Fashion MNIST Image Classification - Black and White

Name: Soh Hong Yu

Admin Number: P2100775

Class: DAAA/FT/2B/01

Module Code: ST1504 Deep Learning

References (In Harvard format):

1. Team, K. (2022) Keras Documentation: Fashion Mnist Dataset, an alternative to mnist, Keras. Available at: https://keras.io/api/datasets/fashion_mnist/ (Accessed: October 17, 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. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. Han Xiao, Kashif Rasul, Roland Vollgraf. arXiv:1708.07747

Project Objective

Implement an image classifier using a deep learning network

Background Information

With new fashion trends rising daily, being able to categories fashion outfits can be ways to help consumers buy what they want to get.

Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

The original MNIST dataset contains a lot of handwritten digits. Members of the AI/ML/Data Science community love this dataset and use it as a benchmark to validate their algorithms. In fact, MNIST is often the first dataset researchers try. "If it doesn't work on MNIST, it won't work at all", they said. "Well, if it does work on MNIST, it may still fail on others."

Zalando seeks to replace the original MNIST dataset

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
import visualkeras
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
```

In [ ]:
```
# Fix random seed for reproducibility
seed = 88
np.random.seed(seed)
```

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.fashion_mnist.load_data()
```

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 - 48000
Validation set - 12000
Testing set - 10000

In [ ]:
```
train_size = 48000
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 (48000, 28, 28), containing the training data.

y_train: uint8 NumPy array of labels (integers in range 0-9) with shape (48000,) for the training data.

x_val: uint8 NumPy array of grayscale image data with shapes (12000, 28, 28), containing the validation data.

y_val: uint8 NumPy array of labels (integers in range 0-9) with shape (12000,) for the validation data.

x_test: uint8 NumPy array of grayscale image data with shapes (10000, 28, 28), containing the test data.

y_test: uint8 NumPy array of labels (integers in range 0-9) with shape (10000,) for the test data.

There are 10 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 : T-shirt/top
- 1 : Trouser
- 2 : Pullover
- 3 : Dress

- 4 : Coat
- 5 : Sandal
- 6 : Shirt
- 7 : Sneaker
- 8 : Bag
- 9 : Ankle boot

In [ ]:
```
# class labels
class_labels = {
    0: "T-shirt/top",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle boot"
}

NUM_CLASS = 10
```

Each image is a 28x28 image as well as only a single color channel (grayscale image). Therefore, we can set the IMG_SIZE as a tuple (28, 28, 1)

In [ ]:
```
IMG_SIZE = (28, 28, 1)
```

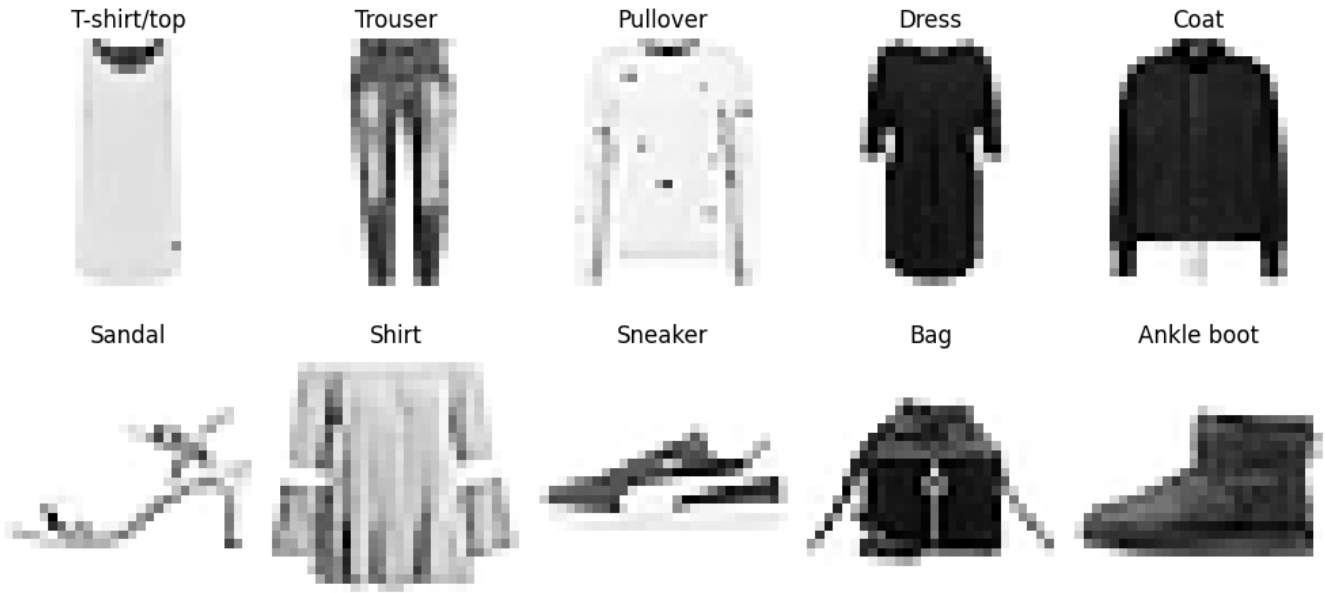
Visualising the Dataset

Let's look at what the images look like.

In [ ]:
```
fig, ax = plt.subplots(2, 5, figsize=(10, 5), tight_layout=True)

for label, subplot in enumerate(ax.ravel()):
    subplot.axis("off")
    subplot.imshow(x_train[y_train == label][np.random.randint(
        0, len(x_train[y_train == label]))], cmap='Greys')
    subplot.set_title(class_labels[label])

plt.show()
```



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

plt.show()
```

Observations

- Zalando is very consistent in the orientation of the images.
  - The shoes always seem to have their tip pointing to the left
- Data augmentation where images can be flipped could be possible to do better prediction
- The images are zoomed in so that the entire image is only the item.

- Based on the selection of images which are 10 from each class [Total 100 samples], there is no clear indication of missed identified items from the dataset. [No extra cleaning/data labelling is required]

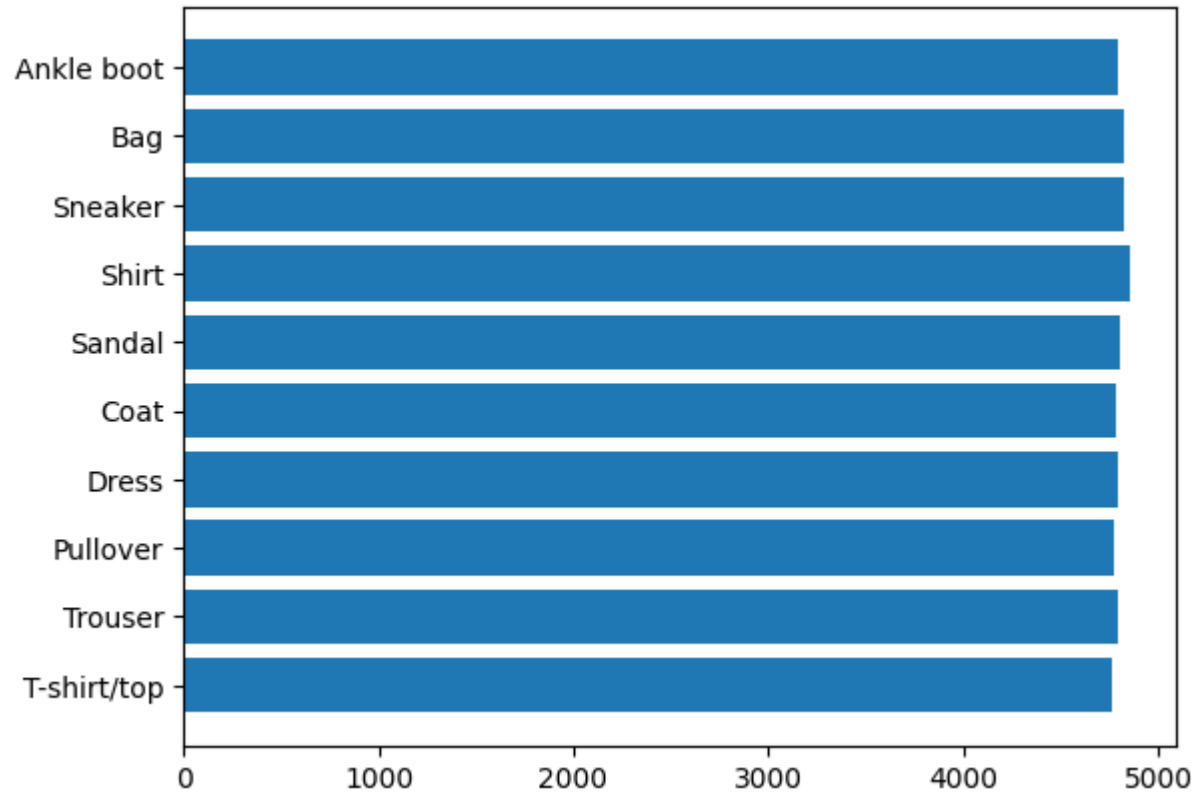
Class Distribution

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}")
```

```
T-shirt/top: 4764
Trouser: 4794
Pullover: 4768
Dress: 4796
Coat: 4785
Sandal: 4806
Shirt: 4851
Sneaker: 4820
Bag: 4820
Ankle boot: 4796
```

In [ ]:
```
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 [ ]:
```
print("Mean:", np.mean(x_train))
print("std:", np.std(x_train))
```

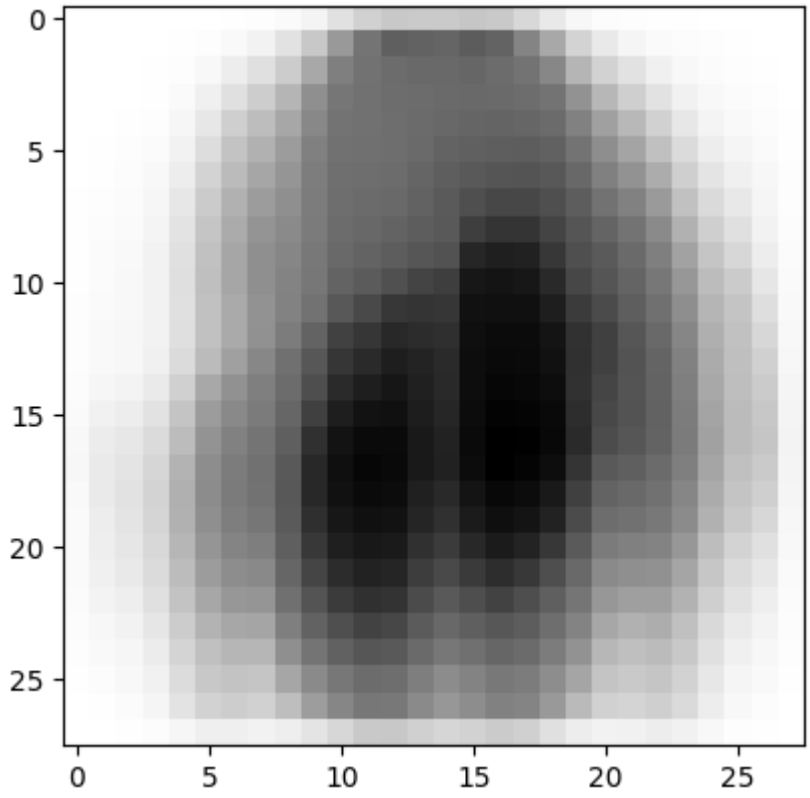
```
Mean: 72.80250212585034
std: 89.97115739980539
```

From the mean pixel value, we see that the average pixel is not very bright, but there is some significant variation in the pixel intensities

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 as well as observe all images in the dataset.

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



Observations

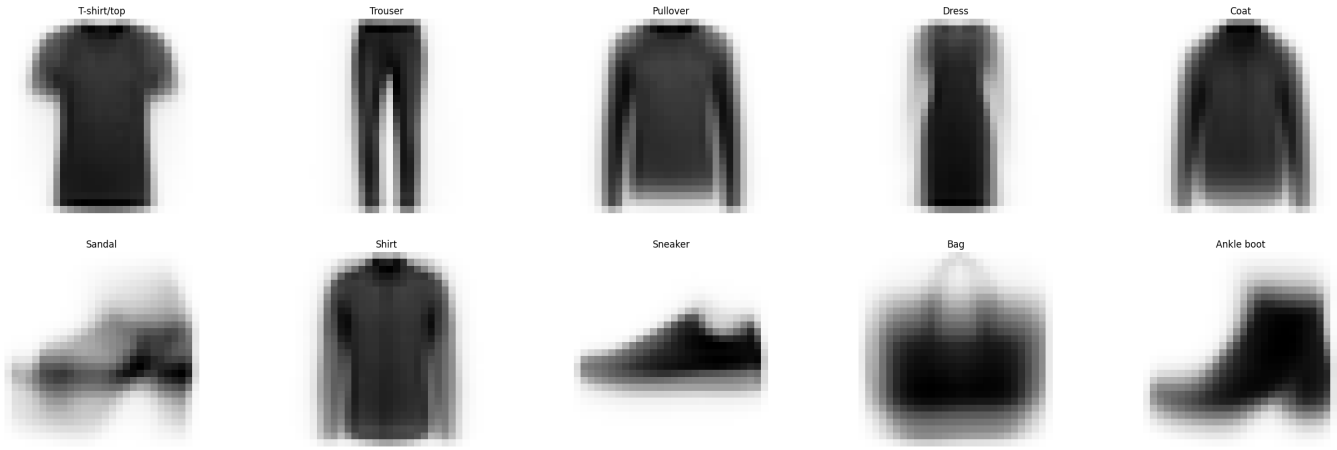
We see that although it is not very clear, we can actually make out a shoe, pants and a shirt in the image. We can also see that the corners images are not touched as well. [Image Padding not needed]

Let's see what is the average image for each class label to see how each image is similar

In [ ]:
```
fig, ax = plt.subplots(2, 5, figsize=(32, 10))

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

As we can see, the average image of each class label is very clear for us to see and predict what the image is about.

However, you can see that Pullovers, Coats and Shirts all look quite similar to one another as they have long sleeves which makes means that model accuracy might not be very good at deciphering the different images. In the Sandal's average image, we can see that the middle of the image looks similar to the sneakers while the remaining pixels that are not as strong looks similar to the ankle boots and the model might have a hard time trying to find features to map to the different images.

Data Preprocessing

Before modelling, its is important to perform data preprocessing

One Hot Encoding

As they are, the current labels are encoded from 0-9, 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. 1.]
Label: tf.Tensor(9, 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 [ ]:
```
pre_processing_v1.mean
```

Out[ ]:
```
<tf.Tensor: shape=(1, 1, 28), dtype=float32, numpy=
array([[[  2.8499734,   8.859591 ,  12.368395 ,  20.10678  ,
          36.203266 ,  53.429104 ,  64.09722  ,  71.28215  ,
          84.05552  , 100.16229  , 113.58216  , 119.21431  ,
         121.25406  , 117.447205 , 115.07686  , 123.79108  ,
         127.0856   , 124.27342  , 116.62026  , 101.4611   ,
          87.20446  ,  80.27911  ,  73.2667   ,  59.864285 ,
          43.230743 ,  32.446007 ,  23.313972 ,   5.6430893]]],
      dtype=float32)>
```

In [ ]:
```
pre_processing_v1.variance
```

Out[ ]:
```
<tf.Tensor: shape=(1, 1, 28), dtype=float32, numpy=
array([[[ 288.0153 , 1392.1926 , 1887.0803 , 2914.9216 , 4850.9116 ,
         6858.577  , 7760.283  , 7918.106  , 8247.954  , 8332.75   ,
         8158.617  , 7963.1963 , 8016.325  , 8044.842  , 8130.704  ,
         7922.196  , 7676.797  , 7676.688  , 8003.404  , 8299.23   ,
         8450.069  , 8588.875  , 8359.848  , 7256.945  , 5751.3223 ,
         4798.869  , 3700.376  ,  549.91345]]], dtype=float32)>
```

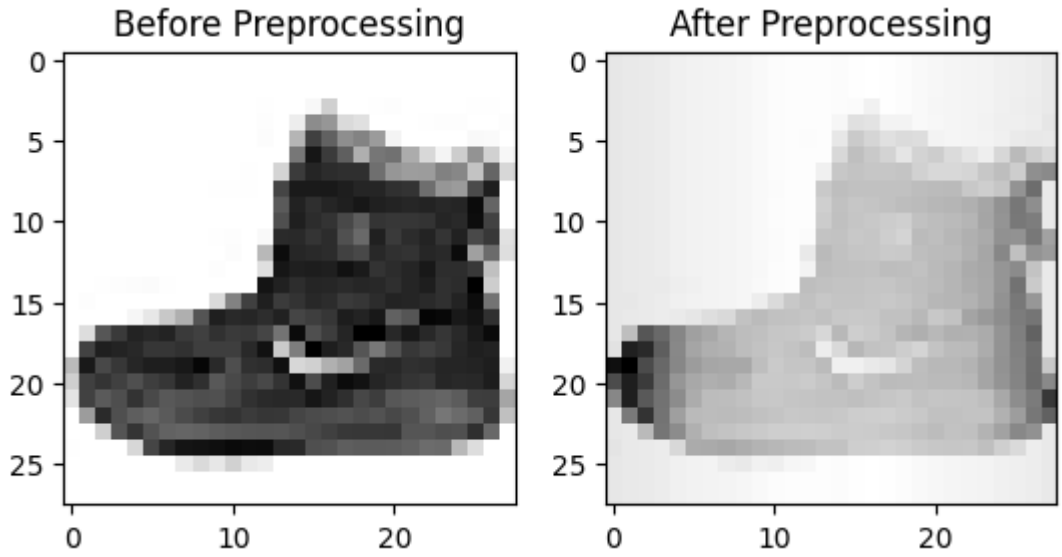
Observations

We see that the values that are closer to the center of the image has higher mean and variance. This has been seen in the average image as most of the images have a brighter pixel in the center of image.

Let's see what happened to the image after we have pre processed.

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

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



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 and shifting. 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 [Shirts can be flipped as both are symmetrical]. We will also be shifting the images up and down, as shown in the EDA, some of the images are long or shorter. By shifting the dataset, we are helping the model identify that the most important part of the image is that the shirt layer is straight etc.

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
```

In [ ]:
```
def data_augmentation(x_train):
    imageArr = []
    for images in x_train:
        images = images.reshape(28, 28, 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])
        image = tf.image.random_crop(
            image, size=IMG_SIZE
        )
        imageArr.append(tf.reshape(image, (28, 28)))
    return np.array(imageArr)
```

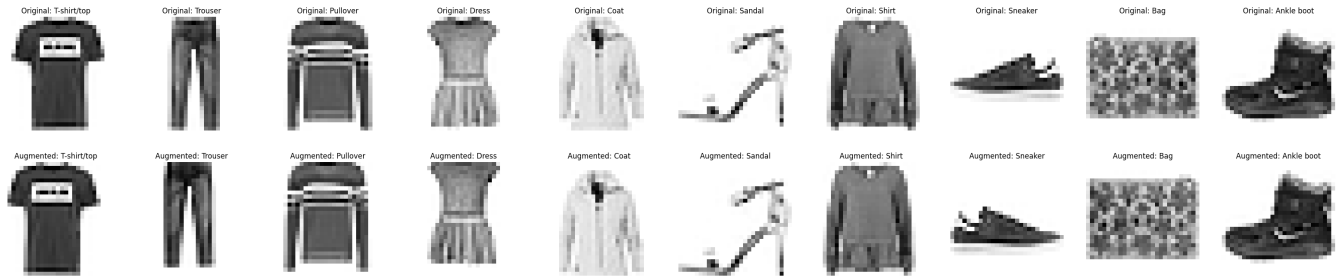
In [ ]:
```
x_train_aug = np.copy(x_train)
```

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

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

In [ ]:
```
fig, ax = plt.subplots(2, 10, figsize=(40, 8))
for idx in range(20):
    subplot = ax.ravel()[idx]
    y_label = np.argmax(y_train, axis=1)
    if idx >= 10:
        subplot.set_title(f"Augmented: {class_labels[idx % 10]}")
        subplot.imshow(x_train_aug[y_label == idx % 10][0], cmap='Greys')
    else:
        subplot.set_title(f"Original: {class_labels[idx % 10]}")
```

```
        subplot.imshow(x_train[y_label == idx % 10][0], cmap='Greys')
    subplot.axis("off")
plt.show()
```



Observations

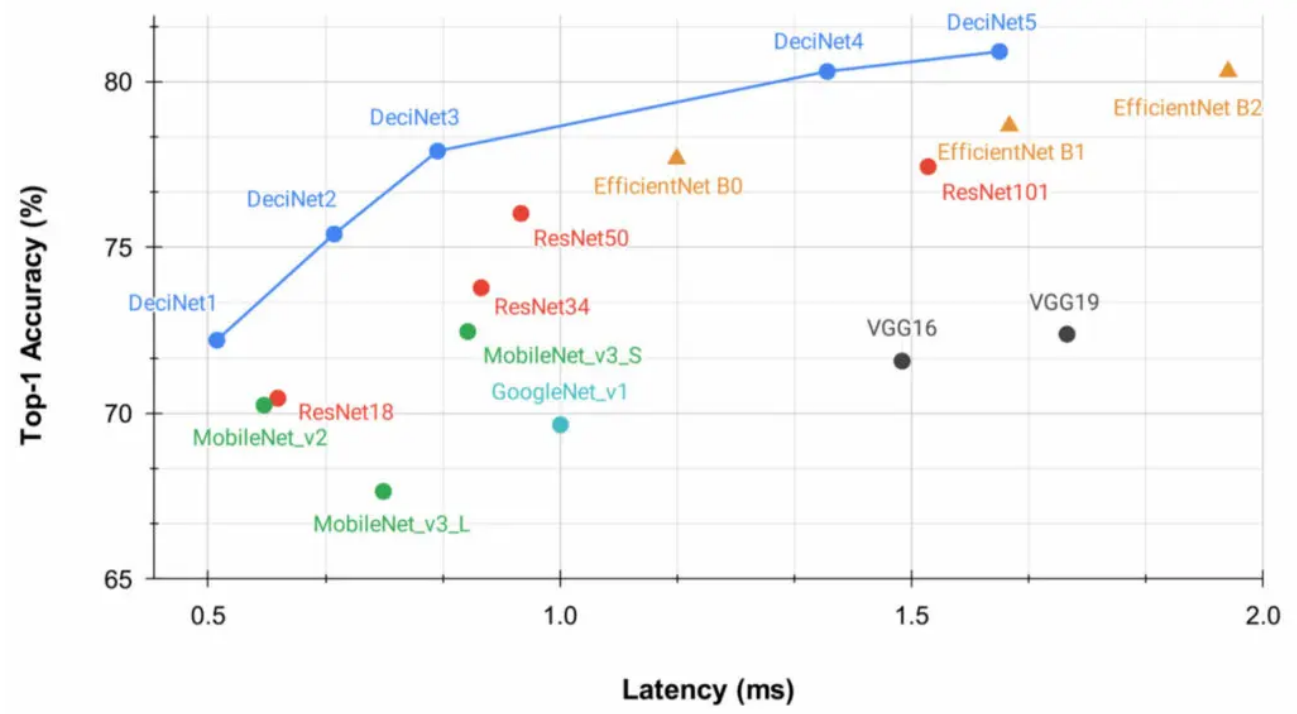
As we can see, some of the images have been shifted, rotated and cropped. This shows that the image augmentation works

Building Models

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

Model List:

1. Fully Connected Neural Network Model (Baseline)
2. Conv2D Neural Network Model
3. CustomVGG Model
4. CustomResNet 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(history):
    history = pd.DataFrame(history)
    epochs = list(range(1, len(history) + 1))
    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()
```

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 [ ]:
```
baseModel.summary()
```

```
Model: "baseline"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 normalization (Normalizatio  (None, 28, 28, 28)       57
 n)

 flatten (Flatten)           (None, 21952)             0

 dense (Dense)               (None, 128)               2809984

 dense_1 (Dense)             (None, 128)               16512

 dense_2 (Dense)             (None, 128)               16512

 dense_3 (Dense)             (None, 10)                1290

=================================================================
Total params: 2,844,355
Trainable params: 2,844,298
Non-trainable params: 57
_________________________________________________________________
```

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

```
Epoch 1/100
750/750 [==============================] - 6s 6ms/step - loss: 0.8432 - accuracy: 0.7232 - va
l_loss: 0.5537 - val_accuracy: 0.7977
Epoch 2/100
750/750 [==============================] - 4s 6ms/step - loss: 0.4954 - accuracy: 0.8201 - va
l_loss: 0.5024 - val_accuracy: 0.8248
Epoch 3/100
750/750 [==============================] - 4s 6ms/step - loss: 0.4543 - accuracy: 0.8339 - va
l_loss: 0.5088 - val_accuracy: 0.8278
Epoch 4/100
750/750 [==============================] - 4s 5ms/step - loss: 0.4336 - accuracy: 0.8443 - va
l_loss: 0.4356 - val_accuracy: 0.8444
Epoch 5/100
750/750 [==============================] - 4s 5ms/step - loss: 0.4134 - accuracy: 0.8496 - va
l_loss: 0.4432 - val_accuracy: 0.8432
Epoch 6/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3991 - accuracy: 0.8565 - va
l_loss: 0.4388 - val_accuracy: 0.8426
Epoch 7/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3931 - accuracy: 0.8571 - va
l_loss: 0.4038 - val_accuracy: 0.8523
Epoch 8/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3759 - accuracy: 0.8620 - va
l_loss: 0.4090 - val_accuracy: 0.8563
Epoch 9/100
750/750 [==============================] - 5s 6ms/step - loss: 0.3747 - accuracy: 0.8622 - va
l_loss: 0.3936 - val_accuracy: 0.8618
Epoch 10/100
750/750 [==============================] - 4s 6ms/step - loss: 0.3684 - accuracy: 0.8660 - va
l_loss: 0.4259 - val_accuracy: 0.8476
Epoch 11/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3624 - accuracy: 0.8686 - va
l_loss: 0.4662 - val_accuracy: 0.8373
Epoch 12/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3569 - accuracy: 0.8698 - va
l_loss: 0.4112 - val_accuracy: 0.8602
Epoch 13/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3529 - accuracy: 0.8703 - va
l_loss: 0.3979 - val_accuracy: 0.8538
Epoch 14/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3481 - accuracy: 0.8719 - va
l_loss: 0.4157 - val_accuracy: 0.8524
Epoch 15/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3497 - accuracy: 0.8736 - va
l_loss: 0.4279 - val_accuracy: 0.8519
Epoch 16/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3414 - accuracy: 0.8729 - va
l_loss: 0.4181 - val_accuracy: 0.8587
Epoch 17/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3409 - accuracy: 0.8762 - va
l_loss: 0.3981 - val_accuracy: 0.8597
Epoch 18/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3327 - accuracy: 0.8798 - va
l_loss: 0.4074 - val_accuracy: 0.8649
Epoch 19/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3303 - accuracy: 0.8782 - va
l_loss: 0.4422 - val_accuracy: 0.8532
Epoch 20/100
750/750 [==============================] - 4s 5ms/step - loss: 0.3279 - accuracy: 0.8813 - va
l_loss: 0.4119 - val_accuracy: 0.8589
Epoch 21/100
750/750 [==============================] - 4s 6ms/step - loss: 0.3223 - accuracy: 0.8809 - va
l_loss: 0.4121 - val_accuracy: 0.8579
Epoch 22/100
750/750 [==============================] - 4s 6ms/step - loss: 0.3242 - accuracy: 0.8813 - va
l_loss: 0.4117 - val_accuracy: 0.8657
```

