==============================] - 3s 5ms/step - loss: 2.9827 - accuracy: 0.2594 - va
l_loss: 3.4580 - val_accuracy: 0.1962
Epoch 9/100
625/625 [==============================] - 3s 5ms/step - loss: 2.9244 - accuracy: 0.2693 - va
l_loss: 3.4818 - val_accuracy: 0.1965
Epoch 10/100
625/625 [==============================] - 3s 5ms/step - loss: 2.8616 - accuracy: 0.2826 - va
l_loss: 3.5125 - val_accuracy: 0.1984
Epoch 11/100
625/625 [==============================] - 4s 6ms/step - loss: 2.8068 - accuracy: 0.2919 - va
l_loss: 3.5273 - val_accuracy: 0.1986
Epoch 12/100
625/625 [==============================] - 4s 6ms/step - loss: 2.7556 - accuracy: 0.3009 - va
l_loss: 3.5817 - val_accuracy: 0.1994
Epoch 13/100
625/625 [==============================] - 4s 6ms/step - loss: 2.7090 - accuracy: 0.3120 - va
l_loss: 3.6290 - val_accuracy: 0.1969
Epoch 14/100
625/625 [==============================] - 3s 6ms/step - loss: 2.6594 - accuracy: 0.3178 - va
l_loss: 3.6922 - val_accuracy: 0.1948
Epoch 15/100
625/625 [==============================] - 4s 6ms/step - loss: 2.6327 - accuracy: 0.3234 - va
l_loss: 3.6715 - val_accuracy: 0.1973
Epoch 16/100
625/625 [==============================] - 3s 5ms/step - loss: 2.5749 - accuracy: 0.3326 - va
l_loss: 3.7438 - val_accuracy: 0.1885
Epoch 17/100
625/625 [==============================] - 3s 6ms/step - loss: 2.5493 - accuracy: 0.3403 - va
l_loss: 3.7736 - val_accuracy: 0.1990
Epoch 18/100
625/625 [==============================] - 3s 6ms/step - loss: 2.5009 - accuracy: 0.3491 - va
l_loss: 3.8399 - val_accuracy: 0.1900
Epoch 19/100
625/625 [==============================] - 4s 6ms/step - loss: 2.4733 - accuracy: 0.3557 - va
l_loss: 3.9056 - val_accuracy: 0.1876
Epoch 20/100
625/625 [==============================] - 3s 6ms/step - loss: 2.4553 - accuracy: 0.3583 - va
l_loss: 3.9245 - val_accuracy: 0.1859
Epoch 21/100
625/625 [==============================] - 3s 6ms/step - loss: 2.4136 - accuracy: 0.3672 - va
l_loss: 3.9885 - val_accuracy: 0.1890
Epoch 22/100
625/625 [==============================] - 3s 6ms/step - loss: 2.4029 - accuracy: 0.3682 - va
l_loss: 4.0200 - val_accuracy: 0.1840
```

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

```
{'Model Name': 'baselineAug', 'Epochs': 22, 'Batch Size': 64, 'Train Loss': 2.755602359771728
5, 'Val Loss': 3.581662178039551, 'Train Acc': 0.3008750081062317, 'Val Acc': 0.1993999928236
0077, '[Train - Val] Acc': 0.10147501528263092}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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 decrease in validation loss which means that the model is becoming more 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.

Training baseline model with CutMix Augmentation

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

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)
baseCutMixModel = Model(inputs=inputs, outputs=x, name="baselineCutMix")
baseCutMixModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                  loss='categorical_crossentropy', metrics=['accuracy'])
```

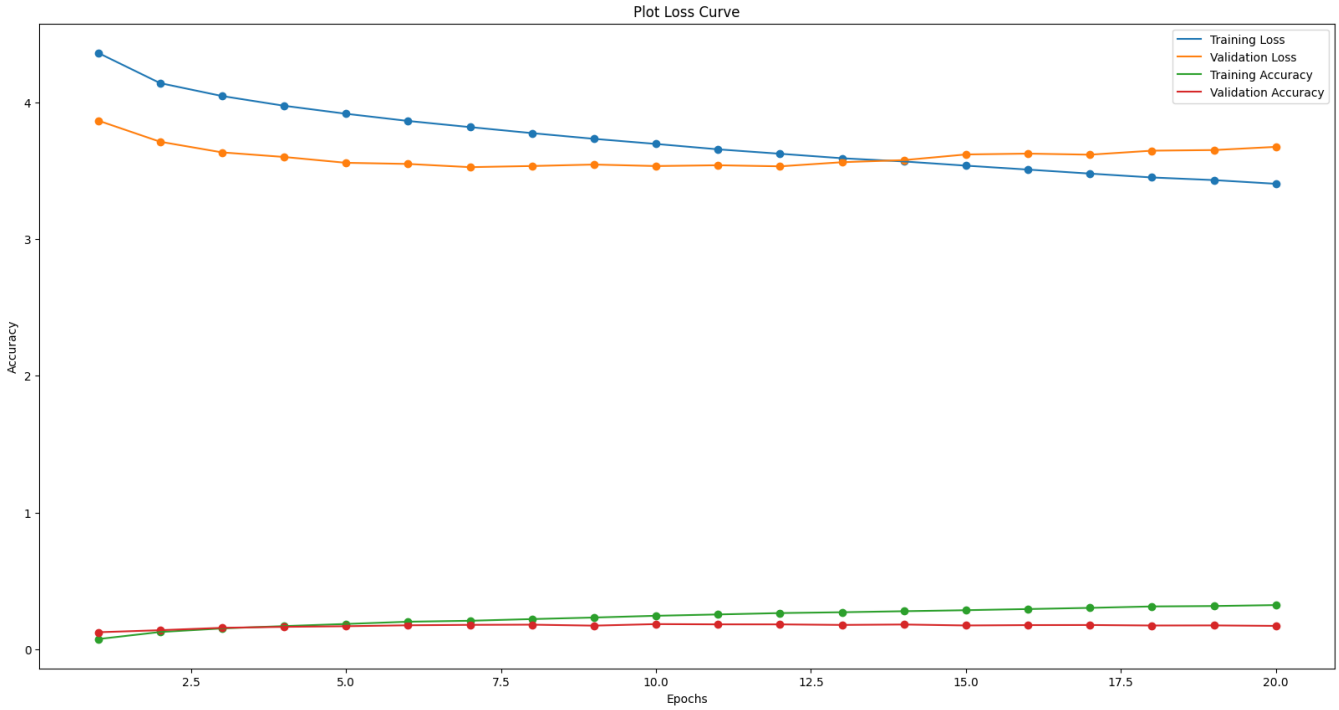
In [ ]:
```
baseCutMixModelHistory = baseCutMixModel.fit(train_ds_cutmix, epochs=100,
                                validation_data=(x_val, y_val), batch_size=BATCH_SIZE, callb
```

```
Epoch 1/100
625/625 [==============================] - 5s 8ms/step - loss: 4.3627 - accuracy: 0.0739 - va
l_loss: 3.8675 - val_accuracy: 0.1239
Epoch 2/100
625/625 [==============================] - 5s 8ms/step - loss: 4.1414 - accuracy: 0.1256 - va
l_loss: 3.7126 - val_accuracy: 0.1388
Epoch 3/100
625/625 [==============================] - 4s 7ms/step - loss: 4.0469 - accuracy: 0.1518 - va
l_loss: 3.6346 - val_accuracy: 0.1563
Epoch 4/100
625/625 [==============================] - 4s 7ms/step - loss: 3.9757 - accuracy: 0.1688 - va
l_loss: 3.6011 - val_accuracy: 0.1633
Epoch 5/100
625/625 [==============================] - 5s 8ms/step - loss: 3.9172 - accuracy: 0.1845 - va
l_loss: 3.5585 - val_accuracy: 0.1680
Epoch 6/100
625/625 [==============================] - 4s 7ms/step - loss: 3.8646 - accuracy: 0.2008 - va
l_loss: 3.5501 - val_accuracy: 0.1748
Epoch 7/100
625/625 [==============================] - 4s 7ms/step - loss: 3.8196 - accuracy: 0.2079 - va
l_loss: 3.5266 - val_accuracy: 0.1779
Epoch 8/100
625/625 [==============================] - 5s 8ms/step - loss: 3.7755 - accuracy: 0.2202 - va
l_loss: 3.5350 - val_accuracy: 0.1793
Epoch 9/100
625/625 [==============================] - 4s 7ms/step - loss: 3.7339 - accuracy: 0.2315 - va
l_loss: 3.5457 - val_accuracy: 0.1722
Epoch 10/100
625/625 [==============================] - 4s 6ms/step - loss: 3.6965 - accuracy: 0.2444 - va
l_loss: 3.5348 - val_accuracy: 0.1835
Epoch 11/100
625/625 [==============================] - 4s 6ms/step - loss: 3.6570 - accuracy: 0.2540 - va
l_loss: 3.5406 - val_accuracy: 0.1816
Epoch 12/100
625/625 [==============================] - 4s 6ms/step - loss: 3.6247 - accuracy: 0.2642 - va
l_loss: 3.5327 - val_accuracy: 0.1814
Epoch 13/100
625/625 [==============================] - 4s 6ms/step - loss: 3.5919 - accuracy: 0.2702 - va
l_loss: 3.5628 - val_accuracy: 0.1774
Epoch 14/100
625/625 [==============================] - 4s 6ms/step - loss: 3.5683 - accuracy: 0.2777 - va
l_loss: 3.5781 - val_accuracy: 0.1805
Epoch 15/100
625/625 [==============================] - 4s 6ms/step - loss: 3.5376 - accuracy: 0.2852 - va
l_loss: 3.6196 - val_accuracy: 0.1735
Epoch 16/100
625/625 [==============================] - 4s 6ms/step - loss: 3.5089 - accuracy: 0.2936 - va
l_loss: 3.6257 - val_accuracy: 0.1761
Epoch 17/100
625/625 [==============================] - 4s 6ms/step - loss: 3.4795 - accuracy: 0.3022 - va
l_loss: 3.6179 - val_accuracy: 0.1768
Epoch 18/100
625/625 [==============================] - 4s 6ms/step - loss: 3.4512 - accuracy: 0.3126 - va
l_loss: 3.6472 - val_accuracy: 0.1732
Epoch 19/100
625/625 [==============================] - 4s 6ms/step - loss: 3.4320 - accuracy: 0.3156 - va
l_loss: 3.6519 - val_accuracy: 0.1738
Epoch 20/100
625/625 [==============================] - 4s 6ms/step - loss: 3.4049 - accuracy: 0.3227 - va
l_loss: 3.6750 - val_accuracy: 0.1704
```

In [ ]:

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\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': 'baselineCutMix', 'Epochs': 20, 'Batch Size': 64, 'Train Loss': 3.696510076522
827, 'Val Loss': 3.534803628921509, 'Train Acc': 0.24435000121593475, 'Val Acc': 0.1835000067
949295, '[Train - Val] Acc': 0.06084999442100525}
```



Observation

We can also see that by cut mixing the data, we see that the accuracy for both training and validation decreased, this is likely because the model is too simple and it was not able to find the similarity and between the different images. Furthermore, the dataset is relatively small which makes the cutmix algorithm not as strong as the cutmix algorithm works better with larger dataset like imagenet.

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, 100)               12900

=================================================================
Total params: 818,283
Trainable params: 818,276
Non-trainable params: 7
_________________________________________________________________
```

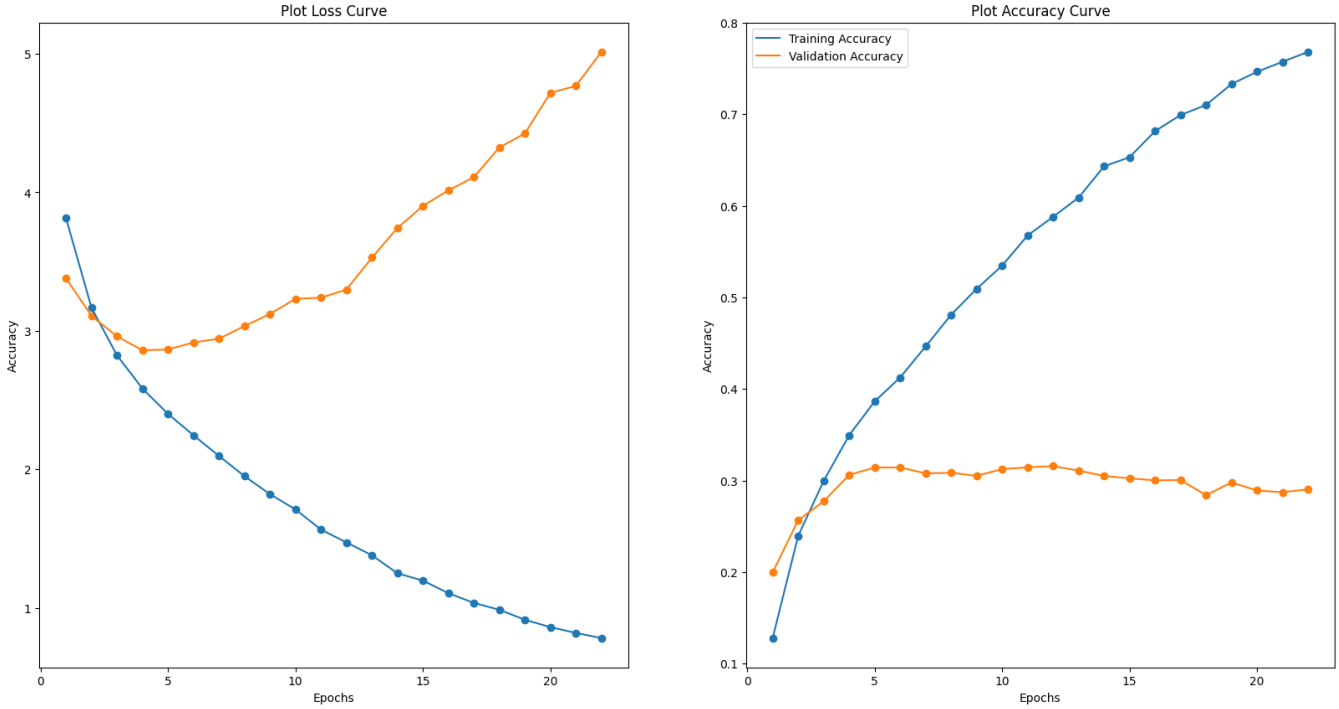
In [ ]:
```
conv2DModelHistory = conv2DModel.fit(x_train, y_train, epochs=100,
                                     validation_data=(x_val, y_val), batch_size=BATCH_SIZE, c
```

```
Epoch 1/100
625/625 [==============================] - 4s 6ms/step - loss: 3.8144 - accuracy: 0.1280 - va
l_loss: 3.3800 - val_accuracy: 0.1998
Epoch 2/100
625/625 [==============================] - 3s 5ms/step - loss: 3.1669 - accuracy: 0.2395 - va
l_loss: 3.1078 - val_accuracy: 0.2567
Epoch 3/100
625/625 [==============================] - 4s 6ms/step - loss: 2.8232 - accuracy: 0.2996 - va
l_loss: 2.9606 - val_accuracy: 0.2773
Epoch 4/100
625/625 [==============================] - 3s 5ms/step - loss: 2.5829 - accuracy: 0.3494 - va
l_loss: 2.8591 - val_accuracy: 0.3062
Epoch 5/100
625/625 [==============================] - 4s 6ms/step - loss: 2.4009 - accuracy: 0.3865 - va
l_loss: 2.8660 - val_accuracy: 0.3143
Epoch 6/100
625/625 [==============================] - 3s 6ms/step - loss: 2.2461 - accuracy: 0.4125 - va
l_loss: 2.9159 - val_accuracy: 0.3143
Epoch 7/100
625/625 [==============================] - 3s 5ms/step - loss: 2.0967 - accuracy: 0.4465 - va
l_loss: 2.9434 - val_accuracy: 0.3079
Epoch 8/100
625/625 [==============================] - 3s 5ms/step - loss: 1.9509 - accuracy: 0.4811 - va
l_loss: 3.0342 - val_accuracy: 0.3085
Epoch 9/100
625/625 [==============================] - 3s 6ms/step - loss: 1.8208 - accuracy: 0.5094 - va
l_loss: 3.1223 - val_accuracy: 0.3052
Epoch 10/100
625/625 [==============================] - 3s 5ms/step - loss: 1.7108 - accuracy: 0.5347 - va
l_loss: 3.2311 - val_accuracy: 0.3125
Epoch 11/100
625/625 [==============================] - 3s 5ms/step - loss: 1.5652 - accuracy: 0.5676 - va
l_loss: 3.2395 - val_accuracy: 0.3145
Epoch 12/100
625/625 [==============================] - 3s 5ms/step - loss: 1.4723 - accuracy: 0.5881 - va
l_loss: 3.2979 - val_accuracy: 0.3158
Epoch 13/100
625/625 [==============================] - 3s 5ms/step - loss: 1.3796 - accuracy: 0.6090 - va
l_loss: 3.5284 - val_accuracy: 0.3109
Epoch 14/100
625/625 [==============================] - 3s 5ms/step - loss: 1.2498 - accuracy: 0.6431 - va
l_loss: 3.7432 - val_accuracy: 0.3051
Epoch 15/100
625/625 [==============================] - 3s 5ms/step - loss: 1.1967 - accuracy: 0.6531 - va
l_loss: 3.9018 - val_accuracy: 0.3025
Epoch 16/100
625/625 [==============================] - 3s 5ms/step - loss: 1.1059 - accuracy: 0.6816 - va
l_loss: 4.0146 - val_accuracy: 0.3002
Epoch 17/100
625/625 [==============================] - 3s 5ms/step - loss: 1.0357 - accuracy: 0.6992 - va
l_loss: 4.1100 - val_accuracy: 0.3006
Epoch 18/100
625/625 [==============================] - 3s 5ms/step - loss: 0.9860 - accuracy: 0.7101 - va
l_loss: 4.3243 - val_accuracy: 0.2841
Epoch 19/100
625/625 [==============================] - 3s 5ms/step - loss: 0.9141 - accuracy: 0.7330 - va
l_loss: 4.4254 - val_accuracy: 0.2979
Epoch 20/100
625/625 [==============================] - 3s 6ms/step - loss: 0.8608 - accuracy: 0.7464 - va
l_loss: 4.7184 - val_accuracy: 0.2892
Epoch 21/100
625/625 [==============================] - 3s 5ms/step - loss: 0.8191 - accuracy: 0.7573 - va
l_loss: 4.7689 - val_accuracy: 0.2873
Epoch 22/100
625/625 [==============================] - 3s 5ms/step - loss: 0.7818 - accuracy: 0.7681 - va
l_loss: 5.0145 - val_accuracy: 0.2904
```

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

```
{'Model Name': 'conv2D', 'Epochs': 22, 'Batch Size': 64, 'Train Loss': 1.4723334312438965, 'V
al Loss': 3.2979133129119873, 'Train Acc': 0.5880500078201294, 'Val Acc': 0.3158000111579895,
'[Train - Val] Acc': 0.2722499966621399}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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 Conv2D model with the baseline model, we can see that there is a huge difference in the training accuracy but the validation accuracy still remain around 30%. This suggest that there is some form of overfitting.

Training conv2D model with Data Augmentation

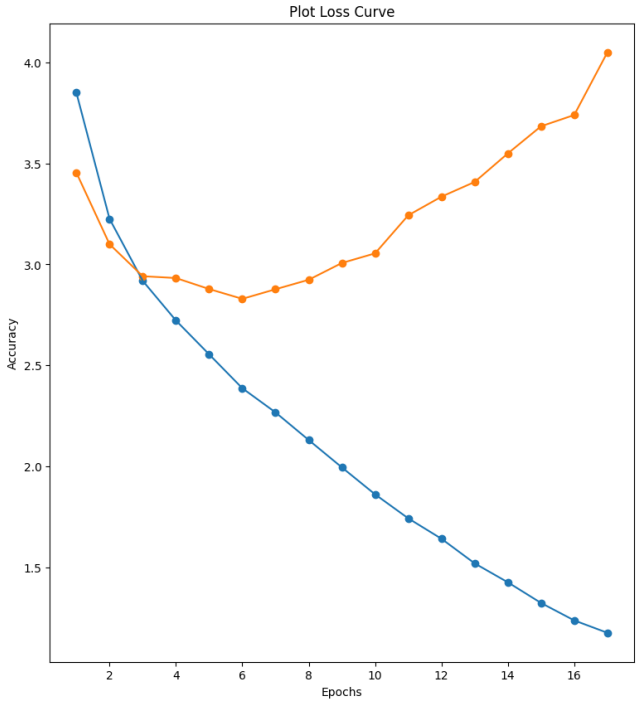
In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
x = pre_processing_v1(inputs)
x = Conv2D(32, (5, 5), input_shape=IMG_SIZE, activation='relu')(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.2)(x)
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
x = Dense(NUM_CLASS, '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'])
```

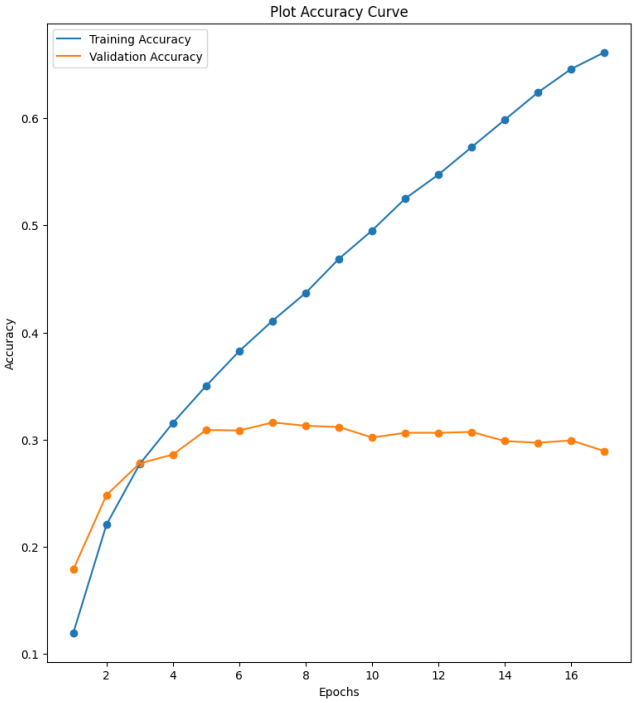
In [ ]:
```
conv2DAugModelHistory = conv2DAugModel.fit(x_train_aug, y_train, epochs=100,
                                           validation_data=(x_val, y_val), batch_size=BATCH_SIZE, c
```

```
Epoch 1/100
625/625 [==============================] - 4s 6ms/step - loss: 3.8514 - accuracy: 0.1198 - va
l_loss: 3.4553 - val_accuracy: 0.1790
Epoch 2/100
625/625 [==============================] - 4s 6ms/step - loss: 3.2258 - accuracy: 0.2209 - va
l_loss: 3.1002 - val_accuracy: 0.2484
Epoch 3/100
625/625 [==============================] - 4s 6ms/step - loss: 2.9186 - accuracy: 0.2772 - va
l_loss: 2.9416 - val_accuracy: 0.2779
Epoch 4/100
625/625 [==============================] - 3s 6ms/step - loss: 2.7223 - accuracy: 0.3156 - va
l_loss: 2.9326 - val_accuracy: 0.2861
Epoch 5/100
625/625 [==============================] - 4s 6ms/step - loss: 2.5545 - accuracy: 0.3501 - va
l_loss: 2.8781 - val_accuracy: 0.3090
Epoch 6/100
625/625 [==============================] - 3s 6ms/step - loss: 2.3866 - accuracy: 0.3826 - va
l_loss: 2.8294 - val_accuracy: 0.3086
Epoch 7/100
625/625 [==============================] - 4s 6ms/step - loss: 2.2668 - accuracy: 0.4108 - va
l_loss: 2.8770 - val_accuracy: 0.3160
Epoch 8/100
625/625 [==============================] - 4s 6ms/step - loss: 2.1300 - accuracy: 0.4369 - va
l_loss: 2.9240 - val_accuracy: 0.3130
Epoch 9/100
625/625 [==============================] - 3s 6ms/step - loss: 1.9943 - accuracy: 0.4686 - va
l_loss: 3.0072 - val_accuracy: 0.3118
Epoch 10/100
625/625 [==============================] - 4s 6ms/step - loss: 1.8608 - accuracy: 0.4951 - va
l_loss: 3.0548 - val_accuracy: 0.3020
Epoch 11/100
625/625 [==============================] - 4s 6ms/step - loss: 1.7418 - accuracy: 0.5250 - va
l_loss: 3.2434 - val_accuracy: 0.3064
Epoch 12/100
625/625 [==============================] - 4s 6ms/step - loss: 1.6407 - accuracy: 0.5472 - va
l_loss: 3.3357 - val_accuracy: 0.3064
Epoch 13/100
625/625 [==============================] - 3s 6ms/step - loss: 1.5188 - accuracy: 0.5729 - va
l_loss: 3.4086 - val_accuracy: 0.3072
Epoch 14/100
625/625 [==============================] - 4s 6ms/step - loss: 1.4250 - accuracy: 0.5985 - va
l_loss: 3.5500 - val_accuracy: 0.2987
Epoch 15/100
625/625 [==============================] - 4s 6ms/step - loss: 1.3226 - accuracy: 0.6240 - va
l_loss: 3.6846 - val_accuracy: 0.2971
Epoch 16/100
625/625 [==============================] - 3s 6ms/step - loss: 1.2361 - accuracy: 0.6458 - va
l_loss: 3.7404 - val_accuracy: 0.2993
Epoch 17/100
625/625 [==============================] - 4s 6ms/step - loss: 1.1747 - accuracy: 0.6613 - va
l_loss: 4.0510 - val_accuracy: 0.2893
```

In [ ]:
```
print(storeResult(conv2DAugModelHistory))
plot_loss_curve(conv2DAugModelHistory)
plt.show()
```

```
{'Model Name': 'conv2DAug', 'Epochs': 17, 'Batch Size': 64, 'Train Loss': 2.26682710647583,
'Val Loss': 2.876986026763916, 'Train Acc': 0.41084998846054077, 'Val Acc': 0.316000014543533
3, '[Train - Val] Acc': 0.09484997391700745}
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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 training accuracy decreased, the validation accuracy increased. This suggest that using the data augmentation method it decreased overfitting as well as improved the overall model.

