CISC 867 Project 2:

Applying CNN architectures on Fashion-MNIST dataset

Prepared By:

[Esraa Elsayed] **ID: 21ESMS**

[Fatma Bleity] **ID: 21FKMA**

[Gehan Essa] **ID: 21GHHE**

Submitted to:

[Dr. Hazem Abbas]

Table of content

Table of figures:

1. Introduction

1.1. Objective of the project

Implement a LeNet-5 network to recognize the Fashion MNIST digits.

1.2. The Dataset Used

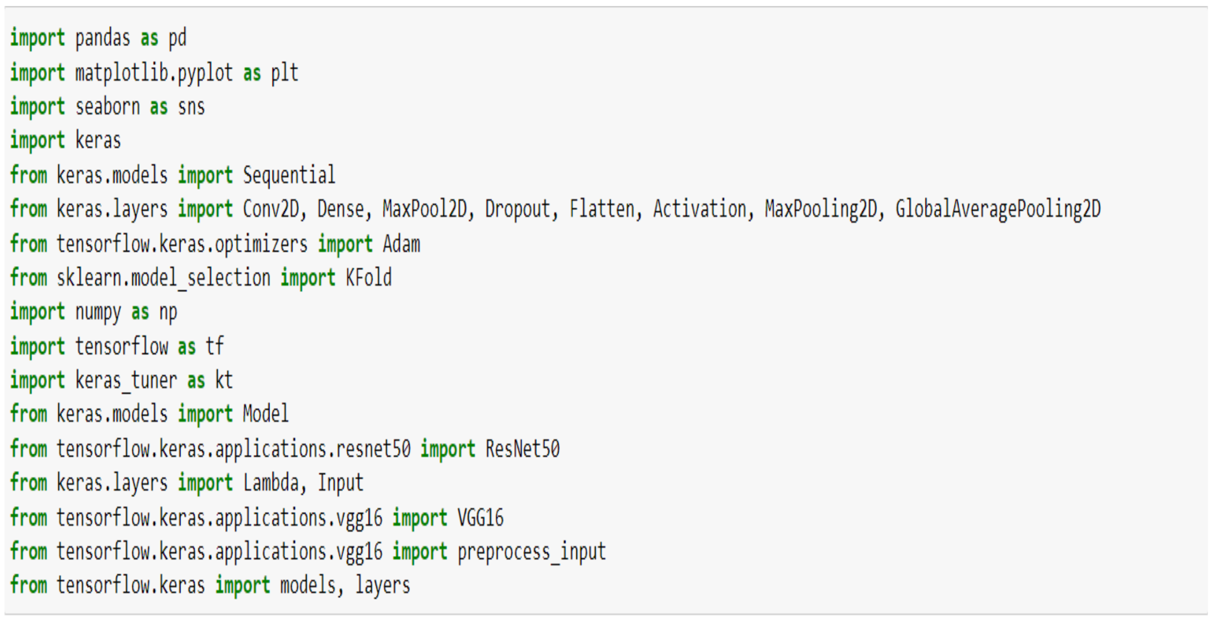
Fashion-MNIST is a dataset consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

Labels:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

2.The libraries

We used some important libraries in python to help us for training a CNN neural network that shown in the figure below.



```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import keras
from keras.models import Sequential
from keras.layers import Conv2D, Dense, MaxPool2D, Dropout, Flatten, Activation, MaxPooling2D, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np
import tensorflow as tf
import keras_tuner as kt
from keras.models import Model
from tensorflow.keras.applications.resnet50 import ResNet50
from keras.layers import Lambda, Input
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras import models, layers
```

Figure 1. Import the required libraries

- **Numpy**

The name “Numpy” stands for “Numerical Python”. It is the commonly used library. It is a popular machine learning library that supports large matrices and multi- dimensional data. It consists of in-built mathematical functions for easy computations. Even libraries like TensorFlow use Numpy internally to perform several operations on tensors. Array Interface is one of the key features of this library.

- **Pandas**

  Pandas are an important library for data scientists. It is an open- source machine learning library that provides flexible high-level data structures and a varietyof analysis tools. It eases data analysis, data manipulation, and cleaning of data. Pandas support operations like Sorting, Re-indexing, Iteration, Concatenation, Conversion of data, Visualizations, Aggregations, etc.

- **Matplotlib**

  This library is responsible for plotting numerical data. And that's why it is used in data analysis. It is also an open-source library and plots high-defined figures like pie charts, histograms, scatterplots, graphs, etc.

- **SciKit-learn**

  It is a famous Python library to work with complex data. Scikit-learn isan open-source library that supports machine learning. It supports variously supervised and unsupervised algorithms like linear regression, classification, clustering, etc. This library works in association with Numpy and SciPy.

- **Tensorflow**

  It is a foundation library that can be used to create Deep Learning models directly or by using wrapper libraries that simplify the process built on top of TensorFlow, and used for fast numerical computing.

- **Keras**

  It is a powerful and easy-to-use free open source Python library for developing and evaluating deep learning models.

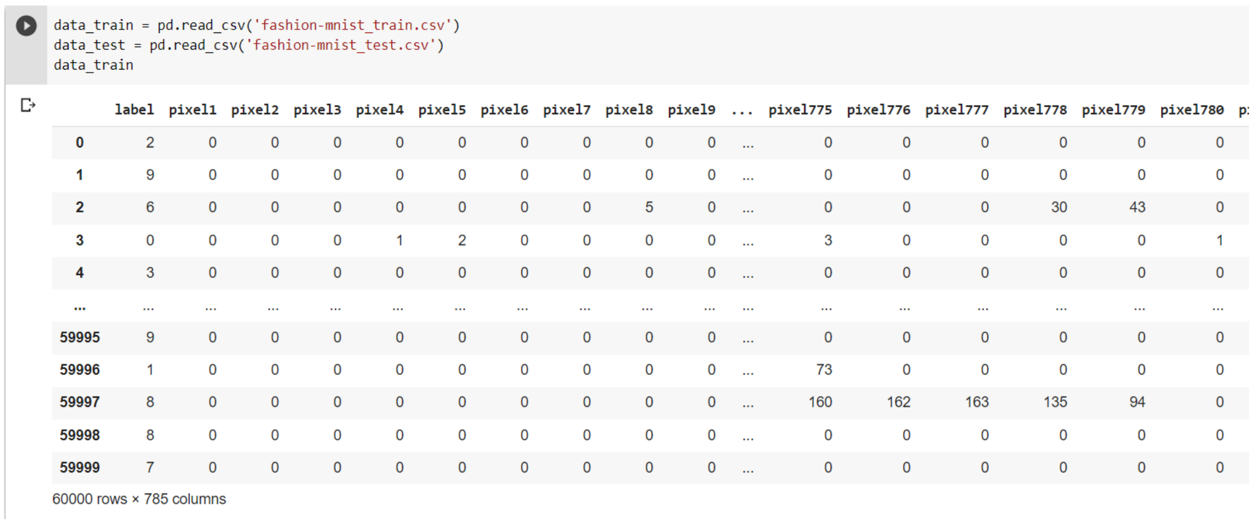
- **Seaborn**

  Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

3. Part I

3.1. Data Preparation

3.1.1. Download the data file and load it



```
data_train = pd.read_csv('fashion-mnist_train.csv')
data_test = pd.read_csv('fashion-mnist_test.csv')
data_train
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	p:
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
59995	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	
59996	1	0	0	0	0	0	0	0	0	0	...	73	0	0	0	0	0	
59997	8	0	0	0	0	0	0	0	0	0	...	160	162	163	135	94	0	
59998	8	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	
59999	7	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	

60000 rows × 785 columns

Figure 2. Reading the data

We use ‘read_csv’ function to read the dataset file.

As we saw after running the training data, it shows that it consists of **60000 rows × 785 columns.**

3.1.2. Describe the data

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels, and represents the article of clothing. The rest of the columns contain the pixel-values of the associated image.

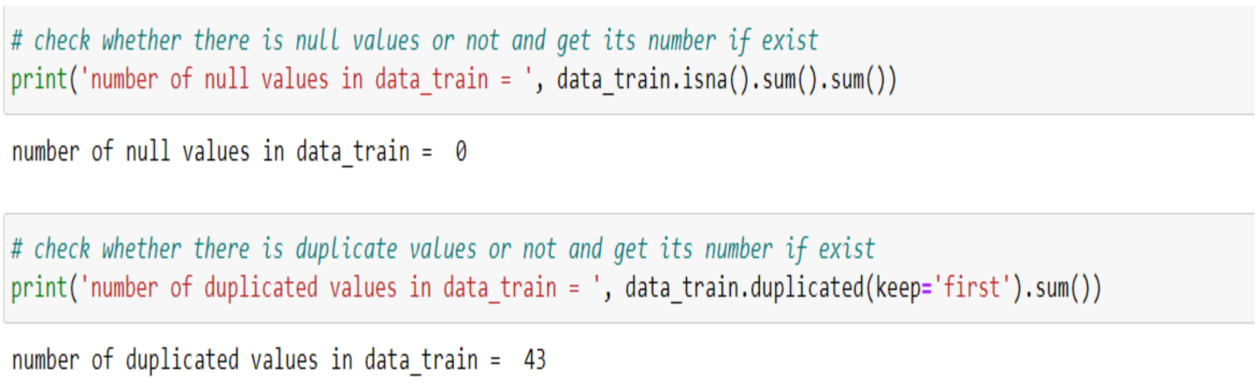
- Each row is a separate image
- Column 1 is the class label
- The columns remained are pixel numbers (784 total).
- Each value is the darkness of the pixel (1 to 255).

3.1.3. Clean the data

According to **Garbage in Garbage out** phrase:

If I let the dataset full of Nulls or duplicated rows so, our model will not train well and gives rubbish output.

So, we checked if there is any missing or duplicated values in our dataset to avoid it by some preprocessing.

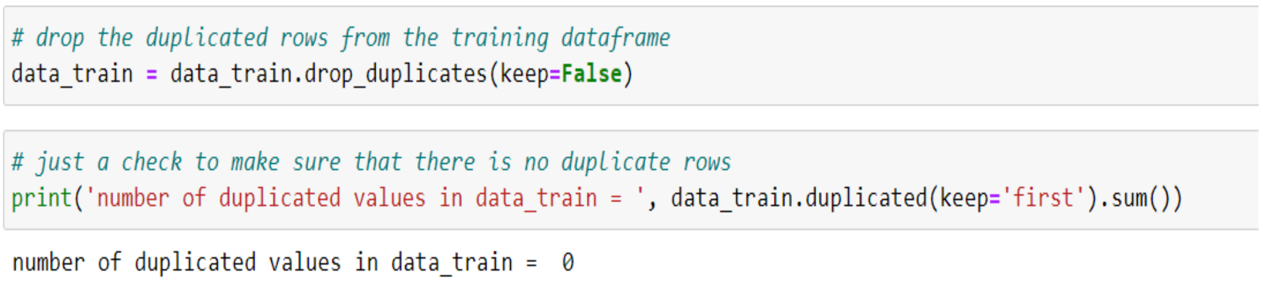


```
# check whether there is null values or not and get its number if exist
print('number of null values in data_train = ', data_train.isna().sum().sum())

number of null values in data_train =  0

# check whether there is duplicate values or not and get its number if exist
print('number of duplicated values in data_train = ', data_train.duplicated(keep='first').sum())

number of duplicated values in data_train =  43
```

Figure 3. Check missing and duplicated values

As shown in the figure above:

Our training dataset doesn’t contain any null values but has one duplicated value so we will drop it to help our model train well as clarified in the figure below.