```
Epoch 23/100
750/750 [==============================] - 7s 9ms/step - loss: 0.3225 - accuracy: 0.8830 - va
l_loss: 0.3988 - val_accuracy: 0.8683
Epoch 24/100
750/750 [==============================] - 5s 6ms/step - loss: 0.3145 - accuracy: 0.8855 - va
l_loss: 0.4037 - val_accuracy: 0.8690
Epoch 25/100
750/750 [==============================] - 6s 7ms/step - loss: 0.3145 - accuracy: 0.8843 - va
l_loss: 0.3977 - val_accuracy: 0.8703
Epoch 26/100
750/750 [==============================] - 5s 6ms/step - loss: 0.3085 - accuracy: 0.8862 - va
l_loss: 0.4487 - val_accuracy: 0.8603
Epoch 27/100
750/750 [==============================] - 5s 7ms/step - loss: 0.3105 - accuracy: 0.8850 - va
l_loss: 0.4061 - val_accuracy: 0.8671
Epoch 28/100
750/750 [==============================] - 9s 12ms/step - loss: 0.3077 - accuracy: 0.8881 - v
al_loss: 0.4208 - val_accuracy: 0.8622
Epoch 29/100
750/750 [==============================] - 10s 13ms/step - loss: 0.3031 - accuracy: 0.8884 -
val_loss: 0.4183 - val_accuracy: 0.8679
Epoch 30/100
750/750 [==============================] - 7s 10ms/step - loss: 0.3044 - accuracy: 0.8890 - v
al_loss: 0.4472 - val_accuracy: 0.8524
Epoch 31/100
750/750 [==============================] - 26s 35ms/step - loss: 0.3025 - accuracy: 0.8901 -
val_loss: 0.4069 - val_accuracy: 0.8622
Epoch 32/100
750/750 [==============================] - 15s 20ms/step - loss: 0.3016 - accuracy: 0.8909 -
val_loss: 0.4763 - val_accuracy: 0.8335
Epoch 33/100
750/750 [==============================] - 7s 9ms/step - loss: 0.3072 - accuracy: 0.8880 - va
l_loss: 0.4161 - val_accuracy: 0.8653
Epoch 34/100
750/750 [==============================] - 6s 8ms/step - loss: 0.3053 - accuracy: 0.8879 - va
l_loss: 0.4097 - val_accuracy: 0.8608
Epoch 35/100
750/750 [==============================] - 12s 16ms/step - loss: 0.2983 - accuracy: 0.8902 -
val_loss: 0.4572 - val_accuracy: 0.8524
```

In [ ]:

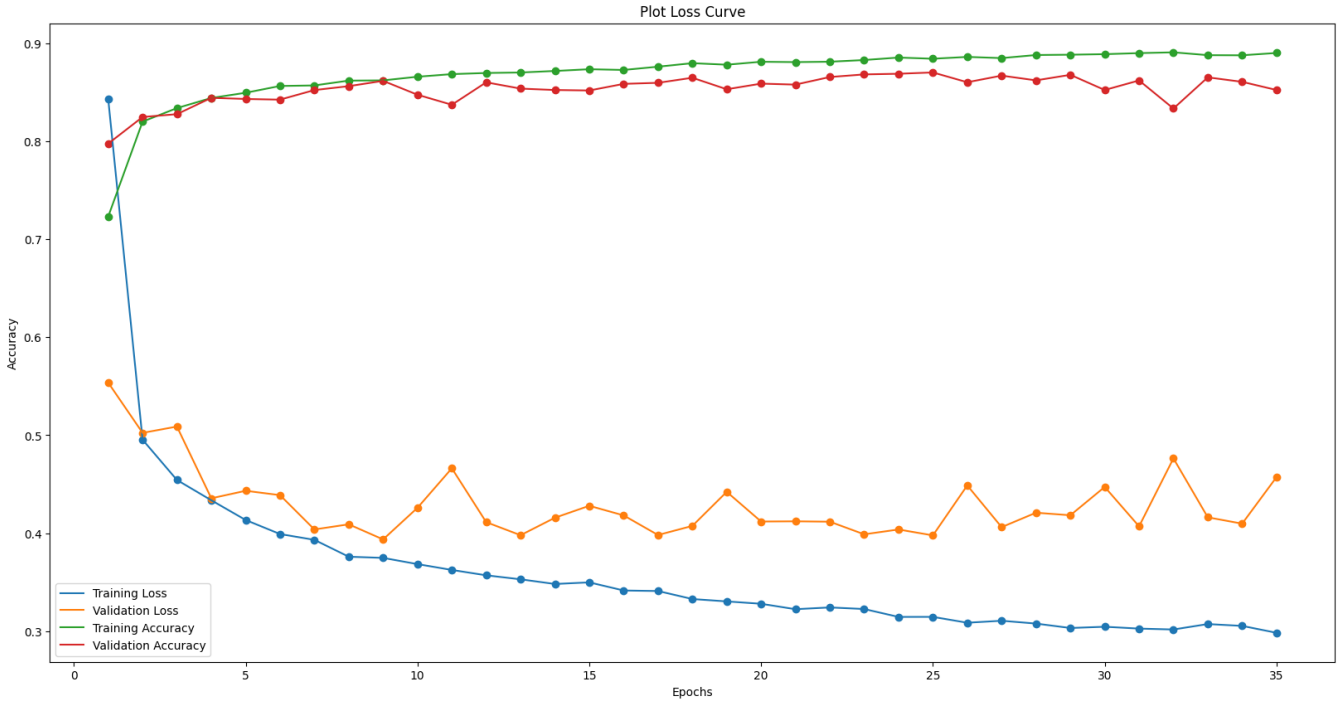
```
baseModelHistory = baseModelHistory.history
best_val_idx = np.argmax(baseModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "baseline"
result["Epochs"] = len(baseModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = baseModelHistory["loss"][best_val_idx]
result["Val Loss"] = baseModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = baseModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = baseModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\3298365700.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:

```
{'Model Name': 'baseline',
 'Epochs': 35,
 'Batch Size': 64,
 'Train Loss': 0.3145187497138977,
 'Val Loss': 0.39770007133483887,
 'Train Acc': 0.8843333125114441,
 'Val Acc': 0.8703333139419556,
 '[Train - Val] Acc': 0.013999998569488525}
```

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



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. This suggest that the baseline model is performing quite well.

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="baseline_Aug")
baseAugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                     loss='categorical_crossentropy', metrics=['accuracy'])
```

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

```
Model: "baseline_Aug"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 normalization (Normalizatio  (None, 28, 28, 28)       57
 n)

 flatten (Flatten)           (None, 21952)             0

 dense (Dense)               (None, 128)               2809984

 dense_1 (Dense)             (None, 128)               16512

 dense_2 (Dense)             (None, 128)               16512

 dense_3 (Dense)             (None, 10)                1290

=================================================================
Total params: 2,844,355
Trainable params: 2,844,298
Non-trainable params: 57
_________________________________________________________________
```

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

```
Epoch 1/100
750/750 [==============================] - 81s 96ms/step - loss: 1.3620 - accuracy: 0.5501 -
val_loss: 0.8151 - val_accuracy: 0.6883
Epoch 2/100
750/750 [==============================] - 39s 52ms/step - loss: 0.7542 - accuracy: 0.6940 -
val_loss: 0.6734 - val_accuracy: 0.7122
Epoch 3/100
750/750 [==============================] - 80s 107ms/step - loss: 0.6836 - accuracy: 0.7224 -
val_loss: 0.6201 - val_accuracy: 0.7414
Epoch 4/100
750/750 [==============================] - 26s 34ms/step - loss: 0.6634 - accuracy: 0.7356 -
val_loss: 0.6211 - val_accuracy: 0.7559
Epoch 5/100
750/750 [==============================] - 29s 38ms/step - loss: 0.6353 - accuracy: 0.7542 -
val_loss: 0.6010 - val_accuracy: 0.7786
Epoch 6/100
750/750 [==============================] - 33s 43ms/step - loss: 0.6139 - accuracy: 0.7671 -
val_loss: 0.6208 - val_accuracy: 0.7708
Epoch 7/100
750/750 [==============================] - 34s 45ms/step - loss: 0.5974 - accuracy: 0.7739 -
val_loss: 0.5706 - val_accuracy: 0.7887
Epoch 8/100
750/750 [==============================] - 30s 40ms/step - loss: 0.5901 - accuracy: 0.7765 -
val_loss: 0.5799 - val_accuracy: 0.7848
Epoch 9/100
750/750 [==============================] - 42s 57ms/step - loss: 0.5839 - accuracy: 0.7807 -
val_loss: 0.5725 - val_accuracy: 0.7931
Epoch 10/100
750/750 [==============================] - 34s 46ms/step - loss: 0.5696 - accuracy: 0.7891 -
val_loss: 0.5875 - val_accuracy: 0.7928
Epoch 11/100
750/750 [==============================] - 29s 38ms/step - loss: 0.5661 - accuracy: 0.7912 -
val_loss: 0.5659 - val_accuracy: 0.7912
Epoch 12/100
750/750 [==============================] - 25s 33ms/step - loss: 0.5613 - accuracy: 0.7946 -
val_loss: 0.5906 - val_accuracy: 0.7991
Epoch 13/100
750/750 [==============================] - 32s 43ms/step - loss: 0.5423 - accuracy: 0.8011 -
val_loss: 0.5485 - val_accuracy: 0.8054
Epoch 14/100
750/750 [==============================] - 25s 34ms/step - loss: 0.5328 - accuracy: 0.8064 -
val_loss: 0.5581 - val_accuracy: 0.8065
Epoch 15/100
750/750 [==============================] - 49s 66ms/step - loss: 0.5277 - accuracy: 0.8080 -
val_loss: 0.5520 - val_accuracy: 0.8021
Epoch 16/100
750/750 [==============================] - 29s 39ms/step - loss: 0.5576 - accuracy: 0.7983 -
val_loss: 0.5359 - val_accuracy: 0.8099
Epoch 17/100
750/750 [==============================] - 28s 38ms/step - loss: 0.5252 - accuracy: 0.8111 -
val_loss: 0.5056 - val_accuracy: 0.8167
Epoch 18/100
750/750 [==============================] - 35s 47ms/step - loss: 0.5179 - accuracy: 0.8133 -
val_loss: 0.5308 - val_accuracy: 0.8158
Epoch 19/100
750/750 [==============================] - 85s 114ms/step - loss: 0.5034 - accuracy: 0.8185 -
val_loss: 0.5141 - val_accuracy: 0.8216
Epoch 20/100
750/750 [==============================] - 87s 116ms/step - loss: 0.5034 - accuracy: 0.8184 -
val_loss: 0.5478 - val_accuracy: 0.8106
Epoch 21/100
750/750 [==============================] - 51s 68ms/step - loss: 0.5017 - accuracy: 0.8180 -
val_loss: 0.5140 - val_accuracy: 0.8232
Epoch 22/100
750/750 [==============================] - 33s 44ms/step - loss: 0.5011 - accuracy: 0.8201 -
val_loss: 0.5044 - val_accuracy: 0.8217
```

```
Epoch 23/100
750/750 [==============================] - 54s 72ms/step - loss: 0.4909 - accuracy: 0.8211 -
val_loss: 0.5693 - val_accuracy: 0.7968
Epoch 24/100
750/750 [==============================] - 51s 68ms/step - loss: 0.4818 - accuracy: 0.8244 -
val_loss: 0.5229 - val_accuracy: 0.8187
Epoch 25/100
750/750 [==============================] - 70s 93ms/step - loss: 0.4839 - accuracy: 0.8257 -
val_loss: 0.5179 - val_accuracy: 0.8253
Epoch 26/100
750/750 [==============================] - 70s 93ms/step - loss: 0.4764 - accuracy: 0.8270 -
val_loss: 0.5023 - val_accuracy: 0.8217
Epoch 27/100
750/750 [==============================] - 68s 91ms/step - loss: 0.4767 - accuracy: 0.8284 -
val_loss: 0.5358 - val_accuracy: 0.8145
Epoch 28/100
750/750 [==============================] - 51s 68ms/step - loss: 0.4675 - accuracy: 0.8292 -
val_loss: 0.5737 - val_accuracy: 0.8142
Epoch 29/100
750/750 [==============================] - 56s 75ms/step - loss: 0.4792 - accuracy: 0.8276 -
val_loss: 0.5311 - val_accuracy: 0.8166
Epoch 30/100
750/750 [==============================] - 49s 60ms/step - loss: 0.4637 - accuracy: 0.8332 -
val_loss: 0.5139 - val_accuracy: 0.8284
Epoch 31/100
750/750 [==============================] - 86s 115ms/step - loss: 0.4674 - accuracy: 0.8303 -
val_loss: 0.5012 - val_accuracy: 0.8244
Epoch 32/100
750/750 [==============================] - 97s 129ms/step - loss: 0.4590 - accuracy: 0.8355 -
val_loss: 0.5132 - val_accuracy: 0.8279
Epoch 33/100
750/750 [==============================] - 101s 134ms/step - loss: 0.4561 - accuracy: 0.8344
- val_loss: 0.5328 - val_accuracy: 0.8192
Epoch 34/100
750/750 [==============================] - 79s 106ms/step - loss: 0.4595 - accuracy: 0.8349 -
val_loss: 0.5250 - val_accuracy: 0.8300
Epoch 35/100
750/750 [==============================] - 82s 109ms/step - loss: 0.4559 - accuracy: 0.8356 -
val_loss: 0.5071 - val_accuracy: 0.8283
Epoch 36/100
750/750 [==============================] - 77s 103ms/step - loss: 0.4542 - accuracy: 0.8342 -
val_loss: 0.4953 - val_accuracy: 0.8329
Epoch 37/100
750/750 [==============================] - 103s 132ms/step - loss: 0.4443 - accuracy: 0.8401
- val_loss: 0.5131 - val_accuracy: 0.8328
Epoch 38/100
750/750 [==============================] - 137s 182ms/step - loss: 0.4385 - accuracy: 0.8411
- val_loss: 0.5091 - val_accuracy: 0.8217
Epoch 39/100
750/750 [==============================] - 181s 237ms/step - loss: 0.4340 - accuracy: 0.8417
- val_loss: 0.5172 - val_accuracy: 0.8247
Epoch 40/100
750/750 [==============================] - 44s 58ms/step - loss: 0.4318 - accuracy: 0.8409 -
val_loss: 0.5131 - val_accuracy: 0.8302
Epoch 41/100
750/750 [==============================] - 34s 45ms/step - loss: 0.4324 - accuracy: 0.8434 -
val_loss: 0.5256 - val_accuracy: 0.8258
Epoch 42/100
750/750 [==============================] - 30s 40ms/step - loss: 0.4323 - accuracy: 0.8425 -
val_loss: 0.5307 - val_accuracy: 0.8282
Epoch 43/100
750/750 [==============================] - 34s 45ms/step - loss: 0.4184 - accuracy: 0.8479 -
val_loss: 0.5184 - val_accuracy: 0.8292
Epoch 44/100
750/750 [==============================] - 34s 45ms/step - loss: 0.4305 - accuracy: 0.8445 -
val_loss: 0.5554 - val_accuracy: 0.8080
```

```
Epoch 45/100
750/750 [==============================] - 39s 51ms/step - loss: 0.4195 - accuracy: 0.8485 -
val_loss: 0.5193 - val_accuracy: 0.8334
Epoch 46/100
750/750 [==============================] - 33s 44ms/step - loss: 0.4133 - accuracy: 0.8489 -
val_loss: 0.5227 - val_accuracy: 0.8267
Epoch 47/100
750/750 [==============================] - 31s 41ms/step - loss: 0.4197 - accuracy: 0.8477 -
val_loss: 0.5008 - val_accuracy: 0.8313
Epoch 48/100
750/750 [==============================] - 30s 40ms/step - loss: 0.4165 - accuracy: 0.8489 -
val_loss: 0.5005 - val_accuracy: 0.8317
Epoch 49/100
750/750 [==============================] - 45s 56ms/step - loss: 0.4095 - accuracy: 0.8500 -
val_loss: 0.5000 - val_accuracy: 0.8320
Epoch 50/100
750/750 [==============================] - 35s 47ms/step - loss: 0.4143 - accuracy: 0.8507 -
val_loss: 0.5273 - val_accuracy: 0.8349
Epoch 51/100
750/750 [==============================] - 40s 54ms/step - loss: 0.4128 - accuracy: 0.8512 -
val_loss: 0.5106 - val_accuracy: 0.8351
Epoch 52/100
750/750 [==============================] - 39s 53ms/step - loss: 0.4675 - accuracy: 0.8328 -
val_loss: 0.5239 - val_accuracy: 0.8244
Epoch 53/100
750/750 [==============================] - 40s 53ms/step - loss: 0.4464 - accuracy: 0.8402 -
val_loss: 0.5426 - val_accuracy: 0.8215
Epoch 54/100
750/750 [==============================] - 53s 70ms/step - loss: 0.4224 - accuracy: 0.8481 -
val_loss: 0.5072 - val_accuracy: 0.8338
Epoch 55/100
750/750 [==============================] - 70s 93ms/step - loss: 0.4074 - accuracy: 0.8527 -
val_loss: 0.5297 - val_accuracy: 0.8277
Epoch 56/100
750/750 [==============================] - 89s 119ms/step - loss: 0.4017 - accuracy: 0.8542 -
val_loss: 0.5268 - val_accuracy: 0.8292
Epoch 57/100
750/750 [==============================] - 103s 137ms/step - loss: 0.4036 - accuracy: 0.8528
- val_loss: 0.5512 - val_accuracy: 0.8284
Epoch 58/100
750/750 [==============================] - 78s 104ms/step - loss: 0.4127 - accuracy: 0.8510 -
val_loss: 0.4871 - val_accuracy: 0.8410
Epoch 59/100
750/750 [==============================] - 62s 83ms/step - loss: 0.3978 - accuracy: 0.8535 -
val_loss: 0.4926 - val_accuracy: 0.8391
Epoch 60/100
750/750 [==============================] - 72s 97ms/step - loss: 0.3960 - accuracy: 0.8560 -
val_loss: 0.5564 - val_accuracy: 0.8280
Epoch 61/100
750/750 [==============================] - 72s 96ms/step - loss: 0.3938 - accuracy: 0.8566 -
val_loss: 0.4908 - val_accuracy: 0.8406
Epoch 62/100
750/750 [==============================] - 56s 75ms/step - loss: 0.4022 - accuracy: 0.8561 -
val_loss: 0.5051 - val_accuracy: 0.8356
Epoch 63/100
750/750 [==============================] - 67s 89ms/step - loss: 0.3996 - accuracy: 0.8555 -
val_loss: 0.5139 - val_accuracy: 0.8362
Epoch 64/100
750/750 [==============================] - 82s 110ms/step - loss: 0.3944 - accuracy: 0.8568 -
val_loss: 0.5294 - val_accuracy: 0.8325
Epoch 65/100
750/750 [==============================] - 113s 151ms/step - loss: 0.3927 - accuracy: 0.8589
- val_loss: 0.5221 - val_accuracy: 0.8352
Epoch 66/100
750/750 [==============================] - 78s 98ms/step - loss: 0.3914 - accuracy: 0.8576 -
val_loss: 0.5096 - val_accuracy: 0.8378
```

```
Epoch 67/100
750/750 [==============================] - 58s 77ms/step - loss: 0.3823 - accuracy: 0.8619 -
val_loss: 0.5349 - val_accuracy: 0.8390
Epoch 68/100
750/750 [==============================] - 62s 83ms/step - loss: 0.3815 - accuracy: 0.8600 -
val_loss: 0.4910 - val_accuracy: 0.8421
Epoch 69/100
750/750 [==============================] - 66s 88ms/step - loss: 0.3802 - accuracy: 0.8635 -
val_loss: 0.5262 - val_accuracy: 0.8324
Epoch 70/100
750/750 [==============================] - 75s 100ms/step - loss: 0.3786 - accuracy: 0.8622 -
val_loss: 0.5303 - val_accuracy: 0.8381
Epoch 71/100
750/750 [==============================] - 57s 76ms/step - loss: 0.3895 - accuracy: 0.8586 -
val_loss: 0.5340 - val_accuracy: 0.8303
Epoch 72/100
750/750 [==============================] - 63s 85ms/step - loss: 0.3868 - accuracy: 0.8610 -
val_loss: 0.5170 - val_accuracy: 0.8402
Epoch 73/100
750/750 [==============================] - 63s 84ms/step - loss: 0.3795 - accuracy: 0.8618 -
val_loss: 0.5345 - val_accuracy: 0.8392
Epoch 74/100
750/750 [==============================] - 59s 79ms/step - loss: 0.3776 - accuracy: 0.8638 -
val_loss: 0.5717 - val_accuracy: 0.8186
Epoch 75/100
750/750 [==============================] - 75s 100ms/step - loss: 0.3828 - accuracy: 0.8624 -
val_loss: 0.5264 - val_accuracy: 0.8442
Epoch 76/100
750/750 [==============================] - 80s 107ms/step - loss: 0.3753 - accuracy: 0.8639 -
val_loss: 0.5065 - val_accuracy: 0.8377
Epoch 77/100
750/750 [==============================] - 65s 87ms/step - loss: 0.3767 - accuracy: 0.8637 -
val_loss: 0.5127 - val_accuracy: 0.8338
Epoch 78/100
750/750 [==============================] - 70s 94ms/step - loss: 0.3752 - accuracy: 0.8649 -
val_loss: 0.5134 - val_accuracy: 0.8378
Epoch 79/100
750/750 [==============================] - 91s 122ms/step - loss: 0.3721 - accuracy: 0.8650 -
val_loss: 0.5374 - val_accuracy: 0.8397
Epoch 80/100
750/750 [==============================] - 109s 145ms/step - loss: 0.3686 - accuracy: 0.8654
- val_loss: 0.5609 - val_accuracy: 0.8317
Epoch 81/100
750/750 [==============================] - 105s 140ms/step - loss: 0.3729 - accuracy: 0.8666
- val_loss: 0.5415 - val_accuracy: 0.8284
Epoch 82/100
750/750 [==============================] - 114s 152ms/step - loss: 0.3714 - accuracy: 0.8655
- val_loss: 0.5202 - val_accuracy: 0.8405
Epoch 83/100
750/750 [==============================] - 80s 107ms/step - loss: 0.3709 - accuracy: 0.8653 -
val_loss: 0.5958 - val_accuracy: 0.8157
Epoch 84/100
750/750 [==============================] - 114s 152ms/step - loss: 0.3695 - accuracy: 0.8658
- val_loss: 0.5568 - val_accuracy: 0.8298
Epoch 85/100
750/750 [==============================] - 102s 136ms/step - loss: 0.3638 - accuracy: 0.8688
- val_loss: 0.5316 - val_accuracy: 0.8343
```

In [ ]:

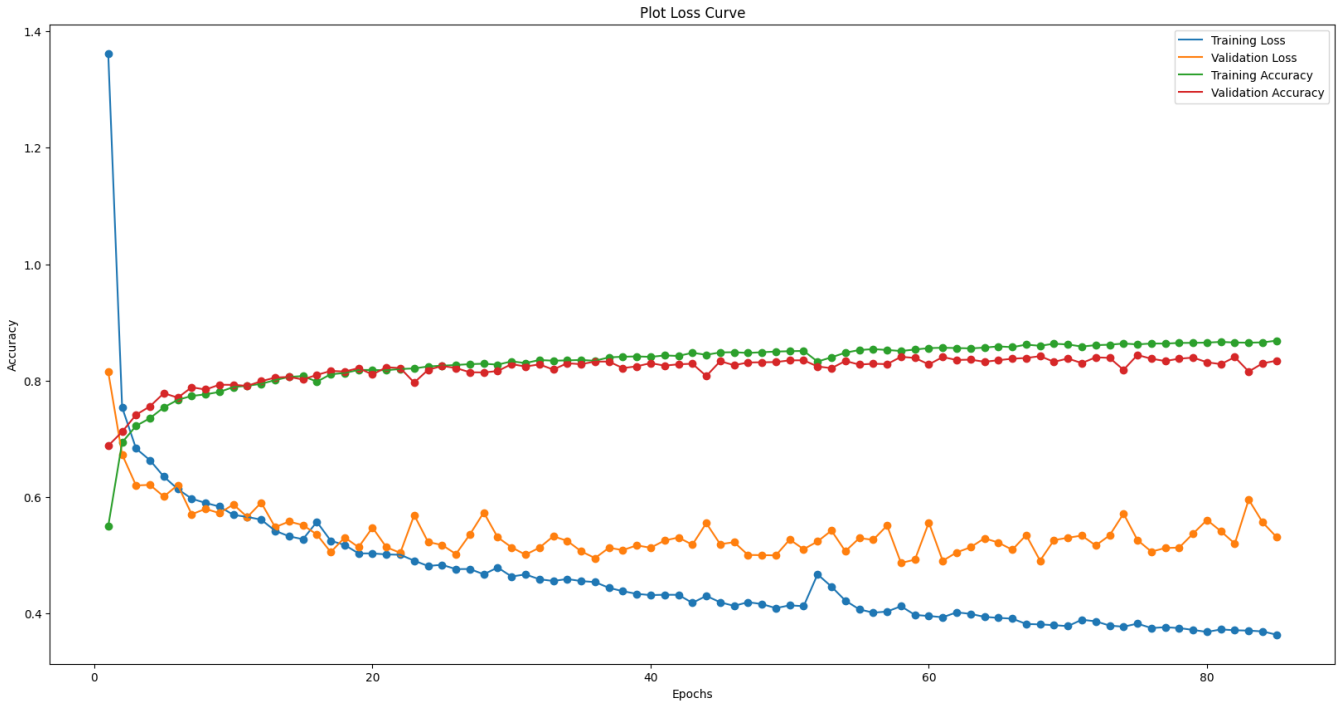
```
baseAugModelHistory = baseAugModelHistory.history
best_val_idx = np.argmax(baseAugModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "baselineAug"
result["Epochs"] = len(baseAugModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = baseAugModelHistory["loss"][best_val_idx]
result["Val Loss"] = baseAugModelHistory["val_loss"][best_val_idx]
```

```
result["Train Acc"] = baseAugModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = baseAugModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_37932\410227960.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'baselineAug',
 'Epochs': 85,
 'Batch Size': 64,
 'Train Loss': 0.3828447163105011,
 'Val Loss': 0.5263938307762146,
 'Train Acc': 0.862375020980835,
 'Val Acc': 0.8441666960716248,
 '[Train - Val] Acc': 0.018208324909210205}
```

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



Observations

Comparing the data augmented dataset to the unaugmented dataset, the model performs worse and become less accurate. This is likely due to the validation data not having and augmented data and this caused the model to understand patterns that are not found in the validation data which causes both accuracy to drop. This suggest that augmenting of data for the fashion MNIST dataset is not a good method of improving accuracy.

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, 28, 28, 1)]       0

 normalization (Normalizatio  (None, 28, 28, 28)       57
 n)

 conv2d (Conv2D)             (None, 24, 24, 32)        22432

 max_pooling2d (MaxPooling2D  (None, 12, 12, 32)       0
 )

 dropout (Dropout)           (None, 12, 12, 32)        0

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 128)               589952

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 613,731
Trainable params: 613,674
Non-trainable params: 57
_________________________________________________________________
```

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

```
Epoch 1/100
750/750 [==============================] - 9s 6ms/step - loss: 0.5424 - accuracy: 0.8065 - va
l_loss: 0.4246 - val_accuracy: 0.8467
Epoch 2/100
750/750 [==============================] - 5s 6ms/step - loss: 0.3863 - accuracy: 0.8584 - va
l_loss: 0.3616 - val_accuracy: 0.8672
Epoch 3/100
750/750 [==============================] - 5s 6ms/step - loss: 0.3475 - accuracy: 0.8718 - va
l_loss: 0.3431 - val_accuracy: 0.8689
Epoch 4/100
750/750 [==============================] - 5s 6ms/step - loss: 0.3216 - accuracy: 0.8799 - va
l_loss: 0.3486 - val_accuracy: 0.8738
Epoch 5/100
750/750 [==============================] - 5s 7ms/step - loss: 0.3033 - accuracy: 0.8859 - va
l_loss: 0.3390 - val_accuracy: 0.8748
Epoch 6/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2874 - accuracy: 0.8919 - va
l_loss: 0.3267 - val_accuracy: 0.8811
Epoch 7/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2716 - accuracy: 0.8975 - va
l_loss: 0.3480 - val_accuracy: 0.8727
Epoch 8/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2622 - accuracy: 0.8998 - va
l_loss: 0.3133 - val_accuracy: 0.8916
Epoch 9/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2482 - accuracy: 0.9069 - va
l_loss: 0.3403 - val_accuracy: 0.8789
Epoch 10/100
750/750 [==============================] - 5s 7ms/step - loss: 0.2430 - accuracy: 0.9075 - va
l_loss: 0.3215 - val_accuracy: 0.8875
Epoch 11/100
750/750 [==============================] - 5s 7ms/step - loss: 0.2329 - accuracy: 0.9121 - va
l_loss: 0.3305 - val_accuracy: 0.8898
Epoch 12/100
750/750 [==============================] - 5s 7ms/step - loss: 0.2303 - accuracy: 0.9116 - va
l_loss: 0.3376 - val_accuracy: 0.8867
Epoch 13/100
750/750 [==============================] - 5s 7ms/step - loss: 0.2177 - accuracy: 0.9184 - va
l_loss: 0.3539 - val_accuracy: 0.8837
Epoch 14/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2123 - accuracy: 0.9196 - va
l_loss: 0.3423 - val_accuracy: 0.8891
Epoch 15/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2045 - accuracy: 0.9222 - va
l_loss: 0.3351 - val_accuracy: 0.8888
Epoch 16/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2001 - accuracy: 0.9246 - va
l_loss: 0.3706 - val_accuracy: 0.8824
Epoch 17/100
750/750 [==============================] - 5s 6ms/step - loss: 0.2010 - accuracy: 0.9256 - va
l_loss: 0.3489 - val_accuracy: 0.8878
Epoch 18/100
750/750 [==============================] - 5s 6ms/step - loss: 0.1977 - accuracy: 0.9271 - va
l_loss: 0.3588 - val_accuracy: 0.8885
```

In [ ]:

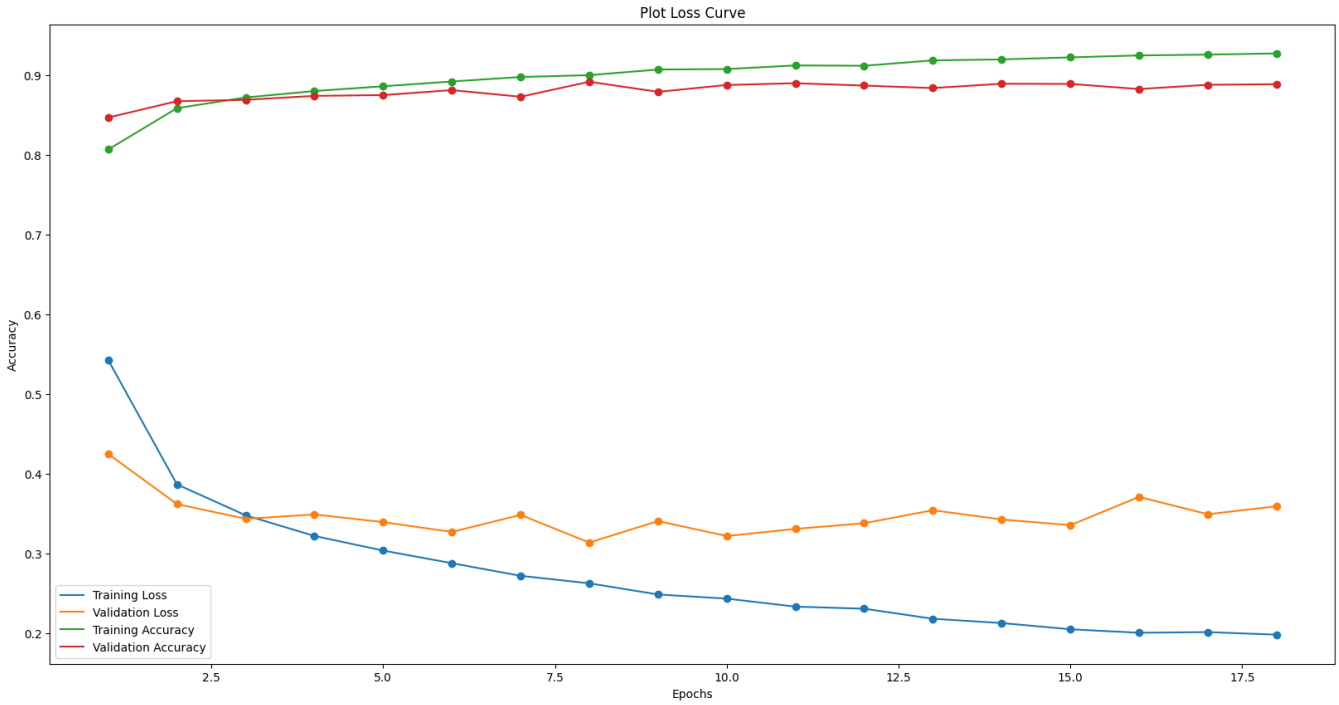
```
conv2DModelHistory = conv2DModelHistory.history
best_val_idx = np.argmax(conv2DModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "conv2D"
result["Epochs"] = len(conv2DModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = conv2DModelHistory["loss"][best_val_idx]
result["Val Loss"] = conv2DModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = conv2DModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = conv2DModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
```

```
result = pd.Series(result, name=result["Model Name"])
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\3935077863.py:13: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
Model Name             conv2D
Epochs                     18
Batch Size                 64
Train Loss           0.262191
Val Loss             0.313302
Train Acc            0.899833
Val Acc              0.891583
[Train - Val] Acc     0.00825
Name: conv2D, dtype: object
```

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



Observations

Comparing out conv2D model compared to our baseline model, we can clearly see that conv2D model performed greater than baseline model as not only did the accuracy increase, the loss values also decreased. This shows and proves that the conv2D model is better than baseline neural networks as model is able to use neighbouring pixels to help in prediction and understanding the different features of the images.