Training conv2D model with CutMix 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)
conv2DCutMixModel = Model(inputs=inputs, outputs=x, name="conv2DCutMix")
conv2DCutMixModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                    loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
conv2DCutMixModelHistory = conv2DCutMixModel.fit(train_ds_cutmix, epochs=100,
                                    validation_data=val_ds, batch_size=BATCH_SIZE, callbacks
```

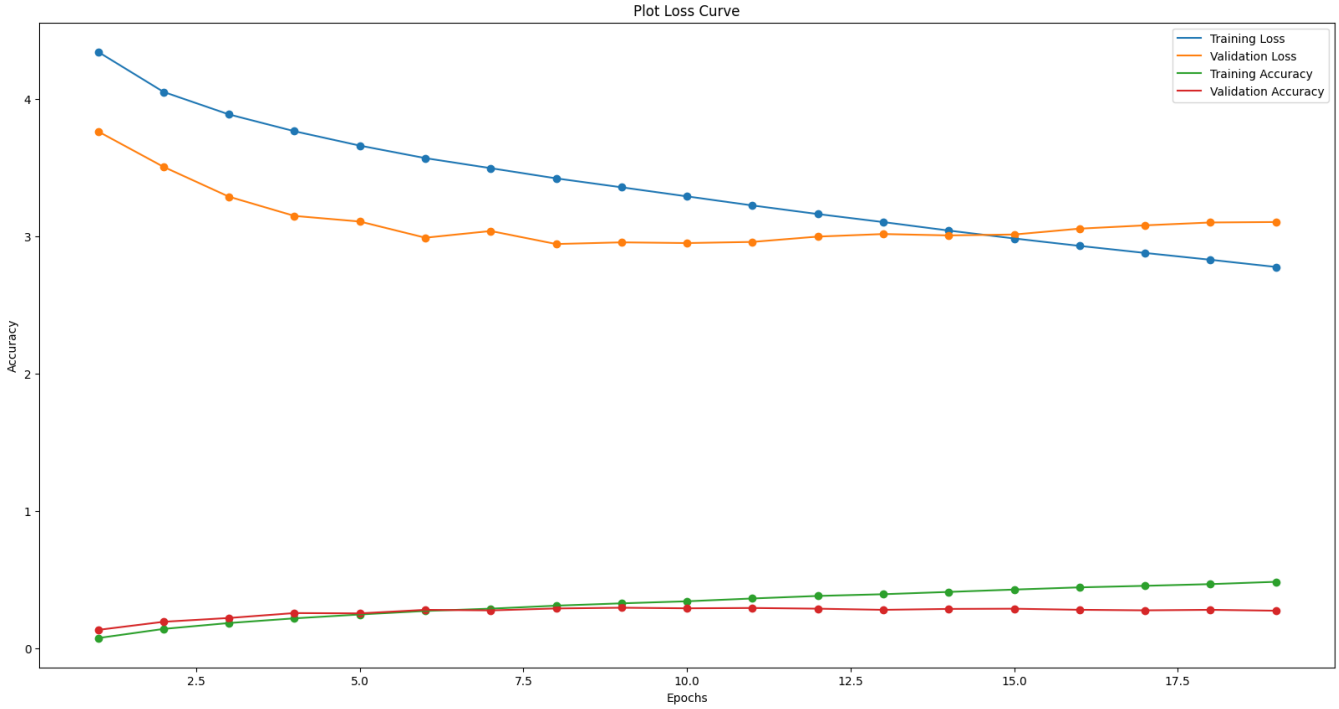
```
Epoch 1/100
625/625 [==============================] - 6s 7ms/step - loss: 4.3417 - accuracy: 0.0758 - va
l_loss: 3.7627 - val_accuracy: 0.1353
Epoch 2/100
625/625 [==============================] - 4s 6ms/step - loss: 4.0502 - accuracy: 0.1424 - va
l_loss: 3.5050 - val_accuracy: 0.1943
Epoch 3/100
625/625 [==============================] - 4s 6ms/step - loss: 3.8864 - accuracy: 0.1853 - va
l_loss: 3.2877 - val_accuracy: 0.2220
Epoch 4/100
625/625 [==============================] - 4s 7ms/step - loss: 3.7643 - accuracy: 0.2193 - va
l_loss: 3.1480 - val_accuracy: 0.2578
Epoch 5/100
625/625 [==============================] - 5s 8ms/step - loss: 3.6594 - accuracy: 0.2475 - va
l_loss: 3.1071 - val_accuracy: 0.2555
Epoch 6/100
625/625 [==============================] - 5s 8ms/step - loss: 3.5685 - accuracy: 0.2730 - va
l_loss: 2.9894 - val_accuracy: 0.2812
Epoch 7/100
625/625 [==============================] - 4s 7ms/step - loss: 3.4953 - accuracy: 0.2899 - va
l_loss: 3.0381 - val_accuracy: 0.2772
Epoch 8/100
625/625 [==============================] - 4s 6ms/step - loss: 3.4211 - accuracy: 0.3116 - va
l_loss: 2.9434 - val_accuracy: 0.2912
Epoch 9/100
625/625 [==============================] - 4s 6ms/step - loss: 3.3561 - accuracy: 0.3286 - va
l_loss: 2.9556 - val_accuracy: 0.2968
Epoch 10/100
625/625 [==============================] - 4s 7ms/step - loss: 3.2901 - accuracy: 0.3435 - va
l_loss: 2.9504 - val_accuracy: 0.2924
Epoch 11/100
625/625 [==============================] - 4s 6ms/step - loss: 3.2241 - accuracy: 0.3638 - va
l_loss: 2.9588 - val_accuracy: 0.2948
Epoch 12/100
625/625 [==============================] - 4s 7ms/step - loss: 3.1615 - accuracy: 0.3824 - va
l_loss: 2.9975 - val_accuracy: 0.2896
Epoch 13/100
625/625 [==============================] - 4s 7ms/step - loss: 3.1030 - accuracy: 0.3948 - va
l_loss: 3.0159 - val_accuracy: 0.2814
Epoch 14/100
625/625 [==============================] - 4s 7ms/step - loss: 3.0414 - accuracy: 0.4117 - va
l_loss: 3.0061 - val_accuracy: 0.2879
Epoch 15/100
625/625 [==============================] - 4s 6ms/step - loss: 2.9838 - accuracy: 0.4287 - va
l_loss: 3.0126 - val_accuracy: 0.2894
Epoch 16/100
625/625 [==============================] - 4s 7ms/step - loss: 2.9298 - accuracy: 0.4448 - va
l_loss: 3.0552 - val_accuracy: 0.2817
Epoch 17/100
625/625 [==============================] - 4s 6ms/step - loss: 2.8784 - accuracy: 0.4560 - va
l_loss: 3.0793 - val_accuracy: 0.2775
Epoch 18/100
625/625 [==============================] - 4s 7ms/step - loss: 2.8288 - accuracy: 0.4683 - va
l_loss: 3.1001 - val_accuracy: 0.2814
Epoch 19/100
625/625 [==============================] - 4s 6ms/step - loss: 2.7764 - accuracy: 0.4853 - va
l_loss: 3.1035 - val_accuracy: 0.2748
```

In [ ]:

```
print(storeResult(conv2DCutMixModelHistory))
plot_loss_curve(conv2DCutMixModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\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': 'conv2DCutMix', 'Epochs': 19, 'Batch Size': 64, 'Train Loss': 3.3561222553253174, 'Val Loss': 2.9555504322052, 'Train Acc': 0.3285500109195709, 'Val Acc': 0.29679998755455017, '[Train - Val] Acc': 0.03175002336502075}

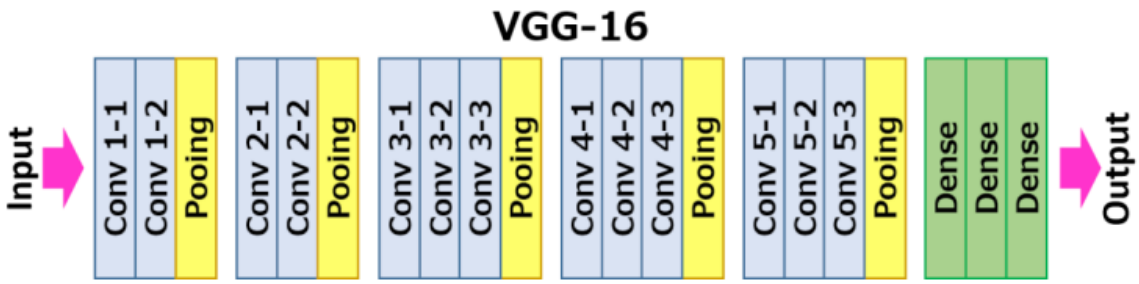


Observations

We note that training loss and training accuracy increased. However, there is no improvement in the validation accuracy and loss. This suggest that the model is overfitted to the training dataset.

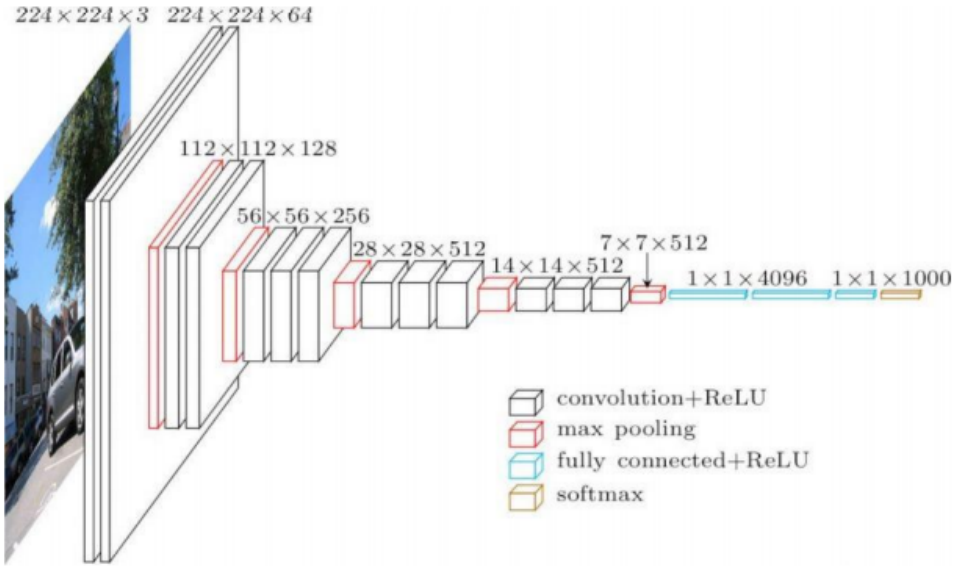
CustomVGG Model

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 [==============================] - 13s 17ms/step - loss: 3.9421 - accuracy: 0.1040 -
val_loss: 3.5052 - val_accuracy: 0.1611
Epoch 2/50
625/625 [==============================] - 10s 16ms/step - loss: 3.2797 - accuracy: 0.2005 -
val_loss: 3.4450 - val_accuracy: 0.1911
Epoch 3/50
625/625 [==============================] - 10s 17ms/step - loss: 2.8478 - accuracy: 0.2772 -
val_loss: 2.7374 - val_accuracy: 0.3035
Epoch 4/50
625/625 [==============================] - 11s 17ms/step - loss: 2.5009 - accuracy: 0.3469 -
val_loss: 2.4229 - val_accuracy: 0.3696
Epoch 5/50
625/625 [==============================] - 11s 18ms/step - loss: 2.2164 - accuracy: 0.4069 -
val_loss: 2.3237 - val_accuracy: 0.3880
Epoch 6/50
625/625 [==============================] - 11s 18ms/step - loss: 1.9888 - accuracy: 0.4580 -
val_loss: 2.1975 - val_accuracy: 0.4202
Epoch 7/50
625/625 [==============================] - 12s 18ms/step - loss: 1.7707 - accuracy: 0.5049 -
val_loss: 2.1444 - val_accuracy: 0.4373
Epoch 8/50
625/625 [==============================] - 12s 19ms/step - loss: 1.5805 - accuracy: 0.5504 -
val_loss: 2.1545 - val_accuracy: 0.4514
Epoch 9/50
625/625 [==============================] - 12s 19ms/step - loss: 1.3922 - accuracy: 0.5965 -
val_loss: 2.2005 - val_accuracy: 0.4477
Epoch 10/50
625/625 [==============================] - 12s 20ms/step - loss: 1.2270 - accuracy: 0.6386 -
val_loss: 2.3193 - val_accuracy: 0.4488
Epoch 11/50
625/625 [==============================] - 13s 21ms/step - loss: 1.0709 - accuracy: 0.6786 -
val_loss: 2.1773 - val_accuracy: 0.4669
Epoch 12/50
625/625 [==============================] - 14s 23ms/step - loss: 0.9199 - accuracy: 0.7199 -
val_loss: 2.3426 - val_accuracy: 0.4556
Epoch 13/50
625/625 [==============================] - 13s 21ms/step - loss: 0.8108 - accuracy: 0.7495 -
val_loss: 2.3902 - val_accuracy: 0.4632
Epoch 14/50
625/625 [==============================] - 15s 24ms/step - loss: 0.7042 - accuracy: 0.7791 -
val_loss: 2.4644 - val_accuracy: 0.4710
Epoch 15/50
625/625 [==============================] - 14s 22ms/step - loss: 0.6097 - accuracy: 0.8072 -
val_loss: 2.6177 - val_accuracy: 0.4553
Epoch 16/50
625/625 [==============================] - 15s 24ms/step - loss: 0.5166 - accuracy: 0.8353 -
val_loss: 2.6118 - val_accuracy: 0.4705
Epoch 17/50
625/625 [==============================] - 15s 24ms/step - loss: 0.4540 - accuracy: 0.8547 -
val_loss: 2.6415 - val_accuracy: 0.4799
Epoch 18/50
625/625 [==============================] - 15s 23ms/step - loss: 0.3806 - accuracy: 0.8770 -
val_loss: 2.9238 - val_accuracy: 0.4584
Epoch 19/50
625/625 [==============================] - 14s 22ms/step - loss: 0.3446 - accuracy: 0.8888 -
val_loss: 2.8548 - val_accuracy: 0.4740
Epoch 20/50
625/625 [==============================] - 17s 27ms/step - loss: 0.2905 - accuracy: 0.9057 -
val_loss: 2.9398 - val_accuracy: 0.4724
Epoch 21/50
625/625 [==============================] - 17s 28ms/step - loss: 0.2439 - accuracy: 0.9213 -
val_loss: 3.0177 - val_accuracy: 0.4805
Epoch 22/50
625/625 [==============================] - 19s 30ms/step - loss: 0.2234 - accuracy: 0.9276 -
val_loss: 3.0808 - val_accuracy: 0.4821
```

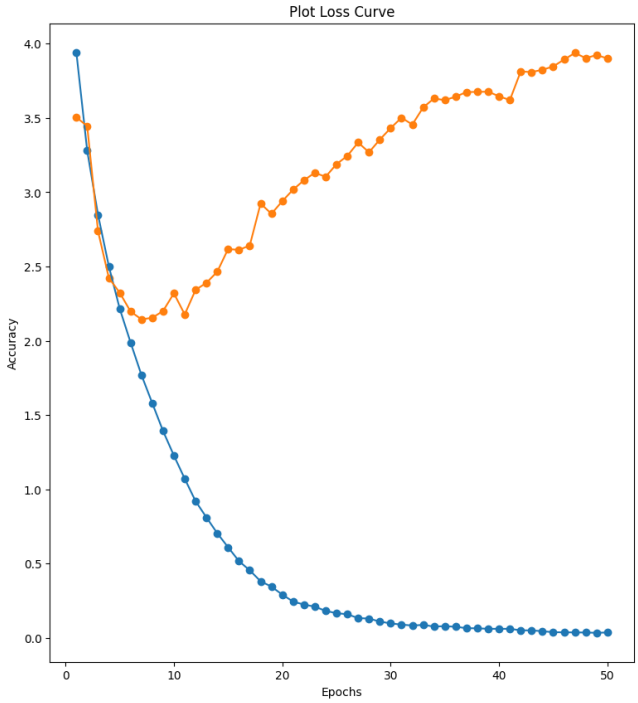
```
Epoch 23/50
625/625 [==============================] - 18s 29ms/step - loss: 0.2099 - accuracy: 0.9316 -
val_loss: 3.1311 - val_accuracy: 0.4801
Epoch 24/50
625/625 [==============================] - 17s 27ms/step - loss: 0.1810 - accuracy: 0.9407 -
val_loss: 3.1042 - val_accuracy: 0.4887
Epoch 25/50
625/625 [==============================] - 15s 24ms/step - loss: 0.1658 - accuracy: 0.9471 -
val_loss: 3.1890 - val_accuracy: 0.4873
Epoch 26/50
625/625 [==============================] - 16s 25ms/step - loss: 0.1584 - accuracy: 0.9495 -
val_loss: 3.2447 - val_accuracy: 0.4864
Epoch 27/50
625/625 [==============================] - 16s 25ms/step - loss: 0.1326 - accuracy: 0.9586 -
val_loss: 3.3351 - val_accuracy: 0.4902
Epoch 28/50
625/625 [==============================] - 18s 28ms/step - loss: 0.1294 - accuracy: 0.9592 -
val_loss: 3.2690 - val_accuracy: 0.4915
Epoch 29/50
625/625 [==============================] - 14s 23ms/step - loss: 0.1090 - accuracy: 0.9649 -
val_loss: 3.3545 - val_accuracy: 0.4944
Epoch 30/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0970 - accuracy: 0.9682 -
val_loss: 3.4322 - val_accuracy: 0.4862
Epoch 31/50
625/625 [==============================] - 17s 27ms/step - loss: 0.0886 - accuracy: 0.9711 -
val_loss: 3.5019 - val_accuracy: 0.4959
Epoch 32/50
625/625 [==============================] - 16s 26ms/step - loss: 0.0827 - accuracy: 0.9730 -
val_loss: 3.4559 - val_accuracy: 0.4963
Epoch 33/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0866 - accuracy: 0.9722 -
val_loss: 3.5709 - val_accuracy: 0.4897
Epoch 34/50
625/625 [==============================] - 14s 22ms/step - loss: 0.0759 - accuracy: 0.9757 -
val_loss: 3.6316 - val_accuracy: 0.4943
Epoch 35/50
625/625 [==============================] - 16s 26ms/step - loss: 0.0782 - accuracy: 0.9750 -
val_loss: 3.6200 - val_accuracy: 0.4964
Epoch 36/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0742 - accuracy: 0.9762 -
val_loss: 3.6438 - val_accuracy: 0.4968
Epoch 37/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0647 - accuracy: 0.9791 -
val_loss: 3.6746 - val_accuracy: 0.4969
Epoch 38/50
625/625 [==============================] - 16s 26ms/step - loss: 0.0638 - accuracy: 0.9798 -
val_loss: 3.6760 - val_accuracy: 0.4902
Epoch 39/50
625/625 [==============================] - 18s 29ms/step - loss: 0.0605 - accuracy: 0.9804 -
val_loss: 3.6771 - val_accuracy: 0.4970
Epoch 40/50
625/625 [==============================] - 16s 26ms/step - loss: 0.0605 - accuracy: 0.9798 -
val_loss: 3.6464 - val_accuracy: 0.4949
Epoch 41/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0604 - accuracy: 0.9804 -
val_loss: 3.6205 - val_accuracy: 0.4984
Epoch 42/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0502 - accuracy: 0.9842 -
val_loss: 3.8133 - val_accuracy: 0.4903
Epoch 43/50
625/625 [==============================] - 17s 27ms/step - loss: 0.0498 - accuracy: 0.9837 -
val_loss: 3.8093 - val_accuracy: 0.4980
Epoch 44/50
625/625 [==============================] - 17s 28ms/step - loss: 0.0443 - accuracy: 0.9861 -
val_loss: 3.8244 - val_accuracy: 0.4916
```

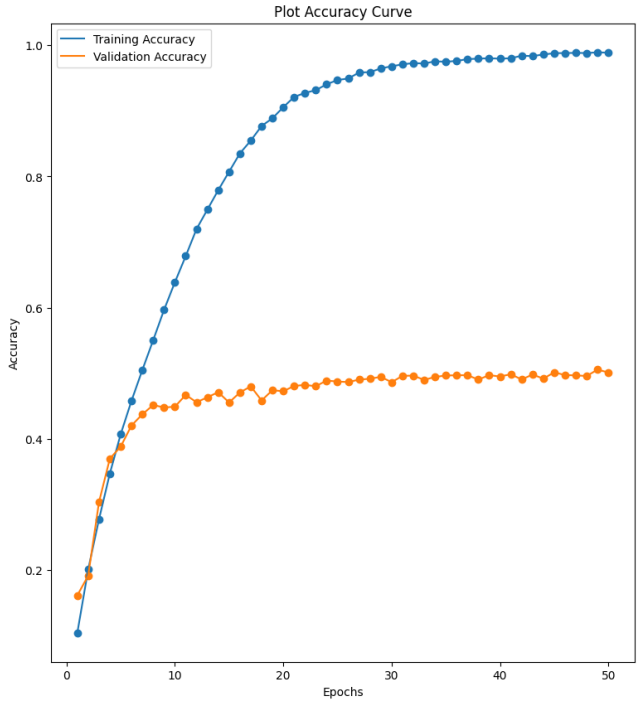
```
Epoch 45/50
625/625 [==============================] - 20s 32ms/step - loss: 0.0377 - accuracy: 0.9881 -
val_loss: 3.8472 - val_accuracy: 0.5013
Epoch 46/50
625/625 [==============================] - 16s 26ms/step - loss: 0.0366 - accuracy: 0.9883 -
val_loss: 3.8933 - val_accuracy: 0.4971
Epoch 47/50
625/625 [==============================] - 17s 27ms/step - loss: 0.0362 - accuracy: 0.9885 -
val_loss: 3.9379 - val_accuracy: 0.4966
Epoch 48/50
625/625 [==============================] - 17s 28ms/step - loss: 0.0353 - accuracy: 0.9883 -
val_loss: 3.9032 - val_accuracy: 0.4957
Epoch 49/50
625/625 [==============================] - 18s 28ms/step - loss: 0.0336 - accuracy: 0.9894 -
val_loss: 3.9238 - val_accuracy: 0.5054
Epoch 50/50
625/625 [==============================] - 17s 28ms/step - loss: 0.0355 - accuracy: 0.9888 -
val_loss: 3.9013 - val_accuracy: 0.5011
```

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

