Fashion Classification Using Convolutional Neural Network Model

SUBMITTING TO:

JASPREET KAUR MA'AM

NAME: CHINTHALA MAHESH

REG.NO: 11907444

SECTION: KM053

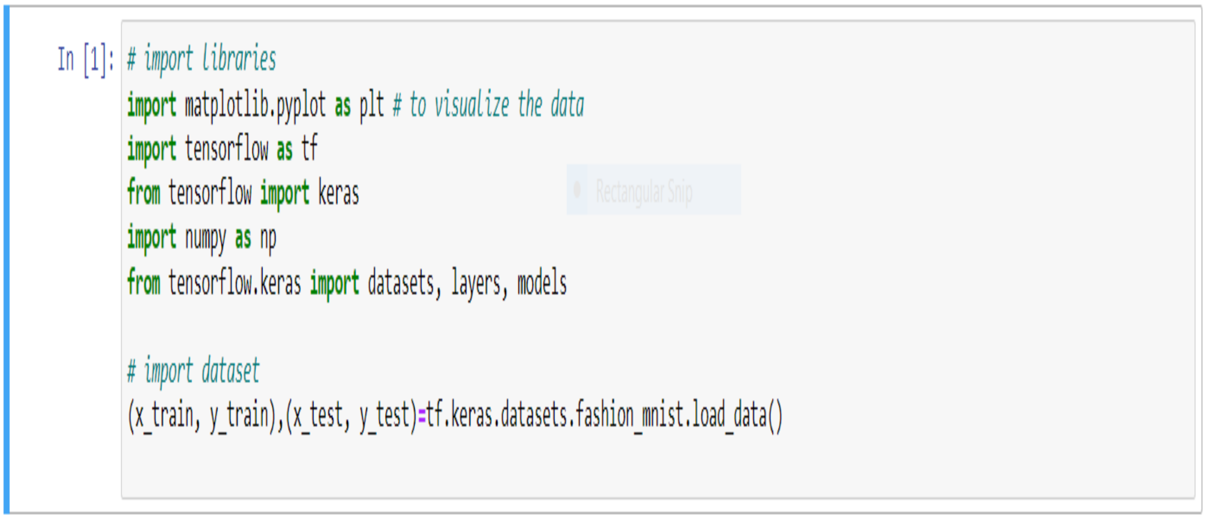
ROLL.NO: A14

Colab Link: https://github.com/GudladhanaHarshith/FakeNewsClassification

INTRODUCTION:

1. Online fashion market is constantly growing, and an algorithm capable of identifying garments can help companies in the clothing sales sector to understand the profile of potential buyers and focus on sales targeting specific niches, as well as developing campaigns based on the taste of customers and improve user experience
2. Artificial Intelligence approaches able to understand and label humans' clothes are necessary, and can be used to improve sales, or better understanding users.
3. Convolutional Neural Network models have been shown efficiency in image c1assification.
4. The main goal of this project is to provide future research with better comparisons between classification.
5. In this project I had build a Convolutional Neural Network(CNN) Model and implemented it on Fashion-MNST dataset. Fashion-MNIST is a dataset made to help researchers finding models to classify this kind of product such as clothes etc.
6. This CNN Model identifies that the given image belongs to which class/category of clothing like shirt or top, trouser, dress or footwear like shoe, sandals etc.
7. In this project, I use lot of modules which are imported at the starting of the project. I performed data visualization, prepared data for training, prepared model, trained that model and finally assessed model’s performance by testing the model.

Imported Libraries



```
# import libraries
import matplotlib.pyplot as plt # to visualize the data
import tensorflow as tf
from tensorflow import keras
import numpy as np
from tensorflow.keras import datasets, layers, models

# import dataset
(x_train, y_train),(x_test, y_test)=tf.keras.datasets.fashion_mnist.load_data()
```

About Libraries and Data Set that I had Imported and used in this Project :

TensorFlow:

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML, and gives developers the ability to easily build and deploy ML-powered applications.

TensorFlow provides a collection of workflows with intuitive, high-level APIs for both beginners and experts to create machine learning models in numerous languages. Developers have the option to deploy models on a number of platforms such as on servers, in the cloud, on mobile and edge devices, in browsers, and on many other JavaScript platforms.

Keras:

- Keras is a high-level neural networks library, written in Python and capable of running on top of either Tensor Flow or Theano.
- It was developed with a focus on enabling fast experimentation.
- The core data structure of Keras is a **model**, a way to organize layers.

- The main type of model is the Sequential model, a linear stack of layers.
- For more complex architectures, use the Keras functional API.

Benefits Of Keras:

- Keras leverages various optimization techniques to make high level neural network API easier and more performant.
- It supports the following features –
  - Consistent, simple and extensible API.
  - Minimal structure - easy to achieve the result without any frills.
  - It supports multiple platforms and backends.
  - It is user friendly framework which runs on both CPU and GPU.
  - Highly scalability of computation.

Matplotlib:

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible.

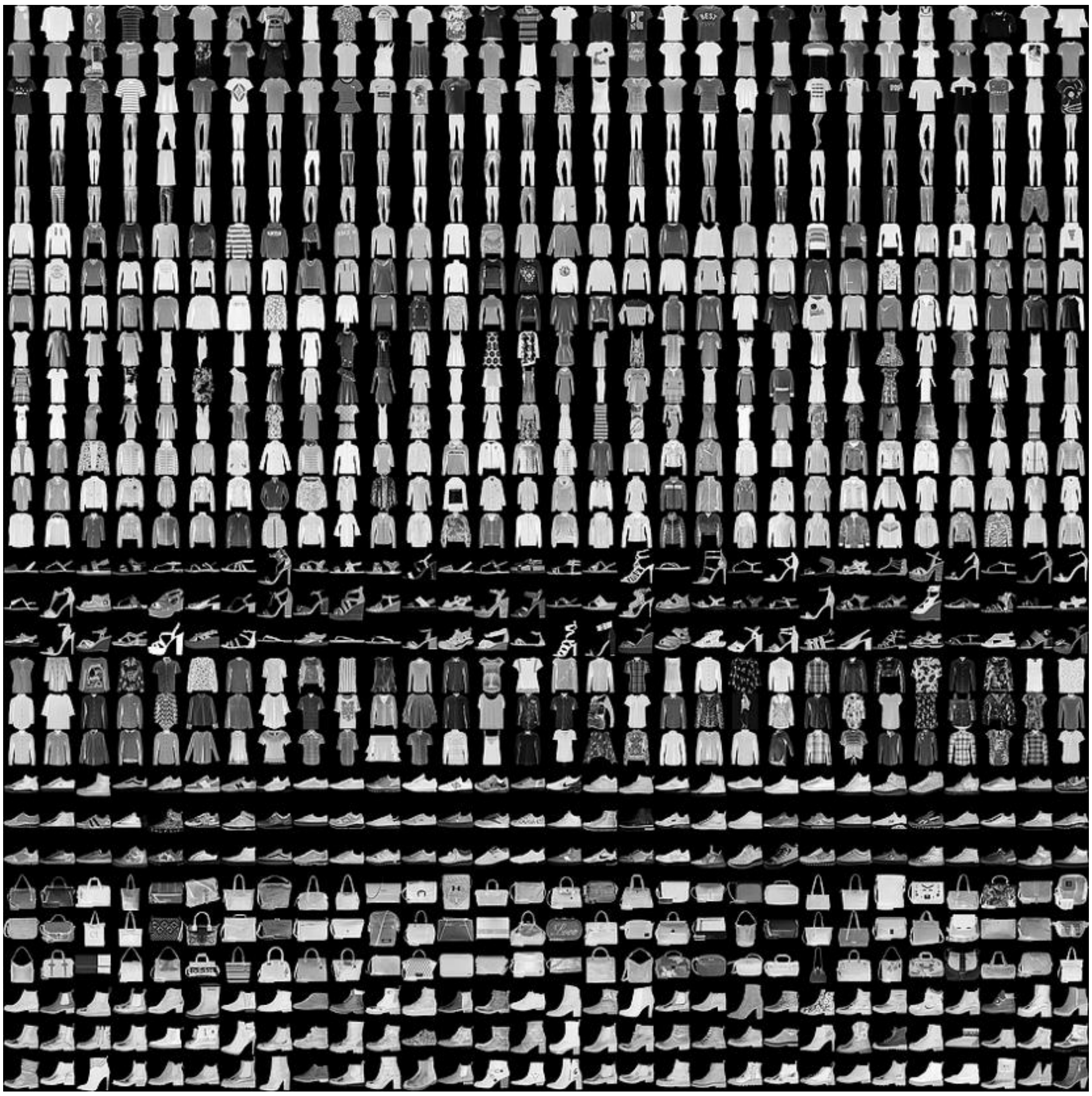
- Create publication quality plots.
- Make interactive figures that can zoom, pan, update.
- Customize visual style and layout.
- Export to many file formats.
- Embed in JupyterLab and Graphical User Interfaces.
- Use a rich array of third-party packages built on Matplotlib.

Data Set Description:

In this project I had used Fashion MNST dataset.