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="conv2D")
```

```
conv2DAugModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                       loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
conv2DAugModel.summary()
```

```
Model: "conv2D"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 normalization (Normalizatio  (None, 28, 28, 28)       57
 n)

 conv2d (Conv2D)             (None, 24, 24, 32)        22432

 max_pooling2d (MaxPooling2D  (None, 12, 12, 32)       0
 )

 dropout (Dropout)           (None, 12, 12, 32)        0

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 128)               589952

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 613,731
Trainable params: 613,674
Non-trainable params: 57
_________________________________________________________________
```

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

```
Epoch 1/100
750/750 [==============================] - 83s 110ms/step - loss: 0.2255 - accuracy: 0.9156 -
val_loss: 0.4413 - val_accuracy: 0.8699
Epoch 2/100
750/750 [==============================] - 69s 92ms/step - loss: 0.2286 - accuracy: 0.9158 -
val_loss: 0.4770 - val_accuracy: 0.8571
Epoch 3/100
750/750 [==============================] - 57s 76ms/step - loss: 0.2150 - accuracy: 0.9193 -
val_loss: 0.4800 - val_accuracy: 0.8668
Epoch 4/100
750/750 [==============================] - 50s 66ms/step - loss: 0.2084 - accuracy: 0.9211 -
val_loss: 0.4560 - val_accuracy: 0.8693
Epoch 5/100
750/750 [==============================] - 47s 62ms/step - loss: 0.2131 - accuracy: 0.9222 -
val_loss: 0.4672 - val_accuracy: 0.8708
Epoch 6/100
750/750 [==============================] - 57s 75ms/step - loss: 0.2047 - accuracy: 0.9255 -
val_loss: 0.4828 - val_accuracy: 0.8587
Epoch 7/100
750/750 [==============================] - 65s 87ms/step - loss: 0.1954 - accuracy: 0.9275 -
val_loss: 0.5021 - val_accuracy: 0.8588
Epoch 8/100
750/750 [==============================] - 82s 109ms/step - loss: 0.1999 - accuracy: 0.9248 -
val_loss: 0.4617 - val_accuracy: 0.8686
Epoch 9/100
750/750 [==============================] - 104s 139ms/step - loss: 0.1991 - accuracy: 0.9281
- val_loss: 0.5194 - val_accuracy: 0.8572
Epoch 10/100
750/750 [==============================] - 105s 141ms/step - loss: 0.1907 - accuracy: 0.9290
- val_loss: 0.5251 - val_accuracy: 0.8683
Epoch 11/100
750/750 [==============================] - 100s 133ms/step - loss: 0.1939 - accuracy: 0.9301
- val_loss: 0.5601 - val_accuracy: 0.8545
Epoch 12/100
750/750 [==============================] - 106s 141ms/step - loss: 0.1898 - accuracy: 0.9307
- val_loss: 0.5070 - val_accuracy: 0.8696
Epoch 13/100
750/750 [==============================] - 114s 152ms/step - loss: 0.1868 - accuracy: 0.9321
- val_loss: 0.5391 - val_accuracy: 0.8609
Epoch 14/100
750/750 [==============================] - 97s 130ms/step - loss: 0.1929 - accuracy: 0.9317 -
val_loss: 0.5254 - val_accuracy: 0.8649
Epoch 15/100
750/750 [==============================] - 97s 129ms/step - loss: 0.1859 - accuracy: 0.9318 -
val_loss: 0.5326 - val_accuracy: 0.8645
```

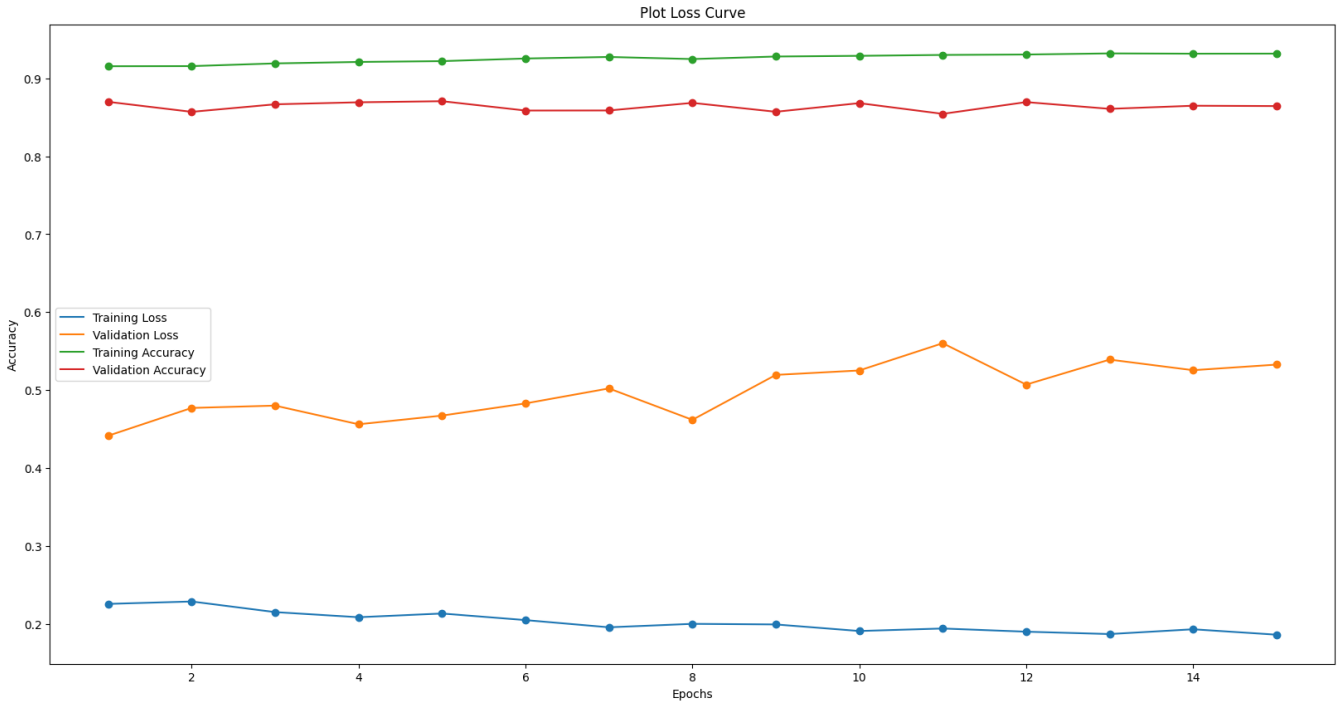
In [ ]:

```python
conv2DAugModelHistory = conv2DAugModelHistory.history
best_val_idx = np.argmax(conv2DAugModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "conv2DAug"
result["Epochs"] = len(conv2DAugModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = conv2DAugModelHistory["loss"][best_val_idx]
result["Val Loss"] = conv2DAugModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = conv2DAugModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = conv2DAugModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
result = pd.Series(result, name=result["Model Name"])
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_37932\1807718742.py:13: 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)
```

Out[ ]:
```
Model Name           conv2DAug
Epochs                      15
Batch Size                  64
Train Loss             0.21314
Val Loss              0.467183
Train Acc             0.922167
Val Acc                0.87075
[Train - Val] Acc     0.051417
Name: conv2DAug, dtype: object
```

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

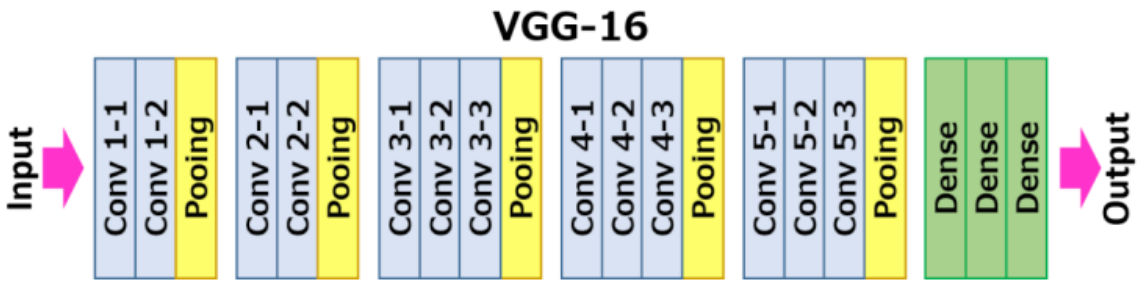


Observations

Comparing out conv2D model with aug compared to our conv2D model without aug, we can clearly see that conv2D model without aug performed better than conv2D model with aug in terms of validation accuracy. However, the training accuracy increased which suggest that with the conv2D layers, the model is able to better predict the augmented data. [overfit]

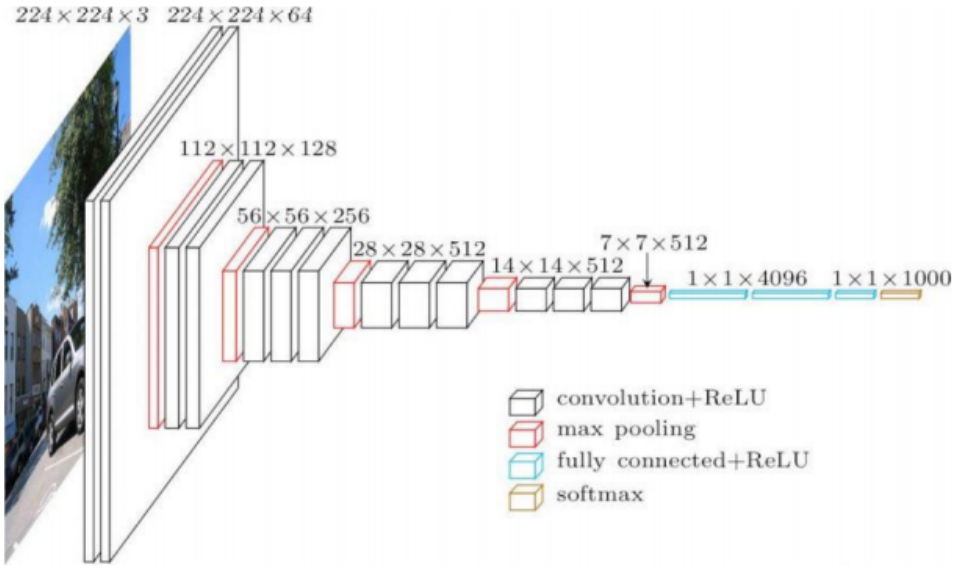
CustomVGG Model

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



The Architecture

The architecture depicted below is VGG16.



VGG16 Model Deep Dive

In [ ]:
```
vgg16Model = tf.keras.applications.vgg16.VGG16(
    include_top=True,
    weights='imagenet',
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000,
    classifier_activation='softmax'
)
```

In [ ]:
```
vgg16Model.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_2 (InputLayer)         [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0

 flatten (Flatten)            (None, 25088)             0

 fc1 (Dense)                  (None, 4096)              102764544

 fc2 (Dense)                  (None, 4096)              16781312

 predictions (Dense)          (None, 1000)              4097000

=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

Observations

We see that despite only having 16 layers, the VGG16 model, but there is a large parameter size. This is because each layer has a complex input shape which will make training slow. And because of time and training constraint, we will be reducing the number of layers and making a custom vgg model.

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(ReLU())
    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 = Flatten()(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 [ ]:
```
customVGGModel.summary()
```

```
Model: "CustomVGG"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 normalization (Normalizatio  (None, 28, 28, 28)       57
 n)

 sequential (Sequential)     (None, 14, 14, 32)        17600

 sequential_1 (Sequential)   (None, 7, 7, 64)          55936

 sequential_2 (Sequential)   (None, 3, 3, 128)         370560

 sequential_3 (Sequential)   (None, 1, 1, 256)         1478400

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 256)               65792

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 1,990,915
Trainable params: 1,988,170
Non-trainable params: 2,745
_________________________________________________________________
```

In [ ]:
```
customVGGModelHistory = customVGGModel.fit(x_train, y_train, epochs=100,
                                           validation_data=(x_val, y_val), batch_size=BATCH_S
```

```
Epoch 1/100
750/750 [==============================] - 19s 22ms/step - loss: 0.5630 - accuracy: 0.7966 -
val_loss: 0.4889 - val_accuracy: 0.8184
Epoch 2/100
750/750 [==============================] - 16s 21ms/step - loss: 0.3372 - accuracy: 0.8769 -
val_loss: 0.3386 - val_accuracy: 0.8717
Epoch 3/100
750/750 [==============================] - 15s 20ms/step - loss: 0.2807 - accuracy: 0.8957 -
val_loss: 0.2654 - val_accuracy: 0.9035
Epoch 4/100
750/750 [==============================] - 15s 20ms/step - loss: 0.2449 - accuracy: 0.9101 -
val_loss: 0.2457 - val_accuracy: 0.9108
Epoch 5/100
750/750 [==============================] - 15s 20ms/step - loss: 0.2178 - accuracy: 0.9202 -
val_loss: 0.2397 - val_accuracy: 0.9129
Epoch 6/100
750/750 [==============================] - 15s 20ms/step - loss: 0.2017 - accuracy: 0.9257 -
val_loss: 0.3356 - val_accuracy: 0.8768
Epoch 7/100
750/750 [==============================] - 15s 19ms/step - loss: 0.1824 - accuracy: 0.9320 -
val_loss: 0.2525 - val_accuracy: 0.9140
Epoch 8/100
750/750 [==============================] - 15s 20ms/step - loss: 0.1643 - accuracy: 0.9393 -
val_loss: 0.2289 - val_accuracy: 0.9184
Epoch 9/100
750/750 [==============================] - 15s 20ms/step - loss: 0.1528 - accuracy: 0.9429 -
val_loss: 0.2924 - val_accuracy: 0.8947
Epoch 10/100
750/750 [==============================] - 15s 20ms/step - loss: 0.1389 - accuracy: 0.9469 -
val_loss: 0.2650 - val_accuracy: 0.9143
Epoch 11/100
750/750 [==============================] - 15s 20ms/step - loss: 0.1256 - accuracy: 0.9528 -
val_loss: 0.2747 - val_accuracy: 0.9068
Epoch 12/100
750/750 [==============================] - 15s 20ms/step - loss: 0.1162 - accuracy: 0.9574 -
val_loss: 0.2555 - val_accuracy: 0.9197
Epoch 13/100
750/750 [==============================] - 15s 20ms/step - loss: 0.1036 - accuracy: 0.9610 -
val_loss: 0.2562 - val_accuracy: 0.9173
Epoch 14/100
750/750 [==============================] - 23s 31ms/step - loss: 0.1002 - accuracy: 0.9626 -
val_loss: 0.2784 - val_accuracy: 0.9158
Epoch 15/100
750/750 [==============================] - 19s 25ms/step - loss: 0.0880 - accuracy: 0.9676 -
val_loss: 0.2808 - val_accuracy: 0.9155
Epoch 16/100
750/750 [==============================] - 17s 23ms/step - loss: 0.0806 - accuracy: 0.9693 -
val_loss: 0.3653 - val_accuracy: 0.9002
Epoch 17/100
750/750 [==============================] - 18s 24ms/step - loss: 0.0735 - accuracy: 0.9725 -
val_loss: 0.3004 - val_accuracy: 0.9205
Epoch 18/100
750/750 [==============================] - 18s 24ms/step - loss: 0.0685 - accuracy: 0.9749 -
val_loss: 0.3073 - val_accuracy: 0.9151
Epoch 19/100
750/750 [==============================] - 17s 23ms/step - loss: 0.0636 - accuracy: 0.9769 -
val_loss: 0.3998 - val_accuracy: 0.9002
Epoch 20/100
750/750 [==============================] - 18s 23ms/step - loss: 0.0555 - accuracy: 0.9804 -
val_loss: 0.3045 - val_accuracy: 0.9194
Epoch 21/100
750/750 [==============================] - 18s 24ms/step - loss: 0.0467 - accuracy: 0.9832 -
val_loss: 0.3671 - val_accuracy: 0.9133
Epoch 22/100
750/750 [==============================] - 17s 23ms/step - loss: 0.0448 - accuracy: 0.9841 -
val_loss: 0.3610 - val_accuracy: 0.9198
```

```
Epoch 23/100
750/750 [==============================] - 17s 22ms/step - loss: 0.0436 - accuracy: 0.9844 -
val_loss: 0.3564 - val_accuracy: 0.9187
Epoch 24/100
750/750 [==============================] - 17s 23ms/step - loss: 0.0396 - accuracy: 0.9856 -
val_loss: 0.3492 - val_accuracy: 0.9197
Epoch 25/100
750/750 [==============================] - 18s 24ms/step - loss: 0.0390 - accuracy: 0.9865 -
val_loss: 0.3951 - val_accuracy: 0.9128
Epoch 26/100
750/750 [==============================] - 17s 23ms/step - loss: 0.0343 - accuracy: 0.9879 -
val_loss: 0.3544 - val_accuracy: 0.9172
Epoch 27/100
750/750 [==============================] - 16s 22ms/step - loss: 0.0289 - accuracy: 0.9902 -
val_loss: 0.3918 - val_accuracy: 0.9220
Epoch 28/100
750/750 [==============================] - 17s 22ms/step - loss: 0.0275 - accuracy: 0.9910 -
val_loss: 0.3608 - val_accuracy: 0.9221
Epoch 29/100
750/750 [==============================] - 16s 22ms/step - loss: 0.0244 - accuracy: 0.9914 -
val_loss: 0.3985 - val_accuracy: 0.9240
Epoch 30/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0241 - accuracy: 0.9917 -
val_loss: 0.3856 - val_accuracy: 0.9210
Epoch 31/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0207 - accuracy: 0.9933 -
val_loss: 0.3821 - val_accuracy: 0.9233
Epoch 32/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0210 - accuracy: 0.9929 -
val_loss: 0.3908 - val_accuracy: 0.9243
Epoch 33/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0194 - accuracy: 0.9936 -
val_loss: 0.3997 - val_accuracy: 0.9262
Epoch 34/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0164 - accuracy: 0.9946 -
val_loss: 0.4214 - val_accuracy: 0.9193
Epoch 35/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0170 - accuracy: 0.9944 -
val_loss: 0.4105 - val_accuracy: 0.9178
Epoch 36/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0173 - accuracy: 0.9942 -
val_loss: 0.4454 - val_accuracy: 0.9122
Epoch 37/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0193 - accuracy: 0.9931 -
val_loss: 0.4087 - val_accuracy: 0.9201
Epoch 38/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0166 - accuracy: 0.9948 -
val_loss: 0.4599 - val_accuracy: 0.9160
Epoch 39/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0151 - accuracy: 0.9953 -
val_loss: 0.3982 - val_accuracy: 0.9255
Epoch 40/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0094 - accuracy: 0.9972 -
val_loss: 0.4509 - val_accuracy: 0.9233
Epoch 41/100
750/750 [==============================] - 16s 22ms/step - loss: 0.0140 - accuracy: 0.9956 -
val_loss: 0.4308 - val_accuracy: 0.9198
Epoch 42/100
750/750 [==============================] - 16s 21ms/step - loss: 0.0139 - accuracy: 0.9952 -
val_loss: 0.4595 - val_accuracy: 0.9212
Epoch 43/100
750/750 [==============================] - 16s 22ms/step - loss: 0.0105 - accuracy: 0.9966 -
val_loss: 0.4878 - val_accuracy: 0.9221
```

In [ ]:

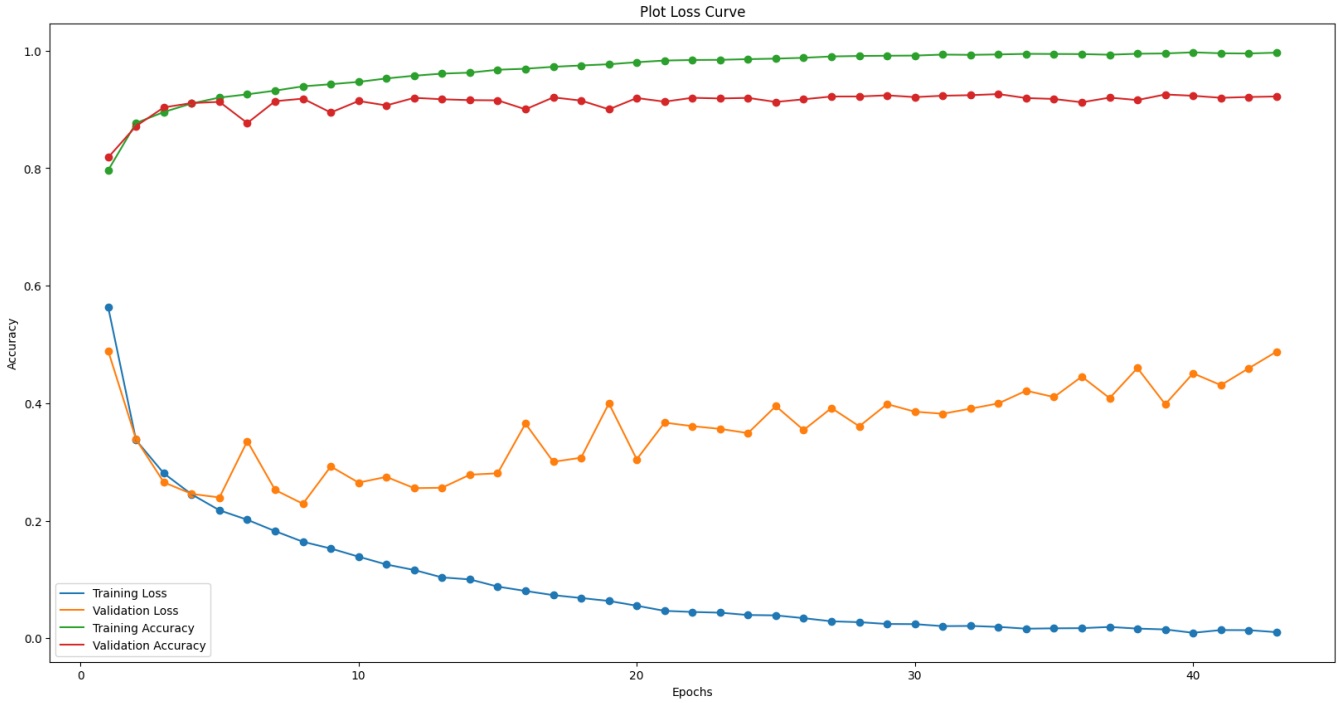
```
customVGGModelHistory = customVGGModelHistory.history
best_val_idx = np.argmax(customVGGModelHistory["val_accuracy"])
```

```
result = {}
result["Model Name"] = "customVGG"
result["Epochs"] = len(customVGGModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customVGGModelHistory["loss"][best_val_idx]
result["Val Loss"] = customVGGModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customVGGModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customVGGModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\300855129.py:12: FutureWarning: The fr
ame.append method is deprecated and will be removed from pandas in a future version. Use pand
as.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'customVGG',
 'Epochs': 43,
 'Batch Size': 64,
 'Train Loss': 0.019438067451119423,
 'Val Loss': 0.3996966481208801,
 'Train Acc': 0.9935833215713501,
 'Val Acc': 0.9261666536331177,
 '[Train - Val] Acc': 0.06741666793823242}
```

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



Observations

Comparing our CustomVGGModel with our baseline model, we can see that the VGG model is able to predict and classify the images better with higher accuracy and is generalising the data. As the validation loss increased over time with more epochs, it seems that the VGG model is quite overfitted. To reduce the validation loss, we will try to use the augmented data.