```
{'Model Name': 'CustomVGG', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 0.0335755087435245
5, 'Val Loss': 3.9237570762634277, 'Train Acc': 0.9894000291824341, 'Val Acc': 0.505400002002
7161, '[Train - Val] Acc': 0.484000027179718}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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 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
```

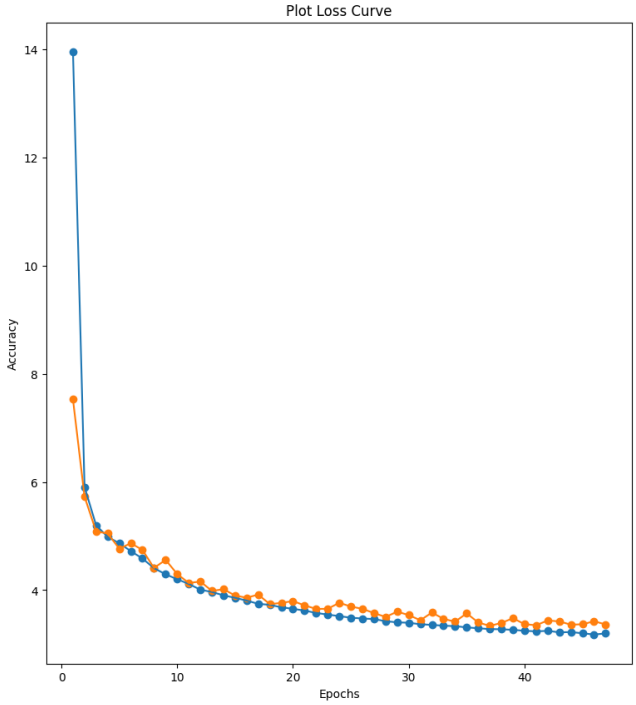
```
Epoch 1/50
625/625 [==============================] - 20s 27ms/step - loss: 13.9563 - accuracy: 0.0861 -
val_loss: 7.5263 - val_accuracy: 0.0546
Epoch 2/50
625/625 [==============================] - 15s 24ms/step - loss: 5.9061 - accuracy: 0.1020 -
val_loss: 5.7275 - val_accuracy: 0.0663
Epoch 3/50
625/625 [==============================] - 15s 24ms/step - loss: 5.1977 - accuracy: 0.1112 -
val_loss: 5.0844 - val_accuracy: 0.1214
Epoch 4/50
625/625 [==============================] - 15s 24ms/step - loss: 4.9844 - accuracy: 0.1307 -
val_loss: 5.0463 - val_accuracy: 0.1240
Epoch 5/50
625/625 [==============================] - 15s 24ms/step - loss: 4.8669 - accuracy: 0.1521 -
val_loss: 4.7668 - val_accuracy: 0.1657
Epoch 6/50
625/625 [==============================] - 16s 25ms/step - loss: 4.7200 - accuracy: 0.1723 -
val_loss: 4.8663 - val_accuracy: 0.1647
Epoch 7/50
625/625 [==============================] - 15s 24ms/step - loss: 4.5843 - accuracy: 0.1993 -
val_loss: 4.7428 - val_accuracy: 0.1783
Epoch 8/50
625/625 [==============================] - 15s 24ms/step - loss: 4.4057 - accuracy: 0.2265 -
val_loss: 4.4095 - val_accuracy: 0.2355
Epoch 9/50
625/625 [==============================] - 15s 24ms/step - loss: 4.2924 - accuracy: 0.2497 -
val_loss: 4.5588 - val_accuracy: 0.2212
Epoch 10/50
625/625 [==============================] - 15s 23ms/step - loss: 4.2012 - accuracy: 0.2718 -
val_loss: 4.2985 - val_accuracy: 0.2599
Epoch 11/50
625/625 [==============================] - 16s 25ms/step - loss: 4.1142 - accuracy: 0.2857 -
val_loss: 4.1316 - val_accuracy: 0.2855
Epoch 12/50
625/625 [==============================] - 16s 25ms/step - loss: 4.0075 - accuracy: 0.3034 -
val_loss: 4.1587 - val_accuracy: 0.2817
Epoch 13/50
625/625 [==============================] - 16s 25ms/step - loss: 3.9648 - accuracy: 0.3167 -
val_loss: 3.9853 - val_accuracy: 0.3185
Epoch 14/50
625/625 [==============================] - 16s 26ms/step - loss: 3.9035 - accuracy: 0.3280 -
val_loss: 4.0164 - val_accuracy: 0.3182
Epoch 15/50
625/625 [==============================] - 16s 25ms/step - loss: 3.8573 - accuracy: 0.3361 -
val_loss: 3.8994 - val_accuracy: 0.3282
Epoch 16/50
625/625 [==============================] - 15s 24ms/step - loss: 3.8060 - accuracy: 0.3449 -
val_loss: 3.8589 - val_accuracy: 0.3355
Epoch 17/50
625/625 [==============================] - 15s 24ms/step - loss: 3.7462 - accuracy: 0.3548 -
val_loss: 3.9171 - val_accuracy: 0.3334
Epoch 18/50
625/625 [==============================] - 16s 26ms/step - loss: 3.7281 - accuracy: 0.3612 -
val_loss: 3.7406 - val_accuracy: 0.3614
Epoch 19/50
625/625 [==============================] - 16s 25ms/step - loss: 3.6819 - accuracy: 0.3673 -
val_loss: 3.7636 - val_accuracy: 0.3530
Epoch 20/50
625/625 [==============================] - 15s 24ms/step - loss: 3.6544 - accuracy: 0.3744 -
val_loss: 3.7976 - val_accuracy: 0.3542
Epoch 21/50
625/625 [==============================] - 15s 24ms/step - loss: 3.6212 - accuracy: 0.3785 -
val_loss: 3.7172 - val_accuracy: 0.3652
Epoch 22/50
625/625 [==============================] - 15s 25ms/step - loss: 3.5782 - accuracy: 0.3852 -
val_loss: 3.6525 - val_accuracy: 0.3802
```

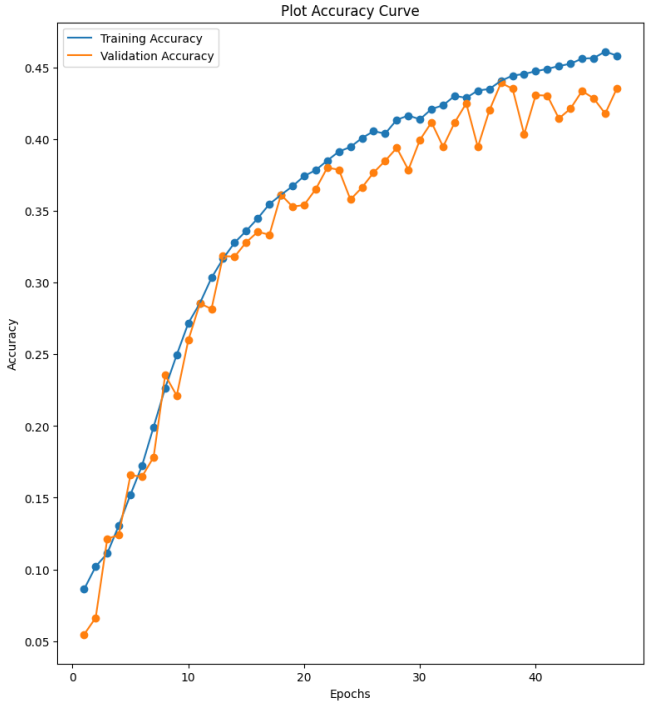
```
Epoch 23/50
625/625 [==============================] - 15s 24ms/step - loss: 3.5473 - accuracy: 0.3914 -
val_loss: 3.6544 - val_accuracy: 0.3788
Epoch 24/50
625/625 [==============================] - 15s 24ms/step - loss: 3.5189 - accuracy: 0.3946 -
val_loss: 3.7662 - val_accuracy: 0.3580
Epoch 25/50
625/625 [==============================] - 15s 24ms/step - loss: 3.4897 - accuracy: 0.4009 -
val_loss: 3.6953 - val_accuracy: 0.3664
Epoch 26/50
625/625 [==============================] - 15s 24ms/step - loss: 3.4753 - accuracy: 0.4056 -
val_loss: 3.6540 - val_accuracy: 0.3768
Epoch 27/50
625/625 [==============================] - 15s 25ms/step - loss: 3.4645 - accuracy: 0.4040 -
val_loss: 3.5748 - val_accuracy: 0.3850
Epoch 28/50
625/625 [==============================] - 16s 26ms/step - loss: 3.4254 - accuracy: 0.4136 -
val_loss: 3.5038 - val_accuracy: 0.3940
Epoch 29/50
625/625 [==============================] - 16s 26ms/step - loss: 3.4039 - accuracy: 0.4166 -
val_loss: 3.6020 - val_accuracy: 0.3787
Epoch 30/50
625/625 [==============================] - 18s 28ms/step - loss: 3.3965 - accuracy: 0.4139 -
val_loss: 3.5375 - val_accuracy: 0.3996
Epoch 31/50
625/625 [==============================] - 19s 31ms/step - loss: 3.3686 - accuracy: 0.4211 -
val_loss: 3.4430 - val_accuracy: 0.4118
Epoch 32/50
625/625 [==============================] - 21s 33ms/step - loss: 3.3568 - accuracy: 0.4238 -
val_loss: 3.5854 - val_accuracy: 0.3948
Epoch 33/50
625/625 [==============================] - 20s 32ms/step - loss: 3.3421 - accuracy: 0.4303 -
val_loss: 3.4681 - val_accuracy: 0.4117
Epoch 34/50
625/625 [==============================] - 17s 27ms/step - loss: 3.3339 - accuracy: 0.4289 -
val_loss: 3.4163 - val_accuracy: 0.4251
Epoch 35/50
625/625 [==============================] - 18s 29ms/step - loss: 3.3105 - accuracy: 0.4340 -
val_loss: 3.5710 - val_accuracy: 0.3946
Epoch 36/50
625/625 [==============================] - 19s 30ms/step - loss: 3.2941 - accuracy: 0.4351 -
val_loss: 3.4006 - val_accuracy: 0.4202
Epoch 37/50
625/625 [==============================] - 21s 34ms/step - loss: 3.2800 - accuracy: 0.4408 -
val_loss: 3.3387 - val_accuracy: 0.4393
Epoch 38/50
625/625 [==============================] - 18s 29ms/step - loss: 3.2756 - accuracy: 0.4444 -
val_loss: 3.3935 - val_accuracy: 0.4353
Epoch 39/50
625/625 [==============================] - 18s 29ms/step - loss: 3.2656 - accuracy: 0.4456 -
val_loss: 3.4809 - val_accuracy: 0.4035
Epoch 40/50
625/625 [==============================] - 22s 35ms/step - loss: 3.2478 - accuracy: 0.4475 -
val_loss: 3.3696 - val_accuracy: 0.4308
Epoch 41/50
625/625 [==============================] - 17s 28ms/step - loss: 3.2378 - accuracy: 0.4491 -
val_loss: 3.3509 - val_accuracy: 0.4305
Epoch 42/50
625/625 [==============================] - 20s 33ms/step - loss: 3.2469 - accuracy: 0.4511 -
val_loss: 3.4389 - val_accuracy: 0.4145
Epoch 43/50
625/625 [==============================] - 20s 33ms/step - loss: 3.2196 - accuracy: 0.4527 -
val_loss: 3.4212 - val_accuracy: 0.4214
Epoch 44/50
625/625 [==============================] - 21s 33ms/step - loss: 3.2247 - accuracy: 0.4563 -
val_loss: 3.3607 - val_accuracy: 0.4337
```

```
Epoch 45/50
625/625 [==============================] - 19s 30ms/step - loss: 3.2015 - accuracy: 0.4566 -
val_loss: 3.3683 - val_accuracy: 0.4286
Epoch 46/50
625/625 [==============================] - 19s 30ms/step - loss: 3.1795 - accuracy: 0.4612 -
val_loss: 3.4237 - val_accuracy: 0.4179
Epoch 47/50
625/625 [==============================] - 21s 33ms/step - loss: 3.1987 - accuracy: 0.4584 -
val_loss: 3.3656 - val_accuracy: 0.4354
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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': 47, 'Batch Size': 64, 'Train Loss': 3.28001523017883
3, 'Val Loss': 3.3387374877929688, 'Train Acc': 0.4408000111579895, 'Val Acc': 0.439300000667
572, '[Train - Val] Acc': 0.0015000104904174805}
```





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'])
```

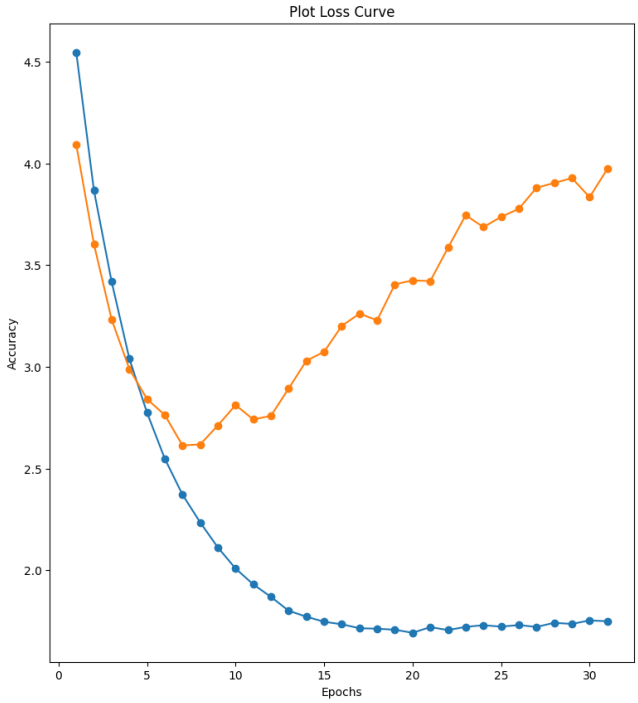
In [ ]:
```
customVGGL2ModelHistory = customVGGL2Model.fit(x_train, y_train, epochs=50,
                                               validation_data=(x_val, y_val), batch_size=BAT
```

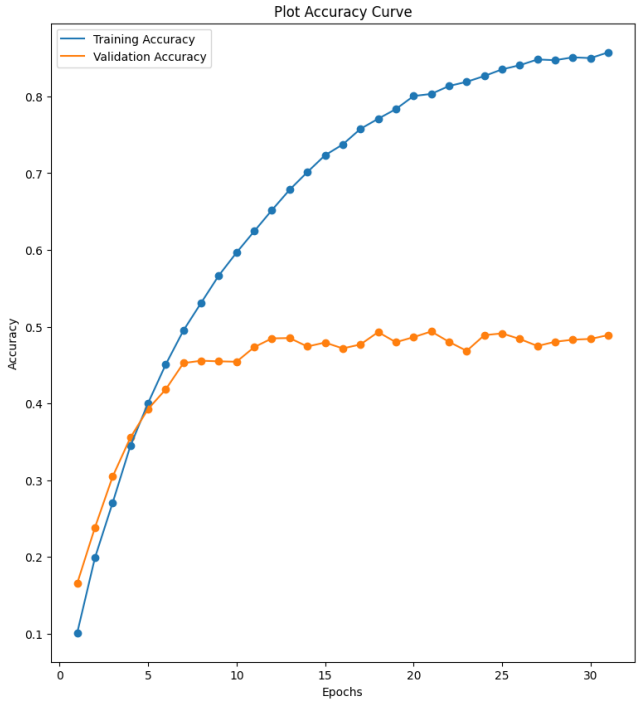
```
Epoch 1/50
625/625 [==============================] - 24s 33ms/step - loss: 4.5456 - accuracy: 0.1012 -
val_loss: 4.0920 - val_accuracy: 0.1657
Epoch 2/50
625/625 [==============================] - 16s 26ms/step - loss: 3.8665 - accuracy: 0.1989 -
val_loss: 3.6026 - val_accuracy: 0.2383
Epoch 3/50
625/625 [==============================] - 17s 27ms/step - loss: 3.4179 - accuracy: 0.2706 -
val_loss: 3.2325 - val_accuracy: 0.3047
Epoch 4/50
625/625 [==============================] - 17s 27ms/step - loss: 3.0401 - accuracy: 0.3446 -
val_loss: 2.9892 - val_accuracy: 0.3554
Epoch 5/50
625/625 [==============================] - 18s 29ms/step - loss: 2.7742 - accuracy: 0.4004 -
val_loss: 2.8413 - val_accuracy: 0.3927
Epoch 6/50
625/625 [==============================] - 17s 27ms/step - loss: 2.5488 - accuracy: 0.4509 -
val_loss: 2.7633 - val_accuracy: 0.4184
Epoch 7/50
625/625 [==============================] - 17s 27ms/step - loss: 2.3720 - accuracy: 0.4951 -
val_loss: 2.6139 - val_accuracy: 0.4526
Epoch 8/50
625/625 [==============================] - 17s 28ms/step - loss: 2.2356 - accuracy: 0.5311 -
val_loss: 2.6186 - val_accuracy: 0.4557
Epoch 9/50
625/625 [==============================] - 18s 29ms/step - loss: 2.1122 - accuracy: 0.5668 -
val_loss: 2.7119 - val_accuracy: 0.4549
Epoch 10/50
625/625 [==============================] - 17s 28ms/step - loss: 2.0086 - accuracy: 0.5966 -
val_loss: 2.8122 - val_accuracy: 0.4544
Epoch 11/50
625/625 [==============================] - 18s 28ms/step - loss: 1.9308 - accuracy: 0.6244 -
val_loss: 2.7412 - val_accuracy: 0.4734
Epoch 12/50
625/625 [==============================] - 17s 27ms/step - loss: 1.8694 - accuracy: 0.6520 -
val_loss: 2.7594 - val_accuracy: 0.4848
Epoch 13/50
625/625 [==============================] - 17s 27ms/step - loss: 1.8010 - accuracy: 0.6785 -
val_loss: 2.8933 - val_accuracy: 0.4852
Epoch 14/50
625/625 [==============================] - 19s 30ms/step - loss: 1.7720 - accuracy: 0.7014 -
val_loss: 3.0298 - val_accuracy: 0.4742
Epoch 15/50
625/625 [==============================] - 18s 29ms/step - loss: 1.7470 - accuracy: 0.7234 -
val_loss: 3.0736 - val_accuracy: 0.4794
Epoch 16/50
625/625 [==============================] - 19s 30ms/step - loss: 1.7347 - accuracy: 0.7375 -
val_loss: 3.2012 - val_accuracy: 0.4717
Epoch 17/50
625/625 [==============================] - 21s 34ms/step - loss: 1.7151 - accuracy: 0.7577 -
val_loss: 3.2612 - val_accuracy: 0.4769
Epoch 18/50
625/625 [==============================] - 19s 31ms/step - loss: 1.7130 - accuracy: 0.7709 -
val_loss: 3.2287 - val_accuracy: 0.4930
Epoch 19/50
625/625 [==============================] - 20s 33ms/step - loss: 1.7078 - accuracy: 0.7835 -
val_loss: 3.4053 - val_accuracy: 0.4798
Epoch 20/50
625/625 [==============================] - 18s 28ms/step - loss: 1.6928 - accuracy: 0.8005 -
val_loss: 3.4248 - val_accuracy: 0.4864
Epoch 21/50
625/625 [==============================] - 20s 32ms/step - loss: 1.7208 - accuracy: 0.8033 -
val_loss: 3.4212 - val_accuracy: 0.4938
Epoch 22/50
625/625 [==============================] - 17s 27ms/step - loss: 1.7062 - accuracy: 0.8136 -
val_loss: 3.5851 - val_accuracy: 0.4803
```

```
Epoch 23/50
625/625 [==============================] - 21s 33ms/step - loss: 1.7220 - accuracy: 0.8191 -
val_loss: 3.7451 - val_accuracy: 0.4685
Epoch 24/50
625/625 [==============================] - 15s 23ms/step - loss: 1.7301 - accuracy: 0.8266 -
val_loss: 3.6869 - val_accuracy: 0.4890
Epoch 25/50
625/625 [==============================] - 18s 29ms/step - loss: 1.7231 - accuracy: 0.8353 -
val_loss: 3.7367 - val_accuracy: 0.4913
Epoch 26/50
625/625 [==============================] - 18s 29ms/step - loss: 1.7308 - accuracy: 0.8407 -
val_loss: 3.7764 - val_accuracy: 0.4840
Epoch 27/50
625/625 [==============================] - 18s 29ms/step - loss: 1.7215 - accuracy: 0.8482 -
val_loss: 3.8798 - val_accuracy: 0.4749
Epoch 28/50
625/625 [==============================] - 15s 25ms/step - loss: 1.7419 - accuracy: 0.8472 -
val_loss: 3.9035 - val_accuracy: 0.4804
Epoch 29/50
625/625 [==============================] - 15s 24ms/step - loss: 1.7363 - accuracy: 0.8510 -
val_loss: 3.9277 - val_accuracy: 0.4831
Epoch 30/50
625/625 [==============================] - 14s 23ms/step - loss: 1.7537 - accuracy: 0.8500 -
val_loss: 3.8338 - val_accuracy: 0.4841
Epoch 31/50
625/625 [==============================] - 15s 25ms/step - loss: 1.7495 - accuracy: 0.8575 -
val_loss: 3.9721 - val_accuracy: 0.4891
```

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

```
{'Model Name': 'CustomVGG_L2', 'Epochs': 31, 'Batch Size': 64, 'Train Loss': 1.7207714319229126, 'Val Loss': 3.4211928444451904, 'Train Acc': 0.8033000230789185, 'Val Acc': 0.4938000142574310 3, '[Train - Val] Acc': 0.3095000088214874}
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\3161967465.py:14: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```





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
```

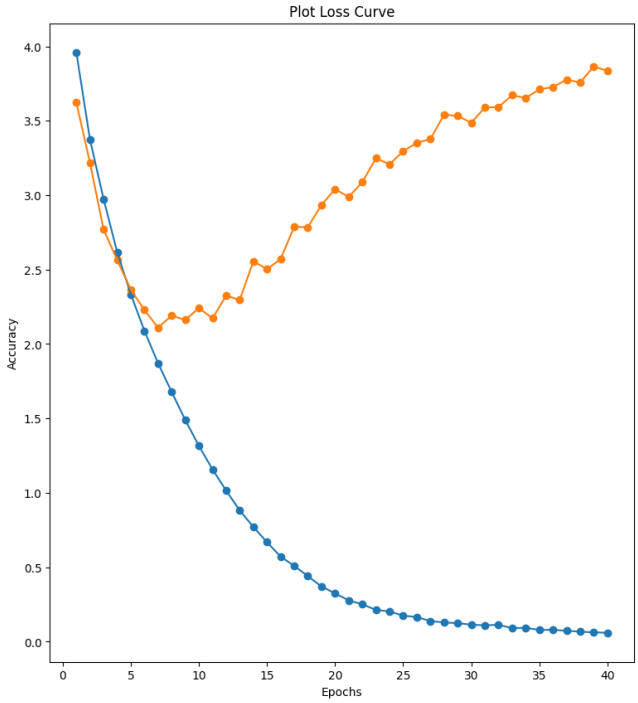
```
Epoch 1/50
625/625 [==============================] - 18s 25ms/step - loss: 3.9599 - accuracy: 0.0990 -
val_loss: 3.6223 - val_accuracy: 0.1408
Epoch 2/50
625/625 [==============================] - 12s 20ms/step - loss: 3.3739 - accuracy: 0.1860 -
val_loss: 3.2180 - val_accuracy: 0.2093
Epoch 3/50
625/625 [==============================] - 14s 22ms/step - loss: 2.9742 - accuracy: 0.2557 -
val_loss: 2.7690 - val_accuracy: 0.2888
Epoch 4/50
625/625 [==============================] - 16s 26ms/step - loss: 2.6168 - accuracy: 0.3237 -
val_loss: 2.5638 - val_accuracy: 0.3382
Epoch 5/50
625/625 [==============================] - 15s 24ms/step - loss: 2.3341 - accuracy: 0.3822 -
val_loss: 2.3654 - val_accuracy: 0.3847
Epoch 6/50
625/625 [==============================] - 13s 20ms/step - loss: 2.0888 - accuracy: 0.4358 -
val_loss: 2.2284 - val_accuracy: 0.4127
Epoch 7/50
625/625 [==============================] - 12s 19ms/step - loss: 1.8707 - accuracy: 0.4835 -
val_loss: 2.1099 - val_accuracy: 0.4452
Epoch 8/50
625/625 [==============================] - 14s 22ms/step - loss: 1.6778 - accuracy: 0.5278 -
val_loss: 2.1909 - val_accuracy: 0.4372
Epoch 9/50
625/625 [==============================] - 15s 24ms/step - loss: 1.4894 - accuracy: 0.5738 -
val_loss: 2.1621 - val_accuracy: 0.4516
Epoch 10/50
625/625 [==============================] - 14s 22ms/step - loss: 1.3152 - accuracy: 0.6162 -
val_loss: 2.2421 - val_accuracy: 0.4472
Epoch 11/50
625/625 [==============================] - 14s 22ms/step - loss: 1.1576 - accuracy: 0.6561 -
val_loss: 2.1727 - val_accuracy: 0.4613
Epoch 12/50
625/625 [==============================] - 13s 20ms/step - loss: 1.0151 - accuracy: 0.6943 -
val_loss: 2.3249 - val_accuracy: 0.4603
Epoch 13/50
625/625 [==============================] - 13s 21ms/step - loss: 0.8818 - accuracy: 0.7269 -
val_loss: 2.2961 - val_accuracy: 0.4671
Epoch 14/50
625/625 [==============================] - 14s 22ms/step - loss: 0.7711 - accuracy: 0.7604 -
val_loss: 2.5564 - val_accuracy: 0.4592
Epoch 15/50
625/625 [==============================] - 13s 21ms/step - loss: 0.6687 - accuracy: 0.7898 -
val_loss: 2.5025 - val_accuracy: 0.4660
Epoch 16/50
625/625 [==============================] - 13s 21ms/step - loss: 0.5716 - accuracy: 0.8177 -
val_loss: 2.5711 - val_accuracy: 0.4745
Epoch 17/50
625/625 [==============================] - 13s 20ms/step - loss: 0.5092 - accuracy: 0.8371 -
val_loss: 2.7883 - val_accuracy: 0.4692
Epoch 18/50
625/625 [==============================] - 13s 21ms/step - loss: 0.4406 - accuracy: 0.8590 -
val_loss: 2.7852 - val_accuracy: 0.4665
Epoch 19/50
625/625 [==============================] - 14s 22ms/step - loss: 0.3704 - accuracy: 0.8795 -
val_loss: 2.9345 - val_accuracy: 0.4645
Epoch 20/50
625/625 [==============================] - 13s 21ms/step - loss: 0.3250 - accuracy: 0.8943 -
val_loss: 3.0405 - val_accuracy: 0.4702
Epoch 21/50
625/625 [==============================] - 16s 25ms/step - loss: 0.2772 - accuracy: 0.9108 -
val_loss: 2.9884 - val_accuracy: 0.4781
Epoch 22/50
625/625 [==============================] - 15s 25ms/step - loss: 0.2514 - accuracy: 0.9190 -
val_loss: 3.0877 - val_accuracy: 0.4732
```

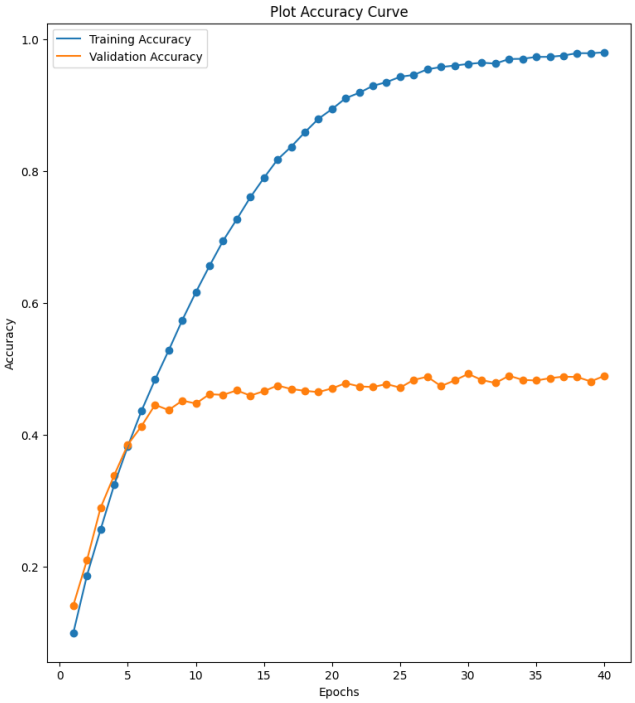
```
Epoch 23/50
625/625 [==============================] - 18s 29ms/step - loss: 0.2135 - accuracy: 0.9296 -
val_loss: 3.2484 - val_accuracy: 0.4722
Epoch 24/50
625/625 [==============================] - 17s 27ms/step - loss: 0.2034 - accuracy: 0.9352 -
val_loss: 3.2069 - val_accuracy: 0.4767
Epoch 25/50
625/625 [==============================] - 60s 96ms/step - loss: 0.1745 - accuracy: 0.9433 -
val_loss: 3.2960 - val_accuracy: 0.4713
Epoch 26/50
625/625 [==============================] - 61s 98ms/step - loss: 0.1663 - accuracy: 0.9462 -
val_loss: 3.3514 - val_accuracy: 0.4831
Epoch 27/50
625/625 [==============================] - 14s 22ms/step - loss: 0.1380 - accuracy: 0.9547 -
val_loss: 3.3772 - val_accuracy: 0.4878
Epoch 28/50
625/625 [==============================] - 12s 20ms/step - loss: 0.1293 - accuracy: 0.9581 -
val_loss: 3.5411 - val_accuracy: 0.4736
Epoch 29/50
625/625 [==============================] - 19s 30ms/step - loss: 0.1246 - accuracy: 0.9603 -
val_loss: 3.5329 - val_accuracy: 0.4822
Epoch 30/50
625/625 [==============================] - 41s 66ms/step - loss: 0.1145 - accuracy: 0.9628 -
val_loss: 3.4867 - val_accuracy: 0.4922
Epoch 31/50
625/625 [==============================] - 14s 22ms/step - loss: 0.1098 - accuracy: 0.9647 -
val_loss: 3.5894 - val_accuracy: 0.4828
Epoch 32/50
625/625 [==============================] - 13s 22ms/step - loss: 0.1133 - accuracy: 0.9634 -
val_loss: 3.5921 - val_accuracy: 0.4784
Epoch 33/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0912 - accuracy: 0.9703 -
val_loss: 3.6716 - val_accuracy: 0.4893
Epoch 34/50
625/625 [==============================] - 16s 26ms/step - loss: 0.0932 - accuracy: 0.9706 -
val_loss: 3.6539 - val_accuracy: 0.4831
Epoch 35/50
625/625 [==============================] - 14s 22ms/step - loss: 0.0791 - accuracy: 0.9735 -
val_loss: 3.7109 - val_accuracy: 0.4821
Epoch 36/50
625/625 [==============================] - 15s 23ms/step - loss: 0.0795 - accuracy: 0.9736 -
val_loss: 3.7268 - val_accuracy: 0.4857
Epoch 37/50
625/625 [==============================] - 14s 23ms/step - loss: 0.0733 - accuracy: 0.9756 -
val_loss: 3.7758 - val_accuracy: 0.4880
Epoch 38/50
625/625 [==============================] - 15s 24ms/step - loss: 0.0671 - accuracy: 0.9792 -
val_loss: 3.7580 - val_accuracy: 0.4874
Epoch 39/50
625/625 [==============================] - 15s 23ms/step - loss: 0.0627 - accuracy: 0.9791 -
val_loss: 3.8651 - val_accuracy: 0.4807
Epoch 40/50
625/625 [==============================] - 16s 25ms/step - loss: 0.0597 - accuracy: 0.9804 -
val_loss: 3.8353 - val_accuracy: 0.4892
```