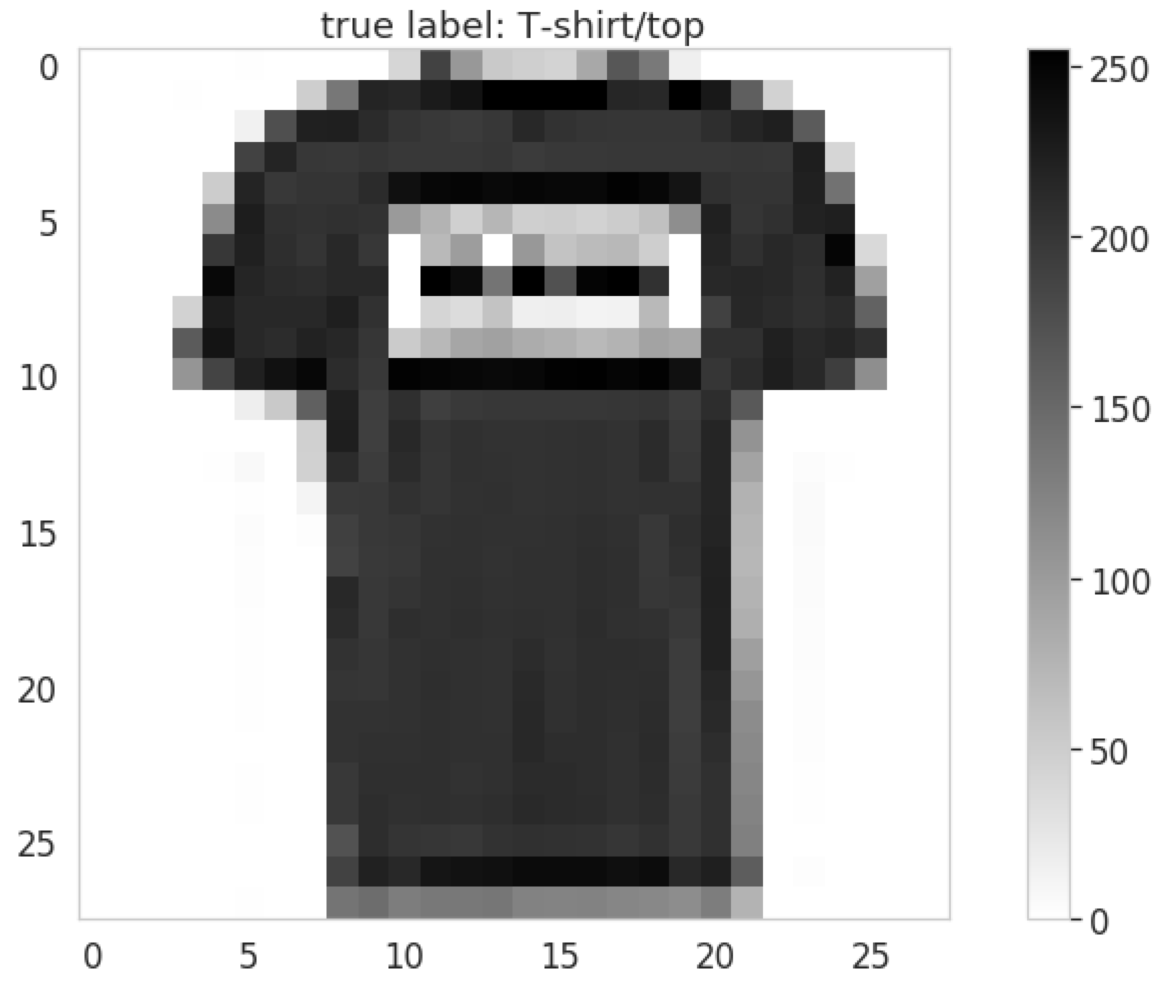
Fashion-MNIST is a dataset of Zalando's article images — consisting of a training set of _60,000_ examples and a test set of _10,000_ examples. Each example is a _28x28_ grayscale image, associated with a label from _10_ classes. We intend Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

Here is a sample of the images:



Exploration:

The product images are grayscale, 28x28 pixels and look something like this:

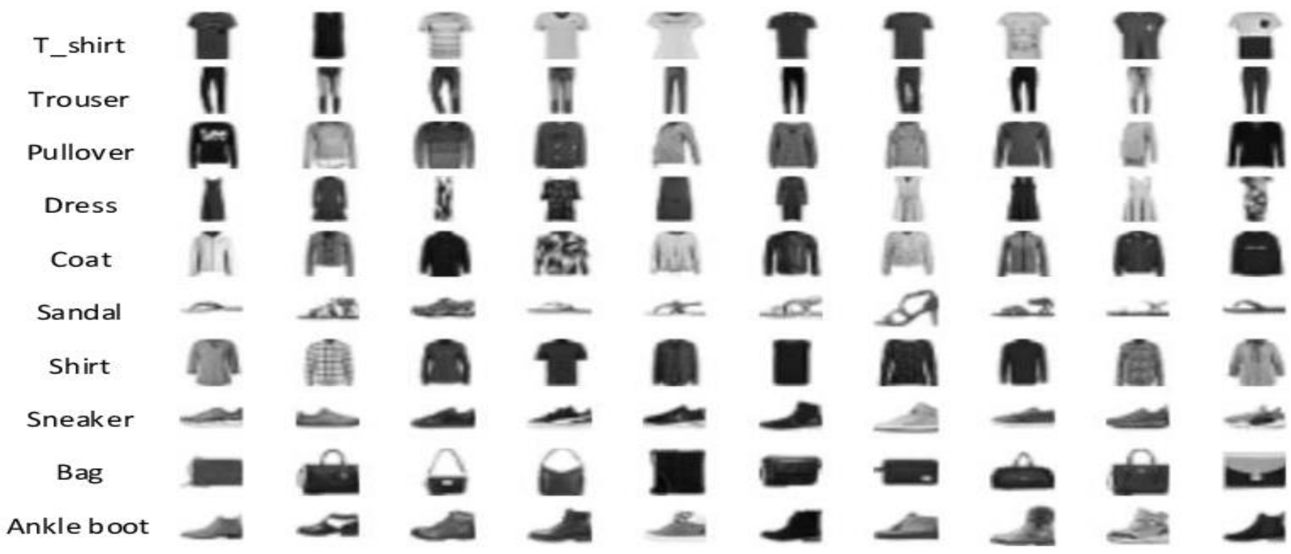


Dataset sample consists of 28*28 grayscale images associated with a label from 10 classes.

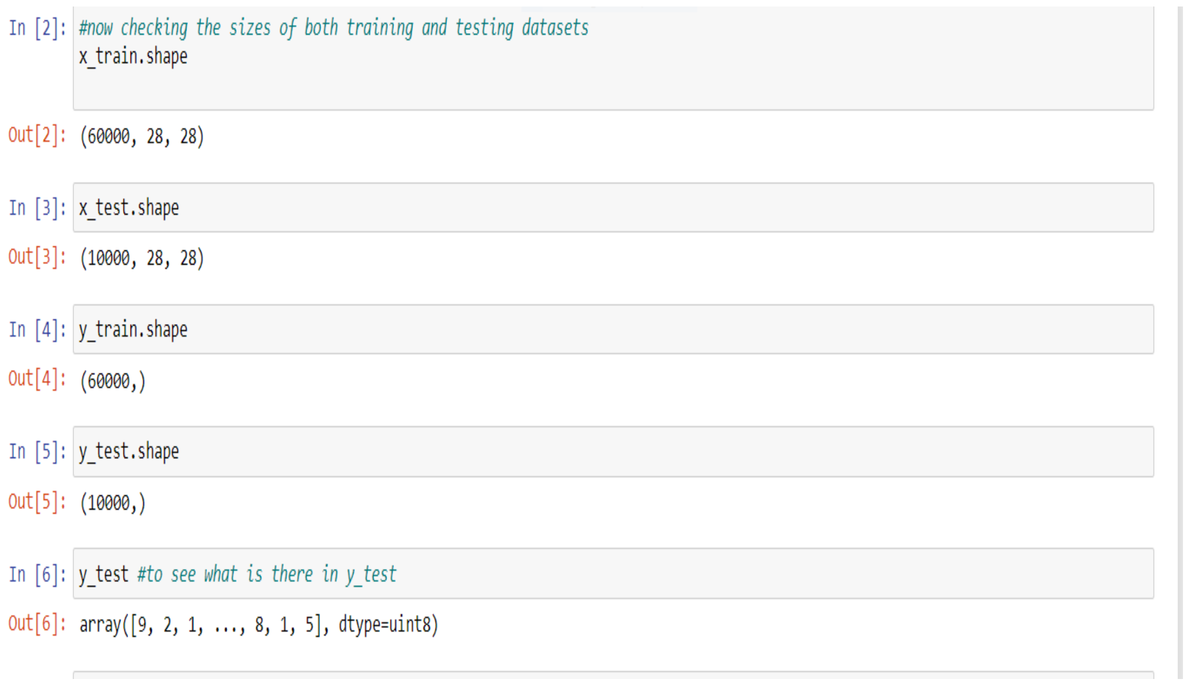
Each image 28*28(height*width) for a total 784 pixels.

