**Amity School of Engineering and Technology**

**Deep Learning and Neural Network Practical File**

**Deep Learning and Neural Network**

**(AIML-302)**

****

**Faculty Name**

**Dr Partha Sarathi Mangipudi**

**Submitted By: Shaurya Guliani**

**Enrollment No.: A2305219086**

**Section: 7CSE2X**

**Branch: B. TECH CSE**

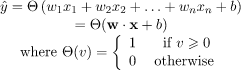
**INDEX**

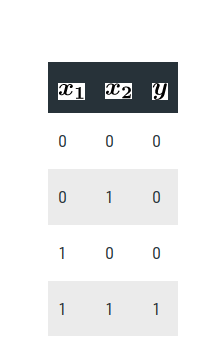
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No.** | **Name of Experiment** | **Date of Allotment of Experiment** | **Date of Evaluation** | **Signature of Faculty** |
| **1.** | **Gates implementation using perceptron.** | **4/08/2022** | **18/08/2022** |  |
| **2.** | **XOR gate propagation using Back-propagation.** | **18/08/2022** | **25/08/2022** |  |
| **3.** | **Classification(MLP).** | **25/08/2022** | **01/09/2022** |  |
| **4.** | **Regression(MLP)** | **01/09/2022** | **08/09/2022** |  |
| **5.** | **Implementation of the basic architecture of CNN.** | **08/09/2022** | **15/09/2022** |  |
| **6.** | **Classification using Alex-net.** | **15/09/2022** | **22/09/2022** |  |
| **7.** | **Classification using VGG-net.** | **22/09/2022** | **29/09/2022** |  |
| **8.** | **Classification using Google-net.** | **29/09/2022** | **06/10/2022** |  |
| **9.** | **Implementation of RNN/LSTM.** | **06/10/2022** | **13/10/2022** |  |

**Practical - 1**

**Statement: Gates implementation using perceptron.**

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





**Code:**

import numpy as np

def unitStep(v):

if v >= 0:

return 1

else:

return 0

def perceptronModel(x, w, b):

v = np.dot(w, x) + b

y = unitStep(v)

return y

def AND\_logicFunction(x):

w = np.array([1, 1])

b = -1.5

return perceptronModel(x, w, b)

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

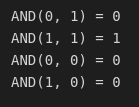
print("AND({}, {}) = {}".format(0, 1, AND\_logicFunction(test1)))

print("AND({}, {}) = {}".format(1, 1, AND\_logicFunction(test2)))

print("AND({}, {}) = {}".format(0, 0, AND\_logicFunction(test3)))

print("AND({}, {}) = {}".format(1, 0, AND\_logicFunction(test4)))

**Output:**

****

**Practical - 2**

**Statement: XOR gate propagation using Back-propagation.**

**Theory:** Artificial Neural Network (ANN) is a computational model based on the biological neural networks of animal brains. ANN is modeled with three types of layers: an input layer, hidden layers (one or more), and an output layer. Each layer comprises nodes (like biological neurons) are called Artificial Neurons. All nodes are connected with weighted edges (like synapses in biological brains) between two layers. Initially, with the forward propagation function, the output is predicted. Then through backpropagation, the weight and bias to the nodes are updated to minimizing the error in prediction to attain the convergence of cost function in determining the final output.

**Code:**

import numpy as np

from matplotlib import pyplot as plt

def sigmoid(z):

return 1 / (1 + np.exp(-z))

def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):

W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)

W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)

b1 = np.zeros((neuronsInHiddenLayers, 1))

b2 = np.zeros((outputFeatures, 1))

parameters = {"W1" : W1, "b1": b1,

"W2" : W2, "b2": b2}

return parameters

# Forward Propagation

def forwardPropagation(X, Y, parameters):

m = X.shape[1]

W1 = parameters["W1"]

W2 = parameters["W2"]

b1 = parameters["b1"]

b2 = parameters["b2"]

Z1 = np.dot(W1, X) + b1

A1 = sigmoid(Z1)

Z2 = np.dot(W2, A1) + b2

A2 = sigmoid(Z2)

cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)

logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))

cost = -np.sum(logprobs) / m

return cost, cache, A2

# Backward Propagation

def backwardPropagation(X, Y, cache):

m = X.shape[1]

(Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

dZ2 = A2 - Y

dW2 = np.dot(dZ2, A1.T) / m

db2 = np.sum(dZ2, axis = 1, keepdims = True)

dA1 = np.dot(W2.T, dZ2)

dZ1 = np.multiply(dA1, A1 \* (1- A1))

dW1 = np.dot(dZ1, X.T) / m

db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,

"dZ1": dZ1, "dW1": dW1, "db1": db1}

return gradients

# Updating the weights based on the negative gradients

def updateParameters(parameters, gradients, learningRate):

parameters["W1"] = parameters["W1"] - learningRate \* gradients["dW1"]

parameters["W2"] = parameters["W2"] - learningRate \* gradients["dW2"]

parameters["b1"] = parameters["b1"] - learningRate \* gradients["db1"]

parameters["b2"] = parameters["b2"] - learningRate \* gradients["db2"]

return parameters

# Model to learn the XOR truth table

X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) # XOR input

Y = np.array([[0, 1, 1, 0]]) # XOR output

# Define model parameters

neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)

inputFeatures = X.shape[0] # number of input features (2)

outputFeatures = Y.shape[0] # number of output features (1)

parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures)

epoch = 100000

learningRate = 0.01

losses = np.zeros((epoch, 1))

for i in range(epoch):

losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)

gradients = backwardPropagation(X, Y, cache)

parameters = updateParameters(parameters, gradients, learningRate)

plt.figure()

plt.plot(losses)

plt.xlabel("EPOCHS")

plt.ylabel("Loss value")

plt.show()

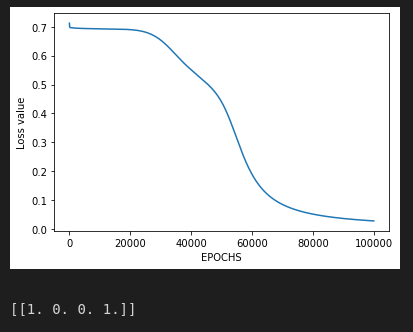
X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # XOR input

cost, \_, A2 = forwardPropagation(X, Y, parameters)

prediction = (A2 > 0.5) \* 1.0

print(prediction)

**Output:**

****

**Practical - 3**

**Statement: Classification using neural networks.**

**Theory:** Classification is the task of categorizing the known classes based on their features. In most classification problems, machine learning algorithms will do the job, but while classifying a large dataset of images, you will need to use a neural network. I hope you liked this article on classification with neural networks using Python. Feel free to ask valuable questions in the comments section below.