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

```
{'Model Name': 'CustomVGGAug', 'Epochs': 40, 'Batch Size': 64, 'Train Loss': 0.11450070887804
031, 'Val Loss': 3.486694574356079, 'Train Acc': 0.9628000259399414, 'Val Acc': 0.49219998717
308044, '[Train - Val] Acc': 0.47060003876686096}
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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 customVGG model without Data Augmentation to the customVGG model with Data Augmentation, we can see that by applying augmentation, the train loss decreased and the train accuracy increase slightly.

Training CustomVGG model with CutMix 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)
customVGGCutMixModel = Model(inputs=inputs, outputs=x, name="CustomVGGCutMix")
customVGGCutMixModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                        loss='categorical_crossentropy', metrics=['accuracy'])
```

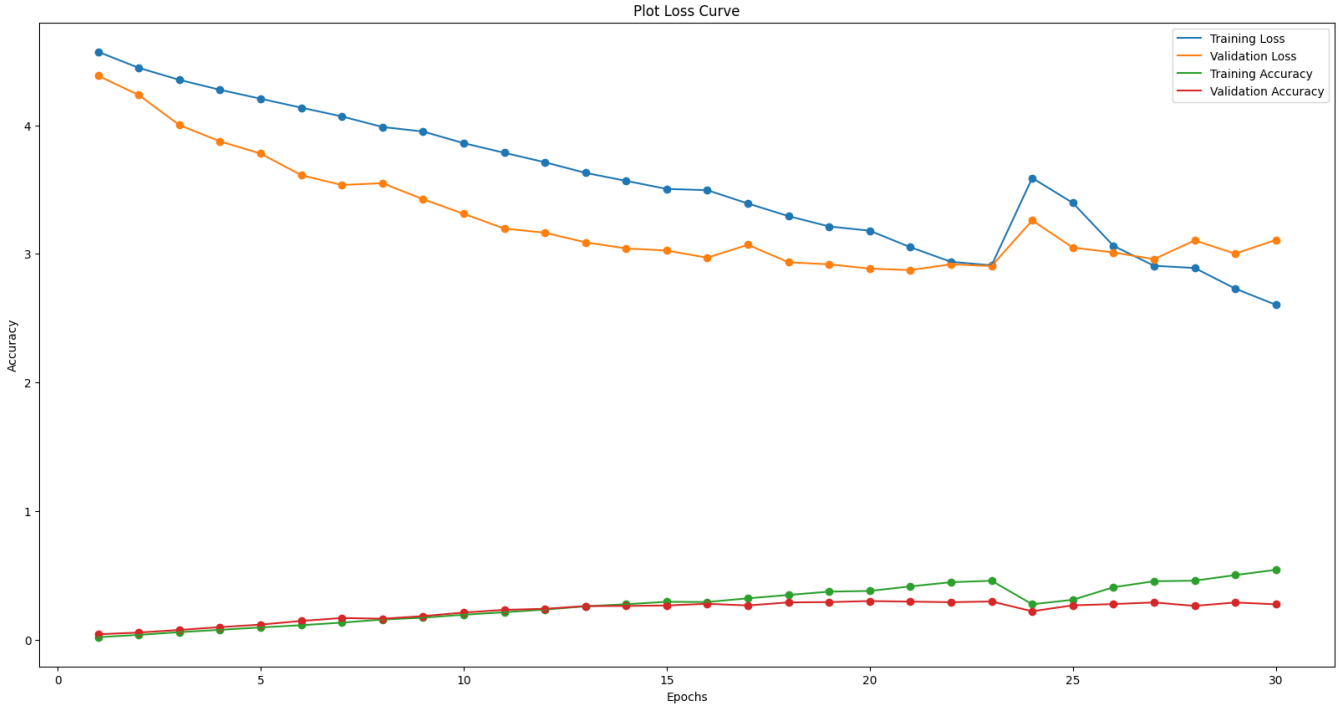
In [ ]:
```
customVGGCutMixModelHistory = customVGGCutMixModel.fit(x_train_cutmix, y_train_cutmix, epochs
                                                       validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/50
625/625 [==============================] - 16s 23ms/step - loss: 4.5711 - accuracy: 0.0192 -
val_loss: 4.3847 - val_accuracy: 0.0415
Epoch 2/50
625/625 [==============================] - 14s 22ms/step - loss: 4.4463 - accuracy: 0.0370 -
val_loss: 4.2351 - val_accuracy: 0.0552
Epoch 3/50
625/625 [==============================] - 14s 22ms/step - loss: 4.3528 - accuracy: 0.0584 -
val_loss: 4.0014 - val_accuracy: 0.0754
Epoch 4/50
625/625 [==============================] - 14s 22ms/step - loss: 4.2756 - accuracy: 0.0763 -
val_loss: 3.8750 - val_accuracy: 0.0974
Epoch 5/50
625/625 [==============================] - 14s 23ms/step - loss: 4.2061 - accuracy: 0.0950 -
val_loss: 3.7796 - val_accuracy: 0.1169
Epoch 6/50
625/625 [==============================] - 14s 23ms/step - loss: 4.1368 - accuracy: 0.1123 -
val_loss: 3.6122 - val_accuracy: 0.1453
Epoch 7/50
625/625 [==============================] - 14s 23ms/step - loss: 4.0688 - accuracy: 0.1324 -
val_loss: 3.5352 - val_accuracy: 0.1683
Epoch 8/50
625/625 [==============================] - 14s 22ms/step - loss: 3.9865 - accuracy: 0.1563 -
val_loss: 3.5497 - val_accuracy: 0.1627
Epoch 9/50
625/625 [==============================] - 14s 22ms/step - loss: 3.9512 - accuracy: 0.1706 -
val_loss: 3.4250 - val_accuracy: 0.1831
Epoch 10/50
625/625 [==============================] - 14s 22ms/step - loss: 3.8609 - accuracy: 0.1936 -
val_loss: 3.3114 - val_accuracy: 0.2106
Epoch 11/50
625/625 [==============================] - 14s 22ms/step - loss: 3.7859 - accuracy: 0.2129 -
val_loss: 3.1970 - val_accuracy: 0.2316
Epoch 12/50
625/625 [==============================] - 14s 22ms/step - loss: 3.7122 - accuracy: 0.2342 -
val_loss: 3.1644 - val_accuracy: 0.2399
Epoch 13/50
625/625 [==============================] - 14s 22ms/step - loss: 3.6297 - accuracy: 0.2591 -
val_loss: 3.0896 - val_accuracy: 0.2610
Epoch 14/50
625/625 [==============================] - 14s 23ms/step - loss: 3.5675 - accuracy: 0.2753 -
val_loss: 3.0421 - val_accuracy: 0.2621
Epoch 15/50
625/625 [==============================] - 14s 23ms/step - loss: 3.5047 - accuracy: 0.2943 -
val_loss: 3.0256 - val_accuracy: 0.2657
Epoch 16/50
625/625 [==============================] - 14s 23ms/step - loss: 3.4950 - accuracy: 0.2927 -
val_loss: 2.9699 - val_accuracy: 0.2789
Epoch 17/50
625/625 [==============================] - 14s 23ms/step - loss: 3.3914 - accuracy: 0.3214 -
val_loss: 3.0706 - val_accuracy: 0.2659
Epoch 18/50
625/625 [==============================] - 14s 23ms/step - loss: 3.2930 - accuracy: 0.3474 -
val_loss: 2.9343 - val_accuracy: 0.2897
Epoch 19/50
625/625 [==============================] - 15s 24ms/step - loss: 3.2130 - accuracy: 0.3733 -
val_loss: 2.9186 - val_accuracy: 0.2921
Epoch 20/50
625/625 [==============================] - 14s 23ms/step - loss: 3.1798 - accuracy: 0.3785 -
val_loss: 2.8853 - val_accuracy: 0.3002
Epoch 21/50
625/625 [==============================] - 14s 22ms/step - loss: 3.0517 - accuracy: 0.4141 -
val_loss: 2.8734 - val_accuracy: 0.2957
Epoch 22/50
625/625 [==============================] - 14s 23ms/step - loss: 2.9383 - accuracy: 0.4466 -
val_loss: 2.9186 - val_accuracy: 0.2912
```

```
Epoch 23/50
625/625 [==============================] - 14s 23ms/step - loss: 2.9106 - accuracy: 0.4579 -
val_loss: 2.9050 - val_accuracy: 0.2968
Epoch 24/50
625/625 [==============================] - 14s 22ms/step - loss: 3.5898 - accuracy: 0.2752 -
val_loss: 3.2602 - val_accuracy: 0.2208
Epoch 25/50
625/625 [==============================] - 14s 22ms/step - loss: 3.3962 - accuracy: 0.3098 -
val_loss: 3.0488 - val_accuracy: 0.2667
Epoch 26/50
625/625 [==============================] - 14s 22ms/step - loss: 3.0624 - accuracy: 0.4074 -
val_loss: 3.0107 - val_accuracy: 0.2767
Epoch 27/50
625/625 [==============================] - 14s 22ms/step - loss: 2.9072 - accuracy: 0.4541 -
val_loss: 2.9586 - val_accuracy: 0.2894
Epoch 28/50
625/625 [==============================] - 14s 22ms/step - loss: 2.8901 - accuracy: 0.4585 -
val_loss: 3.1043 - val_accuracy: 0.2624
Epoch 29/50
625/625 [==============================] - 14s 22ms/step - loss: 2.7295 - accuracy: 0.5020 -
val_loss: 3.0011 - val_accuracy: 0.2889
Epoch 30/50
625/625 [==============================] - 14s 22ms/step - loss: 2.6036 - accuracy: 0.5428 -
val_loss: 3.1083 - val_accuracy: 0.2748
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\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': 'CustomVGGCutMix', 'Epochs': 30, 'Batch Size': 64, 'Train Loss': 3.17982316017
1509, 'Val Loss': 2.8852908611297607, 'Train Acc': 0.3785249888896942, 'Val Acc': 0.300199985
5041504, '[Train - Val] Acc': 0.07832500338554382}
```



Observations

We can see that by applying cutmix, the training and validation accuracy decrease drastically. The decrease in the loss suggest that by applying the cutmix training set it makes the model more generalised to the validation set.

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, 100)               51300

=================================================================
Total params: 15,047,595
Trainable params: 15,038,116
Non-trainable params: 9,479
_________________________________________________________________
```

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

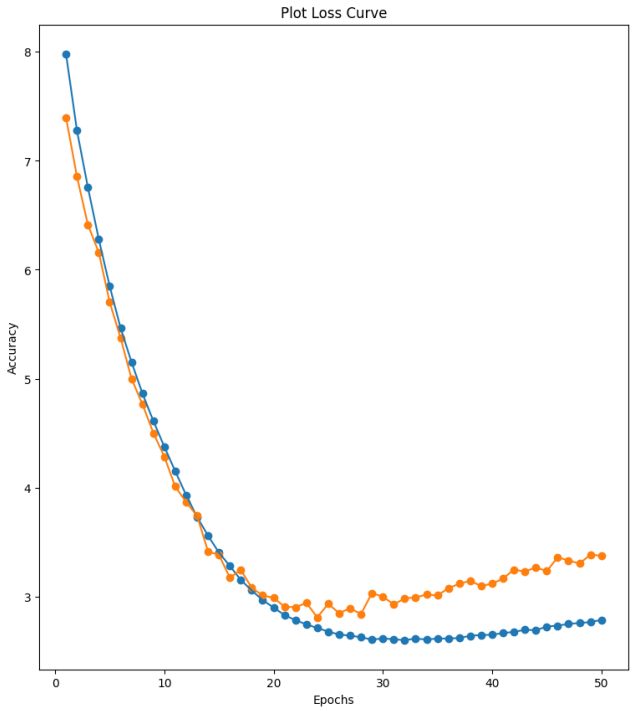
```
Epoch 1/50
 16/625 [..............................] - ETA: 48s - loss: 8.5818 - accuracy: 0.0098
```

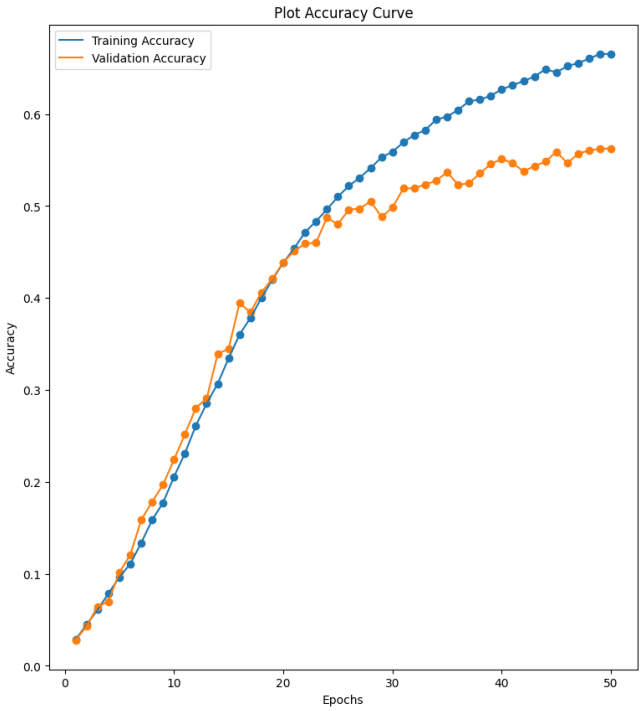
```
---------------------------------------------------------------------------
ResourceExhaustedError                    Traceback (most recent call last)
Cell In [194], line 1
----> 1 customVGG16ModelHistory = customVGG16Model.fit(x_train, y_train, epochs=50,
      2                                                validation_data=(x_val, y_val), batch_
size=BATCH_SIZE, callbacks=EarlyStopping(monitor='val_accuracy', patience=10, restore_best_we
ights=True))

File c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\utils\traceback_util
s.py:70, in filter_traceback.<locals>.error_handler(*args, **kwargs)
     67     filtered_tb = _process_traceback_frames(e.__traceback__)
     68     # To get the full stack trace, call:
     69     # `tf.debugging.disable_traceback_filtering()`
---> 70     raise e.with_traceback(filtered_tb) from None
     71 finally:
     72     del filtered_tb

File c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\tensorflow\python\eager\ex
ecute.py:54, in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     52 try:
     53   ctx.ensure_initialized()
---> 54   tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name, op_name,
     55                                       inputs, attrs, num_outputs)
     56 except core._NotOkStatusException as e:
     57   if name is not None:

ResourceExhaustedError: Graph execution error:

Detected at node 'gradient_tape/CustomVGG16/sequential_4/conv2d_12/Conv2D/Conv2DBackpropFilte
r' defined at (most recent call last):
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\runpy.py", line 194, in _run_module
_as_main
      return _run_code(code, main_globals, None,
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\runpy.py", line 87, in _run_code
      exec(code, run_globals)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel_launcher.p
y", line 17, in <module>
      app.launch_new_instance()
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\traitlets\config\appl
ication.py", line 982, in launch_instance
      app.start()
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel\kernelapp.p
y", line 712, in start
      self.io_loop.start()
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\tornado\platform\asyn
cio.py", line 215, in start
      self.asyncio_loop.run_forever()
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\asyncio\base_events.py", line 570,
 in run_forever
      self._run_once()
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\asyncio\base_events.py", line 1859,
in _run_once
      handle._run()
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\asyncio\events.py", line 81, in _ru
n
      self._context.run(self._callback, *self._args)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel\kernelbase.
py", line 510, in dispatch_queue
      await self.process_one()
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel\kernelbase.
py", line 499, in process_one
      await dispatch(*args)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel\kernelbase.
py", line 406, in dispatch_shell
      await result
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel\kernelbase.
```

```
py", line 730, in execute_request
      reply_content = await reply_content
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel\ipkernel.p
y", line 383, in do_execute
      res = shell.run_cell(
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\ipykernel\zmqshell.p
y", line 528, in run_cell
      return super().run_cell(*args, **kwargs)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\IPython\core\interact
iveshell.py", line 2940, in run_cell
      result = self._run_cell(
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\IPython\core\interact
iveshell.py", line 2995, in _run_cell
      return runner(coro)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\IPython\core\async_he
lpers.py", line 129, in _pseudo_sync_runner
      coro.send(None)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\IPython\core\interact
iveshell.py", line 3194, in run_cell_async
      has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\IPython\core\interact
iveshell.py", line 3373, in run_ast_nodes
      if await self.run_code(code, result, async_=asy):
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\IPython\core\interact
iveshell.py", line 3433, in run_code
      exec(code_obj, self.user_global_ns, self.user_ns)
    File "C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\987705819.py", line 1, in <
module>
      customVGG16ModelHistory = customVGG16Model.fit(x_train, y_train, epochs=50,
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\utils\traceback
_utils.py", line 65, in error_handler
      return fn(*args, **kwargs)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\engine\trainin
g.py", line 1564, in fit
      tmp_logs = self.train_function(iterator)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\engine\trainin
g.py", line 1160, in train_function
      return step_function(self, iterator)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\engine\trainin
g.py", line 1146, in step_function
      outputs = model.distribute_strategy.run(run_step, args=(data,))
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\engine\trainin
g.py", line 1135, in run_step
      outputs = model.train_step(data)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\engine\trainin
g.py", line 997, in train_step
      self.optimizer.minimize(loss, self.trainable_variables, tape=tape)
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\optimizers\opti
mizer_v2\optimizer_v2.py", line 576, in minimize
      grads_and_vars = self._compute_gradients(
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\optimizers\opti
mizer_v2\optimizer_v2.py", line 634, in _compute_gradients
      grads_and_vars = self._get_gradients(
    File "c:\Users\Soh Hong Yu\anaconda3\envs\gpu_env\lib\site-packages\keras\optimizers\opti
mizer_v2\optimizer_v2.py", line 510, in _get_gradients
      grads = tape.gradient(loss, var_list, grad_loss)
Node: 'gradient_tape/CustomVGG16/sequential_4/conv2d_12/Conv2D/Conv2DBackpropFilter'
OOM when allocating tensor with shape[512,512,3,3] and type float on /job:localhost/replica:
0/task:0/device:GPU:0 by allocator GPU_0_bfc
         [[{{node gradient_tape/CustomVGG16/sequential_4/conv2d_12/Conv2D/Conv2DBackpropFilte
r}}]]
Hint: If you want to see a list of allocated tensors when OOM happens, add report_tensor_allo
cations_upon_oom to RunOptions for current allocation info. This isn't available when running
in Eager mode.
 [Op:__inference_train_function_5943068]
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10452\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': 2.78526377677917
5, 'Val Loss': 3.3738694190979004, 'Train Acc': 0.6652250289916992, 'Val Acc': 0.562600016593
9331, '[Train - Val] Acc': 0.10262501239776611}
```





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'])
```

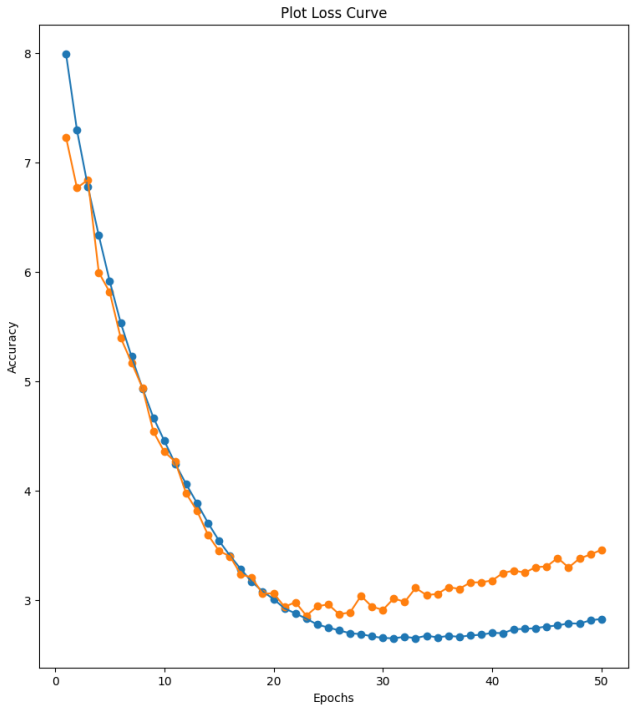
In [ ]:
```
customVGG16AugModelHistory = customVGG16AugModel.fit(x_train_aug, y_train, epochs=50,
```

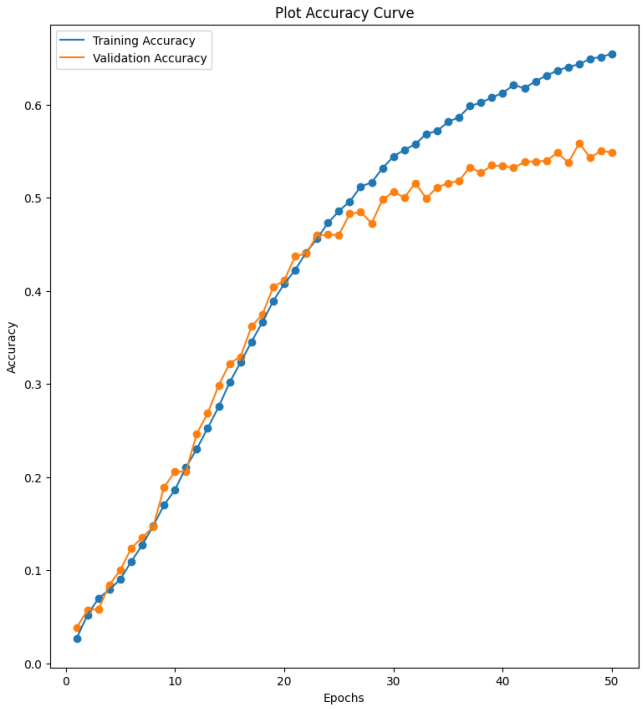
```
                                                validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/50
625/625 [==============================] - 44s 64ms/step - loss: 8.0697 - accuracy: 0.0257 -
val_loss: 7.4062 - val_accuracy: 0.0252
Epoch 2/50
625/625 [==============================] - 39s 62ms/step - loss: 7.3149 - accuracy: 0.0454 -
val_loss: 7.7932 - val_accuracy: 0.0211
Epoch 3/50
625/625 [==============================] - ETA: 0s - loss: 6.7744 - accuracy: 0.0608
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_35012\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': 2.789385795593
2617, 'Val Loss': 3.2992160320281982, 'Train Acc': 0.6434999704360962, 'Val Acc': 0.558799982
0709229, '[Train - Val] Acc': 0.08469998836517334}
```





Observations

We can see that the model has been very generalise and performance was better than the model that was not trained with the data augmentation. This means data augmentation for this model is very good.

CustomVGG-16 model with CutMix 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)
```

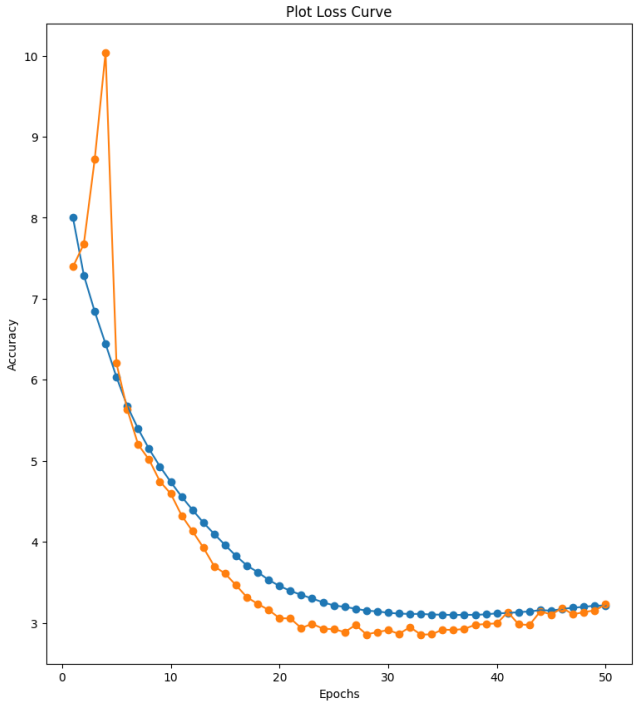
```
x = Dropout(0.5)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGG16CutMixModel = Model(inputs=inputs, outputs=x, name="CustomVGG16CutMix")
customVGG16CutMixModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                      loss='categorical_crossentropy', metrics=['accuracy'])
```

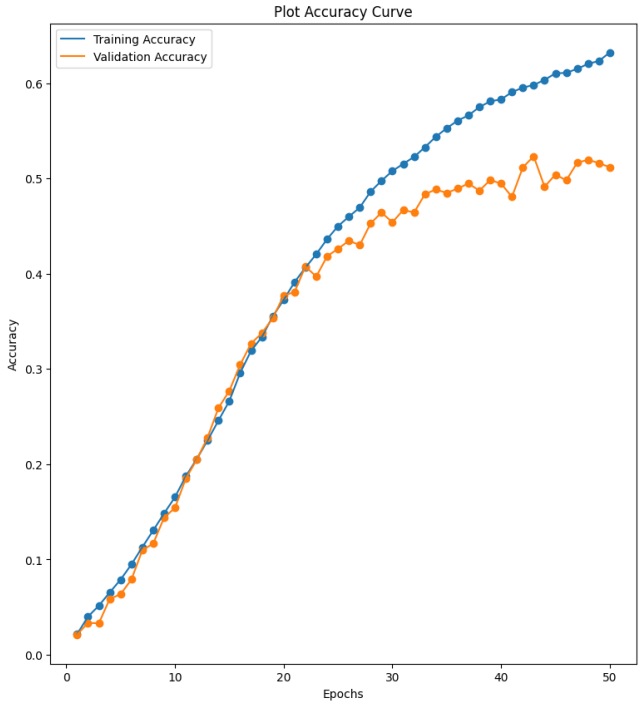
In [ ]:
```
customVGG16CutMixModelHistory = customVGG16CutMixModel.fit(x_train_cutmix, y_train_cutmix, ep
                                                validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/50
1/1 [==============================] - 10s 10s/step - loss: 8.2785 - accuracy: 0.0000e+00 - v
al_loss: 6.7970 - val_accuracy: 0.0118
Epoch 2/50
1/1 [==============================] - 3s 3s/step - loss: 8.5206 - accuracy: 0.0156 - val_los
s: 6.8063 - val_accuracy: 0.0105
Epoch 3/50
1/1 [==============================] - 3s 3s/step - loss: 8.6141 - accuracy: 0.0000e+00 - val
_loss: 6.8237 - val_accuracy: 0.0092
Epoch 4/50
1/1 [==============================] - 3s 3s/step - loss: 8.4043 - accuracy: 0.0000e+00 - val
_loss: 6.8682 - val_accuracy: 0.0070
Epoch 5/50
1/1 [==============================] - 3s 3s/step - loss: 8.1857 - accuracy: 0.0000e+00 - val
_loss: 7.2884 - val_accuracy: 0.0103
Epoch 6/50
1/1 [==============================] - 3s 3s/step - loss: 8.4571 - accuracy: 0.0156 - val_los
s: 10.8335 - val_accuracy: 0.0109
Epoch 7/50
1/1 [==============================] - 4s 4s/step - loss: 8.4896 - accuracy: 0.0000e+00 - val
_loss: 26.0565 - val_accuracy: 0.0109
Epoch 8/50
1/1 [==============================] - 4s 4s/step - loss: 8.7853 - accuracy: 0.0000e+00 - val
_loss: 62.8605 - val_accuracy: 0.0109
Epoch 9/50
1/1 [==============================] - 4s 4s/step - loss: 8.5580 - accuracy: 0.0000e+00 - val
_loss: 135.6938 - val_accuracy: 0.0109
Epoch 10/50
1/1 [==============================] - 4s 4s/step - loss: 8.2217 - accuracy: 0.0156 - val_los
s: 274.8455 - val_accuracy: 0.0075
Epoch 11/50
1/1 [==============================] - 4s 4s/step - loss: 8.4665 - accuracy: 0.0156 - val_los
s: 512.4149 - val_accuracy: 0.0088
```

In [ ]:
```
print(storeResult(customVGG16CutMixModelHistory))
plot_loss_curve(customVGG16CutMixModelHistory)
plt.show()
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_30216\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': 'CustomVGG16CutMix', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 3.141379594
8028564, 'Val Loss': 2.9746015071868896, 'Train Acc': 0.5979750156402588, 'Val Acc': 0.523199
9754905701, '[Train - Val] Acc': 0.07477504014968872}
```





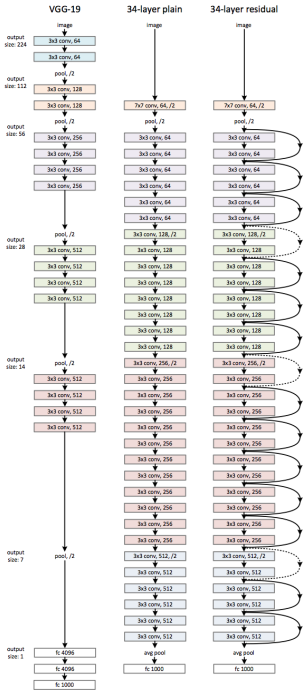
Observations

Comparing the customVGG-16 model that was not train with cutmix and was more generalised and the one trained on cutmix was more generalise to the validation set but has a lower accuracy. This suggest the model was slightly underfitted and this caused the accuracy to decrease.

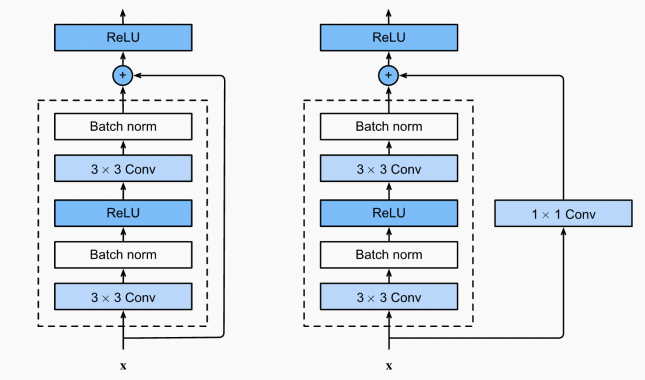
CustomResNet Model

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'])
```