Each pixel has single value associated wit it, indicating lightness/darknes of that pixel, if higher the value means darker and lower the value means lighter, value between 0 and 255.

We have 10 classes of possible fashion products:



Observation of Data Sets:

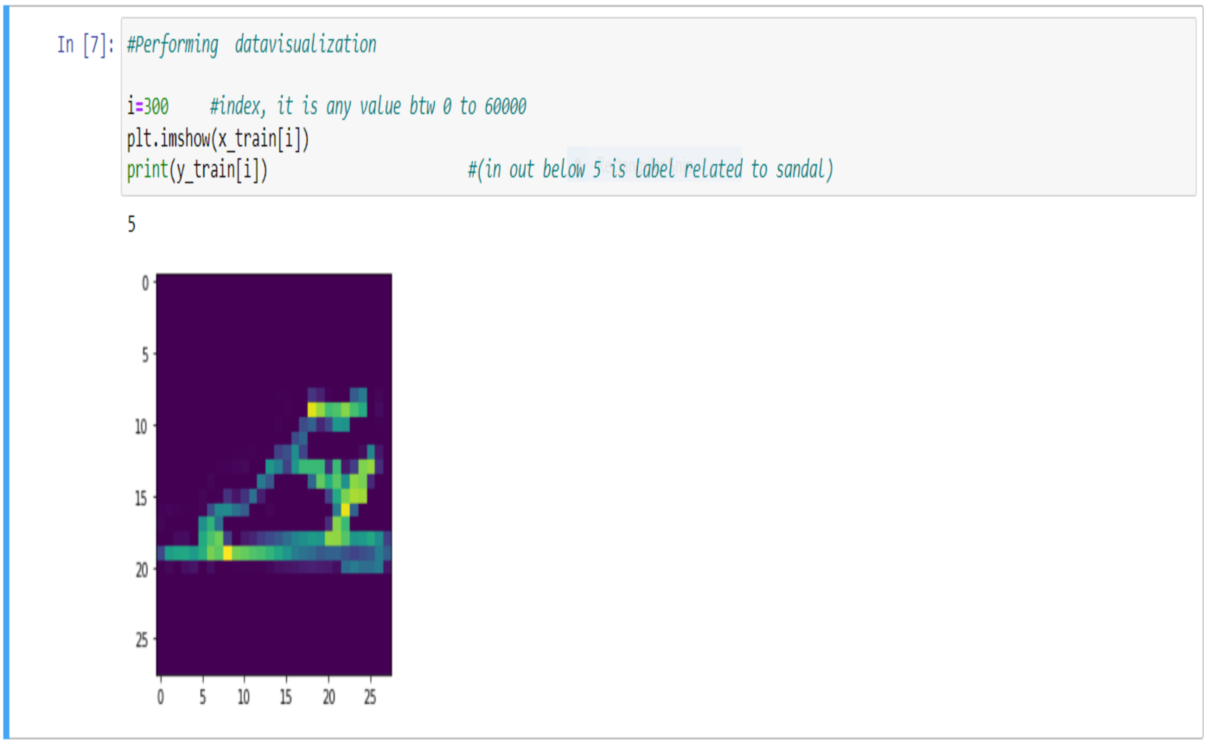


```
In [2]: #now checking the sizes of both training and testing datasets
        x_train.shape

Out[2]: (60000, 28, 28)

In [3]: x_test.shape

Out[3]: (10000, 28, 28)

In [4]: y_train.shape

Out[4]: (60000,)

In [5]: y_test.shape

Out[5]: (10000,)

In [6]: y_test #to see what is there in y_test

Out[6]: array([9, 2, 1, ..., 8, 1, 5], dtype=uint8)
```

We can observe from above that our MNST dataset has 60000 training images with dimensions 28*28 and 10000 testing images.

Performing Data Visualization:

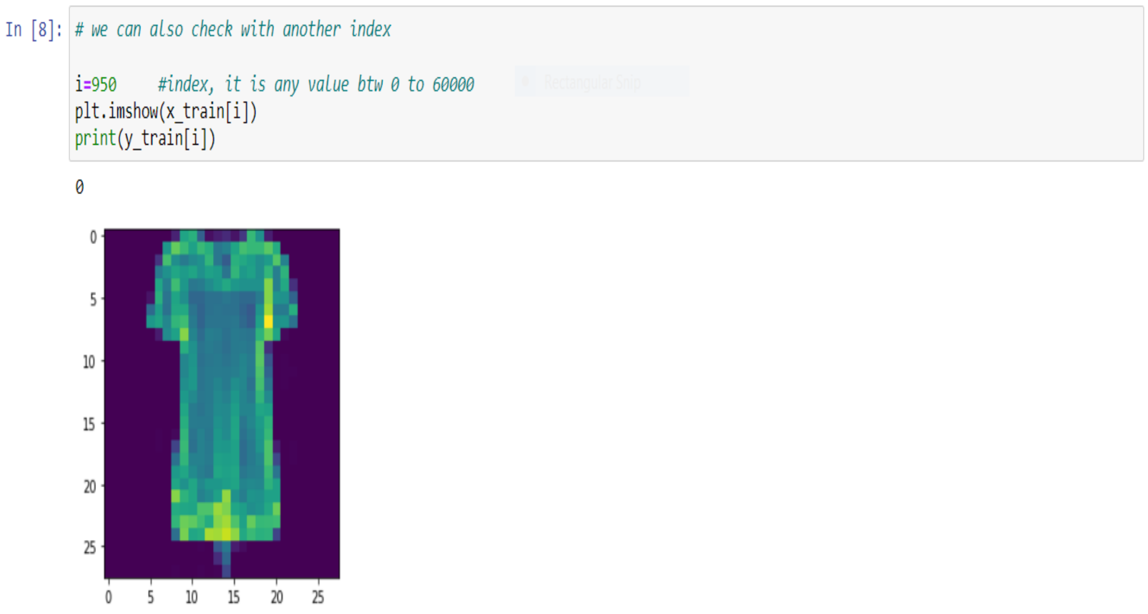
In this step we are going to see the image which is there at ith index and label of that image.



```
In [7]: #Performing  datavisualization

        i=300     #index, it is any value btw 0 to 60000
        plt.imshow(x_train[i])
        print(y_train[i])                    #(in out below 5 is label related to sandal)

5
```

From the above picture we can observe the image which is there at 300 th index (index any value between 0 to 60000). So at 300th index there is sandal and we can see its label. In above example sandal belongs to 5th category.

We can see another image which is there at another index.



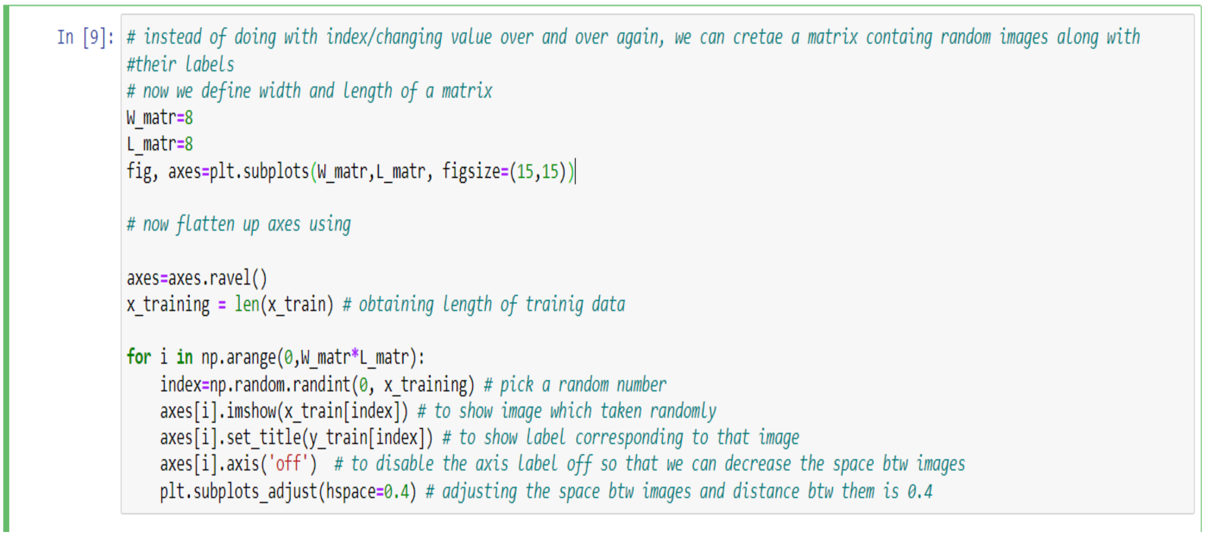
```
In [8]: # we can also check with another index

        i=950     #index, it is any value btw 0 to 60000
        plt.imshow(x_train[i])
        print(y_train[i])

0
```

From the above picture we can observe the image which is there at 950 th index (index any value between 0 to 60000). So at 950th index there is sandal and we can see its label. In above example sandal belongs to 0th category.