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import tensorflow as tf

print(tf.\_\_version\_\_)

from sklearn.datasets import make\_circles

samples = 1000

X, y = make\_circles(samples,

noise = 0.03,

random\_state = 42)

print('X : ', X)

print('\n')

print('y : ', y)

circle = pd.DataFrame({ 'X0' : X[:, 0], 'X1' : X[:, 1], 'label' : y})

circle.head()

circle.label.value\_counts()

plt.scatter(X[:,0], X[:,1], c = y, cmap = plt.cm.RdYlBu)

print(X.shape, y.shape)

print(len(X), len(y))

tf.random.set\_seed(42)

model\_1 = tf.keras.Sequential([tf.keras.layers.Dense(1)])

model\_1.compile(loss = tf.keras.losses.BinaryCrossentropy(),optimizer = tf.keras.optimizers.SGD(),

#SGD stands for Stochastic Gradient Descent

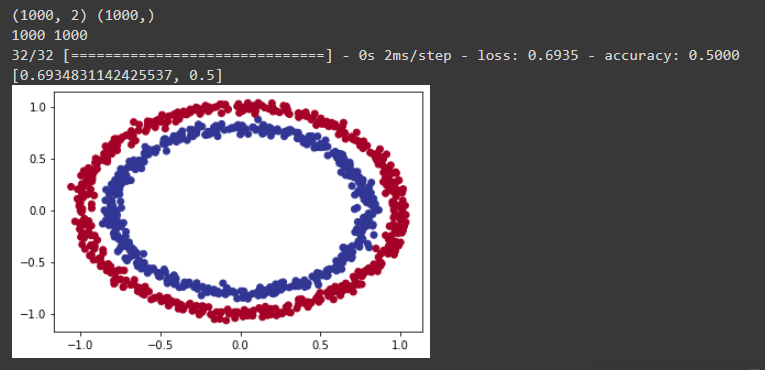
metrics = ['accuracy'])

model\_1.fit(X, y, epochs = 200, verbose = 0)

#we set verbose = 0 to remove training procedure )

model\_1.evaluate(X, y)

**Output -**

****

**Practical - 4**

**Statement: Implement regression using neural networks.**

**Theory:** Regression models are one of the most commonly used machine learning models used in predicting a continuous value. Moreover, Regression analysis is a statistical method to model the relationship between a dependent (target) and independent (predictor) variables with one or more independent variables. More specifically, Regression analysis helps us to understand how the value of the dependent variable is changing corresponding to an independent variable when other independent variables are held fixed. It predicts continuous/real values such as temperature, age, salary, price, etc.

Regression Commonly used two variables:

1. Dependent Variable: The main factor in Regression analysis which we want to predict or understand is called the dependent variable. It is also called target variable.
2. Independent Variable: The factors which affect the dependent variables or which are used to predict the values of the dependent variables are called independent variables, also called as a predictor.

**Code and Output:**

#Importing the required Libraries

import pandas as pd

import numpy as np

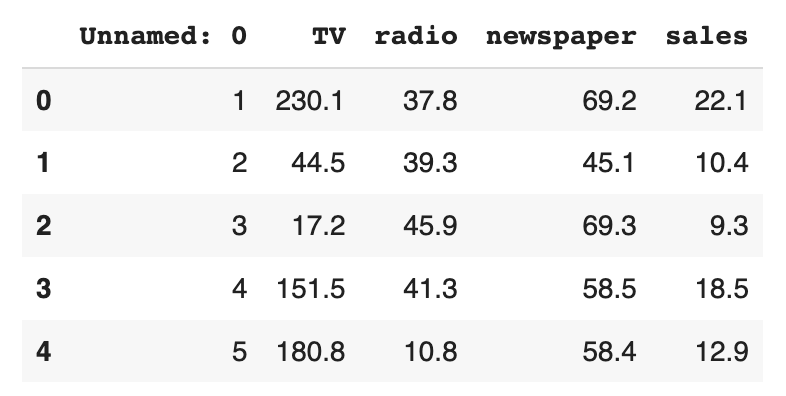
import matplotlib.pyplot as plt

from sklearn.neural\_network import MLPRegressor

from sklearn.metrics import r2\_score

data = pd.read\_csv('Advertising.csv')

data.head()

****

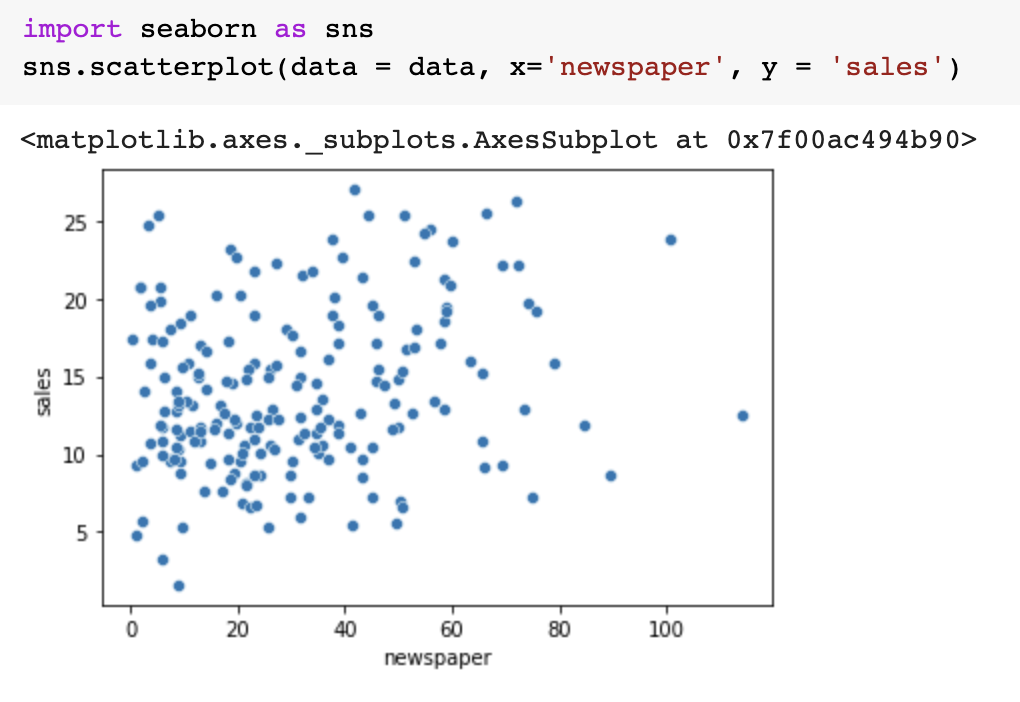
**data.count**

**data.shape**

**(200, 5)**

**import seaborn as sns**

**sns.scatterplot(data = data, x='newspaper', y = 'sales')**

****

**#Function for MLPRegressor**

**def MLP\_Regressor(x\_train, y\_train):**

**print("MLP Regressor: ")**

**mlp\_regressor = MLPRegressor(activation='relu', hidden\_layer\_sizes=(13,13,13), solver='lbfgs', verbose = True, max\_iter= 20000)**

