

# IMPLEMENTATION OF DEEP LEARNING FRAMEWORKS



**ARTIFICIAL NEURAL NETWORKS  
USING SCIKIT LEARN**

## ARTIFICIAL NEURAL NETWORKS

---

### Introduction

One of the most fascinating machine learning modeling technique

Generally uses back propagation algorithm

Relatively complex (due to deep learning with many hidden layers)

Structure is inspired by brain functioning

Generally computationally expensive

## ARTIFICIAL NEURAL NETWORKS

### Instructions

1. Normalize the data – Use **Min – Max transformation (optional)**

$$\text{Normalized data} = \text{Data} - \text{Minimum} / (\text{Maximum} - \text{Minimum})$$

2. Number of hidden layers required = 1 for vast number of application
3. Number of neurons required =  $2/3$  of the number of predictor variables or input layers

**Remark:** The optimum number of layers and neurons are the ones which would minimize mean square error or misclassification error which can be obtained by testing again and again

**ARTIFICIAL NEURAL NETWORKS**

**Example:** Develop a model to predict the non payment of overdrafts by customers of a multinational banking institution. The data collected is given in Logistic\_Reg.csv file. The factors and response considered are given below. Use 80% of the data to develop the model and validate the model using remaining 20% of the data?

SL No	Factor
1	Individual expected level of activity score
2	Transaction speed score
3	Peer comparison score in terms of transaction volume

Response	Values
Outcome	0: Not Paid and 1: Paid

## ARTIFICIAL NEURAL NETWORKS

### Example

Importing packages

```
import pandas as mypd  
from sklearn.cross_validation import train_test_split  
from sklearn.neural_network import MLPClassifier
```

Reading the data

```
mydata = mypd.read_csv("E:/ISI/Course_Material/Data/Logistic_Reg.csv")  
x = mydata.values[:, 0:3]  
y = mydata.Outcome
```

Splitting the data into training and test

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state  
= 100)
```

## ARTIFICIAL NEURAL NETWORKS

### Example

Develop the model

```
mymodel =MLPClassifier(solver = 'lbfgs', alpha = 1e-5, hidden_layer_sizes = (2),  
random_state = 100)
```

```
mymodel.fit(x_train, y_train)
```

### Note:

Classification problem: Use **MLPClassifier**

Value estimation: Use **MLPRegressor**

### Solver:

**'lbfgs'** : Uses quasi-Newton method optimization algorithm.

**'sgd'** :Uses stochastic gradient descent optimization algorithm.

**'adam'** :Uses stochastic gradient-based optimizer

## ARTIFICIAL NEURAL NETWORKS

---

Example: Interpretation

`hidden_layer_sizes` : a vector representing hidden layers and hidden neurons in each layer

`hidden_layer_sizes = (l)` : one hidden layers with `l` hidden neurons



## ARTIFICIAL NEURAL NETWORKS

### Output

```
mymodel.score(x_train, y_train)
```

Statistics	Value
% Accuracy	96.81
% Error	3.19

```
mymodel.predict_proba(x_train)
```

## ARTIFICIAL NEURAL NETWORKS

Output: Validation

```
predtest = mymodel.predict(x_test)
```

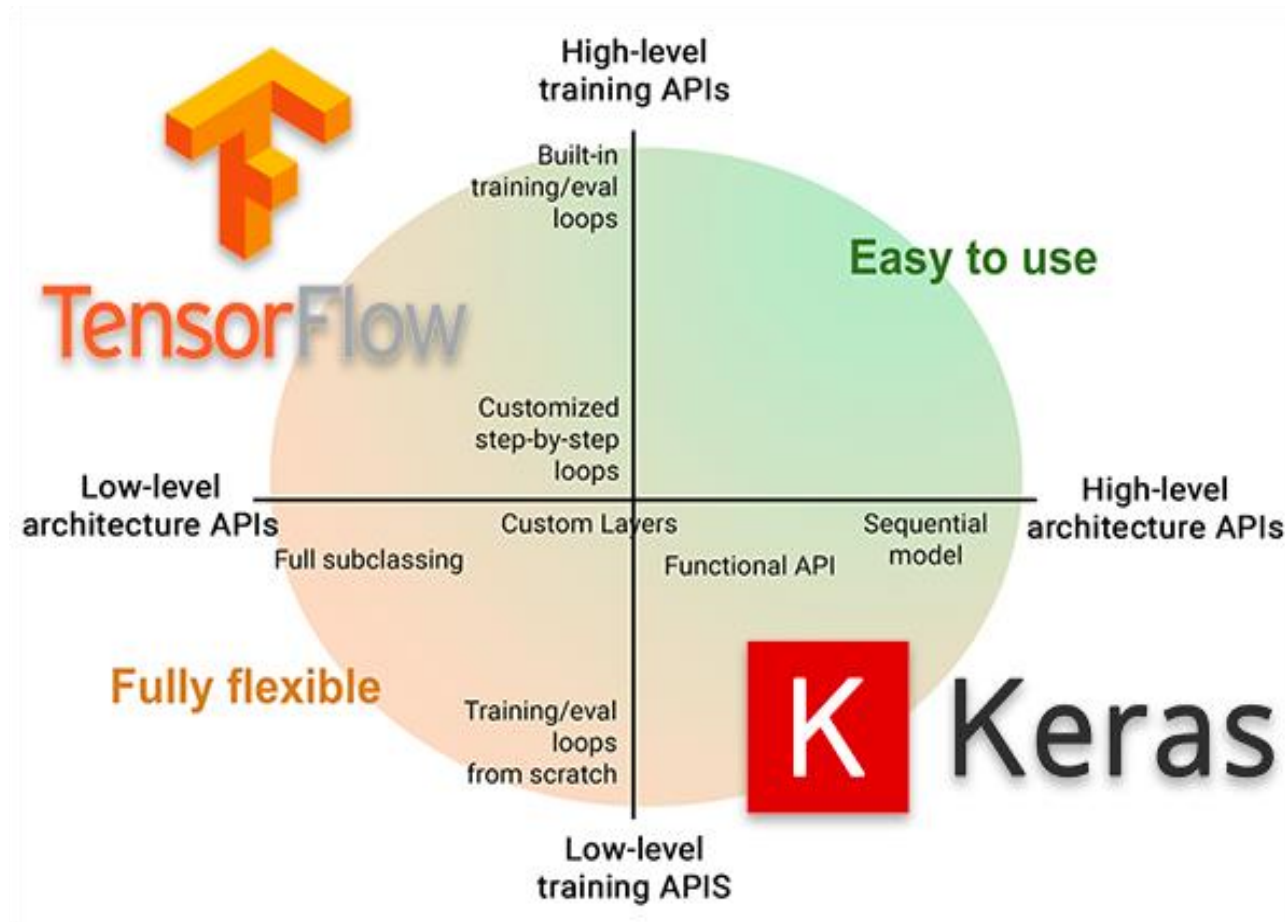
```
mytable = mypd.crosstab(y_test, predtest)
```