Instead of doing with index/changing value over and over again, we can cretae a matrix containing random images along with their labels

now we define width and length of a matrix

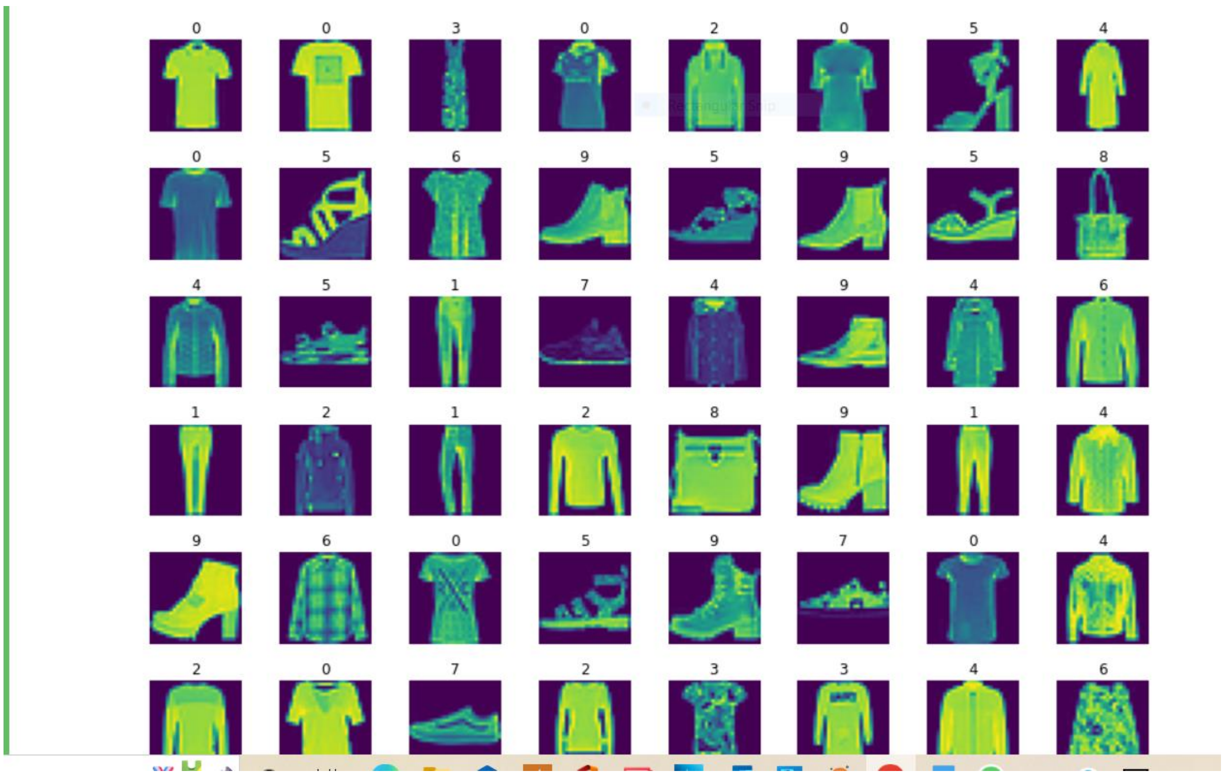
So now we are going to subplot our image set. Which take random images.

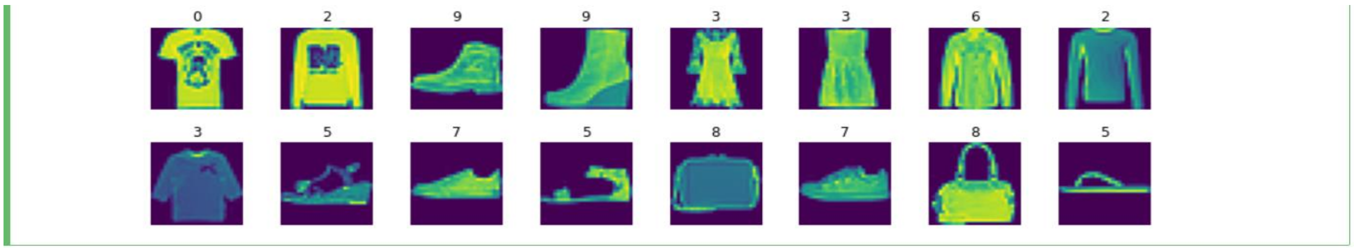


```
In [9]: # instead of doing with index/changing value over and over again, we can cretae a matrix containg random images along with
        #their labels
        # now we define width and length of a matrix
        W_matr=8
        L_matr=8
        fig, axes=plt.subplots(W_matr,L_matr, figsize=(15,15))

        # now flatten up axes using

        axes=axes.ravel()
        x_training = len(x_train) # obtaining length of trainig data

        for i in np.arange(0,W_matr*L_matr):
            index=np.random.randint(0, x_training) # pick a random number
            axes[i].imshow(x_train[index]) # to show image which taken randomly
            axes[i].set_title(y_train[index]) # to show label corresponding to that image
            axes[i].axis('off')  # to disable the axis label off so that we can decrease the space btw images
            plt.subplots_adjust(hspace=0.4) # adjusting the space btw images and distance btw them is 0.4
```

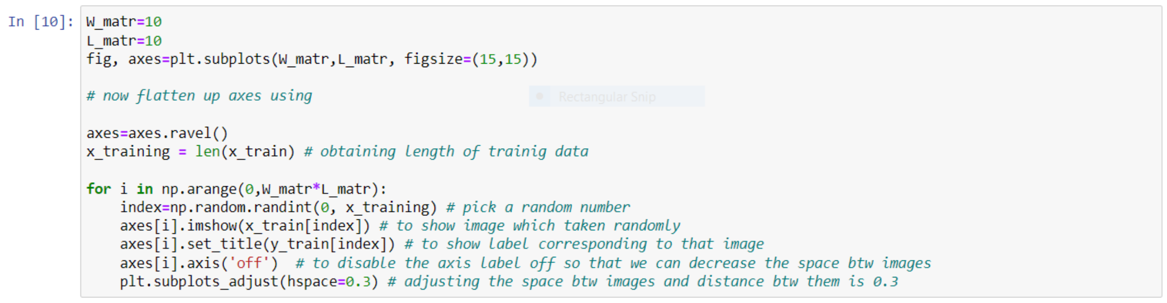
Here we have to declare dimensions of then matrix/grid and figure size. I had given size as 8*8 and figure size is 15*15.

Axis label should be off when we take more images so that the space between images would be decreased and we can see all images.



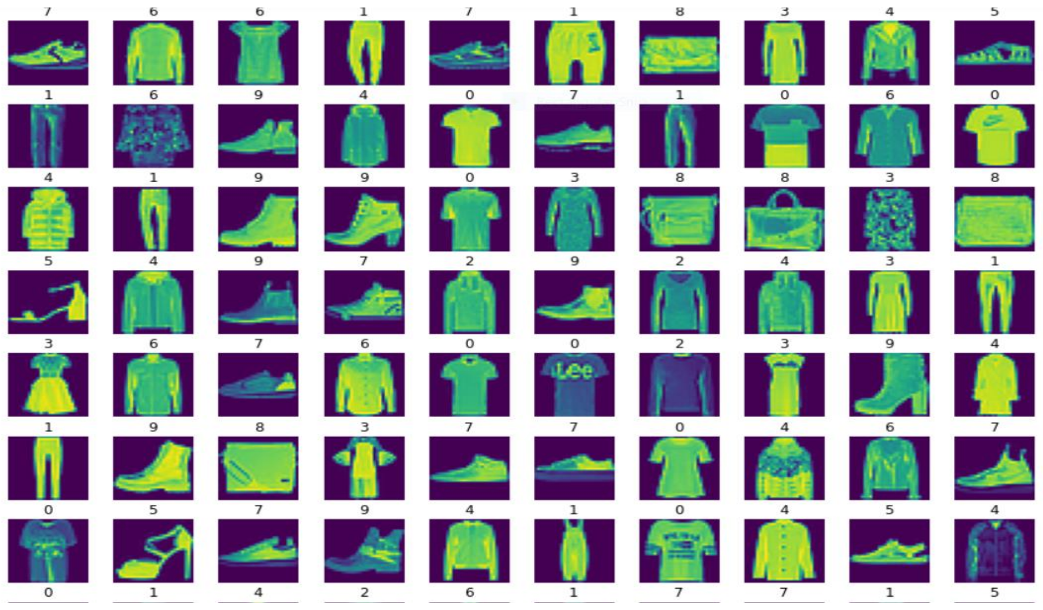


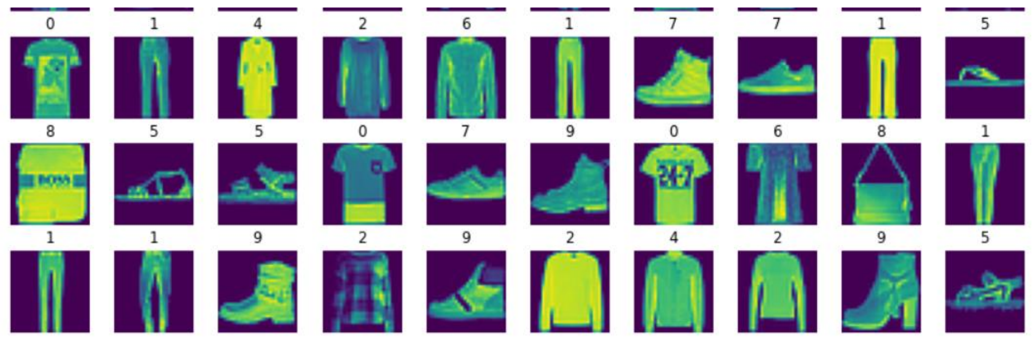
Above are the 8*8 total of 64 images. Value above image indicates the label or category the images would be related.