**mlp\_regressor.fit(x\_train, y\_train)**

**return mlp\_regressor**

**def build\_and\_train\_model(data, target\_name, reg\_fn):**

**X = data.drop(target\_name, axis=1)**

**Y = data[target\_name]**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, shuffle=True, test\_size=0.2)**

**scaler = StandardScaler()**

**scaler.fit(x\_train)**

**x\_train = scaler.transform(x\_train)**

**x\_test = scaler.transform(x\_test)**

**model = reg\_fn(x\_train, y\_train)**

**score = model.score(x\_train, y\_train)**

**print("Training score: ", score)**

**y\_pred = model.predict(x\_test)**

**r\_score = r2\_score(y\_test, y\_pred)**

**print("Testing Score : ", r\_score)**

**df\_y = pd.DataFrame({'y\_test' : y\_test, 'y\_pred' : y\_pred})**

**print(df\_y.sample(10))**

**plt.figure(figsize=(10, 8))**

**plt.plot(y\_pred, label='Predicted')**

**plt.plot(y\_test.values, label='Actual')**

**plt.ylabel("medv of home")**

**plt.legend()**

**plt.show()**

**return{'model' : model,**

**'x\_train' : x\_train, 'x\_test' : x\_test,**

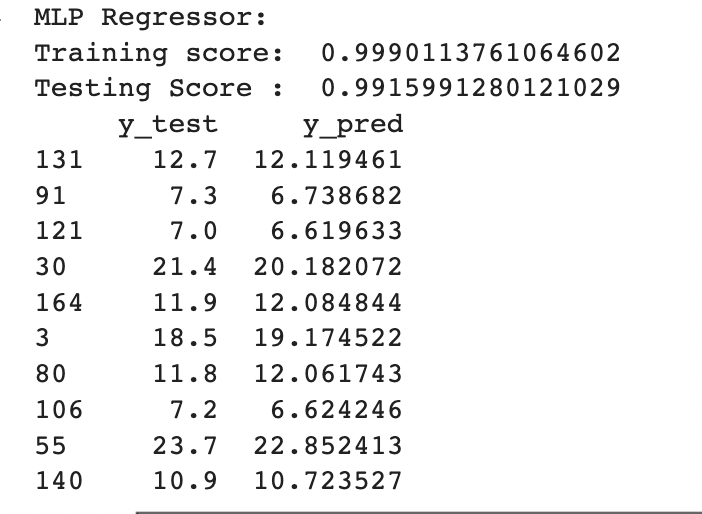
**'y\_train' : y\_train, 'y\_test' : y\_test,**

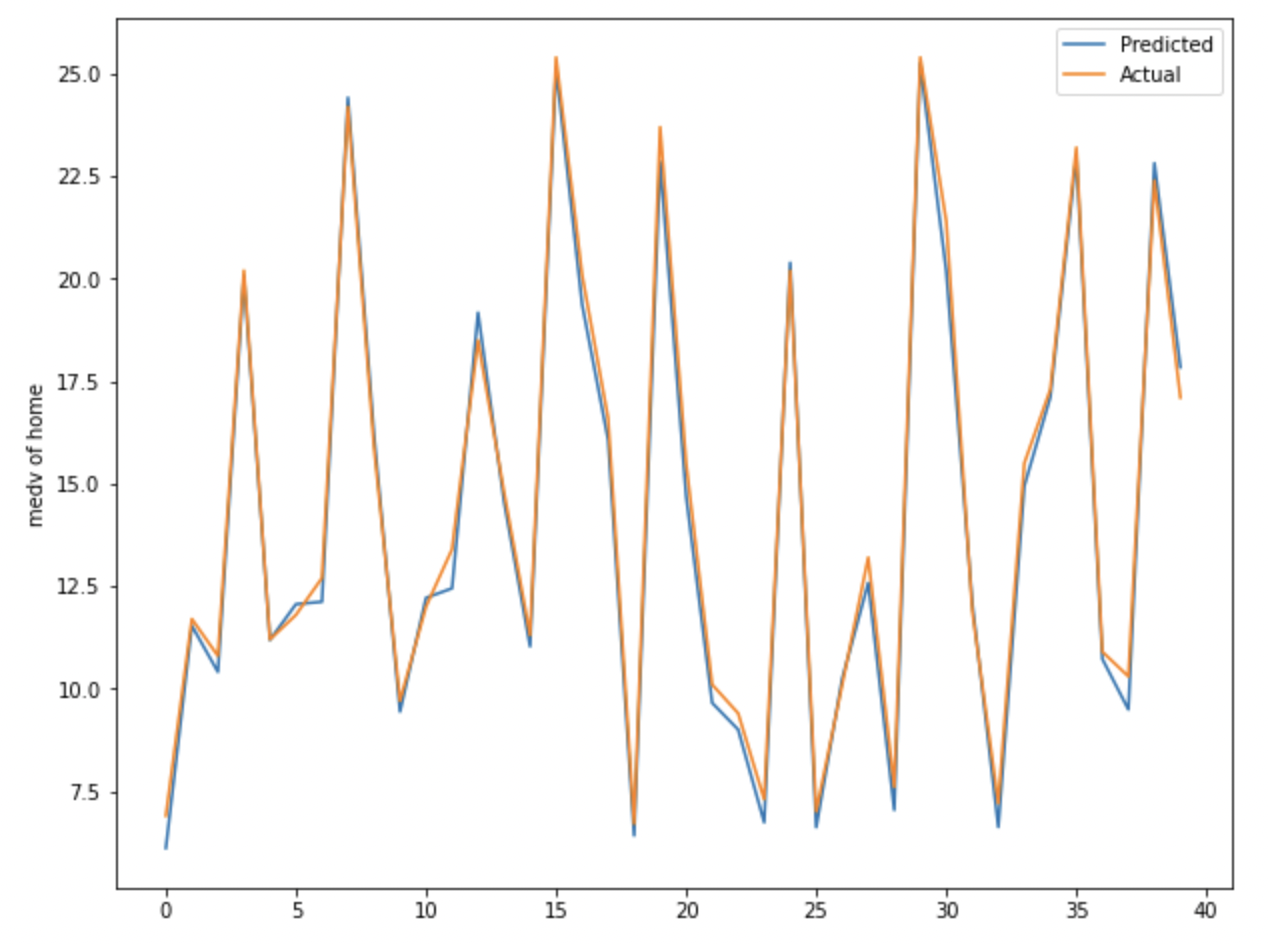
**'y\_pred' : y\_pred, 'sample' : df\_y.sample(10)**

**}**

**MLP\_Regression = build\_and\_train\_model(data, "sales", MLP\_Regressor)**

**Final output:**

****

****

**Conclusion:**

It is a regression problem which is predicting the sales based on given set attributes (methods of advertising): Using Multilayer Perceptron MLP Regressor to predict the sales based on numerical and categorical variables.