```
mytable
```

Actual Vs Predicted

		Predicted	
		0	1
Actual	0	54	4
	1	0	138

**ARTIFICIAL NEURAL NETWORKS  
USING TENSORFLOW & KERAS**

**RELATIONSHIP BETWEEN KERAS & TENSORFLOW**

## WHAT IS KERAS?

- Francois Chollet, the author of Keras, says:  
*“The framework was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.”*
- Keras is open source framework written in Python
- ONEIROS ( Open Ended Neuro-Electronic Intelligent Robot OS)
- Contains neural-network building blocks like layers, optimizer, activation functions
- Support CNN and RNN

## DEVELOPMENT OF KERAS

---

- Francois Chollet, Google AI Developer/Researcher developed Keras on 27 March 2015 to facilitate his own research and experiments.
- With the explosion of deep learning popularity, many developers, programmers, and machine learning practitioners flocked to Keras due to its easy-to-use API.
- At that time the popular deep learning libraries (Torch, Theano, and Caffe) **tedious, time-consuming, and inefficient.**
- **Keras, on the other hand, was extremely easy to use**

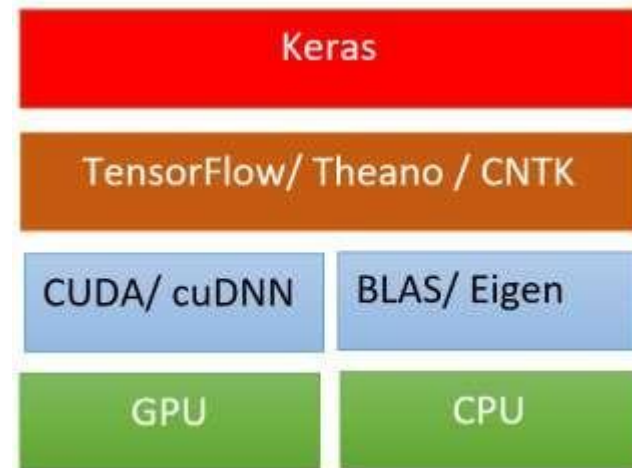
## BACKEND OF KERAS

---

- **A *backend* is a computational engine** — it builds the network graph/topology, runs the optimizers, and performs the actual number crunching.
- Keras might have several backend one at a time and can be thought as a set of abstractions that makes it easier to perform deep learning.
- Keras' default backend was ***Theano*** until v1.1.0.
- With the release of **TensorFlow** by Google, Keras started supporting TensorFlow as a backend, **resulting in TensorFlow being the *default backend* starting from the release of Keras v1.1.0.**

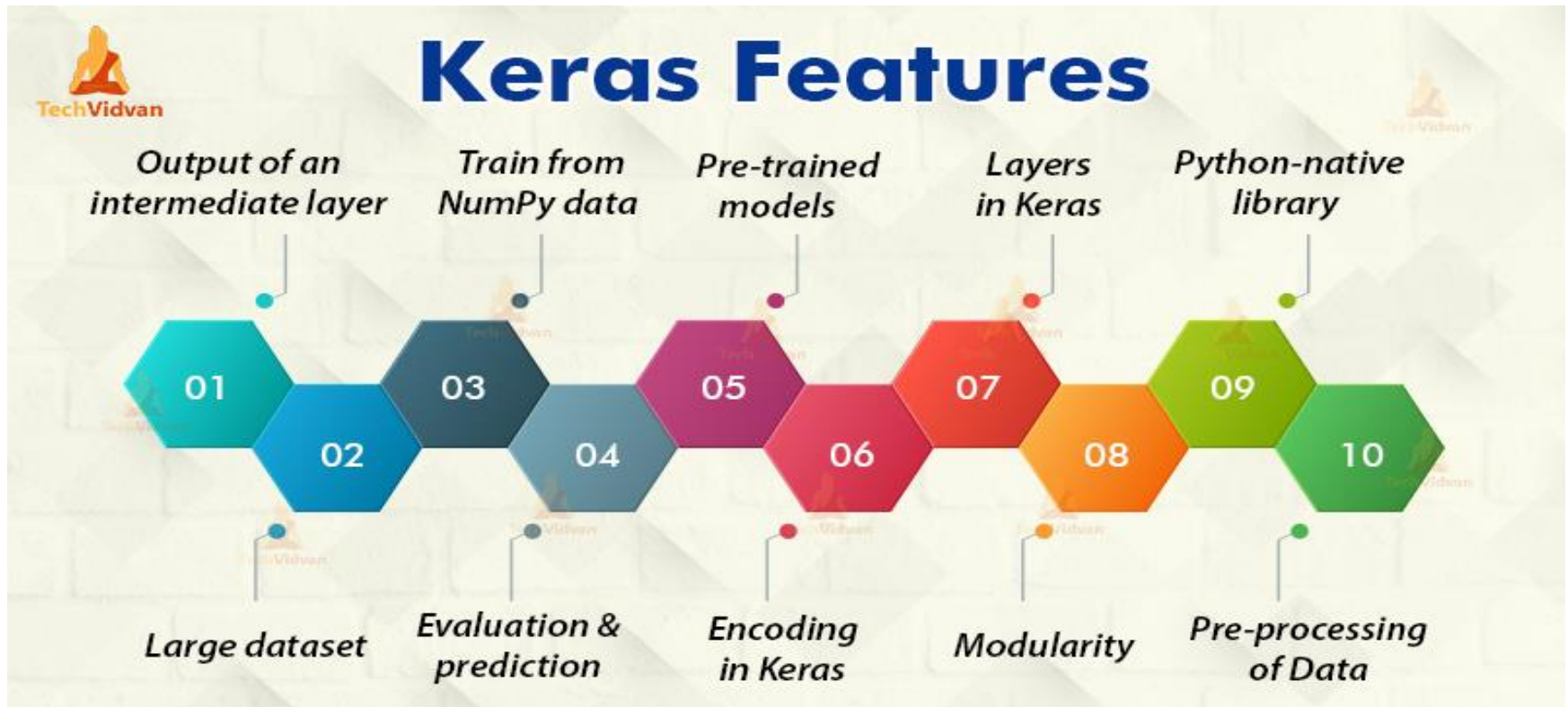
## KERAS FEATURES

- Contains datasets and some pre-trained deep learning applications.
- Model check-pointing, early stopping
- Uses libraries TensorFlow, Theano, CNTK as backend, only one at a time
- Backend does all computations
- Keras call backend functions
- Works for both CPU and GPU





## KERAS FEATURES



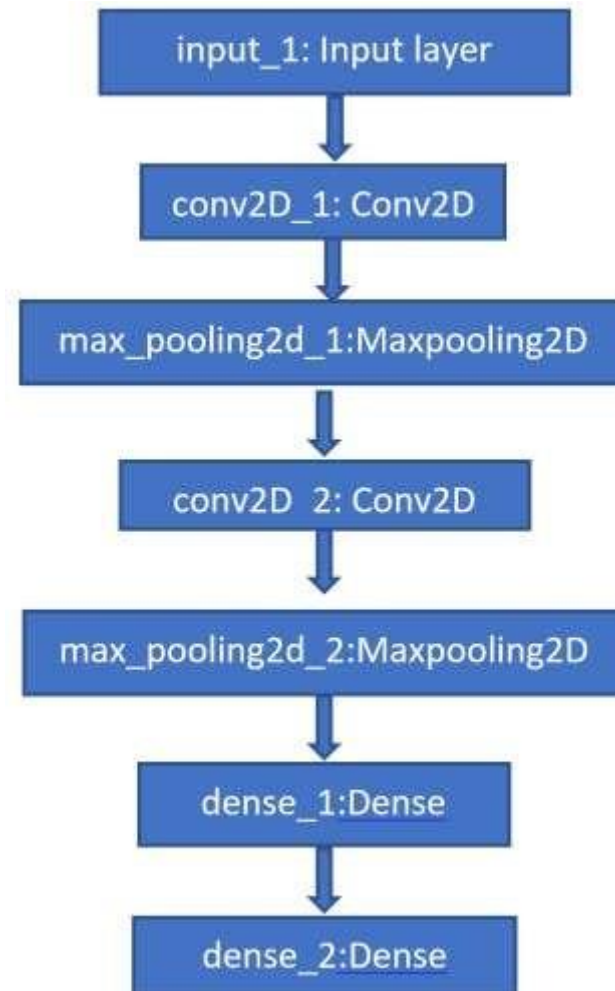
## KERAS FEATURES

- Rapid prototyping-
  1. Build neural network with minimal lines of code
  2. Build simple or complex neural networks within a few minutes
- Flexibility-

Sometime it is desired to define own metrics, layers, a cost function, Keras provide freedom for the same.
- Two types of built in models
  1. Sequential
  2. Functional
- All models have following common properties
  - Inputs that contain a list of input tensors
  - Layers, which comprise the model graph
  - Outputs, a list of output tensors.

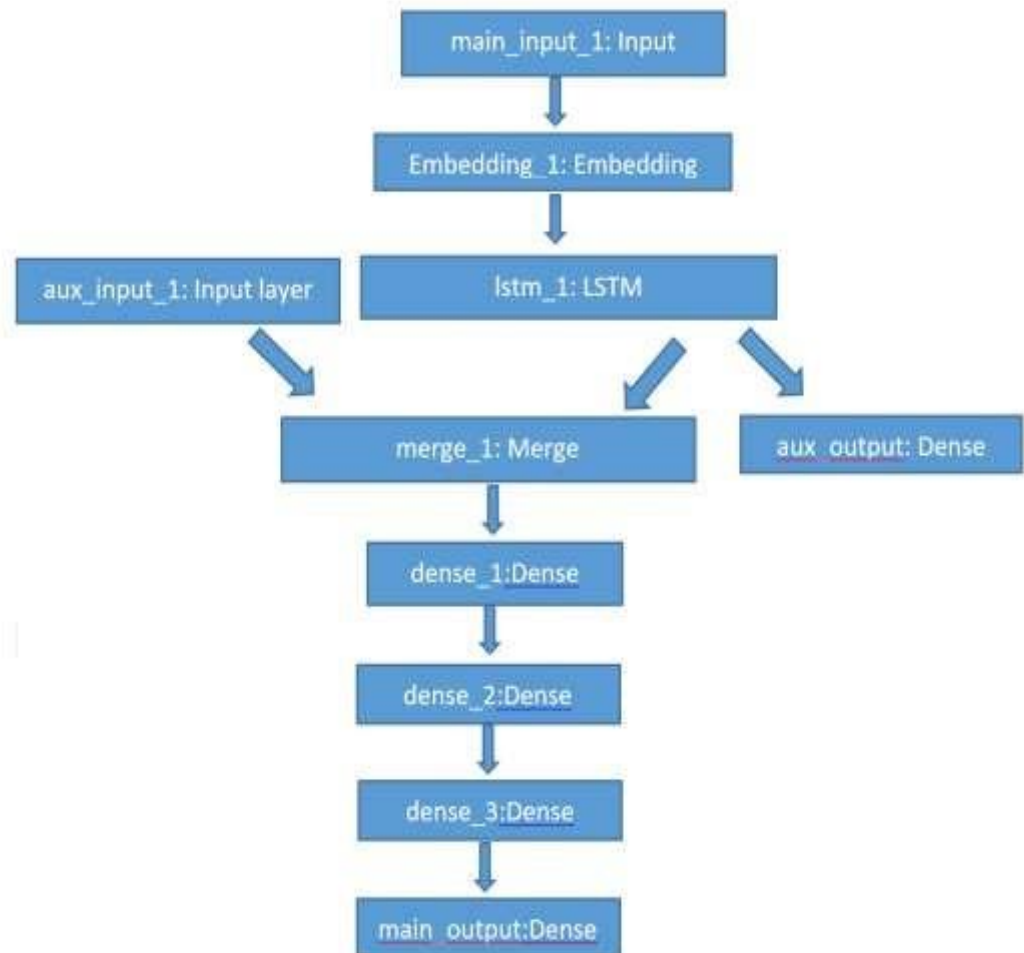
## SEQUENTIAL MODEL

- It is linear stack of layers
- Output of previous layer is input to the next layer.
- Create models by stacking one layer on top of other
- Useful in situation where task is not complex
- Provides higher level of abstraction

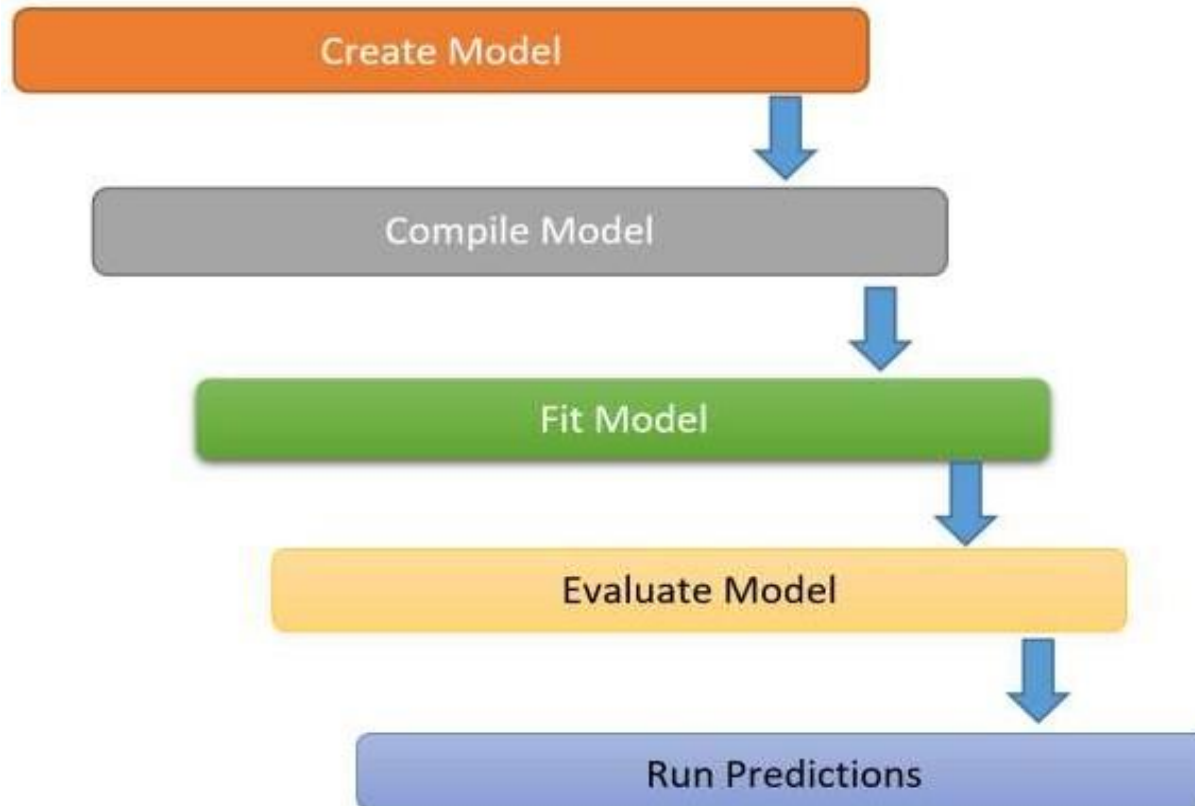


## FUNCTIONAL MODEL

- It define more complex models
- Such as directed acyclic graphs
- Multi input output models
- Model with shared layers
- Possible to connect a layer with any other layer



## STEPS IN BUILDING A MODEL



## MODEL METHODS

---

1. Compile: It is used to configure model. It accept following parameters
  - **Optimizer:**
    - This specifies type of optimiser to use in back-propagation algorithm
    - SGD, Adadelata, Adagrad , Adam , Nadam optimizer and many others.
  - **Loss:**
    - It is the objective function
    - It track losses or the drift from the function during training the model.
    - For regression: mean squared error, mean absolute error etc.
    - For classification: Categorical cross-entropy, binary cross entropy
    - Different loss functions for different outputs

## MODEL METHODS

---

- **Metrics:**
  - It is similar to objective function.
  - Results from metric aren't used when training model.
  - It specifies the list of metrics that model evaluate during training and testing.
  - The commonly used metric is the accuracy metric.
  - Possible to specify different metrics for different output

## MODEL METHODS

---

1. **Compile:** It is used to configure model. It accept following parameters

- **Optimizer:**

- This specifies type of optimiser to use in back-propagation algorithm
- SGD, Adadelata, Adagrad , Adam , Nadam optimizer and many others.

- **Loss:**

- It is the objective function
- It track losses or the drift from the function during training the model.
- For regression: mean squared error, mean absolute error etc.
- For classification: Categorical cross-entropy, binary cross entropy
- Different loss functions for different outputs



## MODEL METHODS

---

- **epochs:**
  - An epoch is an iteration
  - It specifies number of times training data is feed to the model.
- **validation\_split:**
- Validation data is selected from the end samples
  - At the end of each epoch, loss and metrics are calculated for this data.
- **validation\_data:**
  - Fraction of the data that is used as validation.

## KERAS DATASETS

---

Keras contains various datasets that are used to build neural networks.

The datasets are described below

### 1. **Boston House Pricing dataset:**

It contains 13 attributes of houses of Boston suburbs in the late 1970s  
Used in regression problems

### 2. **CIFAR10:**

- It is used for classification problems.
- This dataset contains 50,000 32×32 colour images
- Images are labelled over 10 categories
- 10,000 test images.

### 3. **CIFAR100:**

- Same as CIFAR10 but it has 100 categories

## KERAS DATASETS

---

### 4. MNIST:

- This dataset contains 60,000 28×28 grayscale images of 10 digits
- Also include 10,000 test images.

### 5. Fashion-MNIST:

- This dataset is used for classification problems.
- This dataset contains 60,000 28×28 grayscale images of 10 categories, along

### 6. IMDB movie reviews data:

- Dataset of 25,000 movies reviews from IMDB
- labeled by sentiment (positive/negative).

### 7. Reuters newswire topics classification:

- Dataset of 11,228 newswires from Reuters, labeled over 46 topics

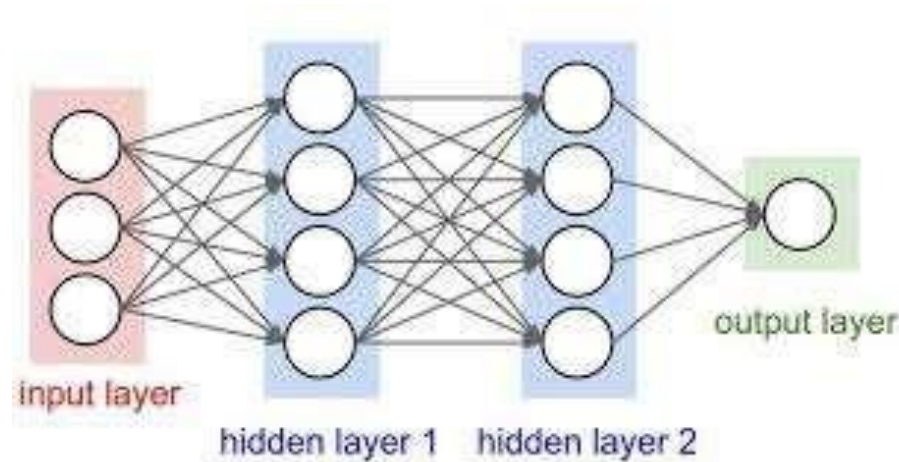
## KERAS LAYERS

It consist of different types of layers used in deep learning such as

:

### 1. Dense layer:

- A Dense layer is fully connected neural network layer

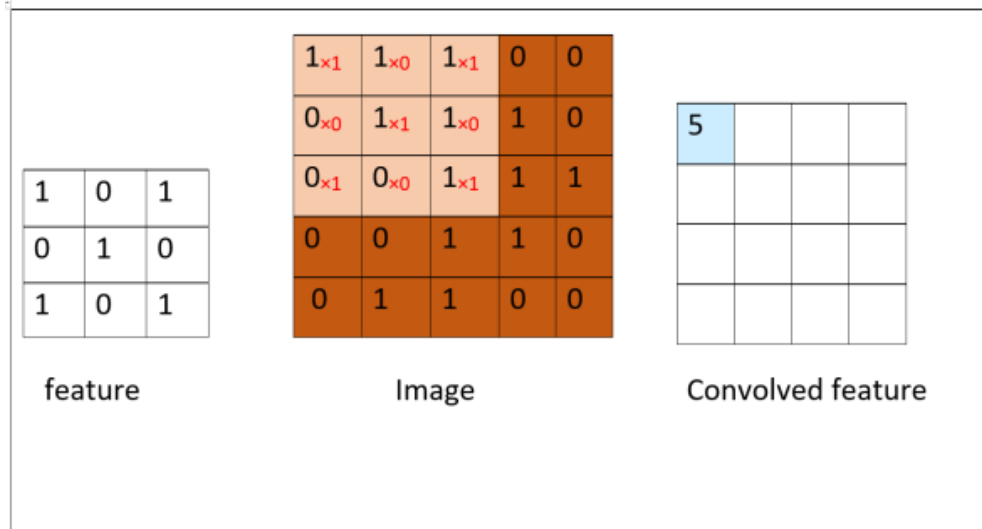


(Ramchoun et al, 2016)

## KERAS LAYERS

### 2. Convolutional layer:

- Mostly used in computer vision.
- It extract features from the input image.
- It preserves the spatial relationships between pixels
- It learn image features using small squares of input data
- Finally, the obtained matrix is known as the feature map



### **3. Pooling layer**

- Also called as subsampling or down-sampling layer
- Pooling reduces the dimensionality of feature map
- Retains the most important information
- There are many types of pooling layers such as
- MaxPooling and AveragePooling
- In case of max pooling, take the largest element from the rectified feature map within that window.
- In average pooling, average of all elements in that window is taken.

## KERAS LAYERS

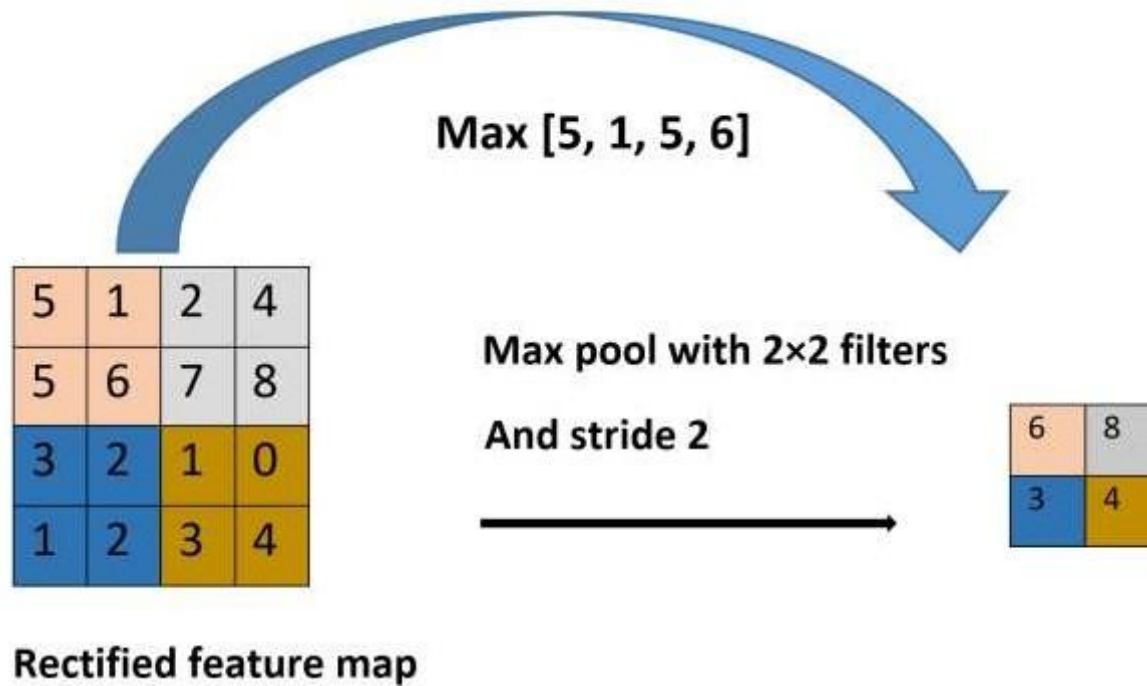


Figure: Max Pooling Concept

## KERAS LAYERS

---

### 4. Recurrent layer

- Basic building block of RNN
- This is mostly used in sequential and time series modelling.

### 5. Embedding layers

- Required when input is text
- These are mostly used in Natural Language Processing.

### 6. Batch Normalisation layer

- Normalize the activations of the previous layer at each batch
- Applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.



## IMPLEMENTATION OF ACTIVATION FUNCTION

### PACKAGE INSTALLATION

