HW2

Problem 1:

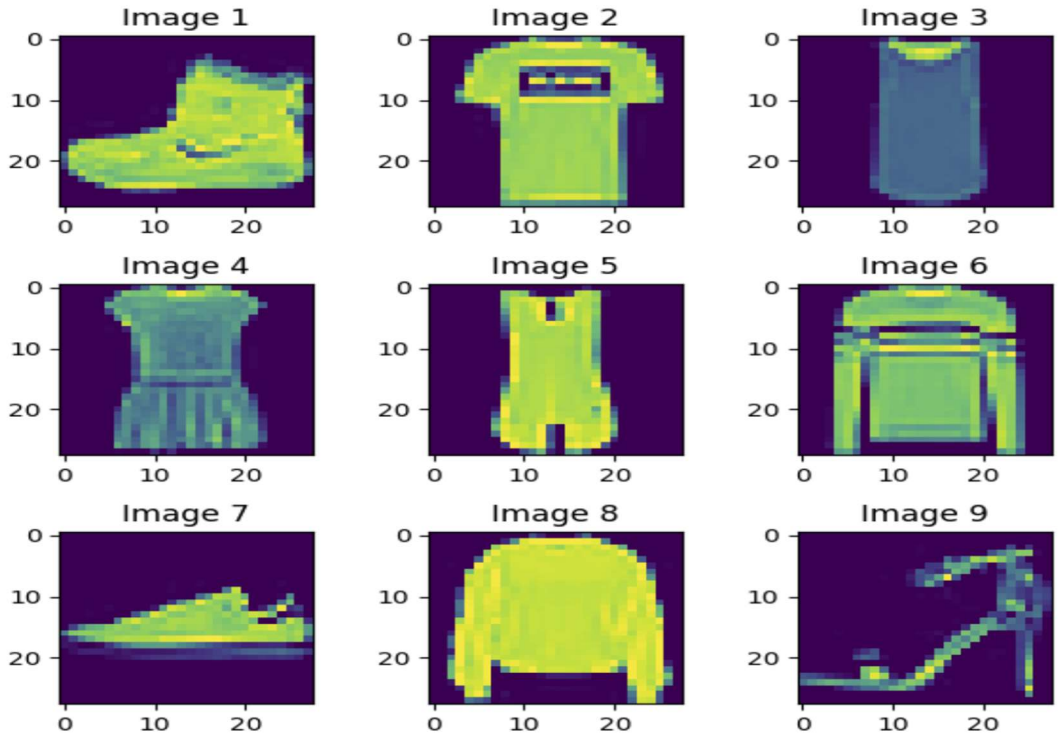
Load Dataset

```
# Create a 3x3 grid of subplots
fig, axes = plt.subplots(3, 3, figsize=(6, 6))

# Plot the first nine images in the grid
for i, ax in enumerate(axes.ravel()):
    ax.imshow(train_images[i])
    ax.set_title(f"Image {i + 1}")

# Adjust layout spacing
plt.tight_layout()

# Show the figure
plt.show()
```




```
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

a. CNN model from scratch:

```
# Creating CNN from scratch using Keras with all the layers as mentioned in the HW Questiondef CNN_model():

    model = keras.Sequential([

    # Convolutional Layer 1
    layers.Conv2D(32, (3, 3), strides=(1, 1), padding='same', activation='relu',input_shape=(28, 28, 1)),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Convolutional Layer 2
    layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Convolutional Layer 3
    layers.Conv2D(128, (3, 3), strides=(1, 1), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Convolutional Layer 4
    layers.Conv2D(256, (3, 3), strides=(1, 1), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Convolutional Layer 5
    layers.Conv2D(512, (3, 3), strides=(1, 1), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Flatten the output
    layers.Flatten(),

    # Fully Connected Layer 1
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),

    # Fully Connected Layer 2
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),

    # Output Layer
    layers.Dense(10, activation='softmax')
    ])

    return model

```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-509a22fad16b> in <cell line: 1>()
----> 1 model1 = CNN_model()

 3 frames
/usr/local/lib/python3.10/dist-packages/tensorflow/python/framework/ops.py in _create_c_op(graph, node_def, inputs, control_inputs, op_def, extract_traceback)
   1749   except errors.InvalidArgumentError as e:
   1750     # Convert to ValueError for backwards compatibility.
-> 1751     raise ValueError(e.message)
   1752
   1753   # Record the current Python stack trace as the creating stacktrace of this

ValueError: Exception encountered when calling layer "max_pooling2d_4" (type MaxPooling2D).

Negative dimension size caused by subtracting 2 from 1 for '{{node max_pooling2d_4/MaxPool}} = MaxPool[T=DT_FLOAT, data_format="NHWC", explicit_paddings=[], ksize=[1, 2, 2,
1], padding="VALID", strides=[1, 2, 2, 1]](Placeholder)' with input shapes: [?,1,1,512].

Call arguments received by layer "max_pooling2d_4" (type MaxPooling2D):
  • inputs=tf.Tensor(shape=(None, 1, 1, 512), dtype=float32)
```