```
# drop the duplicated rows from the training dataframe
data_train = data_train.drop_duplicates(keep=False)
```

```
# just a check to make sure that there is no duplicate rows
print('number of duplicated values in data_train = ', data_train.duplicated(keep='first').sum())
```

```
number of duplicated values in data_train =  0
```

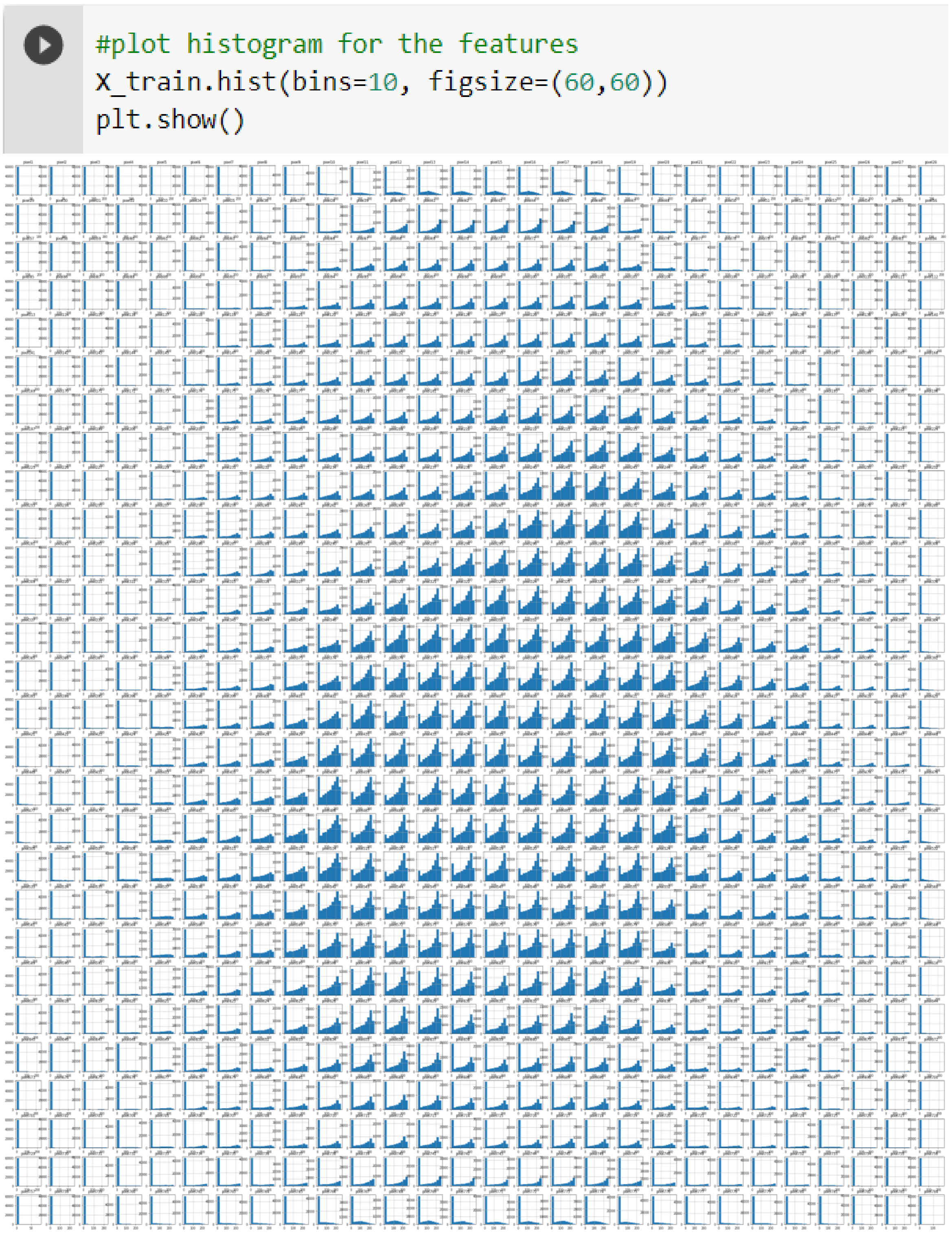
Figure 4. Drop duplicated values

3.1.4. Visualize the data using proper visualization methods

1. **Plot the histogram for the features.**

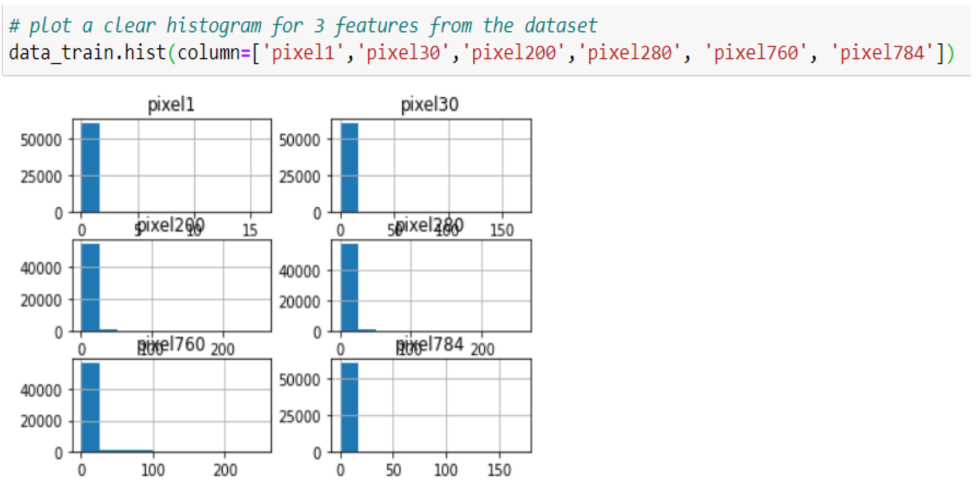
   Perhaps the most common approach to visualize a distribution and evaluate the data is the *histogram*. A histogram is a bar plot where the axis representing the data variable is divided into a set of discrete bins and the count of observations falling within each bin is shown using the height of the corresponding bar.

These figures show that the value of each pixel's intensity at the edges of each image is completely black. However, the rest is what defines the content of image in terms of labels.



```
#plot histogram for the features
X_train.hist(bins=10, figsize=(60,60))
plt.show()
```

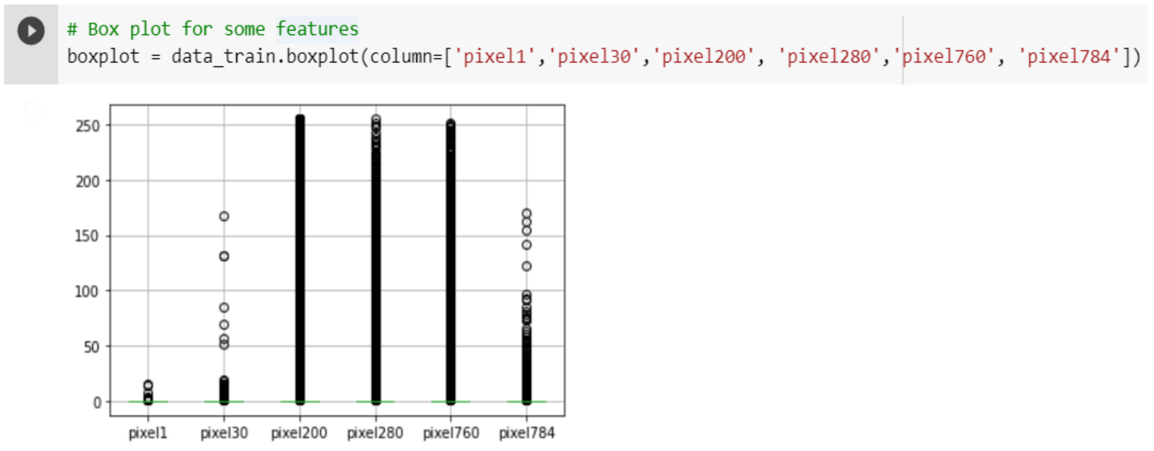
Figure 5. Visualize the data using histogram