```
! pip install tensorflow
import tensorflow as tf
import numpy as np
import math
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
```

### DEFINING PLOT FUNCTION

```
def do_plot(x, y, title):
    plt.figure(figsize=(10,5))
    plt.plot(x,y)
    plt.title(title)
    plt.ylabel('Y axis')
    plt.xlabel('X axis')
    plt.show()
```

## IMPLEMENTATION OF ACTIVATION FUNCTION

### DATA

```
x = tf.Variable(tf.range(-10, 10, 0.1), dtype=tf.float32)
```

```
y_predicted = np.array([1,1,0,0,1])
```

```
y_true = np.array([0.30,0.7,1,0,0.5])
```

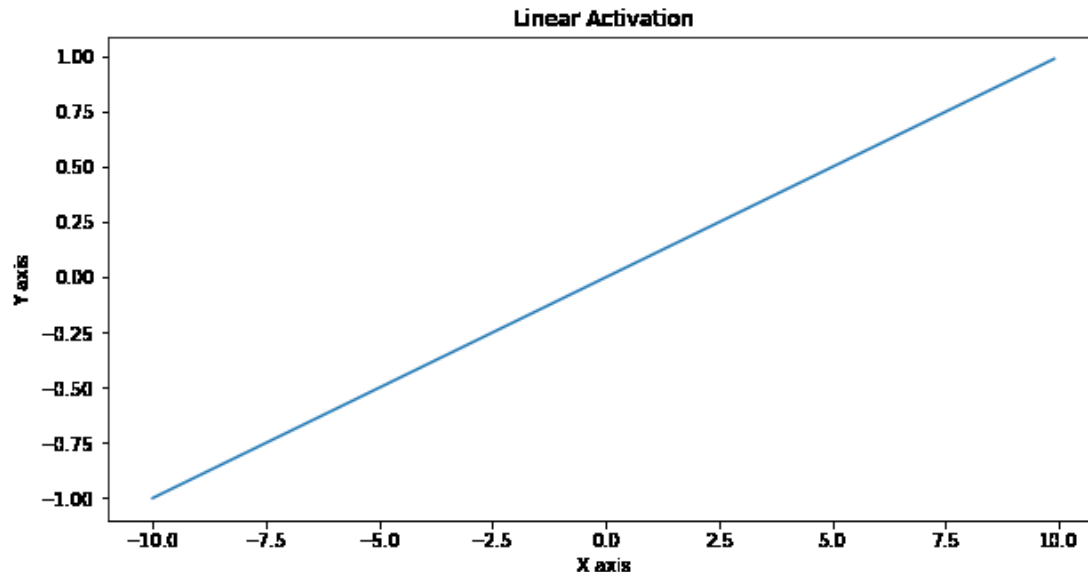
### LINEAR ACTIVATION