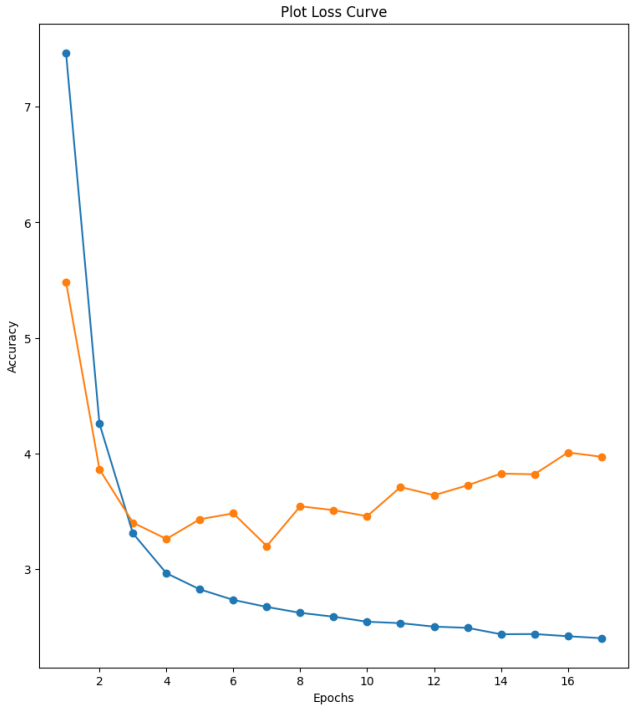
In [ ]:

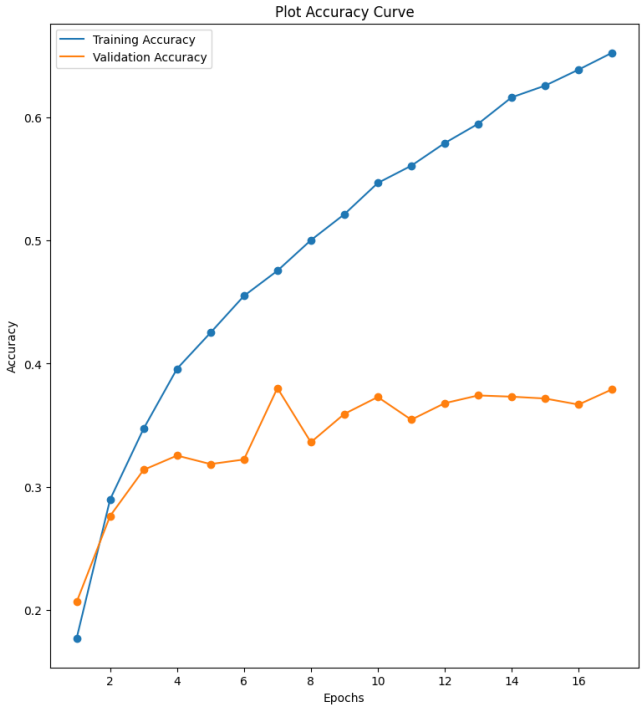
```
customResNetModelHistory = customResNetModel.fit(x_train, y_train, epochs=50,
                                                  validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/50
625/625 [==============================] - 23s 32ms/step - loss: 7.4661 - accuracy: 0.1772 -
val_loss: 5.4840 - val_accuracy: 0.2066
Epoch 2/50
625/625 [==============================] - 18s 28ms/step - loss: 4.2588 - accuracy: 0.2898 -
val_loss: 3.8644 - val_accuracy: 0.2764
Epoch 3/50
625/625 [==============================] - 19s 31ms/step - loss: 3.3108 - accuracy: 0.3472 -
val_loss: 3.4037 - val_accuracy: 0.3138
Epoch 4/50
625/625 [==============================] - 18s 28ms/step - loss: 2.9659 - accuracy: 0.3959 -
val_loss: 3.2623 - val_accuracy: 0.3254
Epoch 5/50
625/625 [==============================] - 18s 29ms/step - loss: 2.8265 - accuracy: 0.4251 -
val_loss: 3.4327 - val_accuracy: 0.3185
Epoch 6/50
625/625 [==============================] - 18s 28ms/step - loss: 2.7353 - accuracy: 0.4550 -
val_loss: 3.4837 - val_accuracy: 0.3222
Epoch 7/50
625/625 [==============================] - 18s 30ms/step - loss: 2.6745 - accuracy: 0.4754 -
val_loss: 3.2010 - val_accuracy: 0.3802
Epoch 8/50
625/625 [==============================] - 20s 31ms/step - loss: 2.6236 - accuracy: 0.5002 -
val_loss: 3.5448 - val_accuracy: 0.3362
Epoch 9/50
625/625 [==============================] - 20s 32ms/step - loss: 2.5901 - accuracy: 0.5212 -
val_loss: 3.5114 - val_accuracy: 0.3592
Epoch 10/50
625/625 [==============================] - 22s 35ms/step - loss: 2.5466 - accuracy: 0.5466 -
val_loss: 3.4592 - val_accuracy: 0.3729
Epoch 11/50
625/625 [==============================] - 20s 33ms/step - loss: 2.5343 - accuracy: 0.5606 -
val_loss: 3.7115 - val_accuracy: 0.3545
Epoch 12/50
625/625 [==============================] - 19s 30ms/step - loss: 2.5044 - accuracy: 0.5790 -
val_loss: 3.6404 - val_accuracy: 0.3679
Epoch 13/50
625/625 [==============================] - 20s 32ms/step - loss: 2.4936 - accuracy: 0.5947 -
val_loss: 3.7261 - val_accuracy: 0.3742
Epoch 14/50
625/625 [==============================] - 19s 30ms/step - loss: 2.4386 - accuracy: 0.6161 -
val_loss: 3.8276 - val_accuracy: 0.3732
Epoch 15/50
625/625 [==============================] - 20s 31ms/step - loss: 2.4402 - accuracy: 0.6256 -
val_loss: 3.8214 - val_accuracy: 0.3717
Epoch 16/50
625/625 [==============================] - 18s 29ms/step - loss: 2.4214 - accuracy: 0.6386 -
val_loss: 4.0100 - val_accuracy: 0.3668
Epoch 17/50
625/625 [==============================] - 17s 27ms/step - loss: 2.4044 - accuracy: 0.6523 -
val_loss: 3.9732 - val_accuracy: 0.3791
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_35012\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': 'CustomResNet', 'Epochs': 17, 'Batch Size': 64, 'Train Loss': 2.67454671859741
2, 'Val Loss': 3.201004981994629, 'Train Acc': 0.47540000081062317, 'Val Acc': 0.380199998617
17224, '[Train - Val] Acc': 0.09520000219345093}
```





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

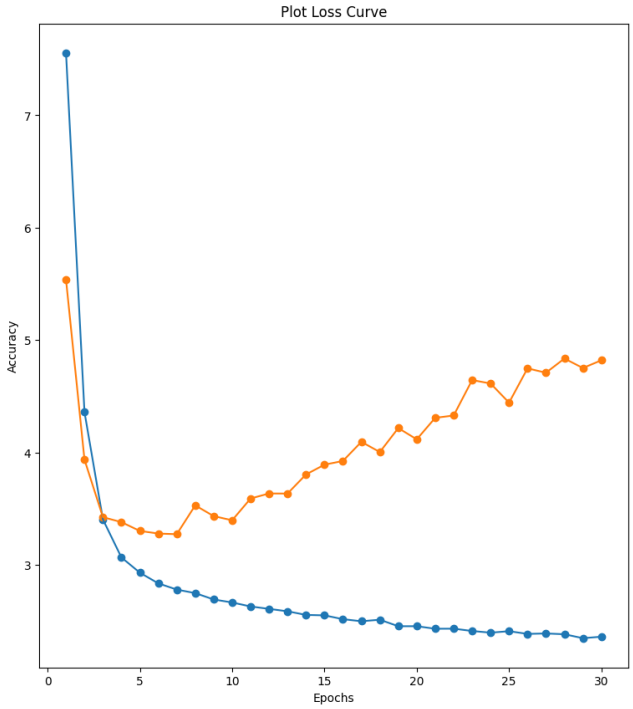
In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
weight_decay=0.005
x = pre_processing_v1(inputs)
x = ZeroPadding2D((3, 3))(x)
x = Conv2D(64, kernel_size=7, strides=2, padding='same', kernel_regularizer=l2(weight_decay))
x = BatchNormalization()(x)
x = Activation('relu')(x)
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'])
```

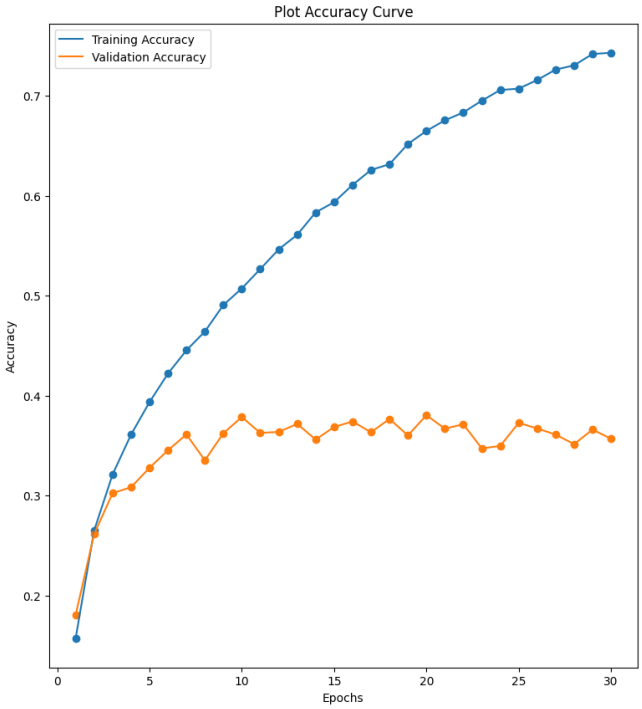
In [ ]:
```
customResNetAugModelHistory = customResNetAugModel.fit(x_train_aug, y_train, epochs=50,
                                                        validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/50
625/625 [==============================] - 20s 28ms/step - loss: 7.5562 - accuracy: 0.1541 -
val_loss: 5.5738 - val_accuracy: 0.1859
Epoch 2/50
625/625 [==============================] - 17s 28ms/step - loss: 4.3617 - accuracy: 0.2613 -
val_loss: 4.1984 - val_accuracy: 0.2280
Epoch 3/50
625/625 [==============================] - 18s 29ms/step - loss: 3.4240 - accuracy: 0.3198 -
val_loss: 3.6390 - val_accuracy: 0.2634
Epoch 4/50
625/625 [==============================] - 17s 28ms/step - loss: 3.0797 - accuracy: 0.3605 -
val_loss: 3.4258 - val_accuracy: 0.2982
Epoch 5/50
625/625 [==============================] - 18s 29ms/step - loss: 2.9377 - accuracy: 0.3914 -
val_loss: 3.4849 - val_accuracy: 0.3067
Epoch 6/50
625/625 [==============================] - 18s 28ms/step - loss: 2.8456 - accuracy: 0.4189 -
val_loss: 3.2379 - val_accuracy: 0.3556
Epoch 7/50
625/625 [==============================] - 18s 28ms/step - loss: 2.7945 - accuracy: 0.4425 -
val_loss: 3.4696 - val_accuracy: 0.3294
Epoch 8/50
625/625 [==============================] - 18s 29ms/step - loss: 2.7345 - accuracy: 0.4688 -
val_loss: 3.4808 - val_accuracy: 0.3449
Epoch 9/50
625/625 [==============================] - 19s 31ms/step - loss: 2.7110 - accuracy: 0.4896 -
val_loss: 3.4884 - val_accuracy: 0.3477
Epoch 10/50
625/625 [==============================] - 18s 29ms/step - loss: 2.6814 - accuracy: 0.5055 -
val_loss: 3.4922 - val_accuracy: 0.3631
Epoch 11/50
625/625 [==============================] - 19s 30ms/step - loss: 2.6343 - accuracy: 0.5296 -
val_loss: 3.5544 - val_accuracy: 0.3699
Epoch 12/50
583/625 [==========================>...] - ETA: 1s - loss: 2.5900 - accuracy: 0.5479
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_15772\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': 'CustomResNetAug', 'Epochs': 30, 'Batch Size': 64, 'Train Loss': 2.45558857917
78564, 'Val Loss': 4.1176228523254395, 'Train Acc': 0.6650000214576721, 'Val Acc': 0.38049998
87943268, '[Train - Val] Acc': 0.28450003266334534}
```





Observations

By comparing the ResNet with Augmentation to ResNet without Augmentation, we can see that the validation accuracy plateau more and become more generalised. The validation accuracy and training accuracy also increase as well this suggest that augmentation improved the model.

Training CustomResNet model with CutMix Augmentation

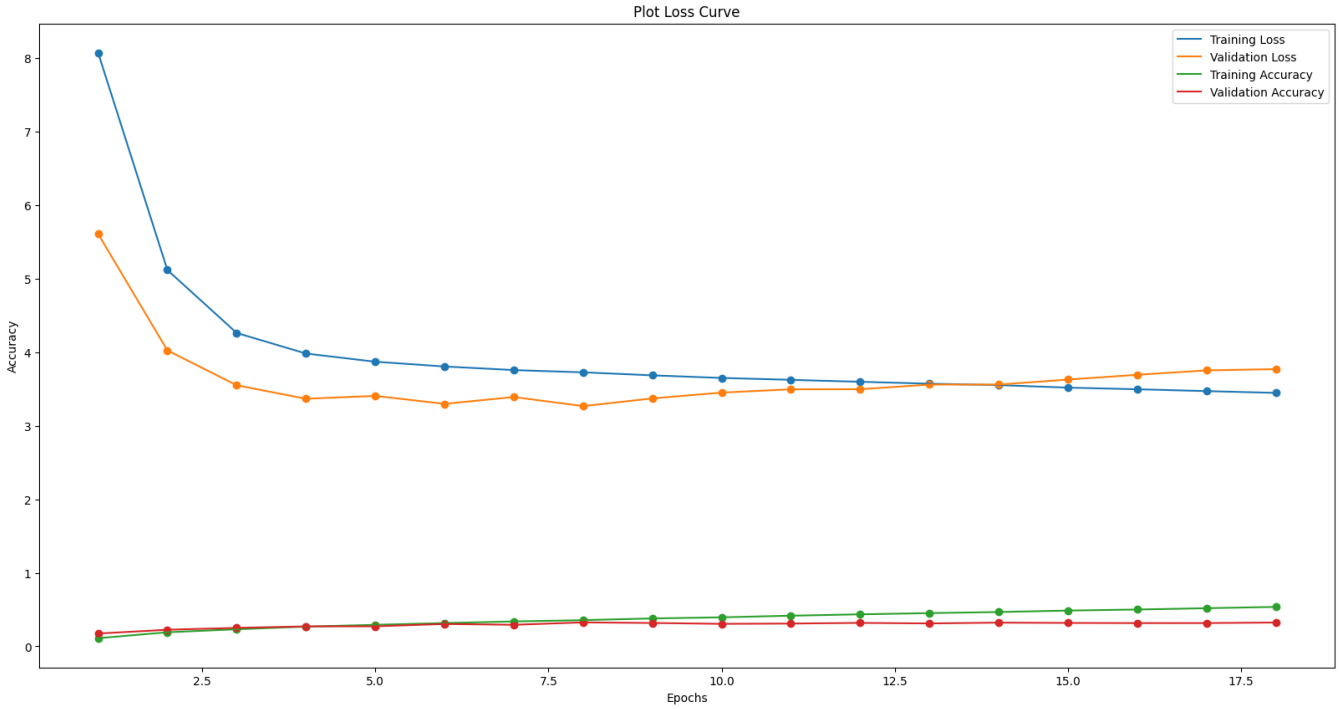
In [ ]:
```
tf.keras.backend.clear_session()
inputs = Input(IMG_SIZE)  # Input
weight_decay=0.005
x = pre_processing_v1(inputs)
x = ZeroPadding2D((3, 3))(x)
x = Conv2D(64, kernel_size=7, strides=2, padding='same', kernel_regularizer=l2(weight_decay))
x = BatchNormalization()(x)
x = Activation('relu')(x)
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)
customResNetCutMixModel = Model(inputs=inputs, outputs=x, name="CustomResNetCutMix")
customResNetCutMixModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                          loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
customResNetCutMixModelHistory = customResNetCutMixModel.fit(x_train_cutmix, y_train_cutmix,
                                                validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/50
625/625 [==============================] - 20s 27ms/step - loss: 8.0739 - accuracy: 0.1102 -
val_loss: 5.6092 - val_accuracy: 0.1736
Epoch 2/50
625/625 [==============================] - 18s 29ms/step - loss: 5.1185 - accuracy: 0.1909 -
val_loss: 4.0249 - val_accuracy: 0.2249
Epoch 3/50
625/625 [==============================] - 17s 27ms/step - loss: 4.2625 - accuracy: 0.2316 -
val_loss: 3.5506 - val_accuracy: 0.2504
Epoch 4/50
625/625 [==============================] - 17s 27ms/step - loss: 3.9826 - accuracy: 0.2673 -
val_loss: 3.3675 - val_accuracy: 0.2708
Epoch 5/50
625/625 [==============================] - 16s 26ms/step - loss: 3.8712 - accuracy: 0.2919 -
val_loss: 3.4064 - val_accuracy: 0.2717
Epoch 6/50
625/625 [==============================] - 17s 26ms/step - loss: 3.8065 - accuracy: 0.3159 -
val_loss: 3.2979 - val_accuracy: 0.3035
Epoch 7/50
625/625 [==============================] - 16s 26ms/step - loss: 3.7575 - accuracy: 0.3378 -
val_loss: 3.3912 - val_accuracy: 0.2922
Epoch 8/50
625/625 [==============================] - 16s 26ms/step - loss: 3.7269 - accuracy: 0.3553 -
val_loss: 3.2672 - val_accuracy: 0.3251
Epoch 9/50
625/625 [==============================] - 16s 25ms/step - loss: 3.6858 - accuracy: 0.3787 -
val_loss: 3.3721 - val_accuracy: 0.3168
Epoch 10/50
625/625 [==============================] - 16s 26ms/step - loss: 3.6505 - accuracy: 0.3940 -
val_loss: 3.4511 - val_accuracy: 0.3052
Epoch 11/50
625/625 [==============================] - 16s 26ms/step - loss: 3.6246 - accuracy: 0.4161 -
val_loss: 3.4961 - val_accuracy: 0.3092
Epoch 12/50
625/625 [==============================] - 17s 27ms/step - loss: 3.5984 - accuracy: 0.4347 -
val_loss: 3.4972 - val_accuracy: 0.3180
Epoch 13/50
625/625 [==============================] - 19s 31ms/step - loss: 3.5722 - accuracy: 0.4513 -
val_loss: 3.5601 - val_accuracy: 0.3109
Epoch 14/50
625/625 [==============================] - 20s 32ms/step - loss: 3.5535 - accuracy: 0.4661 -
val_loss: 3.5603 - val_accuracy: 0.3217
Epoch 15/50
625/625 [==============================] - 17s 27ms/step - loss: 3.5186 - accuracy: 0.4860 -
val_loss: 3.6285 - val_accuracy: 0.3177
Epoch 16/50
625/625 [==============================] - 17s 27ms/step - loss: 3.4967 - accuracy: 0.4999 -
val_loss: 3.6934 - val_accuracy: 0.3152
Epoch 17/50
625/625 [==============================] - 16s 26ms/step - loss: 3.4716 - accuracy: 0.5179 -
val_loss: 3.7534 - val_accuracy: 0.3158
Epoch 18/50
625/625 [==============================] - 16s 26ms/step - loss: 3.4472 - accuracy: 0.5358 -
val_loss: 3.7703 - val_accuracy: 0.3232
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\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': 'CustomResNetCutMix', 'Epochs': 18, 'Batch Size': 64, 'Train Loss': 3.72687578
20129395, 'Val Loss': 3.2672340869903564, 'Train Acc': 0.35534998774528503, 'Val Acc': 0.3251
0000467300415, '[Train - Val] Acc': 0.030249983072280884}
```



Observations

We can see that by applying the cutmix algorithm, the model becomes more generalise as the accuracy are very similar. However, although the model is generalise the accuracy is very low which means some changes needs to be done to make it better

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, 100)          51300       ['flatten[0][0]']

==================================================================================================
=====
Total params: 21,357,931
Trainable params: 21,342,692
Non-trainable params: 15,239
__________________________________________________________________________________________________
_____
```

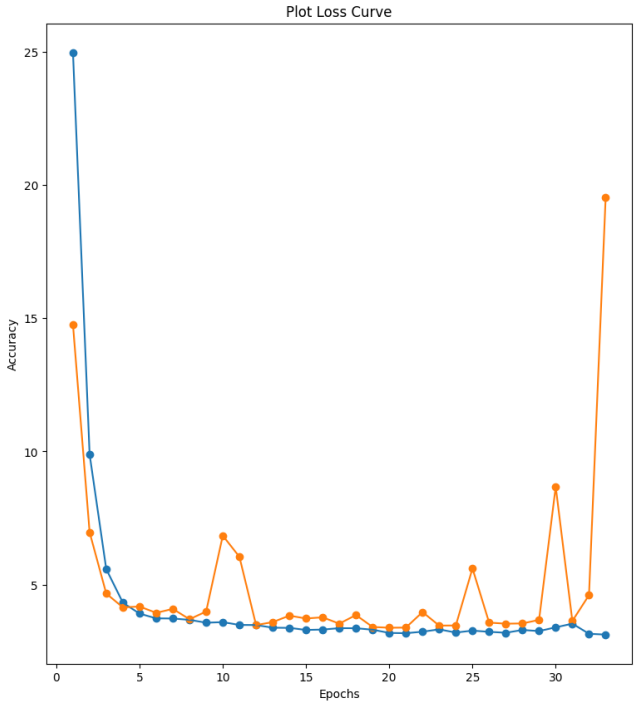
In [ ]:
```
customResNetDropModelHistory = customResNetDropModel.fit(x_train, y_train, epochs=50,
                                                validation_data=(x_val, y_val), batch_size=BAT
```