```
# plot a clear histogram for 3 features from the dataset
data_train.hist(column=['pixel1','pixel30','pixel200','pixel280', 'pixel760', 'pixel784'])
```

Figure 6. Visualize the data using histogram for some features

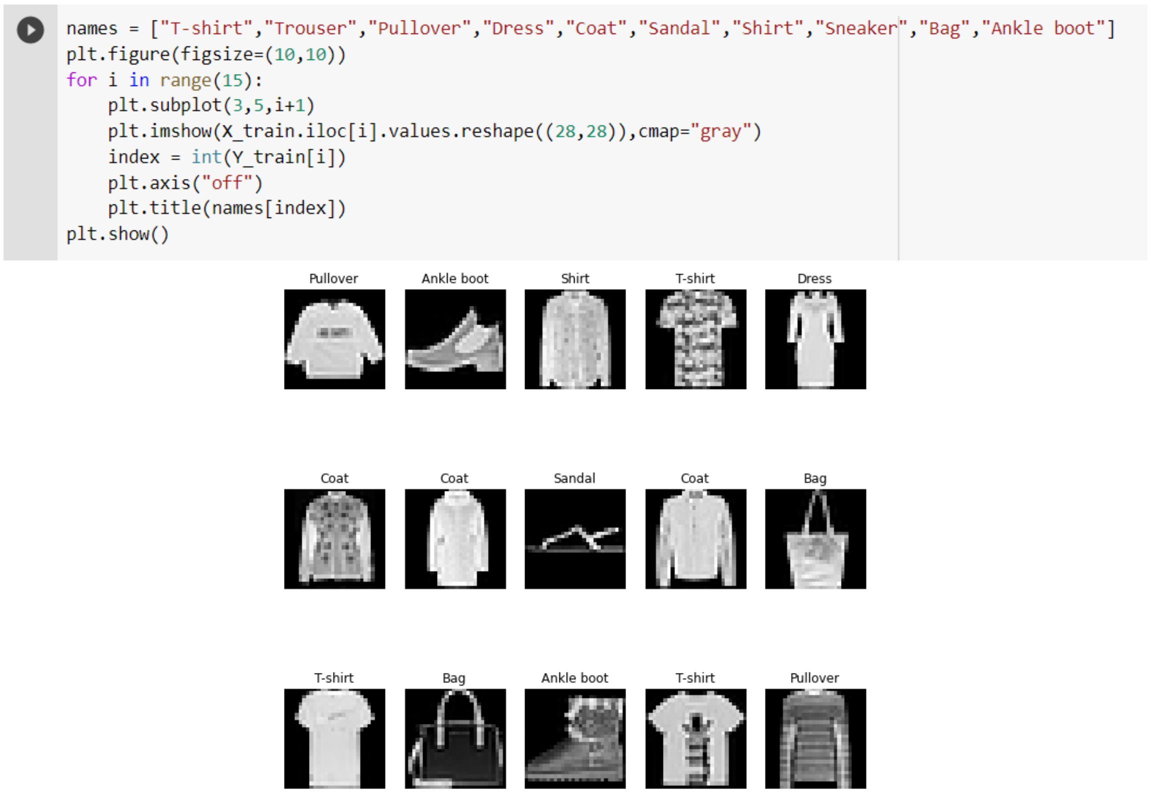
2. Using box plot to visualize some of the features and check whether there are outliers or not.



```
# Box plot for some features
boxplot = data_train.boxplot(column=['pixel1','pixel30','pixel200', 'pixel280','pixel760', 'pixel784'])
```

Figure 7. Box plot for some of the features

3.1.5. Draw some of the images

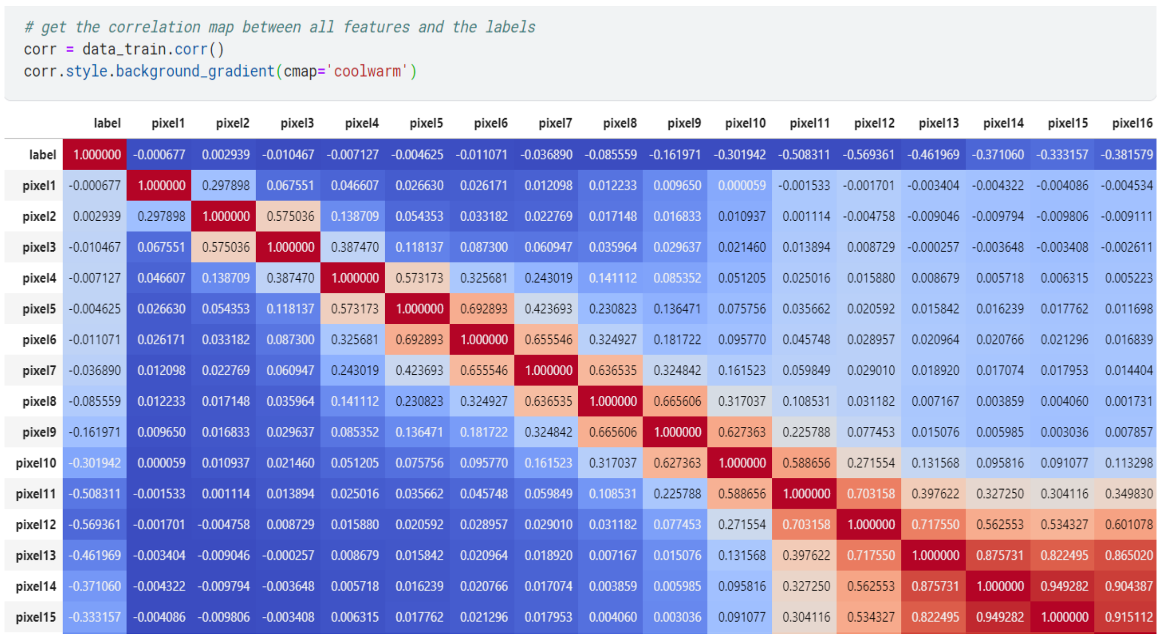


```
names = ["T-shirt","Trouser","Pullover","Dress","Coat","Sandal","Shirt","Sneaker","Bag","Ankle boot"]
plt.figure(figsize=(10,10))
for i in range(15):
    plt.subplot(3,5,i+1)
    plt.imshow(X_train.iloc[i].values.reshape((28,28)),cmap="gray")
    index = int(Y_train[i])
    plt.axis("off")
    plt.title(names[index])
plt.show()
```

Figure 8. Draw some of the images

3.1.6. Carry out required correlation analysis

A **correlation matrix** is a table that displays the coefficients of correlation between variables. It can show whether or not two variables are correlated and how strongly they are related. The correlation between two variables is shown in each cell of the table.



```
# get the correlation map between all features and the labels
corr = data_train.corr()
corr.style.background_gradient(cmap='coolwarm')
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	pixel11	pixel12	pixel13	pixel14	pixel15	pixel16
label	1.000000	-0.000677	0.002939	-0.010467	-0.007127	-0.004625	-0.011071	-0.036890	-0.085559	-0.161971	-0.301942	-0.508311	-0.569361	-0.461969	-0.371060	-0.333157	-0.381579
pixel1	-0.000677	1.000000	0.297898	0.067551	0.046607	0.026630	0.026171	0.012098	0.012233	0.009650	0.000059	-0.001533	-0.001701	-0.003404	-0.004322	-0.004086	-0.004534
pixel2	0.002939	0.297898	1.000000	0.575036	0.138709	0.054353	0.033182	0.022769	0.017148	0.016833	0.010937	0.001114	-0.004758	-0.009046	-0.009794	-0.009806	-0.009111
pixel3	-0.010467	0.067551	0.575036	1.000000	0.387470	0.118137	0.087300	0.060947	0.035964	0.029637	0.021460	0.013894	0.008729	-0.000257	-0.003648	-0.003408	-0.002611
pixel4	-0.007127	0.046607	0.138709	0.387470	1.000000	0.573173	0.325681	0.243019	0.141112	0.085352	0.051205	0.025016	0.015880	0.008679	0.005718	0.006315	0.005223
pixel5	-0.004625	0.026630	0.054353	0.118137	0.573173	1.000000	0.692893	0.423693	0.230823	0.136471	0.075756	0.035662	0.020592	0.015842	0.016239	0.017762	0.011698
pixel6	-0.011071	0.026171	0.033182	0.087300	0.325681	0.692893	1.000000	0.655546	0.324927	0.181722	0.095770	0.045748	0.028957	0.020964	0.020766	0.021296	0.016839
pixel7	-0.036890	0.012098	0.022769	0.060947	0.243019	0.423693	0.655546	1.000000	0.636535	0.324842	0.161523	0.059849	0.029010	0.018920	0.017074	0.017953	0.014404
pixel8	-0.085559	0.012233	0.017148	0.035964	0.141112	0.230823	0.324927	0.636535	1.000000	0.665606	0.317037	0.108531	0.031182	0.007167	0.003859	0.004060	0.001731
pixel9	-0.161971	0.009650	0.016833	0.029637	0.085352	0.136471	0.181722	0.324842	0.665606	1.000000	0.627363	0.225788	0.077453	0.015076	0.005985	0.003036	0.007857
pixel10	-0.301942	0.000059	0.010937	0.021460	0.051205	0.075756	0.095770	0.161523	0.317037	0.627363	1.000000	0.588656	0.271554	0.131568	0.095816	0.091077	0.113298
pixel11	-0.508311	-0.001533	0.001114	0.013894	0.025016	0.035662	0.045748	0.059849	0.108531	0.225788	0.588656	1.000000	0.703158	0.397622	0.327250	0.304116	0.349830
pixel12	-0.569361	-0.001701	-0.004758	0.008729	0.015880	0.020592	0.028957	0.029010	0.031182	0.077453	0.271554	0.703158	1.000000	0.717550	0.562553	0.534327	0.601078
pixel13	-0.461969	-0.003404	-0.009046	-0.000257	0.008679	0.015842	0.020964	0.018920	0.007167	0.015076	0.131568	0.397622	0.717550	1.000000	0.875731	0.822495	0.865020
pixel14	-0.371060	-0.004322	-0.009794	-0.003648	0.005718	0.016239	0.020766	0.017074	0.003859	0.005985	0.095816	0.327250	0.562553	0.875731	1.000000	0.949282	0.904387
pixel15	-0.333157	-0.004086	-0.009806	-0.003408	0.006315	0.017762	0.021296	0.017953	0.004060	0.003036	0.091077	0.304116	0.534327	0.822495	0.949282	1.000000	0.915112

Figure 9. Part of the correlation analysis

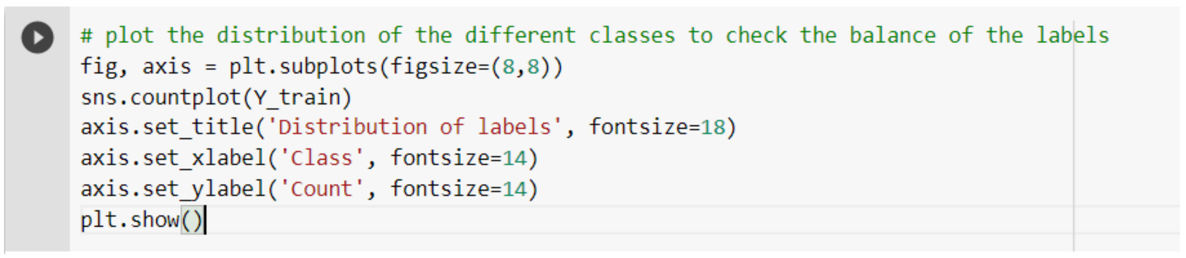
3.2. Carry out any required preprocessing operations on the data

3.2.1. Check data balance for the label

Having a balanced data set for a model, would generate higher accuracy models, higher balanced accuracy and balanced detection rate. Hence, it’s important to have a balanced dataset for a classification model.