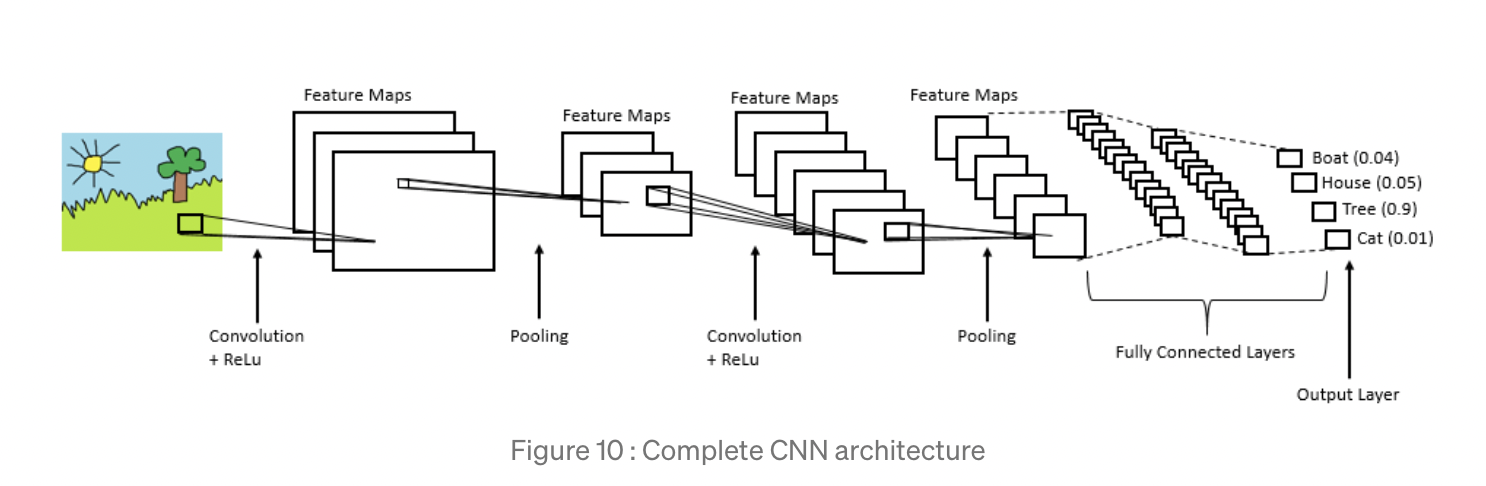
**Practical - 5**

**Statement: Implementation of the basic architecture of CNN.**

**Theory:**

CNN stands for convolutional neural network, It is a type of neural network which is used to process images and has a wide range of applications for image and video classifications, processing, face recognition etc. CNN usually takes images as inputs which it works upon to learn and make appropriate classification decisions.

CNN image classifications take an input image, process it and classify it under certain categories (Eg., Dog, Cat, Tiger, Lion). Computers see an input image as an array of pixels and it depends on the image resolution. Based on the image resolution, it will see h x w x d( h = Height, w = Width, d = Dimension ). Eg., An image of a 6 x 6 x 3 array of matrix of RGB (3 refers to RGB values) and an image of 4 x 4 x 1 array of matrix of grayscale image.Padding or zero padding may be done as and when required to process an image. A simple CNN architecture can be seen in the figure below.



1. Sequential: Creates a linear stack of layers
2. Drouput: Ensures minimum overfitting. it does this my selecting random nodes and setting them to 0
3. Dense: This essentially is the output layer. It performs the output = activation(dot(input, weights) + bias)
4. Flatten: This rolls out our array into 2 dimensions, [numberOfData, features]
5. SGD: Stochastic Gradient Descent, this is the optimizer
6. Conv2D: This is the convolution layer
7. MaxPooling2D: This function performs max pooling np\_utils: Some tools to allow us to format our data

**Data set used:**

[**https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz**](https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)

**Code and Output:**

from keras.models import Sequential

from keras.layers import Dropout, Dense, Flatten

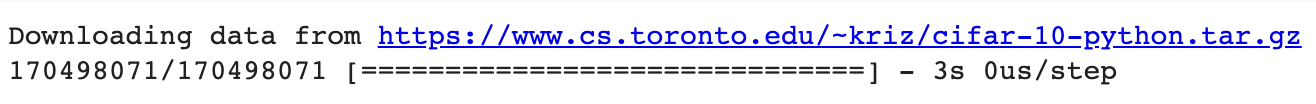
from keras.optimizers import SGD

from keras.layers.convolutional import Conv2D, MaxPooling2D

from keras.utils import np\_utils as u

from keras.datasets import cifar10

(X, y), (X\_test, y\_test) = cifar10.load\_data()

 #RGB images being used , thus we convert data into float and keeping in mind the images are RGB

X, X\_test = X.astype('float32')/255.0, X\_test.astype('float32')/255.0

y, y\_test = u.to\_categorical(y, 10), u.to\_categorical(y\_test, 10)

model = Sequential()

#Output is having 32 features maps. The kernel size is going to be 3 channels

model.add(Conv2D(32, (3, 3), input\_shape=(32, 32, 3), padding='same', activation='relu'))

model.add(Dropout(0.2))

model.add(Conv2D(32, (3, 3), activation='relu', padding='valid'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

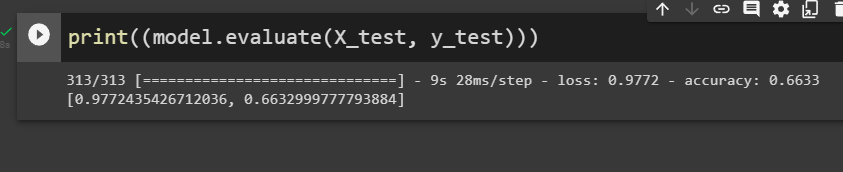
model.add(Dense(512, activation='relu'))

model.add(Dropout(0.3))

model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam')