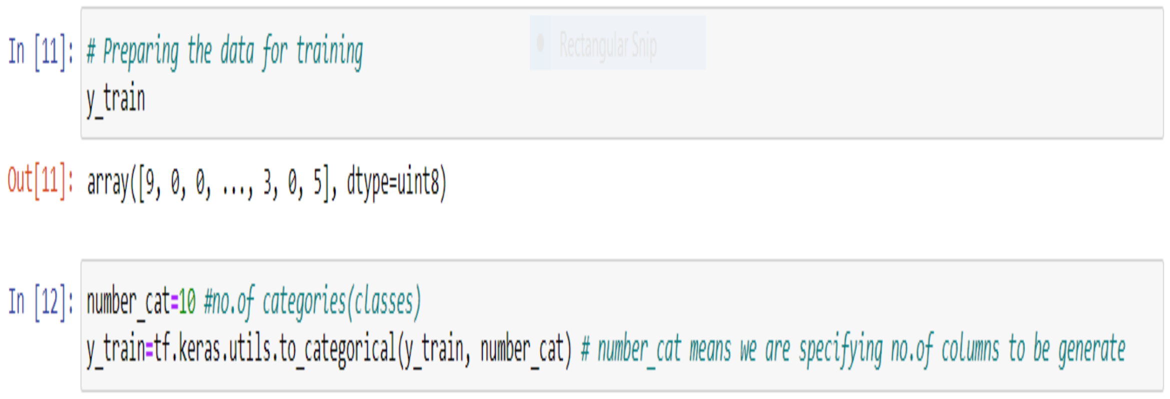
```
In [10]: W_matr=10
         L_matr=10
         fig, axes=plt.subplots(W_matr,L_matr, figsize=(15,15))

         # now flatten up axes using

         axes=axes.ravel()
         x_training = len(x_train) # obtaining length of trainig data

         for i in np.arange(0,W_matr*L_matr):
             index=np.random.randint(0, x_training) # pick a random number
             axes[i].imshow(x_train[index]) # to show image which taken randomly
             axes[i].set_title(y_train[index]) # to show label corresponding to that image
             axes[i].axis('off')  # to disable the axis label off so that we can decrease the space btw images
             plt.subplots_adjust(hspace=0.3) # adjusting the space btw images and distance btw them is 0.3
```

This is the code for another set of images with imageset containing 10*10 total of 100 images with dimensions 15*15.





Now we have to prepare the data for training and testing.

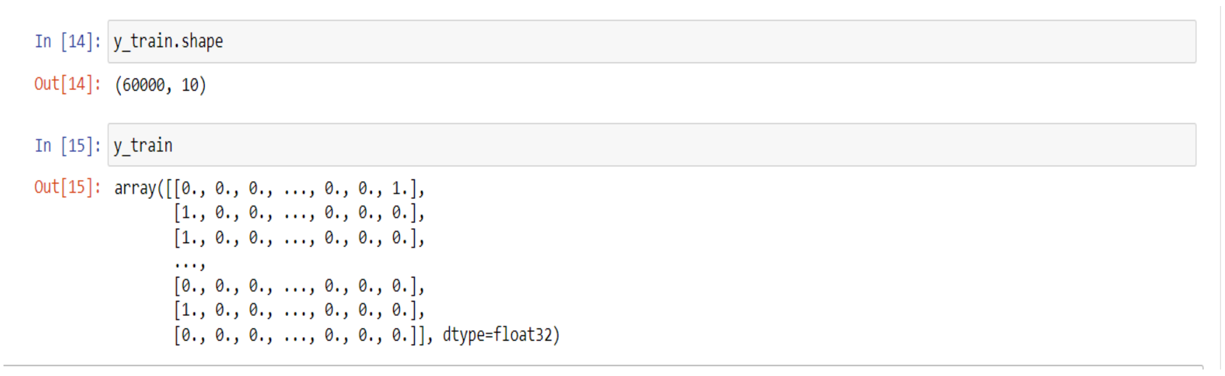


```
In [11]: # Preparing the data for training
         y_train

Out[11]: array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)

In [12]: number_cat=10 #no.of categories(classes)
         y_train=tf.keras.utils.to_categorical(y_train, number_cat) # number_cat means we are specifying no.of columns to be generate
```

We observed that y_train has array which contains 10 values from 0 to 9.

Number_cat is no.of categories or classes dataset contains.

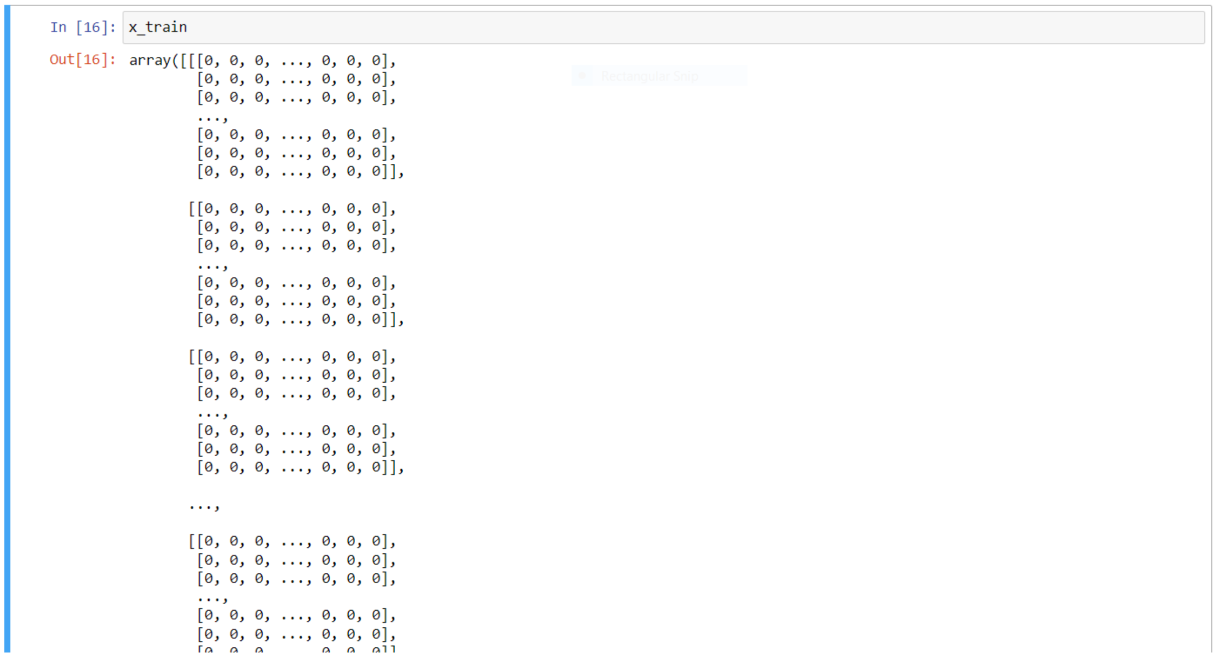
Here it is going to convert y_train (bunch of integers) into a matrix contains 0's and 1's in which no.of columns is essentially no .of categories or classes in our data.



```
In [14]: y_train.shape
Out[14]: (60000, 10)

In [15]: y_train
Out[15]: array([[0., 0., 0., ..., 0., 0., 1.],
                [1., 0., 0., ..., 0., 0., 0.],
                [1., 0., 0., ..., 0., 0., 0.],
                ...,
                [0., 0., 0., ..., 0., 0., 0.],
                [1., 0., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

# 60000 by 10 columns. Means dimension of y aixs is 10.

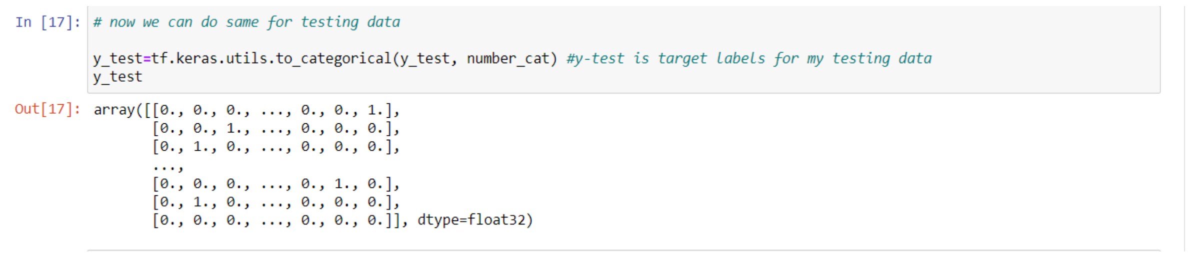
From out put of y_train we can observe that dimension of output now we have is 10 outputs. Now if the item is like dress it would be there at 1’s place and rest would 0.



