JSPM's

RAJARSHI SHAHU COLLEGE OF ENGINEERING

TATHAWADE, PUNE-33

(AN AUTONOMOUS INSTITUTE AFFILIATED TO SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE)

DEPARTMENT OF COMPUTER ENGINEERING

LAB MANUAL

OF

Deep Learning Lab

<u>**CONTENTS**</u>

JSPM’s

RAJARSHI SHAHU COLLEGE OF ENGINEERING

TATHAWADE, PUNE-33

(AN AUTONOMOUS INSTITUTE AFFILIATED TO SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE)

DEPARTMENT OF COMPUTER ENGINEERING

Vision

To create quality computer professionals through an excellent academic environment.

Mission

1. To empower students with the fundamentals of Computer Engineering for being successful professionals.
2. To motivate the students for higher studies, research, and entrepreneurship by imparting quality education.
3. To create social awareness among the students.

PEOs

1. Graduate shall have successful professional careers, lead and manage teams.
2. Graduate shall exhibit disciplinary skills to resolve real life problems.
3. Graduate shall evolve as professionals or researchers and continue to learn emerging technologies.

PROGRAM OUTCOMES

Engineering Graduates will be able to:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOME

A graduate of the Computer Engineering Program will demonstrate-

PSO1: Domain Specialization - The ability to understand, analyze and develop computer programs related to algorithms, system software, multimedia, web design, data science, and networking for efficient design of computer-based systems.

PSO2: Problem-Solving Skills - Applying standard practices and strategies in software project development using open-ended programming environments to deliver advanced computing systems.

PSO3: Professional Career and Entrepreneurship -The ability to employ modern computer languages, operating environments, and platforms in creating innovative career paths to be an entrepreneur.

Course Outcomes and its mapping with POs and PSOs

CO1	Describe the working of single layer perceptron.	BL 2
CO2	Explain the working of multilayer layer perceptron.	BL 2
CO3:	Comprehend the fundamental concepts of deep neural network and hyperparameter tuning.	BL 2
CO4:	Apply concepts of CNN,RNN and its variant to solve real life problems.	BL 3
CO5:.	Summarize different applications of deep learning.	BL 2

Lab Plan

Lect. No	**Lab Assignment**	**CO**	**Teaching Aids**	**Planned Date**	**Actual Date**
1	Implement logic gates using single layer perceptron	CO1	Demonstration, Github, Colab,	25/7/22 10/8/22	25/7/22 10/8/22
2	Implement a two class neural network with a hidden layer on any standard dataset.	CO2	Demonstration	16/8/22 25/8/22	16/8/22 25/8/22
3	Using CNN and MNIST dataset, perform digit classification	CO3	Demonstration	27/8/22 10/9/22	
4	Develop an Autoencoder for handwritten digits dataset MNIST.	CO4	Demonstration	10/9/22 25/9/22	
5	Using IMDB review dataset, perform sentiment classification using LSTM and BiLSTM and compare result	CO 4	Demonstration	25/9/22 1/10/22	
6	Using IMDB review dataset, perform sentiment classification using RNN.	CO 5	Demonstration	1/10/22 15/10/22	

LAB ASSIGNMENT NO:1

Aim: Implement logic gates using single layer perceptron

Mapped CO 1: Describe the working of single layer perceptron.

Learning Objectives:

To understand the theory and implementation of single layer Perceptron

Problem statement: Implement logic gates using single layer perceptron

Software and Hardware Requirement

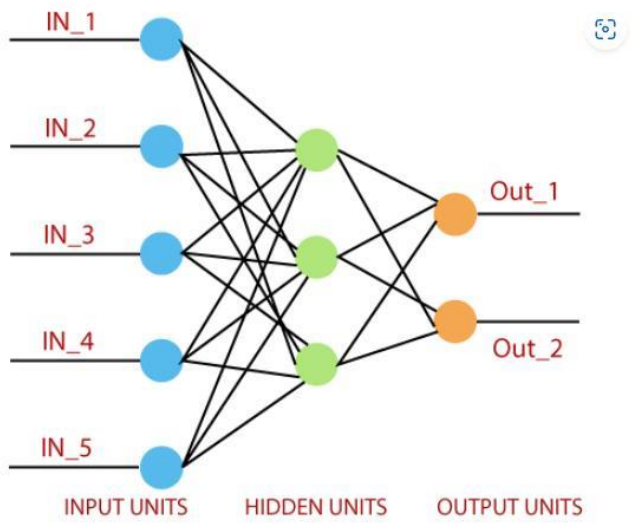
1. 64 bit machine

2. Windows 8 / Open Source Operating System.

3. Google Colab, Notebook Jupyter Notebook

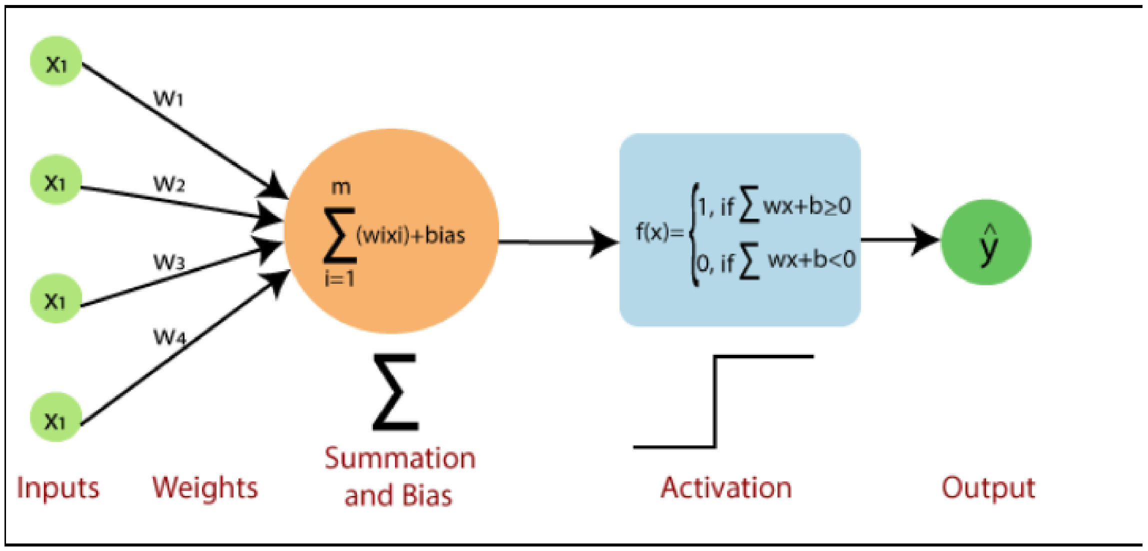
Theory:

A perceptron is a neural network unit that does a precise computation to detect features in the input data. Perceptron is mainly used to classify the data into two parts. Therefore, it is also known as Linear Binary Classifier.



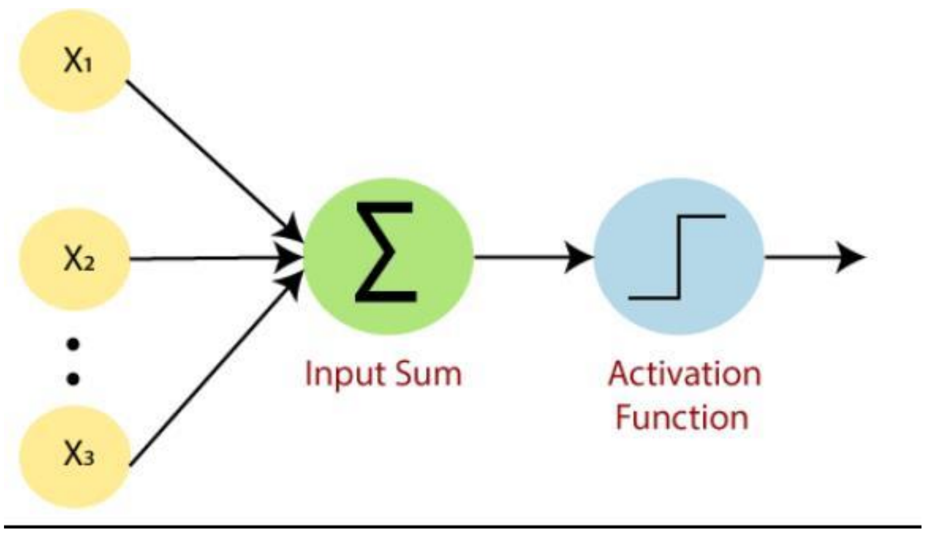
erceptron uses the step function that returns +1 if the weighted sum of its input 0 and -1.

The activation function is used to map the input between the required value like (0, 1) or (-1, 1).


X1
X1
X1
X1
W1
W2
W3
W4
$\sum_{i=1}^{m}(wixi)+bias$
$\sum$
$f(x)=\begin{cases}1, \text{if} \sum wx+b\geq 0\\ 0, \text{if} \sum wx+b<0\end{cases}$
$\hat{y}$
Inputs
Weights
Summation and Bias
Activation
Output

The perceptron consists of 4 parts.

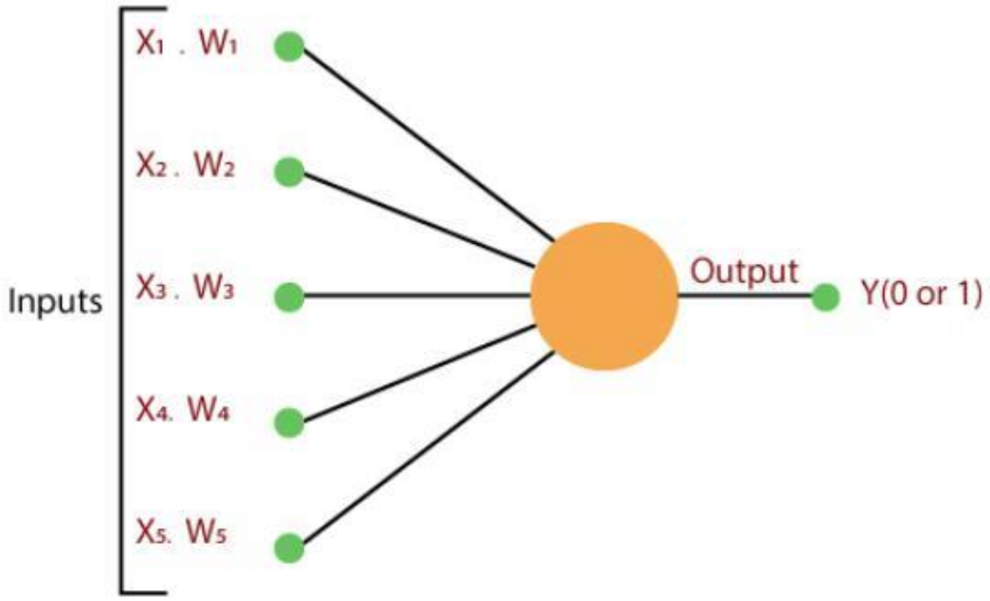
- **Input value or One input layer:** The input layer of the perceptron is made of artificial input neurons and takes the initial data into the system for further processing.
- **Weights and Bias:**
  Weight: It represents the dimension or strength of the connection between units. If the weight to node 1 to node 2 has a higher quantity, then neuron 1 has a more considerable influence on the neuron.
  Bias: It is the same as the intercept added in a linear equation. It is an additional parameter which task is to modify the output along with the weighted sum of the input to the other neuron.
- **Net sum:** It calculates the total sum.
- **Activation Function:** A neuron can be activated or not, is determined by an activation function. The activation function calculates a weighted sum and further adding bias with it to give the result.