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 = Flatten()(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 [ ]:
```
customVGGAugModel.summary()
```

```
Model: "CustomVGGAug"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 normalization (Normalizatio  (None, 28, 28, 28)       57
 n)

 sequential (Sequential)     (None, 14, 14, 32)        17600

 sequential_1 (Sequential)   (None, 7, 7, 64)          55936

 sequential_2 (Sequential)   (None, 3, 3, 128)         370560

 sequential_3 (Sequential)   (None, 1, 1, 256)         1478400

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 256)               65792

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 1,990,915
Trainable params: 1,988,170
Non-trainable params: 2,745
_________________________________________________________________
```

In [ ]:
```
customVGGAugModelHistory = customVGGAugModel.fit(x_train_aug, y_train, epochs=100,
                                                 validation_data=(x_val, y_val), batch_size=B
```

```
Epoch 1/100
750/750 [==============================] - 320s 402ms/step - loss: 0.6558 - accuracy: 0.7600
- val_loss: 0.3762 - val_accuracy: 0.8590
Epoch 2/100
750/750 [==============================] - 174s 233ms/step - loss: 0.3891 - accuracy: 0.8577
- val_loss: 0.3345 - val_accuracy: 0.8773
Epoch 3/100
750/750 [==============================] - 183s 244ms/step - loss: 0.3210 - accuracy: 0.8816
- val_loss: 0.3160 - val_accuracy: 0.8870
Epoch 4/100
750/750 [==============================] - 199s 260ms/step - loss: 0.2821 - accuracy: 0.8976
- val_loss: 0.2783 - val_accuracy: 0.8964
Epoch 5/100
750/750 [==============================] - 192s 257ms/step - loss: 0.2524 - accuracy: 0.9084
- val_loss: 0.3077 - val_accuracy: 0.8876
Epoch 6/100
750/750 [==============================] - 210s 280ms/step - loss: 0.2254 - accuracy: 0.9182
- val_loss: 0.3201 - val_accuracy: 0.8947
Epoch 7/100
750/750 [==============================] - 152s 203ms/step - loss: 0.2042 - accuracy: 0.9244
- val_loss: 0.2756 - val_accuracy: 0.9000
Epoch 8/100
750/750 [==============================] - 135s 180ms/step - loss: 0.1858 - accuracy: 0.9310
- val_loss: 0.3063 - val_accuracy: 0.8994
Epoch 9/100
750/750 [==============================] - 129s 172ms/step - loss: 0.1743 - accuracy: 0.9358
- val_loss: 0.2676 - val_accuracy: 0.9075
Epoch 10/100
750/750 [==============================] - 140s 187ms/step - loss: 0.1585 - accuracy: 0.9408
- val_loss: 0.2627 - val_accuracy: 0.9125
Epoch 11/100
750/750 [==============================] - 133s 177ms/step - loss: 0.1372 - accuracy: 0.9487
- val_loss: 0.2863 - val_accuracy: 0.9084
Epoch 12/100
750/750 [==============================] - 130s 173ms/step - loss: 0.1246 - accuracy: 0.9530
- val_loss: 0.2855 - val_accuracy: 0.9086
Epoch 13/100
750/750 [==============================] - 136s 182ms/step - loss: 0.1148 - accuracy: 0.9578
- val_loss: 0.3017 - val_accuracy: 0.9073
Epoch 14/100
750/750 [==============================] - 125s 167ms/step - loss: 0.1058 - accuracy: 0.9615
- val_loss: 0.3181 - val_accuracy: 0.9018
Epoch 15/100
750/750 [==============================] - 134s 179ms/step - loss: 0.0961 - accuracy: 0.9644
- val_loss: 0.3120 - val_accuracy: 0.9097
Epoch 16/100
750/750 [==============================] - 156s 208ms/step - loss: 0.0860 - accuracy: 0.9684
- val_loss: 0.3620 - val_accuracy: 0.9074
Epoch 17/100
750/750 [==============================] - 156s 209ms/step - loss: 0.0819 - accuracy: 0.9704
- val_loss: 0.3639 - val_accuracy: 0.8991
Epoch 18/100
750/750 [==============================] - 148s 198ms/step - loss: 0.0734 - accuracy: 0.9738
- val_loss: 0.3200 - val_accuracy: 0.9110
Epoch 19/100
750/750 [==============================] - 157s 209ms/step - loss: 0.0652 - accuracy: 0.9766
- val_loss: 0.3168 - val_accuracy: 0.9162
Epoch 20/100
750/750 [==============================] - 155s 207ms/step - loss: 0.0586 - accuracy: 0.9786
- val_loss: 0.3671 - val_accuracy: 0.9119
Epoch 21/100
750/750 [==============================] - 177s 236ms/step - loss: 0.0534 - accuracy: 0.9810
- val_loss: 0.3792 - val_accuracy: 0.9064
Epoch 22/100
750/750 [==============================] - 230s 306ms/step - loss: 0.0496 - accuracy: 0.9819
- val_loss: 0.3817 - val_accuracy: 0.9082
```

```
Epoch 23/100
750/750 [==============================] - 197s 263ms/step - loss: 0.0458 - accuracy: 0.9838
- val_loss: 0.3636 - val_accuracy: 0.9083
Epoch 24/100
750/750 [==============================] - 209s 278ms/step - loss: 0.0429 - accuracy: 0.9844
- val_loss: 0.4019 - val_accuracy: 0.9121
Epoch 25/100
750/750 [==============================] - 191s 256ms/step - loss: 0.0347 - accuracy: 0.9879
- val_loss: 0.3945 - val_accuracy: 0.9142
Epoch 26/100
750/750 [==============================] - 194s 253ms/step - loss: 0.0310 - accuracy: 0.9891
- val_loss: 0.4327 - val_accuracy: 0.9112
Epoch 27/100
750/750 [==============================] - 242s 323ms/step - loss: 0.0300 - accuracy: 0.9895
- val_loss: 0.4656 - val_accuracy: 0.9061
Epoch 28/100
750/750 [==============================] - 198s 264ms/step - loss: 0.0322 - accuracy: 0.9888
- val_loss: 0.4452 - val_accuracy: 0.9099
Epoch 29/100
750/750 [==============================] - 200s 267ms/step - loss: 0.0290 - accuracy: 0.9898
- val_loss: 0.3887 - val_accuracy: 0.9170
Epoch 30/100
750/750 [==============================] - 196s 262ms/step - loss: 0.0222 - accuracy: 0.9924
- val_loss: 0.3961 - val_accuracy: 0.9170
Epoch 31/100
750/750 [==============================] - 203s 271ms/step - loss: 0.0204 - accuracy: 0.9930
- val_loss: 0.4709 - val_accuracy: 0.9164
Epoch 32/100
750/750 [==============================] - 201s 268ms/step - loss: 0.0247 - accuracy: 0.9912
- val_loss: 0.4398 - val_accuracy: 0.9145
Epoch 33/100
750/750 [==============================] - 190s 253ms/step - loss: 0.0194 - accuracy: 0.9932
- val_loss: 0.4705 - val_accuracy: 0.9118
Epoch 34/100
750/750 [==============================] - 184s 240ms/step - loss: 0.0228 - accuracy: 0.9920
- val_loss: 0.4688 - val_accuracy: 0.9123
Epoch 35/100
750/750 [==============================] - 195s 260ms/step - loss: 0.0194 - accuracy: 0.9936
- val_loss: 0.4549 - val_accuracy: 0.9160
Epoch 36/100
750/750 [==============================] - 223s 298ms/step - loss: 0.0201 - accuracy: 0.9933
- val_loss: 0.4321 - val_accuracy: 0.9177
Epoch 37/100
750/750 [==============================] - 209s 274ms/step - loss: 0.0167 - accuracy: 0.9945
- val_loss: 0.4350 - val_accuracy: 0.9195
Epoch 38/100
750/750 [==============================] - 177s 237ms/step - loss: 0.0126 - accuracy: 0.9957
- val_loss: 0.4678 - val_accuracy: 0.9171
Epoch 39/100
750/750 [==============================] - 186s 249ms/step - loss: 0.0131 - accuracy: 0.9954
- val_loss: 0.4623 - val_accuracy: 0.9150
Epoch 40/100
750/750 [==============================] - 174s 232ms/step - loss: 0.0137 - accuracy: 0.9955
- val_loss: 0.4736 - val_accuracy: 0.9168
Epoch 41/100
750/750 [==============================] - 170s 227ms/step - loss: 0.0114 - accuracy: 0.9962
- val_loss: 0.4899 - val_accuracy: 0.9165
Epoch 42/100
750/750 [==============================] - 161s 215ms/step - loss: 0.0128 - accuracy: 0.9959
- val_loss: 0.5045 - val_accuracy: 0.9123
Epoch 43/100
750/750 [==============================] - 173s 230ms/step - loss: 0.0113 - accuracy: 0.9965
- val_loss: 0.4794 - val_accuracy: 0.9162
Epoch 44/100
750/750 [==============================] - 166s 222ms/step - loss: 0.0119 - accuracy: 0.9961
- val_loss: 0.5106 - val_accuracy: 0.9137
```

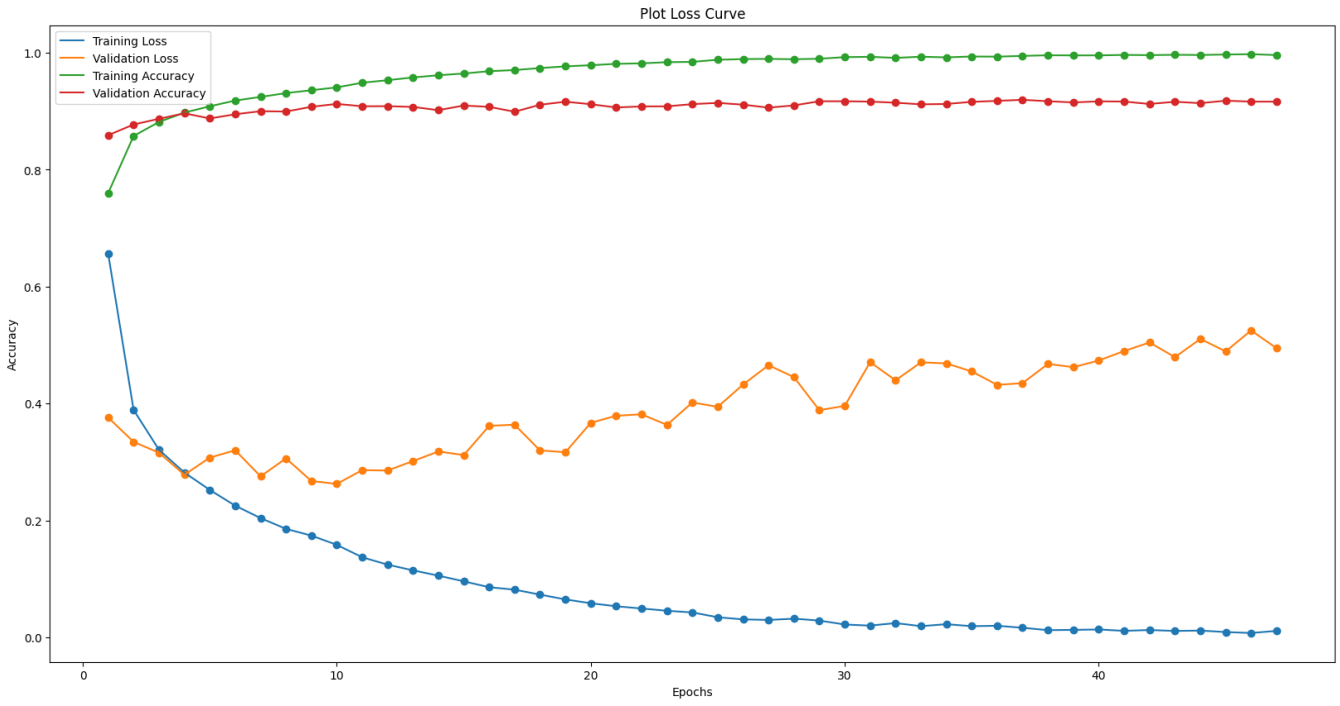
```
Epoch 45/100
750/750 [==============================] - 187s 250ms/step - loss: 0.0093 - accuracy: 0.9970
- val_loss: 0.4893 - val_accuracy: 0.9180
Epoch 46/100
750/750 [==============================] - 221s 295ms/step - loss: 0.0077 - accuracy: 0.9975
- val_loss: 0.5251 - val_accuracy: 0.9164
Epoch 47/100
750/750 [==============================] - 212s 283ms/step - loss: 0.0113 - accuracy: 0.9960
- val_loss: 0.4947 - val_accuracy: 0.9163
```

In [ ]:
```
customVGGAugModelHistory = customVGGAugModelHistory.history
best_val_idx = np.argmax(customVGGAugModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "customVGGAug"
result["Epochs"] = len(customVGGAugModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customVGGAugModelHistory["loss"][best_val_idx]
result["Val Loss"] = customVGGAugModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customVGGAugModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customVGGAugModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_37932\1215830663.py:12: 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)
```

Out[ ]:
```
{'Model Name': 'customVGGAug',
 'Epochs': 47,
 'Batch Size': 64,
 'Train Loss': 0.016653146594762802,
 'Val Loss': 0.4349783957004547,
 'Train Acc': 0.9945416450500488,
 'Val Acc': 0.9194999933242798,
 '[Train - Val] Acc': 0.07504165172576904}
```

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



Observations

Comparing our CustomVGGModel with aug to our CustomVGGModel without aug, we note that the train accuracy is higher but the test accuracy is lower but it is not as significant compared to the other models. However, the validation loss is still higher than the none augmented data. This suggest that more needs to be done like regularisation or other forms of data augmentation.

In [ ]:
```
allResults = allResults.append({
    "Model Name": "CustomVGG",
    "Epochs": 50,
    "Batch Size": 64,
    "Train Loss": 0.033576,
    "Val Loss": 3.923757,
    "Train Acc": 0.989400,
    "Val Acc": 0.505400,
    "[Train - Val] Acc": 0.484000,
},ignore_index=True)
```

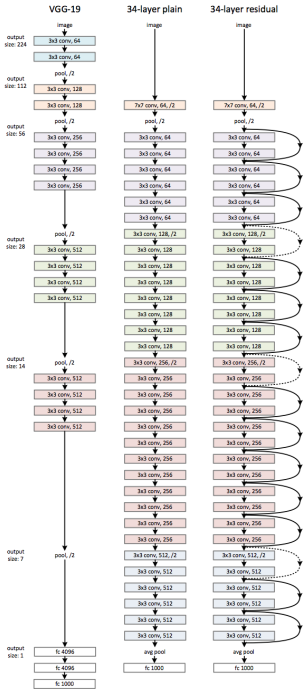
```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_10728\93878457.py:1: FutureWarning: The fra
me.append method is deprecated and will be removed from pandas in a future version. Use panda
s.concat instead.
  allResults = allResults.append({
```

CustomResNet Model

ResNet50 Model Deep Dive

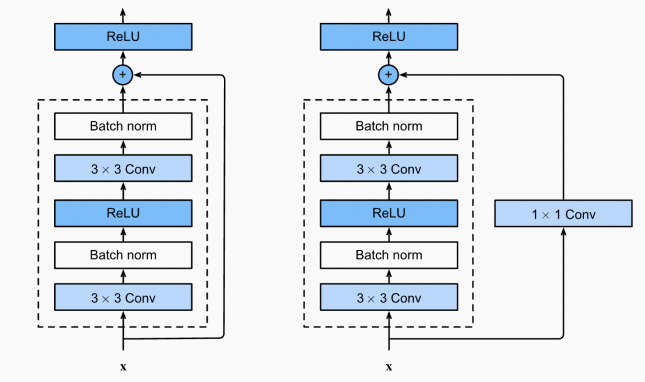
ResNets are called Residual Networks. ResNet is a special type of convolutional neural network (CNN). It was first introduced in 2015 by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their paper – "Deep Residual Learning for Image Recognition".

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.

In [ ]:
```
resNet50Model = tf.keras.applications.resnet50.ResNet50(
    include_top=True,
    weights='imagenet',
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000,
)
```

In [ ]:
```
resNet50Model.summary()
```

```
Model: "resnet50"
__________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==========================================================================================
 input_2 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

 conv1_pad (ZeroPadding2D)      (None, 230, 230, 3)  0           ['input_2[0][0]']

 conv1_conv (Conv2D)            (None, 112, 112, 64  9472        ['conv1_pad[0][0]']
                                )

 conv1_bn (BatchNormalization)  (None, 112, 112, 64  256         ['conv1_conv[0][0]']
                                )

 conv1_relu (Activation)        (None, 112, 112, 64  0           ['conv1_bn[0][0]']
                                )

 pool1_pad (ZeroPadding2D)      (None, 114, 114, 64  0           ['conv1_relu[0][0]']
                                )

 pool1_pool (MaxPooling2D)      (None, 56, 56, 64)   0           ['pool1_pad[0][0]']

 conv2_block1_1_conv (Conv2D)   (None, 56, 56, 64)   4160        ['pool1_pool[0][0]']

 conv2_block1_1_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block1_1_conv[0]
[0]']
 ization)

 conv2_block1_1_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block1_1_bn[0][0]']
 n)

 conv2_block1_2_conv (Conv2D)   (None, 56, 56, 64)   36928       ['conv2_block1_1_relu[0]
[0]']

 conv2_block1_2_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block1_2_conv[0]
[0]']
 ization)

 conv2_block1_2_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block1_2_bn[0][0]']
 n)

 conv2_block1_0_conv (Conv2D)   (None, 56, 56, 256)  16640       ['pool1_pool[0][0]']

 conv2_block1_3_conv (Conv2D)   (None, 56, 56, 256)  16640       ['conv2_block1_2_relu[0]
[0]']

 conv2_block1_0_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block1_0_conv[0]
[0]']
 ization)

 conv2_block1_3_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block1_3_conv[0]
[0]']
 ization)

 conv2_block1_add (Add)         (None, 56, 56, 256)  0           ['conv2_block1_0_bn[0][0]',
                                                                  'conv2_block1_3_bn[0][0]']

 conv2_block1_out (Activation)  (None, 56, 56, 256)  0           ['conv2_block1_add[0][0]']

 conv2_block2_1_conv (Conv2D)   (None, 56, 56, 64)   16448       ['conv2_block1_out[0][0]']

 conv2_block2_1_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block2_1_conv[0]
[0]']
```

```
 ization)

 conv2_block2_1_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block2_1_bn[0][0]']
 n)

 conv2_block2_2_conv (Conv2D)   (None, 56, 56, 64)   36928       ['conv2_block2_1_relu[0]
[0]']

 conv2_block2_2_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block2_2_conv[0]
[0]']
 ization)

 conv2_block2_2_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block2_2_bn[0][0]']
 n)

 conv2_block2_3_conv (Conv2D)   (None, 56, 56, 256)  16640       ['conv2_block2_2_relu[0]
[0]']

 conv2_block2_3_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block2_3_conv[0]
[0]']
 ization)

 conv2_block2_add (Add)         (None, 56, 56, 256)  0           ['conv2_block1_out[0][0]',
                                                                  'conv2_block2_3_bn[0][0]']

 conv2_block2_out (Activation)  (None, 56, 56, 256)  0           ['conv2_block2_add[0][0]']

 conv2_block3_1_conv (Conv2D)   (None, 56, 56, 64)   16448       ['conv2_block2_out[0][0]']

 conv2_block3_1_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block3_1_conv[0]
[0]']
 ization)

 conv2_block3_1_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block3_1_bn[0][0]']
 n)

 conv2_block3_2_conv (Conv2D)   (None, 56, 56, 64)   36928       ['conv2_block3_1_relu[0]
[0]']

 conv2_block3_2_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block3_2_conv[0]
[0]']
 ization)

 conv2_block3_2_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block3_2_bn[0][0]']
 n)

 conv2_block3_3_conv (Conv2D)   (None, 56, 56, 256)  16640       ['conv2_block3_2_relu[0]
[0]']

 conv2_block3_3_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block3_3_conv[0]
[0]']
 ization)

 conv2_block3_add (Add)         (None, 56, 56, 256)  0           ['conv2_block2_out[0][0]',
                                                                  'conv2_block3_3_bn[0][0]']

 conv2_block3_out (Activation)  (None, 56, 56, 256)  0           ['conv2_block3_add[0][0]']

 conv3_block1_1_conv (Conv2D)   (None, 28, 28, 128)  32896       ['conv2_block3_out[0][0]']

 conv3_block1_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block1_1_conv[0]
[0]']
 ization)

 conv3_block1_1_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block1_1_bn[0][0]']
 n)
```

```
 conv3_block1_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block1_1_relu[0]
[0]']

 conv3_block1_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block1_2_conv[0]
[0]']
 ization)

 conv3_block1_2_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block1_2_bn[0][0]']
 n)

 conv3_block1_0_conv (Conv2D)   (None, 28, 28, 512)  131584      ['conv2_block3_out[0][0]']

 conv3_block1_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block1_2_relu[0]
[0]']

 conv3_block1_0_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block1_0_conv[0]
[0]']
 ization)

 conv3_block1_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block1_3_conv[0]
[0]']
 ization)

 conv3_block1_add (Add)         (None, 28, 28, 512)  0          ['conv3_block1_0_bn[0][0]',
                                                                 'conv3_block1_3_bn[0][0]']

 conv3_block1_out (Activation)  (None, 28, 28, 512)  0          ['conv3_block1_add[0][0]']

 conv3_block2_1_conv (Conv2D)   (None, 28, 28, 128)  65664       ['conv3_block1_out[0][0]']

 conv3_block2_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block2_1_conv[0]
[0]']
 ization)

 conv3_block2_1_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block2_1_bn[0][0]']
 n)

 conv3_block2_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block2_1_relu[0]
[0]']

 conv3_block2_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block2_2_conv[0]
[0]']
 ization)

 conv3_block2_2_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block2_2_bn[0][0]']
 n)

 conv3_block2_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block2_2_relu[0]
[0]']

 conv3_block2_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block2_3_conv[0]
[0]']
 ization)

 conv3_block2_add (Add)         (None, 28, 28, 512)  0          ['conv3_block1_out[0][0]',
                                                                 'conv3_block2_3_bn[0][0]']

 conv3_block2_out (Activation)  (None, 28, 28, 512)  0          ['conv3_block2_add[0][0]']

 conv3_block3_1_conv (Conv2D)   (None, 28, 28, 128)  65664       ['conv3_block2_out[0][0]']

 conv3_block3_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block3_1_conv[0]
[0]']
 ization)
```

```
 conv3_block3_1_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block3_1_bn[0][0]']
 n)

 conv3_block3_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block3_1_relu[0]
[0]']

 conv3_block3_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block3_2_conv[0]
[0]']
 ization)

 conv3_block3_2_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block3_2_bn[0][0]']
 n)

 conv3_block3_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block3_2_relu[0]
[0]']

 conv3_block3_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block3_3_conv[0]
[0]']
 ization)

 conv3_block3_add (Add)         (None, 28, 28, 512)  0          ['conv3_block2_out[0][0]',
                                                                 'conv3_block3_3_bn[0][0]']

 conv3_block3_out (Activation)  (None, 28, 28, 512)  0          ['conv3_block3_add[0][0]']

 conv3_block4_1_conv (Conv2D)   (None, 28, 28, 128)  65664       ['conv3_block3_out[0][0]']

 conv3_block4_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block4_1_conv[0]
[0]']
 ization)

 conv3_block4_1_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block4_1_bn[0][0]']
 n)

 conv3_block4_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block4_1_relu[0]
[0]']

 conv3_block4_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block4_2_conv[0]
[0]']
 ization)

 conv3_block4_2_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block4_2_bn[0][0]']
 n)

 conv3_block4_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block4_2_relu[0]
[0]']

 conv3_block4_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block4_3_conv[0]
[0]']
 ization)

 conv3_block4_add (Add)         (None, 28, 28, 512)  0          ['conv3_block3_out[0][0]',
                                                                 'conv3_block4_3_bn[0][0]']

 conv3_block4_out (Activation)  (None, 28, 28, 512)  0          ['conv3_block4_add[0][0]']

 conv4_block1_1_conv (Conv2D)   (None, 14, 14, 256)  131328      ['conv3_block4_out[0][0]']

 conv4_block1_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block1_1_conv[0]
[0]']
 ization)

 conv4_block1_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block1_1_bn[0][0]']
 n)

 conv4_block1_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block1_1_relu[0]
```

```
[0]']

 conv4_block1_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block1_2_conv[0]
[0]']
 ization)

 conv4_block1_2_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block1_2_bn[0][0]']
 n)

 conv4_block1_0_conv (Conv2D)   (None, 14, 14, 1024  525312      ['conv3_block4_out[0][0]']
                                )

 conv4_block1_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block1_2_relu[0]
[0]']
                                )

 conv4_block1_0_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block1_0_conv[0]
[0]']
 ization)                       )

 conv4_block1_3_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block1_3_conv[0]
[0]']
 ization)                       )

 conv4_block1_add (Add)         (None, 14, 14, 1024  0          ['conv4_block1_0_bn[0][0]',
                                )                                 'conv4_block1_3_bn[0][0]']

 conv4_block1_out (Activation)  (None, 14, 14, 1024  0          ['conv4_block1_add[0][0]']
                                )

 conv4_block2_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block1_out[0][0]']

 conv4_block2_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block2_1_conv[0]
[0]']
 ization)

 conv4_block2_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block2_1_bn[0][0]']
 n)

 conv4_block2_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block2_1_relu[0]
[0]']

 conv4_block2_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block2_2_conv[0]
[0]']
 ization)

 conv4_block2_2_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block2_2_bn[0][0]']
 n)

 conv4_block2_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block2_2_relu[0]
[0]']
                                )

 conv4_block2_3_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block2_3_conv[0]
[0]']
 ization)                       )

 conv4_block2_add (Add)         (None, 14, 14, 1024  0          ['conv4_block1_out[0][0]',
                                )                                 'conv4_block2_3_bn[0][0]']

 conv4_block2_out (Activation)  (None, 14, 14, 1024  0          ['conv4_block2_add[0][0]']
                                )

 conv4_block3_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block2_out[0][0]']

 conv4_block3_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block3_1_conv[0]
```

```
[0]']
 ization)

 conv4_block3_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block3_1_bn[0][0]']
 n)

 conv4_block3_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block3_1_relu[0]
[0]']

 conv4_block3_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block3_2_conv[0]
[0]']
 ization)

 conv4_block3_2_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block3_2_bn[0][0]']
 n)

 conv4_block3_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block3_2_relu[0]
[0]']
                                )

 conv4_block3_3_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block3_3_conv[0]
[0]']
 ization)                       )

 conv4_block3_add (Add)         (None, 14, 14, 1024  0          ['conv4_block2_out[0][0]',
                                )                                 'conv4_block3_3_bn[0][0]']

 conv4_block3_out (Activation)  (None, 14, 14, 1024  0          ['conv4_block3_add[0][0]']
                                )

 conv4_block4_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block3_out[0][0]']

 conv4_block4_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block4_1_conv[0]
[0]']
 ization)

 conv4_block4_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block4_1_bn[0][0]']
 n)

 conv4_block4_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block4_1_relu[0]
[0]']

 conv4_block4_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block4_2_conv[0]
[0]']
 ization)

 conv4_block4_2_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block4_2_bn[0][0]']
 n)

 conv4_block4_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block4_2_relu[0]
[0]']
                                )

 conv4_block4_3_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block4_3_conv[0]
[0]']
 ization)                       )

 conv4_block4_add (Add)         (None, 14, 14, 1024  0          ['conv4_block3_out[0][0]',
                                )                                 'conv4_block4_3_bn[0][0]']

 conv4_block4_out (Activation)  (None, 14, 14, 1024  0          ['conv4_block4_add[0][0]']
                                )

 conv4_block5_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block4_out[0][0]']

 conv4_block5_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block5_1_conv[0]
```

```
[0]']
 ization)

 conv4_block5_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block5_1_bn[0][0]']
 n)

 conv4_block5_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block5_1_relu[0]
[0]']

 conv4_block5_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block5_2_conv[0]
[0]']
 ization)

 conv4_block5_2_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block5_2_bn[0][0]']
 n)

 conv4_block5_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block5_2_relu[0]
[0]']
                                )

 conv4_block5_3_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block5_3_conv[0]
[0]']
 ization)                       )

 conv4_block5_add (Add)         (None, 14, 14, 1024  0          ['conv4_block4_out[0][0]',
                                )                                 'conv4_block5_3_bn[0][0]']

 conv4_block5_out (Activation)  (None, 14, 14, 1024  0          ['conv4_block5_add[0][0]']
                                )

 conv4_block6_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block5_out[0][0]']

 conv4_block6_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block6_1_conv[0]
[0]']
 ization)

 conv4_block6_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block6_1_bn[0][0]']
 n)

 conv4_block6_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block6_1_relu[0]
[0]']

 conv4_block6_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block6_2_conv[0]
[0]']
 ization)

 conv4_block6_2_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block6_2_bn[0][0]']
 n)

 conv4_block6_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block6_2_relu[0]
[0]']
                                )

 conv4_block6_3_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block6_3_conv[0]
[0]']
 ization)                       )

 conv4_block6_add (Add)         (None, 14, 14, 1024  0          ['conv4_block5_out[0][0]',
                                )                                 'conv4_block6_3_bn[0][0]']

 conv4_block6_out (Activation)  (None, 14, 14, 1024  0          ['conv4_block6_add[0][0]']
                                )

 conv5_block1_1_conv (Conv2D)   (None, 7, 7, 512)    524800      ['conv4_block6_out[0][0]']

 conv5_block1_1_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block1_1_conv[0]
```

```
[0]']
 ization)

 conv5_block1_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block1_1_bn[0][0]']
 n)

 conv5_block1_2_conv (Conv2D)   (None, 7, 7, 512)    2359808     ['conv5_block1_1_relu[0]
[0]']

 conv5_block1_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block1_2_conv[0]
[0]']
 ization)

 conv5_block1_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block1_2_bn[0][0]']
 n)

 conv5_block1_0_conv (Conv2D)   (None, 7, 7, 2048)   2099200     ['conv4_block6_out[0][0]']

 conv5_block1_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block1_2_relu[0]
[0]']

 conv5_block1_0_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block1_0_conv[0]
[0]']
 ization)

 conv5_block1_3_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block1_3_conv[0]
[0]']
 ization)

 conv5_block1_add (Add)         (None, 7, 7, 2048)   0           ['conv5_block1_0_bn[0][0]',
                                                                  'conv5_block1_3_bn[0][0]']

 conv5_block1_out (Activation)  (None, 7, 7, 2048)   0           ['conv5_block1_add[0][0]']

 conv5_block2_1_conv (Conv2D)   (None, 7, 7, 512)    1049088     ['conv5_block1_out[0][0]']

 conv5_block2_1_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block2_1_conv[0]
[0]']
 ization)

 conv5_block2_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block2_1_bn[0][0]']
 n)

 conv5_block2_2_conv (Conv2D)   (None, 7, 7, 512)    2359808     ['conv5_block2_1_relu[0]
[0]']

 conv5_block2_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block2_2_conv[0]
[0]']
 ization)

 conv5_block2_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block2_2_bn[0][0]']
 n)

 conv5_block2_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block2_2_relu[0]
[0]']

 conv5_block2_3_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block2_3_conv[0]
[0]']
 ization)

 conv5_block2_add (Add)         (None, 7, 7, 2048)   0           ['conv5_block1_out[0][0]',
                                                                  'conv5_block2_3_bn[0][0]']

 conv5_block2_out (Activation)  (None, 7, 7, 2048)   0           ['conv5_block2_add[0][0]']

 conv5_block3_1_conv (Conv2D)   (None, 7, 7, 512)    1049088     ['conv5_block2_out[0][0]']
```

```
 conv5_block3_1_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block3_1_conv[0]
[0]']
 ization)

 conv5_block3_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_1_bn[0][0]']
 n)

 conv5_block3_2_conv (Conv2D)   (None, 7, 7, 512)    2359808     ['conv5_block3_1_relu[0]
[0]']

 conv5_block3_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block3_2_conv[0]
[0]']
 ization)

 conv5_block3_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_2_bn[0][0]']
 n)

 conv5_block3_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block3_2_relu[0]
[0]']

 conv5_block3_3_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block3_3_conv[0]
[0]']
 ization)

 conv5_block3_add (Add)         (None, 7, 7, 2048)   0           ['conv5_block2_out[0][0]',
                                                                  'conv5_block3_3_bn[0][0]']

 conv5_block3_out (Activation)  (None, 7, 7, 2048)   0           ['conv5_block3_add[0][0]']

 avg_pool (GlobalAveragePooling  (None, 2048)        0           ['conv5_block3_out[0][0]']
 2D)

 predictions (Dense)            (None, 1000)         2049000     ['avg_pool[0][0]']

==================================================================================================
=====
Total params: 25,636,712
Trainable params: 25,583,592
Non-trainable params: 53,120
__________________________________________________________________________________________________
_____
```