If we get the number of each label, we will get:

label 0 → 5996, label 1 → 5992, label 2 → 5976, label 3 → 5994, label 4 → 5990, label 5 → 6000, label 6 → 5978, label 7 → 5992, label 8 → 6000, label 9 → 5996



```python
# plot the distribution of the different classes to check the balance of the labels
fig, axis = plt.subplots(figsize=(8,8))
sns.countplot(Y_train)
axis.set_title('Distribution of labels', fontsize=18)
axis.set_xlabel('Class', fontsize=14)
axis.set_ylabel('Count', fontsize=14)
plt.show()
```

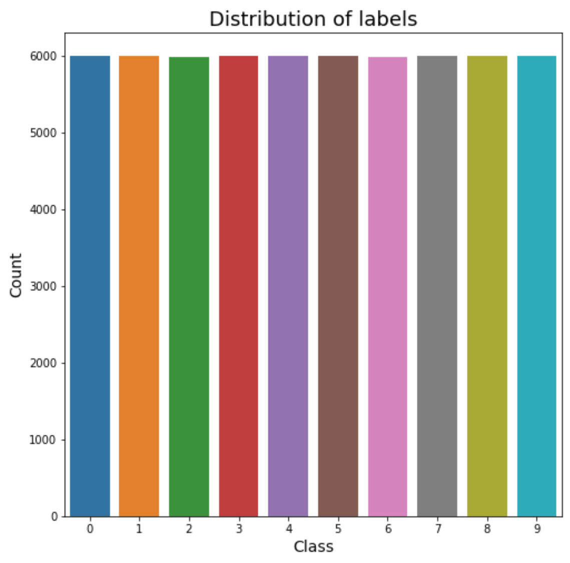
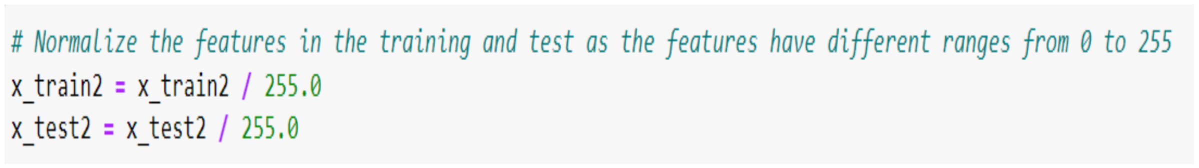


Figure 10. Check the data balance

This means that there is balance in the weights of each class in the data.

3.2.2. Normalize the data

As our data has different ranges from 0 to 255, so we need make it normalized. Rescaling real-valued numeric characteristics into a 0 to 1 range is referred to as normalization. Data normalization is used to make model training less sensitive to feature scale. As a result, our model can converge to better weights, resulting in a more accurate model. So we will normalize the inputs of the dataset.



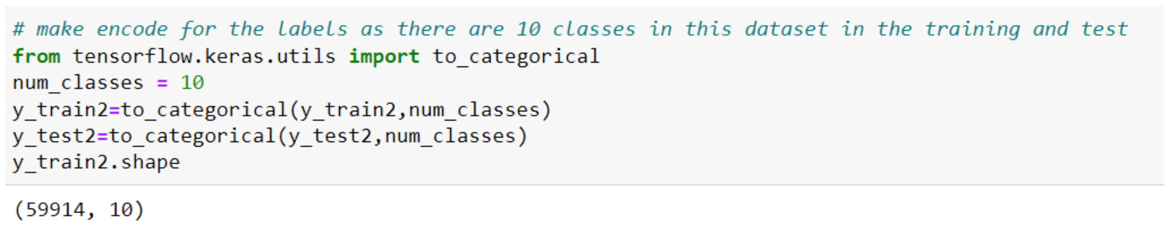
```python
# Normalize the features in the training and test as the features have different ranges from 0 to 255
x_train2 = x_train2 / 255.0
x_test2 = x_test2 / 255.0
```

Figure 11. Normalize the data

3.3. Encode the labels

Models require all input and output variables to be numeric, which means that in case of categorical data, we must encode it to numbers before I can fit and evaluate a model.

We'll apply 'to_categorical' on it to encode the label which represents in y_train2 and y_test2 variables.



```
# make encode for the labels as there are 10 classes in this dataset in the training and test
from tensorflow.keras.utils import to_categorical
num_classes = 10
y_train2=to_categorical(y_train2,num_classes)
y_test2=to_categorical(y_test2,num_classes)
y_train2.shape
```

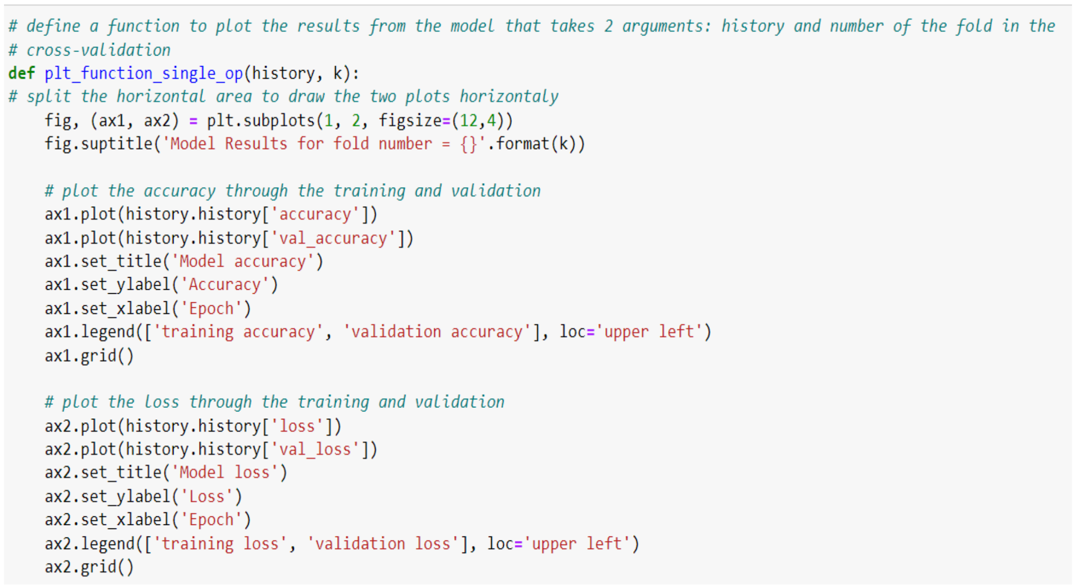
```
(59914, 10)
```

Figure 12. Encode the label

4. Part II

4.1. Make the function to plot the results for each model

We implemented a function to plot the loss and accuracy for the model during the training phase. So, we can use the difference in the loss and accuracy between the training and validation datasets.



```python
# define a function to plot the results from the model that takes 2 arguments: history and number of the fold in the
# cross-validation
def plt_function_single_op(history, k):
# split the horizontal area to draw the two plots horizontaly
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,4))
    fig.suptitle('Model Results for fold number = {}'.format(k))

    # plot the accuracy through the training and validation
    ax1.plot(history.history['accuracy'])
    ax1.plot(history.history['val_accuracy'])
    ax1.set_title('Model accuracy')
    ax1.set_ylabel('Accuracy')
    ax1.set_xlabel('Epoch')
    ax1.legend(['training accuracy', 'validation accuracy'], loc='upper left')
    ax1.grid()

    # plot the loss through the training and validation
    ax2.plot(history.history['loss'])
    ax2.plot(history.history['val_loss'])
    ax2.set_title('Model loss')
    ax2.set_ylabel('Loss')
    ax2.set_xlabel('Epoch')
    ax2.legend(['training loss', 'validation loss'], loc='upper left')
    ax2.grid()
```

Figure 13. Plotting function

4.2. Build the LeNet 5 model

4.2.1. Steps of the model

First as we mentioned before that our datasets are images with size 28x28 with 1 channel as they are grayscale images. So our model will take the input shape as (28, 28, 1).

Our model consists from convolution layer with filter size 5x5, activation: ReLU, and with making padding for the input to get the output with the same size of the input.

So according to the equation to get the shape we will have ((N + 2P - F) / S) + 1 = ((28 + 4 - 5) / 1) + 1 = 28 x 28. With applying 32 filter so the output size from this layer is 28x28x32.

Then the second layer is Max pooling with stride = 2 to make dimensionality reduction, so the output shape is ((N + 2P -F) / S) + 1 = ((28 - 2) / 2) + 1 = 14 x 14 x 32.