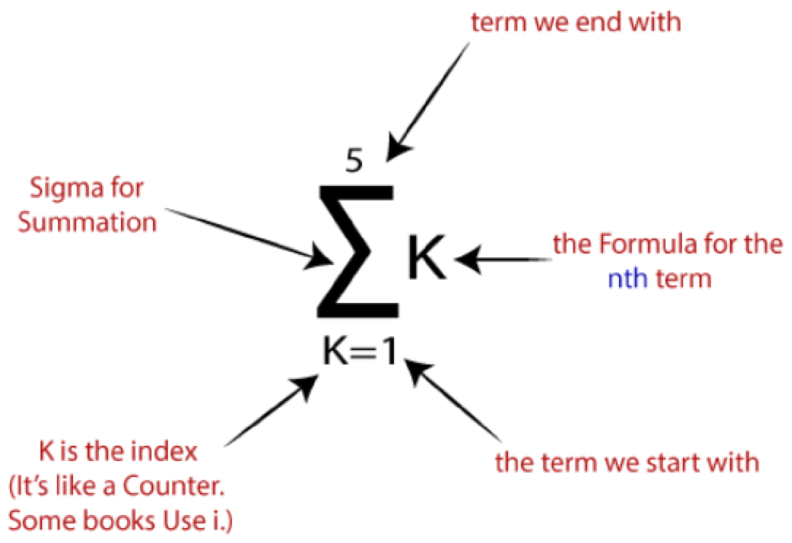
How does it work?

The perceptron works on these simple steps which are given below:

a. In the first step, all the inputs x are multiplied with their weights **w**.



b. In this step, add all the increased values and call them the **Weighted sum**.


term we end with
5
Sigma for Summation
K
the Formula for the nth term
K=1
K is the index (It's like a Counter. Some books Use i.)
the term we start with

Implementation of Perceptron Algorithm for OR Logic Gate with 2-bit Binary Input

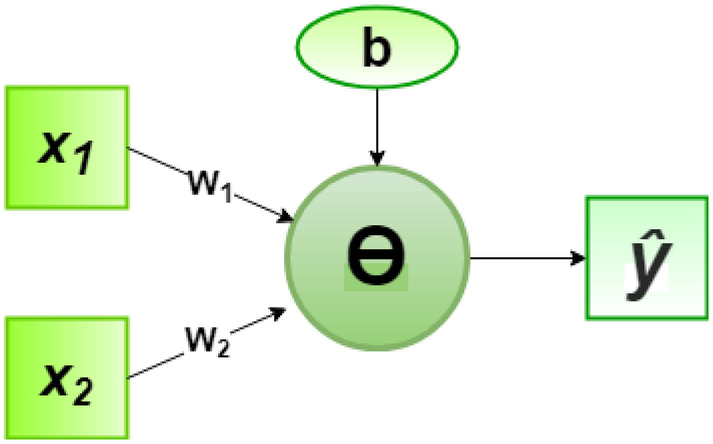
In the field of Machine Learning, the Perceptron is a Supervised Learning Algorithm for binary classifiers. The Perceptron Model implements the following function:

$$\begin{aligned}\hat{y} &= \Theta\left(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b\right)\\ &= \Theta(\mathbf{w}\cdot\mathbf{x} + b)\\ \text{where } \Theta(v) &= \begin{cases} 1 & \text{if } v \geqslant 0 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

For a particular choice of the weight vector and bias parameter , the model predicts output for the corresponding input vector .

OR logical function truth table for ***2-bit binary variables***,

x1	x2	Y
0	0	0
0	1	1
1	0	1
1	1	1



For the implementation, considered weight parameters are   and the bias parameter is   .

Python Implementation:

```
# importing Python library

import numpy as np



# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1
    else:
        return 0


# design Perceptron Model
def perceptronModel(x, w, b):
    v = np.dot(w, x) + b
    y = unitStep(v)
    return y


# OR Logic Function
# w1 = 1, w2 = 1, b = -0.5
def OR_logicFunction(x):
    w = np.array([1, 1])
    b = -0.5
    return perceptronModel(x, w, b)


# testing the Perceptron Model
test1 = np.array([0, 1])
test2 = np.array([1, 1])
test3 = np.array([0, 0])
test4 = np.array([1, 0])
```

```
print("OR({}, {}) = {}".format(0, 1, OR_logicFunction(test1)))

print("OR({}, {}) = {}".format(1, 1, OR_logicFunction(test2)))

print("OR({}, {}) = {}".format(0, 0, OR_logicFunction(test3)))

print("OR({}, {}) = {}".format(1, 0, OR_logicFunction(test4)))
```

Output:

```
OR(0, 1) = 1

OR(1, 1) = 1

OR(0, 0) = 0

OR(1, 0) = 1
```

Here, the model predicted output for each of the test inputs are exactly matched with the OR logic gate conventional output (Y) according to the truth table for 2-bit binary input.

CONCLUSION:

Understood the theory and implementation of logic gates single layer Perceptron

LAB ASSIGNMENT NO:2

Aim: Implement a two class neural network with a hidden layer on any standard dataset.

Mapped CO 1: Explain the working of multilayer layer perceptron.

Learning Objectives:

To understand the theory and implementation of multi layer Perceptron

Problem statement: Implement a two class neural network with a hidden layer on any standard dataset.

Software and Hardware Requirement

1. 64 bit machine

2. Windows 8 / Open Source Operating System.

3. Google Colab, Notebook Jupyter Notebook

Theory:

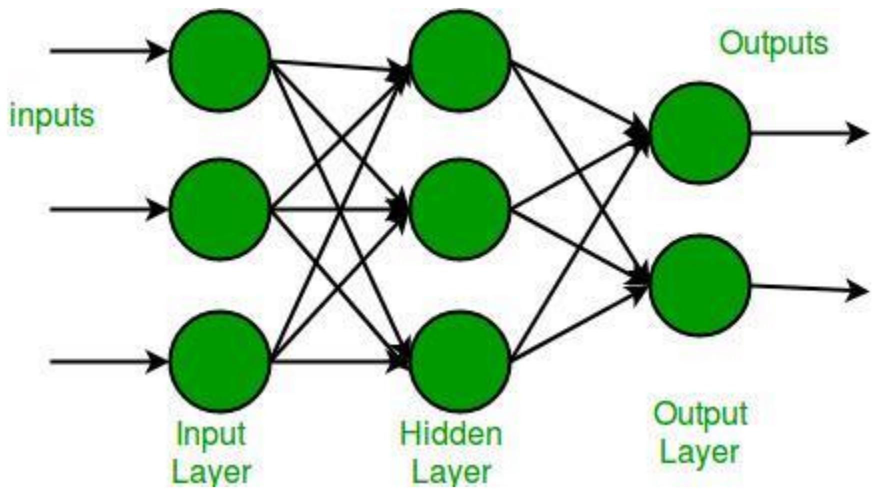
Multi-layer Perceptron

Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

A gentle introduction to **neural networks and TensorFlow** can be found here:

- Neural Networks
- Introduction to TensorFlow

A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.



In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

Every node in the multi-layer perception uses a sigmoid activation function. The sigmoid activation function takes real values as input and converts them to numbers between 0 and 1 using the sigmoid formula.

Now that we are done with the theory part of multi-layer perception, let’s go ahead and implement some code in **python** using the **TensorFlow** library.

Stepwise Implementation

Step 1: Import the necessary libraries.

Python3

```
# importing modules

import tensorflow as tf

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Flatten

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Activation
```

```
import matplotlib.pyplot as plt
```

Step 2: Download the dataset.
TensorFlow allows us to read the MNIST dataset and we can load it directly in the program as a train and test dataset.

Python3

```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data()
```

Output:
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

11493376/11490434 [==============================] – 2s 0us/step

Step 3: Now we will convert the pixels into floating-point values.

Python3

```
# Cast the records into float values

x_train = x_train.astype('float32')

x_test = x_test.astype('float32')



# normalize image pixel values by dividing

# by 255 gray_scale

= 255 x_train /=

gray_scale x_test /=

gray_scale
```

We are converting the pixel values into floating-point values to make the predictions. Changing the numbers into **grayscale** values will be beneficial as the values become small and the computation becomes easier and faster. As the pixel values range from

0 to 256, apart from 0 the range is 255. So dividing all the values by 255 will convert it to range from 0 to 1

Step 4: Understand the structure of the dataset

Python3

```
print("Feature matrix:", x_train.shape)

print("Target matrix:", x_test.shape)

print("Feature matrix:", y_train.shape)

print("Target matrix:", y_test.shape)
```

Output:

```
Feature matrix: (60000, 28, 28)
Target matrix: (10000, 28, 28)
Feature matrix: (60000,)
Target matrix: (10000,)
```

Thus we get that we have 60,000 records in the training dataset and 10,000 records in the test dataset and Every image in the dataset is of the size 28×28.

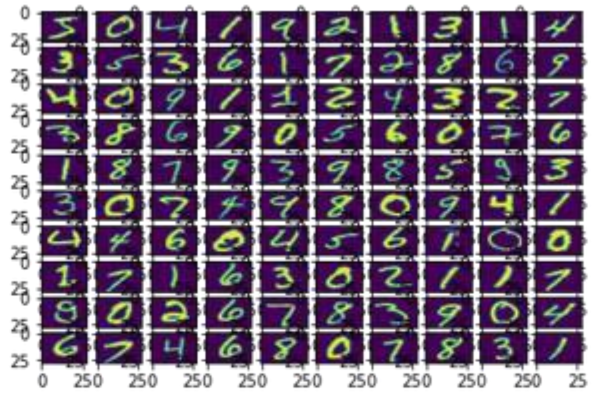
Step 5: Visualize the data.

Python3

```
fig, ax = plt.subplots(10, 10)

k = 0

for i in range(10):

    for j in range(10):

        ax[i][j].imshow(x_train[k].reshape(28, 28),

                        aspect='auto')

        k += 1
```

```
plt.show()
```

Output



Step 6: Form the Input, hidden, and output layers.