```
def linear_activation(x):
```

```
    c = 0.1
```

```
    return c*x.numpy()
```

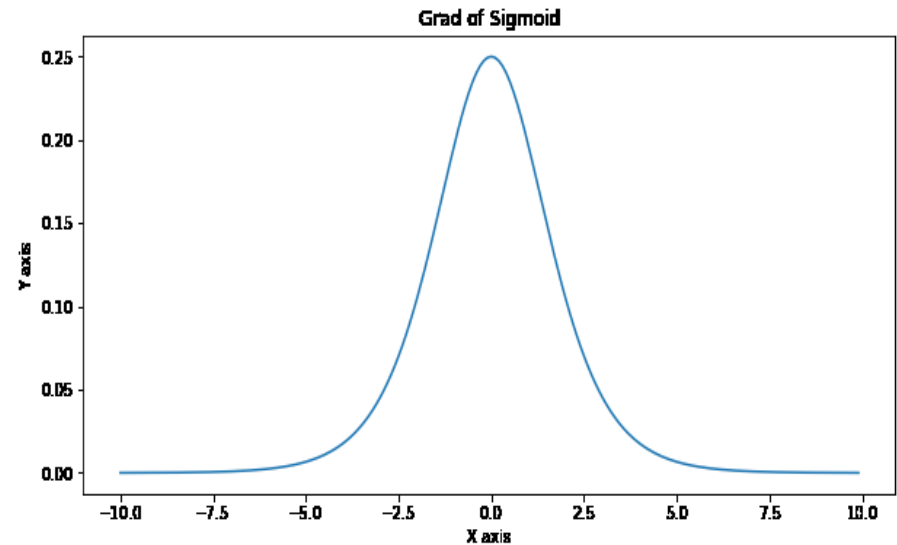
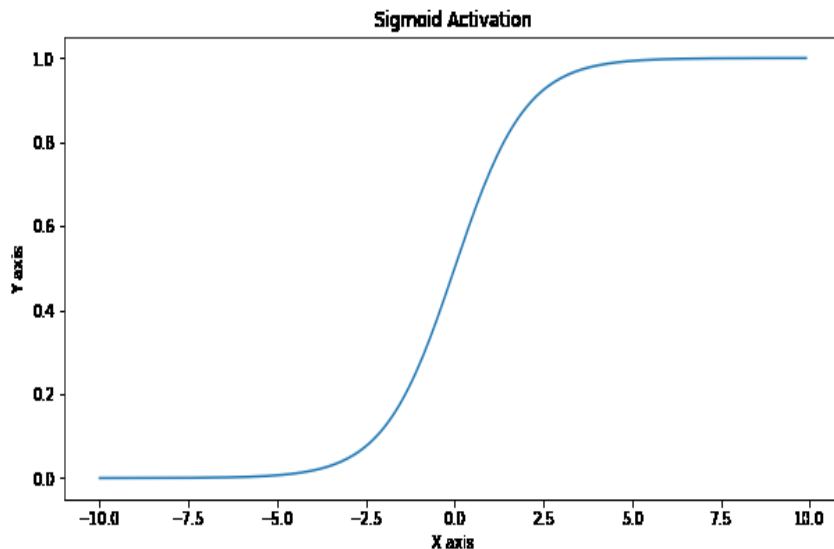
```
    do_plot(x.numpy(), linear_activation(x), 'Linear Activation')
```



## IMPLEMENTATION OF ACTIVATION FUNCTION

### SIGMOID ACTIVATION

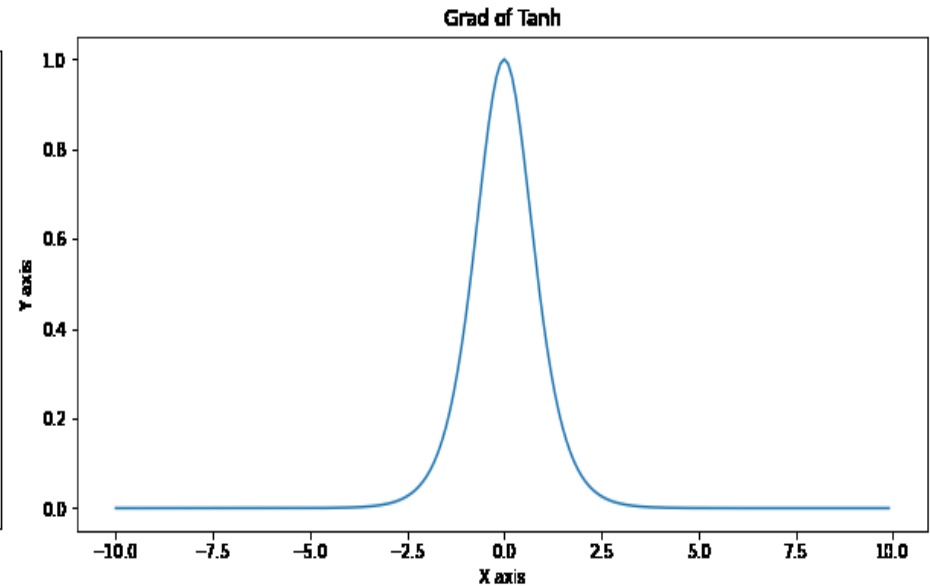
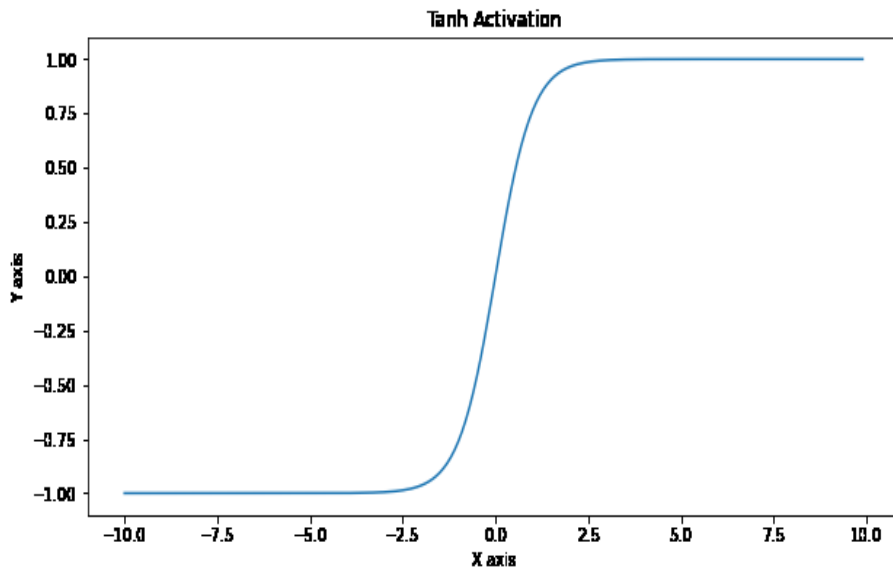
```
y = tf.nn.sigmoid(x)  
do_plot(x.numpy(), y.numpy(), 'Sigmoid Activation')  
with tf.GradientTape() as t:  
    y = tf.nn.sigmoid(x)  
do_plot(x.numpy(), t.gradient(y, x).numpy(), 'Grad of Sigmoid')
```



# IMPLEMENTATION OF ACTIVATION FUNCTION

## TANH ACTIVATION

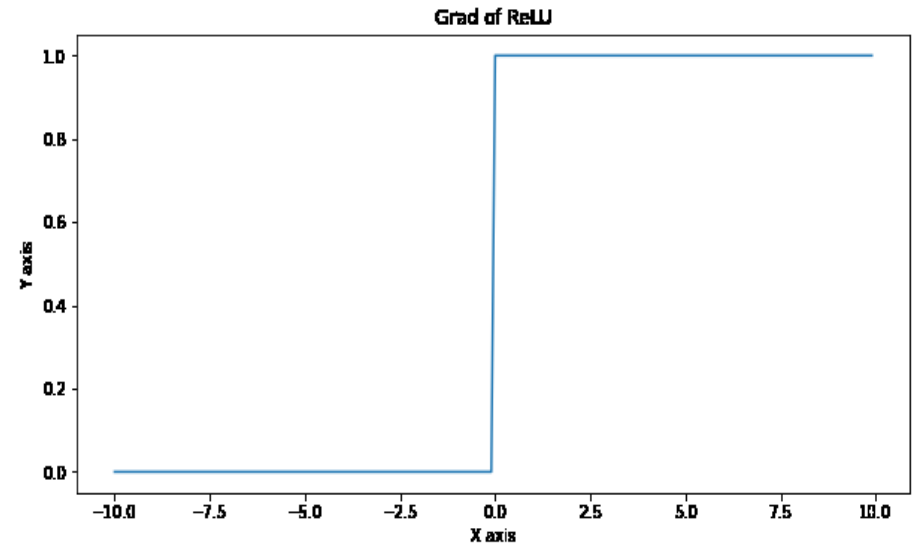
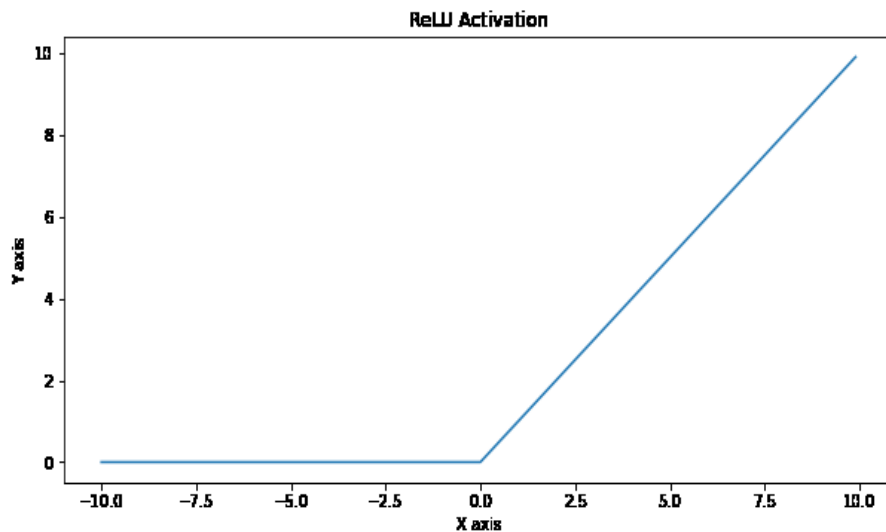
```
def tanh(x):  
    return (math.exp(x) - math.exp(-x)) / (math.exp(x) + math.exp(-x))  
y = tf.nn.tanh(x)  
do_plot(x.numpy(), y.numpy(), 'Tanh Activation')  
with tf.GradientTape() as t:  
    y = tf.nn.tanh(x)  
do_plot(x.numpy(), t.gradient(y, x).numpy(), 'Grad of Tanh')
```



# IMPLEMENTATION OF ACTIVATION FUNCTION

## RELU ACTIVATION

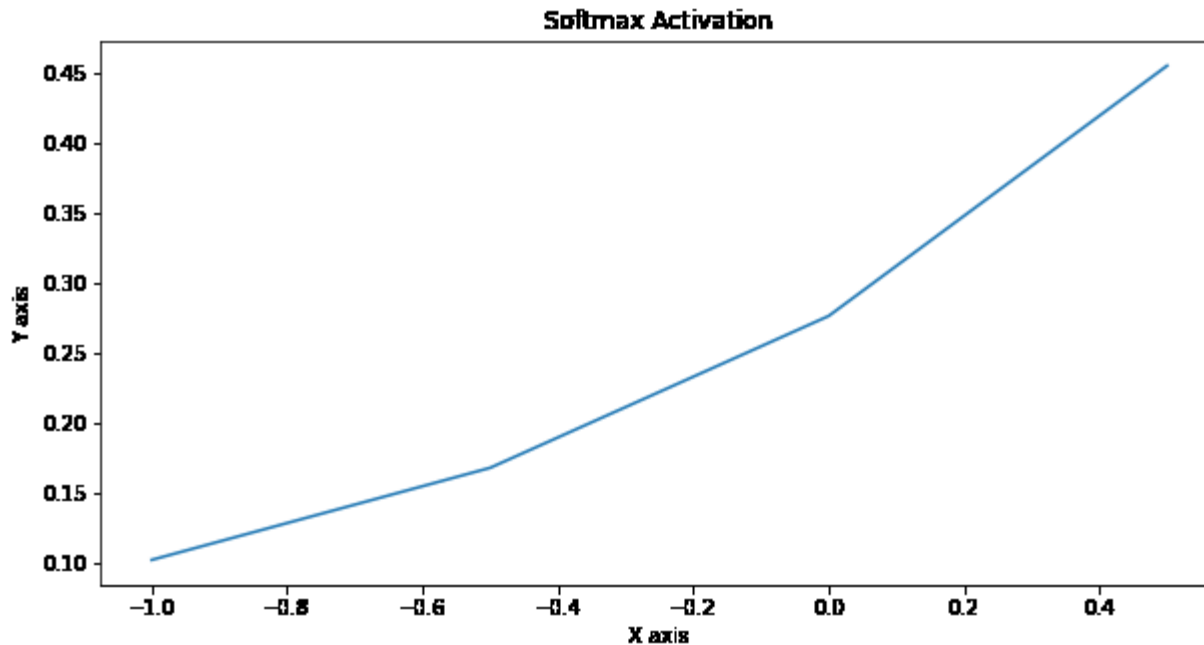
```
def relu(x):  
    return max(0,x)  
y = tf.nn.relu(x)  
do_plot(x.numpy(), y.numpy(), 'ReLU Activation')  
with tf.GradientTape() as t:  
    y = tf.nn.relu(x)  
do_plot(x.numpy(), t.gradient(y, x).numpy(), 'Grad of ReLU')
```



## IMPLEMENTATION OF ACTIVATION FUNCTION

### SOFTMAX ACTIVATION

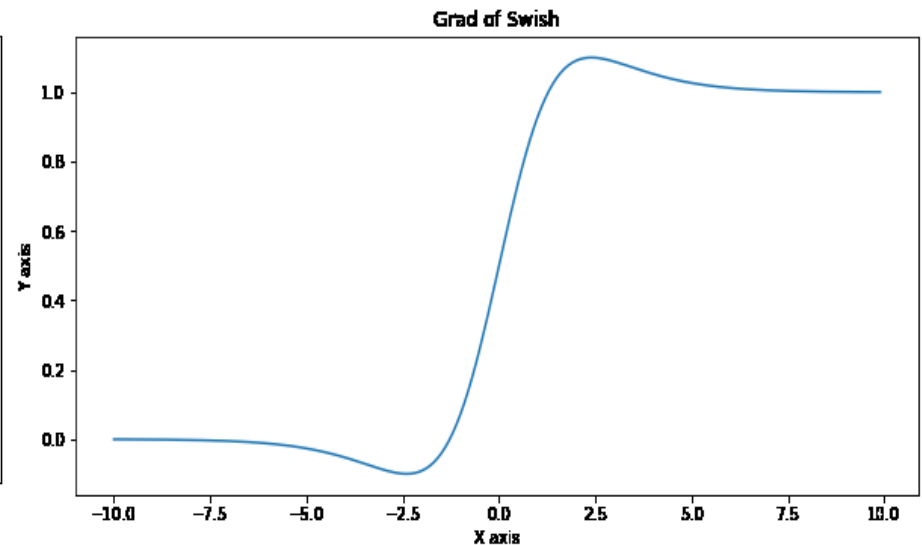
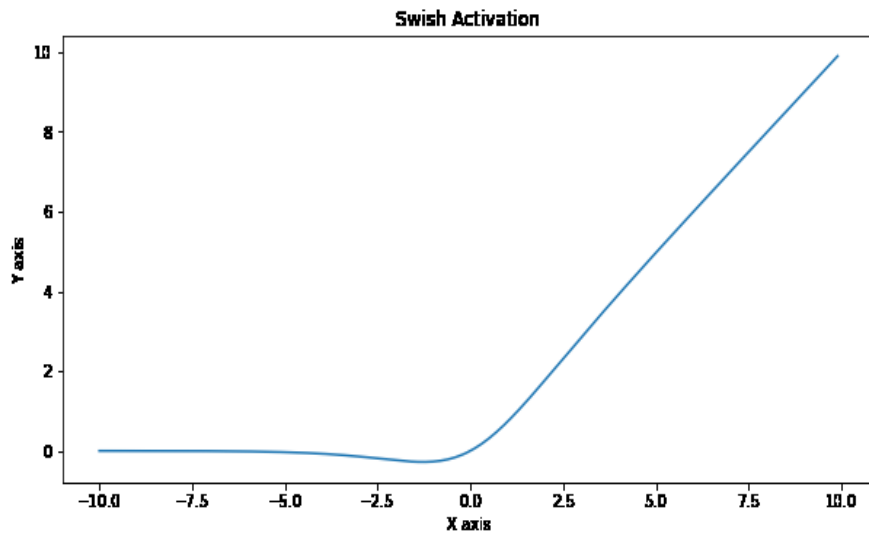
```
x1 = tf.Variable(tf.range(-1, 1, .5), dtype=tf.float32)  
y = tf.nn.softmax(x1)  
do_plot(x1.numpy(), y.numpy(), 'Softmax Activation')
```



# IMPLEMENTATION OF ACTIVATION FUNCTION

## SWISH ACTIVATION