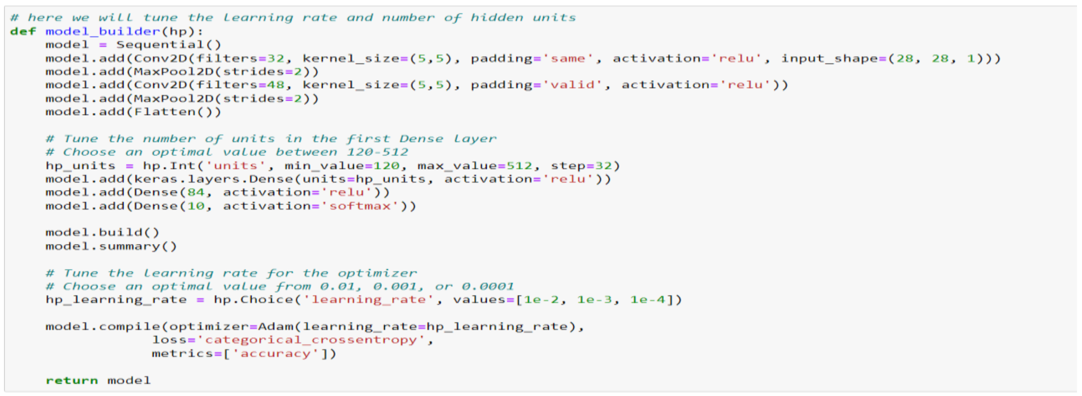
Then third layer is convolution with filters = 48, and kernel size = 5x5, and padding valid which means that there is no padding. So, the output shape is ((N + 2P - F) / S) + 1 = ((14 - 5) / 1) + 1 = 10 x 10 x 48.

Then in the fourth layer, Max pooling with stride = 2, so the output shape is = ((N + 2P -F) / S) + 1 = ((10 - 2) / 2) + 1 = 5 x 5 x 48.

Then we will make flatten for the results from the previous layer to use the results with some dense layers.

In the first layer in the dense layers, we used automated tuning to get the best number of the hidden units in it, by using 'keras tuner'. Then the next layer with 84 hidden units, then the next one with 10 units, and this is our output layer. As we have 10 categories, so we need 10 units in the last dense layer with activation = 'softmax' as it is multi-classification problem. In the dense layers, weights according to the number of units multiplied by the output of the previous layer, and bias by shape number of units multiplied by 1. As each unit has its bias. So, for example the last dense layer will have weights by shape = 84x10, and bias by shape = 10x1.

In the model compilation, we will use adam optimizer with tune its learning rate, loss = 'categorical_crossentropy' as we have multi categories, and accuracy metric.

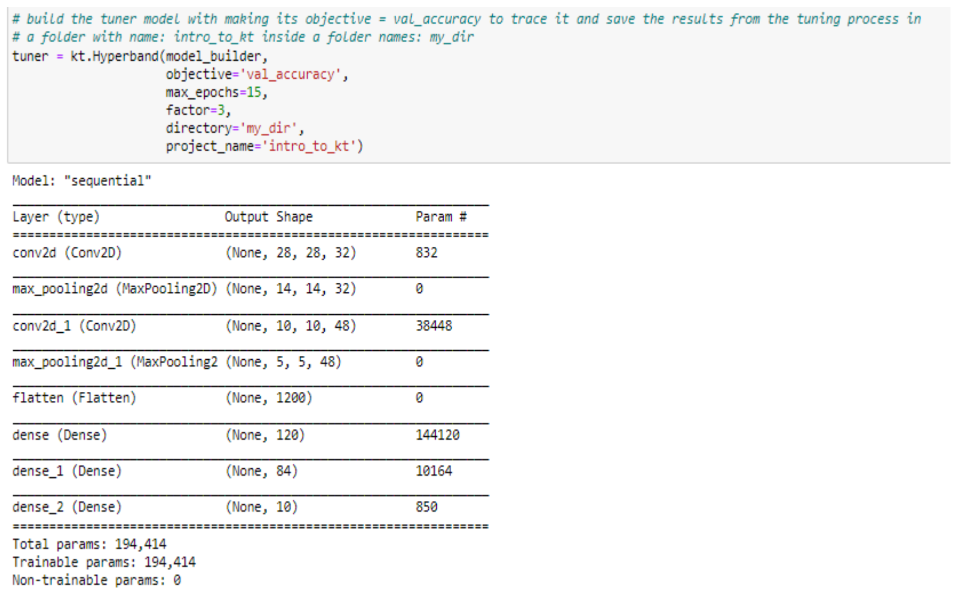


```
# here we will tune the learning rate and number of hidden units
def model_builder(hp):
    model = Sequential()
    model.add(Conv2D(filters=32, kernel_size=(5,5), padding='same', activation='relu', input_shape=(28, 28, 1)))
    model.add(MaxPool2D(strides=2))
    model.add(Conv2D(filters=48, kernel_size=(5,5), padding='valid', activation='relu'))
    model.add(MaxPool2D(strides=2))
    model.add(Flatten())

    # Tune the number of units in the first Dense layer
    # Choose an optimal value between 120-512
    hp_units = hp.Int('units', min_value=120, max_value=512, step=32)
    model.add(keras.layers.Dense(units=hp_units, activation='relu'))
    model.add(Dense(84, activation='relu'))
    model.add(Dense(10, activation='softmax'))

    model.build()
    model.summary()

    # Tune the learning rate for the optimizer
    # Choose an optimal value from 0.01, 0.001, or 0.0001
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.compile(optimizer=Adam(learning_rate=hp_learning_rate),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model
```

Figure 14. Our LeNet 5 model

4.2.2. Tuning the hyper-parameters

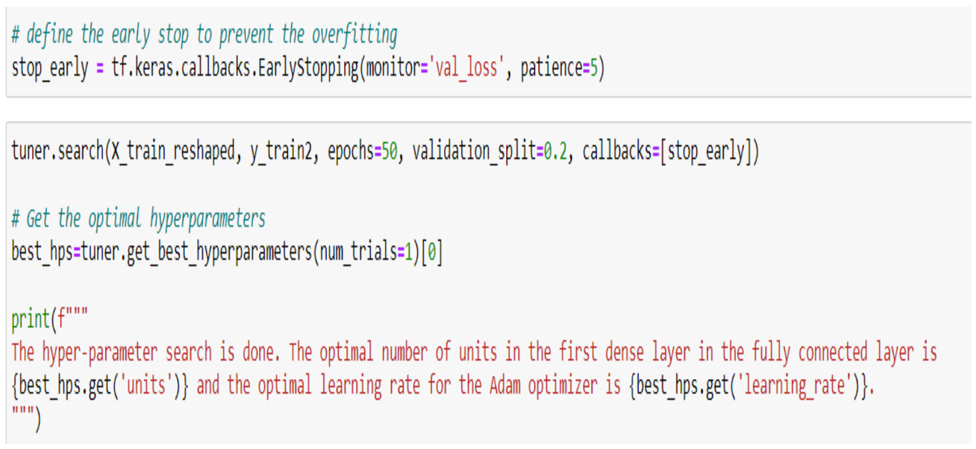
As we mentioned before, we will use ‘keras tuner’ to tune the number of the units in the first dense layer in the fully connected layers, and for tune the learning rate that will be used with the optimizer. With making early stopping to prevent the overfitting on the training data while making tuning for those hyper-parameters.



```
# build the tuner model with making its objective = val_accuracy to trace it and save the results from the tuning process in
# a folder with name: intro_to_kt inside a folder names: my_dir
tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=15,
                     factor=3,
                     directory='my_dir',
                     project_name='intro_to_kt')
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 48)        38448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 48)          0
_________________________________________________________________
flatten (Flatten)            (None, 1200)              0
_________________________________________________________________
dense (Dense)                (None, 120)               144120
_________________________________________________________________
dense_1 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 194,414
Trainable params: 194,414
Non-trainable params: 0
```

Figure 15. Tune the hyper-parameters with keras tuner



```
# define the early stop to prevent the overfitting
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

```
tuner.search(X_train_reshaped, y_train2, epochs=50, validation_split=0.2, callbacks=[stop_early])

# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
The hyper-parameter search is done. The optimal number of units in the first dense layer in the fully connected layer is
{best_hps.get('units')} and the optimal learning rate for the Adam optimizer is {best_hps.get('learning_rate')}.
""")
```

Figure 16. Get the best results from keras tuner

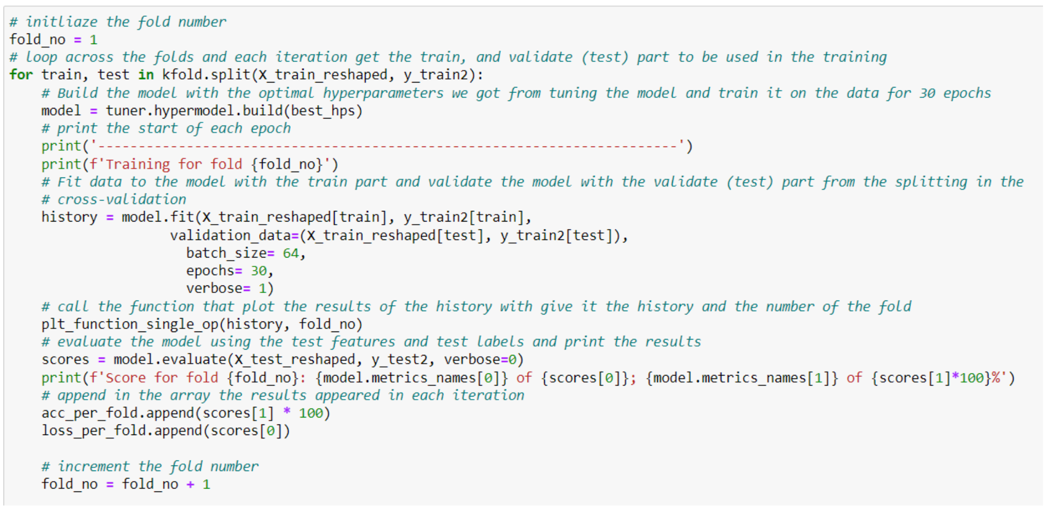
For tune the rest of the hyper-parameters, we used trial and error method by trying many different number of each parameters, and continue working with the best number that gave the best results. This happened with the number of the epochs, and batch size.

4.2.3. Cross-validation process

We split the data into 5 folds, with making shuffling between the split parts from the training set. As it split the training set into 5 parts, and with each iteration from the 5 iterations, it takes 4 parts from the split parts as the training data and one as the validation data. This is done with shuffling to make change each time in which is the validation set. Then using the tuned parameters that resulted from the tuning by keras tuner, we build the model.

Then in each iteration, we make fitting for the model on the training part, and the validation part. Then evaluate the model on the test dataset that we prepared it in the beginning.

Then is done after trying many different values for the number of the epochs, and the batch size. Then use number of epochs = 30, and batch size = 64 in the fitting the model.