Python3

```
model = Sequential([



    # reshape 28 row * 28 column data to 28*28 rows

    Flatten(input_shape=(28, 28)),



      # dense layer 1

    Dense(256, activation='sigmoid'),



    # dense layer 2

    Dense(128, activation='sigmoid'),
        # output layer

     Dense(10, activation='sigmoid'),

])
```

Some important points to note:

- The **Sequential model** allows us to create models layer-by-layer as we need in a multi-layer perceptron and is limited to single-input, single-output stacks of layers.
- **Flatten** flattens the input provided without affecting the batch size. For example, If inputs are shaped (batch_size,) without a feature axis, then flattening adds an extra channel dimension and output shape is (batch_size, 1).
- **Activation** is for using the sigmoid activation function.
- The first two **Dense** layers are used to make a fully connected model and are the hidden layers.
- The **last Dense layer** is the output layer which contains 10 neurons that decide which category the image belongs to.

Step 7: Compile the model.

- Python

```
model.compile(optimizer='adam',

              loss='sparse_categorical_crossentropy',

              metrics=['accuracy'])
```

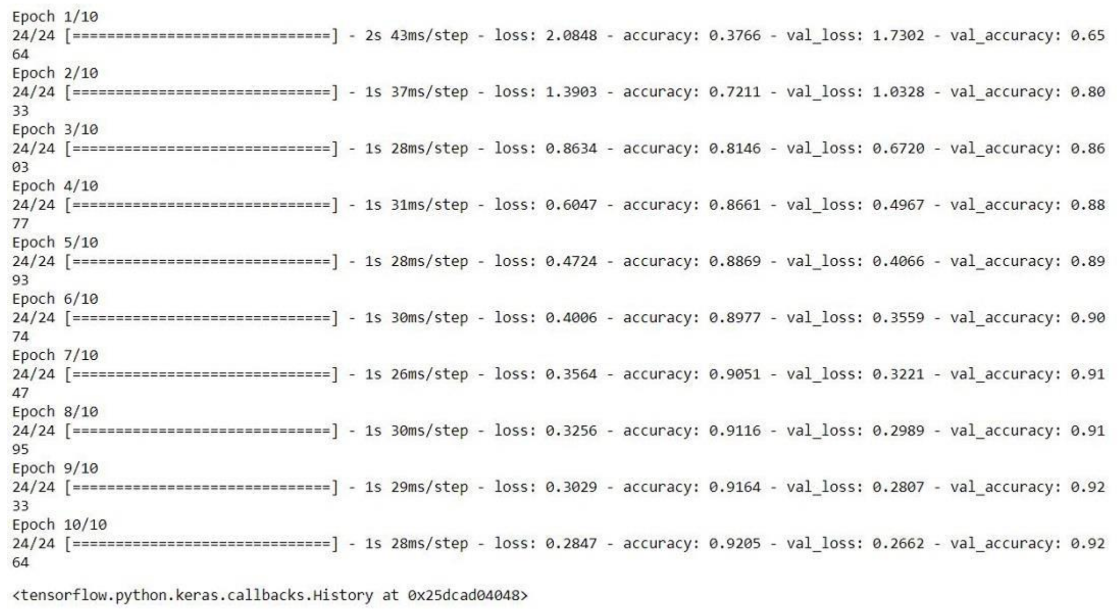
Compile function is used here that involves the use of loss, optimizers, and metrics. Here loss function used is **sparse_categorical_crossentropy**, optimizer used is **adam**.

Step 8: Fit the model.

- Python3

```
model.fit(x_train, y_train, epochs=10,

          batch_size=2000,

          validation_split=0.2)
```

Output:



```
Epoch 1/10
24/24 [==============================] - 2s 43ms/step - loss: 2.0848 - accuracy: 0.3766 - val_loss: 1.7302 - val_accuracy: 0.65
64
Epoch 2/10
24/24 [==============================] - 1s 37ms/step - loss: 1.3903 - accuracy: 0.7211 - val_loss: 1.0328 - val_accuracy: 0.80
33
Epoch 3/10
24/24 [==============================] - 1s 28ms/step - loss: 0.8634 - accuracy: 0.8146 - val_loss: 0.6720 - val_accuracy: 0.86
03
Epoch 4/10
24/24 [==============================] - 1s 31ms/step - loss: 0.6047 - accuracy: 0.8661 - val_loss: 0.4967 - val_accuracy: 0.88
77
Epoch 5/10
24/24 [==============================] - 1s 28ms/step - loss: 0.4724 - accuracy: 0.8869 - val_loss: 0.4066 - val_accuracy: 0.89
93
Epoch 6/10
24/24 [==============================] - 1s 30ms/step - loss: 0.4006 - accuracy: 0.8977 - val_loss: 0.3559 - val_accuracy: 0.90
74
Epoch 7/10
24/24 [==============================] - 1s 26ms/step - loss: 0.3564 - accuracy: 0.9051 - val_loss: 0.3221 - val_accuracy: 0.91
47
Epoch 8/10
24/24 [==============================] - 1s 30ms/step - loss: 0.3256 - accuracy: 0.9116 - val_loss: 0.2989 - val_accuracy: 0.91
95
Epoch 9/10
24/24 [==============================] - 1s 29ms/step - loss: 0.3029 - accuracy: 0.9164 - val_loss: 0.2807 - val_accuracy: 0.92
33
Epoch 10/10
24/24 [==============================] - 1s 28ms/step - loss: 0.2847 - accuracy: 0.9205 - val_loss: 0.2662 - val_accuracy: 0.92
64

<tensorflow.python.keras.callbacks.History at 0x25dcad04048>
```

Some important points to note:

- **Epochs** tell us the number of times the model will be trained in forwarding and backward passes.
- **Batch Size** represents the number of samples, If it’s unspecified, batch_size will default to 32.
- **Validation Split** is a float value between 0 and 1. The model will set apart this fraction of the training data to evaluate the loss and any model metrics at the end of each epoch. (The model will not be trained on this data)

Step 9: Find Accuracy of the model.

- Python3

```
results = model.evaluate(x_test,  y_test, verbose = 0)

print('test loss, test acc:', results)
```

Output:

```
test loss, test acc: [0.27210235595703125, 0.9223999977111816]
```

We got the accuracy of our model 92% by using **model.evaluate()** on the test samples.

Stepwise Implementation:

Stage 1: Import the fundamental libraries.

Stage 2: Download the dataset. TensorFlow permits us to peruse the MNIST dataset and we can stack it straightforwardly in the program as a train and test dataset.

Stage 3: Now we will change over the pixels into drifting point values. Stage 4: Understand the design of the dataset

Stage 5: Visualize the information.

Stage 6: Form the Input, stowed away, and yield layers.

Stage 7: Compile the model.

Stage 8: Fit the model.

Stage 9: Find Accuracy of the model.

CONCLUSION :

Understood the theory and how the implementation of multi layer perceptron

LAB ASSIGNMENT NO:3

Aim: Using CNN and MNIST dataset, perform digit classification

Mapped CO 1: Apply concepts of CNN,RNN and its variant to solve real life problems.

Learning Objectives:

To understand the theory and implementation of multi layer Perceptron

Problem statement:

Implement CNN and MNIST dataset, perform digit classification

Software and Hardware Requirement

1. 64 bit machine
2. Windows 8 / Open Source Operating System.
3. Google Colab, Notebook Jupyter Notebook

Theory: Convolutional Neural Networks (CNNs) are the current state-of-art architecture for image classification task. Whether it is facial recognition, self driving cars or object detection, CNNs are being used everywhere. In this post, a simple 2-D Convolutional Neural Network (CNN) model is designed using keras with tensorflow backend for the well known MNIST digit recognition task. The whole work flow can be:

1. Preparing the data
2. Building and compiling of the model
3. Training and evaluating the model
4. Saving the model to disk for reuse

1. **Preparing the data**

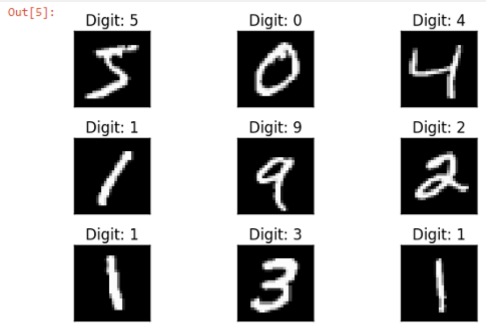
   The data set used here is MNIST dataset as mentioned above. The **MNIST database** (Modified National Institute of Standards and Technology database) is a large database of handwritten digits (0 to 9). The database contains 60,000 training images and

10,000 testing images each of size 28x28. The first step is to load the dataset, which can be easily done through the keras api.

```
import keras
from keras.datasets import mnist
#load mnist dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data() #everytime loading data won't be so
easy :)
```

Here X_train contains 60,000 training images' data each of size 28x28 and y_train contains their corresponding labels. Similarly, X_test contains 10,000 testing images' data each of dimension 28x28 and y_test contains their corresponding labels. Let's visualize few data from training to get a better idea about the purpose of the deep learning model.

```
import matplotlib.pyplot as plt
fig = plt.figure() for
i in range(9):
plt.subplot(3,3,i+1)
plt.tight_layout()
  plt.imshow(X_train[i], cmap='gray', interpolation='none')
  plt.title("Digit: {}".format(y_train[i]))
  plt.xticks([])
plt.yticks([])
fig
```



2.

As can be seen here, at left top corner the image of '5' is stored is X_train[0] and y_train[0] contains label '5'. Our deep learning model should be able to only take the handwritten image and predict the actual digit written.

Now, to prepare the data we need some processing on the images like resizing images, normalizing the pixel values etc.

```
#reshaping
#this assumes our data format
#For 3D data, "channels_last" assumes (conv_dim1, conv_dim2, conv_dim3, channels) while
```

```
#"channels_first" assumes (channels, conv_dim1, conv_dim2, conv_dim3).
if k.image_data_format() == 'channels_first':
    X_train = X_train.reshape(X_train.shape[0], 1, img_rows, img_cols)
    X_test = X_test.reshape(X_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
    X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
#more reshaping
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print('X_train shape:', X_train.shape) #X_train shape: (60000, 28, 28, 1)
```

After doing the necessary processing on the image informations, the label data i.e. y_train and y_test need to be converted into categorical formats like label **‘3’** should be converted to a vector **[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]** for model building.

```
import keras
#set number of categories
num_category = 10
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_category)
y_test = keras.utils.to_categorical(y_test, num_category)
```

3. **Building and compiling of the model**

After the data is ready to be fed to the model, we need to define the architecture of the model and compile it with necessary optimizer function, loss function and performance metrics.