Creating 5 layers of Convolution with batchnormalization and maxpooling(2,2) is giving errors. For the last layer, **Maxpoolong by(2,2) was making the output dimensions negative** as the layer four output shape for maxpooling was(None, 1,1,256). So, it couldnot further reduce to negative. **Changing last layer of MAxpooling to (1,1) can fixed this issue**. **But since fourth layer has already been reduced to MaxPooling(1,1) this last maxpooling can be removed without any effect as shown below**.

```
def cnn_model_without_maxpoollastLayer(activation, optimizer, learning_rate):

    optimizers = {
        'adam': keras.optimizers.Adam,
        'adagrad': keras.optimizers.Adagrad
    }
    model = keras.Sequential([
    # CNN 1
    layers.Conv2D(32, (3, 3), strides=(1, 1), padding='same', activation=activation, kernel_initializer='he_uniform', input_s
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # CNN 2
    layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same', activation=activation,kernel_initializer='he_uniform'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Cnn Layer 3
    layers.Conv2D(128, (3, 3), strides=(1, 1), padding='same', activation=activation,kernel_initializer='he_uniform'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Cnn 4
    layers.Conv2D(256, (3, 3), strides=(1, 1), padding='same', activation=activation,kernel_initializer='he_uniform'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    # Cnn 5
    layers.Conv2D(512, (3, 3), strides=(1, 1), padding='same', activation=activation,kernel_initializer='he_uniform'),
    layers.BatchNormalization(),
    #Layers.MaxPooling2D((1, 1)),

    layers.Flatten(),     # Flatten the output

    layers.Dense(256, activation=activation), # Fully Connected Layer 1
    layers.Dropout(0.5),

    layers.Dense(128, activation=activation), # Fully Connected Layer 2
    layers.Dropout(0.5),

    # Output Layer
    layers.Dense(10, activation='softmax')
    ])

    # Compile the model with adam and adagrad whichever is passed as an argument.
    model.compile(optimizer=optimizers[optimizer](learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    return model
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_45 (Conv2D)          (None, 28, 28, 32)        320

 batch_normalization_45 (Ba  (None, 28, 28, 32)        128
 tchNormalization)

 max_pooling2d_37 (MaxPooli  (None, 14, 14, 32)        0
 ng2D)

 conv2d_46 (Conv2D)          (None, 14, 14, 64)        18496

 batch_normalization_46 (Ba  (None, 14, 14, 64)        256
 tchNormalization)

 max_pooling2d_38 (MaxPooli  (None, 7, 7, 64)          0
 ng2D)

 conv2d_47 (Conv2D)          (None, 7, 7, 128)         73856

 batch_normalization_47 (Ba  (None, 7, 7, 128)         512
 tchNormalization)

 max_pooling2d_39 (MaxPooli  (None, 3, 3, 128)         0
 ng2D)

 conv2d_48 (Conv2D)          (None, 3, 3, 256)         295168

 batch_normalization_48 (Ba  (None, 3, 3, 256)         1024
 tchNormalization)

 max_pooling2d_40 (MaxPooli  (None, 1, 1, 256)         0
 ng2D)

 conv2d_49 (Conv2D)          (None, 1, 1, 512)         1180160

 batch_normalization_49 (Ba  (None, 1, 1, 512)         2048
 tchNormalization)

 flatten_9 (Flatten)         (None, 512)               0

 dense_27 (Dense)            (None, 256)               131328

 dropout_18 (Dropout)        (None, 256)               0

 dense_28 (Dense)            (None, 128)               32896

 dropout_19 (Dropout)        (None, 128)               0

 dense_29 (Dense)            (None, 10)                1290
```

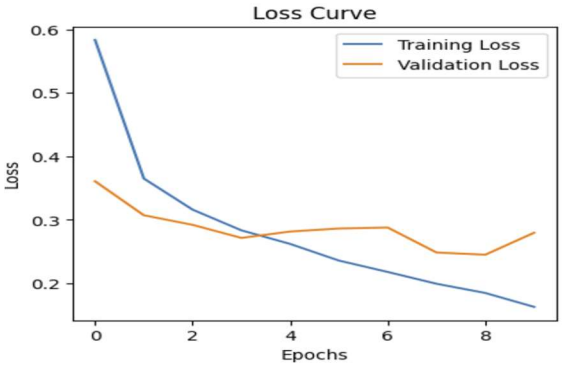
```
=================================================================
Total params: 1737482 (6.63 MB)
Trainable params: 1735498 (6.62 MB)
Non-trainable params: 1984 (7.75 KB)
_________________________________________________________________
```

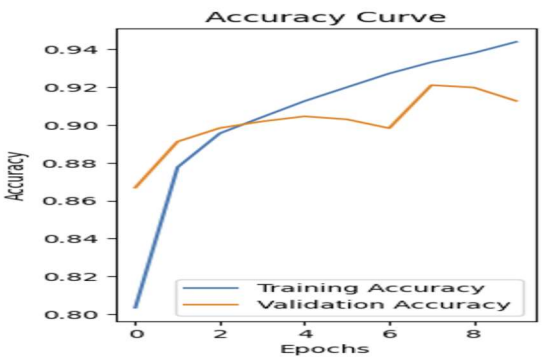
```python
# Train the model
history = model.fit(train_images[..., tf.newaxis], train_labels, epochs=10, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 16s 8ms/step - loss: 0.5833 - accuracy: 0.8034 - val_loss: 0.3608 - val_accuracy:
0.8669
Epoch 2/10
1500/1500 [==============================] - 11s 8ms/step - loss: 0.3651 - accuracy: 0.8777 - val_loss: 0.3073 - val_accuracy:
0.8913
Epoch 3/10
1500/1500 [==============================] - 11s 8ms/step - loss: 0.3161 - accuracy: 0.8956 - val_loss: 0.2922 - val_accuracy:
0.8984
Epoch 4/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.2835 - accuracy: 0.9042 - val_loss: 0.2716 - val_accuracy:
0.9019
Epoch 5/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.2622 - accuracy: 0.9126 - val_loss: 0.2817 - val_accuracy:
0.9046
Epoch 6/10
1500/1500 [==============================] - 11s 8ms/step - loss: 0.2359 - accuracy: 0.9200 - val_loss: 0.2864 - val_accuracy:
0.9030
Epoch 7/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.2180 - accuracy: 0.9272 - val_loss: 0.2879 - val_accuracy:
0.8983
Epoch 8/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.1995 - accuracy: 0.9332 - val_loss: 0.2487 - val_accuracy:
0.9211
Epoch 9/10
1500/1500 [==============================] - 16s 11ms/step - loss: 0.1850 - accuracy: 0.9381 - val_loss: 0.2452 - val_accuracy:
0.9198
Epoch 10/10
1500/1500 [==============================] - 11s 8ms/step - loss: 0.1631 - accuracy: 0.9440 - val_loss: 0.2798 - val_accuracy:
0.9128
```

```python
# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(test_images[..., tf.newaxis], test_labels)
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.3157 - accuracy: 0.9100
Test accuracy: 91.00%
```