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 [ ]:
```python
def identity_block(x, filter):
    x_skip = x
    x = Conv2D(filter, (3, 3), padding='same')(x)
    x = BatchNormalization(axis=3)(x)
    x = Activation('relu')(x)
    x = Conv2D(filter, (3, 3), padding='same')(x)
    x = BatchNormalization(axis=3)(x)
    x = Add()([x, x_skip])
    x = Activation('relu')(x)
    return x
```

In [ ]:
```python
def convolutional_block(x, filter):
    x_skip = x
    x = Conv2D(filter, (3, 3), padding='same', strides=(2, 2))(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))(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
x = pre_processing_v1(inputs)
x = ZeroPadding2D((3, 3))(x)
x = Conv2D(64, kernel_size=7, strides=2, padding='same')(x)
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 [ ]:
```
customResNetModel.summary()
```

```
Model: "CustomResNet"
_____
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0         []

 normalization (Normalization)  (None, 28, 28, 28)     57        ['input_1[0][0]']

 zero_padding2d (ZeroPadding2D)  (None, 34, 34, 28)    0         ['normalization[7][0]']

 conv2d (Conv2D)             (None, 17, 17, 64)        87872     ['zero_padding2d[0][0]']

 batch_normalization (BatchNorm  (None, 17, 17, 64)    256       ['conv2d[0][0]']
 alization)

 activation (Activation)     (None, 17, 17, 64)        0         ['batch_normalization[0]
[0]']

 max_pooling2d (MaxPooling2D)   (None, 9, 9, 64)       0         ['activation[0][0]']

 conv2d_1 (Conv2D)           (None, 9, 9, 64)          36928     ['max_pooling2d[0][0]']

 batch_normalization_1 (BatchNo  (None, 9, 9, 64)      256       ['conv2d_1[0][0]']
 rmalization)

 activation_1 (Activation)   (None, 9, 9, 64)          0         ['batch_normalization_1[0]
[0]']

 conv2d_2 (Conv2D)           (None, 9, 9, 64)          36928     ['activation_1[0][0]']

 batch_normalization_2 (BatchNo  (None, 9, 9, 64)      256       ['conv2d_2[0][0]']
 rmalization)

 add (Add)                   (None, 9, 9, 64)          0         ['batch_normalization_2[0]
[0]',
                                                                  'max_pooling2d[0][0]']

 activation_2 (Activation)   (None, 9, 9, 64)          0         ['add[0][0]']

 conv2d_3 (Conv2D)           (None, 5, 5, 128)         73856     ['activation_2[0][0]']

 batch_normalization_3 (BatchNo  (None, 5, 5, 128)     512       ['conv2d_3[0][0]']
 rmalization)

 activation_3 (Activation)   (None, 5, 5, 128)         0         ['batch_normalization_3[0]
[0]']

 conv2d_4 (Conv2D)           (None, 5, 5, 128)         147584    ['activation_3[0][0]']

 batch_normalization_4 (BatchNo  (None, 5, 5, 128)     512       ['conv2d_4[0][0]']
 rmalization)

 conv2d_5 (Conv2D)           (None, 5, 5, 128)         8320      ['activation_2[0][0]']

 add_1 (Add)                 (None, 5, 5, 128)         0         ['batch_normalization_4[0]
[0]',
                                                                  'conv2d_5[0][0]']

 activation_4 (Activation)   (None, 5, 5, 128)         0         ['add_1[0][0]']

 conv2d_6 (Conv2D)           (None, 3, 3, 256)         295168    ['activation_4[0][0]']

 batch_normalization_5 (BatchNo  (None, 3, 3, 256)     1024      ['conv2d_6[0][0]']
 rmalization)
```

```
 activation_5 (Activation)      (None, 3, 3, 256)    0           ['batch_normalization_5[0]
[0]']

 conv2d_7 (Conv2D)              (None, 3, 3, 256)    590080      ['activation_5[0][0]']

 batch_normalization_6 (BatchNo  (None, 3, 3, 256)   1024        ['conv2d_7[0][0]']
 rmalization)

 conv2d_8 (Conv2D)              (None, 3, 3, 256)    33024       ['activation_4[0][0]']

 add_2 (Add)                    (None, 3, 3, 256)    0           ['batch_normalization_6[0]
[0]',
                                                                  'conv2d_8[0][0]']

 activation_6 (Activation)      (None, 3, 3, 256)    0           ['add_2[0][0]']

 conv2d_9 (Conv2D)              (None, 2, 2, 512)    1180160     ['activation_6[0][0]']

 batch_normalization_7 (BatchNo  (None, 2, 2, 512)   2048        ['conv2d_9[0][0]']
 rmalization)

 activation_7 (Activation)      (None, 2, 2, 512)    0           ['batch_normalization_7[0]
[0]']

 conv2d_10 (Conv2D)             (None, 2, 2, 512)    2359808     ['activation_7[0][0]']

 batch_normalization_8 (BatchNo  (None, 2, 2, 512)   2048        ['conv2d_10[0][0]']
 rmalization)

 conv2d_11 (Conv2D)             (None, 2, 2, 512)    131584      ['activation_6[0][0]']

 add_3 (Add)                    (None, 2, 2, 512)    0           ['batch_normalization_8[0]
[0]',
                                                                  'conv2d_11[0][0]']

 activation_8 (Activation)      (None, 2, 2, 512)    0           ['add_3[0][0]']

 average_pooling2d (AveragePool  (None, 1, 1, 512)   0           ['activation_8[0][0]']
 ing2D)

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

 dense (Dense)                  (None, 10)           5130        ['flatten[0][0]']

==================================================================================================
Total params: 4,994,435
Trainable params: 4,990,410
Non-trainable params: 4,025
__________________________________________________________________________________________________
```

In [ ]:
```
customResNetModelHistory = customResNetModel.fit(x_train, y_train, epochs=100,
                                                 validation_data=(x_val, y_val), batch_size=B
```

```
Epoch 1/100
750/750 [==============================] - 29s 39ms/step - loss: 0.1476 - accuracy: 0.9440 -
val_loss: 0.2648 - val_accuracy: 0.9107
Epoch 2/100
750/750 [==============================] - 21s 28ms/step - loss: 0.1299 - accuracy: 0.9501 -
val_loss: 0.2741 - val_accuracy: 0.9056
Epoch 3/100
750/750 [==============================] - 20s 27ms/step - loss: 0.1137 - accuracy: 0.9571 -
val_loss: 0.2749 - val_accuracy: 0.9098
Epoch 4/100
750/750 [==============================] - 21s 28ms/step - loss: 0.1012 - accuracy: 0.9606 -
val_loss: 0.3078 - val_accuracy: 0.9125
Epoch 5/100
750/750 [==============================] - 33s 44ms/step - loss: 0.0880 - accuracy: 0.9674 -
val_loss: 0.3122 - val_accuracy: 0.9033
Epoch 6/100
750/750 [==============================] - 22s 30ms/step - loss: 0.0812 - accuracy: 0.9688 -
val_loss: 0.3139 - val_accuracy: 0.9055
Epoch 7/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0647 - accuracy: 0.9756 -
val_loss: 0.3206 - val_accuracy: 0.9136
Epoch 8/100
750/750 [==============================] - 19s 25ms/step - loss: 0.0602 - accuracy: 0.9768 -
val_loss: 0.3397 - val_accuracy: 0.9102
Epoch 9/100
750/750 [==============================] - 19s 25ms/step - loss: 0.0584 - accuracy: 0.9781 -
val_loss: 0.3566 - val_accuracy: 0.9071
Epoch 10/100
750/750 [==============================] - 19s 26ms/step - loss: 0.0449 - accuracy: 0.9830 -
val_loss: 0.3663 - val_accuracy: 0.9172
Epoch 11/100
750/750 [==============================] - 19s 25ms/step - loss: 0.0420 - accuracy: 0.9847 -
val_loss: 0.3347 - val_accuracy: 0.9168
Epoch 12/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0334 - accuracy: 0.9881 -
val_loss: 0.3898 - val_accuracy: 0.9165
Epoch 13/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0290 - accuracy: 0.9897 -
val_loss: 0.4019 - val_accuracy: 0.9121
Epoch 14/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0246 - accuracy: 0.9915 -
val_loss: 0.4177 - val_accuracy: 0.9055
Epoch 15/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0246 - accuracy: 0.9915 -
val_loss: 0.4268 - val_accuracy: 0.9144
Epoch 16/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0251 - accuracy: 0.9913 -
val_loss: 0.4242 - val_accuracy: 0.9103
Epoch 17/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0224 - accuracy: 0.9921 -
val_loss: 0.4293 - val_accuracy: 0.9176
Epoch 18/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0151 - accuracy: 0.9945 -
val_loss: 0.4299 - val_accuracy: 0.9147
Epoch 19/100
750/750 [==============================] - 20s 26ms/step - loss: 0.0139 - accuracy: 0.9952 -
val_loss: 0.4408 - val_accuracy: 0.9171
Epoch 20/100
750/750 [==============================] - 21s 27ms/step - loss: 0.0185 - accuracy: 0.9934 -
val_loss: 0.4286 - val_accuracy: 0.9176
Epoch 21/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0133 - accuracy: 0.9953 -
val_loss: 0.4535 - val_accuracy: 0.9190
Epoch 22/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0109 - accuracy: 0.9965 -
val_loss: 0.4637 - val_accuracy: 0.9157
```

```
Epoch 23/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0104 - accuracy: 0.9965 -
val_loss: 0.4574 - val_accuracy: 0.9175
Epoch 24/100
750/750 [==============================] - 20s 27ms/step - loss: 0.0070 - accuracy: 0.9978 -
val_loss: 0.4737 - val_accuracy: 0.9207
Epoch 25/100
750/750 [==============================] - 20s 27ms/step - loss: 0.0084 - accuracy: 0.9971 -
val_loss: 0.4800 - val_accuracy: 0.9161
Epoch 26/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0070 - accuracy: 0.9978 -
val_loss: 0.4994 - val_accuracy: 0.9172
Epoch 27/100
750/750 [==============================] - 20s 27ms/step - loss: 0.0073 - accuracy: 0.9977 -
val_loss: 0.5047 - val_accuracy: 0.9162
Epoch 28/100
750/750 [==============================] - 20s 27ms/step - loss: 0.0048 - accuracy: 0.9984 -
val_loss: 0.4708 - val_accuracy: 0.9201
Epoch 29/100
750/750 [==============================] - 20s 27ms/step - loss: 0.0042 - accuracy: 0.9988 -
val_loss: 0.5018 - val_accuracy: 0.9178
Epoch 30/100
750/750 [==============================] - 20s 27ms/step - loss: 0.0056 - accuracy: 0.9983 -
val_loss: 0.4999 - val_accuracy: 0.9200
Epoch 31/100
750/750 [==============================] - 20s 27ms/step - loss: 0.0049 - accuracy: 0.9985 -
val_loss: 0.5024 - val_accuracy: 0.9195
Epoch 32/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0043 - accuracy: 0.9987 -
val_loss: 0.5018 - val_accuracy: 0.9192
Epoch 33/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0059 - accuracy: 0.9983 -
val_loss: 0.5186 - val_accuracy: 0.9180
Epoch 34/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0057 - accuracy: 0.9983 -
val_loss: 0.4920 - val_accuracy: 0.9235
Epoch 35/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0060 - accuracy: 0.9981 -
val_loss: 0.5107 - val_accuracy: 0.9194
Epoch 36/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0048 - accuracy: 0.9987 -
val_loss: 0.5069 - val_accuracy: 0.9201
Epoch 37/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0036 - accuracy: 0.9987 -
val_loss: 0.5502 - val_accuracy: 0.9193
Epoch 38/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0030 - accuracy: 0.9991 -
val_loss: 0.5593 - val_accuracy: 0.9218
Epoch 39/100
750/750 [==============================] - 21s 28ms/step - loss: 0.0015 - accuracy: 0.9996 -
val_loss: 0.5158 - val_accuracy: 0.9222
Epoch 40/100
750/750 [==============================] - 21s 28ms/step - loss: 1.9136e-04 - accuracy: 1.000
0 - val_loss: 0.5193 - val_accuracy: 0.9237
Epoch 41/100
750/750 [==============================] - 21s 28ms/step - loss: 9.7227e-05 - accuracy: 1.000
0 - val_loss: 0.5255 - val_accuracy: 0.9236
Epoch 42/100
750/750 [==============================] - 21s 28ms/step - loss: 8.0548e-05 - accuracy: 1.000
0 - val_loss: 0.5274 - val_accuracy: 0.9241
Epoch 43/100
750/750 [==============================] - 20s 27ms/step - loss: 6.7255e-05 - accuracy: 1.000
0 - val_loss: 0.5317 - val_accuracy: 0.9245
Epoch 44/100
750/750 [==============================] - 20s 27ms/step - loss: 6.1310e-05 - accuracy: 1.000
0 - val_loss: 0.5336 - val_accuracy: 0.9246
```

```
Epoch 45/100
750/750 [==============================] - 21s 28ms/step - loss: 4.7730e-05 - accuracy: 1.000
0 - val_loss: 0.5362 - val_accuracy: 0.9246
Epoch 46/100
750/750 [==============================] - 21s 28ms/step - loss: 4.5418e-05 - accuracy: 1.000
0 - val_loss: 0.5383 - val_accuracy: 0.9244
Epoch 47/100
750/750 [==============================] - 20s 27ms/step - loss: 3.7738e-05 - accuracy: 1.000
0 - val_loss: 0.5403 - val_accuracy: 0.9244
Epoch 48/100
750/750 [==============================] - 20s 27ms/step - loss: 3.6542e-05 - accuracy: 1.000
0 - val_loss: 0.5436 - val_accuracy: 0.9246
Epoch 49/100
750/750 [==============================] - 20s 27ms/step - loss: 3.6194e-05 - accuracy: 1.000
0 - val_loss: 0.5452 - val_accuracy: 0.9247
Epoch 50/100
750/750 [==============================] - 20s 27ms/step - loss: 3.2343e-05 - accuracy: 1.000
0 - val_loss: 0.5461 - val_accuracy: 0.9248
Epoch 51/100
750/750 [==============================] - 21s 28ms/step - loss: 3.2030e-05 - accuracy: 1.000
0 - val_loss: 0.5481 - val_accuracy: 0.9253
Epoch 52/100
750/750 [==============================] - 20s 27ms/step - loss: 2.8303e-05 - accuracy: 1.000
0 - val_loss: 0.5494 - val_accuracy: 0.9251
Epoch 53/100
750/750 [==============================] - 20s 27ms/step - loss: 2.6267e-05 - accuracy: 1.000
0 - val_loss: 0.5503 - val_accuracy: 0.9247
Epoch 54/100
750/750 [==============================] - 20s 27ms/step - loss: 2.7012e-05 - accuracy: 1.000
0 - val_loss: 0.5523 - val_accuracy: 0.9249
Epoch 55/100
750/750 [==============================] - 20s 27ms/step - loss: 2.5679e-05 - accuracy: 1.000
0 - val_loss: 0.5543 - val_accuracy: 0.9248
Epoch 56/100
750/750 [==============================] - 21s 28ms/step - loss: 2.3955e-05 - accuracy: 1.000
0 - val_loss: 0.5558 - val_accuracy: 0.9250
Epoch 57/100
750/750 [==============================] - 20s 27ms/step - loss: 2.7030e-05 - accuracy: 1.000
0 - val_loss: 0.5555 - val_accuracy: 0.9250
Epoch 58/100
750/750 [==============================] - 20s 27ms/step - loss: 2.1416e-05 - accuracy: 1.000
0 - val_loss: 0.5585 - val_accuracy: 0.9251
Epoch 59/100
750/750 [==============================] - 20s 27ms/step - loss: 2.3188e-05 - accuracy: 1.000
0 - val_loss: 0.5587 - val_accuracy: 0.9252
Epoch 60/100
750/750 [==============================] - 21s 28ms/step - loss: 1.9942e-05 - accuracy: 1.000
0 - val_loss: 0.5599 - val_accuracy: 0.9250
Epoch 61/100
750/750 [==============================] - 21s 27ms/step - loss: 2.0847e-05 - accuracy: 1.000
0 - val_loss: 0.5608 - val_accuracy: 0.9248
```

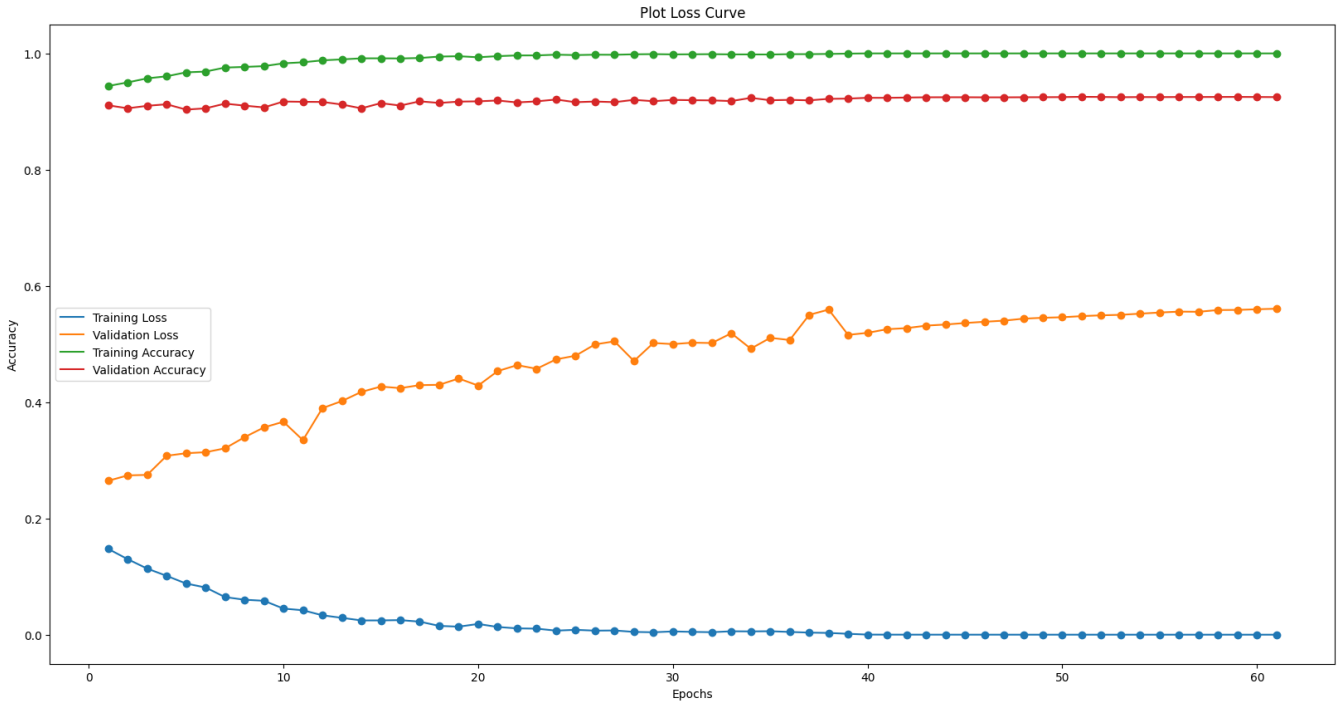
In [ ]:

```
customResNetModelHistory = customResNetModelHistory.history
best_val_idx = np.argmax(customResNetModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "customResNet"
result["Epochs"] = len(customResNetModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customResNetModelHistory["loss"][best_val_idx]
result["Val Loss"] = customResNetModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customResNetModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customResNetModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\2627699130.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'customResNet',
 'Epochs': 61,
 'Batch Size': 64,
 'Train Loss': 3.202999505447224e-05,
 'Val Loss': 0.5480642318725586,
 'Train Acc': 1.0,
 'Val Acc': 0.9253333210945129,
 '[Train - Val] Acc': 0.07466667890548706}
```

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



Observations

Comparing CustomResNet to the baseline model, we can see that the CustomResNet model is very good compared to the baseline in terms of both training and validation accuracy. However, the issue is that the validation loss function increase over epochs. This suggest that improvements need to be made to reduce the validation loss.

We also note that the model is able to be very good at a low epochs and not become too overfit [Not overfitted easily as epochs is very high before Early Stopping kicks in]

Training CustomResNet model with Data Augmentation

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

```
customResNetAugModel.summary()
```

```
Model: "CustomResNetAug"
_____
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0         []

 normalization (Normalization)  (None, 28, 28, 28)     57        ['input_1[0][0]']

 zero_padding2d (ZeroPadding2D)  (None, 34, 34, 28)    0         ['normalization[3][0]']

 conv2d (Conv2D)             (None, 17, 17, 64)        87872     ['zero_padding2d[0][0]']

 batch_normalization (BatchNorm  (None, 17, 17, 64)    256       ['conv2d[0][0]']
 alization)

 activation (Activation)     (None, 17, 17, 64)        0         ['batch_normalization[0]
[0]']

 max_pooling2d (MaxPooling2D)   (None, 9, 9, 64)       0         ['activation[0][0]']

 conv2d_1 (Conv2D)           (None, 9, 9, 64)          36928     ['max_pooling2d[0][0]']

 batch_normalization_1 (BatchNo  (None, 9, 9, 64)      256       ['conv2d_1[0][0]']
 rmalization)

 activation_1 (Activation)   (None, 9, 9, 64)          0         ['batch_normalization_1[0]
[0]']

 conv2d_2 (Conv2D)           (None, 9, 9, 64)          36928     ['activation_1[0][0]']

 batch_normalization_2 (BatchNo  (None, 9, 9, 64)      256       ['conv2d_2[0][0]']
 rmalization)

 add (Add)                   (None, 9, 9, 64)          0         ['batch_normalization_2[0]
[0]',
                                                                  'max_pooling2d[0][0]']

 activation_2 (Activation)   (None, 9, 9, 64)          0         ['add[0][0]']

 conv2d_3 (Conv2D)           (None, 5, 5, 128)         73856     ['activation_2[0][0]']

 batch_normalization_3 (BatchNo  (None, 5, 5, 128)     512       ['conv2d_3[0][0]']
 rmalization)

 activation_3 (Activation)   (None, 5, 5, 128)         0         ['batch_normalization_3[0]
[0]']

 conv2d_4 (Conv2D)           (None, 5, 5, 128)         147584    ['activation_3[0][0]']

 batch_normalization_4 (BatchNo  (None, 5, 5, 128)     512       ['conv2d_4[0][0]']
 rmalization)

 conv2d_5 (Conv2D)           (None, 5, 5, 128)         8320      ['activation_2[0][0]']

 add_1 (Add)                 (None, 5, 5, 128)         0         ['batch_normalization_4[0]
[0]',
                                                                  'conv2d_5[0][0]']

 activation_4 (Activation)   (None, 5, 5, 128)         0         ['add_1[0][0]']

 conv2d_6 (Conv2D)           (None, 3, 3, 256)         295168    ['activation_4[0][0]']

 batch_normalization_5 (BatchNo  (None, 3, 3, 256)     1024      ['conv2d_6[0][0]']
 rmalization)
```

```
 activation_5 (Activation)      (None, 3, 3, 256)    0           ['batch_normalization_5[0]
[0]']

 conv2d_7 (Conv2D)              (None, 3, 3, 256)    590080      ['activation_5[0][0]']

 batch_normalization_6 (BatchNo  (None, 3, 3, 256)   1024        ['conv2d_7[0][0]']
 rmalization)

 conv2d_8 (Conv2D)              (None, 3, 3, 256)    33024       ['activation_4[0][0]']

 add_2 (Add)                    (None, 3, 3, 256)    0           ['batch_normalization_6[0]
[0]',
                                                                  'conv2d_8[0][0]']

 activation_6 (Activation)      (None, 3, 3, 256)    0           ['add_2[0][0]']

 conv2d_9 (Conv2D)              (None, 2, 2, 512)    1180160     ['activation_6[0][0]']

 batch_normalization_7 (BatchNo  (None, 2, 2, 512)   2048        ['conv2d_9[0][0]']
 rmalization)

 activation_7 (Activation)      (None, 2, 2, 512)    0           ['batch_normalization_7[0]
[0]']

 conv2d_10 (Conv2D)             (None, 2, 2, 512)    2359808     ['activation_7[0][0]']

 batch_normalization_8 (BatchNo  (None, 2, 2, 512)   2048        ['conv2d_10[0][0]']
 rmalization)

 conv2d_11 (Conv2D)             (None, 2, 2, 512)    131584      ['activation_6[0][0]']

 add_3 (Add)                    (None, 2, 2, 512)    0           ['batch_normalization_8[0]
[0]',
                                                                  'conv2d_11[0][0]']

 activation_8 (Activation)      (None, 2, 2, 512)    0           ['add_3[0][0]']

 average_pooling2d (AveragePool  (None, 1, 1, 512)   0           ['activation_8[0][0]']
 ing2D)

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

 dense (Dense)                  (None, 10)           5130        ['flatten[0][0]']

==================================================================================================
Total params: 4,994,435
Trainable params: 4,990,410
Non-trainable params: 4,025
__________________________________________________________________________________________________
```

In [ ]:
```
customResNetAugModelHistory = customResNetAugModel.fit(x_train_aug, y_train, epochs=100,
                                                       validation_data=(x_val, y_val), batch_
```

```
Epoch 1/100
750/750 [==============================] - 301s 376ms/step - loss: 0.4846 - accuracy: 0.8183
- val_loss: 0.3962 - val_accuracy: 0.8562
Epoch 2/100
750/750 [==============================] - 284s 378ms/step - loss: 0.3339 - accuracy: 0.8755
- val_loss: 0.3058 - val_accuracy: 0.8878
Epoch 3/100
750/750 [==============================] - 192s 257ms/step - loss: 0.2815 - accuracy: 0.8932
- val_loss: 0.3163 - val_accuracy: 0.8837
Epoch 4/100
750/750 [==============================] - 543s 725ms/step - loss: 0.2458 - accuracy: 0.9071
- val_loss: 0.3008 - val_accuracy: 0.8907
Epoch 5/100
750/750 [==============================] - 708s 945ms/step - loss: 0.2190 - accuracy: 0.9162
- val_loss: 0.2796 - val_accuracy: 0.8981
Epoch 6/100
750/750 [==============================] - 1004s 1s/step - loss: 0.1955 - accuracy: 0.9254 -
val_loss: 0.3017 - val_accuracy: 0.8934
Epoch 7/100
750/750 [==============================] - 770s 1s/step - loss: 0.1710 - accuracy: 0.9343 - v
al_loss: 0.2842 - val_accuracy: 0.9008
Epoch 8/100
750/750 [==============================] - 321s 429ms/step - loss: 0.1525 - accuracy: 0.9411
- val_loss: 0.2797 - val_accuracy: 0.9012
Epoch 9/100
750/750 [==============================] - 296s 395ms/step - loss: 0.1330 - accuracy: 0.9494
- val_loss: 0.3034 - val_accuracy: 0.9051
Epoch 10/100
750/750 [==============================] - 679s 901ms/step - loss: 0.1171 - accuracy: 0.9543
- val_loss: 0.3141 - val_accuracy: 0.9022
Epoch 11/100
750/750 [==============================] - 636s 849ms/step - loss: 0.1013 - accuracy: 0.9614
- val_loss: 0.3166 - val_accuracy: 0.9079
Epoch 12/100
750/750 [==============================] - 728s 971ms/step - loss: 0.0917 - accuracy: 0.9662
- val_loss: 0.3488 - val_accuracy: 0.9011
Epoch 13/100
750/750 [==============================] - 475s 634ms/step - loss: 0.0795 - accuracy: 0.9695
- val_loss: 0.3632 - val_accuracy: 0.8988
Epoch 14/100
750/750 [==============================] - 259s 346ms/step - loss: 0.0677 - accuracy: 0.9740
- val_loss: 0.3707 - val_accuracy: 0.9012
Epoch 15/100
750/750 [==============================] - 268s 358ms/step - loss: 0.0591 - accuracy: 0.9770
- val_loss: 0.3718 - val_accuracy: 0.9023
Epoch 16/100
750/750 [==============================] - 273s 364ms/step - loss: 0.0534 - accuracy: 0.9805
- val_loss: 0.3943 - val_accuracy: 0.9013
Epoch 17/100
750/750 [==============================] - 285s 380ms/step - loss: 0.0430 - accuracy: 0.9846
- val_loss: 0.3890 - val_accuracy: 0.9043
Epoch 18/100
750/750 [==============================] - 275s 367ms/step - loss: 0.0386 - accuracy: 0.9856
- val_loss: 0.4157 - val_accuracy: 0.9023
Epoch 19/100
750/750 [==============================] - 290s 387ms/step - loss: 0.0339 - accuracy: 0.9877
- val_loss: 0.4394 - val_accuracy: 0.9047
Epoch 20/100
750/750 [==============================] - 707s 943ms/step - loss: 0.0325 - accuracy: 0.9882
- val_loss: 0.4351 - val_accuracy: 0.9064
Epoch 21/100
750/750 [==============================] - 592s 791ms/step - loss: 0.0287 - accuracy: 0.9892
- val_loss: 0.4287 - val_accuracy: 0.9097
Epoch 22/100
750/750 [==============================] - 342s 456ms/step - loss: 0.0249 - accuracy: 0.9912
- val_loss: 0.4654 - val_accuracy: 0.9000
```