```
In [16]: x_train
Out[16]: array([[[0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 ...,
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0]],

                [[0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 ...,
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0]],

                [[0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 ...,
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0]],

                ...,

                [[0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 ...,
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
```

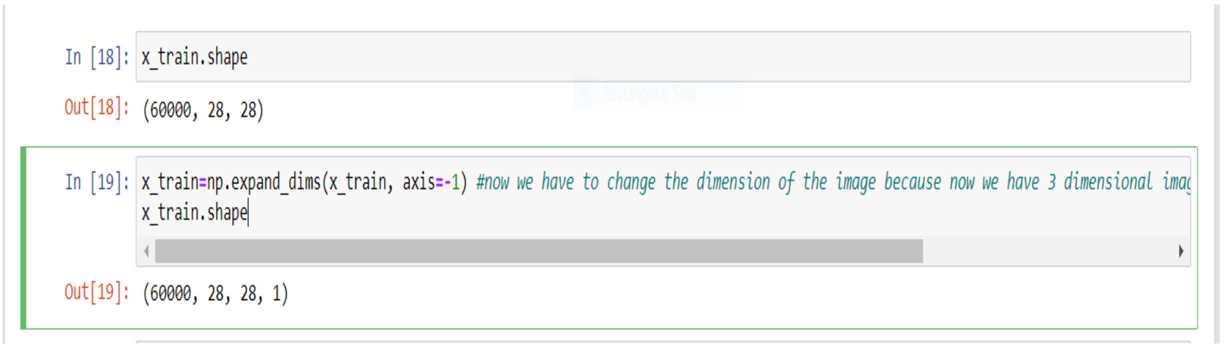
Above all are the actual values of the pixel for inputs means images in x_train.

Now we can do same for testing.



```
In [17]: # now we can do same for testing data

         y_test=tf.keras.utils.to_categorical(y_test, number_cat) #y-test is target labels for my testing data
         y_test

Out[17]: array([[0., 0., 0., ..., 0., 0., 1.],
                [0., 0., 1., ..., 0., 0., 0.],
                [0., 1., 0., ..., 0., 0., 0.],
                ...,
                [0., 0., 0., ..., 0., 1., 0.],
                [0., 1., 0., ..., 0., 0., 0.],
                [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Now we have to change the dimension of the image because now we have 3 dimensional image but our model takes 4 dimensional image.

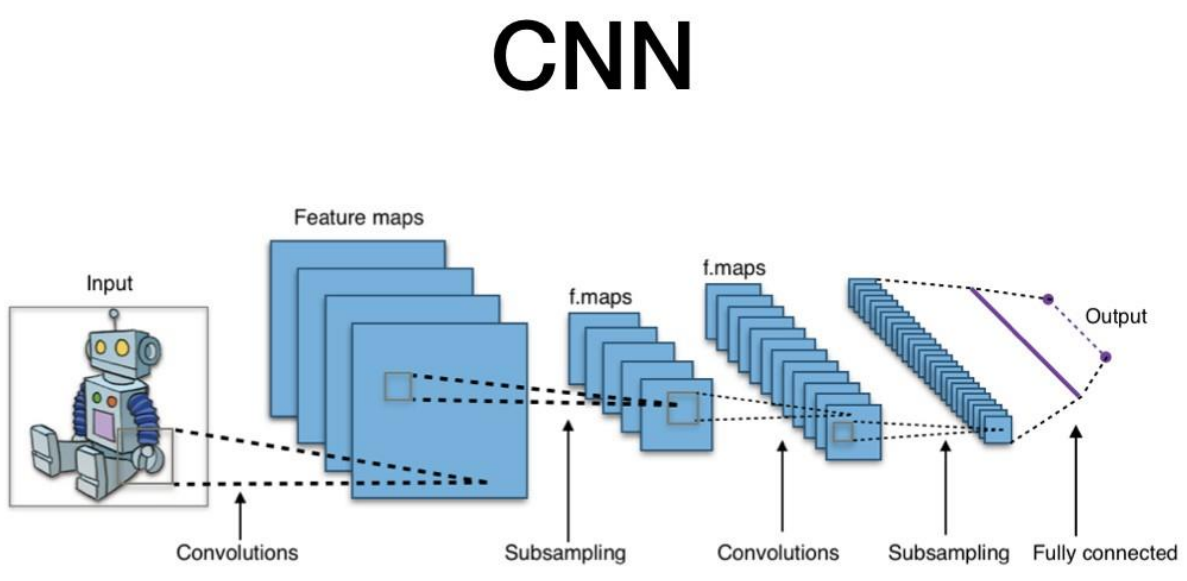


```
In [18]: x_train.shape

Out[18]: (60000, 28, 28)

In [19]: x_train=np.expand_dims(x_train, axis=-1) #now we have to change the dimension of the image because now we have 3 dimensional imag
         x_train.shape

Out[19]: (60000, 28, 28, 1)
```

From the above image we can observe that dimension of the image changed to 4 dimensional.

BUILDING A CNN MODEL.

Convolutional Neural Network.



Convolution Neural Network

Source: Wikipedia

The Convolutional Neural Network (CNN or ConvNet) is a subtype of Neural Networks that is mainly used for applications in image and speech recognition. Its built-in convolutional layer reduces the high dimensionality of images without losing its information. That is why CNNs are especially suited for this use case.

Image Processing Problems

If we want to use a fully-connected neural network for image processing, we quickly discover that it does not scale very well.

For the computer, an image in RGB notation is the summary of three different matrices. For each pixel of the image, it describes what color that pixel displays. We do this by defining the red component in the first matrix, the green component in the second, and then the blue component in the last. So for an image with the size 3 on 3 pixels, we get three different 3x3 matrices.

To process an image, we enter each pixel as input into the network. So for an image of size 200x200x3 (i.e. 200 pixels on 200 pixels with 3 color channels, e.g. red, green and blue) we have to provide 200 * 200 * 3= 120,000 input neurons. Then each matrix has a size of 200 by 200 pixels, so 200 * 200 entries in total. This matrix then finally exists three times, each for red, blue, and green. The problem then arises in the first hidden layer, because each of the neurons there would have 120,000 weights from the input layer. This means the number of parameters would increase very quickly as we increase the number of neurons in the Hidden Layer.

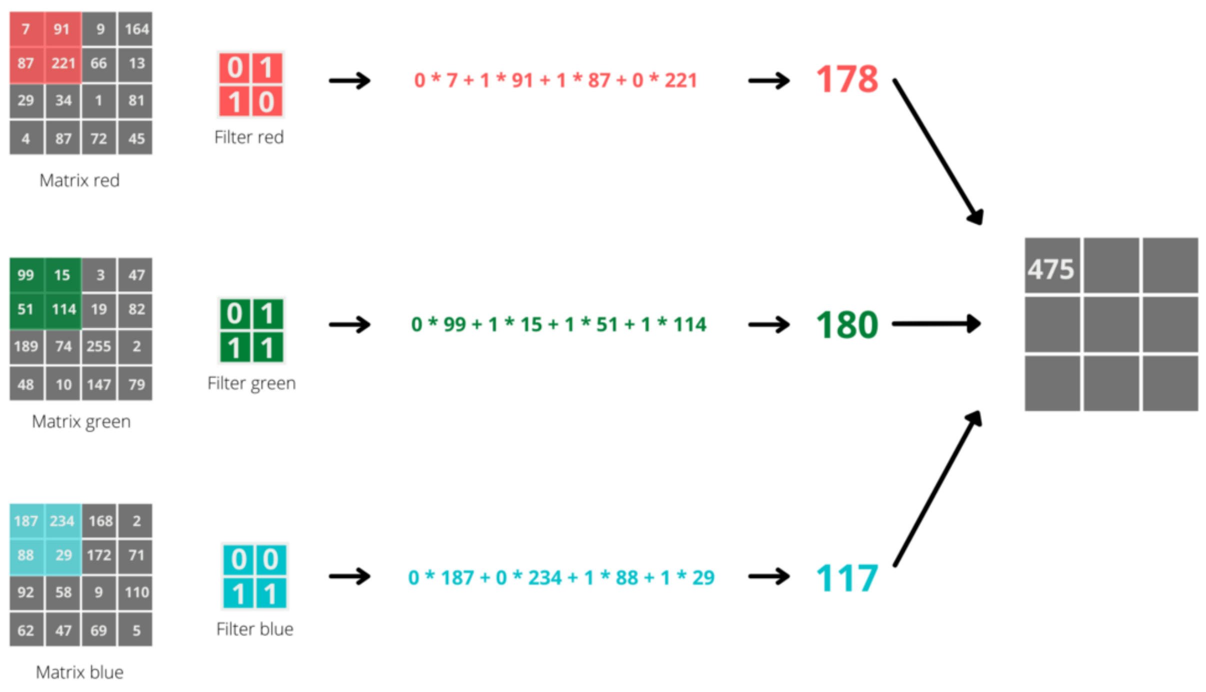
This challenge is exacerbated when we want to process larger images with more pixels and more color channels. Such a network with a huge number of parameters will most likely run into overfitting. This means that the model will give good predictions for the training set, but will not generalize well to new cases that it does not yet know. Additionally, due to a large number of parameters, the network would very likely stop attending to individual image details as they would be lost in sheer mass. However, if we want to classify an image, e.g. whether there is a dog in it or not, these details, such as the nose or the ears, can be the decisive factor for the correct result.