```
y = tf.nn.swish(x)
do_plot(x.numpy(), y.numpy(), 'Swish Activation')
with tf.GradientTape() as t:
    y = tf.nn.swish(x)
do_plot(x.numpy(), t.gradient(y, x).numpy(), 'Grad of Swish')
```



## IMPLEMENTATION OF LOSS FUNCTION

### MEAN ABSOLUTE ERROR

#### USING NUMPY FUNCTION

```
def mae_np(y_predicted, y_true):  
    return np.mean(np.abs(y_predicted-y_true))  
mae_np(y_predicted, y_true)
```

#### USER DEFINED FUNCTION

```
def mae(y_predicted, y_true):  
    total_error = 0  
    for yp, yt in zip(y_predicted, y_true):  
        total_error += abs(yp - yt)  
    print("Total error is:",total_error)  
    mae = total_error/len(y_predicted)  
    print("Mean absolute error is:",mae)  
    return mae  
mae(y_predicted, y_true)
```



**OUTPUT**  
**0.5**



## IMPLEMENTATION OF LOSS FUNCTION


### MEAN SQUARE ERROR

USING NUMPY FUNCTION

```
np.mean(np.square(y_true-y_predicted))
```

USER DEFINED FUNCTION

```
def mse(y_true, y_predicted):  
    total_error = 0  
    for yt, yp in zip(y_true, y_predicted):  
        total_error += (yt-yp)**2  
    print("Total Squared Error:",total_error)  
    mse = total_error/len(y_true)  
    print("Mean Squared Error:",mse)  
    return mse  
mse(y_true, y_predicted)
```



**OUTPUT**  
**0.366**

## IMPLEMENTATION OF LOSS FUNCTION

### LOG LOSS

#### USER DEFINED FUNCTION

```
epsilon = 1e-15  
y_predicted_new = [max(i,epsilon) for i in y_predicted]  
y_predicted_new = [min(i,1-epsilon) for i in y_predicted_new]  
y_predicted_new = np.array(y_predicted_new)  
-np.mean(y_true*np.log(y_predicted_new)+(1-y_true)*np.log(1-y_predicted_new))
```

**OUTPUT**  
**17.26**

## HANDWRITTEN DIGIT RECOGNITION PROBLEM

---

- The dataset was constructed from a number of scanned document dataset available from the National Institute of Standards & Technology (NIST).
- Images of digits were taken from a variety of scanned documents, normalized in size and centred.
- Each image is a 28 by 28 pixel square (784 pixels total). A standard split of the dataset is used to evaluate and compare models, where 60,000 images are used to train a model and a separate set of 10,000 images are used to test it.
- It is a digit recognition task. As such there are 10 digits (0 to 9) or 10 classes to predict. Results are reported using prediction error, which is nothing more than the inverted classification accuracy.

## HANDWRITTEN DIGIT RECOGNITION PROBLEM

Import the packages

```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
```

Load the data

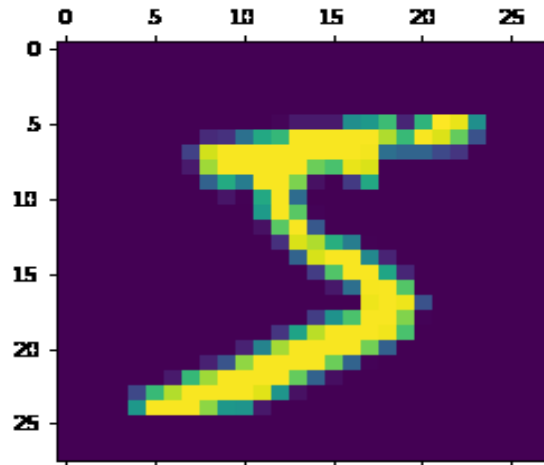
```
(X_train, y_train) , (X_test, y_test) = keras.datasets.mnist.load_data()
```

Visualizing our 1st input data

```
plt.matshow(X_train[0])
```

```
y_train[0]
```

**5**



## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Data Scaling