```
Epoch 23/100
750/750 [==============================] - 317s 423ms/step - loss: 0.0177 - accuracy: 0.9941
- val_loss: 0.4451 - val_accuracy: 0.9086
Epoch 24/100
750/750 [==============================] - 320s 427ms/step - loss: 0.0183 - accuracy: 0.9936
- val_loss: 0.4368 - val_accuracy: 0.9105
Epoch 25/100
750/750 [==============================] - 296s 395ms/step - loss: 0.0142 - accuracy: 0.9951
- val_loss: 0.4625 - val_accuracy: 0.9102
Epoch 26/100
750/750 [==============================] - 312s 417ms/step - loss: 0.0104 - accuracy: 0.9967
- val_loss: 0.4809 - val_accuracy: 0.9122
Epoch 27/100
750/750 [==============================] - 434s 579ms/step - loss: 0.0098 - accuracy: 0.9967
- val_loss: 0.5703 - val_accuracy: 0.9016
Epoch 28/100
750/750 [==============================] - 127s 169ms/step - loss: 0.0099 - accuracy: 0.9968
- val_loss: 0.5022 - val_accuracy: 0.9072
Epoch 29/100
750/750 [==============================] - 348s 465ms/step - loss: 0.0154 - accuracy: 0.9948
- val_loss: 0.4891 - val_accuracy: 0.9078
Epoch 30/100
750/750 [==============================] - 411s 543ms/step - loss: 0.0098 - accuracy: 0.9968
- val_loss: 0.4920 - val_accuracy: 0.9083
Epoch 31/100
750/750 [==============================] - 216s 288ms/step - loss: 0.0063 - accuracy: 0.9982
- val_loss: 0.5409 - val_accuracy: 0.9033
Epoch 32/100
750/750 [==============================] - 200s 267ms/step - loss: 0.0069 - accuracy: 0.9979
- val_loss: 0.5414 - val_accuracy: 0.9090
Epoch 33/100
750/750 [==============================] - 198s 264ms/step - loss: 0.0073 - accuracy: 0.9978
- val_loss: 0.5321 - val_accuracy: 0.9094
Epoch 34/100
750/750 [==============================] - 256s 342ms/step - loss: 0.0086 - accuracy: 0.9972
- val_loss: 0.5448 - val_accuracy: 0.9067
Epoch 35/100
750/750 [==============================] - 281s 375ms/step - loss: 0.0115 - accuracy: 0.9962
- val_loss: 0.5262 - val_accuracy: 0.9074
Epoch 36/100
750/750 [==============================] - 297s 396ms/step - loss: 0.0024 - accuracy: 0.9996
- val_loss: 0.5556 - val_accuracy: 0.9066
```

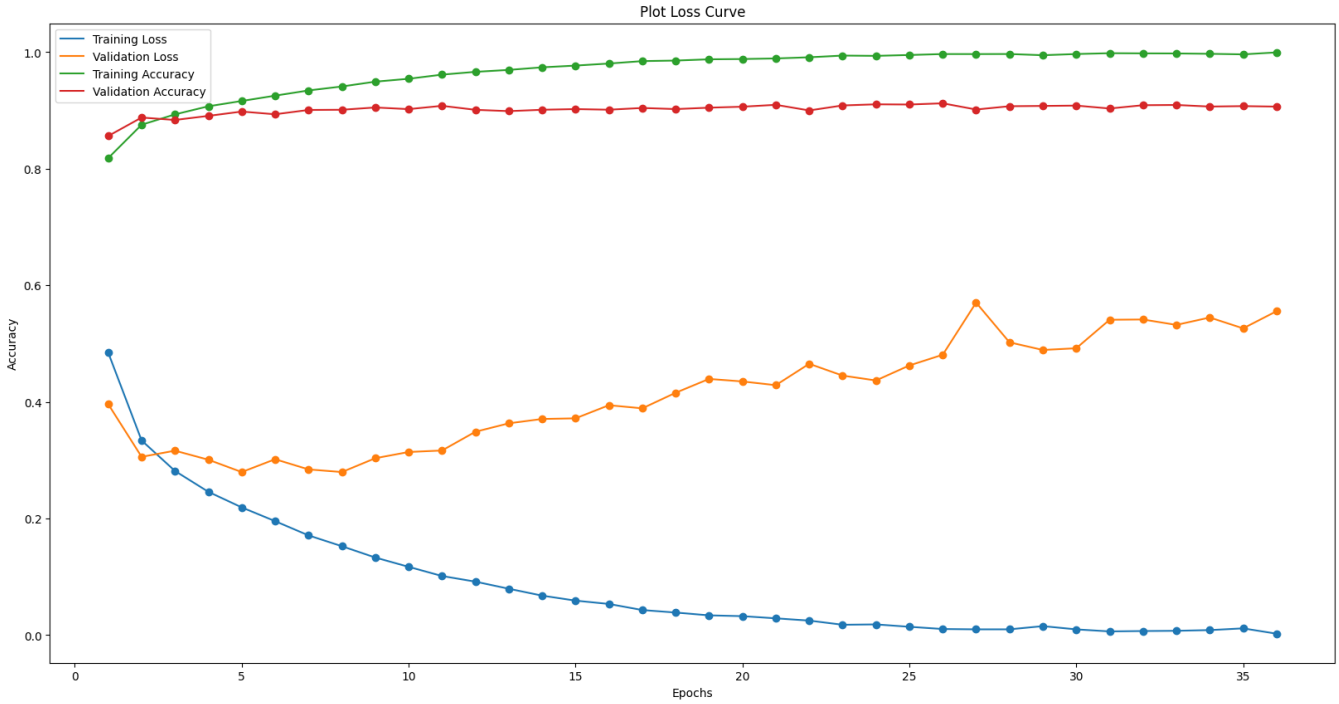
In [ ]:

```
customResNetAugModelHistory = customResNetAugModelHistory.history
best_val_idx = np.argmax(customResNetAugModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "customResNetAug"
result["Epochs"] = len(customResNetAugModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customResNetAugModelHistory["loss"][best_val_idx]
result["Val Loss"] = customResNetAugModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customResNetAugModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customResNetAugModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_37932\817363994.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'customResNetAug',
 'Epochs': 36,
 'Batch Size': 64,
 'Train Loss': 0.010412562638521194,
 'Val Loss': 0.4808880090713501,
 'Train Acc': 0.996749997138977,
 'Val Acc': 0.9122499823570251,
 '[Train - Val] Acc': 0.0845000147819519}
```

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



Observations

Comparing the CustomResNet with data augmentation and CustomResNet without data augmentation, we note that the validation loss decrease which means that the augmentation allowed the model to fit more to the validation set. However, despite that both accuracy dropped which suggest that training using only validation set affected some of the training.

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
4	customVGG	43	64	0.019438	0.399697	0.993583	0.926167	0.067417
6	customResNet	61	64	0.000032	0.548064	1.000000	0.925333	0.074667
7	customResNetAug	81	64	0.000022	0.639590	1.000000	0.912333	0.087667
5	customVGGAug	32	64	0.058219	0.373524	0.978750	0.908917	0.069833
2	conv2D	18	64	0.262191	0.313302	0.899833	0.891583	0.008250
0	baseline	35	64	0.314519	0.397700	0.884333	0.870333	0.014000
3	conv2DAug	24	64	0.353045	0.402542	0.864875	0.854833	0.010042
1	baselineAug	64	64	0.570761	0.616249	0.791417	0.803750	-0.012333

It seems like for this task, the custom VGG Model outperforms in terms of validation accuracy ResNet even though ResNet is the upgraded version of VGG. This is likely due to the the number of layers we set for the ResNet model. State of the art computer vision for ResNet are ResNet34, ResNet50 and ResNet101. As our implementation is a ResNet10, the reduction of layers due to time constraints limited the performance of the model itself. Therefore VGG despite being more simple it is very powerful model that also does not take too long to train too.

However, we will be tuning the customVGG model as well as the customResNet model and doing some model improvements to make the models better and finally deciding on which is the best final model to be evaluated

Model Improvement - customVGG

We will doing the following to tune the VGG models.

- Use different regularisation methods: L1 and L2
- Use different optimizers [SGD vs Adam]
- Using the Cosine Annealing Learning Rate Scheduler
- Use Keras Tuner to do a search to fine

Regularisation

Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting. There are 2 types of regularisation, L1 and L2. We will be trying both regularisation methods and compare which will make a better model.

L1 Regularisation - Lasso Regression

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(ReLU())
```

```
    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 = Flatten()(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=100,
                                               validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/100
750/750 [==============================] - 21s 25ms/step - loss: 10.1680 - accuracy: 0.7782 -
val_loss: 4.0738 - val_accuracy: 0.5781
Epoch 2/100
750/750 [==============================] - 18s 24ms/step - loss: 2.2595 - accuracy: 0.8344 -
val_loss: 1.8932 - val_accuracy: 0.8019
Epoch 3/100
750/750 [==============================] - 18s 24ms/step - loss: 1.4636 - accuracy: 0.8474 -
val_loss: 1.3095 - val_accuracy: 0.8443
Epoch 4/100
750/750 [==============================] - 18s 24ms/step - loss: 1.1969 - accuracy: 0.8581 -
val_loss: 1.1706 - val_accuracy: 0.8404
Epoch 5/100
750/750 [==============================] - 18s 24ms/step - loss: 1.0581 - accuracy: 0.8647 -
val_loss: 1.0598 - val_accuracy: 0.8457
Epoch 6/100
750/750 [==============================] - 18s 24ms/step - loss: 1.0022 - accuracy: 0.8666 -
val_loss: 1.0766 - val_accuracy: 0.8331
Epoch 7/100
750/750 [==============================] - 18s 24ms/step - loss: 0.9217 - accuracy: 0.8706 -
val_loss: 0.9606 - val_accuracy: 0.8601
Epoch 8/100
750/750 [==============================] - 18s 24ms/step - loss: 0.8748 - accuracy: 0.8745 -
val_loss: 0.9735 - val_accuracy: 0.8256
Epoch 9/100
750/750 [==============================] - 18s 24ms/step - loss: 0.8307 - accuracy: 0.8796 -
val_loss: 0.9115 - val_accuracy: 0.8462
Epoch 10/100
750/750 [==============================] - 18s 24ms/step - loss: 0.7972 - accuracy: 0.8811 -
val_loss: 0.9464 - val_accuracy: 0.8153
Epoch 11/100
750/750 [==============================] - 18s 24ms/step - loss: 0.7801 - accuracy: 0.8827 -
val_loss: 0.8060 - val_accuracy: 0.8661
Epoch 12/100
750/750 [==============================] - 18s 24ms/step - loss: 0.7544 - accuracy: 0.8857 -
val_loss: 0.7650 - val_accuracy: 0.8737
Epoch 13/100
750/750 [==============================] - 18s 24ms/step - loss: 0.7213 - accuracy: 0.8882 -
val_loss: 0.7925 - val_accuracy: 0.8614
Epoch 14/100
750/750 [==============================] - 18s 24ms/step - loss: 0.7051 - accuracy: 0.8886 -
val_loss: 0.7257 - val_accuracy: 0.8787
Epoch 15/100
750/750 [==============================] - 18s 25ms/step - loss: 0.6961 - accuracy: 0.8901 -
val_loss: 0.6803 - val_accuracy: 0.8913
Epoch 16/100
750/750 [==============================] - 18s 25ms/step - loss: 0.6851 - accuracy: 0.8900 -
val_loss: 0.7806 - val_accuracy: 0.8589
Epoch 17/100
750/750 [==============================] - 18s 24ms/step - loss: 0.6698 - accuracy: 0.8920 -
val_loss: 0.6329 - val_accuracy: 0.8966
Epoch 18/100
750/750 [==============================] - 18s 24ms/step - loss: 0.6562 - accuracy: 0.8915 -
val_loss: 0.6514 - val_accuracy: 0.8879
Epoch 19/100
750/750 [==============================] - 18s 24ms/step - loss: 0.6527 - accuracy: 0.8927 -
val_loss: 0.7228 - val_accuracy: 0.8678
Epoch 20/100
750/750 [==============================] - 18s 24ms/step - loss: 0.6481 - accuracy: 0.8922 -
val_loss: 0.7327 - val_accuracy: 0.8659
Epoch 21/100
750/750 [==============================] - 18s 24ms/step - loss: 0.6373 - accuracy: 0.8944 -
val_loss: 0.6956 - val_accuracy: 0.8740
Epoch 22/100
750/750 [==============================] - 18s 25ms/step - loss: 0.6236 - accuracy: 0.8951 -
val_loss: 0.6577 - val_accuracy: 0.8752
```

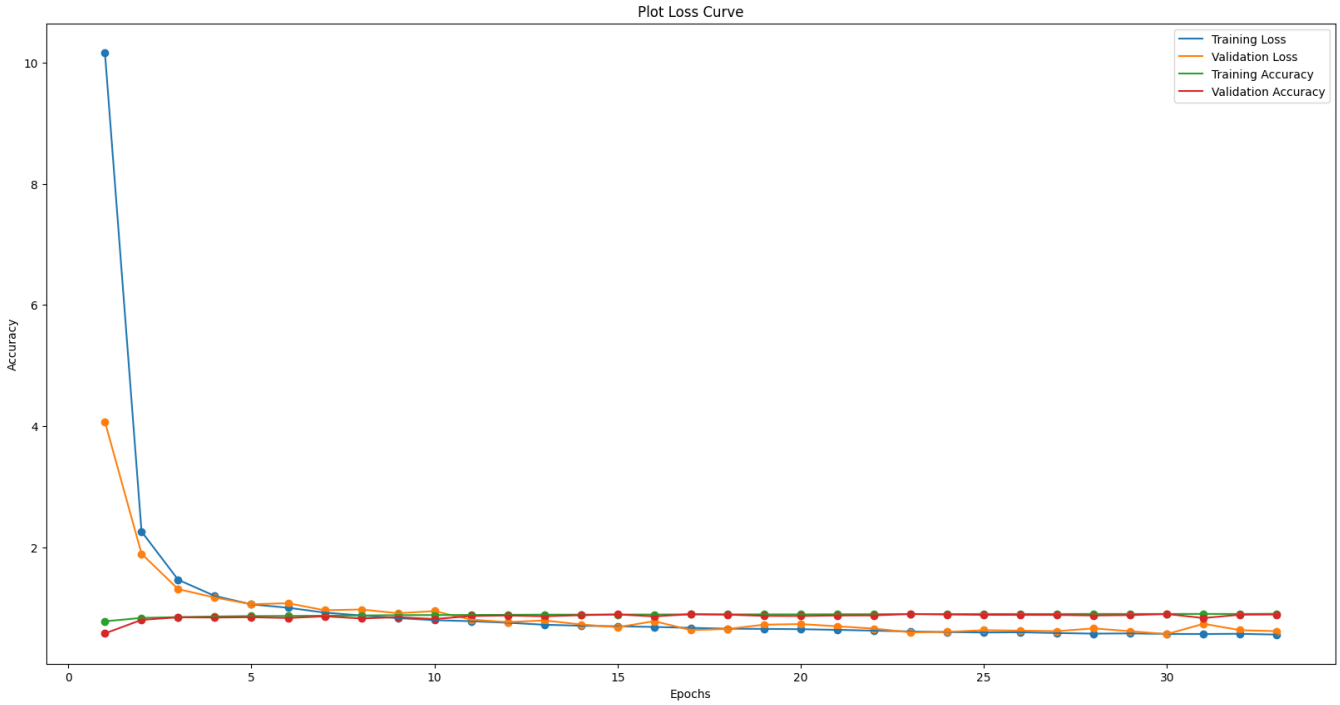
```
Epoch 23/100
750/750 [==============================] - 19s 25ms/step - loss: 0.6105 - accuracy: 0.8970 -
val_loss: 0.5957 - val_accuracy: 0.8993
Epoch 24/100
750/750 [==============================] - 18s 24ms/step - loss: 0.6000 - accuracy: 0.8966 -
val_loss: 0.6037 - val_accuracy: 0.8907
Epoch 25/100
750/750 [==============================] - 19s 25ms/step - loss: 0.5938 - accuracy: 0.8986 -
val_loss: 0.6313 - val_accuracy: 0.8847
Epoch 26/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5972 - accuracy: 0.8988 -
val_loss: 0.6240 - val_accuracy: 0.8841
Epoch 27/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5843 - accuracy: 0.8982 -
val_loss: 0.6153 - val_accuracy: 0.8832
Epoch 28/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5747 - accuracy: 0.8996 -
val_loss: 0.6617 - val_accuracy: 0.8745
Epoch 29/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5781 - accuracy: 0.8994 -
val_loss: 0.6141 - val_accuracy: 0.8807
Epoch 30/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5699 - accuracy: 0.8994 -
val_loss: 0.5677 - val_accuracy: 0.8977
Epoch 31/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5681 - accuracy: 0.8999 -
val_loss: 0.7369 - val_accuracy: 0.8324
Epoch 32/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5720 - accuracy: 0.8993 -
val_loss: 0.6319 - val_accuracy: 0.8869
Epoch 33/100
750/750 [==============================] - 18s 24ms/step - loss: 0.5588 - accuracy: 0.9019 -
val_loss: 0.6148 - val_accuracy: 0.8903
```

In [ ]:
```
customVGGL1ModelHistory = customVGGL1ModelHistory.history
best_val_idx = np.argmax(customVGGL1ModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "customVGG L1"
result["Epochs"] = len(customVGGL1ModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customVGGL1ModelHistory["loss"][best_val_idx]
result["Val Loss"] = customVGGL1ModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customVGGL1ModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customVGGL1ModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\1845154115.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'customVGG L1',
 'Epochs': 33,
 'Batch Size': 64,
 'Train Loss': 0.610473096370697,
 'Val Loss': 0.595684826374054,
 'Train Acc': 0.8969583511352539,
 'Val Acc': 0.8993333578109741,
 '[Train - Val] Acc': -0.002375006675720215}
```

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



Observations

We note that after applying the L1 Lasso Regression Regularizers, the loss functions decrease drastically which suggest that the model is not doing well in both the training and validation data as the loss is very high. We also note that by applying the L1 Lasso Regression Regulariser, the accuracy of both training and validation decrease. Therefore for this case, L1 Lasso is not suitable and does not make model better

L2 Regularisation - Ridge Regression

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(ReLU())
    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 = Flatten()(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=100,
                                               validation_data=(x_val, y_val), batch_size=BAT
```

```
Epoch 1/100
750/750 [==============================] - 20s 24ms/step - loss: 1.3171 - accuracy: 0.7913 -
val_loss: 1.1303 - val_accuracy: 0.8454
Epoch 2/100
750/750 [==============================] - 17s 23ms/step - loss: 1.0020 - accuracy: 0.8749 -
val_loss: 0.9314 - val_accuracy: 0.8814
Epoch 3/100
750/750 [==============================] - 18s 24ms/step - loss: 0.8684 - accuracy: 0.8935 -
val_loss: 0.8959 - val_accuracy: 0.8724
Epoch 4/100
750/750 [==============================] - 17s 23ms/step - loss: 0.7699 - accuracy: 0.9064 -
val_loss: 0.7622 - val_accuracy: 0.9002
Epoch 5/100
750/750 [==============================] - 18s 24ms/step - loss: 0.6940 - accuracy: 0.9135 -
val_loss: 0.7113 - val_accuracy: 0.8974
Epoch 6/100
750/750 [==============================] - 18s 23ms/step - loss: 0.6278 - accuracy: 0.9199 -
val_loss: 0.7308 - val_accuracy: 0.8843
Epoch 7/100
750/750 [==============================] - 17s 23ms/step - loss: 0.5709 - accuracy: 0.9257 -
val_loss: 0.6310 - val_accuracy: 0.8969
Epoch 8/100
750/750 [==============================] - 17s 23ms/step - loss: 0.5277 - accuracy: 0.9286 -
val_loss: 0.6696 - val_accuracy: 0.8826
Epoch 9/100
750/750 [==============================] - 17s 23ms/step - loss: 0.4878 - accuracy: 0.9335 -
val_loss: 0.5389 - val_accuracy: 0.9155
Epoch 10/100
750/750 [==============================] - 17s 23ms/step - loss: 0.4591 - accuracy: 0.9362 -
val_loss: 0.5126 - val_accuracy: 0.9134
Epoch 11/100
750/750 [==============================] - 18s 23ms/step - loss: 0.4288 - accuracy: 0.9394 -
val_loss: 0.4975 - val_accuracy: 0.9128
Epoch 12/100
750/750 [==============================] - 18s 24ms/step - loss: 0.4074 - accuracy: 0.9410 -
val_loss: 0.5165 - val_accuracy: 0.9067
Epoch 13/100
750/750 [==============================] - 18s 23ms/step - loss: 0.3864 - accuracy: 0.9435 -
val_loss: 0.4882 - val_accuracy: 0.9117
Epoch 14/100
750/750 [==============================] - 18s 23ms/step - loss: 0.3693 - accuracy: 0.9454 -
val_loss: 0.4687 - val_accuracy: 0.9133
Epoch 15/100
750/750 [==============================] - 18s 24ms/step - loss: 0.3583 - accuracy: 0.9462 -
val_loss: 0.5027 - val_accuracy: 0.9009
Epoch 16/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3480 - accuracy: 0.9487 -
val_loss: 0.5348 - val_accuracy: 0.8848
Epoch 17/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3384 - accuracy: 0.9503 -
val_loss: 0.4231 - val_accuracy: 0.9227
Epoch 18/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3289 - accuracy: 0.9510 -
val_loss: 0.4254 - val_accuracy: 0.9214
Epoch 19/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3193 - accuracy: 0.9531 -
val_loss: 0.4289 - val_accuracy: 0.9193
Epoch 20/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3192 - accuracy: 0.9523 -
val_loss: 0.4440 - val_accuracy: 0.9171
Epoch 21/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3129 - accuracy: 0.9538 -
val_loss: 0.4227 - val_accuracy: 0.9204
Epoch 22/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2989 - accuracy: 0.9598 -
val_loss: 0.4231 - val_accuracy: 0.9233
```

```
Epoch 23/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2992 - accuracy: 0.9579 -
val_loss: 0.4465 - val_accuracy: 0.9187
Epoch 24/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3012 - accuracy: 0.9584 -
val_loss: 0.4393 - val_accuracy: 0.9153
Epoch 25/100
750/750 [==============================] - 18s 23ms/step - loss: 0.2956 - accuracy: 0.9602 -
val_loss: 0.4432 - val_accuracy: 0.9097
Epoch 26/100
750/750 [==============================] - 18s 24ms/step - loss: 0.2962 - accuracy: 0.9595 -
val_loss: 0.4570 - val_accuracy: 0.9154
Epoch 27/100
750/750 [==============================] - 18s 24ms/step - loss: 0.2933 - accuracy: 0.9611 -
val_loss: 0.4335 - val_accuracy: 0.9168
Epoch 28/100
750/750 [==============================] - 18s 24ms/step - loss: 0.2865 - accuracy: 0.9636 -
val_loss: 0.4455 - val_accuracy: 0.9168
Epoch 29/100
750/750 [==============================] - 18s 24ms/step - loss: 0.2873 - accuracy: 0.9634 -
val_loss: 0.4221 - val_accuracy: 0.9241
Epoch 30/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2880 - accuracy: 0.9643 -
val_loss: 0.4857 - val_accuracy: 0.9140
Epoch 31/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2893 - accuracy: 0.9644 -
val_loss: 0.4527 - val_accuracy: 0.9188
Epoch 32/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2845 - accuracy: 0.9664 -
val_loss: 0.4489 - val_accuracy: 0.9188
Epoch 33/100
750/750 [==============================] - 17s 22ms/step - loss: 0.2831 - accuracy: 0.9665 -
val_loss: 0.4276 - val_accuracy: 0.9222
Epoch 34/100
750/750 [==============================] - 17s 22ms/step - loss: 0.2833 - accuracy: 0.9669 -
val_loss: 0.4493 - val_accuracy: 0.9201
Epoch 35/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2845 - accuracy: 0.9669 -
val_loss: 0.4674 - val_accuracy: 0.9093
Epoch 36/100
750/750 [==============================] - 18s 23ms/step - loss: 0.2870 - accuracy: 0.9657 -
val_loss: 0.4650 - val_accuracy: 0.9168
Epoch 37/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2849 - accuracy: 0.9680 -
val_loss: 0.4385 - val_accuracy: 0.9203
Epoch 38/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2769 - accuracy: 0.9711 -
val_loss: 0.4682 - val_accuracy: 0.9212
Epoch 39/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2821 - accuracy: 0.9689 -
val_loss: 0.4696 - val_accuracy: 0.9171
```

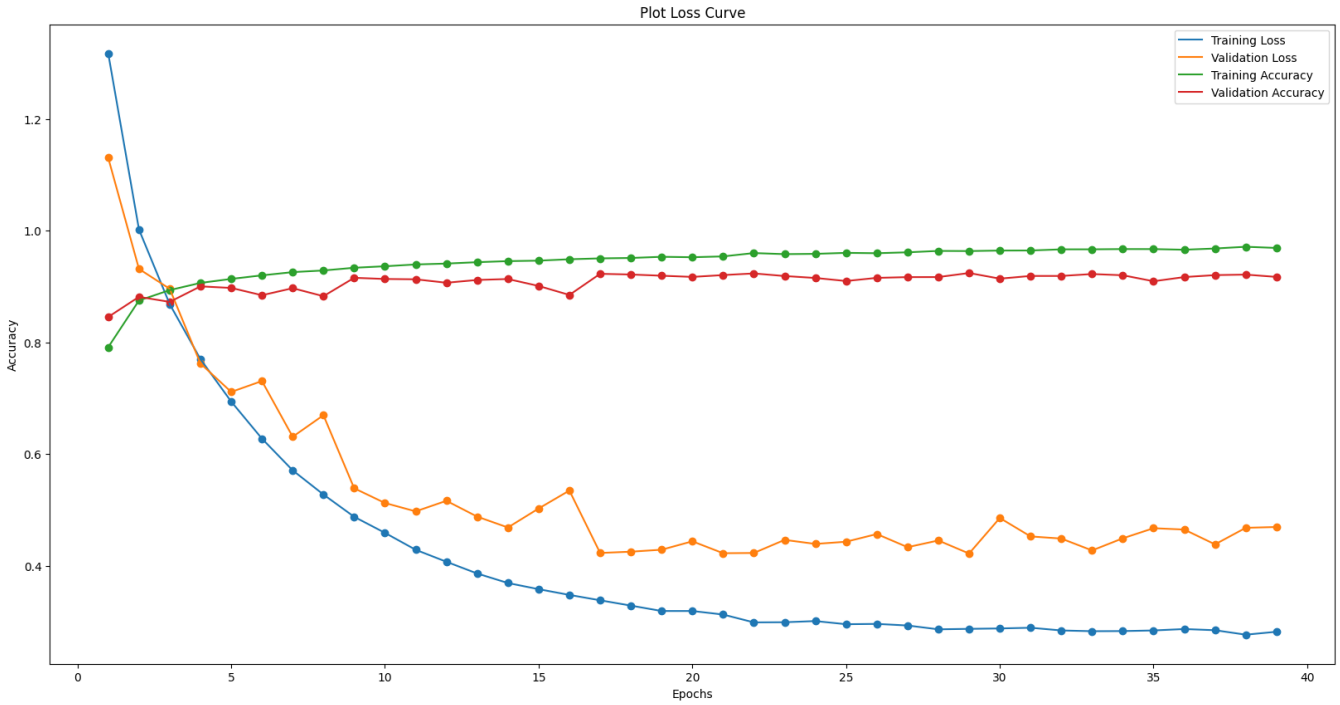
In [ ]:

```
customVGGL2ModelHistory = customVGGL2ModelHistory.history
best_val_idx = np.argmax(customVGGL2ModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "customVGG L2"
result["Epochs"] = len(customVGGL2ModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customVGGL2ModelHistory["loss"][best_val_idx]
result["Val Loss"] = customVGGL2ModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customVGGL2ModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customVGGL2ModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\2854401553.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'customVGG L2',
 'Epochs': 39,
 'Batch Size': 64,
 'Train Loss': 0.2872936725616455,
 'Val Loss': 0.4220990538597107,
 'Train Acc': 0.9633749723434448,
 'Val Acc': 0.9240833520889282,
 '[Train - Val] Acc': 0.0392916202545166}
```

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



Observations

We observed that by applying the L2 Ridge Regression method of regularisation, the accuracy of both training data and validation data has increase slightly. The number of epochs before Early Stopping is also very high which means it is unlikely to overfit. This is compared to the

For this case, as L2 has a higher accuracy for both training and validation as well as a low loss value compared to L1 and by apply the L2 regulariser, the model's performance increased. Therefore, we will be using L2 regularisers!

Optimizers

As mentioned there are 2 types of optimizers that is commonly used, we will be looking and testing the difference and making sure that SGD performs better both theoretically and practically.

Stochastic Gradient Descent

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 = Flatten()(x)
x = Dense(256, 'relu')(x)
x = Dropout(0.3)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGGSGDModel = Model(inputs=inputs, outputs=x, name="CustomVGG_SGD")
customVGGSGDModel.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                          loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:

```
customVGGSGDModelHistory = customVGGSGDModel.fit(x_train, y_train, epochs=100,
                                                 validation_data=(x_val, y_val), batch_size=B
```

```
Epoch 1/100
750/750 [==============================] - 20s 24ms/step - loss: 1.3412 - accuracy: 0.7916 -
val_loss: 1.1313 - val_accuracy: 0.8431
Epoch 2/100
750/750 [==============================] - 17s 23ms/step - loss: 1.0220 - accuracy: 0.8770 -
val_loss: 0.9534 - val_accuracy: 0.8835
Epoch 3/100
750/750 [==============================] - 17s 23ms/step - loss: 0.8929 - accuracy: 0.8942 -
val_loss: 0.8794 - val_accuracy: 0.8867
Epoch 4/100
750/750 [==============================] - 17s 23ms/step - loss: 0.7890 - accuracy: 0.9064 -
val_loss: 0.7973 - val_accuracy: 0.8905
Epoch 5/100
750/750 [==============================] - 17s 23ms/step - loss: 0.7080 - accuracy: 0.9154 -
val_loss: 0.7897 - val_accuracy: 0.8702
Epoch 6/100
750/750 [==============================] - 17s 23ms/step - loss: 0.6409 - accuracy: 0.9212 -
val_loss: 0.7369 - val_accuracy: 0.8797
Epoch 7/100
750/750 [==============================] - 17s 23ms/step - loss: 0.5865 - accuracy: 0.9261 -
val_loss: 0.6225 - val_accuracy: 0.9100
Epoch 8/100
750/750 [==============================] - 17s 23ms/step - loss: 0.5385 - accuracy: 0.9303 -
val_loss: 0.5614 - val_accuracy: 0.9166
Epoch 9/100
750/750 [==============================] - 18s 24ms/step - loss: 0.4964 - accuracy: 0.9337 -
val_loss: 0.5788 - val_accuracy: 0.9052
Epoch 10/100
750/750 [==============================] - 18s 24ms/step - loss: 0.4645 - accuracy: 0.9362 -
val_loss: 0.5755 - val_accuracy: 0.8939
Epoch 11/100
750/750 [==============================] - 18s 24ms/step - loss: 0.4324 - accuracy: 0.9395 -
val_loss: 0.5168 - val_accuracy: 0.9126
Epoch 12/100
750/750 [==============================] - 18s 24ms/step - loss: 0.4085 - accuracy: 0.9427 -
val_loss: 0.5008 - val_accuracy: 0.9112
Epoch 13/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3947 - accuracy: 0.9439 -
val_loss: 0.4936 - val_accuracy: 0.9138
Epoch 14/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3730 - accuracy: 0.9466 -
val_loss: 0.4896 - val_accuracy: 0.9102
Epoch 15/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3641 - accuracy: 0.9478 -
val_loss: 0.4656 - val_accuracy: 0.9172
Epoch 16/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3501 - accuracy: 0.9487 -
val_loss: 0.4390 - val_accuracy: 0.9187
Epoch 17/100
750/750 [==============================] - 17s 23ms/step - loss: 0.3363 - accuracy: 0.9515 -
val_loss: 0.4669 - val_accuracy: 0.9143
Epoch 18/100
750/750 [==============================] - 18s 24ms/step - loss: 0.3290 - accuracy: 0.9528 -
val_loss: 0.4690 - val_accuracy: 0.9144
Epoch 19/100
750/750 [==============================] - 18s 24ms/step - loss: 0.3206 - accuracy: 0.9539 -
val_loss: 0.4356 - val_accuracy: 0.9193
Epoch 20/100
750/750 [==============================] - 18s 24ms/step - loss: 0.3121 - accuracy: 0.9565 -
val_loss: 0.4715 - val_accuracy: 0.9110
Epoch 21/100
750/750 [==============================] - 18s 24ms/step - loss: 0.3090 - accuracy: 0.9567 -
val_loss: 0.4480 - val_accuracy: 0.9153
Epoch 22/100
750/750 [==============================] - 18s 23ms/step - loss: 0.3075 - accuracy: 0.9560 -
val_loss: 0.4517 - val_accuracy: 0.9148
```

```
Epoch 23/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2974 - accuracy: 0.9598 -
val_loss: 0.4400 - val_accuracy: 0.9161
Epoch 24/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2972 - accuracy: 0.9591 -
val_loss: 0.4969 - val_accuracy: 0.9127
Epoch 25/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2952 - accuracy: 0.9605 -
val_loss: 0.4767 - val_accuracy: 0.9092
Epoch 26/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2929 - accuracy: 0.9609 -
val_loss: 0.4394 - val_accuracy: 0.9202
Epoch 27/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2878 - accuracy: 0.9629 -
val_loss: 0.4499 - val_accuracy: 0.9168
Epoch 28/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2843 - accuracy: 0.9639 -
val_loss: 0.4743 - val_accuracy: 0.9133
Epoch 29/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2835 - accuracy: 0.9647 -
val_loss: 0.4441 - val_accuracy: 0.9192
Epoch 30/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2862 - accuracy: 0.9640 -
val_loss: 0.4515 - val_accuracy: 0.9198
Epoch 31/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2845 - accuracy: 0.9661 -
val_loss: 0.5008 - val_accuracy: 0.9032
Epoch 32/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2774 - accuracy: 0.9679 -
val_loss: 0.4718 - val_accuracy: 0.9167
Epoch 33/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2791 - accuracy: 0.9669 -
val_loss: 0.4354 - val_accuracy: 0.9217
Epoch 34/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2788 - accuracy: 0.9663 -
val_loss: 0.5000 - val_accuracy: 0.9032
Epoch 35/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2850 - accuracy: 0.9664 -
val_loss: 0.4519 - val_accuracy: 0.9236
Epoch 36/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2756 - accuracy: 0.9693 -
val_loss: 0.4601 - val_accuracy: 0.9197
Epoch 37/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2771 - accuracy: 0.9701 -
val_loss: 0.4761 - val_accuracy: 0.9068
Epoch 38/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2774 - accuracy: 0.9693 -
val_loss: 0.4644 - val_accuracy: 0.9198
Epoch 39/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2825 - accuracy: 0.9676 -
val_loss: 0.4621 - val_accuracy: 0.9186
Epoch 40/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2758 - accuracy: 0.9710 -
val_loss: 0.4510 - val_accuracy: 0.9218
Epoch 41/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2751 - accuracy: 0.9705 -
val_loss: 0.4771 - val_accuracy: 0.9178
Epoch 42/100
750/750 [==============================] - 17s 22ms/step - loss: 0.2844 - accuracy: 0.9691 -
val_loss: 0.4594 - val_accuracy: 0.9187
Epoch 43/100
750/750 [==============================] - 17s 23ms/step - loss: 0.2747 - accuracy: 0.9716 -
val_loss: 0.4467 - val_accuracy: 0.9251
Epoch 44/100
750/750 [==============================] - 17s 22ms/step - loss: 0.2841 - accuracy: 0.9691 -
val_loss: 0.4635 - val_accuracy: 0.9210
```

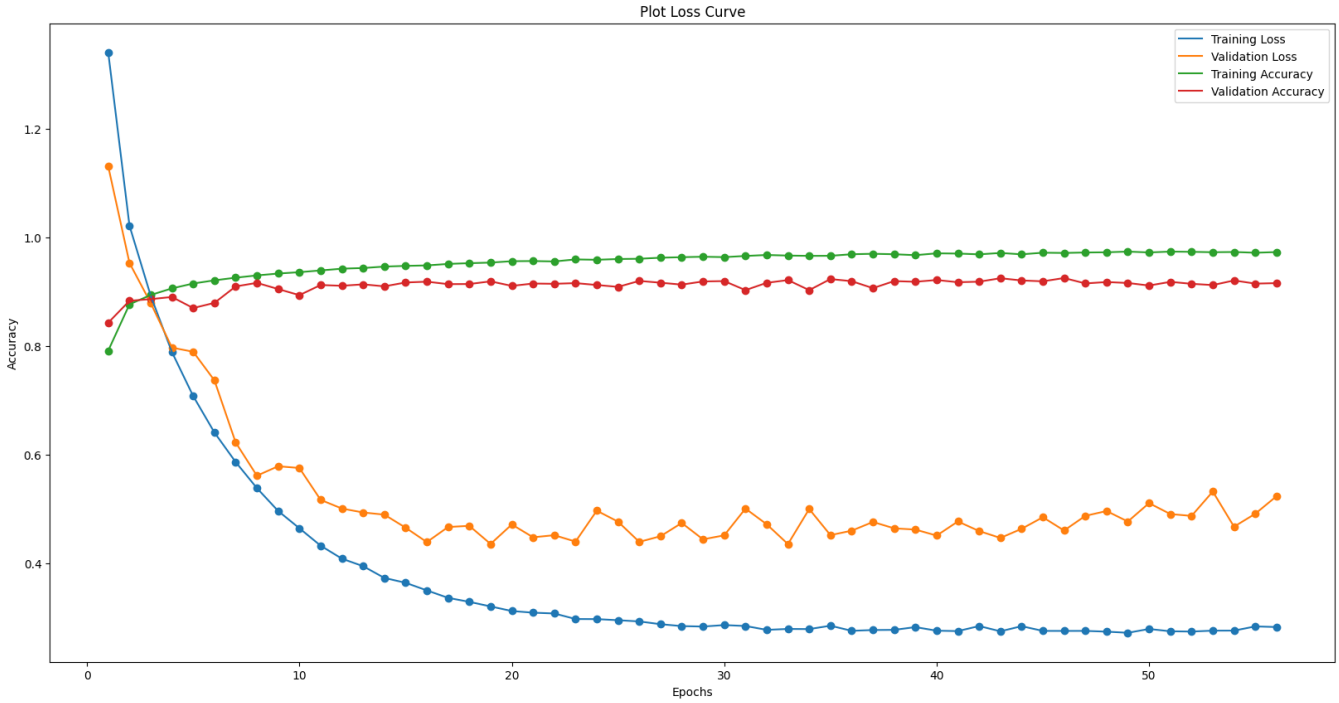
```
Epoch 45/100
750/750 [==============================] - 17s 22ms/step - loss: 0.2755 - accuracy: 0.9721 -
val_loss: 0.4848 - val_accuracy: 0.9194
Epoch 46/100
750/750 [==============================] - 16s 21ms/step - loss: 0.2754 - accuracy: 0.9717 -
val_loss: 0.4604 - val_accuracy: 0.9255
Epoch 47/100
750/750 [==============================] - 16s 21ms/step - loss: 0.2755 - accuracy: 0.9725 -
val_loss: 0.4874 - val_accuracy: 0.9155
Epoch 48/100
750/750 [==============================] - 16s 21ms/step - loss: 0.2741 - accuracy: 0.9729 -
val_loss: 0.4963 - val_accuracy: 0.9181
Epoch 49/100
750/750 [==============================] - 16s 22ms/step - loss: 0.2720 - accuracy: 0.9742 -
val_loss: 0.4769 - val_accuracy: 0.9163
Epoch 50/100
750/750 [==============================] - 16s 22ms/step - loss: 0.2790 - accuracy: 0.9726 -
val_loss: 0.5109 - val_accuracy: 0.9117
Epoch 51/100
750/750 [==============================] - 16s 22ms/step - loss: 0.2748 - accuracy: 0.9742 -
val_loss: 0.4907 - val_accuracy: 0.9184
Epoch 52/100
750/750 [==============================] - 16s 22ms/step - loss: 0.2743 - accuracy: 0.9737 -
val_loss: 0.4873 - val_accuracy: 0.9148
Epoch 53/100
750/750 [==============================] - 16s 22ms/step - loss: 0.2759 - accuracy: 0.9730 -
val_loss: 0.5322 - val_accuracy: 0.9125
Epoch 54/100
750/750 [==============================] - 16s 22ms/step - loss: 0.2760 - accuracy: 0.9734 -
val_loss: 0.4674 - val_accuracy: 0.9210
Epoch 55/100
750/750 [==============================] - 17s 22ms/step - loss: 0.2837 - accuracy: 0.9721 -
val_loss: 0.4913 - val_accuracy: 0.9152
Epoch 56/100
750/750 [==============================] - 17s 22ms/step - loss: 0.2826 - accuracy: 0.9733 -
val_loss: 0.5236 - val_accuracy: 0.9161
```

In [ ]:
```
customVGGSGDModelHistory = customVGGSGDModelHistory.history
best_val_idx = np.argmax(customVGGSGDModelHistory["val_accuracy"])
result = {}
result["Model Name"] = "customVGG SGD"
result["Epochs"] = len(customVGGSGDModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customVGGSGDModelHistory["loss"][best_val_idx]
result["Val Loss"] = customVGGSGDModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customVGGSGDModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customVGGSGDModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\2655392128.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'customVGG SGD',
 'Epochs': 56,
 'Batch Size': 64,
 'Train Loss': 0.2754226624965668,
 'Val Loss': 0.4603902995586395,
 'Train Acc': 0.971666693687439,
 'Val Acc': 0.9254999756813049,
 '[Train - Val] Acc': 0.04616671800613403}
```

In [ ]:
```
plot_loss_curve(customVGGSGDModelHistory)
plt.show()
```



Observations

As we can see that the model becomes generalise as epochs increases. It seems that the SGD optimizer is able to reduce the overall loss and improve accuracy over time as what an optimizer is suppose to do

Adam

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 = Flatten()(x)
x = Dense(256, 'relu')(x)
x = Dropout(0.3)(x)
x = Dense(NUM_CLASS, 'softmax')(x)
customVGGAdamModel = Model(inputs=inputs, outputs=x, name="CustomVGG_Adam")
customVGGAdamModel.compile(optimizer=Adam(learning_rate=0.01),
                           loss='categorical_crossentropy', metrics=['accuracy'])
```

In [ ]:
```
customVGGAdamModelHistory = customVGGAdamModel.fit(x_train, y_train, epochs=100,
                                                   validation_data=(x_val, y_val), batch_size
```

```
Epoch 1/100
750/750 [==============================] - 20s 24ms/step - loss: 1.9327 - accuracy: 0.7395 -
val_loss: 1.4253 - val_accuracy: 0.6770
Epoch 2/100
750/750 [==============================] - 17s 23ms/step - loss: 1.1375 - accuracy: 0.8115 -
val_loss: 1.5565 - val_accuracy: 0.6328
Epoch 3/100
750/750 [==============================] - 17s 23ms/step - loss: 0.8518 - accuracy: 0.8286 -
val_loss: 0.9395 - val_accuracy: 0.7751
Epoch 4/100
750/750 [==============================] - 17s 22ms/step - loss: 0.7927 - accuracy: 0.8326 -
val_loss: 1.1907 - val_accuracy: 0.7113
Epoch 5/100
750/750 [==============================] - 17s 22ms/step - loss: 0.7538 - accuracy: 0.8353 -
val_loss: 2.0343 - val_accuracy: 0.5506
Epoch 6/100
750/750 [==============================] - 17s 22ms/step - loss: 0.7133 - accuracy: 0.8389 -
val_loss: 0.9874 - val_accuracy: 0.7458
Epoch 7/100
750/750 [==============================] - 17s 22ms/step - loss: 0.6649 - accuracy: 0.8471 -
val_loss: 0.7647 - val_accuracy: 0.8082
Epoch 8/100
750/750 [==============================] - 17s 22ms/step - loss: 0.6268 - accuracy: 0.8513 -
val_loss: 0.6257 - val_accuracy: 0.8466
Epoch 9/100
750/750 [==============================] - 17s 23ms/step - loss: 0.6172 - accuracy: 0.8535 -
val_loss: 0.6239 - val_accuracy: 0.8380
Epoch 10/100
750/750 [==============================] - 17s 23ms/step - loss: 0.5823 - accuracy: 0.8571 -
val_loss: 1.2890 - val_accuracy: 0.6053
Epoch 11/100
750/750 [==============================] - 17s 23ms/step - loss: 0.5726 - accuracy: 0.8592 -
val_loss: 0.5713 - val_accuracy: 0.8692
Epoch 12/100
750/750 [==============================] - 17s 22ms/step - loss: 0.5594 - accuracy: 0.8623 -
val_loss: 2.6322 - val_accuracy: 0.5363
Epoch 13/100
750/750 [==============================] - 17s 22ms/step - loss: 0.5581 - accuracy: 0.8604 -
val_loss: 0.6405 - val_accuracy: 0.8298
Epoch 14/100
750/750 [==============================] - 16s 22ms/step - loss: 0.5491 - accuracy: 0.8627 -
val_loss: 0.5571 - val_accuracy: 0.8529
Epoch 15/100
750/750 [==============================] - 16s 22ms/step - loss: 0.5461 - accuracy: 0.8623 -
val_loss: 0.6906 - val_accuracy: 0.8221
Epoch 16/100
750/750 [==============================] - 16s 22ms/step - loss: 0.5428 - accuracy: 0.8635 -
val_loss: 0.5147 - val_accuracy: 0.8648
Epoch 17/100
750/750 [==============================] - 16s 22ms/step - loss: 0.5488 - accuracy: 0.8614 -
val_loss: 0.5751 - val_accuracy: 0.8508
Epoch 18/100
750/750 [==============================] - 16s 22ms/step - loss: 0.5403 - accuracy: 0.8666 -
val_loss: 0.5824 - val_accuracy: 0.8512
Epoch 19/100
750/750 [==============================] - 17s 22ms/step - loss: 0.5392 - accuracy: 0.8657 -
val_loss: 0.6706 - val_accuracy: 0.8097
Epoch 20/100
750/750 [==============================] - 17s 22ms/step - loss: 0.5333 - accuracy: 0.8653 -
val_loss: 1.4327 - val_accuracy: 0.6838
Epoch 21/100
750/750 [==============================] - 17s 22ms/step - loss: 0.5328 - accuracy: 0.8669 -
val_loss: 0.5954 - val_accuracy: 0.8398
```

In [ ]:

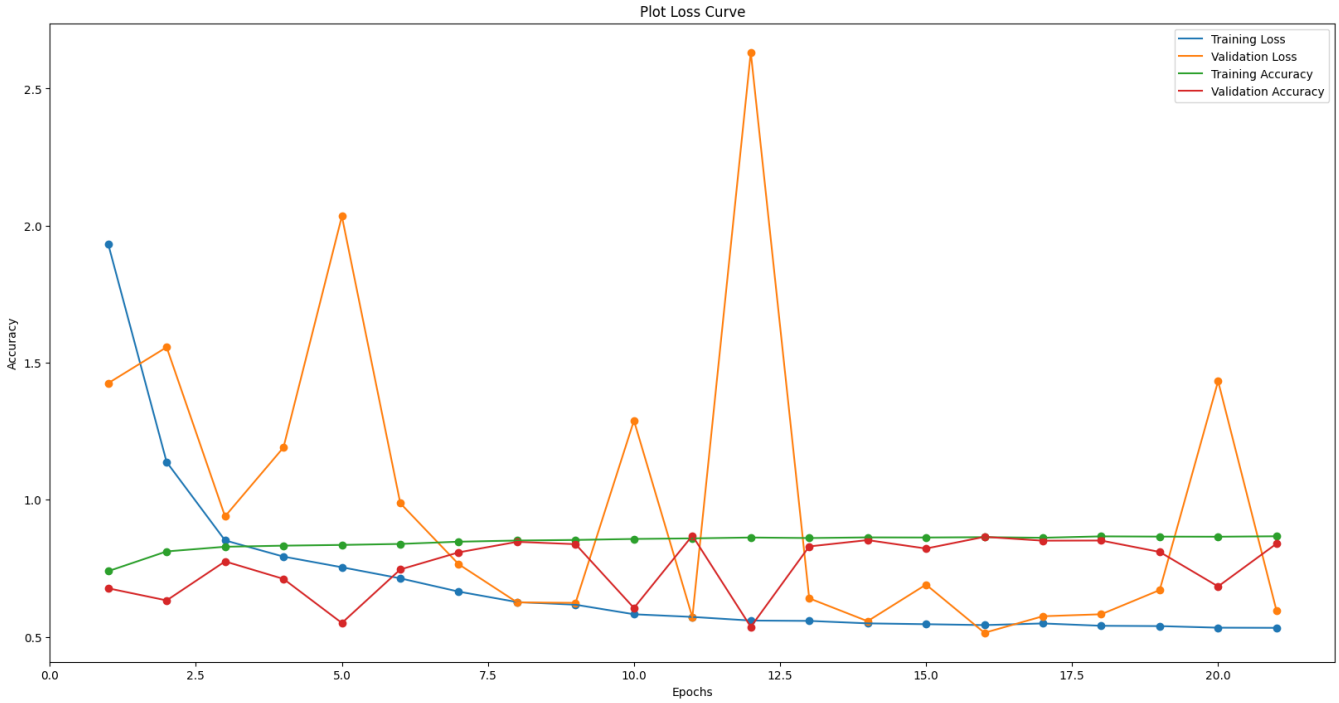
```
customVGGAdamModelHistory = customVGGAdamModelHistory.history
best_val_idx = np.argmax(customVGGAdamModelHistory["val_accuracy"])
```

```
result = {}
result["Model Name"] = "customVGG Adam"
result["Epochs"] = len(customVGGAdamModelHistory["loss"])
result["Batch Size"] = BATCH_SIZE
result["Train Loss"] = customVGGAdamModelHistory["loss"][best_val_idx]
result["Val Loss"] = customVGGAdamModelHistory["val_loss"][best_val_idx]
result["Train Acc"] = customVGGAdamModelHistory["accuracy"][best_val_idx]
result["Val Acc"] = customVGGAdamModelHistory["val_accuracy"][best_val_idx]
result["[Train - Val] Acc"] = result["Train Acc"] - result["Val Acc"]
allResults = allResults.append(result, ignore_index=True)
result
```

```
C:\Users\Soh Hong Yu\AppData\Local\Temp\ipykernel_2760\1521673524.py:12: FutureWarning: The f
rame.append method is deprecated and will be removed from pandas in a future version. Use pan
das.concat instead.
  allResults = allResults.append(result, ignore_index=True)
```

Out[ ]:
```
{'Model Name': 'customVGG Adam',
 'Epochs': 21,
 'Batch Size': 64,
 'Train Loss': 0.5726111531257629,
 'Val Loss': 0.5712665319442749,
 'Train Acc': 0.85916668176651,
 'Val Acc': 0.8692499995231628,
 '[Train - Val] Acc': -0.010083317756652832}
```

In [ ]:
```
plot_loss_curve(customVGGAdamModelHistory)
plt.show()
```



Observations

An optimizer is allowed to modify each weights and minimize the loss function. This will reduce overall loss and improve accuracy. However, we can see from using the Adam optimizer, even though the accuracy loss is low, the validation loss is out of this world and is very bad as it does not fit well with the model. We can also see it affecting the validation accuracy too. Even though the model is very good at predicting the training data, when it comes to the validation data, there are drastic drop in performance which suggest that by using the Adam optimizer, it has made the model overfit to the training data despite having L2 Regularisation applied to it which reduces overfitting.

In Summary, we will be using the SGD optimizer as it very good at improving the accuracy and reducing overall loss compared to Adam and it has been a good choice since the begin to be using SGD as the

optimizer. This suggest that both theoretically and practically SGD is better for this image classification problem.

Keras Tuner

KerasTuner is an easy-to-use, scalable hyperparameter optimization framework that solves the pain points of hyperparameter search. Easily configure your search space with a define-by-run syntax, then leverage one of the available search algorithms to find the best hyperparameter values for your models. KerasTuner comes with Bayesian Optimization, Hyperband, and Random Search algorithms built-in, and is also designed to be easy for researchers to extend in order to experiment with new search algorithms.

To use keras tuner effectively, we will need to make the model and store it inside of a function

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

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

In [ ]:
```
def tune_vgg_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_l2(2, 32, weight_decay=weight_decay)(x)
    x = vgg_block_l2(2, 64, weight_decay=weight_decay)(x)
    x = vgg_block_l2(3, 128, weight_decay=weight_decay)(x)
    x = vgg_block_l2(3, 256, weight_decay=weight_decay)(x)
    x = Flatten()(x)
    x = Dense(256, 'relu')(x)
    x = Dropout(0.3)(x)
    x = Dense(NUM_CLASS, 'softmax')(x)
    model = Model(inputs=inputs, outputs=x)
    model.compile(optimizer=optimizer,
                  loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

In [ ]:
```
VGGTuner = kt.Hyperband(tune_vgg_model, objective="val_accuracy",
                        max_epochs=50, overwrite=True, project_name="fashion_vgg")
```

In [ ]:
```
VGGTuner.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)
    ]
)
VGGTuner.results_summary(num_trials=3)
```

```
Trial 90 Complete [00h 15m 10s]
val_accuracy: 0.9337499737739563

Best val_accuracy So Far: 0.9364166855812073
Total elapsed time: 04h 23m 28s
INFO:tensorflow:Oracle triggered exit
Results summary
Results in .\fashion_vgg
Showing 3 best trials
<keras_tuner.engine.objective.Objective object at 0x000001E47BF335B0>
Trial summary
Hyperparameters:
weight_decay: 0.008229251891996288
learning_rate: 0.0035152323387747804
tuner/epochs: 50
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.9364166855812073
Trial summary
Hyperparameters:
weight_decay: 0.006534537656818744
learning_rate: 0.02987361022983302
tuner/epochs: 50
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.9350833296775818
Trial summary
Hyperparameters:
weight_decay: 0.009550786463434349
learning_rate: 0.02317351776944327
tuner/epochs: 50
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.9337499737739563
```

Model Improvement - customResNet

We will doing the following to tune the ResNet models.

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

In [ ]:
```python
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
```

In [ ]:
```python
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
```

Keras Tuner

KerasTuner is an easy-to-use, scalable hyperparameter optimization framework that solves the pain points of hyperparameter search. Easily configure your search space with a define-by-run syntax, then leverage one of the available search algorithms to find the best hyperparameter values for your models. KerasTuner comes with Bayesian Optimization, Hyperband, and Random Search algorithms built-in, and is also designed to be easy for researchers to extend in order to experiment with new search algorithms.

In [ ]:
```
def tune_resnet_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 = 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 = [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, weight_decay)
        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, weight_decay)
    x = AveragePooling2D((2, 2), padding='same')(x)
    x = Flatten()(x)
    x = Dense(NUM_CLASS, 'softmax')(x)
    model = Model(inputs=inputs, outputs=x)
    model.compile(optimizer=optimizer,
                  loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

In [ ]:
```
ResNetTuner = kt.Hyperband(tune_resnet_model, objective="val_accuracy",
                           max_epochs=50, overwrite=True, project_name="fashion_resnet")
```

In [ ]:
```
ResNetTuner.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)
```

```
    ]
)
ResNetTuner.results_summary(num_trials=3)
```

```
Trial 90 Complete [00h 20m 46s]
val_accuracy: 0.9224166870117188

Best val_accuracy So Far: 0.9243333339691162
Total elapsed time: 09h 41m 13s
INFO:tensorflow:Oracle triggered exit
Results summary
Results in .\fashion_resnet
Showing 3 best trials
<keras_tuner.engine.objective.Objective object at 0x000001E4FA391340>
Trial summary
Hyperparameters:
weight_decay: 0.0017469364666108822
learning_rate: 0.02995853178427231
tuner/epochs: 50
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.9243333339691162
Trial summary
Hyperparameters:
weight_decay: 0.0008436048029335887
learning_rate: 0.08294903912014776
tuner/epochs: 50
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.9242500066757202
Trial summary
Hyperparameters:
weight_decay: 0.0019981184550969546
learning_rate: 0.0016378962306363488
tuner/epochs: 50
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.9225833415985107
```

Tuned Model Selection

After tuning the models, we will compare both models in terms of validation accuracy. We can see that customVGG is better than customResNet model as the validation accuracy is 0.9337499737739563 while the customResNet model has a validation accuracy of 0.9243333339691162.

Therefore after careful selection consideration, we will be using the customVGG model instead of the customResNet model.

