# REPORT

# ASSIGNMENT 2

ZAIN SHAFIQUE
290213
BESE-10A

## 1. Importing necessary libraries:

### Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import fashion_mnist
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
from keras.optimizers import Adam
from keras.utils import to_categorical
from keras.applications.vgg16 import VGG16
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLR
OnPlateau
```

### Explanation:

The above code imports the necessary libraries and modules for building and training a deep learning model for classifying images from the Fashion MNIST dataset using the VGG16 architecture.

## 2. Importing Data:

### Code:

```
# Importing the data
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

### Explanation:

This code imports the Fashion MNIST dataset using Keras' fashion_mnist module, and splits it into training and test sets.

fashion_mnist.load_data() loads the Fashion MNIST dataset, which consists of **70,000 images** of clothing items belonging to 10 classes.
The dataset is split into two sets: **a training set with 60,000 images and a test set with 10,000 images.**

## Working with VGG16:

i) Change the grid input dimension: The VGG16 was initially trained for color images. Therefore, it needs to receive images with color channels. That is, each pixel of the image will be represented by more than one value. Typically, we use the RGB color system, that is, each pixel of the image is composed of three values. We can therefore say that an image 32 pixels wide and 32 pixels high in the RGB system has a dimensionality of 32x32x3. As the Fashion MNIST images are in grayscale (that is, they only have one color channel and, therefore, each image has a size of 28x28x1), it is necessary to resize the dataset images to 32x32x3, adding another two channels to use the VGG16.

ii) Change the output layer: The output layer of the original VGG16 has 1000 neurons, which correspond to the number of classes in the ImageNet dataset. For Fashion MNIST, you need to change the number of neurons in the output layer to 10, which is the number of classes in the dataset. Therefore, it is necessary to replace the VGG16 output layer with a new dense layer with 10 neurons and a softmax activation layer for classification;

iii) Tuning the optimizer learning rate: As Fashion MNIST is a simpler dataset compared to ImageNet, it may be necessary to adjust the optimizer learning rate to a lower value in order to avoid overfitting.

## 3. Pre-processing Data:

Code:

```
def carregamento_dados():

# Define the number of classes in the dataset
   num_classes = 10

# Load the Fashion MNIST dataset
   (X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

# Resize the images to the VGG16 input size (32x32x3)
   X_train = np.array([cv2.resize(img, (32, 32)) for img in X_train]
)
   X_test = np.array([cv2.resize(img, (32, 32)) for img in X_test])

# Add a channel for grayscale (3rd dimension) images
   X_train = np.stack((X_train,) * 3, axis=-1)
   X_test = np.stack((X_test,) * 3, axis=-1)

# Normalize the images
   X_train = X_train.astype('float32') / 255
   X_test = X_test.astype('float32') / 255

# Convert labels to one-hot encoding format
   y_train = to_categorical(y_train)
   y_test = to_categorical(y_test)
```

```
    return X_train, y_train, X_test, y_test
```

Explanation:

This is a function **carregamento_dados()** that loads and pre-processes the Fashion MNIST dataset. The pre-processing steps include:

- Setting the number of classes in the dataset to **num_classes = 10**.
- Loading the Fashion MNIST dataset using fashion_mnist.load_data().
- Resizing the images to the VGG16 input size of **32x32x3** using OpenCV's cv2.resize() function.
- Adding a channel for grayscale images to create 3-channel RGB images using NumPy's np.stack() function.
- Normalizing the pixel values of the images to the range [0, 1].
- Converting the labels to one-hot encoding format using Keras' to_categorical() function.

The function returns the pre-processed training and test sets as NumPy arrays X_train, y_train, X_test, and y_test.

# 4. Model Compiling using VGG16 Architecture:

Code:

```
def modelo_compile():

# Load the VGG16 model without the output layer
    base_model = VGG16(weights = None, include_top=False, input_shape
=(32, 32, 3))

# Add custom dense layers on top of model
    modelo = Sequential()
    modelo.add(base_model)
    modelo.add(Flatten())
    modelo.add(Dense(256, activation='relu'))
    modelo.add(Dropout(0.5))
    modelo.add(Dense(10, activation='softmax'))

# Set the Adam optimizer with reduced learning rate
    optimizer = Adam(learning_rate=0.0002) # Original is 0.0001

# Compile the model
    modelo.compile(loss = 'categorical_crossentropy', optimizer = opt
imizer, metrics = ['accuracy'])

    return modelo
```

Explanation:

This is a function **modelo_compile()** that defines and compiles a custom model using the VGG16 architecture as a base model. The function includes the following steps:

- Loading the VGG16 model without the output layer using Keras' VGG16() function. The weights=None argument initializes the weights of the model randomly.
- Adding custom dense layers on top of the base model using Keras' Sequential() function. The Flatten() layer flattens the output of the base model to a 1D vector. The first Dense() layer has 256 units with ReLU activation, followed by a Dropout() layer with a dropout rate of 0.5 to reduce overfitting. The final Dense() layer has 10 units with softmax activation, corresponding to the 10 classes in the dataset.
- Setting the Adam optimizer with a reduced learning rate of 0.0002 (originally 0.0001).
- Compiling the model with categorical cross-entropy loss, Adam optimizer, and accuracy as the evaluation metric.