```
X_train = X_train / 255
```

```
X_test = X_test / 255
```

### Data Shape

```
X_train[0].shape
```

(28, 28)

Flatten the 28\*28 grid data into a 1-dimensional data

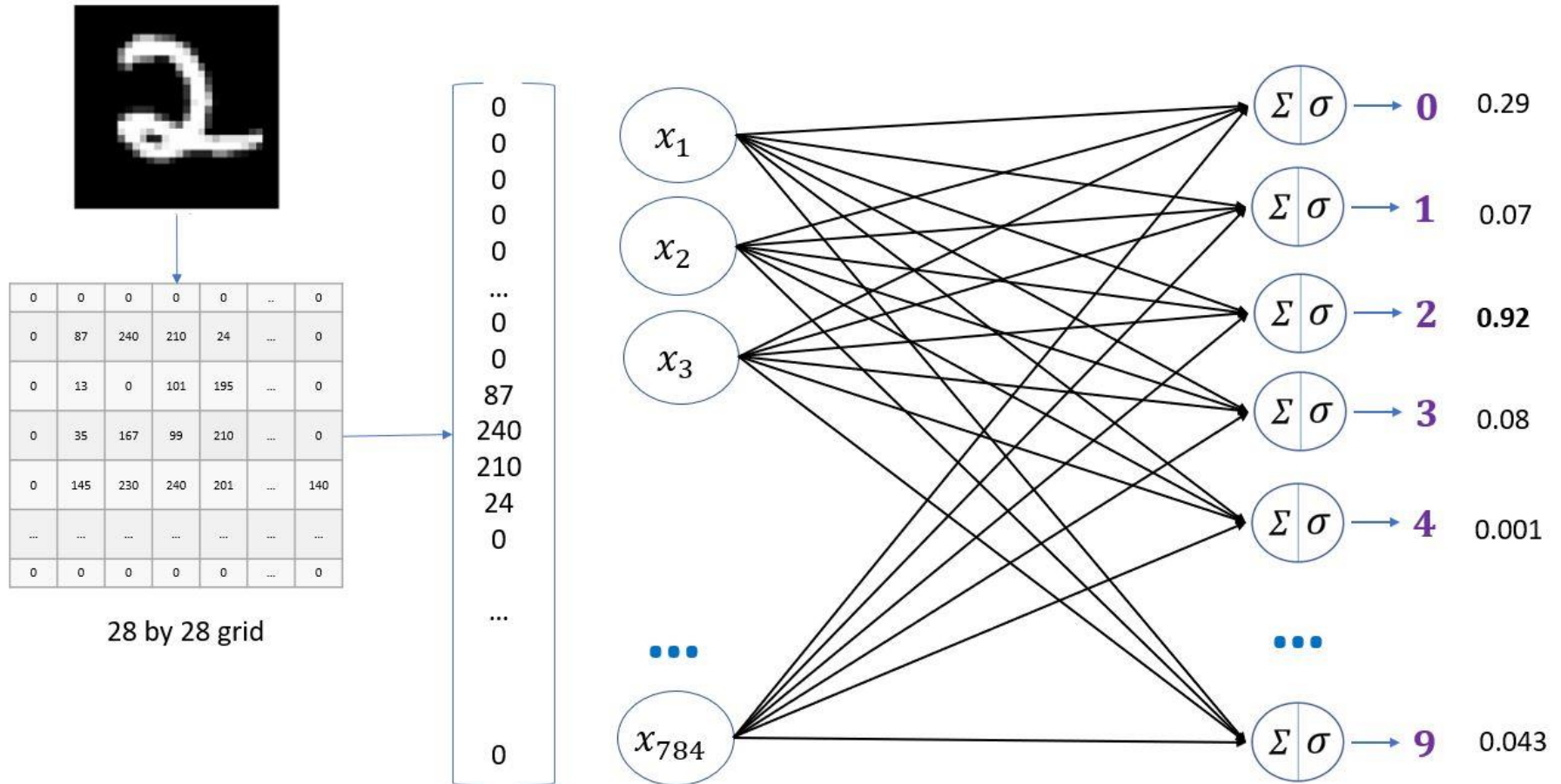
```
X_train_flattened = X_train.reshape(len(X_train), 28*28)
```

```
X_test_flattened = X_test.reshape(len(X_test), 28*28)
```

### Flattened Data Shape

```
X_train_flattened.shape
```

(60000, 784)

**HANDWRITTEN DIGIT RECOGNITION PROBLEM**

**HANDWRITTEN DIGIT RECOGNITION PROBLEM****Simple Neural Network without hidden layer with Flattened Data**

```
model = keras.Sequential([  
    keras.layers.Dense(10, input_shape=(784,), activation='sigmoid')])  
model.compile(optimizer='adam',  
    loss='sparse_categorical_crossentropy',  
    metrics=['accuracy'])  
model.fit(X_train_flattened, y_train, epochs=5)
```

...

```
Epoch 5/5 1875/1875 [=====] - 3s  
1ms/step - loss: 0.2668 - accuracy: 0.9252
```

## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Simple Neural Network without hidden layer with Flattened Data

```
model.evaluate(X_test_flattened, y_test)
```

```
313/313 [=====] - 1s 1ms/step - loss:  
0.2667 - accuracy: 0.9251 [0.2667323350906372, 0.9251000285148621]
```

```
y_predicted = model.predict(X_test_flattened)
```

Testing Our 1<sup>st</sup> Test Data

```
y_predicted[0]
```

```
array([2.27106214e-02, 5.61349339e-07, 8.58167410e-02, 9.66806769e-01,  
3.78498435e-03, 1.18708253e-01, 2.98759551e-06, 9.99864399e-01,  
1.03265375e-01, 6.57766342e-01], dtype=float32)
```



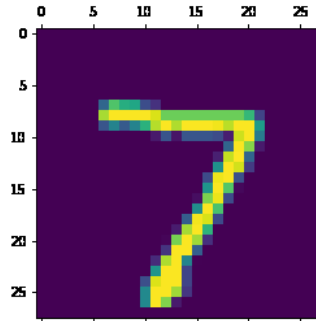
## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Simple Neural Network without hidden layer with Flattened Data

```
plt.matshow(X_test[0])
```

```
np.argmax(y_predicted[0])
```

(This gives the index of the output layer  
where the maximum value occurs)



7

So our model gives accurate prediction for the 1<sup>st</sup> test data.

We shall check the labels for the first 5 test data

```
y_predicted_labels = [np.argmax(i) for i in y_predicted]
```

```
y_predicted_labels[:5]
```

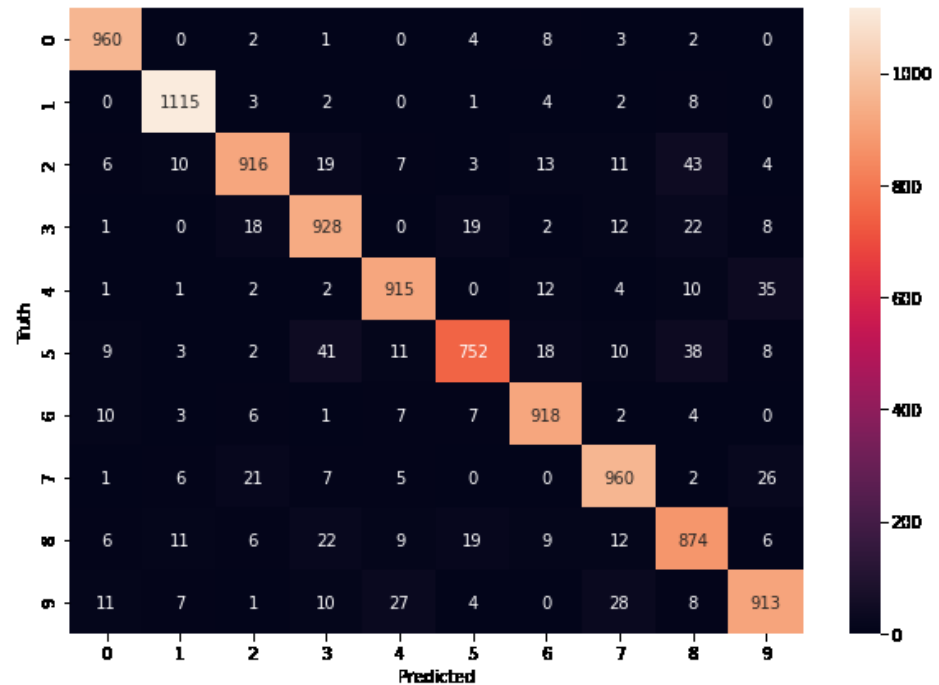
[7, 2, 1, 0, 4]

## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Simple Neural Network without hidden layer with Flattened Data

We shall visualize the Confusion Matrix:

```
cm = tf.math.confusion_matrix(labels=y_test,predictions=y_predicted_labels)
import seaborn as sn
plt.figure(figsize = (10,7))
sn.heatmap(cm, annot=True, fmt='d')
plt.xlabel('Predicted')
plt.ylabel('Truth')
```



## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Neural Network with hidden layer having 100 neuron with Flattened Data

```
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation='relu'),
    keras.layers.Dense(10, activation='sigmoid')])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train_flattened, y_train, epochs=5)
```

```
...
Epoch 5/5 1875/1875 [=====] - 4s
2ms/step - loss: 0.0516 - accuracy: 0.9843
```

## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Neural Network with hidden layer having 100 neuron with Flattened Data

```
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation='relu'),
    keras.layers.Dense(10, activation='sigmoid')])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train_flattened, y_train, epochs=5)
```

```
...
Epoch 5/5 1875/1875 [=====] - 4s
2ms/step - loss: 0.0516 - accuracy: 0.9843
```

## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Neural Network with hidden layer having 100 neuron with Flattened Data

```
model.evaluate(X_test_flattened, y_test)
```

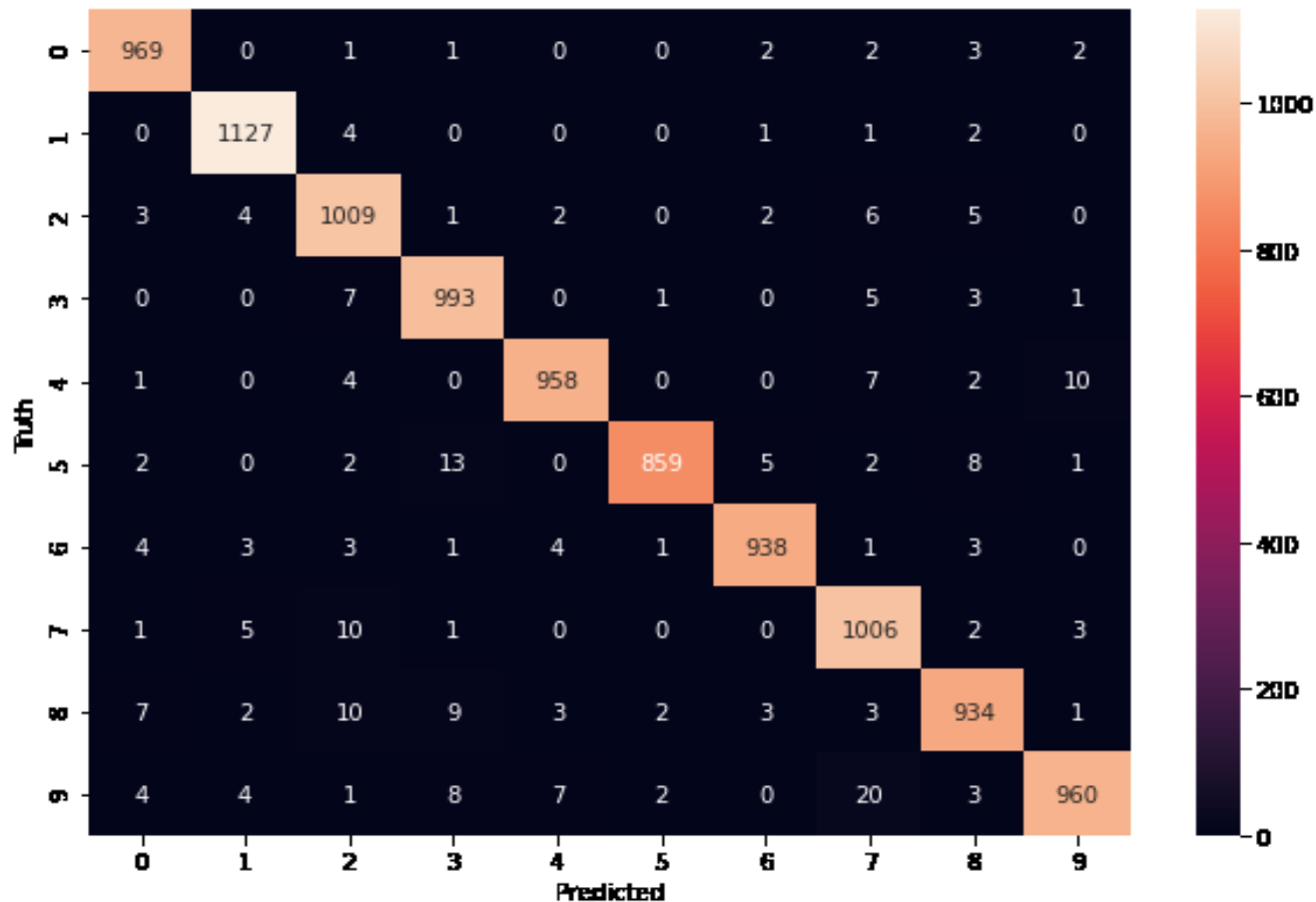
```
313/313 [=====] - 1s 1ms/step - loss: 0.0785 -  
accuracy: 0.9753 [0.07848526537418365, 0.9753000140190125]
```

Thus by adding a hidden layer the accuracy of the neural network has increased from 92.51% to 97.53%.