CNN model from scratch gave an accuracy of 91%

b. Apply 5-Fold Cross Validation on the CNN

StratifiedKFold 5-Fold Cross Validation is used so that class balance is maintained across folds. A list of scores is created to store accuracy of each fold. Average accuracy and standard debiation is calculated based on the stored accuracies.

```
[ ]  # K-Fold Cross-Validation
     k_fold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

     # accuracy scores list
     scores = []

     # Performing 5-Fold Cross Validation
     for train_index, val_index in k_fold.split(train_images, train_labels):
         X_train, X_val = train_images[train_index], train_images[val_index]
         y_train, y_val = train_labels[train_index], train_labels[val_index]

         # Creating and compiling the model
         model = cnn_model_without_maxpoollastLayer('relu','adam',0.001)

         #print(model.summary())
         # Compile the model using sparse categorical cross entropy in which uses multiple class classification with one-hot encoding
         #model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
         # Train the model
         history = model.fit(X_train[..., tf.newaxis], y_train, epochs=10, validation_data=(X_val[..., tf.newaxis], y_val), verbose=0)

         # Evaluate the model on the validation data
         val_loss, val_accuracy = model.evaluate(X_val[..., tf.newaxis], y_val, verbose=0)
         print("val_accuracy", val_accuracy)
         scores.append(val_accuracy)
```

```
val_accuracy 0.9268333315849304
val_accuracy 0.9101666808128357
val_accuracy 0.9253333210945129
val_accuracy 0.921833336353302
val_accuracy 0.9184166789054871
```

```
]  # Calculate average accuracy and standard deviation
   average_accuracy = np.mean(scores)
   std_deviation = np.std(scores)

   # Print the results
   print(f'Average Accuracy: {average_accuracy * 100:.2f}%')
   print(f'Standard Deviation: {std_deviation * 100:.2f}%')
```

```
Average Accuracy: 92.05%
Standard Deviation: 0.59%
```

Stratified 5-Fold increased accuracy by 1% from 91% to 92.05%

c. Apply grid search on the CNN model

Hyperparameter Tuning

```
[ ]  # Different Hyperparameters

     activation_funcs = ['relu', 'LeakyReLU']
     optimizer = ['adam', 'adagrad']
     mini_batch_size = [16, 32, 64]
     learning_rate = [0.001, 0.0001, 0.00001]
```

```
[ ]  # Create an empty DataFrame to store results
     results_df = pd.DataFrame(columns=['Hyperparameters', 'Test Accuracy'])
     best_accuracy = 0
     best_hyperparameters = None
```

Loop over each hyperparameter list to fetch all possible combinations

```
[ ]  for act in activation_funcs:
         for opt in optimizer:
             for batchsize in mini_batch_size:
                 for l_rate in learning_rate:
                     # Build the CNN model with the current hyperparameters
                     model = cnn_model_without_maxpoollastLayer(act, opt, l_rate)

                     # Train the model
                     history = model.fit(train_images[..., tf.newaxis], train_labels, epochs=10, batch_size=batchsize, verbose=0)

                     # Evaluate the model on the test data
                     test_loss, test_accuracy = model.evaluate(test_images[..., tf.newaxis], test_labels, verbose=0)

                     # Store the hyperparameters and accuracy in the DataFrame
                     results_df = results_df.append({'Hyperparameters': (act, opt, batchsize, l_rate), 'Test Accuracy': test_accuracy}, ignore_index=True)
                     #print(results_df)

                     # Check if this combination resulted in the best accuracy so far
                     if test_accuracy > best_accuracy:
                         best_accuracy = test_accuracy
                         best_hyperparameters = (act, opt, batchsize, l_rate)
                         #print(best_hyperparameters,best_accuracy)
```

```
# Display the DataFrame with all results
print("\nDataFrame with Results of all 36 combinations of hyperparameters:")
results_df.sort_values(by=['Test Accuracy'], ascending=False)
```

```
DataFrame with Results of all 36 combinations of hyperparameters:
```

	Hyperparameters	Test Accuracy
24	(LeakyReLU, adam, 64, 0.001)	0.9206
0	(relu, adam, 16, 0.001)	0.9184
18	(LeakyReLU, adam, 16, 0.001)	0.9173
21	(LeakyReLU, adam, 32, 0.001)	0.9172
1	(relu, adam, 16, 0.0001)	0.9113
3	(relu, adam, 32, 0.001)	0.9108
6	(relu, adam, 64, 0.001)	0.9087
19	(LeakyReLU, adam, 16, 0.0001)	0.9075
4	(relu, adam, 32, 0.0001)	0.9032
7	(relu, adam, 64, 0.0001)	0.9029
22	(LeakyReLU, adam, 32, 0.0001)	0.9022
25	(LeakyReLU, adam, 64, 0.0001)	0.8990
20	(LeakyReLU, adam, 16, 1e-05)	0.8896

30	(LeakyReLU, adagrad, 32, 0.001)	0.8832
27	(LeakyReLU, adagrad, 16, 0.001)	0.8821
2	(relu, adam, 16, 1e-05)	0.8809
33	(LeakyReLU, adagrad, 64, 0.001)	0.8795
23	(LeakyReLU, adam, 32, 1e-05)	0.8760
9	(relu, adagrad, 16, 0.001)	0.8731
12	(relu, adagrad, 32, 0.001)	0.8726
26	(LeakyReLU, adam, 64, 1e-05)	0.8713
15	(relu, adagrad, 64, 0.001)	0.8686
5	(relu, adam, 32, 1e-05)	0.8655
8	(relu, adam, 64, 1e-05)	0.8576
28	(LeakyReLU, adagrad, 16, 0.0001)	0.8066
31	(LeakyReLU, adagrad, 32, 0.0001)	0.8038
34	(LeakyReLU, adagrad, 64, 0.0001)	0.8003
13	(relu, adagrad, 32, 0.0001)	0.7649
16	(relu, adagrad, 64, 0.0001)	0.7599
10	(relu, adagrad, 16, 0.0001)	0.7589
29	(LeakyReLU, adagrad, 16, 1e-05)	0.6139
32	(LeakyReLU, adagrad, 32, 1e-05)	0.6079
35	(LeakyReLU, adagrad, 64, 1e-05)	0.5835
14	(relu, adagrad, 32, 1e-05)	0.3845
17	(relu, adagrad, 64, 1e-05)	0.3713

```
[ ]  # Print the results
     print("Best Hyperparameters:")
     print("\nActivation function, Optimizer, Mini-batch size, Learning rate:=> ",best_hyperparameters)
     print("\nTest Accuracy:", best_accuracy)
```

```
Best Hyperparameters:

Activation function, Optimizer, Mini-batch size, Learning rate:=>  ('LeakyReLU', 'adam', 64, 0.001)

Test Accuracy: 0.9205999970436096
```

The **worst performance is given by learning rate of 0.00001** as the learning is so slow that within 10 epochs it is no where near minima. Same for 0.0001 learning rate, the learning rate is less that in 10 epochs it is still not reaching minima. **0.001 seems to be better**. **LeakyReLU that helps with the problem of vanishing gradient is showing better results than ReLU**. **Adam is performing better than Adagrad** here. In general also **Adam performs better** as it **combines** both techniques of **maintaining exponential moving averages of gradients** of parameters to update parameters and **updating the learning rate** of each parameter based on the **previous gradient information**. **Adagrad just uses the updation of learning rate based on the past gradient information**.