```
# initliaze the fold number
fold_no = 1
# loop across the folds and each iteration get the train, and validate (test) part to be used in the training
for train, test in kfold.split(X_train_reshaped, y_train2):
    # Build the model with the optimal hyperparameters we got from tuning the model and train it on the data for 30 epochs
    model = tuner.hypermodel.build(best_hps)
    # print the start of each epoch
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no}')
    # Fit data to the model with the train part and validate the model with the validate (test) part from the splitting in the
    # cross-validation
    history = model.fit(X_train_reshaped[train], y_train2[train],
                    validation_data=(X_train_reshaped[test], y_train2[test]),
                      batch_size= 64,
                      epochs= 30,
                      verbose= 1)
    # call the function that plot the results of the history with give it the history and the number of the fold
    plt_function_single_op(history, fold_no)
    # evaluate the model using the test features and test labels and print the results
    scores = model.evaluate(X_test_reshaped, y_test2, verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}%')
    # append in the array the results appeared in each iteration
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])

    # increment the fold number
    fold_no = fold_no + 1
```

Figure 17. Cross-validation with fitting and evaluate the model

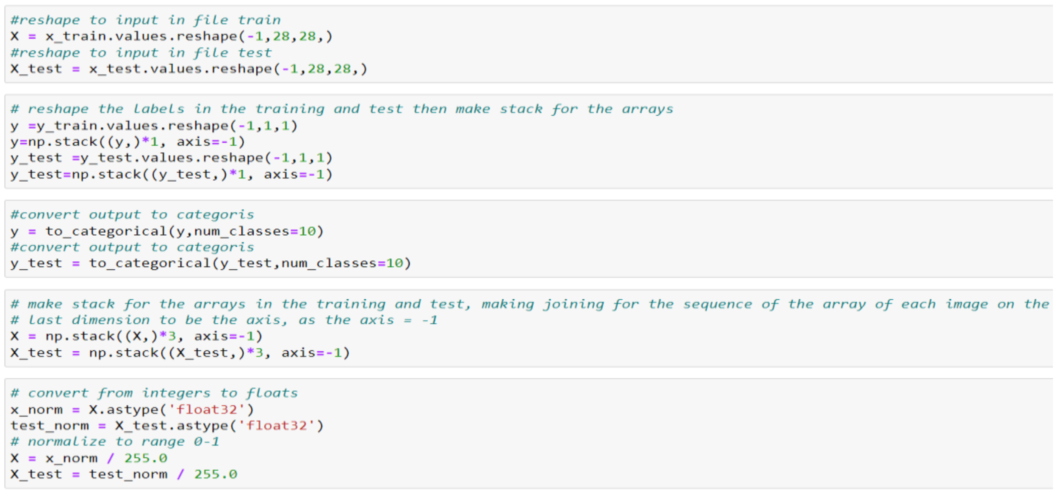
4.3. Transfer learning using ResNet pre-trained model

First before using this pre-trained model, we need to make some preprocessing on the inputs for the model. As this model use input images with 3 channels, and our dataset is 1 channel.

So, we reshaped the images and theirs labels, then making stacking for each image to make it as 3 channels. By making joining for the sequence of the array of each image on the last dimension to be the axis, as the axis = -1.

Then convert the labels into numerical labels as they are categorical.

Then for the images get the new arrays as float then make normalization for them to make the float numbers from 0.0 to 1.0.



```
#reshape to input in file train
X = x_train.values.reshape(-1,28,28,)
#reshape to input in file test
X_test = x_test.values.reshape(-1,28,28,)
```

```
# reshape the labels in the training and test then make stack for the arrays
y =y_train.values.reshape(-1,1,1)
y=np.stack((y,)*1, axis=-1)
y_test =y_test.values.reshape(-1,1,1)
y_test=np.stack((y_test,)*1, axis=-1)
```

```
#convert output to categoris
y = to_categorical(y,num_classes=10)
#convert output to categoris
y_test = to_categorical(y_test,num_classes=10)
```

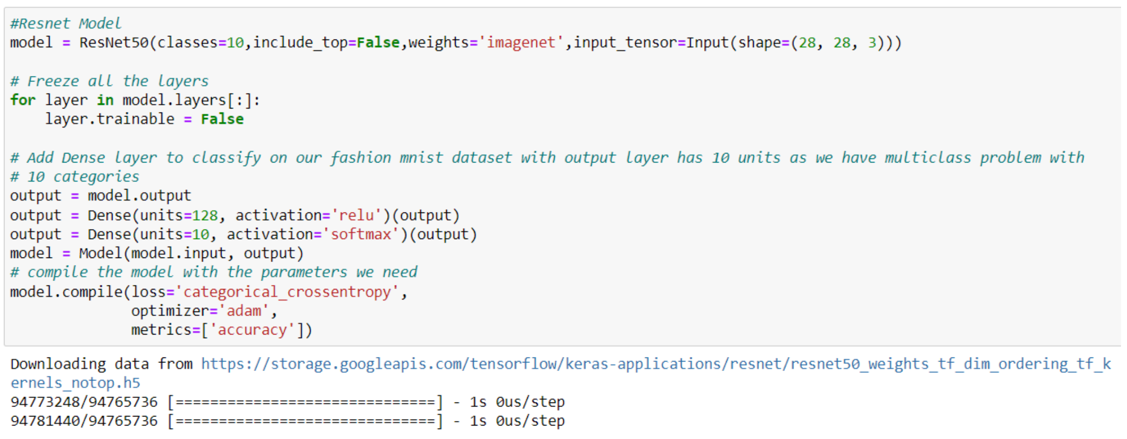
```
# make stack for the arrays in the training and test, making joining for the sequence of the array of each image on the
# last dimension to be the axis, as the axis = -1
X = np.stack((X,)*3, axis=-1)
X_test = np.stack((X_test,)*3, axis=-1)
```

```
# convert from integers to floats
x_norm = X.astype('float32')
test_norm = X_test.astype('float32')
# normalize to range 0-1
X = x_norm / 255.0
X_test = test_norm / 255.0
```

Figure 18. Preprocessing for the inputs and outputs for the ResNet model

After finishing the preprocessing for the inputs, and the outputs, we start in the model. By recall the ResNet50 model with its pre-trained weights with freezing all layers except the last fully connected layer. As we don’t want the model to train again on our dataset, we need only to train the last fully connected layer to be suitable with our problem.

So, in the fully connected layer, we made the first dense layer with 128 units and activation= ReLU. The second dense layer with 10 units as it is our output layer with Softmax activation.
In the model compilation, we will use Adam optimizer with the default learning rate, loss = 'categorical_crossentropy' as we have multi categories, and accuracy metric.



```
#Resnet Model
model = ResNet50(classes=10,include_top=False,weights='imagenet',input_tensor=Input(shape=(28, 28, 3)))

# Freeze all the layers
for layer in model.layers[:]:
    layer.trainable = False

# Add Dense layer to classify on our fashion mnist dataset with output layer has 10 units as we have multiclass problem with
# 10 categories
output = model.output
output = Dense(units=128, activation='relu')(output)
output = Dense(units=10, activation='softmax')(output)
model = Model(model.input, output)
# compile the model with the parameters we need
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_k
ernels_notop.h5
94773248/94765736 [==============================] - 1s 0us/step
94781440/94765736 [==============================] - 1s 0us/step
```

Figure 19. Transfer learning with ResNet model

Then we make fitting for the model on the training features and labels with number of epochs = 30, and batch size = 64.
Then evaluate the model using the test features after finishing the preprocessing on it as we made in the input (training) features.

4.4. Transfer leaning using VGG 16 pre-trained model

Before going into how we used the VGG 16 pre-trained model, first we need to resize the images as our dataset has size 28x28 and the least size for the VGG 16 model can work on is 48x48. This is due to construction and the layers in this model.

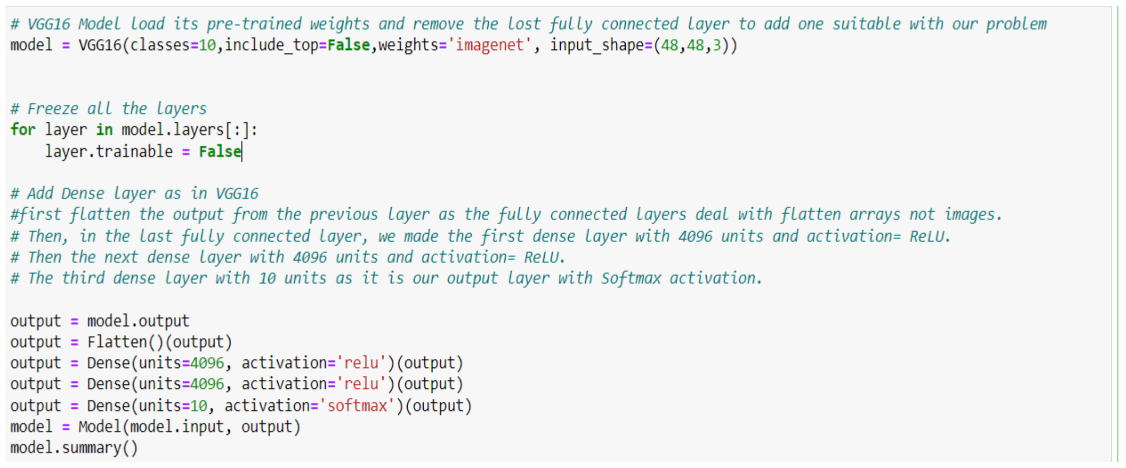
Now, we can start in the model by recall it with its pre-trained weights with freezing all layers except the last fully connected layer. As we don’t want the model to train again on our dataset, we need only to train the last fully connected layer to be suitable with our problem.

Before going through the last fully connected layer, we need to flatten the output from the previous layer as the fully connected layers deal with flatten arrays not images.

Then, in the last fully connected layer, we made the first dense layer with 4096 units and activation= ReLU. Then the next dense layer with 4096 units and activation= ReLU.

The third dense layer with 10 units as it is our output layer with Softmax activation.

In the model compilation, we will use Adam optimizer with the default learning rate, loss = 'categorical_crossentropy' as we have multi categories, and accuracy metric.