**Output:**

****

**Practical - 6**

**Statement: Classification using Alex-net.**

**Theory:** The architecture consists of eight layers: five convolutional layers and three fully-connected layers. But this isn’t what makes AlexNet special; these are some of the features used that are new approaches to convolutional neural networks:

1. ReLU Nonlinearity. AlexNet uses Rectified Linear Units (ReLU) instead of the tanh function, which was standard at the time. ReLU’s advantage is in training time; a CNN using ReLU was able to reach a 25% error on the CIFAR-10 dataset six times faster than a CNN using tanh.
2. Multiple GPUs. Back in the day, GPUs were still rolling around with 3 gigabytes of memory (nowadays those kinds of memory would be rookie numbers). This was especially bad because the training set had 1.2 million images. AlexNet allows for multi-GPU training by putting half of the model’s neurons on one GPU and the other half on another GPU. Not only does this mean that a bigger model can be trained, but it also cuts down on the training time.
3. Overlapping Pooling. CNNs traditionally “pool” outputs of neighboring groups of neurons with no overlapping. However, when the authors introduced overlap, they saw a reduction in error by about 0.5% and found that models with overlapping pooling generally find it harder to overfit.
4. Data Augmentation. The authors used label-preserving transformation to make their data more varied. Specifically, they generated image translations and horizontal reflections, which increased the training set by a factor of 2048. They also performed Principle Component Analysis (PCA) on the RGB pixel values to change the intensities of RGB channels, which reduced the top-1 error rate by more than 1%.
5. Dropout. This technique consists of “turning off” neurons with a predetermined probability (e.g. 50%). This means that every iteration uses a different sample of the model’s parameters, which forces each neuron to have more robust features that can be used with other random neurons. However, dropout also increases the training time needed for the model’s convergence.

**Program Code:**

**import** os

**import** cv2

**from** glob **import** glob

**import** tensorflow **as** tf

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

**from** tensorflow.keras.preprocessing.image **import** ImageDataGenerator,load\_img,img\_to\_array

**from** tensorflow.keras.preprocessing **import** image

**from** tensorflow.keras.utils **import** to\_categorical

**from** keras.models **import** Sequential

**from** tensorflow.keras.utils **import** plot\_model

**from** keras.models **import** Model

**from** keras.layers **import** Input

**from** tensorflow.keras.optimizers **import** Adam

**from** tensorflow.keras.layers **import** Activation,GlobalAveragePooling2D, Dense, BatchNormalization, Dropout, Flatten, Conv2D, MaxPooling2D

**from** tensorflow.keras.optimizers **import** SGD

image\_count **=** []

class\_names **=** []

print('{:18s}'**.**format('class'), end**=**'')

print('Count:')

print('-' **\*** 24)

*#Reading the image from each folder from training path*

**for** folder **in** os**.**listdir(train\_path):

folder\_num **=** len(os**.**listdir(os**.**path**.**join(train\_path,folder)))

image\_count**.**append(folder\_num)

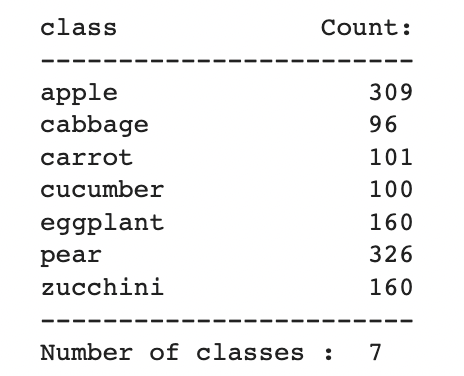
class\_names**.**append(folder)

print('{:20s}'**.**format(folder), end**=**' ')

print(folder\_num)

print('-' **\*** 24)

print("Number of classes : ",len(class\_names))



model **=** tf**.**keras**.**models**.**Sequential([

*#1st Convolutional Layer*

tf**.**keras**.**layers**.**Conv2D(filters**=**96, kernel\_size**=**(11,11), strides**=**(4,4), activation**=**'relu', input\_shape**=**(227,227,3)),

tf**.**keras**.**layers**.**BatchNormalization(),

tf**.**keras**.**layers**.**MaxPool2D(pool\_size**=**(3,3), strides**=**(2,2)),

*#2nd Convolutional Layer*

tf**.**keras**.**layers**.**Conv2D(filters**=**256, kernel\_size**=**(5,5), strides**=**(1,1), activation**=**'relu', padding**=**"same"),

tf**.**keras**.**layers**.**BatchNormalization(),

tf**.**keras**.**layers**.**MaxPool2D(pool\_size**=**(3,3), strides**=**(2,2)),

*#3rd Convolutional Layer*

tf**.**keras**.**layers**.**Conv2D(filters**=**384, kernel\_size**=**(3,3), strides**=**(1,1), activation**=**'relu', padding**=**"same"),

tf**.**keras**.**layers**.**BatchNormalization(),

tf**.**keras**.**layers**.**MaxPool2D(pool\_size**=**(3,3), strides**=**(2,2)),

*#4th Convolutional Layer*

tf**.**keras**.**layers**.**Conv2D(filters**=**384, kernel\_size**=**(3,3), strides**=**(1,1), activation**=**'relu', padding**=**"same"),

tf**.**keras**.**layers**.**BatchNormalization(),

*#5th Convolutional Layer*

tf**.**keras**.**layers**.**Conv2D(filters**=**256, kernel\_size**=**(3,3), strides**=**(1,1), activation**=**'relu', padding**=**"same"),

tf**.**keras**.**layers**.**BatchNormalization(),

*#Passing it to a Fully Connected layer*

tf**.**keras**.**layers**.**Flatten(),

*# 1st Fully Connected Layer*

tf**.**keras**.**layers**.**Dense(4096, activation**=**'relu'),

tf**.**keras**.**layers**.**BatchNormalization(),

tf**.**keras**.**layers**.**Dropout(0.5),*# Add Dropout to prevent overfitting*

*# 2nd Fully Connected Layer*

tf**.**keras**.**layers**.**Dense(4096, activation**=**'relu'),

*#tf.keras.layers.BatchNormalization(),*

*#tf.keras.layers.Dropout(0.5),*

*# 3rd Fully Connected Layer*

tf**.**keras**.**layers**.**Dense(1000, activation**=**'relu'),

*#tf.keras.layers.BatchNormalization(),*

*#tf.keras.layers.Dropout(0.5),*

*#Output Layer*

tf**.**keras**.**layers**.**Dense(7, activation**=**'softmax'),

*#tf.keras.layers.BatchNormalization()*

])