Hyperparameter tuning gave an accuracy of 92.06%

d. Data Augmentation

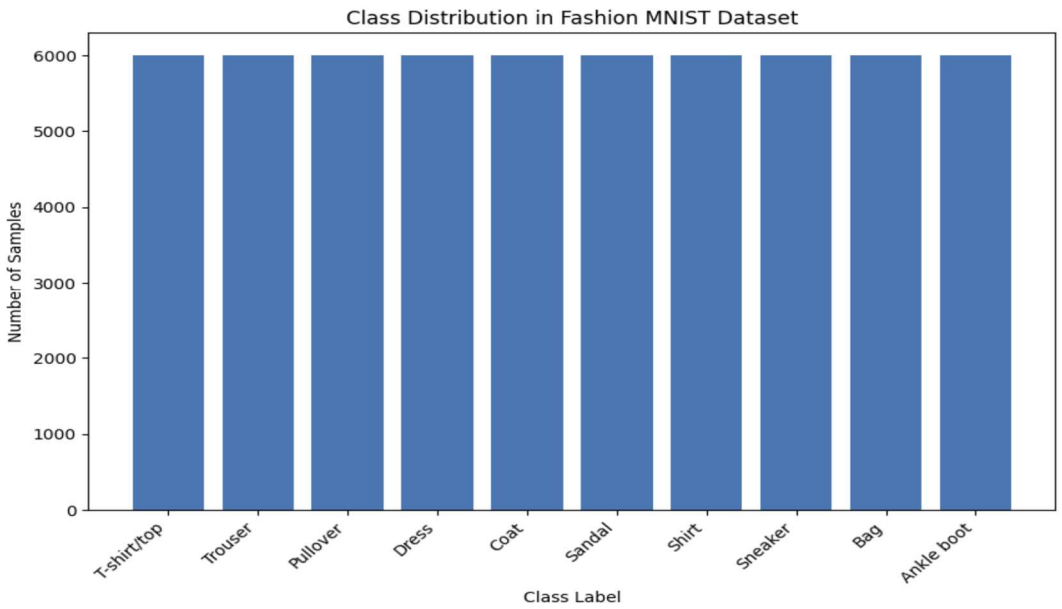
check dataset class distribution

```
]:
# Calculate the number of samples in each class
class_counts = np.bincount(train_labels)

# Define class labels (0 to 9)
class_labels = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

# Plot the class distribution
plt.figure(figsize=(10, 6))
plt.bar(class_labels, class_counts)
plt.xlabel("Class Label")
plt.ylabel("Number of Samples")
plt.title("Class Distribution in Fashion MNIST Dataset")
plt.xticks(rotation=45, ha="right")

plt.show()
```



Classes are equally distributed. But for the sake of learning we will emply Data Augmentation

- Data Augmentation using rotation, width and height shifting, shear, flip,zoom

Using **ImageDataGenerator** with 6 Augmentation techniques - **Rotation, width shift, Height Shift, transparency, Zoom, Horizontal flip**

```
[ ]  # Creating an instance of the ImageDataGenerator with augmentation settings
     datagen = tf.keras.preprocessing.image.ImageDataGenerator(
         rotation_range=25,      # Randomly rotate images by up to 25 degrees
         width_shift_range=0.15,  # Shift images horizontally by up to 15% of the width
         height_shift_range=0.15, # Shift images vertically by up to 15% of the height
         shear_range=0.10,        # Shear intensity (shear angle in radians)
         zoom_range=0.25,         # Zoom into images by up to 25%
         horizontal_flip=True,   # Randomly flip images horizontally
         fill_mode='nearest'     # Fill in newly created pixels after rotation or shifting
     )

     augmented_train_images = [] # augmented images list

     for i in range(len(train_images)): # Appling data augmentation to each of the training image
         image = train_images[i]
         image = np.expand_dims(image, axis=-1)  # Adding channel dimension (1 channel for grayscale)
         augmented_image = datagen.random_transform(image)
         augmented_train_images.append(augmented_image)

     # Converting the list of augmented images back to a numpy array
     augmented_train_images = np.array(augmented_train_images)

     model = cnn_model_without_maxpoollastLayer('relu', 'adam', 0.001)
```

```
# Training the CNN model on the augmented data
history_augmented = model.fit(augmented_train_images,train_labels, epochs=10, validation_split=0.2, batch_size=32)

# Evaluating the model on the test data
test_loss_augmented, test_accuracy_augmented = model.evaluate(test_images[..., tf.newaxis],test_labels, verbose=0)
```

```
Epoch 1/10
1500/1500 [==============================] - 17s 8ms/step - loss: 1.0142 - accuracy: 0.6324 - val_loss: 0.6693 - val_accuracy: 0.7422
Epoch 2/10
1500/1500 [==============================] - 13s 8ms/step - loss: 0.7186 - accuracy: 0.7341 - val_loss: 0.6329 - val_accuracy: 0.7573
Epoch 3/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.6354 - accuracy: 0.7683 - val_loss: 0.5720 - val_accuracy: 0.7846
Epoch 4/10
1500/1500 [==============================] - 11s 8ms/step - loss: 0.5733 - accuracy: 0.7972 - val_loss: 0.5384 - val_accuracy: 0.8110
Epoch 5/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.5269 - accuracy: 0.8133 - val_loss: 0.5746 - val_accuracy: 0.7906
Epoch 6/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.4819 - accuracy: 0.8306 - val_loss: 0.5516 - val_accuracy: 0.7922
Epoch 7/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.4474 - accuracy: 0.8424 - val_loss: 0.5180 - val_accuracy: 0.8067
Epoch 8/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.4051 - accuracy: 0.8555 - val_loss: 0.4883 - val_accuracy: 0.8292
Epoch 9/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.3853 - accuracy: 0.8662 - val_loss: 0.5156 - val_accuracy: 0.8253
Epoch 10/10
1500/1500 [==============================] - 12s 8ms/step - loss: 0.3558 - accuracy: 0.8751 - val_loss: 0.5218 - val_accuracy: 0.8292
```

- call using LeakyReLU with batch size 64 keeping other parameters same didnot give good results (81.5%) on the augmented dataset. Instead using relu with batch_size 32.

```
# the testing accuracy after data augmentation
print("Test Accuracy with Data Augmentation:", test_accuracy_augmented)
```

```
Test Accuracy with Data Augmentation: 0.8378999829292297
```

Data Augmentation gave an accuracy of 83.78%

e. Transfer Learning

Load the VGG-19 model. Drop after the block4 conv1 layer (highlighted in the image below) and on top of it add one global average pooling layer, one fully connected layer, and one final output layer. Keep the base model layers (VGG19) freeze.