```
y_predicted = model.predict(X_test_flattened)  
y_predicted_labels = [np.argmax(i) for i in y_predicted]  
cm = tf.math.confusion_matrix(labels=y_test, predictions=y_predicted_labels)  
plt.figure(figsize = (10,7))  
sn.heatmap(cm, annot=True, fmt='d')  
plt.xlabel('Predicted')  
plt.ylabel('Truth')
```

**HANDWRITTEN DIGIT RECOGNITION PROBLEM**

Neural Network with hidden layer having 100 neuron with Flattened Data



## HANDWRITTEN DIGIT RECOGNITION PROBLEM

### Neural Network with hidden layers having 100 neuron & using Flatten layer

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(10, activation='sigmoid')])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train, y_train, epochs=5)
```

```
...
Epoch 5/5 1875/1875 [=====] - 4s
2ms/step - loss: 0.0517 - accuracy: 0.9839
```

## HANDWRITTEN DIGIT RECOGNITION PROBLEM

**Neural Network with hidden layers having 100 neuron & using Flatten layer**

`model.evaluate(X_test, y_test)`

```
313/313 [=====] - 0s 1ms/step - loss: 0.0794 -  
accuracy: 0.9775 [0.07937731593847275, 0.9775000214576721]
```

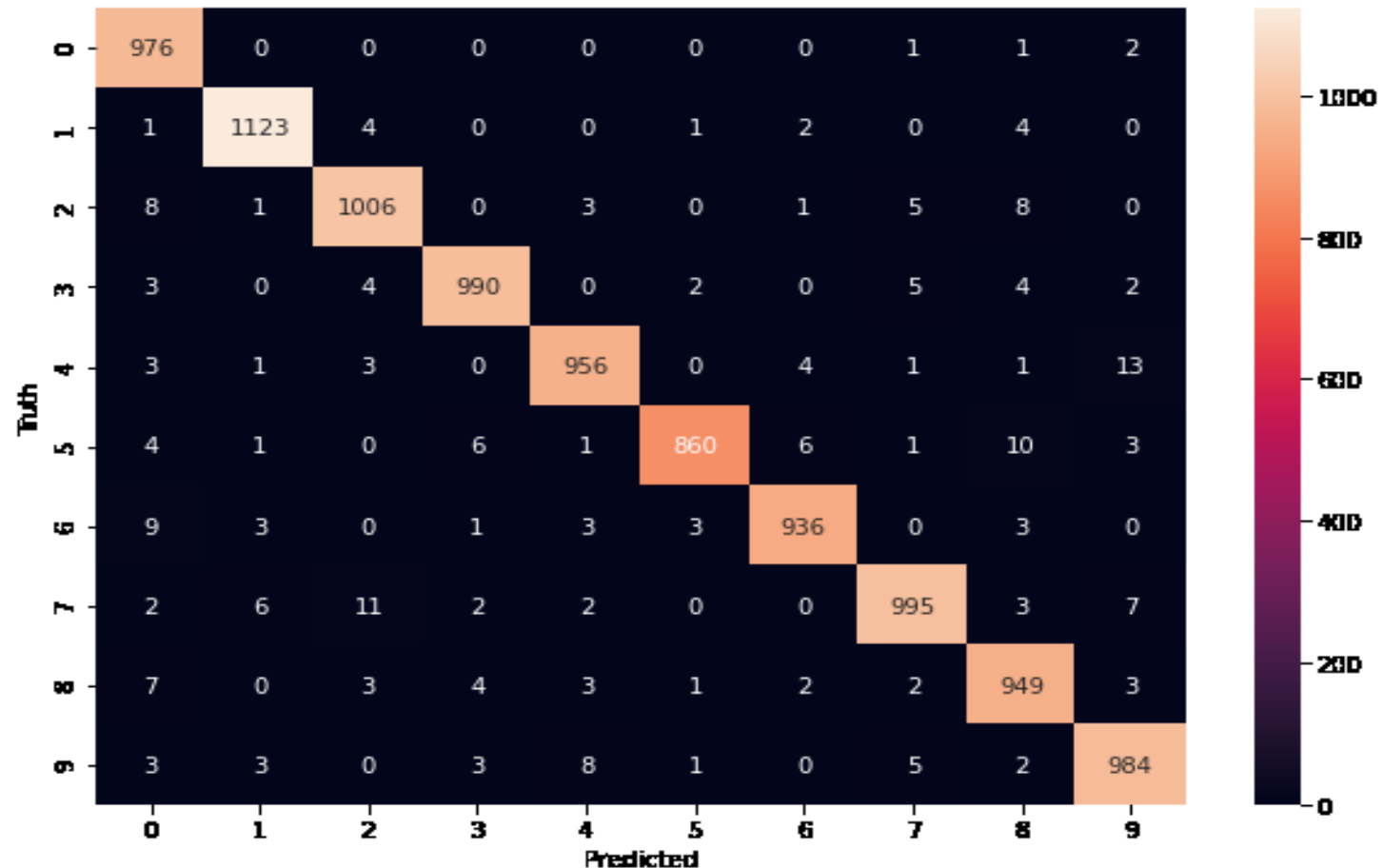
There is not a significant change in the accuracy as compared to the previous model but the advantage is that we need not flatten the data outside the neural network.

```
y_predicted = model.predict(X_test)  
y_predicted_labels = [np.argmax(i) for i in y_predicted]  
cm = tf.math.confusion_matrix(labels=y_test, predictions=y_predicted_labels)  
plt.figure(figsize = (10,7))  
sn.heatmap(cm, annot=True, fmt='d')  
plt.xlabel('Predicted')  
plt.ylabel('Truth')
```



**HANDWRITTEN DIGIT RECOGNITION PROBLEM**

Neural Network with hidden layers having 100 neuron & using Flatten layer



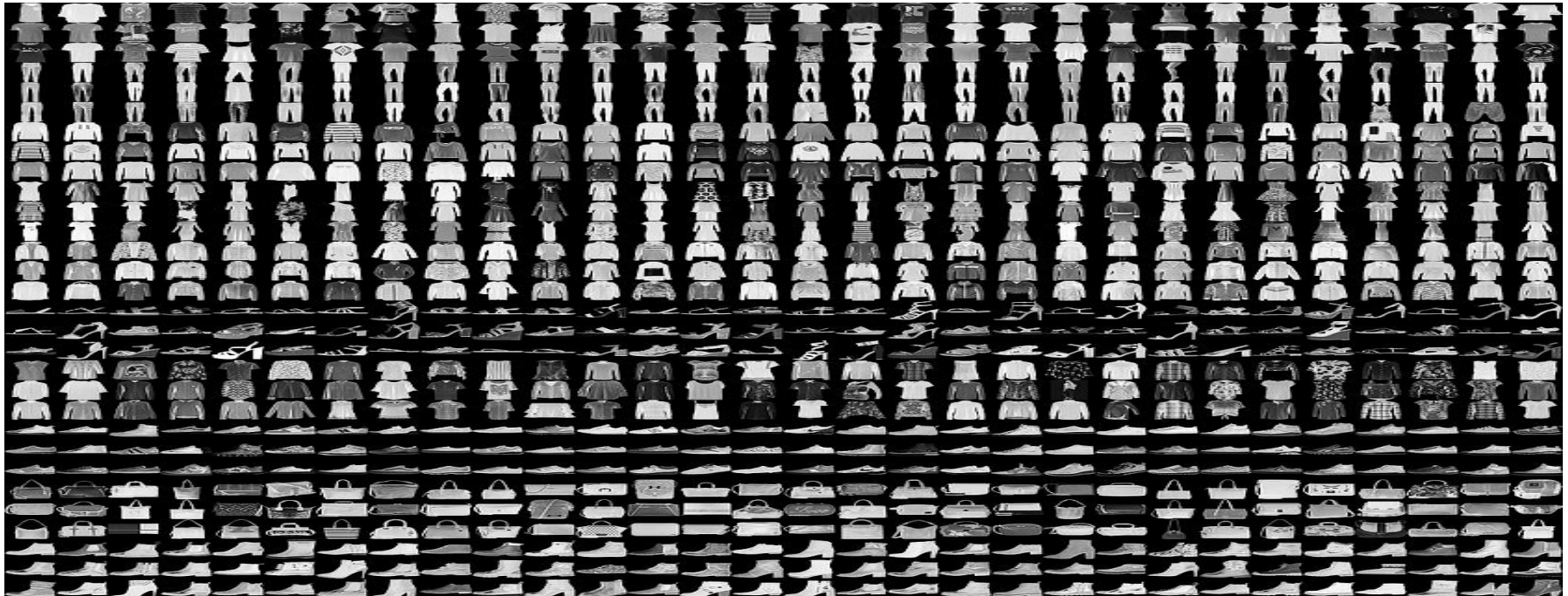
## FASHION MNIST PROBLEM

- The digit recognition problem is one of the classic problems that has been used in the Machine Learning world for quite sometime is the MNIST problem. The objective is to identify the digit based on image. But MNIST is not very great problem because we come up with great accuracy even if we are looking at few pixels in the image. So, another common example problem against which we test algorithms is Fashion-MNIST.
- Fashion-MNIST is a dataset of Zalando's fashion article images —consisting of a **training set of 60,000** examples and a **test set of 10,000** examples. Each instance is a 28x28 greyscale image, associated with a label.
- The objective is to identify (predict) different fashion products from the given images using various best possible Machine Learning Models (Algorithms) and compare their results (performance measures/scores) to arrive at the best ML model

## FASHION MNIST PROBLEM

### CATEGORIES OF PRODUCTS

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



## FASHION MNIST PROBLEM

### LOAD THE DATA

```
fm = tf.keras.datasets.fashion_mnist  
(trainX, trainy), (testX, testy) = fm.load_data()
```

### DATA SIZE

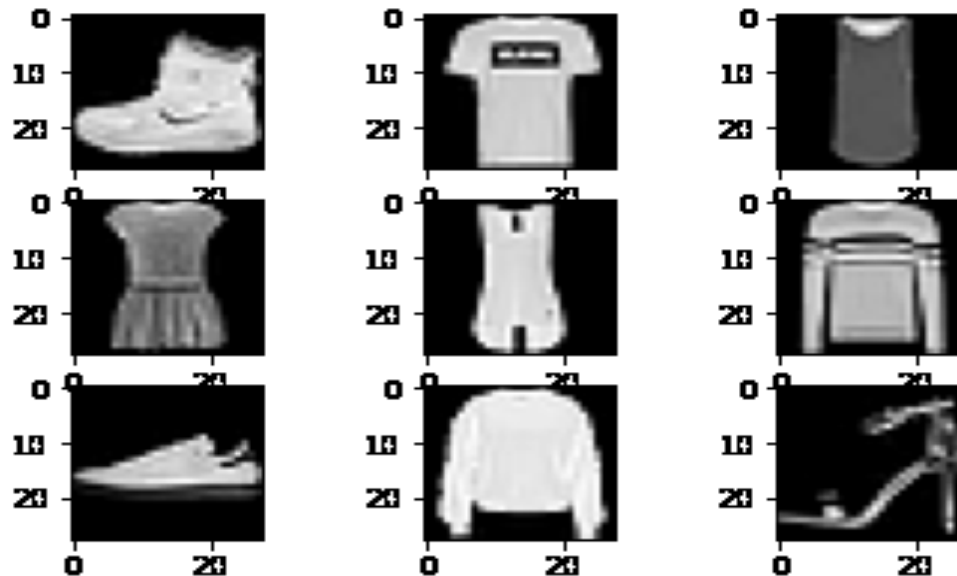
```
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))  
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
```