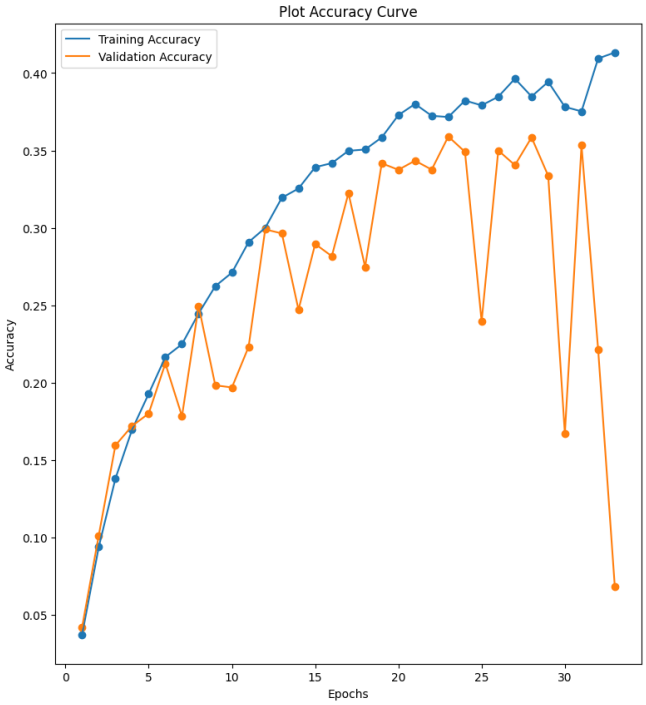
```
Epoch 1/50
625/625 [==============================] - 66s 94ms/step - loss: 24.9811 - accuracy: 0.0373 -
val_loss: 14.7620 - val_accuracy: 0.0421
Epoch 2/50
625/625 [==============================] - 57s 92ms/step - loss: 9.8900 - accuracy: 0.0939 -
val_loss: 6.9625 - val_accuracy: 0.1013
Epoch 3/50
625/625 [==============================] - 57s 91ms/step - loss: 5.5795 - accuracy: 0.1380 -
val_loss: 4.6622 - val_accuracy: 0.1595
Epoch 4/50
625/625 [==============================] - 56s 90ms/step - loss: 4.3186 - accuracy: 0.1696 -
val_loss: 4.1422 - val_accuracy: 0.1720
Epoch 5/50
625/625 [==============================] - 54s 87ms/step - loss: 3.9090 - accuracy: 0.1928 -
val_loss: 4.1753 - val_accuracy: 0.1801
Epoch 6/50
625/625 [==============================] - 53s 85ms/step - loss: 3.7301 - accuracy: 0.2165 -
val_loss: 3.9380 - val_accuracy: 0.2124
Epoch 7/50
625/625 [==============================] - 53s 85ms/step - loss: 3.7247 - accuracy: 0.2249 -
val_loss: 4.0858 - val_accuracy: 0.1783
Epoch 8/50
625/625 [==============================] - 53s 85ms/step - loss: 3.6746 - accuracy: 0.2447 -
val_loss: 3.6978 - val_accuracy: 0.2495
Epoch 9/50
625/625 [==============================] - 53s 85ms/step - loss: 3.5676 - accuracy: 0.2623 -
val_loss: 3.9868 - val_accuracy: 0.1983
Epoch 10/50
625/625 [==============================] - 53s 85ms/step - loss: 3.5891 - accuracy: 0.2711 -
val_loss: 6.8377 - val_accuracy: 0.1969
Epoch 11/50
625/625 [==============================] - 54s 86ms/step - loss: 3.4835 - accuracy: 0.2908 -
val_loss: 6.0414 - val_accuracy: 0.2231
Epoch 12/50
625/625 [==============================] - 53s 85ms/step - loss: 3.4796 - accuracy: 0.3000 -
val_loss: 3.4783 - val_accuracy: 0.2990
Epoch 13/50
625/625 [==============================] - 53s 85ms/step - loss: 3.3804 - accuracy: 0.3194 -
val_loss: 3.5897 - val_accuracy: 0.2964
Epoch 14/50
625/625 [==============================] - 53s 85ms/step - loss: 3.3733 - accuracy: 0.3254 -
val_loss: 3.8322 - val_accuracy: 0.2471
Epoch 15/50
625/625 [==============================] - 53s 84ms/step - loss: 3.2960 - accuracy: 0.3392 -
val_loss: 3.7262 - val_accuracy: 0.2896
Epoch 16/50
625/625 [==============================] - 53s 85ms/step - loss: 3.3103 - accuracy: 0.3418 -
val_loss: 3.7698 - val_accuracy: 0.2817
Epoch 17/50
625/625 [==============================] - 53s 85ms/step - loss: 3.3616 - accuracy: 0.3498 -
val_loss: 3.5315 - val_accuracy: 0.3224
Epoch 18/50
625/625 [==============================] - 53s 85ms/step - loss: 3.3593 - accuracy: 0.3506 -
val_loss: 3.8554 - val_accuracy: 0.2748
Epoch 19/50
625/625 [==============================] - 53s 85ms/step - loss: 3.3048 - accuracy: 0.3584 -
val_loss: 3.4070 - val_accuracy: 0.3417
Epoch 20/50
625/625 [==============================] - 53s 85ms/step - loss: 3.1834 - accuracy: 0.3728 -
val_loss: 3.3786 - val_accuracy: 0.3375
Epoch 21/50
625/625 [==============================] - 53s 85ms/step - loss: 3.1738 - accuracy: 0.3800 -
val_loss: 3.3845 - val_accuracy: 0.3434
Epoch 22/50
625/625 [==============================] - 53s 84ms/step - loss: 3.2293 - accuracy: 0.3724 -
val_loss: 3.9668 - val_accuracy: 0.3376
```

```
Epoch 23/50
625/625 [==============================] - 53s 85ms/step - loss: 3.3161 - accuracy: 0.3716 -
val_loss: 3.4584 - val_accuracy: 0.3590
Epoch 24/50
625/625 [==============================] - 53s 85ms/step - loss: 3.1977 - accuracy: 0.3822 -
val_loss: 3.4633 - val_accuracy: 0.3491
Epoch 25/50
625/625 [==============================] - 53s 84ms/step - loss: 3.2698 - accuracy: 0.3791 -
val_loss: 5.6067 - val_accuracy: 0.2395
Epoch 26/50
625/625 [==============================] - 53s 84ms/step - loss: 3.2253 - accuracy: 0.3847 -
val_loss: 3.5716 - val_accuracy: 0.3500
Epoch 27/50
625/625 [==============================] - 53s 85ms/step - loss: 3.1896 - accuracy: 0.3963 -
val_loss: 3.5297 - val_accuracy: 0.3405
Epoch 28/50
625/625 [==============================] - 53s 85ms/step - loss: 3.2943 - accuracy: 0.3848 -
val_loss: 3.5395 - val_accuracy: 0.3584
Epoch 29/50
625/625 [==============================] - 53s 85ms/step - loss: 3.2532 - accuracy: 0.3943 -
val_loss: 3.6727 - val_accuracy: 0.3336
Epoch 30/50
625/625 [==============================] - 53s 84ms/step - loss: 3.3931 - accuracy: 0.3781 -
val_loss: 8.6747 - val_accuracy: 0.1672
Epoch 31/50
625/625 [==============================] - 53s 85ms/step - loss: 3.5310 - accuracy: 0.3753 -
val_loss: 3.6317 - val_accuracy: 0.3533
Epoch 32/50
625/625 [==============================] - 53s 84ms/step - loss: 3.1519 - accuracy: 0.4094 -
val_loss: 4.6024 - val_accuracy: 0.2213
Epoch 33/50
625/625 [==============================] - 53s 85ms/step - loss: 3.1209 - accuracy: 0.4133 -
val_loss: 19.5371 - val_accuracy: 0.0683
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_15772\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': 33, 'Batch Size': 64, 'Train Loss': 3.31612491
607666, 'Val Loss': 3.458369255065918, 'Train Acc': 0.3716000020503998, 'Val Acc': 0.35899999
737739563, '[Train - Val] Acc': 0.01260000467300415}
```





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.

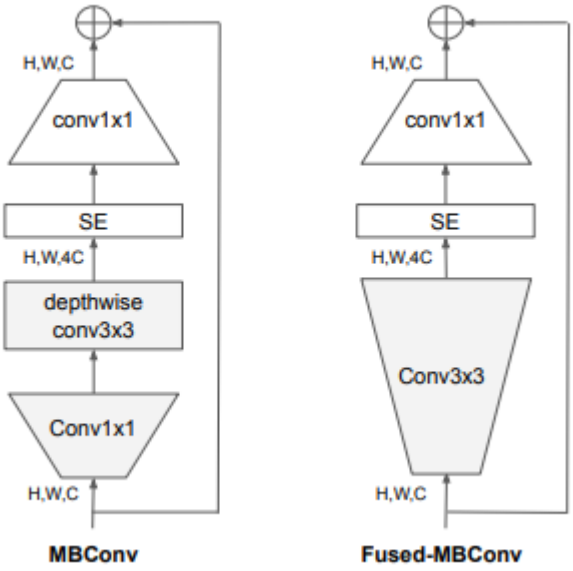
EfficientNetV2 Model

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, 100)               128100

=================================================================
Total params: 6,047,419
Trainable params: 5,986,804
Non-trainable params: 60,615
_________________________________________________________________
```

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

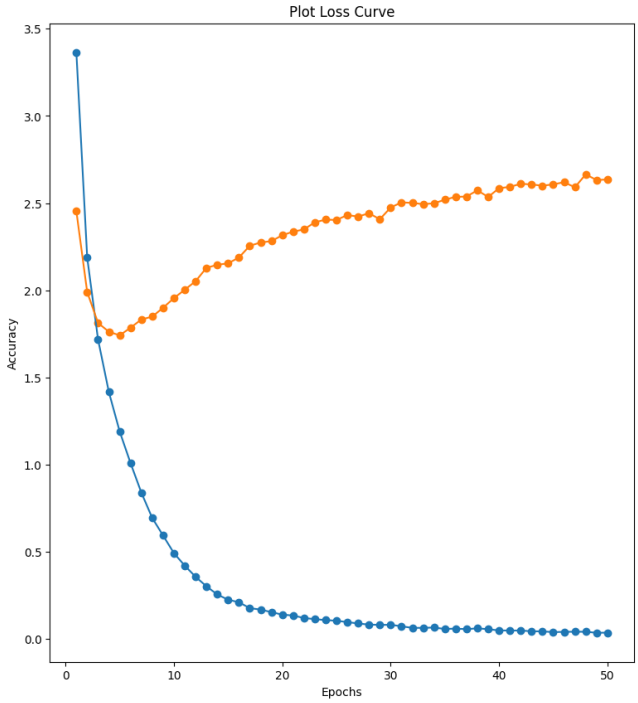
```
Epoch 1/50
625/625 [==============================] - 105s 146ms/step - loss: 3.3650 - accuracy: 0.2138
- val_loss: 2.4541 - val_accuracy: 0.3616
Epoch 2/50
625/625 [==============================] - 103s 164ms/step - loss: 2.1879 - accuracy: 0.4212
- val_loss: 1.9901 - val_accuracy: 0.4664
Epoch 3/50
625/625 [==============================] - 130s 207ms/step - loss: 1.7188 - accuracy: 0.5245
- val_loss: 1.8133 - val_accuracy: 0.5092
Epoch 4/50
625/625 [==============================] - 117s 187ms/step - loss: 1.4167 - accuracy: 0.6008
- val_loss: 1.7624 - val_accuracy: 0.5275
Epoch 5/50
625/625 [==============================] - 136s 217ms/step - loss: 1.1900 - accuracy: 0.6519
- val_loss: 1.7420 - val_accuracy: 0.5445
Epoch 6/50
625/625 [==============================] - 129s 206ms/step - loss: 1.0092 - accuracy: 0.7025
- val_loss: 1.7859 - val_accuracy: 0.5463
Epoch 7/50
625/625 [==============================] - 87s 138ms/step - loss: 0.8397 - accuracy: 0.7462 -
val_loss: 1.8330 - val_accuracy: 0.5514
Epoch 8/50
625/625 [==============================] - 99s 158ms/step - loss: 0.6948 - accuracy: 0.7871 -
val_loss: 1.8493 - val_accuracy: 0.5525
Epoch 9/50
625/625 [==============================] - 450s 721ms/step - loss: 0.5942 - accuracy: 0.8178
- val_loss: 1.9001 - val_accuracy: 0.5576
Epoch 10/50
625/625 [==============================] - 522s 836ms/step - loss: 0.4932 - accuracy: 0.8473
- val_loss: 1.9553 - val_accuracy: 0.5629
Epoch 11/50
625/625 [==============================] - 749s 1s/step - loss: 0.4214 - accuracy: 0.8687 - v
al_loss: 2.0035 - val_accuracy: 0.5594
Epoch 12/50
625/625 [==============================] - 1095s 2s/step - loss: 0.3573 - accuracy: 0.8877 -
val_loss: 2.0523 - val_accuracy: 0.5585
Epoch 13/50
625/625 [==============================] - 2214s 4s/step - loss: 0.3030 - accuracy: 0.9060 -
val_loss: 2.1290 - val_accuracy: 0.5543
Epoch 14/50
625/625 [==============================] - 1617s 3s/step - loss: 0.2560 - accuracy: 0.9206 -
val_loss: 2.1467 - val_accuracy: 0.5636
Epoch 15/50
625/625 [==============================] - 716s 1s/step - loss: 0.2259 - accuracy: 0.9303 - v
al_loss: 2.1548 - val_accuracy: 0.5662
Epoch 16/50
625/625 [==============================] - 687s 1s/step - loss: 0.2114 - accuracy: 0.9348 - v
al_loss: 2.1884 - val_accuracy: 0.5643
Epoch 17/50
625/625 [==============================] - 2249s 4s/step - loss: 0.1776 - accuracy: 0.9453 -
val_loss: 2.2554 - val_accuracy: 0.5653
Epoch 18/50
625/625 [==============================] - 711s 1s/step - loss: 0.1693 - accuracy: 0.9481 - v
al_loss: 2.2747 - val_accuracy: 0.5667
Epoch 19/50
625/625 [==============================] - 643s 1s/step - loss: 0.1546 - accuracy: 0.9514 - v
al_loss: 2.2821 - val_accuracy: 0.5680
Epoch 20/50
625/625 [==============================] - 673s 1s/step - loss: 0.1403 - accuracy: 0.9571 - v
al_loss: 2.3163 - val_accuracy: 0.5726
Epoch 21/50
625/625 [==============================] - 662s 1s/step - loss: 0.1352 - accuracy: 0.9584 - v
al_loss: 2.3366 - val_accuracy: 0.5692
Epoch 22/50
625/625 [==============================] - 728s 1s/step - loss: 0.1212 - accuracy: 0.9626 - v
al_loss: 2.3495 - val_accuracy: 0.5739
```

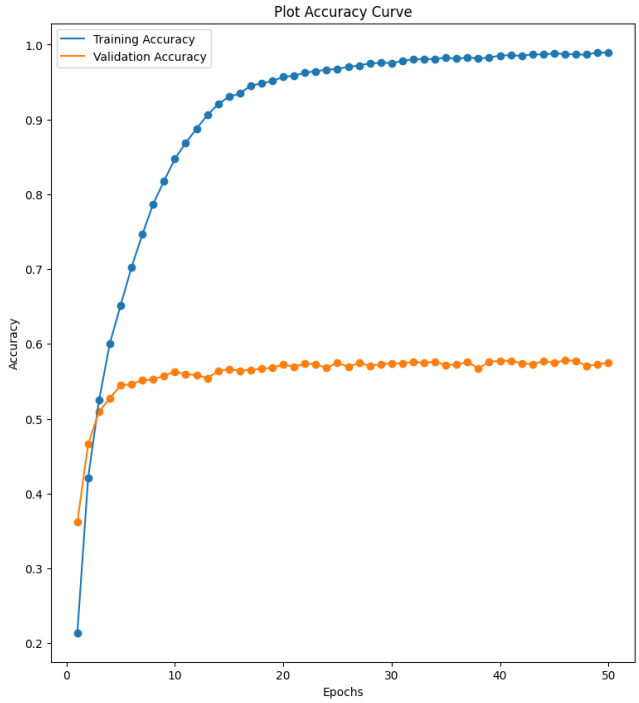
```
Epoch 23/50
625/625 [==============================] - 885s 1s/step - loss: 0.1145 - accuracy: 0.9645 - v
al_loss: 2.3901 - val_accuracy: 0.5728
Epoch 24/50
625/625 [==============================] - 979s 2s/step - loss: 0.1092 - accuracy: 0.9664 - v
al_loss: 2.4057 - val_accuracy: 0.5679
Epoch 25/50
625/625 [==============================] - 905s 1s/step - loss: 0.1052 - accuracy: 0.9678 - v
al_loss: 2.4037 - val_accuracy: 0.5750
Epoch 26/50
625/625 [==============================] - 878s 1s/step - loss: 0.0976 - accuracy: 0.9703 - v
al_loss: 2.4318 - val_accuracy: 0.5694
Epoch 27/50
625/625 [==============================] - 771s 1s/step - loss: 0.0896 - accuracy: 0.9721 - v
al_loss: 2.4231 - val_accuracy: 0.5744
Epoch 28/50
625/625 [==============================] - 842s 1s/step - loss: 0.0834 - accuracy: 0.9751 - v
al_loss: 2.4424 - val_accuracy: 0.5707
Epoch 29/50
625/625 [==============================] - 847s 1s/step - loss: 0.0814 - accuracy: 0.9757 - v
al_loss: 2.4078 - val_accuracy: 0.5728
Epoch 30/50
625/625 [==============================] - 949s 2s/step - loss: 0.0821 - accuracy: 0.9756 - v
al_loss: 2.4750 - val_accuracy: 0.5741
Epoch 31/50
625/625 [==============================] - 883s 1s/step - loss: 0.0735 - accuracy: 0.9780 - v
al_loss: 2.5047 - val_accuracy: 0.5737
Epoch 32/50
625/625 [==============================] - 899s 1s/step - loss: 0.0644 - accuracy: 0.9806 - v
al_loss: 2.5013 - val_accuracy: 0.5762
Epoch 33/50
625/625 [==============================] - 834s 1s/step - loss: 0.0645 - accuracy: 0.9809 - v
al_loss: 2.4943 - val_accuracy: 0.5744
Epoch 34/50
625/625 [==============================] - 835s 1s/step - loss: 0.0664 - accuracy: 0.9806 - v
al_loss: 2.4994 - val_accuracy: 0.5765
Epoch 35/50
625/625 [==============================] - 791s 1s/step - loss: 0.0581 - accuracy: 0.9827 - v
al_loss: 2.5198 - val_accuracy: 0.5714
Epoch 36/50
625/625 [==============================] - 798s 1s/step - loss: 0.0603 - accuracy: 0.9812 - v
al_loss: 2.5365 - val_accuracy: 0.5725
Epoch 37/50
625/625 [==============================] - 1038s 2s/step - loss: 0.0570 - accuracy: 0.9829 -
val_loss: 2.5379 - val_accuracy: 0.5758
Epoch 38/50
625/625 [==============================] - 1133s 2s/step - loss: 0.0619 - accuracy: 0.9815 -
val_loss: 2.5736 - val_accuracy: 0.5674
Epoch 39/50
625/625 [==============================] - 974s 2s/step - loss: 0.0568 - accuracy: 0.9828 - v
al_loss: 2.5355 - val_accuracy: 0.5758
Epoch 40/50
625/625 [==============================] - 871s 1s/step - loss: 0.0497 - accuracy: 0.9854 - v
al_loss: 2.5860 - val_accuracy: 0.5775
Epoch 41/50
625/625 [==============================] - 825s 1s/step - loss: 0.0491 - accuracy: 0.9859 - v
al_loss: 2.5932 - val_accuracy: 0.5774
Epoch 42/50
625/625 [==============================] - 786s 1s/step - loss: 0.0493 - accuracy: 0.9850 - v
al_loss: 2.6111 - val_accuracy: 0.5742
Epoch 43/50
625/625 [==============================] - 797s 1s/step - loss: 0.0449 - accuracy: 0.9871 - v
al_loss: 2.6076 - val_accuracy: 0.5728
Epoch 44/50
625/625 [==============================] - 716s 1s/step - loss: 0.0445 - accuracy: 0.9868 - v
al_loss: 2.6001 - val_accuracy: 0.5772
```

```
Epoch 45/50
625/625 [==============================] - 516s 826ms/step - loss: 0.0400 - accuracy: 0.9884
- val_loss: 2.6079 - val_accuracy: 0.5748
Epoch 46/50
625/625 [==============================] - 125s 199ms/step - loss: 0.0418 - accuracy: 0.9876
- val_loss: 2.6202 - val_accuracy: 0.5782
Epoch 47/50
625/625 [==============================] - 139s 222ms/step - loss: 0.0423 - accuracy: 0.9868
- val_loss: 2.5911 - val_accuracy: 0.5774
Epoch 48/50
625/625 [==============================] - 130s 207ms/step - loss: 0.0422 - accuracy: 0.9867
- val_loss: 2.6652 - val_accuracy: 0.5705
Epoch 49/50
625/625 [==============================] - 122s 195ms/step - loss: 0.0362 - accuracy: 0.9898
- val_loss: 2.6328 - val_accuracy: 0.5725
Epoch 50/50
625/625 [==============================] - 348s 557ms/step - loss: 0.0370 - accuracy: 0.9890
- val_loss: 2.6358 - val_accuracy: 0.5751
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\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': 'efficientNetV2', 'Epochs': 50, 'Batch Size': 64, 'Train Loss': 0.041787739843
13011, 'Val Loss': 2.6202239990234375, 'Train Acc': 0.9876000285148621, 'Val Acc': 0.57819998
26431274, '[Train - Val] Acc': 0.4094000458717346}
```





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
```

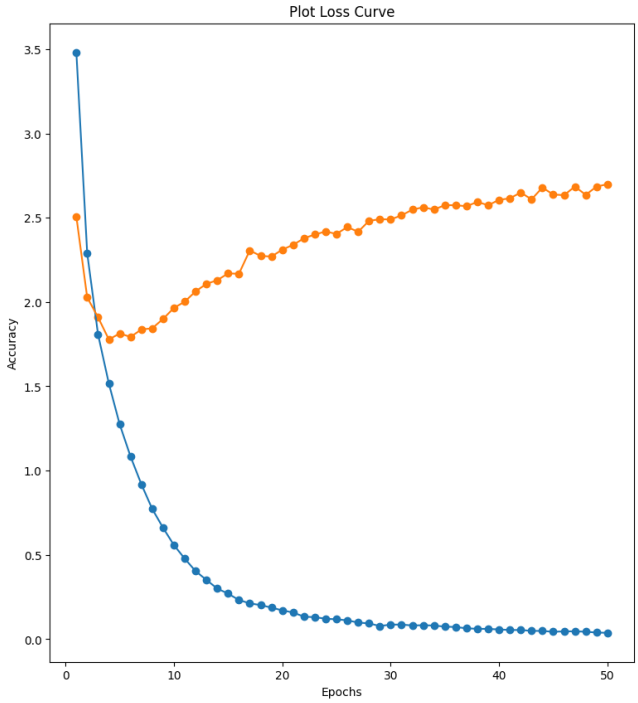
```
Epoch 1/50
625/625 [==============================] - 88s 116ms/step - loss: 3.4810 - accuracy: 0.1921 -
val_loss: 2.5065 - val_accuracy: 0.3564
Epoch 2/50
625/625 [==============================] - 114s 183ms/step - loss: 2.2895 - accuracy: 0.3965
- val_loss: 2.0273 - val_accuracy: 0.4523
Epoch 3/50
625/625 [==============================] - 392s 628ms/step - loss: 1.8082 - accuracy: 0.5042
- val_loss: 1.9098 - val_accuracy: 0.4948
Epoch 4/50
625/625 [==============================] - 495s 787ms/step - loss: 1.5162 - accuracy: 0.5713
- val_loss: 1.7776 - val_accuracy: 0.5269
Epoch 5/50
625/625 [==============================] - 431s 690ms/step - loss: 1.2741 - accuracy: 0.6320
- val_loss: 1.8137 - val_accuracy: 0.5264
Epoch 6/50
625/625 [==============================] - 576s 922ms/step - loss: 1.0835 - accuracy: 0.6798
- val_loss: 1.7929 - val_accuracy: 0.5405
Epoch 7/50
625/625 [==============================] - 514s 824ms/step - loss: 0.9189 - accuracy: 0.7229
- val_loss: 1.8373 - val_accuracy: 0.5417
Epoch 8/50
625/625 [==============================] - 497s 796ms/step - loss: 0.7727 - accuracy: 0.7640
- val_loss: 1.8441 - val_accuracy: 0.5495
Epoch 9/50
625/625 [==============================] - 518s 829ms/step - loss: 0.6609 - accuracy: 0.7943
- val_loss: 1.8996 - val_accuracy: 0.5500
Epoch 10/50
625/625 [==============================] - 296s 474ms/step - loss: 0.5584 - accuracy: 0.8285
- val_loss: 1.9650 - val_accuracy: 0.5482
Epoch 11/50
625/625 [==============================] - 107s 171ms/step - loss: 0.4777 - accuracy: 0.8492
- val_loss: 2.0020 - val_accuracy: 0.5532
Epoch 12/50
625/625 [==============================] - 106s 170ms/step - loss: 0.4041 - accuracy: 0.8733
- val_loss: 2.0631 - val_accuracy: 0.5578
Epoch 13/50
625/625 [==============================] - 106s 170ms/step - loss: 0.3517 - accuracy: 0.8897
- val_loss: 2.1077 - val_accuracy: 0.5598
Epoch 14/50
625/625 [==============================] - 108s 172ms/step - loss: 0.3005 - accuracy: 0.9064
- val_loss: 2.1293 - val_accuracy: 0.5620
Epoch 15/50
625/625 [==============================] - 109s 174ms/step - loss: 0.2704 - accuracy: 0.9141
- val_loss: 2.1697 - val_accuracy: 0.5640
Epoch 16/50
625/625 [==============================] - 105s 168ms/step - loss: 0.2325 - accuracy: 0.9266
- val_loss: 2.1675 - val_accuracy: 0.5647
Epoch 17/50
625/625 [==============================] - 106s 170ms/step - loss: 0.2126 - accuracy: 0.9324
- val_loss: 2.3058 - val_accuracy: 0.5511
Epoch 18/50
625/625 [==============================] - 105s 168ms/step - loss: 0.2026 - accuracy: 0.9371
- val_loss: 2.2735 - val_accuracy: 0.5593
Epoch 19/50
625/625 [==============================] - 105s 169ms/step - loss: 0.1873 - accuracy: 0.9412
- val_loss: 2.2697 - val_accuracy: 0.5606
Epoch 20/50
625/625 [==============================] - 107s 171ms/step - loss: 0.1712 - accuracy: 0.9475
- val_loss: 2.3105 - val_accuracy: 0.5652
Epoch 21/50
625/625 [==============================] - 108s 172ms/step - loss: 0.1584 - accuracy: 0.9506
- val_loss: 2.3393 - val_accuracy: 0.5652
Epoch 22/50
625/625 [==============================] - 106s 169ms/step - loss: 0.1338 - accuracy: 0.9589
- val_loss: 2.3777 - val_accuracy: 0.5658
```