224x224 was causing memory issue on my system so using 48x48 as target size. The shape of the images are changed from 28x28x3 to 48x48x3.This snippet is using the same processing as used by vggnet19 for their preprocessing.

```
# Define the target size for resizing
target_size = (48, 48)

# Initialize lists to store the resized and expanded images
resized_train_images = []
resized_test_images = []
```

```
# Resizing and expanding the training images
for img in train_images:
    resized_img = cv2.resize(img, target_size)
    resized_img = np.repeat(resized_img[..., np.newaxis], 3, -1) # adding rgb channel
    resized_train_images.append(resized_img)
```

```
# Resizing and expanding the test images
for img in test_images:
    resized_img = cv2.resize(img, target_size)
    resized_img = np.repeat(resized_img[..., np.newaxis], 3, -1)
    resized_test_images.append(resized_img)
```

```
# Converting back the lists to NumPy arrays
resized_train_images = np.array(resized_train_images)
resized_test_images = np.array(resized_test_images)
```

```
resized_test_images.shape,resized_train_images.shape
```

```
((10000, 48, 48, 3), (60000, 48, 48, 3))
```

To make predictions on new images, I need to preprocess the input images in the same way that the original training images were preprocessed.

```
[ ]  # Normalize pixel values to be between 0 and 1
     resized_train_images, resized_test_images = resized_train_images / 255.0, resized_test_images / 255.0
```

```
# Loading the VGG-19 model with pre-trained weights (include_top=False this will not include the top layer)
base_model = VGG19(weights='imagenet', include_top=False, input_shape=(48, 48, 3))
```

```
for layer in base_model.layers:
    layer.trainable = False # Freezing the base model layers
```

```
layer = base_model.get_layer('block4_conv1').output # Dropping after the block4 conv1 layer
```

```
# Adding Global Average Pooling layer
layer = GlobalAveragePooling2D()(layer)
```

```
# Adding one fully connected layer
layer = Dense(512, activation='relu')(layer)
```

```
# Adding one final output layer with the 10 classes
output = Dense(10, activation='softmax')(layer)
```

```
# the changed model after adding last few layers
vgg_model = Model(inputs=base_model.input, outputs=output)
```

```
# I didn't one-hot encode as is done in the base vggnet19 model, instead using sparse_categorical_crossentropy which does it internally
vgg_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
vgg_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 48, 48, 3)]       0

 block1_conv1 (Conv2D)       (None, 48, 48, 64)        1792

 block1_conv2 (Conv2D)       (None, 48, 48, 64)        36928

 block1_pool (MaxPooling2D)  (None, 24, 24, 64)        0

 block2_conv1 (Conv2D)       (None, 24, 24, 128)       73856

 block2_conv2 (Conv2D)       (None, 24, 24, 128)       147584

 block2_pool (MaxPooling2D)  (None, 12, 12, 128)       0

 block3_conv1 (Conv2D)       (None, 12, 12, 256)       295168

 block3_conv2 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv3 (Conv2D)       (None, 12, 12, 256)       590080

 block3_conv4 (Conv2D)       (None, 12, 12, 256)       590080

 block3_pool (MaxPooling2D)  (None, 6, 6, 256)         0

 block4_conv1 (Conv2D)       (None, 6, 6, 512)         1180160

 global_average_pooling2d (  (None, 512)               0
 GlobalAveragePooling2D)

 dense (Dense)               (None, 512)               262656

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 3773514 (14.39 MB)
Trainable params: 267786 (1.02 MB)
Non-trainable params: 3505728 (13.37 MB)
_________________________________________________________________
```

Training using the resized fashion train images

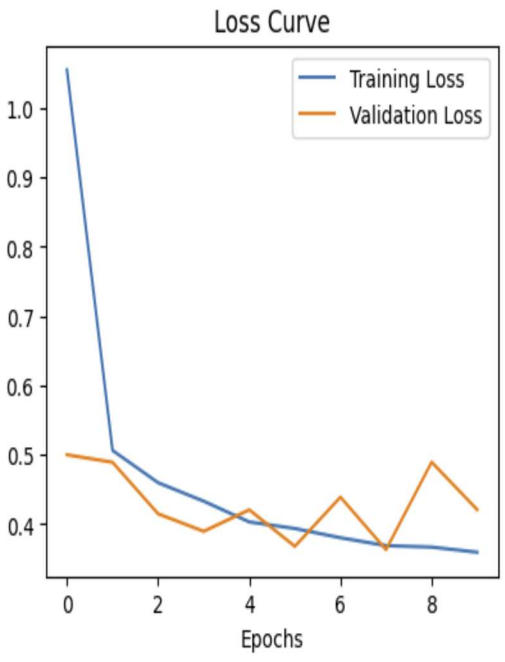
```
[ ] history = vgg_model.fit(resized_train_images, train_labels, epochs=10, validation_split=0.2,batch_size=32)
```