The architecture followed here is 2 convolution layers followed by pooling layer, a fully connected layer and softmax layer respectively. Multiple filters are used at each convolution layer, for different types of feature extraction. One intuitive explanation can be if first filter helps in detecting the straight lines in the image, second filter will help in detecting circles and so on. Explanations for technical execution of each layer will be a part of upcoming post. For better understanding of each layer, may refer to *http://cs231n.github.io/convolutional-networks/*

After both maxpooling and fully connected layer, dropout is introduced as regularization in our model to reduce over-fitting problem.

```
##model building
model = Sequential()
#convolutional layer with rectified linear unit activation
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
#32 convolution filters used each of size 3x3
#again
model.add(Conv2D(64, (3, 3), activation='relu'))
#64 convolution filters used each of size 3x3
#choose the best features via pooling
model.add(MaxPooling2D(pool_size=(2, 2)))
#randomly turn neurons on and off to improve convergence
model.add(Dropout(0.25))
#flatten since too many dimensions, we only want a classification output
model.add(Flatten())
#fully connected to get all relevant data
model.add(Dense(128, activation='relu'))
#one more dropout for convergence' sake :)
model.add(Dropout(0.5))
#output a softmax to squash the matrix into output probabilities
model.add(Dense(num_category, activation='softmax'))
```

After the architecture of the model is defined, the model needs to be compiled. Here, we are using categorical_crossentropy loss function as it is a multi-class classification problem. Since all the labels carry similar weight we prefer accuracy as performance metric. A popular gradient descent technique called AdaDelta is used for optimization of the model parameters.

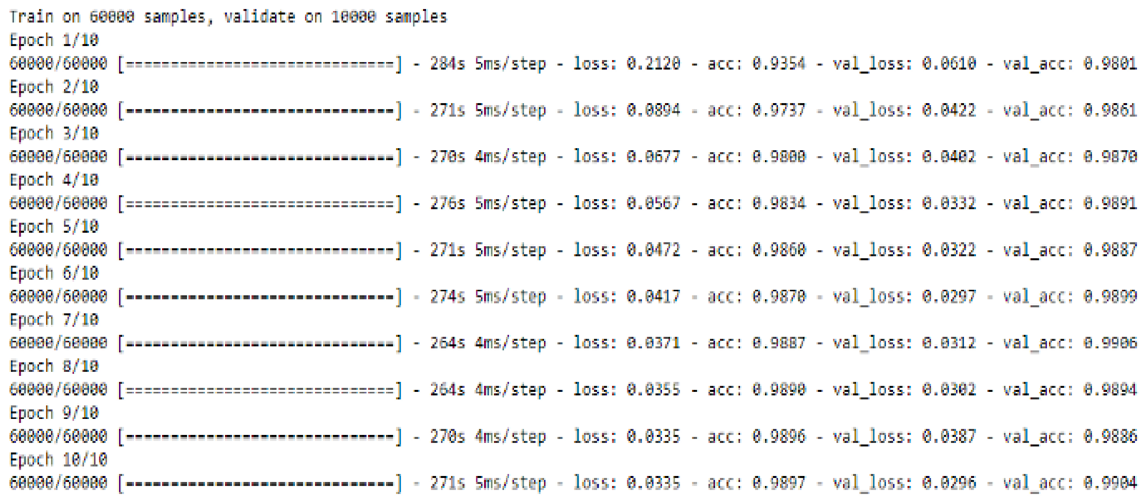
```
#Adaptive learning rate (adaDelta) is a popular form of gradient descent rivaled only by adam and adagrad
#categorical ce since we have multiple classes (10)
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

4. Training and evaluating the model

After the model architecture is defined and compiled, the model needs to be trained with training data to be able to recognize the handwritten digits. Hence we will fit the model with X_train and y_train.

```
batch_size = 128
num_epoch = 10
#model training
model_log = model.fit(X_train, y_train,
          batch_size=batch_size,
          epochs=num_epoch,
          verbose=1,
          validation_data=(X_test, y_test))
```

Here, one epoch means one forward and one backward pass of all the training samples. Batch size implies number of training samples in one forward/backward pass. The training output is:



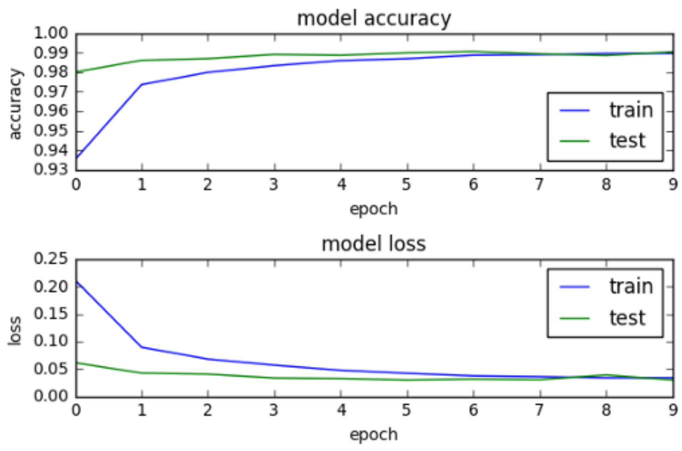
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 284s 5ms/step - loss: 0.2120 - acc: 0.9354 - val_loss: 0.0610 - val_acc: 0.9801
Epoch 2/10
60000/60000 [==============================] - 271s 5ms/step - loss: 0.0894 - acc: 0.9737 - val_loss: 0.0422 - val_acc: 0.9861
Epoch 3/10
60000/60000 [==============================] - 270s 4ms/step - loss: 0.0677 - acc: 0.9800 - val_loss: 0.0402 - val_acc: 0.9870
Epoch 4/10
60000/60000 [==============================] - 276s 5ms/step - loss: 0.0567 - acc: 0.9834 - val_loss: 0.0332 - val_acc: 0.9891
Epoch 5/10
60000/60000 [==============================] - 271s 5ms/step - loss: 0.0472 - acc: 0.9860 - val_loss: 0.0322 - val_acc: 0.9887
Epoch 6/10
60000/60000 [==============================] - 274s 5ms/step - loss: 0.0417 - acc: 0.9870 - val_loss: 0.0297 - val_acc: 0.9899
Epoch 7/10
60000/60000 [==============================] - 264s 4ms/step - loss: 0.0371 - acc: 0.9887 - val_loss: 0.0312 - val_acc: 0.9906
Epoch 8/10
60000/60000 [==============================] - 264s 4ms/step - loss: 0.0355 - acc: 0.9890 - val_loss: 0.0302 - val_acc: 0.9894
Epoch 9/10
60000/60000 [==============================] - 270s 4ms/step - loss: 0.0335 - acc: 0.9896 - val_loss: 0.0387 - val_acc: 0.9886
Epoch 10/10
60000/60000 [==============================] - 271s 5ms/step - loss: 0.0335 - acc: 0.9897 - val_loss: 0.0296 - val_acc: 0.9904
```

Now the trained model needs to be evaluated in terms of performance.

```
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0]) #Test loss: 0.0296396646054
print('Test accuracy:', score[1]) #Test accuracy: 0.9904
```

Test accuracy 99%+ implies the model is trained well for prediction. If we visualize the whole training log, then with more number of epochs the loss and accuracy of the model on training and testing data converged thus making the model a stable one.

```
import os
# plotting the metrics
fig = plt.figure()
plt.subplot(2,1,1)
plt.plot(model_log.history['acc'])
plt.plot(model_log.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')plt.subplot(2,1,2)
plt.plot(model_log.history['loss'])
plt.plot(model_log.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')plt.tight_layout()fig
```



5. **Saving the model to disk for reuse**

Now, the trained model needs to be serialized. The architecture or structure of the model will be stored in a json file and the weights will be stored in hdf5 file format.

```
#Save the model
# serialize model to JSON
model_digit_json = model.to_json()
with open("model_digit.json", "w") as json_file:
    json_file.write(model_digit_json)
# serialize weights to HDF5
model.save_weights("model_digit.h5")
print("Saved model to disk")
```

Hence the saved model can be reused later or easily ported to other environments too. In the upcoming posts, we will see how to deploy this trained model at production.

Stepwise Implementation:

Stage 1: Import the fundamental libraries.

Stage 2: Download the dataset. TensorFlow permits us to peruse the MNIST dataset and we can stack it straightforwardly in the program as a train and test dataset.

Stage 3: Prepare Pixel Data

Stage 4: Define Model

Stage 5: Visualize the information.

Stage 6: Fit the model.

Stage 7: Find Accuracy of the model.

CONCLUSION :

Understood the theory and how the implementation of CNN on MNIST dataset

LAB ASSIGNMENT NO:4

Aim: Develop an Autoencoder for handwritten digits dataset MNIST.

Mapped CO 1: Apply concepts of CNN,RNN and its variant to solve real life problems.

Learning Objectives:

To understand the theory and implementation of multi layer Perceptron , CAE

Problem statement:

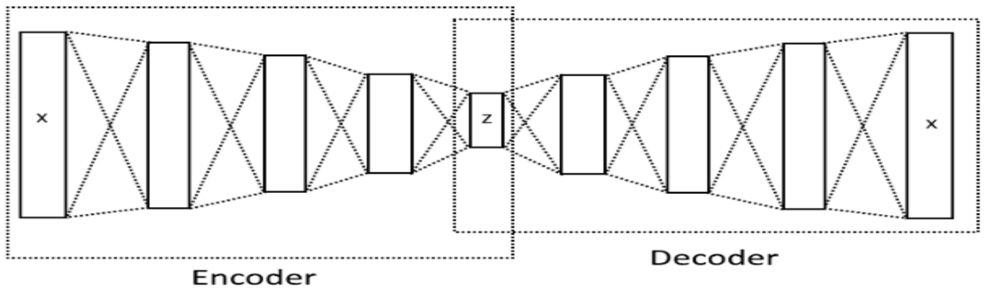
Implement CAE on MNIST dataset, perform digit classification

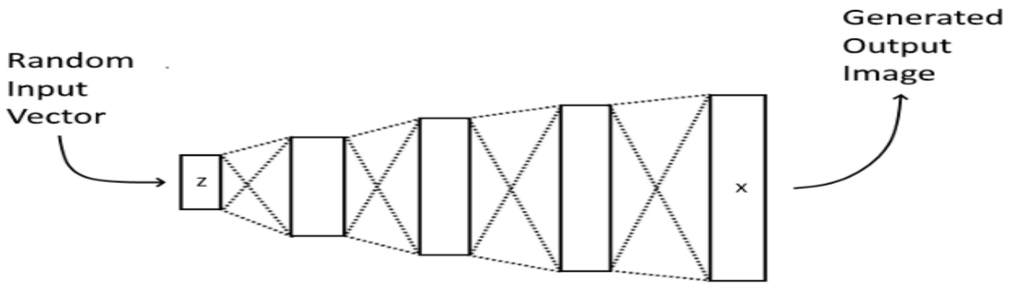
Software and Hardware Requirement

1. 64 bit machine
2. Windows 8 / Open Source Operating System.
3. Google Colab, Notebook Jupyter Notebook

Theory:

Understanding Autoencoders:





Autoencoders are a class of Unsupervised Networks that consist of two major networks: Encoders and Decoders.