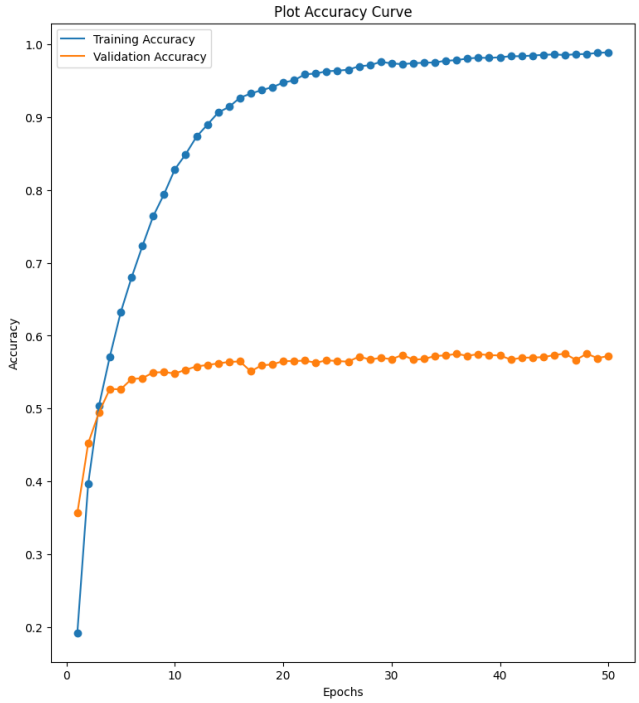
```
Epoch 23/50
625/625 [==============================] - 107s 171ms/step - loss: 0.1307 - accuracy: 0.9599
- val_loss: 2.4006 - val_accuracy: 0.5629
Epoch 24/50
625/625 [==============================] - 106s 169ms/step - loss: 0.1217 - accuracy: 0.9631
- val_loss: 2.4181 - val_accuracy: 0.5662
Epoch 25/50
625/625 [==============================] - 104s 167ms/step - loss: 0.1179 - accuracy: 0.9638
- val_loss: 2.4040 - val_accuracy: 0.5652
Epoch 26/50
625/625 [==============================] - 110s 177ms/step - loss: 0.1120 - accuracy: 0.9652
- val_loss: 2.4446 - val_accuracy: 0.5644
Epoch 27/50
625/625 [==============================] - 106s 169ms/step - loss: 0.0988 - accuracy: 0.9699
- val_loss: 2.4166 - val_accuracy: 0.5713
Epoch 28/50
625/625 [==============================] - 105s 168ms/step - loss: 0.0938 - accuracy: 0.9713
- val_loss: 2.4812 - val_accuracy: 0.5673
Epoch 29/50
625/625 [==============================] - 108s 173ms/step - loss: 0.0782 - accuracy: 0.9760
- val_loss: 2.4912 - val_accuracy: 0.5696
Epoch 30/50
625/625 [==============================] - 106s 169ms/step - loss: 0.0865 - accuracy: 0.9738
- val_loss: 2.4898 - val_accuracy: 0.5678
Epoch 31/50
625/625 [==============================] - 107s 171ms/step - loss: 0.0868 - accuracy: 0.9728
- val_loss: 2.5138 - val_accuracy: 0.5736
Epoch 32/50
625/625 [==============================] - 108s 173ms/step - loss: 0.0815 - accuracy: 0.9742
- val_loss: 2.5479 - val_accuracy: 0.5670
Epoch 33/50
625/625 [==============================] - 106s 169ms/step - loss: 0.0830 - accuracy: 0.9747
- val_loss: 2.5619 - val_accuracy: 0.5681
Epoch 34/50
625/625 [==============================] - 107s 171ms/step - loss: 0.0811 - accuracy: 0.9752
- val_loss: 2.5483 - val_accuracy: 0.5721
Epoch 35/50
625/625 [==============================] - 106s 170ms/step - loss: 0.0747 - accuracy: 0.9775
- val_loss: 2.5746 - val_accuracy: 0.5730
Epoch 36/50
625/625 [==============================] - 105s 168ms/step - loss: 0.0724 - accuracy: 0.9783
- val_loss: 2.5739 - val_accuracy: 0.5750
Epoch 37/50
625/625 [==============================] - 109s 175ms/step - loss: 0.0645 - accuracy: 0.9805
- val_loss: 2.5684 - val_accuracy: 0.5724
Epoch 38/50
625/625 [==============================] - 108s 173ms/step - loss: 0.0620 - accuracy: 0.9816
- val_loss: 2.5937 - val_accuracy: 0.5748
Epoch 39/50
625/625 [==============================] - 107s 171ms/step - loss: 0.0611 - accuracy: 0.9814
- val_loss: 2.5740 - val_accuracy: 0.5733
Epoch 40/50
625/625 [==============================] - 108s 172ms/step - loss: 0.0578 - accuracy: 0.9823
- val_loss: 2.6060 - val_accuracy: 0.5729
Epoch 41/50
625/625 [==============================] - 104s 166ms/step - loss: 0.0548 - accuracy: 0.9837
- val_loss: 2.6149 - val_accuracy: 0.5672
Epoch 42/50
625/625 [==============================] - 103s 165ms/step - loss: 0.0545 - accuracy: 0.9837
- val_loss: 2.6495 - val_accuracy: 0.5697
Epoch 43/50
625/625 [==============================] - 108s 172ms/step - loss: 0.0495 - accuracy: 0.9847
- val_loss: 2.6090 - val_accuracy: 0.5701
Epoch 44/50
625/625 [==============================] - 105s 168ms/step - loss: 0.0491 - accuracy: 0.9855
- val_loss: 2.6789 - val_accuracy: 0.5705
```

```
Epoch 45/50
625/625 [==============================] - 104s 167ms/step - loss: 0.0441 - accuracy: 0.9862
- val_loss: 2.6377 - val_accuracy: 0.5735
Epoch 46/50
625/625 [==============================] - 109s 175ms/step - loss: 0.0466 - accuracy: 0.9853
- val_loss: 2.6339 - val_accuracy: 0.5754
Epoch 47/50
625/625 [==============================] - 104s 167ms/step - loss: 0.0467 - accuracy: 0.9864
- val_loss: 2.6847 - val_accuracy: 0.5667
Epoch 48/50
625/625 [==============================] - 106s 169ms/step - loss: 0.0446 - accuracy: 0.9868
- val_loss: 2.6359 - val_accuracy: 0.5753
Epoch 49/50
625/625 [==============================] - 110s 176ms/step - loss: 0.0394 - accuracy: 0.9882
- val_loss: 2.6847 - val_accuracy: 0.5692
Epoch 50/50
625/625 [==============================] - 104s 167ms/step - loss: 0.0380 - accuracy: 0.9890
- val_loss: 2.6987 - val_accuracy: 0.5722
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\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': 50, 'Batch Size': 64, 'Train Loss': 0.046598475
42643547, 'Val Loss': 2.6339080333709717, 'Train Acc': 0.9853000044822693, 'Val Acc': 0.57539
99948501587, '[Train - Val] Acc': 0.4099000096321106}
```





Observation

Comparing the EfficientNetV2 model with augmentation and model without augmentation, we can see that the model decreased slightly but the training and validation loss decreased which suggest the model fitting more generalise to the augmented data.

EfficientNetV2B0 model with CutMix 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)
efficientNetCutMixModel = Model(
    inputs=inputs, outputs=x, name="efficientNetV2CutMix")
efficientNetCutMixModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                                loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:

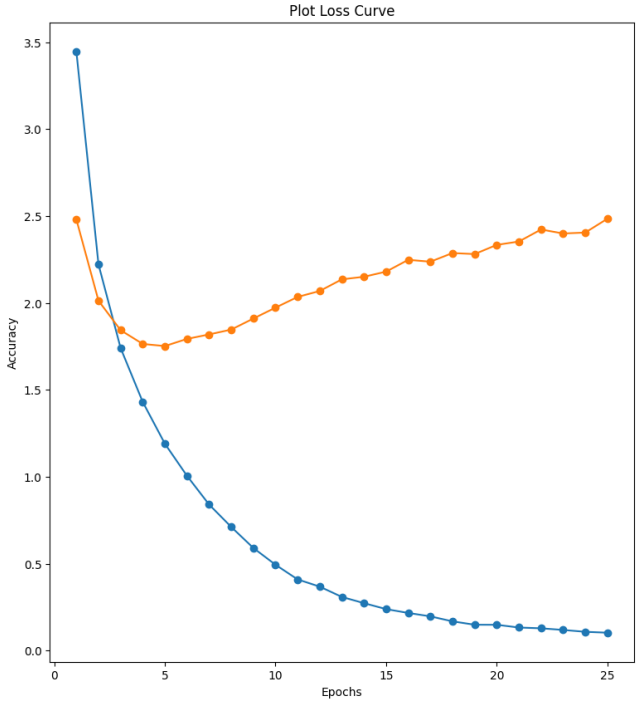
```
efficientNetCutMixModelHistory = efficientNetCutMixModel.fit(train_ds_cutmix, epochs=50,
                                                             validation_data=(x_val, y_val), batch_size=BAT
```

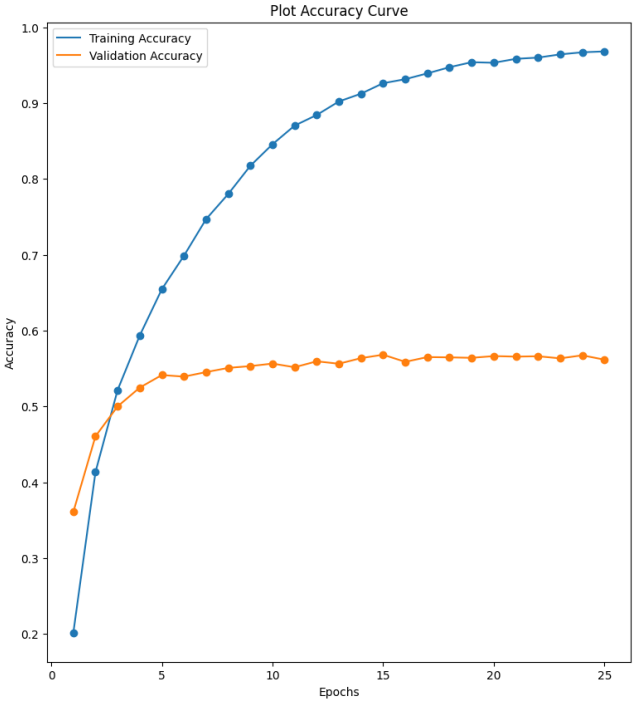
```
Epoch 1/50
625/625 [==============================] - 107s 112ms/step - loss: 3.4472 - accuracy: 0.2016
- val_loss: 2.4806 - val_accuracy: 0.3609
Epoch 2/50
625/625 [==============================] - 86s 137ms/step - loss: 2.2219 - accuracy: 0.4133 -
val_loss: 2.0129 - val_accuracy: 0.4606
Epoch 3/50
625/625 [==============================] - 92s 147ms/step - loss: 1.7405 - accuracy: 0.5216 -
val_loss: 1.8431 - val_accuracy: 0.5000
Epoch 4/50
625/625 [==============================] - 159s 255ms/step - loss: 1.4303 - accuracy: 0.5935
- val_loss: 1.7641 - val_accuracy: 0.5248
Epoch 5/50
625/625 [==============================] - 93s 145ms/step - loss: 1.1901 - accuracy: 0.6546 -
val_loss: 1.7518 - val_accuracy: 0.5414
Epoch 6/50
625/625 [==============================] - 71s 114ms/step - loss: 1.0052 - accuracy: 0.6988 -
val_loss: 1.7931 - val_accuracy: 0.5393
Epoch 7/50
625/625 [==============================] - 74s 119ms/step - loss: 0.8399 - accuracy: 0.7470 -
val_loss: 1.8187 - val_accuracy: 0.5453
Epoch 8/50
625/625 [==============================] - 80s 128ms/step - loss: 0.7110 - accuracy: 0.7801 -
val_loss: 1.8467 - val_accuracy: 0.5508
Epoch 9/50
625/625 [==============================] - 81s 129ms/step - loss: 0.5900 - accuracy: 0.8174 -
val_loss: 1.9102 - val_accuracy: 0.5532
Epoch 10/50
625/625 [==============================] - 82s 131ms/step - loss: 0.4937 - accuracy: 0.8458 -
val_loss: 1.9729 - val_accuracy: 0.5565
Epoch 11/50
625/625 [==============================] - 98s 157ms/step - loss: 0.4092 - accuracy: 0.8704 -
val_loss: 2.0346 - val_accuracy: 0.5517
Epoch 12/50
625/625 [==============================] - 85s 136ms/step - loss: 0.3676 - accuracy: 0.8844 -
val_loss: 2.0687 - val_accuracy: 0.5595
Epoch 13/50
625/625 [==============================] - 85s 136ms/step - loss: 0.3084 - accuracy: 0.9022 -
val_loss: 2.1357 - val_accuracy: 0.5563
Epoch 14/50
625/625 [==============================] - 79s 126ms/step - loss: 0.2726 - accuracy: 0.9125 -
val_loss: 2.1506 - val_accuracy: 0.5637
Epoch 15/50
625/625 [==============================] - 74s 119ms/step - loss: 0.2383 - accuracy: 0.9263 -
val_loss: 2.1801 - val_accuracy: 0.5682
Epoch 16/50
625/625 [==============================] - 78s 124ms/step - loss: 0.2162 - accuracy: 0.9316 -
val_loss: 2.2481 - val_accuracy: 0.5588
Epoch 17/50
625/625 [==============================] - 82s 131ms/step - loss: 0.1967 - accuracy: 0.9393 -
val_loss: 2.2373 - val_accuracy: 0.5651
Epoch 18/50
625/625 [==============================] - 73s 117ms/step - loss: 0.1679 - accuracy: 0.9474 -
val_loss: 2.2870 - val_accuracy: 0.5647
Epoch 19/50
625/625 [==============================] - 80s 128ms/step - loss: 0.1486 - accuracy: 0.9541 -
val_loss: 2.2812 - val_accuracy: 0.5641
Epoch 20/50
625/625 [==============================] - 89s 142ms/step - loss: 0.1482 - accuracy: 0.9533 -
val_loss: 2.3341 - val_accuracy: 0.5664
Epoch 21/50
625/625 [==============================] - 87s 139ms/step - loss: 0.1324 - accuracy: 0.9585 -
val_loss: 2.3532 - val_accuracy: 0.5657
Epoch 22/50
625/625 [==============================] - 87s 139ms/step - loss: 0.1280 - accuracy: 0.9601 -
val_loss: 2.4225 - val_accuracy: 0.5662
```

```
Epoch 23/50
625/625 [==============================] - 79s 126ms/step - loss: 0.1185 - accuracy: 0.9643 -
val_loss: 2.3996 - val_accuracy: 0.5634
Epoch 24/50
625/625 [==============================] - 71s 114ms/step - loss: 0.1077 - accuracy: 0.9671 -
val_loss: 2.4049 - val_accuracy: 0.5674
Epoch 25/50
625/625 [==============================] - 67s 108ms/step - loss: 0.1027 - accuracy: 0.9683 -
val_loss: 2.4845 - val_accuracy: 0.5617
```

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

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\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': 'efficientNetV2CutMix', 'Epochs': 25, 'Batch Size': 64, 'Train Loss': 0.238335
69884300232, 'Val Loss': 2.1801090240478516, 'Train Acc': 0.9263250231742859, 'Val Acc': 0.56
81999921798706, '[Train - Val] Acc': 0.3581250309944153}
```





Observation

We can see that the validation and training accuracy decrease slightly compared to the model that as train using the unaugmented dataset but the model trained with the cutmix model manage to be more generalise as the validation and training loss decreased

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
7	efficientNetV2	50	64	0.041788	2.620224	0.987600	0.578200	0.409400
4	efficientNetV2Aug	50	64	0.046598	2.633908	0.985300	0.575400	0.409900
18	CustomVGG16Aug	50	64	2.817095	3.320672	0.651400	0.569000	0.082400
19	CustomVGG16	50	64	2.782844	3.370079	0.661150	0.562200	0.098950
6	CustomVGG	50	64	0.033576	3.923757	0.989400	0.505400	0.484000
8	CustomVGG_L2	31	64	1.720771	3.421193	0.803300	0.493800	0.309500
9	CustomVGGAug	40	64	0.114501	3.486695	0.962800	0.492200	0.470600
10	CustomVGG_L1	47	64	3.280015	3.338737	0.440800	0.439300	0.001500
12	CustomResNetAug	29	64	2.467899	4.110264	0.655325	0.385300	0.221400
13	CustomResNet	23	64	2.461369	3.677287	0.600800	0.379400	0.221400
14	conv2D	18	64	2.195034	2.905279	0.463550	0.321900	0.141650
11	conv2DAug	17	64	2.266827	2.876986	0.410850	0.316000	0.094850
15	CustomResNetDrop	49	64	3.593024	3.583902	0.289500	0.302000	-0.012500
16	baseline	23	64	2.624983	3.550842	0.331725	0.210900	0.120825
17	baselineAug	19	64	2.912171	3.482317	0.275075	0.203000	0.072075

It seems like the efficientNetV2 model [pre-trained] performed the best. However, out of all the custom models, the VGG16 and VGG10 model performed the best. Therefore we will be making model improvements to the customVGG16 model and comparing the results to see if it is better than the efficientNetV2 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, y_train, validation_data=(x_val, y_val), epochs=60, batch_size=BATCH_SIZE, callb
        EarlyStopping(monitor="val_accuracy", patience=10,
                      restore_best_weights=True)
    ]
)
VGG16Tuner.results_summary(num_trials=3)
```

```
Trial 3 Complete [00h 37m 03s]
val_accuracy: 0.5857999920845032

Best val_accuracy So Far: 0.6098999977111816
Total elapsed time: 03h 12m 54s
INFO:tensorflow:Oracle triggered exit
Results summary
Results in .\cifar100_vgg16
Showing 3 best trials
<keras_tuner.engine.objective.Objective object at 0x000001D2052CC130>
Trial summary
Hyperparameters:
weight_decay: 0.001070054939430907
learning_rate: 0.022339176031524674
Score: 0.6098999977111816
Trial summary
Hyperparameters:
weight_decay: 0.0011071608759999094
learning_rate: 0.010553804410088815
Score: 0.5857999920845032
Trial summary
Hyperparameters:
weight_decay: 0.002211330377554398
learning_rate: 0.0058924994283040275
Score: 0.5669000148773193
```

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/customVGG16 - Final')
```

```
efficientNetModel.save('models/efficientNetV2')
```

```
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 5 of 13). These functions will not be directly callable after loading.
INFO:tensorflow:Assets written to: models/customVGG16 - Final\assets
INFO:tensorflow:Assets written to: models/customVGG16 - Final\assets
```

In [ ]:
```
vgg16_model.save('models/customVGG16 - Final.h5')
efficientNetModel.save('models/efficientNetV2.h5')
```

Initiate model after tuning and saving

In [ ]:
```
final_model = tf.keras.models.load_model('models/customVGG16 - 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, 100)               51300

=================================================================
Total params: 15,047,595
Trainable params: 15,038,116
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)
```

```
313/313 [==============================] - 7s 17ms/step - loss: 2.6634 - accuracy: 0.6155
[2.663435697555542, 0.6154999732971191]
```

Out[ ]:

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

```
313/313 [==============================] - 4s 13ms/step
```

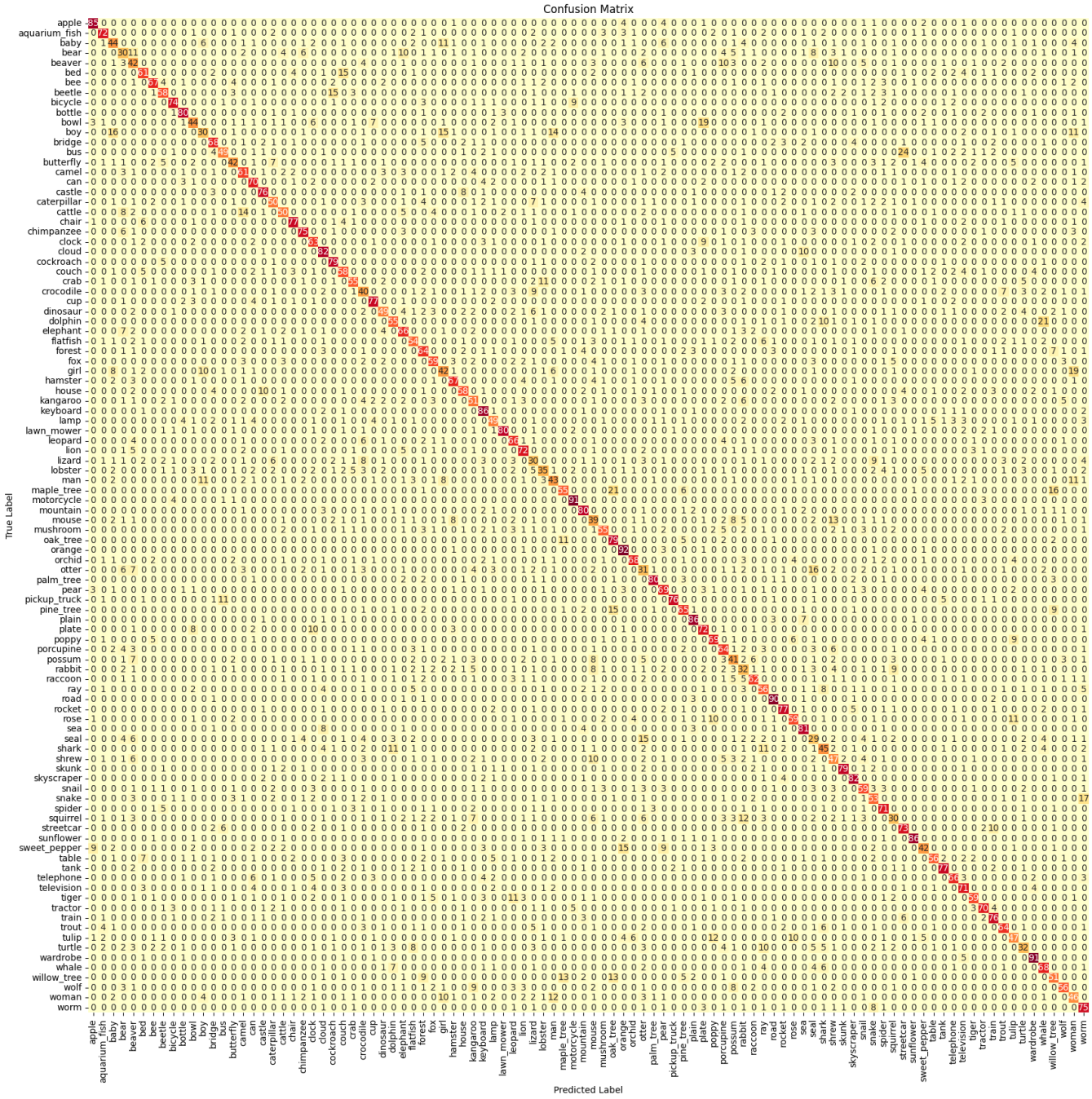
In [ ]:
```
report = classification_report(
    np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1), target_names=class_labels.values()
)
print(report)
```

	precision	recall	f1-score	support
apple	0.83	0.85	0.84	100
aquarium_fish	0.77	0.72	0.75	100
baby	0.48	0.44	0.46	100
bear	0.34	0.30	0.32	100
beaver	0.32	0.42	0.36	100
bed	0.66	0.61	0.64	100
bee	0.77	0.67	0.72	100
beetle	0.65	0.58	0.61	100
bicycle	0.82	0.74	0.78	100
bottle	0.81	0.80	0.80	100
bowl	0.54	0.44	0.49	100
boy	0.46	0.30	0.36	100
bridge	0.66	0.68	0.67	100
bus	0.69	0.49	0.57	100
butterfly	0.58	0.42	0.49	100
camel	0.59	0.61	0.60	100
can	0.63	0.70	0.66	100
castle	0.75	0.76	0.76	100
caterpillar	0.56	0.50	0.53	100
cattle	0.60	0.50	0.55	100
chair	0.80	0.77	0.79	100
chimpanzee	0.77	0.75	0.76	100
clock	0.56	0.63	0.59	100
cloud	0.65	0.82	0.72	100
cockroach	0.71	0.79	0.75	100
couch	0.53	0.58	0.56	100
crab	0.68	0.55	0.61	100
crocodile	0.38	0.40	0.39	100
cup	0.77	0.77	0.77	100
dinosaur	0.64	0.49	0.56	100
dolphin	0.62	0.55	0.59	100
elephant	0.57	0.66	0.61	100
flatfish	0.57	0.54	0.55	100
forest	0.51	0.64	0.57	100
fox	0.66	0.59	0.62	100
girl	0.44	0.42	0.43	100
hamster	0.69	0.67	0.68	100
house	0.69	0.58	0.63	100
kangaroo	0.48	0.51	0.49	100
keyboard	0.64	0.86	0.73	100
lamp	0.56	0.49	0.52	100
lawn_mower	0.92	0.80	0.86	100
leopard	0.60	0.66	0.63	100
lion	0.67	0.72	0.69	100
lizard	0.29	0.30	0.29	100
lobster	0.43	0.35	0.39	100
man	0.44	0.43	0.43	100
maple_tree	0.65	0.55	0.60	100
motorcycle	0.81	0.91	0.85	100
mountain	0.71	0.80	0.75	100
mouse	0.35	0.39	0.37	100
mushroom	0.67	0.55	0.60	100
oak_tree	0.60	0.79	0.68	100
orange	0.68	0.92	0.78	100
orchid	0.72	0.68	0.70	100
otter	0.26	0.31	0.28	100
palm_tree	0.89	0.80	0.84	100
pear	0.59	0.69	0.64	100
pickup_truck	0.89	0.76	0.82	100
pine_tree	0.66	0.65	0.65	100
plain	0.79	0.86	0.82	100
plate	0.59	0.72	0.65	100
poppy	0.66	0.69	0.68	100
porcupine	0.50	0.64	0.56	100