```
# VGG16 Model load its pre-trained weights and remove the lost fully connected layer to add one suitable with our problem
model = VGG16(classes=10,include_top=False,weights='imagenet', input_shape=(48,48,3))


# Freeze all the layers
for layer in model.layers[:]:
    layer.trainable = False

# Add Dense layer as in VGG16
#first flatten the output from the previous layer as the fully connected layers deal with flatten arrays not images.
# Then, in the last fully connected layer, we made the first dense layer with 4096 units and activation= ReLU.
# Then the next dense layer with 4096 units and activation= ReLU.
# The third dense layer with 10 units as it is our output layer with Softmax activation.

output = model.output
output = Flatten()(output)
output = Dense(units=4096, activation='relu')(output)
output = Dense(units=4096, activation='relu')(output)
output = Dense(units=10, activation='softmax')(output)
model = Model(model.input, output)
model.summary()
```

Figure 20. Transfer learning with VGG 16 model

Then we make fitting for the model on the training features and labels with number of epochs = 30, and batch size = 64.

Then evaluate the model using the test features after finishing the preprocessing on it as we made in the input (training) features.

5. Results

5.1. For LeNet 5 model

After tuning the learning rate, and the number of units in the first dense layer in our LeNet 5 model, we got the following:

The hyper-parameter search is done. The optimal number of units in the first dense layer in the fully connected layer is 248 and the optimal learning rate for the Adam optimizer is 0.001.

For our LeNet 5 model after training it with the tuned hyper-parameters that we got from the tuning process with using number of epochs = 30 and batch size = 64, then evaluate the models with the test features and labels.

The loss plot (convergence curve) and accuracy plot during the training for each fold:

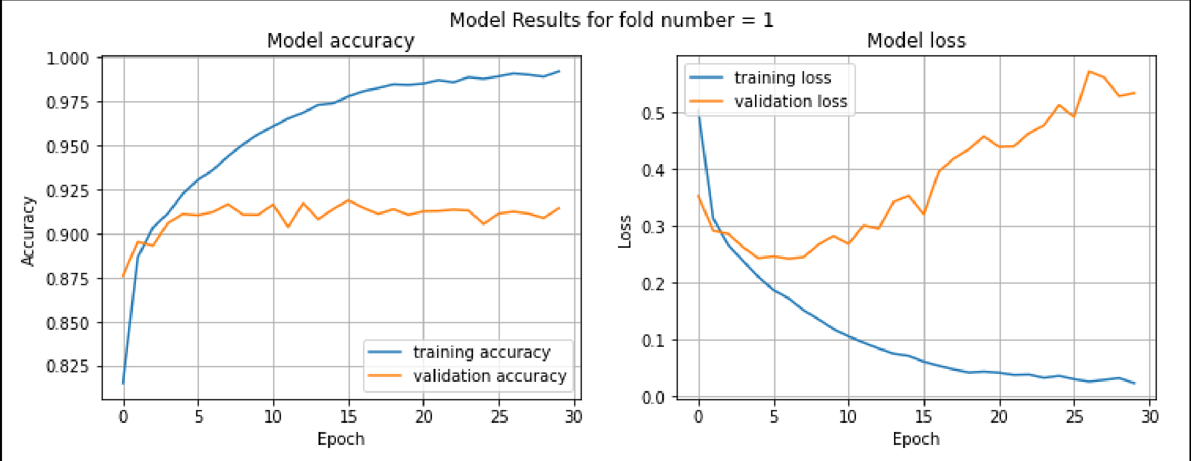


Figure 21

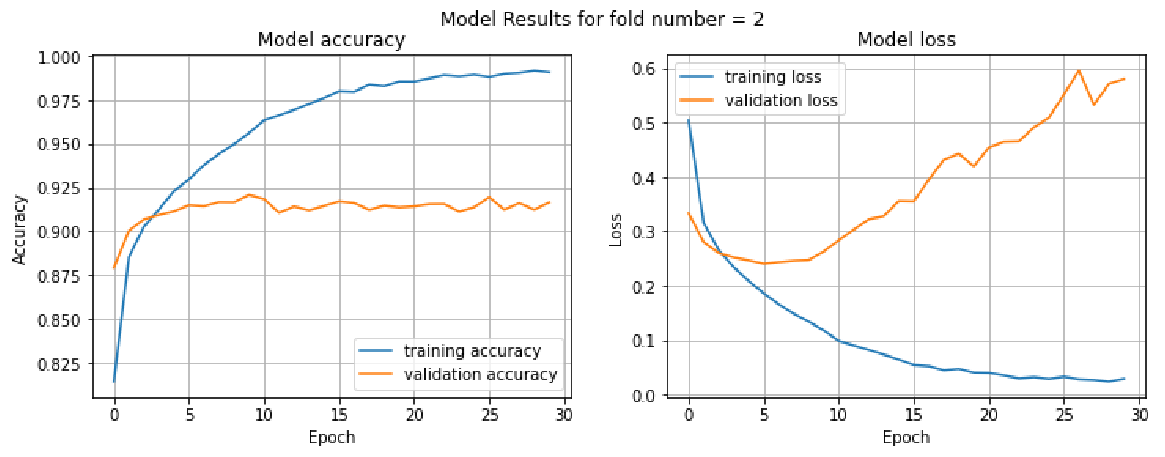


Figure 22

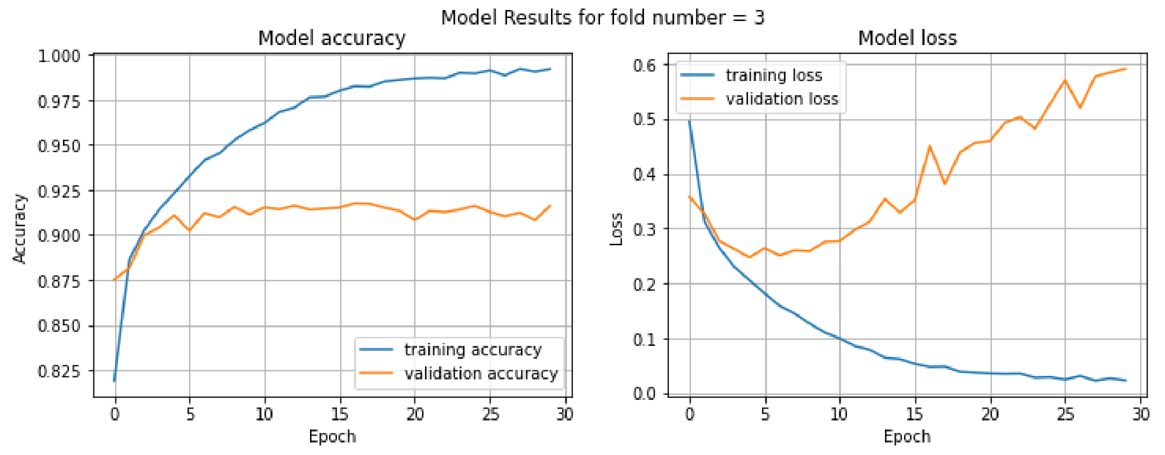


Figure 23

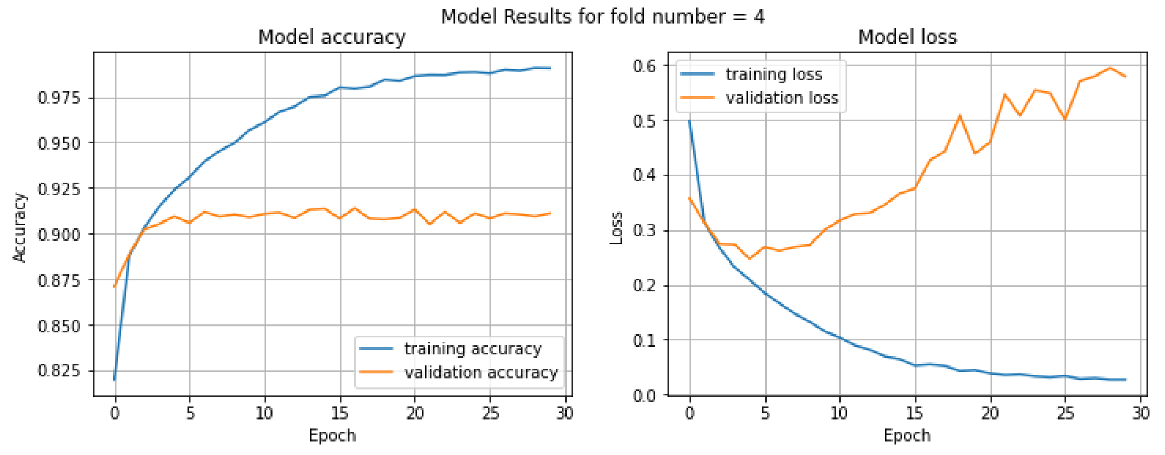


Figure 24

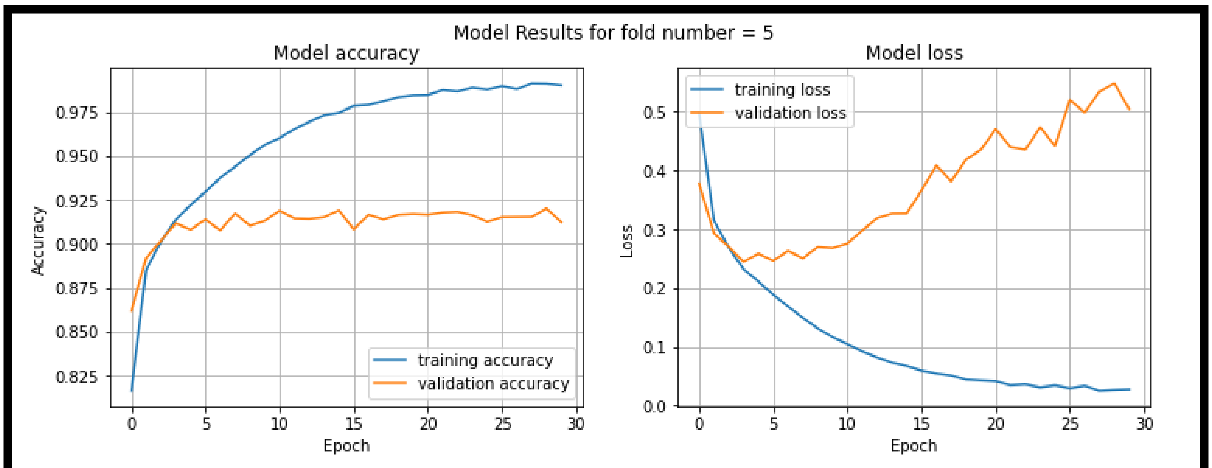


Figure 25

Figures 21, 22, 23, 24, 25. Plots of the results during training phase in LeNet 5 model

Then we got those results at the evaluation during the testing phase across the cross-validation process as we used 5 folds in it:

Fold number	Loss	Accuracy
Fold 1	0.50633	91.64%
Fold 2	0.52324	91.97%
Fold 3	0.52211	92.22%
Fold 4	0.52503	91.68%
Fold 5	0.49211	91.46%

Average scores over all folds:

Accuracy: 91.798%

Loss: 0.51376

From the above results during the training, it seems that the models started to overfit the training data after almost epoch 7 as the loss started to be higher in the validation than the loss in the training. For the accuracy in the validation also it started to be almost fixed in the same number but the accuracy in the training is increasing with each epoch, so this approves that there is overfitting on the training data.

In the testing phase it didn't give high values in the accuracy and low values in the loss, as the results almost close to the results we got from the validation of the model.
But to evaluate the LeNet 5 model right, let's see the results from the other models that we used the pre-trained and didn't train the model again on our datasets.

5.2. For ResNet model

In the training phase, we got in the last epoch → loss: 0.4583 - accuracy: 0.8254 - val_loss: 0.5017 - val_accuracy: 0.8125.
In the testing phase, we got the following:
Loss: 0.48174 - accuracy: 0.8198%.
So, this means there is no overfitting happen in the training phase, as the results in the testing phase are very close to the results in the training phase for the training data and validation data.
For the accuracy and loss plots for this model in the training phase, it approves that there is no overfitting happened until all epochs finished. As we said the results are close to each other, and you can see it clear on the plots.