An Unsupervised Network is a network that learns patterns from data without any training labels. The network finds its patterns in the data without being told what the patterns should be.

In contrast, there are Supervised Networks wherein the network is trained to return specific outputs when given specific inputs.

The Encoder generally uses a series of Dense and/or Convolutional layers to encode an image into a fixed length vector that represents the image a compact form, while the Decoder uses Dense and/or Convolutional layers to convert the latent representation vector back into that same image or another modified image.

The image above shows an example of a simple autoencoder. In this autoencoder, you can see that the input of size X is compressed into a latent vector of size Z and then decompressed into the same image of size X.

To generate an image, a random input vector is given to the Decoder network. The Decoder network will convert the input vector into a full image.

Creating the Autoencoder:

I recommend using Google Colab to run and train the Autoencoder model.

Installing Tensorflow 2.0

```
#If you have a GPU that supports CUDA
$ pip3 install tensorflow-gpu==2.0.0b1#Otherwise
$ pip3 install tensorflow==2.0.0b1
```

Tensorflow 2.0 has Keras built-in as its high-level API. Keras is accessible through this import:

```
import tensorflow.keras as keras
```

Importing Necessary Modules/Packages

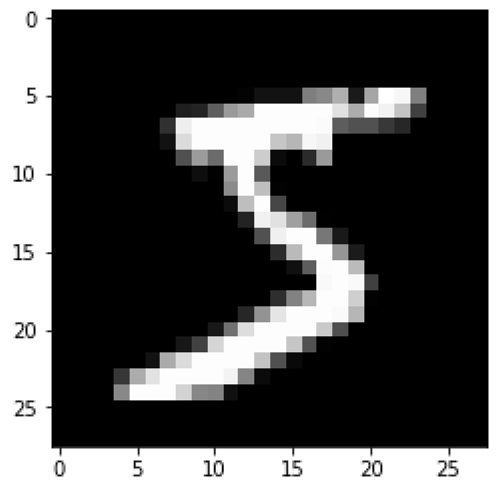
```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Dense, Input, Flatten,\
                                    Reshape, LeakyReLU as LR,\
                                    Activation, Dropout
from tensorflow.keras.models import Model, Sequential
from matplotlib import pyplot as plt
from IPython import display # If using IPython, Colab or Jupyter
import numpy as np
```

Loading MNIST Data

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train/255.0
x_test = x_test/255.0
```

The MNIST dataset is comprised of 70000 28 pixels by 28 pixels images of handwritten digits and 70000 vectors containing information on which digit each one is.

The image training data is scaled from [0, 255] to [0,1] to allow for use of the sigmoid activation function.



Data from x_train[0]

To check our data, we’ll plot the first image in the training dataset.

```
# Plot image data from x_train
plt.imshow(x_train[0], cmap = "gray")
plt.show()
```

Deciding the Latent Size

Latent size is the size of the latent space: the vector holding the information after compression. This value is a crucial hyperparameter. If this value is too small, there won’t be enough data for reconstruction and if the value is too large, overfitting can occur.

I found that a nice, successful latent size was 32 values long.

```
LATENT_SIZE = 32
```

Creating the Encoder

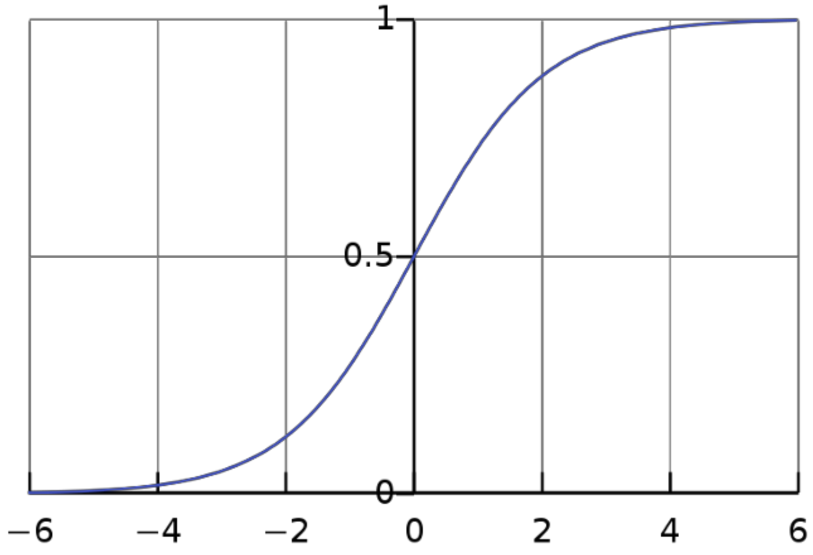
```
encoder = Sequential([
    Flatten(input_shape = (28, 28)),
    Dense(512),
    LR(),
    Dropout(0.5),
    Dense(256),
    LR(),
    Dropout(0.5),
    Dense(128),
    LR(),
    Dropout(0.5),
    Dense(64),
    LR(),
    Dropout(0.5),
    Dense(LATENT_SIZE),
    LR()
])
```

The encoder consists of a series of Dense layers with interstitial Dropout and LeakyReLU layers. The Dense Layers allow for the compression of the 28x28 input tensor down to the latent vector of size 32. The Dropout layers help prevent overfitting and LeakyReLU, being the activation layer, introduces non-linearity into the mix. Dense(LATENT_SIZE) creates the final vector of size 32.

Creating the Decoder

```
decoder = Sequential([
    Dense(64, input_shape = (LATENT_SIZE,)),
    LR(),
    Dropout(0.5),
    Dense(128),
    LR(),
    Dropout(0.5),
    Dense(256),
    LR(),
    Dropout(0.5),
    Dense(512),
    LR(),
    Dropout(0.5),
    Dense(784),
    Activation("sigmoid"),
    Reshape((28, 28))
])
```

The decoder is essentially the same as the encoder but in reverse. The final activation layer is sigmoid, however. The sigmoid activation function output values in the range [0, 1] which fits perfectly with our scaled image data.



Sigmoid Function

Creating the Full Model

To create the full model, the Keras Functional API must be used. The Functional API allows us to string together multiple models.

```
img = Input(shape = (28, 28))
```

This will create a placeholder tensor which we can feed into each network to get the output of the whole model.

```
latent_vector = encoder(img)
output = decoder(latent_vector)
```

The best part about the Keras Functional API is how readable it is. The Keras Functional API allows you to call models directly onto tensors and get the output from that tensor. By calling the encoder model onto the img tensor, I get the latent_vector. The same can be done with the decoder model onto the latent_vector which gives us the output.

```
model = Model(inputs = img, outputs = output)
model.compile("nadam", loss = "binary_crossentropy")
```

To create the model itself, you use the Model class and define what the inputs and outputs of the model are.

To train a model, you must compile it. To compile a model, you have to choose an optimizer and a loss function. For the optimizer, I chose Nadam, which is Nesterov Accelerated Gradient applied to Adaptive Moment Estimation. It is a modified Adam optimizer. For the loss, I chose binary cross-entropy. Binary Cross-Entropy is very commonly used with Autoencoders. Usually, however, binary cross-entropy is used with Binary Classifiers. Additionally, binary cross-entropy can only be used between output values in the range [0, 1].

Training the Model

```
EPOCHS = 60
```

The value EPOCHS is a hyperparameter set to 60. Generally, the more epochs the better, at least until the model plateaus out.

```
#Only do plotting if you have IPython, Jupyter, or using Colab
```

Repeatedly plotting is really only recommended if you are using IPython, Jupyter, or Colab so that the matplotlib plots are inline and not repeatedly creating individual plots.

```
for epoch in range(EPOCHS):
    fig, axs = plt.subplots(4, 4)
    rand = x_test[np.random.randint(0, 10000, 16)].reshape((4, 4, 1, 28, 28))
    display.clear_output() # If you imported display from IPython

    for i in range(4):
        for j in range(4):
            axs[i, j].imshow(model.predict(rand[i, j])[0], cmap = "gray")
            axs[i, j].axis("off")

    plt.subplots_adjust(wspace = 0, hspace = 0)
    plt.show()
    print("-----------", "EPOCH", epoch, "-----------")
    model.fit(x_train, x_train)
```

First, we create plots with 4 rows and 4 columns of subplots and choose 16 random testing data images to check how well the network performs.

Next, we clear the screen (only works on IPython, Jupyter, and Colab) and plot the predictions by the model on the random testing images.

Finally, we train the model. To train the model we simply call model.fit on the training image data. Remember how the autoencoder’s goal is to take the input data, compress it, decompress it, and then output a copy of the input data? Well, that means that the input and the target output are both the training image data.

As you can see, these generated images are pretty good. The biggest problem with the images, however, is with the blurriness. Many of these problems can be fixed with other types of Generative Networks or even other types of Autoencoders.

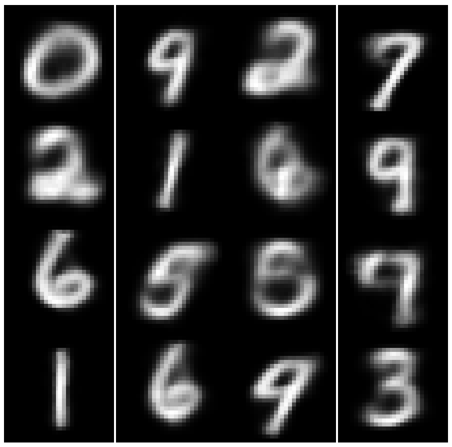
Full Code

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Dense, Input, Flatten,\
                                    Reshape, LeakyReLU as LR,\
                                    Activation, Dropout
from tensorflow.keras.models import Model, Sequential
from matplotlib import pyplot as plt
from IPython import display # If using IPython, Colab or Jupyter
import numpy as np(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train/255.0
x_test = x_test/255.0# Plot image data from x_train
plt.imshow(x_train[0], cmap = "gray")
plt.show()LATENT_SIZE = 32encoder = Sequential([
    Flatten(input_shape = (28, 28)),
    Dense(512),
    LR(),
    Dropout(0.5),
    Dense(256),
    LR(),
    Dropout(0.5),
    Dense(128),
    LR(),
    Dropout(0.5),
    Dense(64),
    LR(),
    Dropout(0.5),
    Dense(LATENT_SIZE),
    LR()
])decoder = Sequential([
    Dense(64, input_shape = (LATENT_SIZE,)),
    LR(),
    Dropout(0.5),
    Dense(128),
    LR(),
    Dropout(0.5),
    Dense(256),
    LR(),
    Dropout(0.5),
    Dense(512),
    LR(),
    Dropout(0.5),
    Dense(784),
    Activation("sigmoid"),
    Reshape((28, 28))
])img = Input(shape = (28, 28))
latent_vector = encoder(img)
output = decoder(latent_vector)model = Model(inputs = img, outputs = output)
model.compile("nadam", loss = "binary_crossentropy")EPOCHS = 60#Only do plotting if you have
IPython, Jupyter, or using Colabfor epoch in range(EPOCHS):
    fig, axs = plt.subplots(4, 4)
    rand = x_test[np.random.randint(0, 10000, 16)].reshape((4, 4, 1, 28, 28))

    display.clear_output() # If you imported display from IPython
```

```
for i in range(4):
    for j in range(4):
        axs[i, j].imshow(model.predict(rand[i, j])[0], cmap = "gray")
        axs[i, j].axis("off")

plt.subplots_adjust(wspace = 0, hspace = 0)
plt.show()
print("-----------", "EPOCH", epoch, "-----------")
model.fit(x_train, x_train)
```

A Google Colab for this Code can be found here.