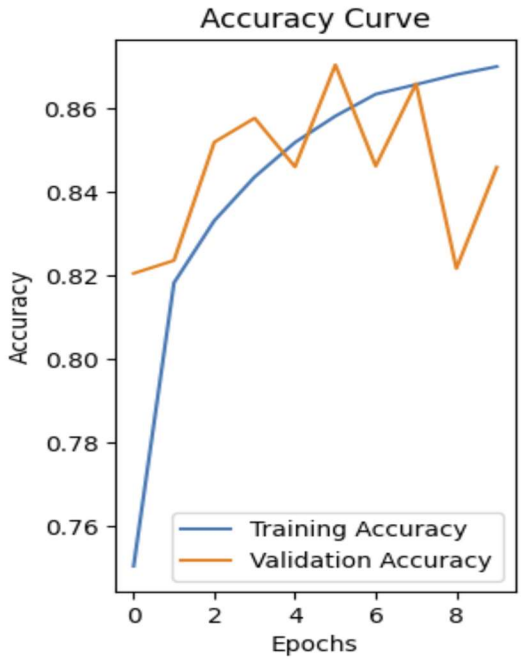
```
Epoch 1/10
1500/1500 [==============================] - 22s 13ms/step - loss: 1.0553 - accuracy: 0.7504 - val_loss: 0.5002 - val_accuracy: 0.8205
Epoch 2/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.5065 - accuracy: 0.8182 - val_loss: 0.4894 - val_accuracy: 0.8236
Epoch 3/10
1500/1500 [==============================] - 18s 12ms/step - loss: 0.4599 - accuracy: 0.8331 - val_loss: 0.4150 - val_accuracy: 0.8519
Epoch 4/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.4332 - accuracy: 0.8436 - val_loss: 0.3898 - val_accuracy: 0.8577
Epoch 5/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.4033 - accuracy: 0.8519 - val_loss: 0.4208 - val_accuracy: 0.8460
Epoch 6/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.3938 - accuracy: 0.8581 - val_loss: 0.3681 - val_accuracy: 0.8704
Epoch 7/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.3805 - accuracy: 0.8634 - val_loss: 0.4391 - val_accuracy: 0.8462
Epoch 8/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.3691 - accuracy: 0.8657 - val_loss: 0.3633 - val_accuracy: 0.8659
Epoch 9/10
1500/1500 [==============================] - 18s 12ms/step - loss: 0.3670 - accuracy: 0.8681 - val_loss: 0.4895 - val_accuracy: 0.8217
Epoch 10/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.3596 - accuracy: 0.8700 - val_loss: 0.4211 - val_accuracy: 0.8459
```

```
test_loss_vgg, test_accuracy_vgg = vgg_model.evaluate(resized_test_images, test_labels, verbose=0)
```

```
print("Test Accuracy of modified pre-trained VGG using Transfer learning is:", test_accuracy_vgg)
```

```
Test Accuracy of modified pre-trained VGG using Transfer learning is: 0.8447999954223633
```





VGGNET19 gave an accuracy of 84.47%

Problem 2:

Developing ResNet model from scratch

```
def residual_block(x, filters, kernel_size=3, stride=1, projection=True):
    # Shortcut
    shortcut = x

    # First convolution
    x = Conv2D(filters, kernel_size=kernel_size, strides=stride, padding='same')(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    # Second convolution
    x = Conv2D(filters, kernel_size=kernel_size, strides=1, padding='same')(x)
    x = BatchNormalization()(x)

    # Check if we need to project the shortcut (change dimensions)
    if projection:
        shortcut = Conv2D(filters, kernel_size=1, strides=stride, padding='same')(shortcut)
        shortcut = BatchNormalization()(shortcut)

    # Add the shortcut to the output
    x = Add()([x, shortcut])
    x = ReLU()(x)

    return x
```

```python
# Build the ResNet-like model
def build_resnet(input_shape, num_classes):

    inputs = Input(shape=input_shape)

    # Initial convolution layer
    x = Conv2D(32, kernel_size=3, strides=1, padding='same')(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    # Residual Blocks
    x = residual_block(x, filters=32)
    x = residual_block(x, filters=32)
    x = residual_block(x, filters=32)

    x = residual_block(x, filters=64, stride=1)
    x = residual_block(x, filters=64)
    x = residual_block(x, filters=64)

    x = residual_block(x, filters=128, stride=1)
    x = residual_block(x, filters=128)
    x = residual_block(x, filters=128)

    x = GlobalAveragePooling2D()(x)

    # Add Flatten layer to convert to a 1D vector
    x = Flatten()(x)

    # Fully connected layer for classification
    x = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs, x)
    return model
```

```python
# Create the ResNet model
resnet_model = build_resnet((28, 28, 1), 10)
```

```
# Print the model summary
resnet_model.summary()
```

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

 conv2d (Conv2D)             (None, 28, 28, 32)        320       ['input_1[0][0]']

 batch_normalization (Batch  (None, 28, 28, 32)        128       ['conv2d[0][0]']
 Normalization)

 re_lu (ReLU)                (None, 28, 28, 32)        0         ['batch_normalization[0][0]']

 conv2d_1 (Conv2D)           (None, 28, 28, 32)        9248      ['re_lu[0][0]']

 batch_normalization_1 (Bat  (None, 28, 28, 32)        128       ['conv2d_1[0][0]']
 chNormalization)

 re_lu_1 (ReLU)              (None, 28, 28, 32)        0         ['batch_normalization_1[0][0]'
                                                                 ]

 conv2d_2 (Conv2D)           (None, 28, 28, 32)        9248      ['re_lu_1[0][0]']

 conv2d_3 (Conv2D)           (None, 28, 28, 32)        1056      ['re_lu[0][0]']

 batch_normalization_2 (Bat  (None, 28, 28, 32)        128       ['conv2d_2[0][0]']
 chNormalization)

 batch_normalization_3 (Bat  (None, 28, 28, 32)        128       ['conv2d_3[0][0]']
 chNormalization)

 add (Add)                   (None, 28, 28, 32)        0         ['batch_normalization_2[0][0]'
                                                                 , 'batch_normalization_3[0][0]
                                                                 ']

 re_lu_2 (ReLU)              (None, 28, 28, 32)        0         ['add[0][0]']

 conv2d_4 (Conv2D)           (None, 28, 28, 32)        9248      ['re_lu_2[0][0]']

 batch_normalization_4 (Bat  (None, 28, 28, 32)        128       ['conv2d_4[0][0]']
 chNormalization)

 re_lu_3 (ReLU)              (None, 28, 28, 32)        0         ['batch_normalization_4[0][0]'
                                                                 ]

 conv2d_5 (Conv2D)           (None, 28, 28, 32)        9248      ['re_lu_3[0][0]']

 conv2d_6 (Conv2D)           (None, 28, 28, 32)        1056      ['re_lu_2[0][0]']

 batch_normalization_5 (Bat  (None, 28, 28, 32)        128       ['conv2d_5[0][0]']
 chNormalization)

 batch_normalization_6 (Bat  (None, 28, 28, 32)        128       ['conv2d_6[0][0]']
 chNormalization)

 add_1 (Add)                 (None, 28, 28, 32)        0         ['batch_normalization_5[0][0]'
                                                                 , 'batch_normalization_6[0][0]
                                                                 ']

 re_lu_4 (ReLU)              (None, 28, 28, 32)        0         ['add_1[0][0]']