The function returns the compiled model.

# 5. Plotting loss and accuracy graphs:

## Code:

```
def plot_training_history(history):
    # Plot training & validation loss values
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Test'], loc='upper left')
    plt.show()

    # Plot training & validation accuracy values
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Test'], loc='upper left')
    plt.show()
```

## Explanation:

The above code plots the graphs of test and train loss and accuracy graphs.

# 6. Model fitting:

## Code:

```
def modelo_fit(modelo, X_train, y_train, X_test, y_test):

# Setting callbacks
```

```
  earlyStopping = EarlyStopping(monitor = 'val_loss', patience = 10,
 verbose = 0, mode = 'min')
  mcp_save = ModelCheckpoint('best_weights.hdf5', save_best_only = T
rue, monitor = 'val_loss', mode = 'min')
  reduce_lr_loss = ReduceLROnPlateau(monitor = 'val_loss', factor =
0.4, patience = 7, verbose = 1, min_delta = 1e-4, mode = 'auto')

# Training model
  history = modelo.fit(X_train, y_train, batch_size = 32, epochs = 2
0, validation_data = (X_test, y_test), callbacks = [earlyStopping, m
cp_save, reduce_lr_loss])

  # Evaluate model
  _, acc = modelo.evaluate(X_test, y_test, verbose=0)
  print('> %.3f' % (acc * 100.0))

# Saving metrics
  scores = acc
  historico = history

  # Plot training history
  plot_training_history(history)

  return scores, historico
```

## Explanation:

This function defines a training procedure for the deep learning model. It takes as input the compiled model, training and testing data, and returns the evaluation score and the training history.

The training procedure involves setting up callbacks, which are functions that can be executed during the training process. In this case, three callbacks are used:

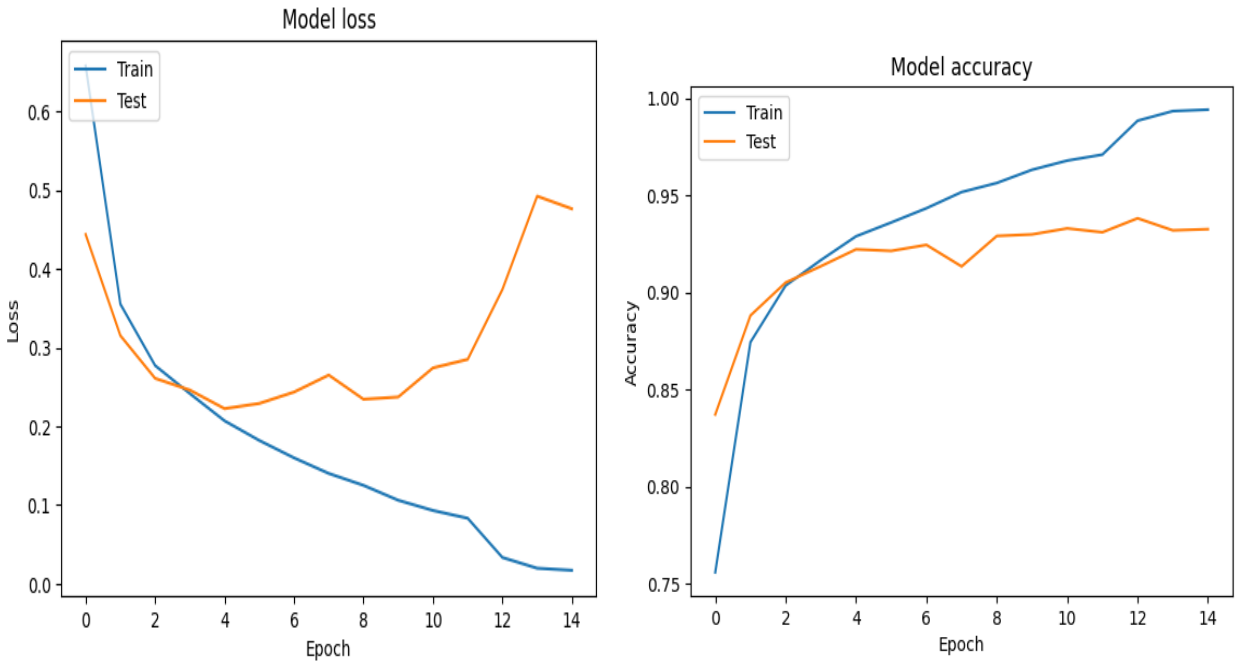
- **EarlyStopping:** This stops the training process if the monitored metric (validation loss in this case) does not improve for a certain number of epochs (patience). It helps to prevent overfitting and saves time by stopping the training process early.

- **ModelCheckpoint:** This saves the weights of the best-performing model based on the monitored metric (validation loss in this case). This is useful in case the training process is interrupted, as the best weights can be loaded later.

- **ReduceLROnPlateau:** This reduces the learning rate when the monitored metric (validation loss in this case) stops improving. This helps to fine-tune the model and avoid getting stuck in local minima.

The model is then trained using the fit() method, with the training data, batch size, number of epochs, validation data, and callbacks specified. After training, the model is evaluated on the testing data using the evaluate() method, and the evaluation score is returned along with the training history.

## Train and Test loss and accuracy graphs:




**Overall Model Accuracy: 93.2%**