After training for 60 epochs, I got this image:



Generated Image after 60 epochs

As you can see, the results are pretty good. The autoencoder successfully encodes and decodes the latent space vectors with pretty good quality. This autoencoder is the “vanilla” variety, but other types like Variational Autoencoders have even better quality images. Also, by increasing the number of epochs, results can be improved further.

Uses for Autoencoders

Autoencoders, put simply, learn how to compress and decompress data efficiently without supervision. This means Autoencoders can be used for dimensionality reduction. The Decoder sections of an Autoencoder can also be used to generate images from a noise vector.

Practical applications of an Autoencoder network include:

- Denoising
- Image Reconstruction
- Image Generation
- Data Compression & Decompression

Stepwise Implementation:

Stage 1: Import the fundamental libraries.

Stage 2: Download the dataset. TensorFlow permits us to peruse the MNIST dataset and we can stack it straightforwardly in the program as a train and test dataset.

Stage 3: Prepare Pixel Data

Stage 4: Define Model

Stage 5: Visualize the information.

Stage 6: Fit the model.

Stage 7: Find Accuracy of the model.

CONCLUSION :

Understood the theory and how the implementation of CAE on MNIST dataset

LAB ASSIGNMENT NO:5

Aim: Using IMDB review dataset, perform sentiment classification using LSTM and BiLSTM and compare result

Mapped CO 1: Apply concepts of CNN,RNN and its variant to solve real life problems.

Learning Objectives:

To understand the theory and implementation of RNN

Problem statement:

Implement & perform sentiment classification using LSTM and BiLSTM and compare result
Using IMDB review dataset,

Software and Hardware Requirement

1. 64 bit machine

2. Windows 8 / Open Source Operating System.

3. Google Colab, Notebook Jupyter Notebook

Theory:

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes can create a cycle, allowing output from some nodes to affect subsequent input to the same nodes. This allows it to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs.This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition. Recurrent neural networks are theoretically Turing complete and can run arbitrary programs to process arbitrary sequences of inputs.

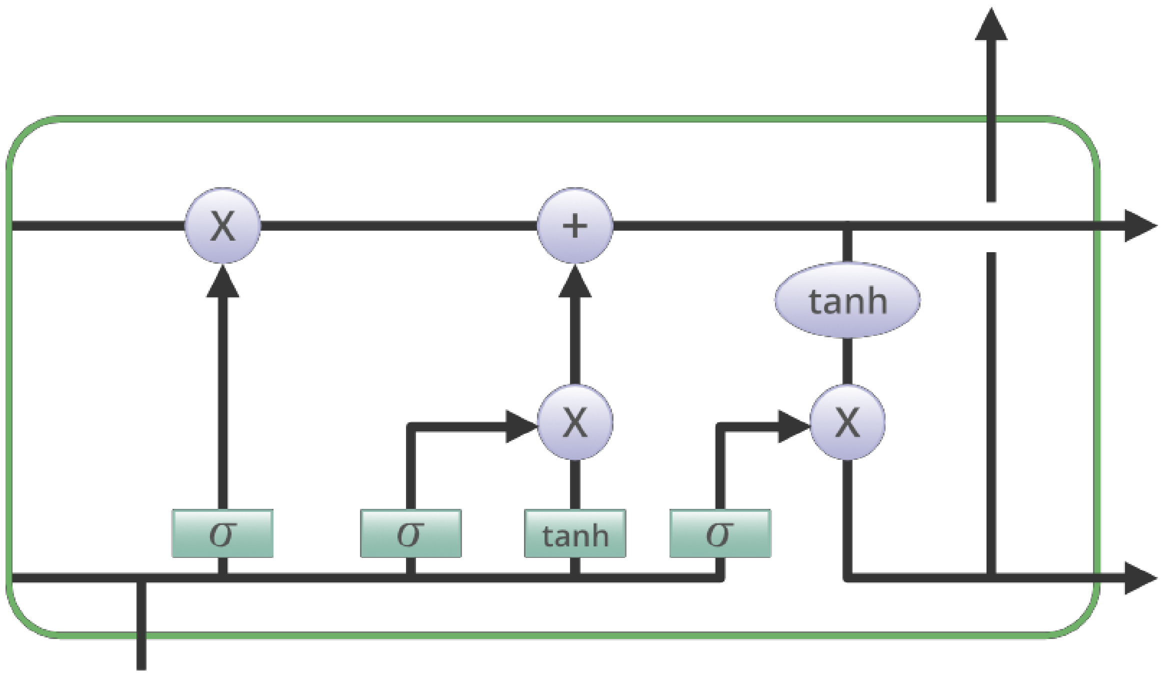
The term "recurrent neural network" is used to refer to the class of networks with an infinite impulse response, whereas "convolutional neural network" refers to the class of finite impulse response. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that can not be unrolled.

Both finite impulse and infinite impulse recurrent networks can have additional stored states, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of long short-term memory networks (LSTMs) and gated recurrent units. This is also called Feedback Neural Network (FNN).

Long Short Term Memory is a kind of recurrent neural network. In RNN output from the last step is fed as input in the current step. LSTM was designed by Hochreiter & Schmidhuber. It tackled the problem of long-term dependencies of RNN in which the RNN cannot predict the word stored in the long-term memory but can give more accurate predictions from the recent information. As the gap length increases RNN does not give an efficient performance. LSTM can by default retain the information for a long period of time. It is used for processing, predicting, and classifying on the basis of time-series data.

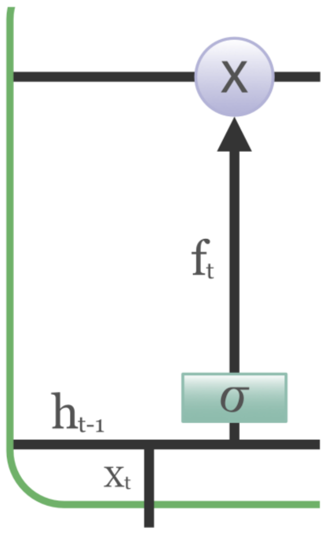
Structure Of LSTM:

LSTM has a chain structure that contains four neural networks and different memory blocks called **cells**.

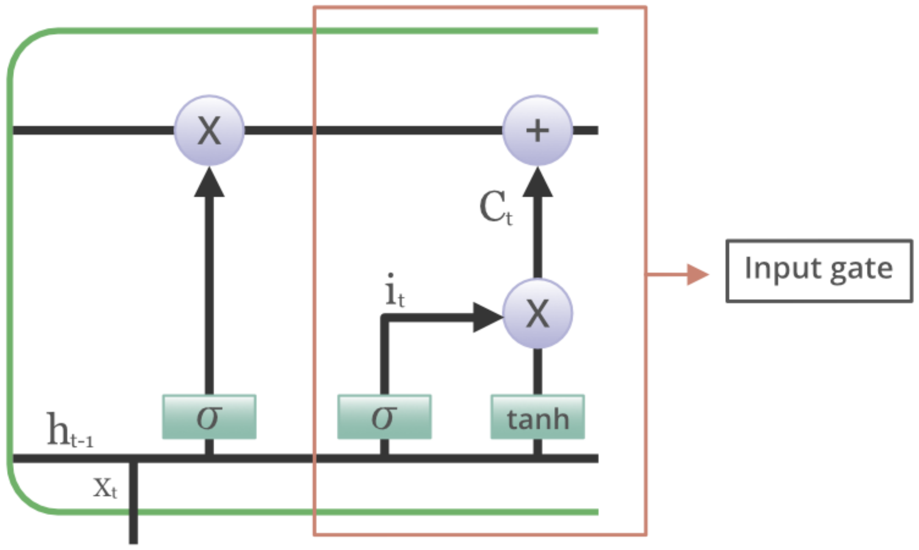


Information is retained by the cells and the memory manipulations are done by the **gates.** There are three gates –

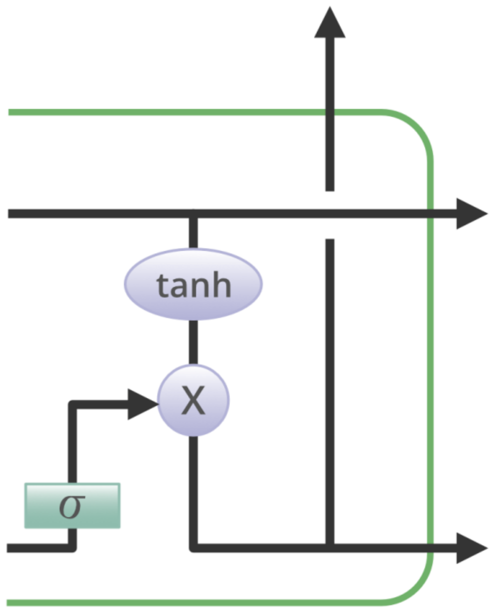
1. Forget Gate: The information that is no longer useful in the cell state is removed with the forget gate. Two inputs *x_t* (input at the particular time) and *h_t-1* (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.



2. Input gate: The addition of useful information to the cell state is done by the input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs *h_t-1* and *x_t*. Then, a vector is created using *tanh* function that gives an output from -1 to +1, which contains all the possible values from h_t-1 and *x_t*. At last, the values of the vector and the regulated values are multiplied to obtain the useful information



3. Output gate: The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter by the values to be remembered using inputs *h_t-1* and *x_t*. At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell.