**from** keras.callbacks **import** ReduceLROnPlateau

*#Callback to save the best model. Using checkpoint and earlystopping to monitor validation accuracy*

callbacks\_list **=** [

tf**.**keras**.**callbacks**.**ReduceLROnPlateau(

monitor**=**'val\_accuracy', factor**=**0.1, patience**=**10, verbose**=**1),

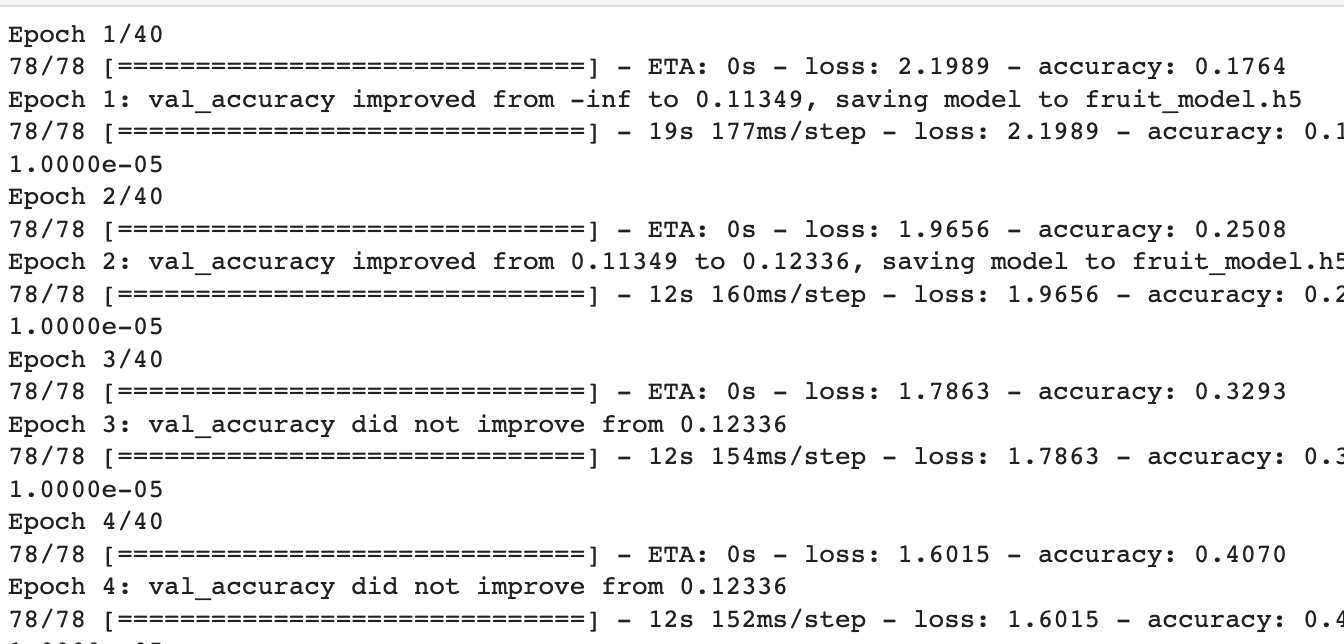
tf**.**keras**.**callbacks**.**ModelCheckpoint(

filepath**=**'fruit\_model.h5',

monitor**=**'val\_accuracy', save\_best\_only**=True**, verbose**=**1),

tf**.**keras**.**callbacks**.**EarlyStopping(monitor**=**'val\_accuracy', patience**=**10,verbose**=**1)

]



*# predict the class*

predict\_x**=**model**.**predict(img)

result**=**np**.**argmax(predict\_x,axis**=**1)

**if** result[0] **==** 0:

print("Apple")

**elif** result[0] **==** 1:

print("cabbage")

**elif** result[0] **==** 2:

print("carrot")

**elif** result[0] **==** 3:

print("cucumber")

**elif** result[0] **==** 4:

print("eggplant")

**elif** result[0] **==** 5:

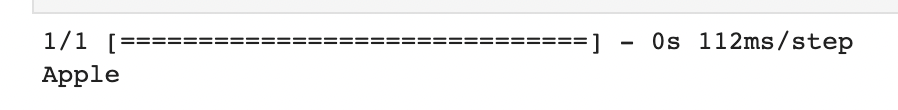
print("pear")

**elif** result[0] **==** 6:

print("zucchini")

**else**:

print("Not in the list")

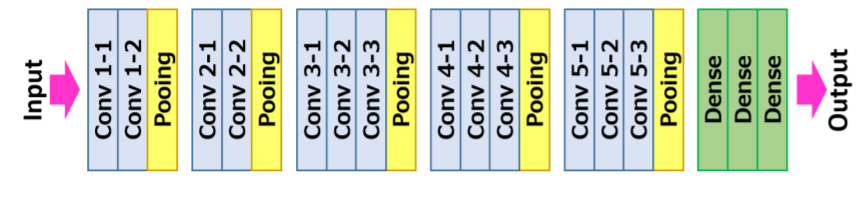


**Practical - 7**

**Statement: Classification using VGG-net.**

**Theory:** VGG- Network is a convolutional neural network model proposed by K. Simonyan and A. Zisserman in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition” [1]. This architecture achieved top-5 test accuracy of 92.7% in ImageNet, which has over 14 million images belonging to 1000 classes.

It is one of the famous architectures in the deep learning field. Replacing large kernel-sized filters with 11 and 5 in the first and second layer respectively showed the improvement over AlexNet architecture, with multiple 3×3 kernel-sized filters one after another. It was trained for weeks and was using NVIDIA Titan Black GPU’s.

****

This article was published as a part of the Data Science Blogathon

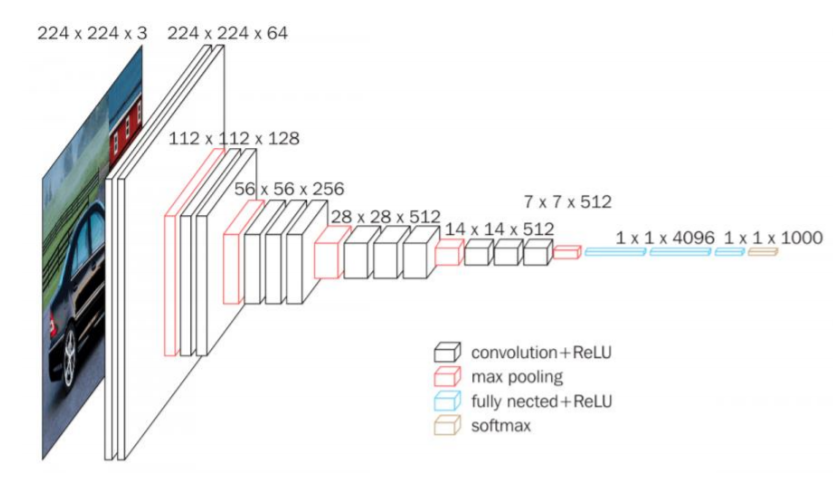
VGG- Network is a convolutional neural network model proposed by K. Simonyan and A. Zisserman in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition” [1]. This architecture achieved top-5 test accuracy of 92.7% in ImageNet, which has over 14 million images belonging to 1000 classes.