```
       possum       0.39      0.41      0.40       100
       rabbit       0.27      0.32      0.29       100
      raccoon       0.71      0.62      0.66       100
          ray       0.48      0.56      0.52       100
         road       0.81      0.90      0.85       100
       rocket       0.72      0.77      0.74       100
         rose       0.74      0.59      0.66       100
          sea       0.70      0.81      0.75       100
         seal       0.27      0.29      0.28       100
        shark       0.51      0.45      0.48       100
        shrew       0.36      0.47      0.41       100
        skunk       0.87      0.79      0.83       100
   skyscraper       0.76      0.82      0.79       100
        snail       0.60      0.59      0.60       100
        snake       0.46      0.53      0.49       100
       spider       0.63      0.71      0.67       100
     squirrel       0.36      0.30      0.33       100
    streetcar       0.63      0.73      0.68       100
    sunflower       0.91      0.86      0.88       100
sweet_pepper       0.55      0.42      0.47       100
        table       0.76      0.56      0.64       100
         tank       0.81      0.77      0.79       100
    telephone       0.73      0.66      0.69       100
   television       0.61      0.71      0.66       100
        tiger       0.78      0.59      0.67       100
      tractor       0.80      0.70      0.74       100
        train       0.70      0.76      0.73       100
        trout       0.70      0.64      0.67       100
        tulip       0.55      0.47      0.51       100
       turtle       0.48      0.32      0.39       100
     wardrobe       0.82      0.91      0.86       100
        whale       0.61      0.68      0.64       100
  willow_tree       0.53      0.51      0.52       100
         wolf       0.74      0.56      0.64       100
        woman       0.43      0.46      0.45       100
         worm       0.54      0.75      0.63       100

     accuracy                           0.62     10000
    macro avg       0.62      0.62      0.61     10000
 weighted avg       0.62      0.62      0.61     10000
```

We can see from the classification report that the model is very good at identifying and differentiating lawn_mower, sunflower, palm_tree and pickup_truck but not as good as predicting lizard, otter, rabbit and seal. Let's do a mini error analysis and find out why.

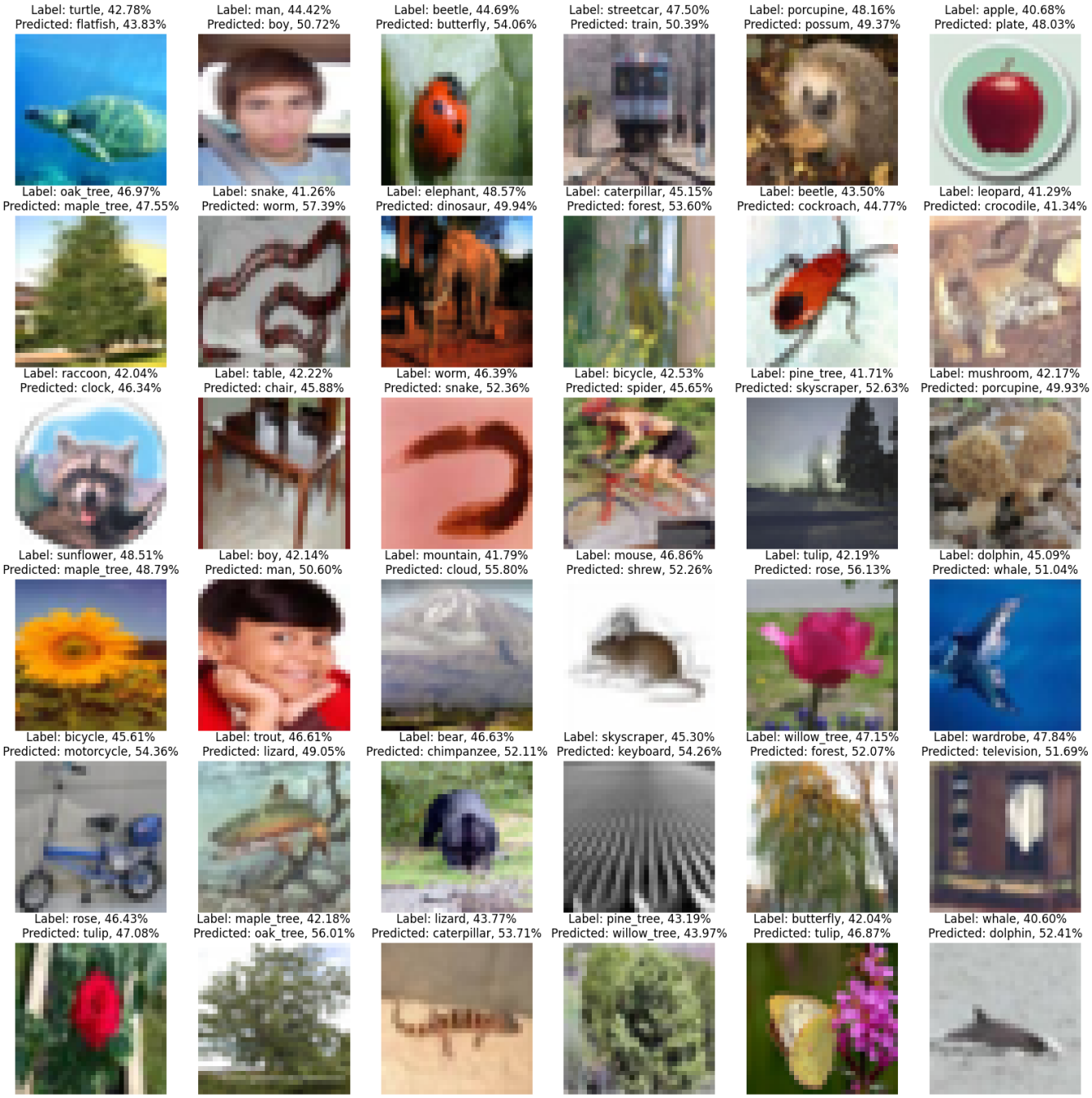
In [ ]:
```
plt.figure(1, figsize=(20, 20))
plt.title("Confusion Matrix")
sns.heatmap(tf.math.confusion_matrix(
    np.argmax(y_test, axis=1),
    np.argmax(y_pred, axis=1),
    num_classes=NUM_CLASS,
    dtype=tf.dtypes.int32,
    name=None
), annot=True, fmt="", cbar=False, cmap="YlOrRd", yticklabels=class_labels.values(), 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 4
      That images looks a bunch of pixels [Due to low resolutions] and it is hard to distinguish the features.
2. Similar features
    - Example: Row 4 Column 4
      The model predicted the values of a mouse and shrew wrongly. This is likely due to both mouse and shrew being mammals and having similar features like having a tail and big ears.
3. Color similarity
    - Example: Row 4 Column 5
      Both roses and tulips come in many different colors. The model is likely confused by the color and therefore mixed up the prediction
4. Background noise

- Example: Row 6 Column 5
  Even though the label is butterfly, almost half the image is a flower. This might have caused the model to predict the flower instead of the butterfly.

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

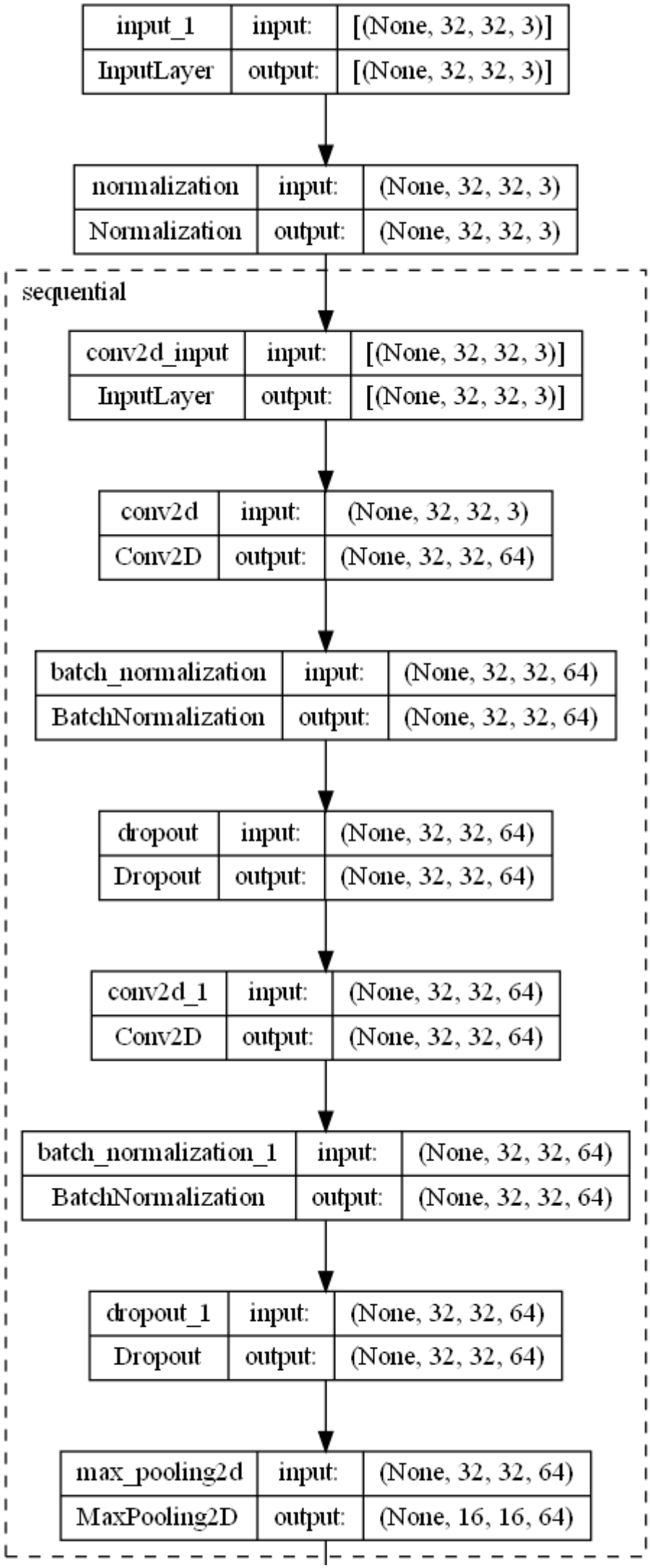
Comparing final_model VS other people's model

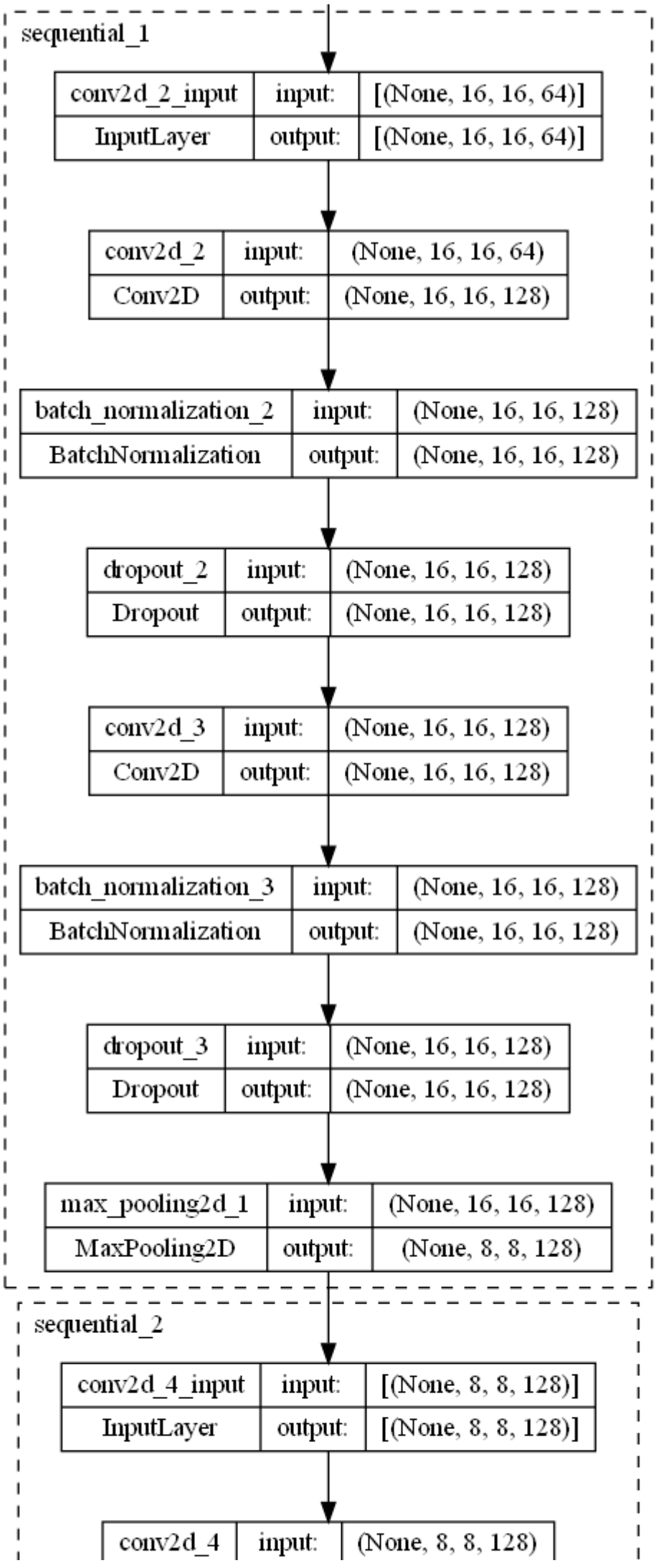
Based on PapersWithCode.com, our final model is ranked 165th out of the 183 models [Not inclusive of ours]. [https://paperswithcode.com/sota/image-classification-on-cifar-100]

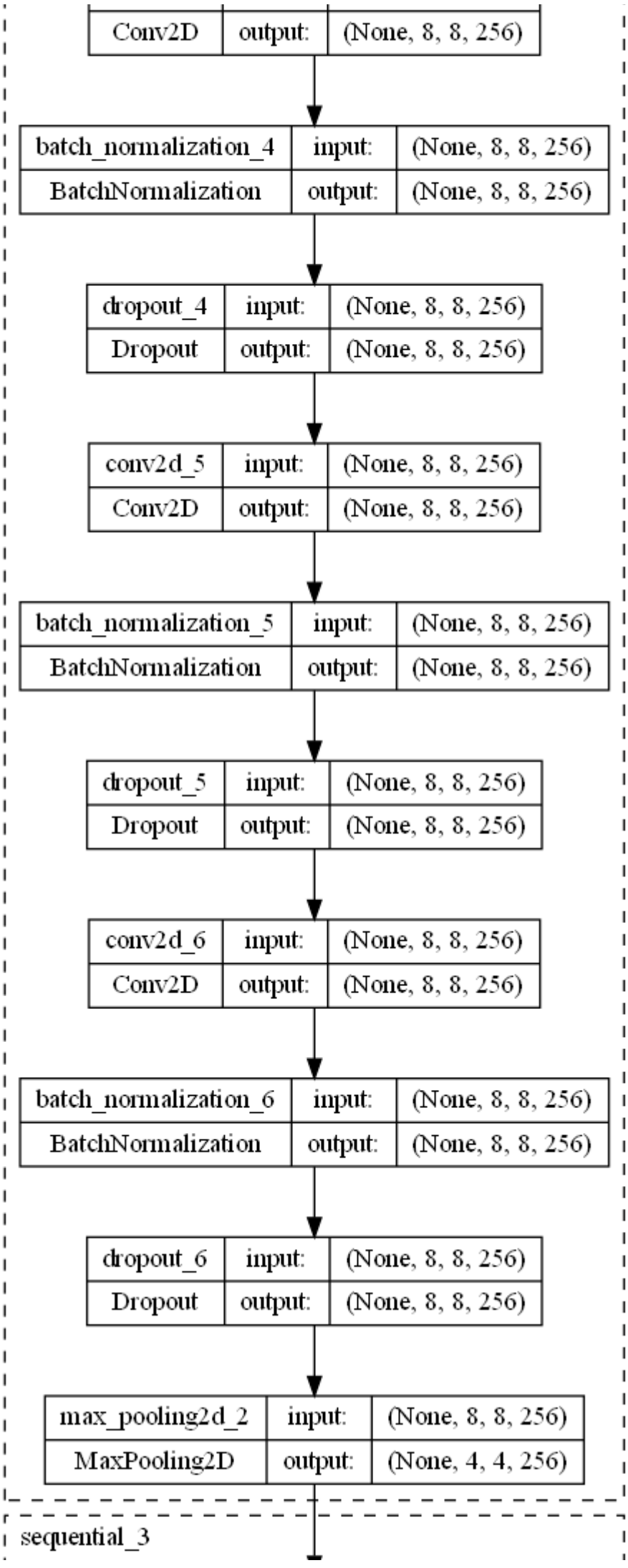
Model Visualisation

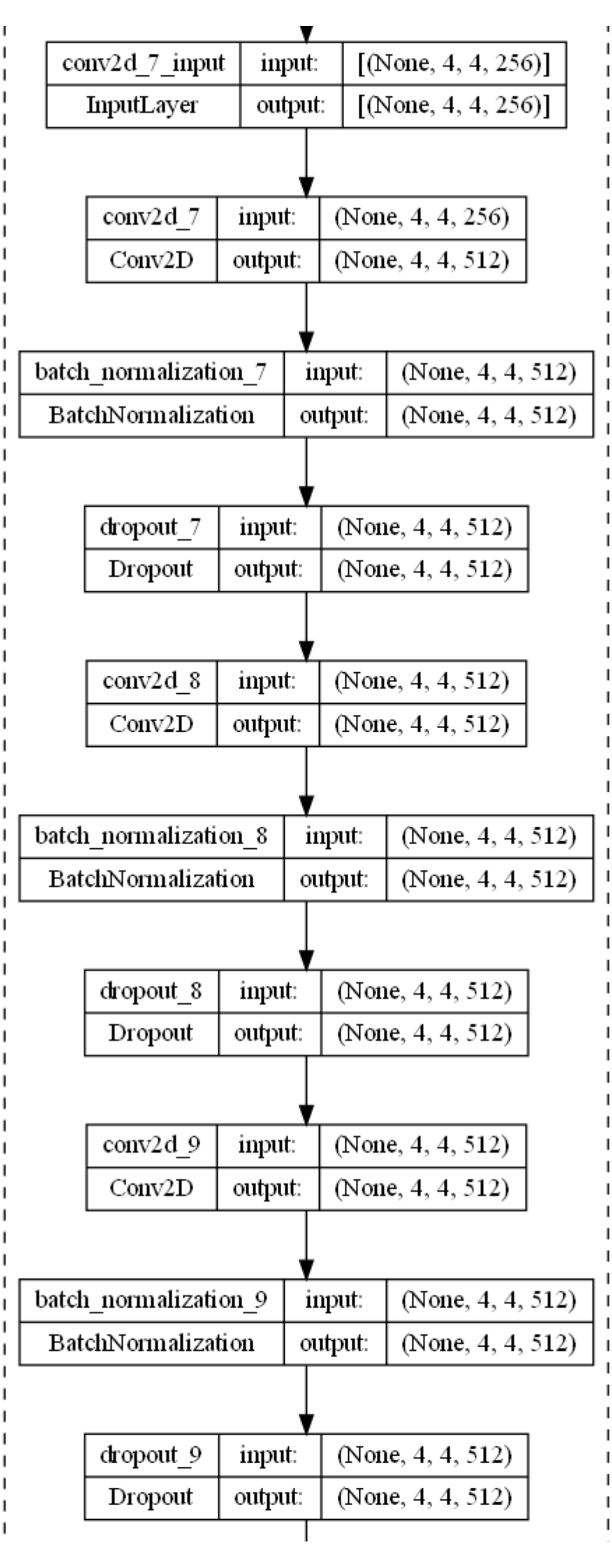
In [ ]:
```
tf.keras.utils.plot_model(final_model, show_shapes=True,
                          expand_nested=True, show_layer_activations=True)
```

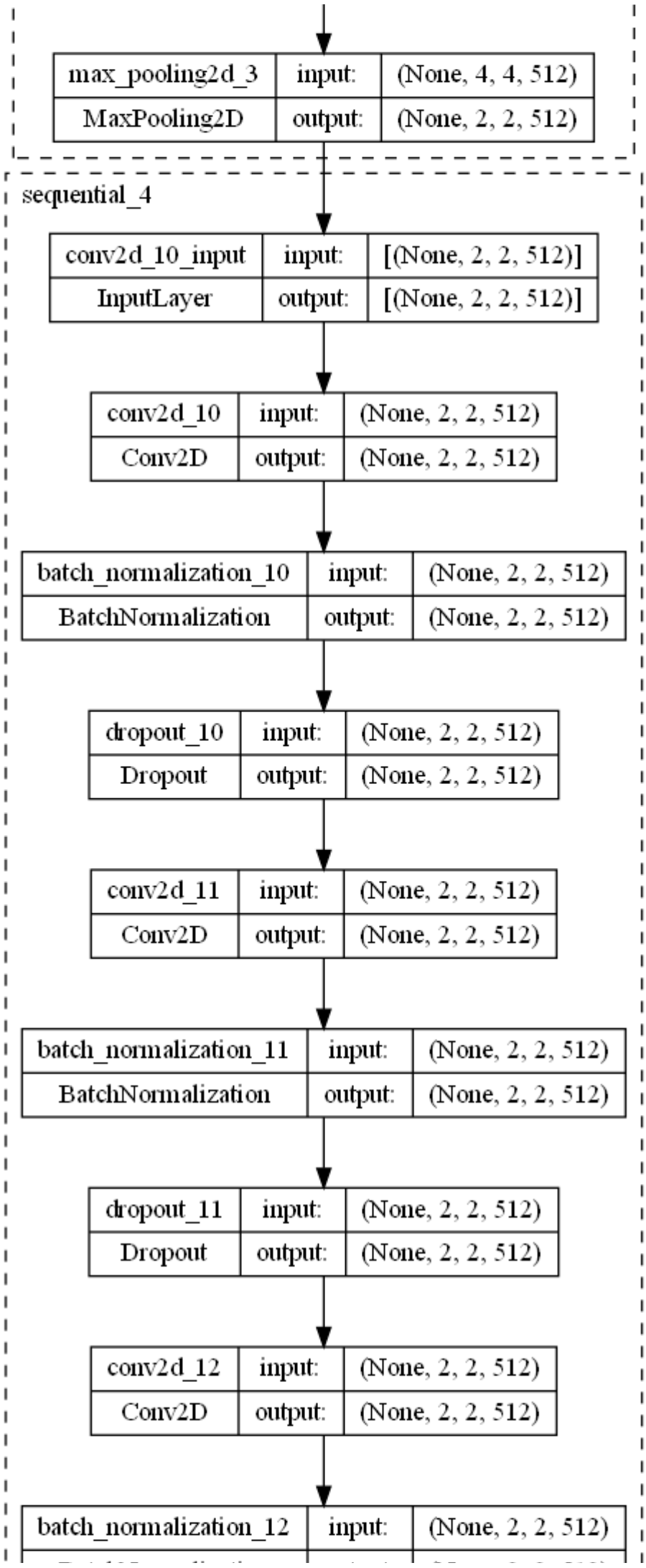
Out[ ]:

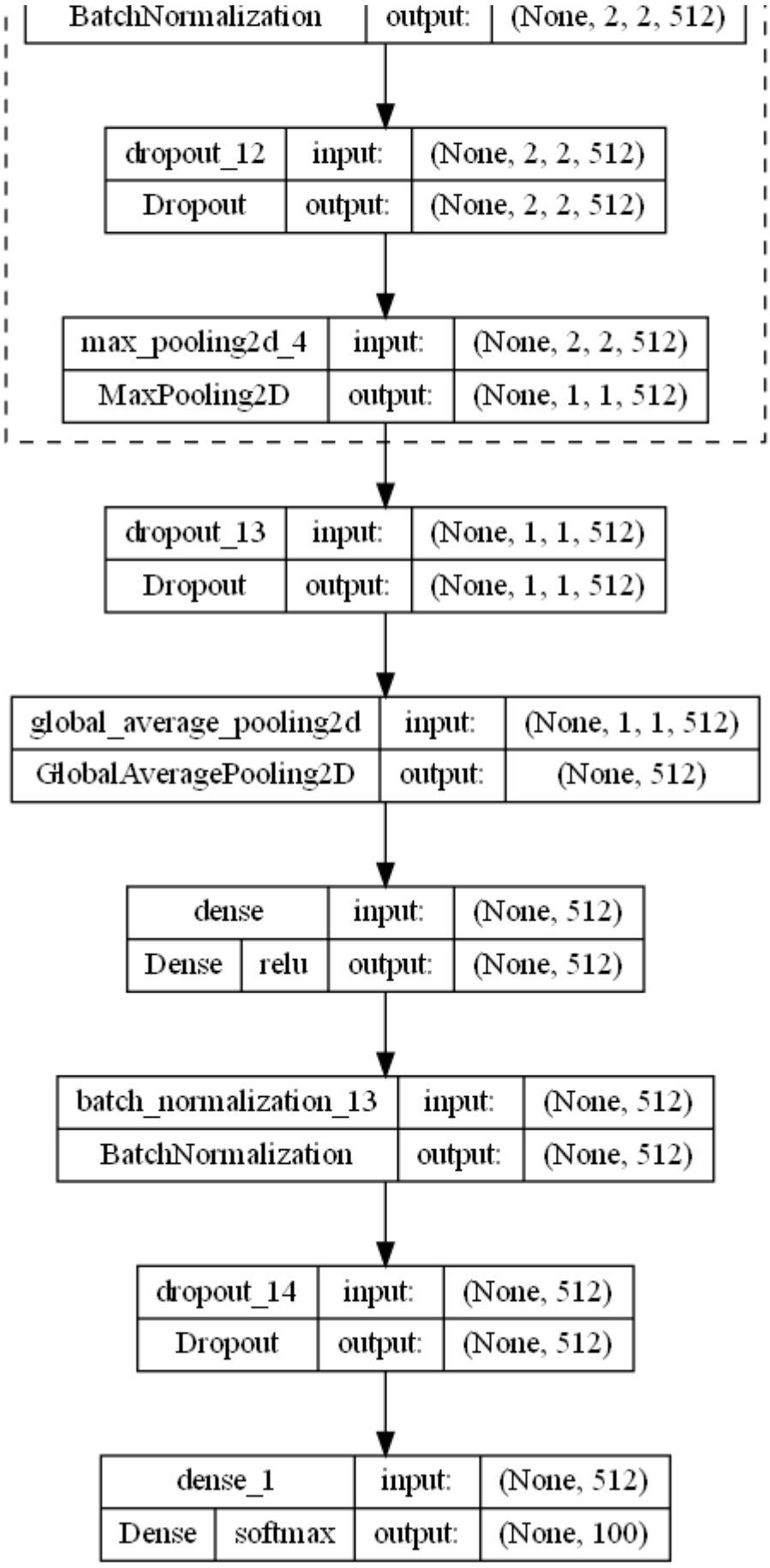

input_1 | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
normalization | input: | (None, 32, 32, 3)
Normalization | output: | (None, 32, 32, 3)
sequential
conv2d_input | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
conv2d | input: | (None, 32, 32, 3)
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

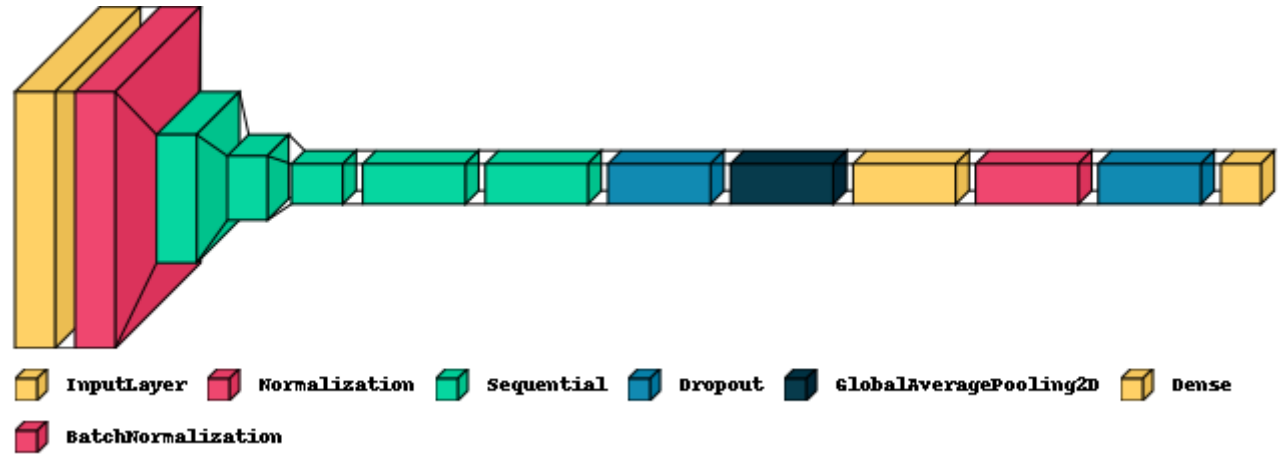

conv2d_7_input | input: | [(None, 4, 4, 256)]
InputLayer | output: | [(None, 4, 4, 256)]
conv2d_7 | input: | (None, 4, 4, 256)
Conv2D | output: | (None, 4, 4, 512)
batch_normalization_7 | input: | (None, 4, 4, 512)
BatchNormalization | output: | (None, 4, 4, 512)
dropout_7 | input: | (None, 4, 4, 512)
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 as well as using other callbacks like ReduceLROnPlateau to make training faster and improve accuracy.