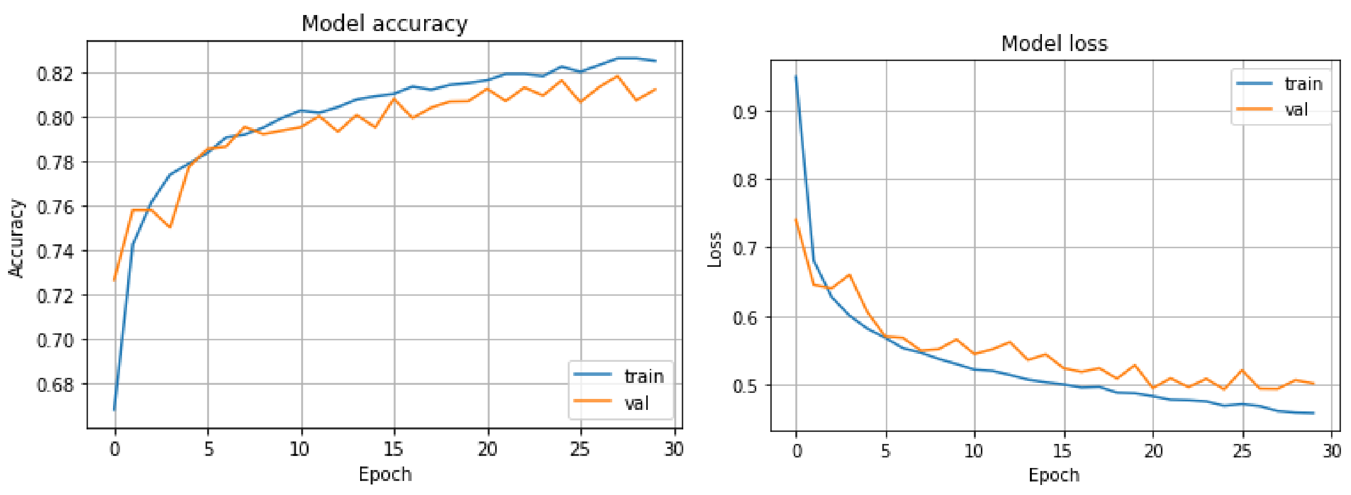


Figure 26. Accuracy & loss in the training phase in ResNet model

But on the other hand, the results of this model are less than the results we got from the trained LeNet 5 model.

5.3. For VGG 16 model

In the training phase, we got in the last epoch → loss: 0.0473 - accuracy: 0.9838 - val_loss: 0.8164 - val_accuracy: 0.8768.

In the testing phase, we got the following:

Loss: 0.7606 - accuracy: 0.8828%.

So, this means there is overfitting happened in the training phase, as the results in the testing phase are very close to the results in the training phase for the validation set but they are less than the accuracy on the training data. So, this means the model learnt very well on the training data but on the unseen data it didn't have correct predictions in many test samples. This model may learnt very well due to its construction as it is very good and it trained from the beginning of categorical image data so we think this helped here to make it be very good in the training phase. But the difference is not too big, and we can see that the model started in the overfitting from the epoch number 5, we can see this clear in the figure 27.

For the accuracy and loss plots for this model in the training phase, it approves that there is overfitting happened after the epoch number 5.

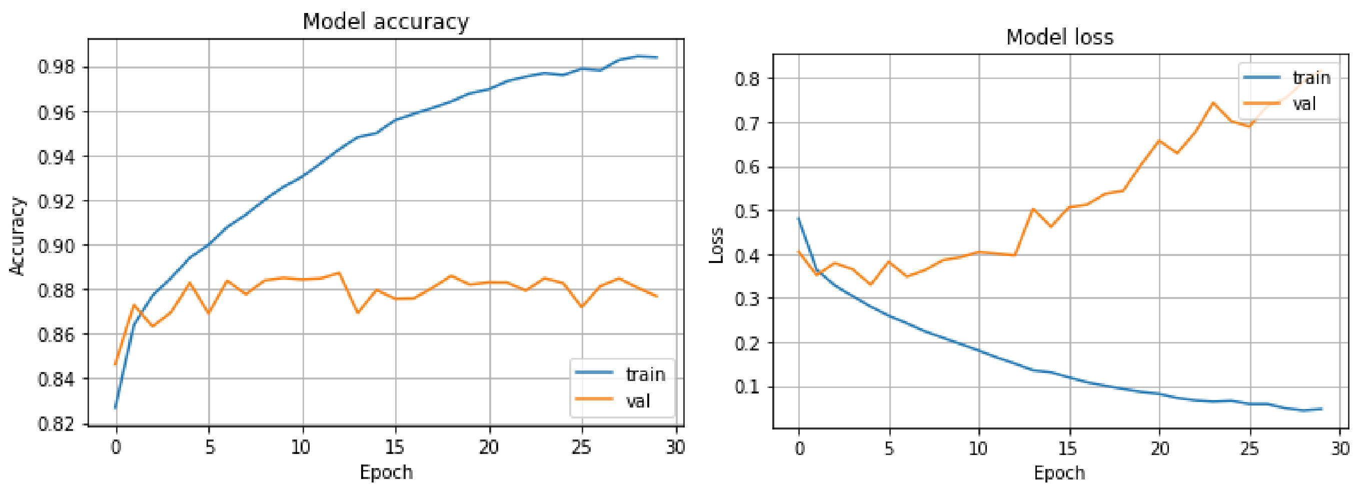


Figure 27. Accuracy & loss in the training phase in VGG 16 model

In the VGG model, the results in the testing phase are still less than the results we got from the LeNet 5 model, but they are higher than the results that we got from the ResNet model.

5.4. Comment on the LeNet results

So, for the comment on why you think LeNet-5 further improves the accuracy if any at all.

We think it improved the accuracy as with the LeNet 5 model we got the best results on the test dataset. This might be done because we made the layers of the model, and tuned many different hyper-parameters for the model. Besides, we made the model learn from our dataset after cleaning and preprocessing it, to make it suitable for the model.

So, as the model learnt well from the training dataset, it gave good accuracy on the test dataset.

6. Conclusion

Neural networks need very large datasets, as our dataset consisting of a training set of 60,000 examples and a test set of 10,000 examples. So, this large dataset helped to get good results from the models that we used.

From the results we got, we can feel how the training phase is important for any model to get good results. As to get good results we need to train the model on our training dataset to make the model learn and get good values for it through the epochs through the forward and backward processes. Besides, choosing optimal values for the hyper-parameters help in making the model learn more from the dataset, and get better results.

Using transfer learning is not good on all cases as the dataset we are working on may be very different from the data that the model trained and learned from, so this causes getting bad results even if the dataset is very big. As in the transfer learning we change only the last fully connected layer, and use the same parameters and weights for the model that it got from the previous training on its dataset. So, changing the last fully connected layer doesn’t have a big effect on the model, we change it just to make it suitable for our classification or regression problem, and it learn a little bit from the dataset that we work on. We saw the difference in this point when we built the LeNet 5 model, and made it learn from the dataset, and used the ResNet and VGG 16 as pre-trained models.

7. References

- https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-keras.md
- https://www.kaggle.com/code/curiousprogrammer/lenet-5-cnn-with-keras-99-48/notebook
- https://www.tensorflow.org/tutorials/keras/keras_tuner
- https://www.kaggle.com/code/viratkothari/image-classification-of-mnist-using-vgg16/notebook
- https://github.com/keras-team/keras/issues/4465