Train: X=(60000, 28, 28), y=(60000,) Test: X=(10000, 28, 28), y=(10000,)

## FASHION MNIST PROBLEM

### VISUALIZATION OF FEW INPUT DATA

```
class_labels = ["T-shirt/ top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker",  
               "Bag",           "Ankle boot"]  
for i in range(9):  
    pyplot.subplot(330 + 1 + i)  
    pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray'))  
pyplot.show()
```



## FASHION MNIST PROBLEM

### BUILDING THE SEQUENTIAL MODEL AND ADD LAYERS INTO IT

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Activation

model = Sequential()
model.add(Flatten(input_shape=[28, 28]))
model.add(Dense(100, activation="relu"))
model.add(Dense(10, activation="softmax"))
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])
model.fit(trainX, trainy, epochs = 10)
```

```
...
Epoch 10/10 1875/1875 [=====] - 4s 2ms/step - loss: 0.5266 -
accuracy: 0.8237
```

## FASHION MNIST PROBLEM

### TESTING MODEL ACCURACY

```
model.evaluate(testX, testy)
```

```
313/313 [=====] - 1s 1ms/step - loss: 0.5724 - accuracy:  
0.8132 [0.5723907947540283, 0.8131999969482422]
```

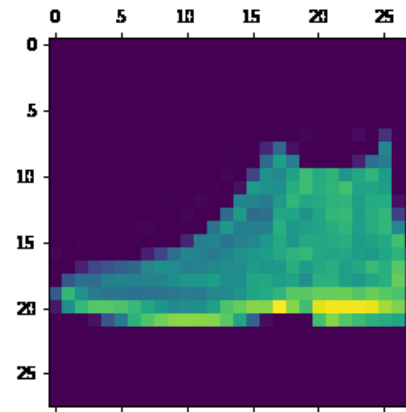
Above shows **accuracy score** of **81.31%**.

```
plt.matshow(testX[0])  
yp = model.predict(testX)  
yp_labels = [np.argmax(i) for i in yp]
```

```
np.argmax(yp[0])
```

9

```
class_labels[np.argmax(yp[0])]
```



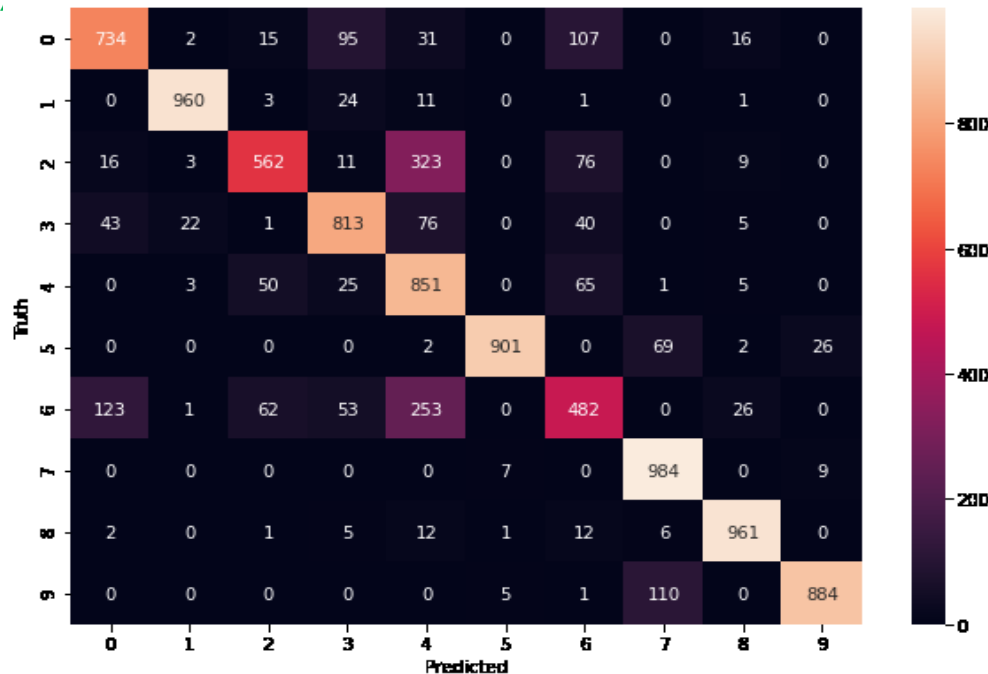
' Ankle Boot '

So our model gives accurate prediction for the 1<sup>st</sup> test data.

## FASHION MNIST PROBLEM

### VISUALIZING THE CONFUSION METRIX

```
cm = tf.math.confusion_matrix(labels=testy, predictions=yp_labels)
plt.figure(figsize = (10,7))
sn.heatmap(cm, annot=True, fmt='d')
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

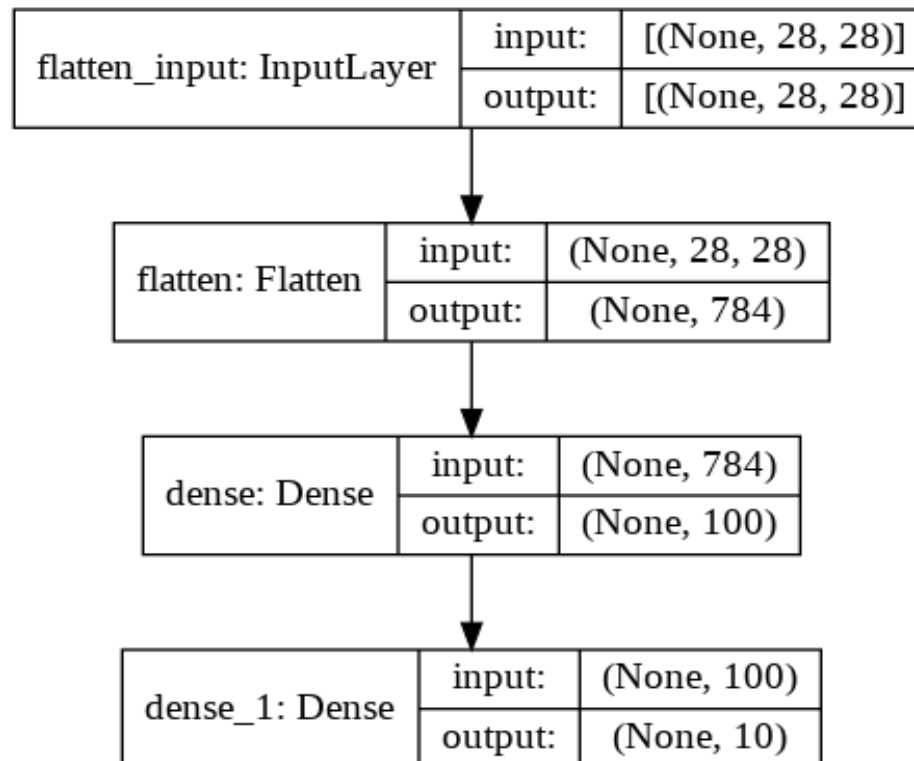




## FASHION MNIST PROBLEM

### VISUALIZING THE NEURAL NETWORK MODEL

```
from keras.utils.vis_utils import plot_model  
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```



## FASHION MNIST PROBLEM

### APPLYING CONVOLUTIONAL NEURAL NETWORK ON OUR DATASET

#### IMPORT THE NECESSARY PACKAGES

```
from keras.layers import Conv2D  
from keras.layers import MaxPooling2D  
from keras.layers import Dense  
from keras.layers import Flatten  
from keras.optimizers import SGD
```

## FASHION MNIST PROBLEM

### BUILDING CNN

```
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', input_shape
                    =(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

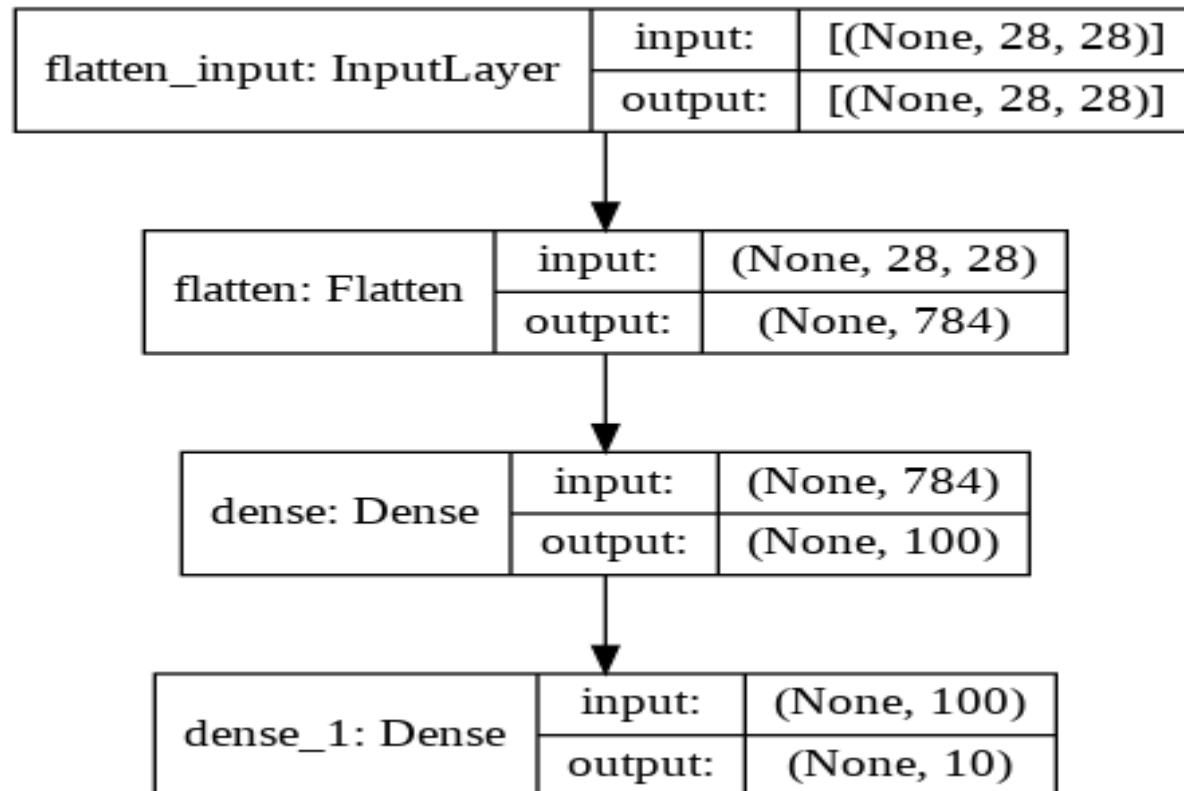
model.fit(trainX, trainy, epochs = 10, batch_size=32, verbose=0)
model.evaluate(testX, testy)
```

```
313/313 [=====] - 0s 1ms/step - loss: 0.6233 - accuracy:
0.8119 [0.6233423352241516, 0.8119000196456909]
```

## FASHION MNIST PROBLEM

### VISUALIZING THE CONVOLUTIONAL NEURAL NETWORK MODEL

```
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```



---

*Thank You*