In [ ]:
```
vgg_model = VGGTuner.get_best_models()[0]
resnet_model = ResNetTuner.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 [ ]:
```
vgg_model.save('models/customVGG - Final')
resnet_model.save('models/customResNet')
```

```
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_con
volution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convol
ution_op while saving (showing 5 of 10). These functions will not be directly callable after
loading.
INFO:tensorflow:Assets written to: models/customVGG - Final\assets
INFO:tensorflow:Assets written to: models/customVGG - Final\assets
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_con
volution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convol
ution_op while saving (showing 5 of 12). These functions will not be directly callable after
loading.
INFO:tensorflow:Assets written to: models/customResNet\assets
INFO:tensorflow:Assets written to: models/customResNet\assets
```

In [ ]:
```
vgg_model.save('models/customVGG - Final.h5')
resnet_model.save('models/customResNet.h5')
```

Initiate model after tuning and saving

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

final_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 normalization (Normalizatio  (None, 28, 28, 28)       57
 n)

 sequential (Sequential)     (None, 14, 14, 32)        17600

 sequential_1 (Sequential)   (None, 7, 7, 64)          55936

 sequential_2 (Sequential)   (None, 3, 3, 128)         370560

 sequential_3 (Sequential)   (None, 1, 1, 256)         1478400

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 256)               65792

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 1,990,915
Trainable params: 1,988,170
Non-trainable params: 2,745
_________________________________________________________________
```

Testing Set

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

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

Out[ ]:
```
313/313 [==============================] - 8s 13ms/step - loss: 0.5096 - accuracy: 0.9331
[0.5096258521080017, 0.9330999851226807]
```

Observations

We note that the model loss and the accuracy is very high and is quiet similar to the loss and accuracy given by the validation set. This suggest that the model is good at generalising and predicting which image is what clothing item.

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

```
313/313 [==============================] - 160s 496ms/step
```

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

In [ ]:
```
print(report)
```

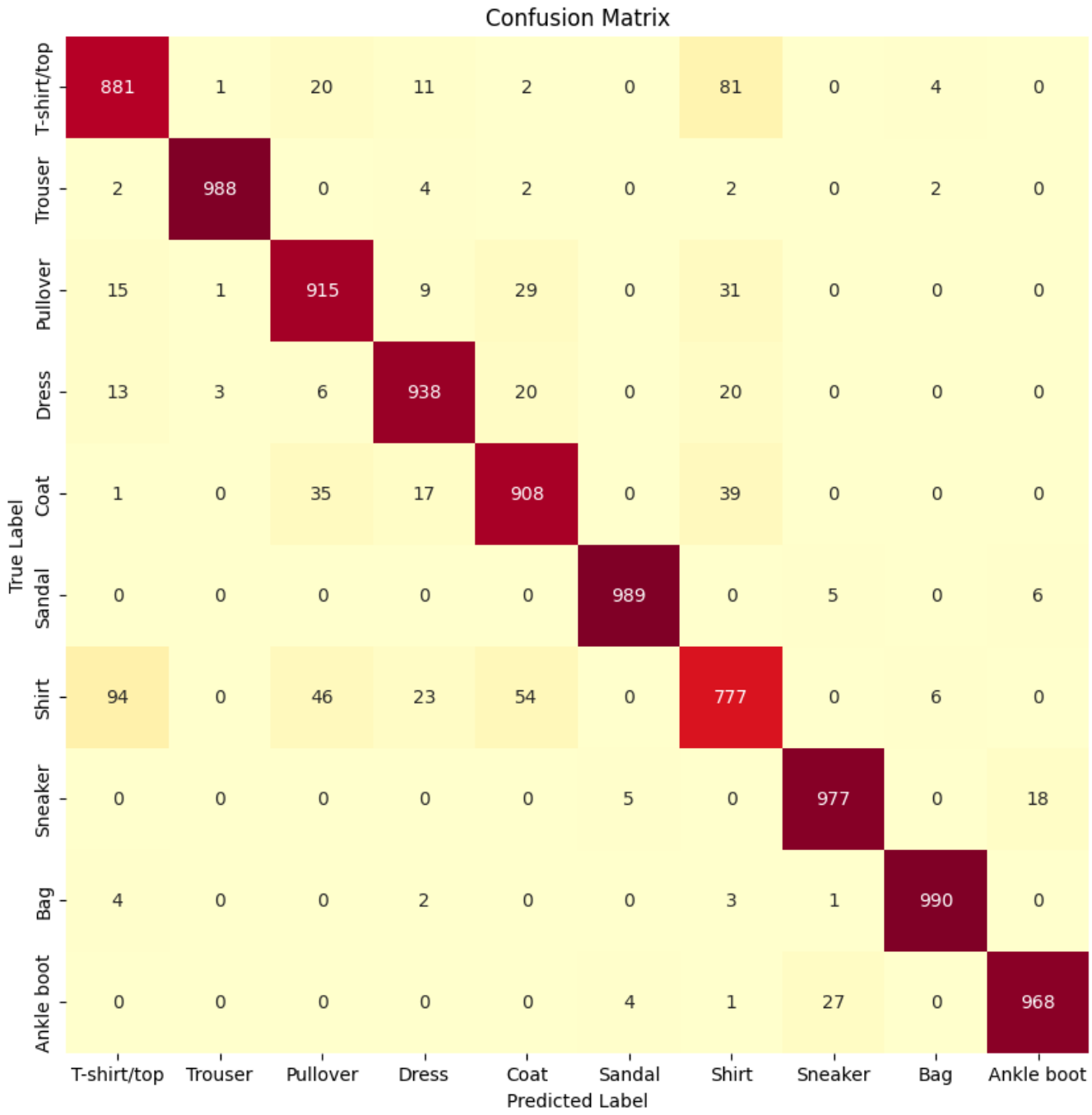
```
              precision    recall  f1-score   support

 T-shirt/top       0.87      0.88      0.88      1000
     Trouser       0.99      0.99      0.99      1000
    Pullover       0.90      0.92      0.91      1000
       Dress       0.93      0.94      0.94      1000
        Coat       0.89      0.91      0.90      1000
      Sandal       0.99      0.99      0.99      1000
       Shirt       0.81      0.78      0.80      1000
     Sneaker       0.97      0.98      0.97      1000
         Bag       0.99      0.99      0.99      1000
  Ankle boot       0.98      0.97      0.97      1000

    accuracy                           0.93     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       0.93      0.93      0.93     10000
```

We can see from the classification report that the model is very good at identifying and differentiating Trouser, Sandal and Bag but not as good as predicting T-shirt/top, pullovers and shirts. Let's do a mini error analysis and find out why.

In [ ]:
```
plt.figure(1, figsize=(10, 10))
plt.title("Confusion Matrix")
sns.heatmap(tf.math.confusion_matrix(
    np.argmax(y_test, axis=1),
    np.argmax(y_pred, axis=1),
    num_classes=10,
    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()
```



Observation

We can see that the model is able to predict all the labels mostly correctly. However, the issue is the shirt class which is quite similar to the T-shirt/top class and other clothing items that is worn on the top. We will do a little error analysis to find out more.

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(4, 5, 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(28,28,1), cmap='Greys')
    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 4 Column 3
      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 coat and pullover wrongly. This is likely due to both coat and pullover having long sleeves which we identified during our EDA could be a problem faced by the AI model.

Therefore, it appears that the model's mistakes are reasonable. In fact, real human performance was conducted by Zalando [Dataset creator] was only 83.5% accuracy. This shows that the model is performing quite well.

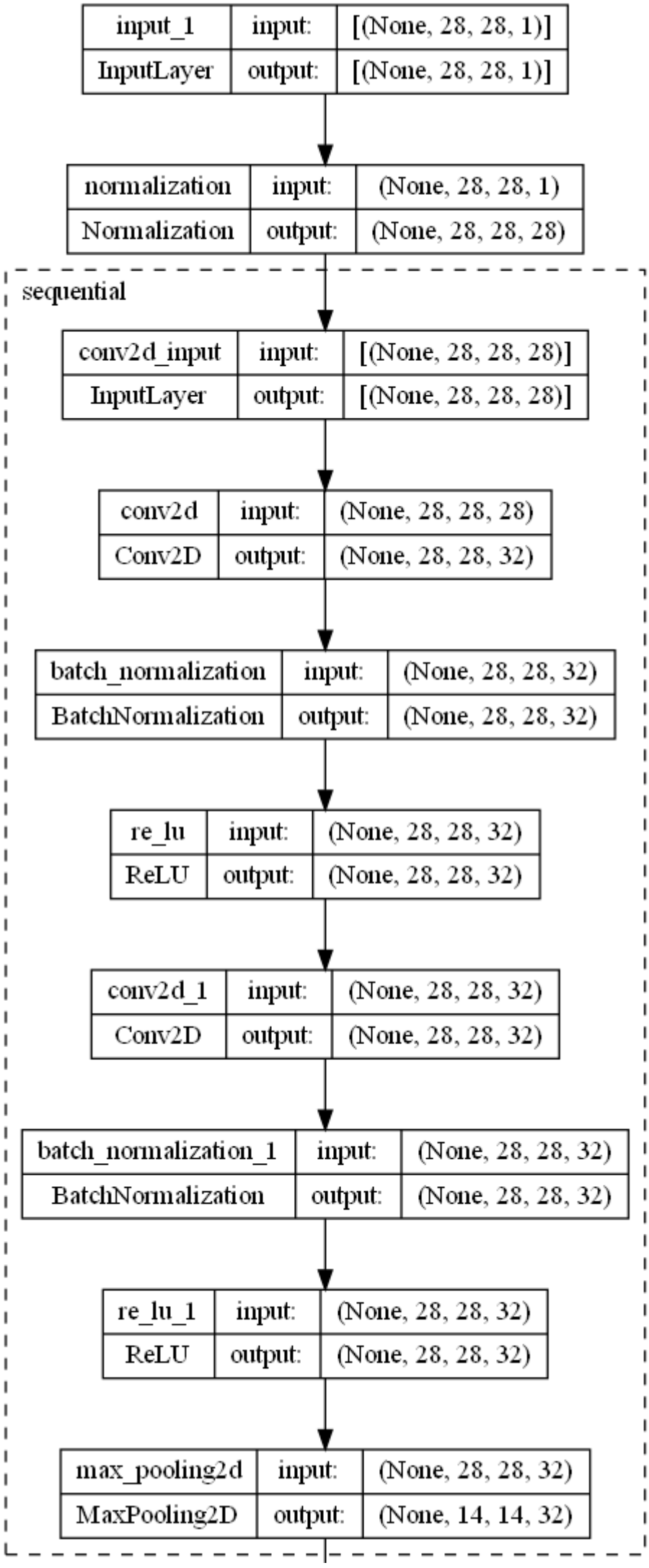
Comparing final_model VS other people's model

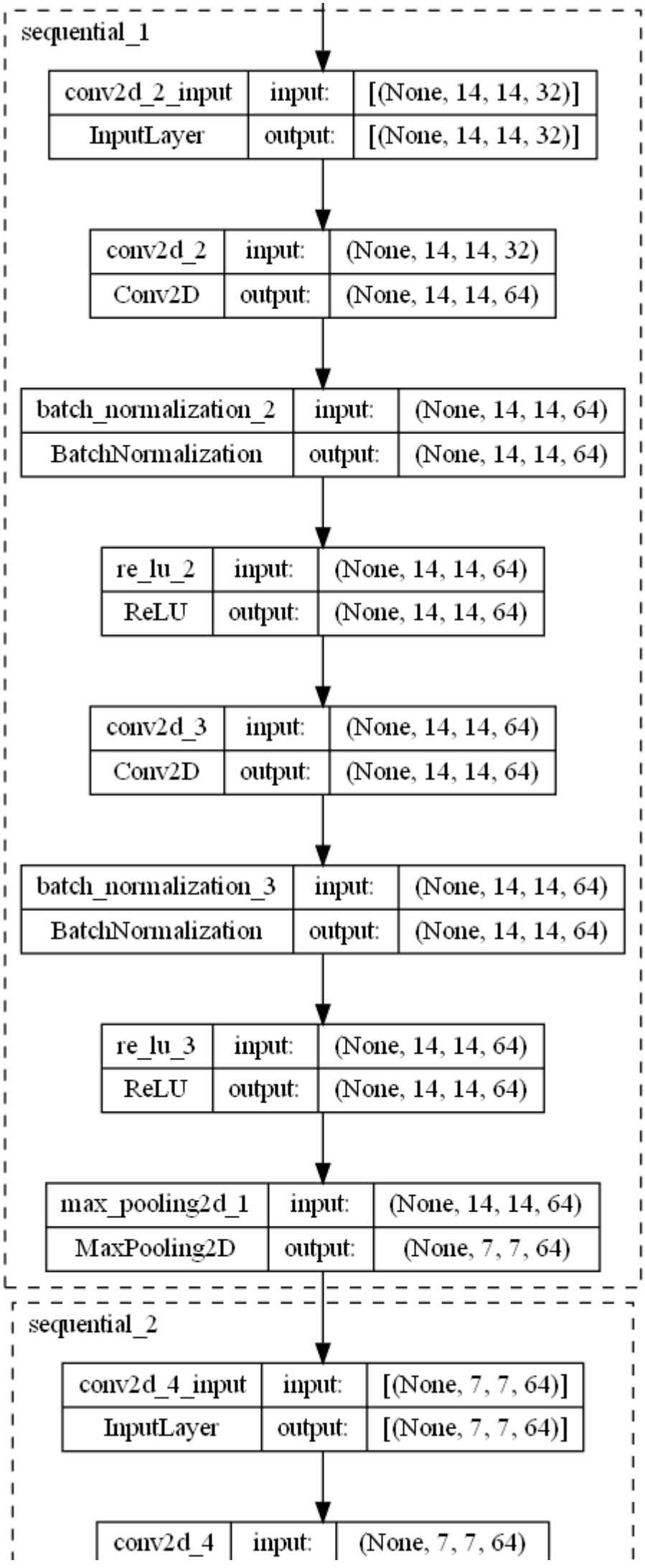
Based on PapersWithCode.com, our final model is ranked 11th out of the 13 models [Not inclusive of ours]. [https://paperswithcode.com/sota/image-classification-on-fashion-mnist]

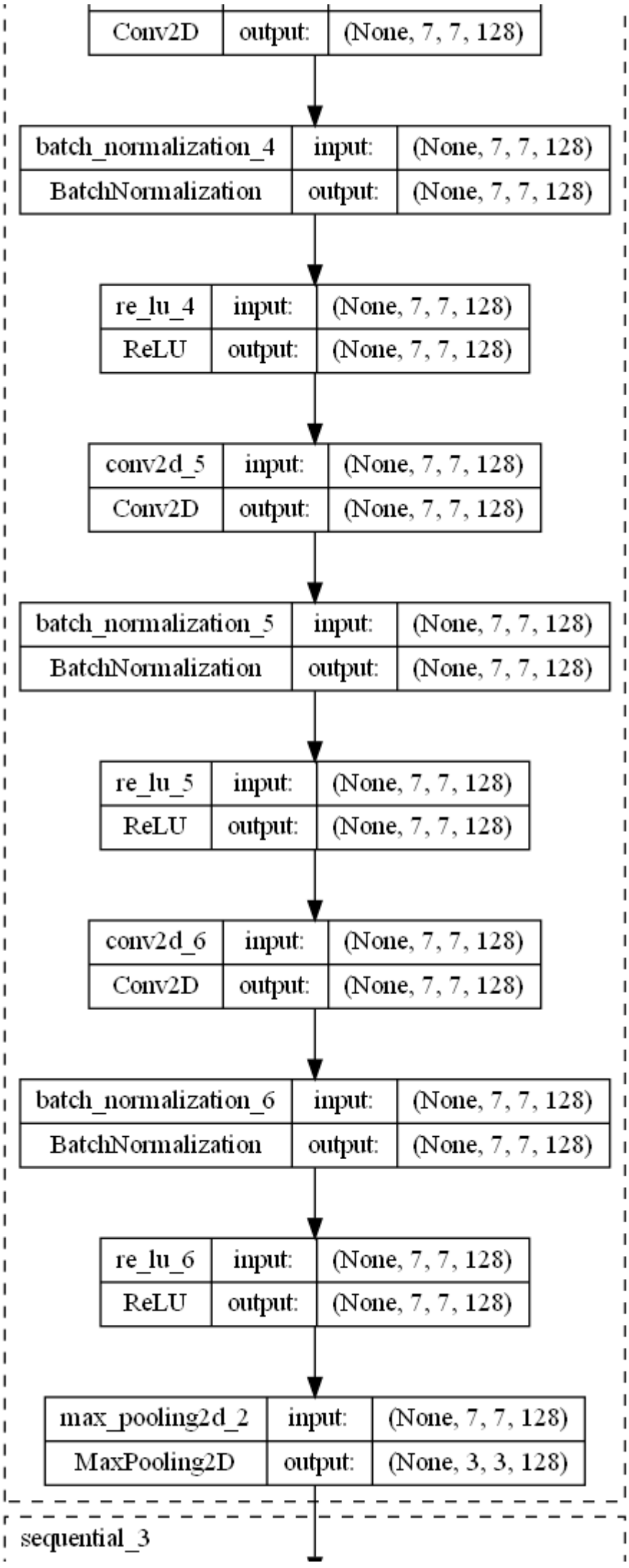
Model Visualisation

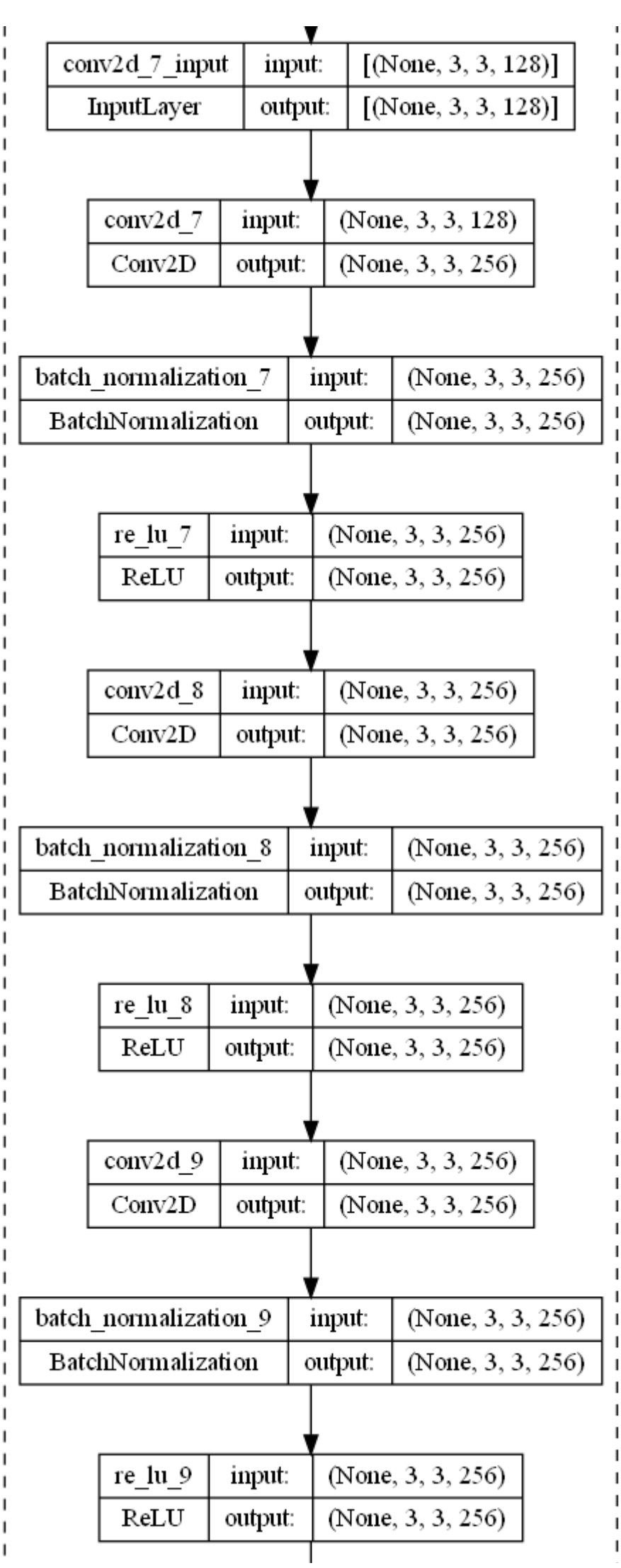
In [ ]:
```
tf.keras.utils.plot_model(final_model, show_shapes=True,
                          expand_nested=True, show_layer_activations=True)
```

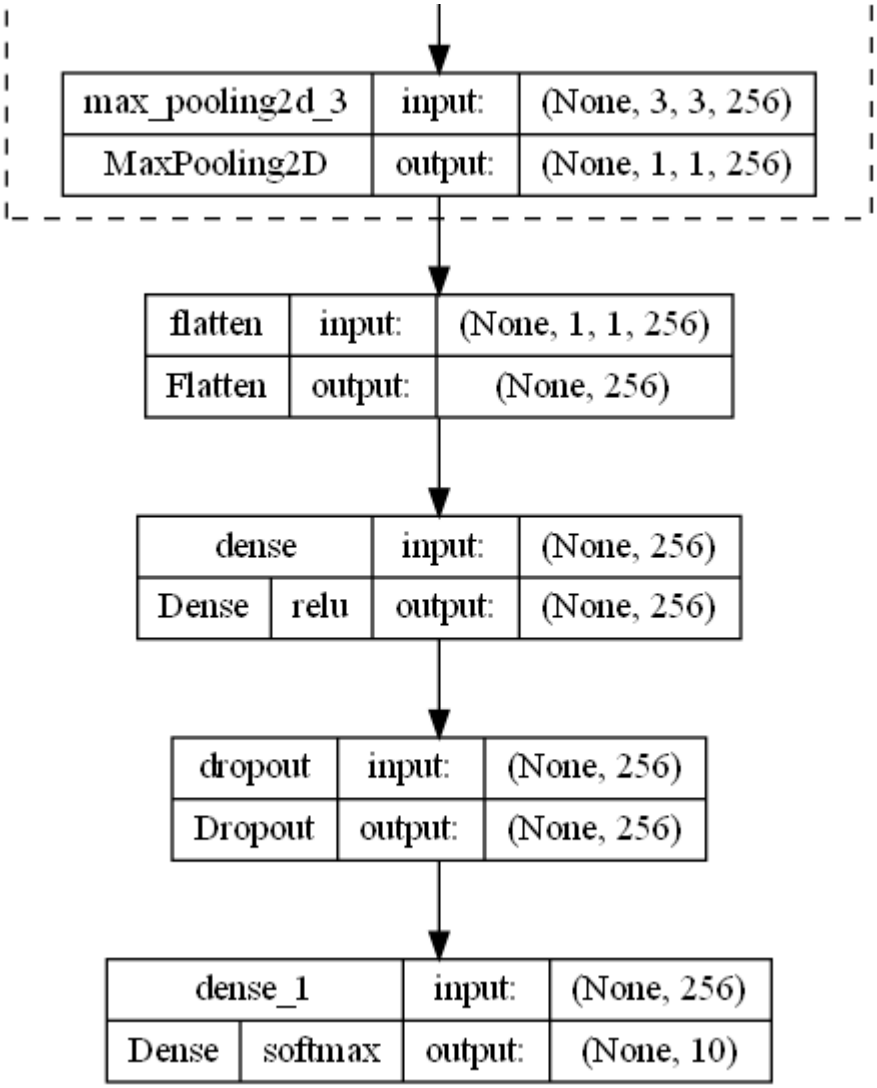
Out[ ]:


input_1 | input: | [(None, 28, 28, 1)]
InputLayer | output: | [(None, 28, 28, 1)]
normalization | input: | (None, 28, 28, 1)
Normalization | output: | (None, 28, 28, 28)
sequential
conv2d_input | input: | [(None, 28, 28, 28)]
InputLayer | output: | [(None, 28, 28, 28)]
conv2d | input: | (None, 28, 28, 28)
Conv2D | output: | (None, 28, 28, 32)
batch_normalization | input: | (None, 28, 28, 32)
BatchNormalization | output: | (None, 28, 28, 32)
re_lu | input: | (None, 28, 28, 32)
ReLU | output: | (None, 28, 28, 32)
conv2d_1 | input: | (None, 28, 28, 32)
Conv2D | output: | (None, 28, 28, 32)
batch_normalization_1 | input: | (None, 28, 28, 32)
BatchNormalization | output: | (None, 28, 28, 32)
re_lu_1 | input: | (None, 28, 28, 32)
ReLU | output: | (None, 28, 28, 32)
max_pooling2d | input: | (None, 28, 28, 32)
MaxPooling2D | output: | (None, 14, 14, 32)


sequential_1
conv2d_2_input | input: | [(None, 14, 14, 32)]
InputLayer | output: | [(None, 14, 14, 32)]
conv2d_2 | input: | (None, 14, 14, 32)
Conv2D | output: | (None, 14, 14, 64)
batch_normalization_2 | input: | (None, 14, 14, 64)
BatchNormalization | output: | (None, 14, 14, 64)
re_lu_2 | input: | (None, 14, 14, 64)
ReLU | output: | (None, 14, 14, 64)
conv2d_3 | input: | (None, 14, 14, 64)
Conv2D | output: | (None, 14, 14, 64)
batch_normalization_3 | input: | (None, 14, 14, 64)
BatchNormalization | output: | (None, 14, 14, 64)
re_lu_3 | input: | (None, 14, 14, 64)
ReLU | output: | (None, 14, 14, 64)
max_pooling2d_1 | input: | (None, 14, 14, 64)
MaxPooling2D | output: | (None, 7, 7, 64)
sequential_2
conv2d_4_input | input: | [(None, 7, 7, 64)]
InputLayer | output: | [(None, 7, 7, 64)]
conv2d_4 | input: | (None, 7, 7, 64)

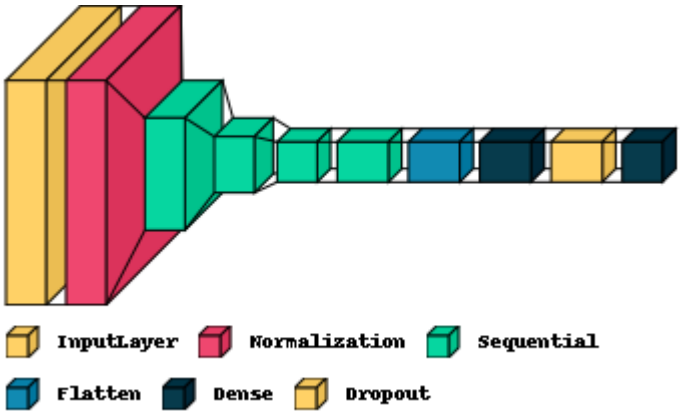

Conv2D output: (None, 7, 7, 128)
batch_normalization_4 input: (None, 7, 7, 128)
BatchNormalization output: (None, 7, 7, 128)
re_lu_4 input: (None, 7, 7, 128)
ReLU output: (None, 7, 7, 128)
conv2d_5 input: (None, 7, 7, 128)
Conv2D output: (None, 7, 7, 128)
batch_normalization_5 input: (None, 7, 7, 128)
BatchNormalization output: (None, 7, 7, 128)
re_lu_5 input: (None, 7, 7, 128)
ReLU output: (None, 7, 7, 128)
conv2d_6 input: (None, 7, 7, 128)
Conv2D output: (None, 7, 7, 128)
batch_normalization_6 input: (None, 7, 7, 128)
BatchNormalization output: (None, 7, 7, 128)
re_lu_6 input: (None, 7, 7, 128)
ReLU output: (None, 7, 7, 128)
max_pooling2d_2 input: (None, 7, 7, 128)
MaxPooling2D output: (None, 3, 3, 128)
sequential_3


conv2d_7_input | input: | [(None, 3, 3, 128)]
InputLayer | output: | [(None, 3, 3, 128)]
conv2d_7 | input: | (None, 3, 3, 128)
Conv2D | output: | (None, 3, 3, 256)
batch_normalization_7 | input: | (None, 3, 3, 256)
BatchNormalization | output: | (None, 3, 3, 256)
re_lu_7 | input: | (None, 3, 3, 256)
ReLU | output: | (None, 3, 3, 256)
conv2d_8 | input: | (None, 3, 3, 256)
Conv2D | output: | (None, 3, 3, 256)
batch_normalization_8 | input: | (None, 3, 3, 256)
BatchNormalization | output: | (None, 3, 3, 256)
re_lu_8 | input: | (None, 3, 3, 256)
ReLU | output: | (None, 3, 3, 256)
conv2d_9 | input: | (None, 3, 3, 256)
Conv2D | output: | (None, 3, 3, 256)
batch_normalization_9 | input: | (None, 3, 3, 256)
BatchNormalization | output: | (None, 3, 3, 256)
re_lu_9 | input: | (None, 3, 3, 256)
ReLU | output: | (None, 3, 3, 256)



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

Out[ ]:



Summary

In summary, I experimented with various models, and found that a custom built CNN with batch normalization and data augmentation to try to reduce overfitting. More room for improvement can be made to the model like increasing the number of layers, using different learning rates and optimizers like RMSProp and Ftrl can be done to make model better.