Convolutional Neural Network
For these reasons, the Convolutional Neural Network takes a different approach, mimicking the way we perceive our environment with our eyes. When we see an image, we automatically divide it into many small sub-images and analyze them one by one.

There are 3 layers in the CNN model.

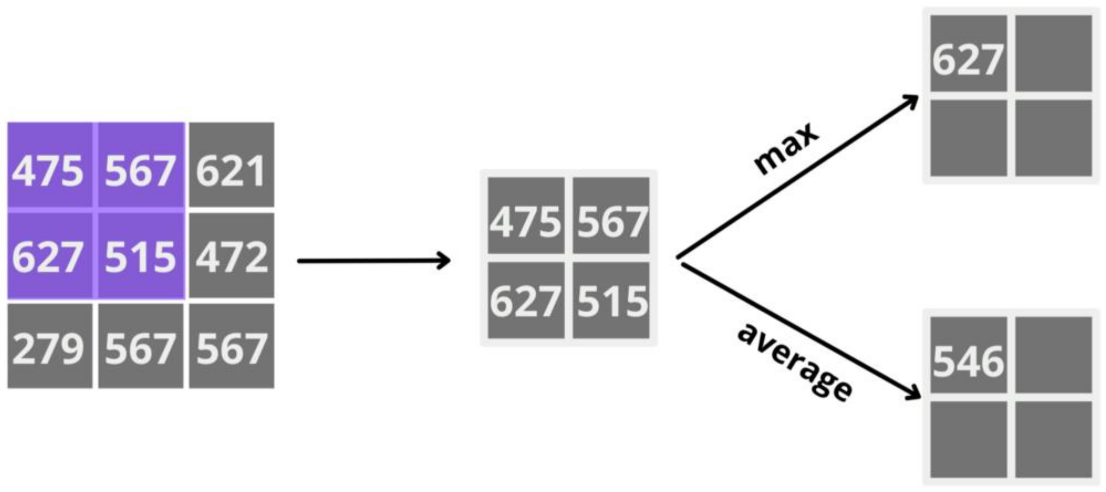
1. Convolution Layer

In the first step, we want to reduce the dimensions of the 4x4x3 image. For this purpose, we define a filter with the dimension 2x2 for each color. In addition, we want a step length of 1, i.e. after each calculation step, the filter should be moved forward by exactly one pixel. This will not reduce the dimension as much, but the details of the image will be preserved. If we migrate a 4x4 matrix with a 2x2 and advance one column or one row in each step, our Convolutional Layer will have a 3x3 matrix as output. The individual values of the matrix are calculated by taking the scalar product of the 2x2 matrices, as shown in the graphic.



Pooling Layer

The (Max) Pooling Layer takes the 3x3 matrix of the convolution layer as input and tries to reduce the dimensionality further and additionally take the important features in the image. We want to generate a 2x2 matrix as the output of this layer, so we divide the input into all possible 2x2 partial matrices and search for the highest value in these fields. This will be the value in the field of the output matrix. If we were to use the average pooling layer instead of a max-pooling layer, we would calculate the average of the four fields instead.



The pooling layer also filters out noise from the image, i.e. elements of the image that do not contribute to the classification. For example, whether the dog is standing in front of a house or in front of a forest is not important at first.

Fully-Connected Layer

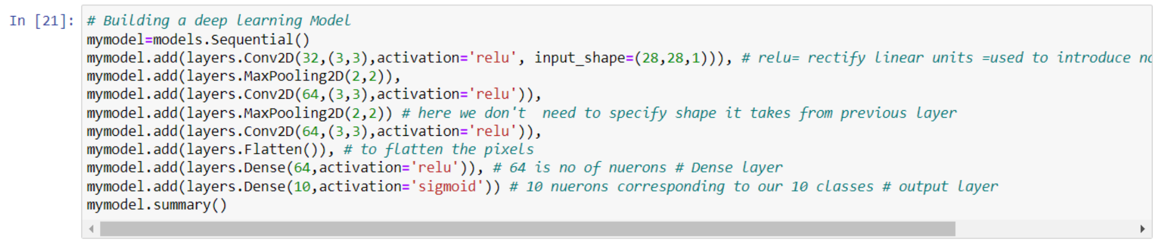
The fully-connected layer now does exactly what we intended to do with the whole image at the beginning. We create a neuron for each entry in the smaller 2x2 matrix and connect them to all neurons in the next layer. This gives us significantly fewer dimensions and requires fewer resources in training.

This layer then finally learns which parts of the image are needed to make the classification dog or non-dog. If we have images that are much larger than our 5x5x3 example, it is of course also possible to set the convolution layer and pooling layer several times in a row before going into the fully-connected layer. This way you can reduce the dimensionality far enough to reduce the training effort.

Build a Convolutional Neural Network Model

In Tensorflow we can now build the Convolutional Neural Network by defining the sequence of each layer. Since we are dealing with relatively small images we will use the stack of Convolutional Layer and Max Pooling Layer twice. The images have, as we already know, 28 height dimensions, 28width dimensions, and 1 color channel.

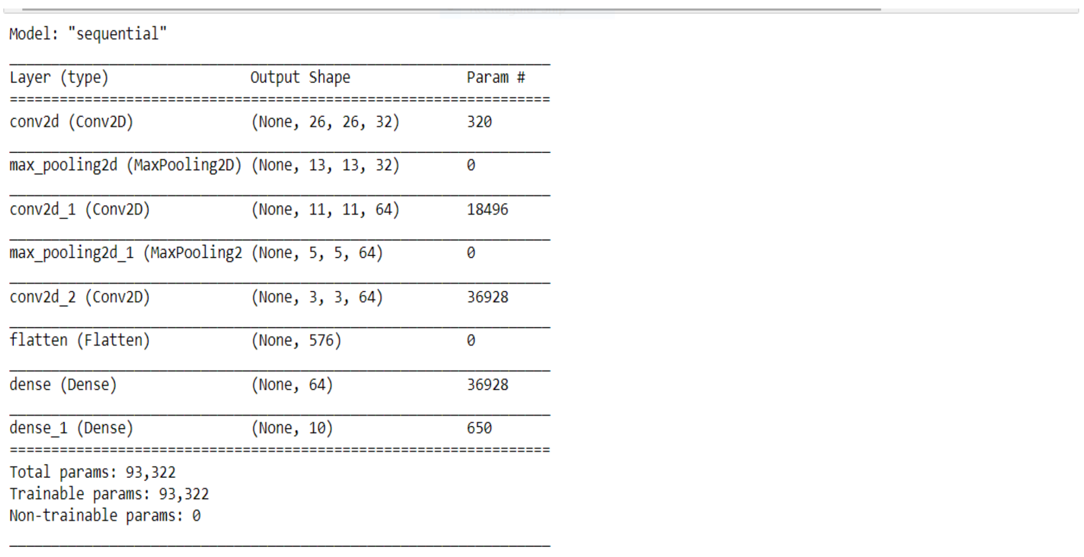
The Convolutional Layer uses first 32 and then 64 filters with a 3×3 kernel as a filter and the Max Pooling Layer searches for the maximum value within a 2×2 matrix.



```
In [21]: # Building a deep learning Model
mymodel=models.Sequential()
mymodel.add(layers.Conv2D(32,(3,3),activation='relu', input_shape=(28,28,1))), # relu= rectify linear units =used to introduce no
mymodel.add(layers.MaxPooling2D(2,2)),
mymodel.add(layers.Conv2D(64,(3,3),activation='relu')),
mymodel.add(layers.MaxPooling2D(2,2)) # here we don't  need to specify shape it takes from previous layer
mymodel.add(layers.Conv2D(64,(3,3),activation='relu')),
mymodel.add(layers.Flatten()), # to flatten the pixels
mymodel.add(layers.Dense(64,activation='relu')), # 64 is no of nuerons # Dense layer
mymodel.add(layers.Dense(10,activation='sigmoid')) # 10 nuerons corresponding to our 10 classes # output layer
mymodel.summary()
```

I had used relu as activation function to introduce non linearity in our network.

The sigmoid function is also called a squashing function as its domain is the set of all real numbers, and its range is (0, 1). Hence, if the input to the function is either a very large negative number or a very large positive number, the output is always between 0 and 1.