Some of the famous applications of LSTM includes:

1. Language Modelling
2. Machine Translation
3. Image Captioning

4. Handwriting generation
5. Question Answering Chatbots

Stepwise Implementation:

6. Load in and visualize the data
7. Data Processing — convert to lower case
8. Data Processing — Remove punctuation
9. Data Processing — Create list of reviews
10. Tokenize — Create Vocab to Int mapping dictionary
11. Tokenize — Encode the words
12. Tokenize — Encode the labels
13. Analyze Reviews Length
14. Removing Outliers — Getting rid of extremely long or short reviews
15. Padding / Truncating the remaining data
16. Training, Validation, Test Dataset Split
17. Dataloaders and Batching
18. Define the LSTM Network Architecture
19. Define the Model Class
20. Training the Network
21. Testing (on Test data and User- generated data)

1**) Load in and visualize the data**

We are using IMDB movies review dataset. If it is stored in your machine in a txt file then we just

load it in

```
# read data from text files
with open('data/reviews.txt', 'r') as f:
 reviews = f.read()
with open('data/labels.txt', 'r') as f:
labels  =  f.read()print(reviews[:50])
print()
print(labels[:26])--- Output ---bromwell high is a cartoon comedy . it ran at the same time as some
other programs about school life  such as  teachers  . my   yearspositive
negative
positive
```

2**) Data Processing — convert to lower case**

```
reviews = reviews.lower()
```

3**) Data Processing — remove punctuation**

```
from string import punctuation
print(punctuation)--- Output ---!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

We saw all the punctuation symbols predefined in python. To get rid of all these punctuation we will simply use

```
all_text = ''.join([c for c in reviews if c not in punctuation])
```

4) Data Processing — create list of reviews

We have got all the strings in one huge string. Now we will separate out individual reviews and store them as individual list elements. Like, [review_1, review_2, review_3……. review_n]

```
reviews_split = all_text.split(‘\n’)
print ('Number of reviews :', len(reviews_split))
```

Number of reviews : 25001

5) Tokenize — Create Vocab to Int mapping dictionary

In most of the NLP tasks, you will create an index mapping dictionary in such a way that your frequently occurring words are assigned lower indexes. One of the most common way of doing this is to use Counter method from Collections library.

```
from collections import Counterall_text2 = ' '.join(reviews_split)
# create a list of words
words = all_text2.split()# Count all the words using Counter Method
count_words = Counter(words)

total_words = len(words)
sorted_words = count_words.most_common(total_words)
```

Let’s have a look at these objects we have created

```
print (count_words)--- Output ---Counter({'the': 336713, 'and': 164107, 'a': 163009, 'of': 145864
```

In order to create a vocab to int mapping dictionary, you would simply do this

```
vocab_to_int = {w:i for i, (w,c) in enumerate(sorted_words)}
```

There is a small trick here, in this mapping index will start from 0 i.e. mapping of ‘the’ will be 0. But later on we are going to do padding for shorter reviews and conventional choice for padding is 0. So we need to start this indexing from 1

```
vocab_to_int = {w:i+1 for i, (w,c) in enumerate(sorted_words)}
```

Let’s have a look at this mapping dictionary. We can see that mapping for ‘the’ is 1 now

```
print (vocab_to_int)--- Output ---{'the': 1, 'and': 2, 'a': 3, 'of': 4,
```

6) Tokenize — Encode the words

So far we have created a) list of reviews and b) index mapping dictionary using vocab from all our reviews. All this was to create an encoding of reviews (replace words in our reviews by integers)

```
reviews_int = []
for review in reviews_split:
    r = [vocab_to_int[w] for w in review.split()]
    reviews_int.append(r)
print (reviews_int[0:3])--- Output ---[[21025, 308, 6, 3, 1050, 207, 8, 2138, 32, 1, 171, 57, 15, 49, 81,
5785, 44, 382, 110, 140, 15, .....], [5194, 60, 154, 9, 1, 4975, 5852, 475, 71, 5, 260, 12, 21025, 308, 13,
1978, 6, 74, 2395, 5, 613, 73, 6, 5194, 1, 24103, 5, ....], [1983, 10166, 1, 5786, 1499, 36, 51, 66, 204,
145, 67, 1199, 5194.....]]
```

Note: what we have created now is a list of lists. Each individual review is a list of integer values and all of them are stored in one huge list

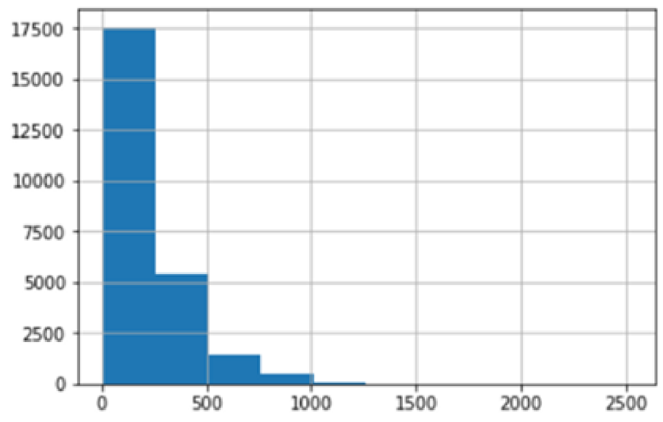
7) Tokenize — Encode the labels

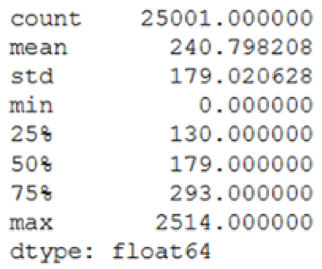
This is simple because we only have 2 output labels. So, we will just label ‘positive’ as 1 and ‘negative’ as 0

```
encoded_labels = [1 if label =='positive' else 0 for label in labels_split]
encoded_labels = np.array(encoded_labels)
```

8) Analyze Reviews Length

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inlinereviews_len = [len(x) for x in reviews_int]
pd.Series(reviews_len).hist()
plt.show()pd.Series(reviews_len).describe()
```





```
count    25001.000000
mean       240.798208
std        179.020628
min          0.000000
25%        130.000000
50%        179.000000
75%        293.000000
max       2514.000000
dtype: float64
```

Review Length Analysis

Observations : a) Mean review length = 240 b) Some reviews are of 0 length. Keeping this review won’t make any sense for our analysis c) Most of the reviews less than 500 words or more d) There are quite a few reviews that are extremely long, we can manually investigate them to check whether we need to include or exclude them from our analysis

9) Removing Outliers — Getting rid of extremely long or short reviews

```
reviews_int = [ reviews_int[i] for i, l in enumerate(reviews_len) if l>0 ]
encoded_labels = [ encoded_labels[i] for i, l in enumerate(reviews_len) if l> 0 ]
```

10) Padding / Truncating the remaining data

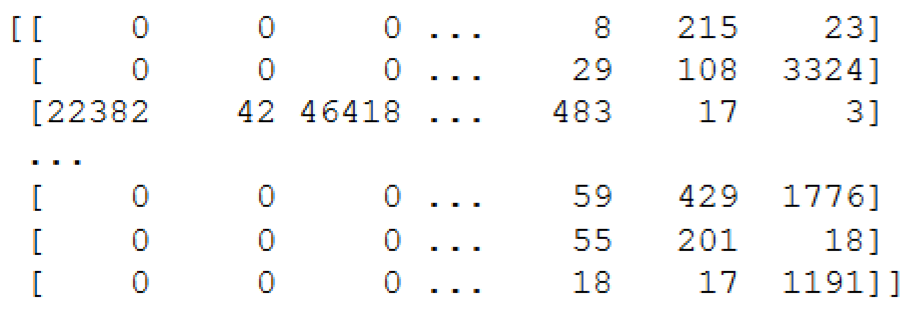
To deal with both short and long reviews, we will pad or truncate all our reviews to a specific length. We define this length by **Sequence Length.** This sequence length is same as number of time steps for LSTM layer.

For reviews shorter than seq_length, we will pad with 0s. For reviews longer than seq_length we will truncate them to the first seq_length words.

```
def pad_features(reviews_int, seq_length):
    ''' Return features of review_ints, where each review is padded with 0's or truncated to the input seq_length.
    '''
    features = np.zeros((len(reviews_int), seq_length), dtype = int)

    for i, review in enumerate(reviews_int):
        review_len = len(review)

        if review_len <= seq_length:
            zeroes = list(np.zeros(seq_length-review_len))
            new = zeroes+review        elif review_len > seq_length:
            new = review[0:seq_length]

        features[i,:] = np.array(new)

    return features
```

Note: We are creating/maintaining a 2D array structure as we created for reviews_int . Output will look like this

```
print (features[:10,:])
```



```
[[    0     0     0 ...     8   215    23]
 [    0     0     0 ...    29   108  3324]
 [22382    42 46418 ...   483    17     3]
 ...
 [    0     0     0 ...    59   429  1776]
 [    0     0     0 ...    55   201    18]
 [    0     0     0 ...    18    17  1191]]
```

11) Training, Validation, Test Dataset Split

Once we have got our data in nice shape, we will split it into training, validation and test sets

train= 80% | valid = 10% | test = 10%

```
split_frac = 0.8
train_x = features[0:int(split_frac*len_feat)]
train_y = encoded_labels[0:int(split_frac*len_feat)]remaining_x = features[int(split_frac*len_feat):]
remaining_y = encoded_labels[int(split_frac*len_feat):]valid_x =
remaining_x[0:int(len(remaining_x)*0.5)]
valid_y = remaining_y[0:int(len(remaining_y)*0.5)]test_x = remaining_x[int(len(remaining_x)*0.5):]
test_y = remaining_y[int(len(remaining_y)*0.5):]
```

CONCLUSION :

Understood the theory and how the implementation of LSTM on sentiment IMDB dataset

LAB ASSIGNMENT NO:6

Aim: Using IMDB review dataset, perform sentiment classification using RNN.

Mapped CO 1: Apply concepts of CNN,RNN and its variant to solve real life problems.

Learning Objectives:

To understand the theory and implementation of RNN

Problem statement:

Implement & perform sentiment classification using RNN
Using IMDB review dataset,

Software and Hardware Requirement

1. 64 bit machine

2. Windows 8 / Open Source Operating System.

3. Google Colab, Notebook Jupyter Notebook

Theory:

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes can create a cycle, allowing output from some nodes to affect subsequent input to the same nodes. This allows it to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs.This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition. Recurrent neural networks are theoretically Turing complete and can run arbitrary programs to process arbitrary sequences of inputs.

The term "recurrent neural network" is used to refer to the class of networks with an infinite impulse response, whereas "convolutional neural network" refers to the class of finite impulse response. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that can not be unrolled.

Both finite impulse and infinite impulse recurrent networks can have additional stored states, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of long short-term memory networks (LSTMs) and gated recurrent units. This is also called Feedback Neural Network (FNN).

Sentiment analysis probably is one the most common applications in Natural Language processing. I don’t have to emphasize how important customer service tool sentiment analysis has become. So here we are, we will train a classifier movie reviews in IMDB data set, using Recurrent Neural Networks. If you want to dive deeper on deep learning for sentiment analysis, this is a good paper.