It is one of the famous architectures in the deep learning field. Replacing large kernel-sized filters with 11 and 5 in the first and second layer respectively showed the improvement over AlexNet architecture, with multiple 3×3 kernel-sized filters one after another. It was trained for weeks and was using NVIDIA Titan Black GPU’s.

Then the image is running through a stack of convolutional (Conv.) layers, where there are filters with a very small receptive field that is 3 × 3, which is the smallest size to capture the notion of left/right, up/down, and center part.

In one of the configurations, it also utilizes 1 × 1 convolution filters, which can be observed as a linear transformation of the input channels followed by non-linearity. The convolutional strides are fixed to 1 pixel; the spatial padding of convolutional layer input is such that the spatial resolution is maintained after convolution, that is the padding is 1 pixel for 3 × 3 Conv. layers.

Then the Spatial pooling is carried out by five max-pooling layers, 16 which follow some of the Conv. layers but not all the Conv. layers are followed by max-pooling. This Max-pooling is performed over a 2 × 2-pixel window, with stride 2.



**Code:**

import torch

import torch.nn as nn

VGG\_types = {

"VGG11": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],

"VGG13": [64, 64, "M", 128, 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],

"VGG16": [64,64,"M",128,128,"M",256,256,256,"M",512,512,512,"M",512,512,512,"M",],

"VGG19": [64,64,"M",128,128,"M",256,256,256,256,"M",512,512,512,512,

"M",512,512,512,512,"M",],}

VGGType = "VGG16"

self.fcs = nn.Sequential(

nn.Linear(512 \* 7 \* 7, 4096),

nn.ReLU(),

nn.Dropout(p=0.5),

nn.Linear(4096, 4096),

nn.ReLU(),

nn.Dropout(p=0.5),

nn.Linear(4096, num\_classes),

)

def forward(self, x):

x = self.conv\_layers(x)

x = x.reshape(x.shape[0], -1)

x = self.fcs(x)

return x

def create\_conv\_layers(self, architecture):

layers = []

in\_channels = self.in\_channels

for x in architecture:

if type(x) == int:

out\_channels = x

layers += [

nn.Conv2d(

in\_channels=in\_channels,

out\_channels=out\_channels,

kernel\_size=(3, 3),

stride=(1, 1),

padding=(1, 1),

),

nn.BatchNorm2d(x),

nn.ReLU(),

]

in\_channels = x

elif x == "M":

layers += [nn.MaxPool2d(kernel\_size=(2, 2), stride=(2, 2))]

return nn.Sequential(\*layers)

if \_\_name\_\_ == "\_\_main\_\_":

device = "cuda" if torch.cuda.is\_available() else "cpu"

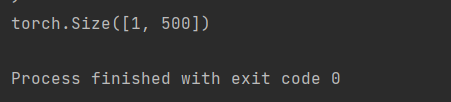
model = VGGnet(in\_channels=3, num\_classes=500).to(device)

# print(model)

x = torch.randn(1, 3, 224, 224).to(device)

print(model(x).shape)

**Output:**



**Practical - 8**

**Statement: Classification using Google-net.**

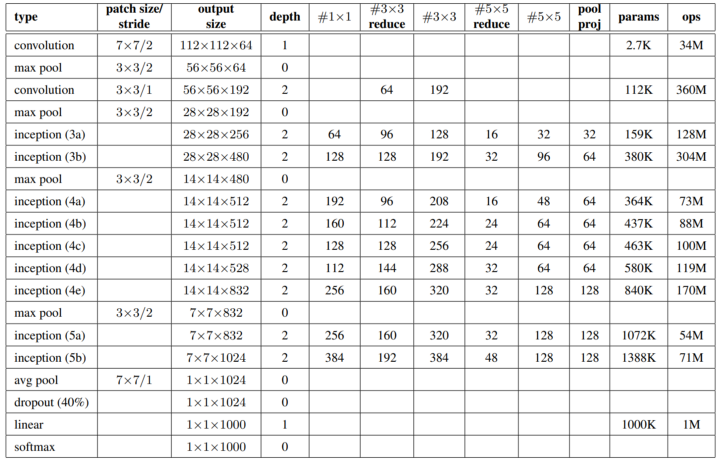
**Theory:** GoogLeNet is a 22-layer deep convolutional neural network that’s a variant of the Inception Network, a Deep Convolutional Neural Network developed by researchers at Google.

The GoogLeNet architecture presented in the ImageNet Large-Scale Visual Recognition Challenge 2014(ILSVRC14) solved computer vision tasks such as image classification and object detection — find out how well it performed at the conclusion section of this article.

Today GoogLeNet is used for other computer vision tasks such as face detection and recognition, adversarial training etc.

The GoogLeNet architecture consists of 22 layers (27 layers including pooling layers), and part of these layers are a total of 9 inception modules(figure4).

The table below depicts the conventional GoogLeNet architecture. Have a quick review of the table before reading more on the table’s characteristics and features.



## Characteristics and features of GoogLeNet configuration table (figure 1)

* The input layer of the GoogLeNet architecture takes in an image of the dimension 224 x 224.
* Type: This refers to the name of the current layer of the component within the architecture
* Patch Size: Refers to the size of the sweeping window utilised across conv and pooling layers. Sweeping windows have equal height and width.
* Stride: Defines the amount of shift the filter/sliding window takes over the input image.
* Output Size: The resulting output dimensions(height, width, number of feature maps) of the current architecture component after the input is passed through the layer.
* Depth: Refer to the number of levels/layers within an architecture component.
* #1x1 #3x3 #5x5: Refers to the various convolutions filters used within the inception module.
* #3X3 reduce #5x5 reduce: Refers to the numbers of 1x1 filters used before the convolutions.
* Pool Proj: This is the number of 1x1 filters used after pooling within an inception module.
* Params: Refers to the number of weights within the current architecture component.
* Ops: Refers to the number of mathematical operations carried out within the component.