 conv2d_7 (Conv2D)           (None, 28, 28, 32)        9248      ['re_lu_4[0][0]']
```

re_lu_4 (ReLU)	(None, 28, 28, 32)	0	['add_1[0][0]']
conv2d_7 (Conv2D)	(None, 28, 28, 32)	9248	['re_lu_4[0][0]']
batch_normalization_7 (BatchNormalization)	(None, 28, 28, 32)	128	['conv2d_7[0][0]']
re_lu_5 (ReLU)	(None, 28, 28, 32)	0	['batch_normalization_7[0][0]']
conv2d_8 (Conv2D)	(None, 28, 28, 32)	9248	['re_lu_5[0][0]']
conv2d_9 (Conv2D)	(None, 28, 28, 32)	1056	['re_lu_4[0][0]']
batch_normalization_8 (BatchNormalization)	(None, 28, 28, 32)	128	['conv2d_8[0][0]']
batch_normalization_9 (BatchNormalization)	(None, 28, 28, 32)	128	['conv2d_9[0][0]']
add_2 (Add)	(None, 28, 28, 32)	0	['batch_normalization_8[0][0]', 'batch_normalization_9[0][0]']
re_lu_6 (ReLU)	(None, 28, 28, 32)	0	['add_2[0][0]']
conv2d_10 (Conv2D)	(None, 28, 28, 64)	18496	['re_lu_6[0][0]']
batch_normalization_10 (BatchNormalization)	(None, 28, 28, 64)	256	['conv2d_10[0][0]']
re_lu_7 (ReLU)	(None, 28, 28, 64)	0	['batch_normalization_10[0][0]']
conv2d_11 (Conv2D)	(None, 28, 28, 64)	36928	['re_lu_7[0][0]']
conv2d_12 (Conv2D)	(None, 28, 28, 64)	2112	['re_lu_6[0][0]']
batch_normalization_11 (BatchNormalization)	(None, 28, 28, 64)	256	['conv2d_11[0][0]']
batch_normalization_12 (BatchNormalization)	(None, 28, 28, 64)	256	['conv2d_12[0][0]']
add_3 (Add)	(None, 28, 28, 64)	0	['batch_normalization_11[0][0]', 'batch_normalization_12[0][0]']
re_lu_8 (ReLU)	(None, 28, 28, 64)	0	['add_3[0][0]']
conv2d_13 (Conv2D)	(None, 28, 28, 64)	36928	['re_lu_8[0][0]']
batch_normalization_13 (BatchNormalization)	(None, 28, 28, 64)	256	['conv2d_13[0][0]']
re_lu_9 (ReLU)	(None, 28, 28, 64)	0	['batch_normalization_13[0][0]']
conv2d_14 (Conv2D)	(None, 28, 28, 64)	36928	['re_lu_9[0][0]']
conv2d_15 (Conv2D)	(None, 28, 28, 64)	4160	['re_lu_8[0][0]']

```
batch_normalization_15 (Ba  (None, 28, 28, 64)          256       ['conv2d_15[0][0]']
tchNormalization)

add_4 (Add)                 (None, 28, 28, 64)          0         ['batch_normalization_14[0][0]
                                                                  ',
                                                                   'batch_normalization_15[0][0]
                                                                  ']

re_lu_10 (ReLU)             (None, 28, 28, 64)          0         ['add_4[0][0]']

conv2d_16 (Conv2D)          (None, 28, 28, 64)          36928     ['re_lu_10[0][0]']

batch_normalization_16 (Ba  (None, 28, 28, 64)          256       ['conv2d_16[0][0]']
tchNormalization)

re_lu_11 (ReLU)             (None, 28, 28, 64)          0         ['batch_normalization_16[0][0]
                                                                  ']

conv2d_17 (Conv2D)          (None, 28, 28, 64)          36928     ['re_lu_11[0][0]']

conv2d_18 (Conv2D)          (None, 28, 28, 64)          4160      ['re_lu_10[0][0]']

batch_normalization_17 (Ba  (None, 28, 28, 64)          256       ['conv2d_17[0][0]']
tchNormalization)

batch_normalization_18 (Ba  (None, 28, 28, 64)          256       ['conv2d_18[0][0]']
tchNormalization)

add_5 (Add)                 (None, 28, 28, 64)          0         ['batch_normalization_17[0][0]
                                                                  ',
                                                                   'batch_normalization_18[0][0]
                                                                  ']


re_lu_12 (ReLU)             (None, 28, 28, 64)          0         ['add_5[0][0]']

conv2d_19 (Conv2D)          (None, 28, 28, 128)         73856     ['re_lu_12[0][0]']

batch_normalization_19 (Ba  (None, 28, 28, 128)         512       ['conv2d_19[0][0]']
tchNormalization)

re_lu_13 (ReLU)             (None, 28, 28, 128)         0         ['batch_normalization_19[0][0]
                                                                  ']

conv2d_20 (Conv2D)          (None, 28, 28, 128)         147584    ['re_lu_13[0][0]']

conv2d_21 (Conv2D)          (None, 28, 28, 128)         8320      ['re_lu_12[0][0]']

batch_normalization_20 (Ba  (None, 28, 28, 128)         512       ['conv2d_20[0][0]']
tchNormalization)

batch_normalization_21 (Ba  (None, 28, 28, 128)         512       ['conv2d_21[0][0]']
tchNormalization)

add_6 (Add)                 (None, 28, 28, 128)         0         ['batch_normalization_20[0][0]
                                                                  ',
                                                                   'batch_normalization_21[0][0]
                                                                  ']

re_lu_14 (ReLU)             (None, 28, 28, 128)         0         ['add_6[0][0]']

conv2d_22 (Conv2D)          (None, 28, 28, 128)         147584    ['re_lu_14[0][0]']

batch_normalization_22 (Ba  (None, 28, 28, 128)         512       ['conv2d_22[0][0]']
tchNormalization)
```