The data

We will use Recurrent Neural Networks, and in particular LSTMs, to perform sentiment analysis in Keras. Conveniently, Keras has a built-in IMDb movie reviews data set that we can use.

```
from keras.datasets import imdb
```

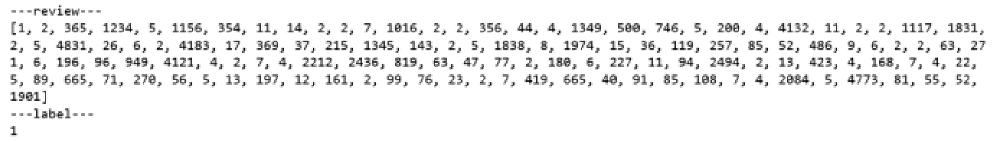
Set the vocabulary size and load in training and test data.

```
vocabulary_size = 5000(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words =
vocabulary_size)
print('Loaded dataset with {} training samples, {} test samples'.format(len(X_train), len(X_test)))
```

Loaded dataset with 25000 training samples, 25000 test samples

Inspect a sample review and its label.

```
print('---review---')
print(X_train[6])
print('---label---')
print(y_train[6])
```



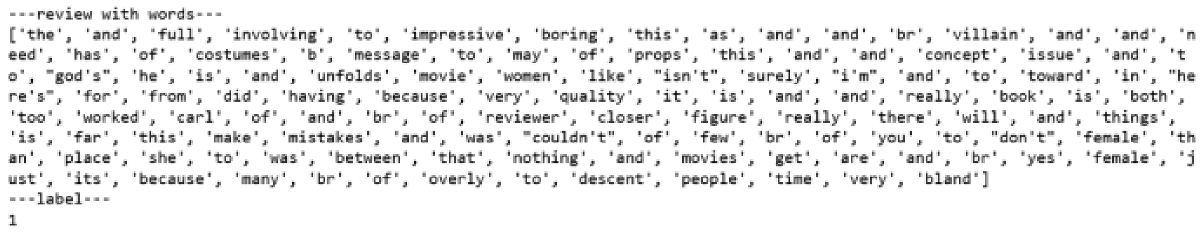
```
---review---
[1, 2, 365, 1234, 5, 1156, 354, 11, 14, 2, 2, 7, 1016, 2, 2, 356, 44, 4, 1349, 500, 746, 5, 200, 4, 4132, 11, 2, 2, 1117, 1831,
2, 5, 4831, 26, 6, 2, 4183, 17, 369, 37, 215, 1345, 143, 2, 5, 1838, 8, 1974, 15, 36, 119, 257, 85, 52, 486, 9, 6, 2, 2, 63, 27
1, 6, 196, 96, 949, 4121, 4, 2, 7, 4, 2212, 2436, 819, 63, 47, 77, 2, 180, 6, 227, 11, 94, 2494, 2, 13, 423, 4, 168, 7, 4, 22,
5, 89, 665, 71, 270, 56, 5, 13, 197, 12, 161, 2, 99, 76, 23, 2, 7, 419, 665, 40, 91, 85, 108, 7, 4, 2084, 5, 4773, 81, 55, 52,
1901]
---label---
1
```

Figure 1

Note that the review is stored as a sequence of integers. These are word IDs that have been pre-assigned to individual words, and the label is an integer (0 for negative, 1 for positive).

We can use the dictionary returned by imdb.get_word_index() to map the review back to the original words.

```
word2id = imdb.get_word_index()
id2word = {i: word for word, i in word2id.items()}
print('---review with words---')
print([id2word.get(i, ' ') for i in X_train[6]])
print('---label---')
print(y_train[6])
```



```
---review with words---
['the', 'and', 'full', 'involving', 'to', 'impressive', 'boring', 'this', 'as', 'and', 'and', 'br', 'villain', 'and', 'and', 'n
eed', 'has', 'of', 'costumes', 'b', 'message', 'to', 'may', 'of', 'props', 'this', 'and', 'and', 'concept', 'issue', 'and', 't
o', "god's", 'he', 'is', 'and', 'unfolds', 'movie', 'women', 'like', "isn't", 'surely', "i'm", 'and', 'to', 'toward', 'in', "he
re's", 'for', 'from', 'did', 'having', 'because', 'very', 'quality', 'it', 'is', 'and', 'and', 'really', 'book', 'is', 'both',
'too', 'worked', 'carl', 'of', 'and', 'br', 'of', 'reviewer', 'closer', 'figure', 'really', 'there', 'will', 'and', 'things',
'is', 'far', 'this', 'make', 'mistakes', 'and', 'was', "couldn't", 'of', 'few', 'br', 'of', 'you', 'to', "don't", 'female', 'th
an', 'place', 'she', 'to', 'was', 'between', 'that', 'nothing', 'and', 'movies', 'get', 'are', 'and', 'br', 'yes', 'female', 'j
ust', 'its', 'because', 'many', 'br', 'of', 'overly', 'to', 'descent', 'people', 'time', 'very', 'bland']
---label---
1
```

Figure 2

Maximum review length and minimum review length.

```
print('Maximum review length: {}'.format(
len(max((X_train + X_test), key=len))))
```

Maximum review length: 2697

```
print('Minimum review length: {}'.format(
len(min((X_test + X_test), key=len))))
```

Minimum review length: 14

Pad sequences

In order to feed this data into our RNN, all input documents must have the same length. We will limit the maximum review length to max_words by truncating longer reviews and padding shorter reviews with a null value (0). We can accomplish this using the pad_sequences() function in Keras. For now, set max_words to 500.

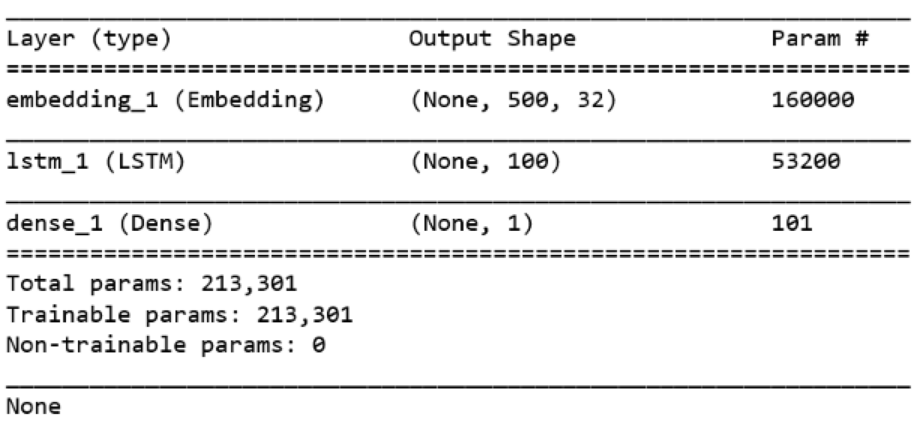
```
from keras.preprocessing import sequencemax_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
```

Design an RNN model for sentiment analysis

We start building our model architecture in the code cell below. We have imported some layers from Keras that you might need but feel free to use any other layers / transformations you like.

Remember that our input is a sequence of words (technically, integer word IDs) of maximum length = max_words, and our output is a binary sentiment label (0 or 1).

```
from keras import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropoutembedding_size=32
model=Sequential()
model.add(Embedding(vocabulary_size, embedding_size, input_length=max_words))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))print(model.summary())
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 32)           160000
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               53200
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 213,301
Trainable params: 213,301
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 3

To summarize, our model is a simple RNN model with 1 embedding, 1 LSTM and 1 dense layers. 213,301 parameters in total need to be trained.

Train and evaluate our model

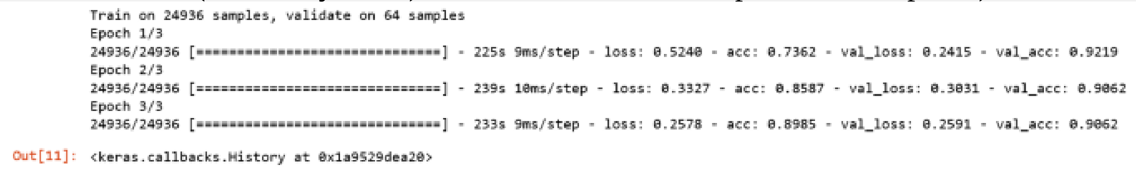
We first need to compile our model by specifying the loss function and optimizer we want to use while training, as well as any evaluation metrics we'd like to measure. Specify the appropriate parameters, including at least one metric 'accuracy'.

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

Once compiled, we can kick off the training process. There are two important training parameters that we have to specify — batch size and number of training epochs, which together with our model architecture determine the total training time.

Training may take a while, so grab a cup of coffee, or better, go for a run!

```
batch_size = 64
num_epochs = 3X_valid, y_valid = X_train[:batch_size], y_train[:batch_size]
X_train2, y_train2 = X_train[batch_size:], y_train[batch_size:]model.fit(X_train2, y_train2,
validation_data=(X_valid, y_valid), batch_size=batch_size, epochs=num_epochs)
```



```
          Train on 24936 samples, validate on 64 samples
          Epoch 1/3
          24936/24936 [==============================] - 225s 9ms/step - loss: 0.5240 - acc: 0.7362 - val_loss: 0.2415 - val_acc: 0.9219
          Epoch 2/3
          24936/24936 [==============================] - 239s 10ms/step - loss: 0.3327 - acc: 0.8587 - val_loss: 0.3031 - val_acc: 0.9062
          Epoch 3/3
          24936/24936 [==============================] - 233s 9ms/step - loss: 0.2578 - acc: 0.8985 - val_loss: 0.2591 - val_acc: 0.9062

Out[11]:  <keras.callbacks.History at 0x1a9529dea20>
```

Figure 4

Once we have trained our model, it's time to see how well it performs on unseen test data.

scores[1] will correspond to accuracy if we pass metrics=['accuracy']

```
scores = model.evaluate(X_test, y_test, verbose=0)
print('Test accuracy:', scores[1])
```

Test accuracy: 0.86964

There are several ways in which we can build our model. We can continue trying and improving the accuracy of our model by experimenting with different architectures, layers and parameters. How good can we get without taking prohibitively long to train? How do we prevent overfitting?

Stepwise Implementation:

22. Load in and visualize the data
23. Data Processing — convert to lower case
24. Data Processing — Remove punctuation
25. Data Processing — Create list of reviews
26. Tokenize — Create Vocab to Int mapping dictionary
27. Tokenize — Encode the words
28. Tokenize — Encode the labels
29. Analyze Reviews Length
30. Removing Outliers — Getting rid of extremely long or short reviews
31. Padding / Truncating the remaining data
32. Training, Validation, Test Dataset Split
33. Dataloaders and Batching
34. Define the RNN Network Architecture
35. Define the Model Class
36. Training the Network
37. Testing (on Test data and User- generated data)

CONCLUSION :

Understood the theory and how the implementation of RNN on sentiment IMDB dataset