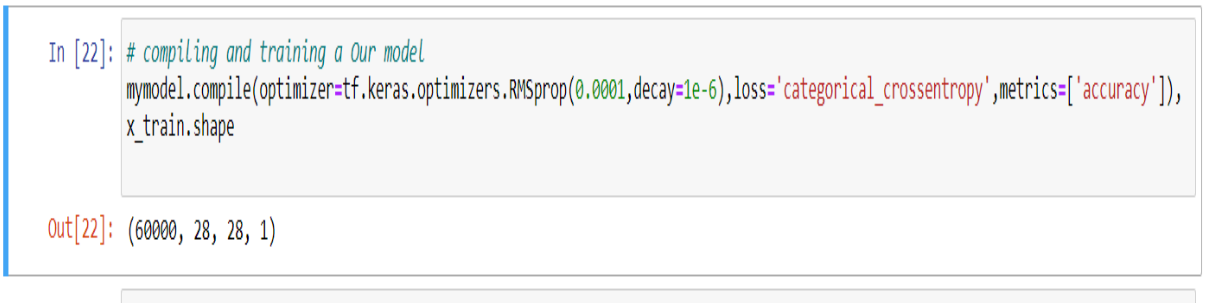
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 64)                36928
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
_________________________________________________________________
```

Now we have sufficiently reduced the dimensions of the images and can add one more hidden layer with a total of 64 neurons before the model ends in the output layer with the ten neurons for the ten different classes.

The model with a total of 93,322 parameters is now ready to be built and trained.

Compile and Train the Model

Before we can start training the Convolutional Neural Network, we have to compile the model. In it we define which loss function the model should be trained according to, the optimizer, i.e. according to which algorithm the parameters change, and which metric we want to be shown in order to be able to monitor the training process.

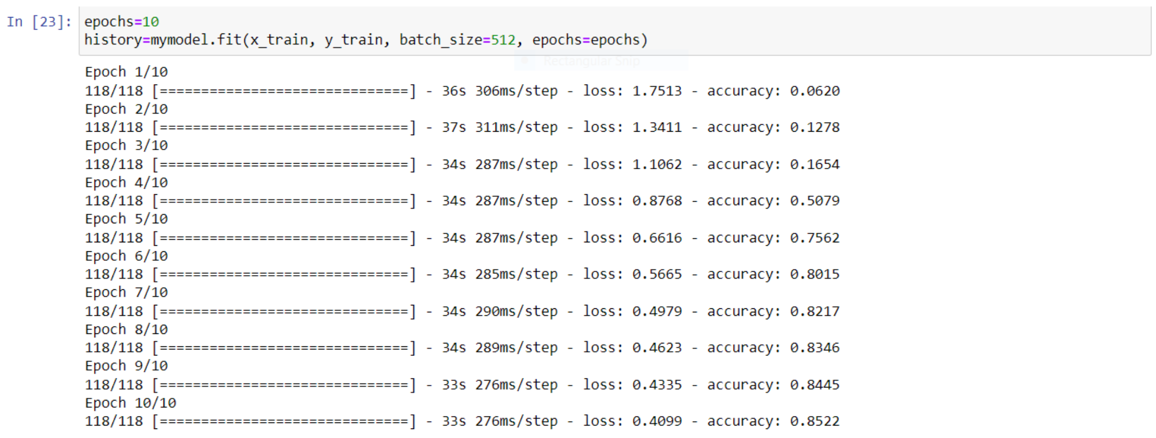


```
In [22]: # compiling and training a Our model
         mymodel.compile(optimizer=tf.keras.optimizers.RMSprop(0.0001,decay=1e-6),loss='categorical_crossentropy',metrics=['accuracy']),
         x_train.shape

Out[22]: (60000, 28, 28, 1)
```

I had used RMSprop Optimizer.

Root Mean Squared Propagation, or RMSProp, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter. The use of a decaying moving average allows the algorithm to forget early gradients and focus on the most recently observed partial gradients seen during the progress of the search, overcoming the limitation of AdaGrad.

RMSProp is a very effective extension of gradient descent and is one of the preferred approaches generally used to fit deep learning neural networks.

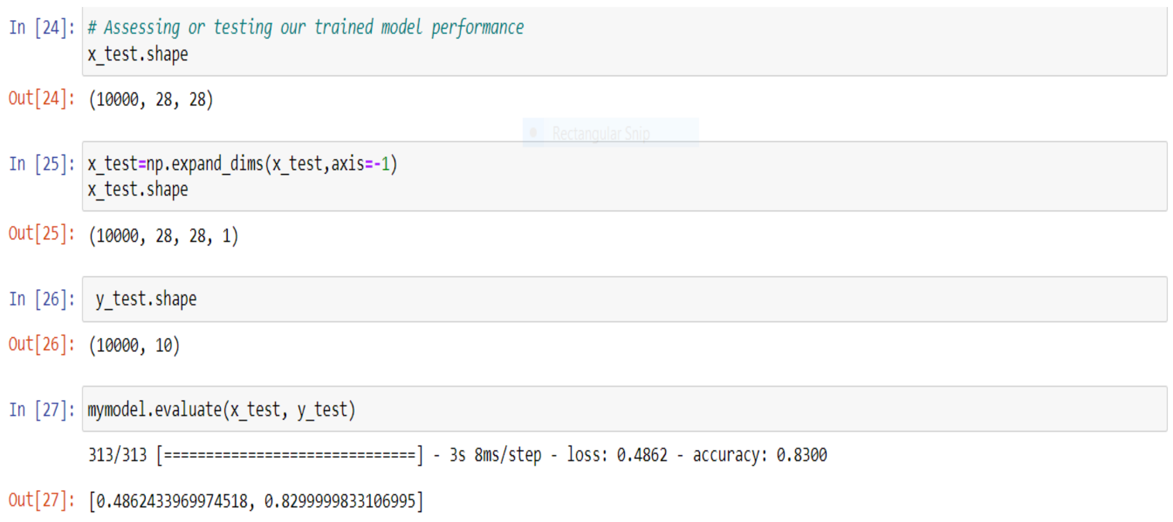


```
In [23]: epochs=10
         history=mymodel.fit(x_train, y_train, batch_size=512, epochs=epochs)

         Epoch 1/10
         118/118 [==============================] - 36s 306ms/step - loss: 1.7513 - accuracy: 0.0620
         Epoch 2/10
         118/118 [==============================] - 37s 311ms/step - loss: 1.3411 - accuracy: 0.1278
         Epoch 3/10
         118/118 [==============================] - 34s 287ms/step - loss: 1.1062 - accuracy: 0.1654
         Epoch 4/10
         118/118 [==============================] - 34s 287ms/step - loss: 0.8768 - accuracy: 0.5079
         Epoch 5/10
         118/118 [==============================] - 34s 287ms/step - loss: 0.6616 - accuracy: 0.7562
         Epoch 6/10
         118/118 [==============================] - 34s 285ms/step - loss: 0.5665 - accuracy: 0.8015
         Epoch 7/10
         118/118 [==============================] - 34s 290ms/step - loss: 0.4979 - accuracy: 0.8217
         Epoch 8/10
         118/118 [==============================] - 34s 289ms/step - loss: 0.4623 - accuracy: 0.8346
         Epoch 9/10
         118/118 [==============================] - 33s 276ms/step - loss: 0.4335 - accuracy: 0.8445
         Epoch 10/10
         118/118 [==============================] - 33s 276ms/step - loss: 0.4099 - accuracy: 0.8522
```

Epochs: It is an iteration every time we save all images to NN model and update the weights once.

Evaluate the Model

After training the Convolutional Neural Network for a total of 10 epochs, we can look at the progression of the model's accuracy to determine if we are satisfied with the training.



```
In [24]: # Assessing or testing our trained model performance
         x_test.shape
Out[24]: (10000, 28, 28)

In [25]: x_test=np.expand_dims(x_test,axis=-1)
         x_test.shape
Out[25]: (10000, 28, 28, 1)

In [26]:  y_test.shape
Out[26]: (10000, 10)

In [27]: mymodel.evaluate(x_test, y_test)
         313/313 [==============================] - 3s 8ms/step - loss: 0.4862 - accuracy: 0.8300
Out[27]: [0.4862433969974518, 0.8299999833106995]
```

Result:

Our prediction of the image class is correct in about 83% of the cases. This is not a bad value, but not a particularly good one either. If we want to increase this even further, we could have the Convolutional Neural Network trained for more epochs or possibly configure the dense layers even differently.

Conclusions:

In this project, we used Convolutional Neural Networks (CNN) for image classification using images form Fashion MNIST data sets. This data sets used both and training and testing purpose using CNN. It provides the accuracy rate 83%. Images used in the training purpose are small and Grayscale images. The computational time for processing these images is very high as compare to other normal JPEG images. Stacking the model with more layers and training the network with more image data using clusters of GPUs will provide more accurate results of classification of images. The future enhancement will focus on classifying the colored images of large size and its very useful for image segmentation process.

References:

https://www.researchgate.net/figure/Examples-from-the-Fashion-MNIST-dataset_fig6_333997546

https://towardsdatascience.com/metrics-to-evaluate-your-machine-learningalgorithm-f10ba6e38234

https://www.kaggle.com/code/arbazkhan971/image-classification-using-cnn-94-accuracy

-------------------------THANK YOU---------------------