```
re_lu_15 (ReLU)             (None, 28, 28, 128)          0         ['batch_normalization_22[0][0]
                                                                   ']

conv2d_23 (Conv2D)          (None, 28, 28, 128)          147584    ['re_lu_15[0][0]']

conv2d_24 (Conv2D)          (None, 28, 28, 128)          16512     ['re_lu_14[0][0]']

batch_normalization_23 (Ba  (None, 28, 28, 128)          512       ['conv2d_23[0][0]']
tchNormalization)

batch_normalization_24 (Ba  (None, 28, 28, 128)          512       ['conv2d_24[0][0]']
tchNormalization)

add_7 (Add)                 (None, 28, 28, 128)          0         ['batch_normalization_23[0][0]
                                                                   ',
                                                                    'batch_normalization_24[0][0]
                                                                   ']

re_lu_16 (ReLU)             (None, 28, 28, 128)          0         ['add_7[0][0]']

conv2d_25 (Conv2D)          (None, 28, 28, 128)          147584    ['re_lu_16[0][0]']

batch_normalization_25 (Ba  (None, 28, 28, 128)          512       ['conv2d_25[0][0]']
tchNormalization)

re_lu_17 (ReLU)             (None, 28, 28, 128)          0         ['batch_normalization_25[0][0]
                                                                   ']


conv2d_26 (Conv2D)          (None, 28, 28, 128)          147584    ['re_lu_17[0][0]']

conv2d_27 (Conv2D)          (None, 28, 28, 128)          16512     ['re_lu_16[0][0]']

batch_normalization_26 (Ba  (None, 28, 28, 128)          512       ['conv2d_26[0][0]']
tchNormalization)

batch_normalization_27 (Ba  (None, 28, 28, 128)          512       ['conv2d_27[0][0]']
tchNormalization)

add_8 (Add)                 (None, 28, 28, 128)          0         ['batch_normalization_26[0][0]
                                                                   ',
                                                                    'batch_normalization_27[0][0]
                                                                   ']

re_lu_18 (ReLU)             (None, 28, 28, 128)          0         ['add_8[0][0]']

global_average_pooling2d (  (None, 128)                  0         ['re_lu_18[0][0]']
GlobalAveragePooling2D)

flatten (Flatten)           (None, 128)                  0         ['global_average_pooling2d[0][
                                                                   0]']

dense (Dense)               (None, 10)                   1290      ['flatten[0][0]']

==================================================================================================
Total params: 1135146 (4.33 MB)
Trainable params: 1131050 (4.31 MB)
Non-trainable params: 4096 (16.00 KB)
__________________________________________________________________________________________________
```

```
# Compile the model (customize optimizer, loss, and metrics as needed)
resnet_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# Train the model with your training data
history = resnet_model.fit(train_images, train_labels, epochs=10, validation_split=0.2,batch_size=32)
```

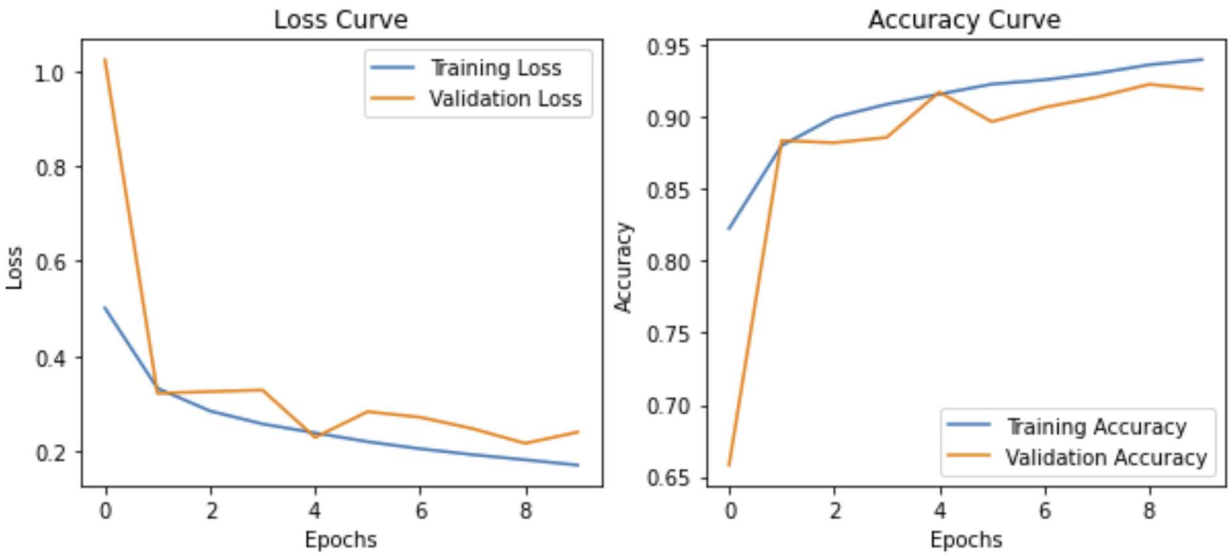
```
Epoch 1/10
1500/1500 [==============================] - 117s 58ms/step - loss: 0.4932 - accuracy: 0.8247 - val_loss: 0.4124 - val_accuracy: 0.8593
Epoch 2/10
1500/1500 [==============================] - 86s 57ms/step - loss: 0.3251 - accuracy: 0.8840 - val_loss: 0.3985 - val_accuracy: 0.8597
Epoch 3/10
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2781 - accuracy: 0.9006 - val_loss: 0.3072 - val_accuracy: 0.8883
Epoch 4/10
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2493 - accuracy: 0.9106 - val_loss: 0.2506 - val_accuracy: 0.9125
Epoch 5/10
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2262 - accuracy: 0.9191 - val_loss: 0.2875 - val_accuracy: 0.8997
Epoch 6/10
1500/1500 [==============================] - 86s 57ms/step - loss: 0.2146 - accuracy: 0.9235 - val_loss: 0.2972 - val_accuracy: 0.8997
Epoch 7/10
1500/1500 [==============================] - 90s 60ms/step - loss: 0.2017 - accuracy: 0.9279 - val_loss: 0.2783 - val_accuracy: 0.9009
Epoch 8/10
1500/1500 [==============================] - 86s 57ms/step - loss: 0.1892 - accuracy: 0.9328 - val_loss: 0.2472 - val_accuracy: 0.9124
Epoch 9/10
1500/1500 [==============================] - 90s 60ms/step - loss: 0.1765 - accuracy: 0.9367 - val_loss: 0.2598 - val_accuracy: 0.9097
Epoch 10/10
1500/1500 [==============================] - 86s 57ms/step - loss: 0.1679 - accuracy: 0.9400 - val_loss: 0.2335 - val_accuracy: 0.9181
```

```
# Save the model to a file
resnet_model.save('fashion_mnist_model.h5')
```

```
# Load the saved model
loaded_model = keras.models.load_model('fashion_mnist_model.h5')
```

```
# Evaluate the loaded model on the test data
test_loss, test_accuracy = loaded_model.evaluate(test_images, test_labels)
print("Test accuracy:", test_accuracy)
```

```
313/313 [==============================] - 55s 174ms/step - loss: 0.2540 - accuracy: 0.9136
Test accuracy: 0.9136000275611877
```



ResNet from scratch gave an accuracy of 91.36%