**Code:**

import keras

from keras.layers.core import Layer

import keras.backend as K

import tensorflow as tf

from keras.datasets import cifar10

from keras.models import Model

from keras.layers import Conv2D, MaxPool2D, \

Dropout, Dense, Input, concatenate, \

GlobalAveragePooling2D, AveragePooling2D,\

Flatten

import cv2

import numpy as np

from keras.datasets import cifar10

from keras import backend as K

from keras.utils import np\_utils

import math

from keras.optimizers import SGD

from keras.callbacks import LearningRateScheduler

num\_classes = 10

def load\_cifar10\_data(img\_rows, img\_cols):

# Load cifar10 training and validation sets

(X\_train, Y\_train), (X\_valid, Y\_valid) = cifar10.load\_data()

# Resize training images

X\_train = np.array([cv2.resize(img, (img\_rows,img\_cols)) for img in X\_train[:,:,:,:]])

X\_valid = np.array([cv2.resize(img, (img\_rows,img\_cols)) for img in X\_valid[:,:,:,:]])

# Transform targets to keras compatible format

Y\_train = np\_utils.to\_categorical(Y\_train, num\_classes)

Y\_valid = np\_utils.to\_categorical(Y\_valid, num\_classes)

X\_train = X\_train.astype('float32')

X\_valid = X\_valid.astype('float32')

# preprocess data

X\_train = X\_train / 255.0

X\_valid = X\_valid / 255.0

return X\_train, Y\_train, X\_valid, Y\_valid

X\_train, y\_train, X\_test, y\_test = load\_cifar10\_data(224, 224)

def inception\_module(x,

filters\_1x1,

filters\_3x3\_reduce,

filters\_3x3,

filters\_5x5\_reduce,

filters\_5x5,

filters\_pool\_proj,

name=None):

conv\_1x1 = Conv2D(filters\_1x1, (1, 1), padding='same', activation='relu', kernel\_initializer=kernel\_init, bias\_initializer=bias\_init)(x)

conv\_3x3 = Conv2D(filters\_3x3\_reduce, (1, 1), padding='same', activation='relu', kernel\_initializer=kernel\_init, bias\_initializer=bias\_init)(x)

conv\_3x3 = Conv2D(filters\_3x3, (3, 3), padding='same', activation='relu', kernel\_initializer=kernel\_init, bias\_initializer=bias\_init)(conv\_3x3)

conv\_5x5 = Conv2D(filters\_5x5\_reduce, (1, 1), padding='same', activation='relu', kernel\_initializer=kernel\_init, bias\_initializer=bias\_init)(x)

conv\_5x5 = Conv2D(filters\_5x5, (5, 5), padding='same', activation='relu', kernel\_initializer=kernel\_init, bias\_initializer=bias\_init)(conv\_5x5)

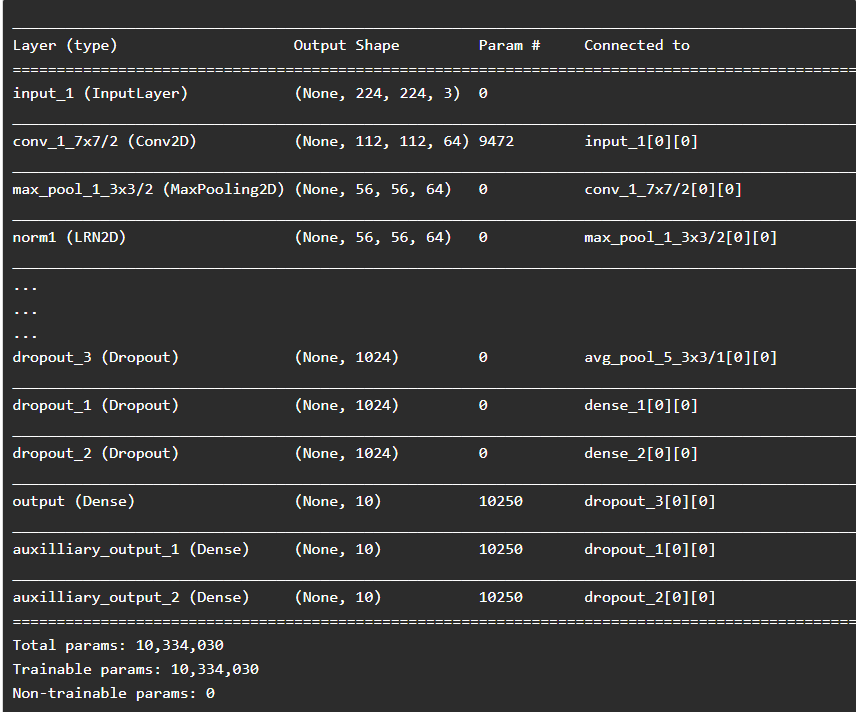
pool\_proj = MaxPool2D((3, 3), strides=(1, 1), padding='same')(x)

pool\_proj = Conv2D(filters\_pool\_proj, (1, 1), padding='same', activation='relu', kernel\_initializer=kernel\_init, bias\_initializer=bias\_init)(pool\_proj)

output = concatenate([conv\_1x1, conv\_3x3, conv\_5x5, pool\_proj], axis=3, name=name)

return output

model = Model(input\_layer, [x, x1, x2], name='inception\_v1')



epochs = 25

initial\_lrate = 0.01

def decay(epoch, steps=100):

initial\_lrate = 0.01

drop = 0.96

epochs\_drop = 8

lrate = initial\_lrate \* math.pow(drop, math.floor((1+epoch)/epochs\_drop))

return lrate

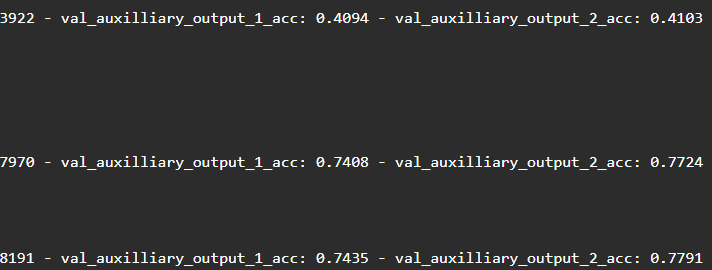
sgd = SGD(lr=initial\_lrate, momentum=0.9, nesterov=False)

lr\_sc = LearningRateScheduler(decay, verbose=1)

model.compile(loss=['categorical\_crossentropy', 'categorical\_crossentropy', 'categorical\_crossentropy'], loss\_weights=[1, 0.3, 0.3], optimizer=sgd, metrics=['accuracy'])

history = model.fit(X\_train, [y\_train, y\_train, y\_train], validation\_data=(X\_test, [y\_test, y\_test, y\_test]), epochs=epochs, batch\_size=256, callbacks=[lr\_sc])

**Output:**

****

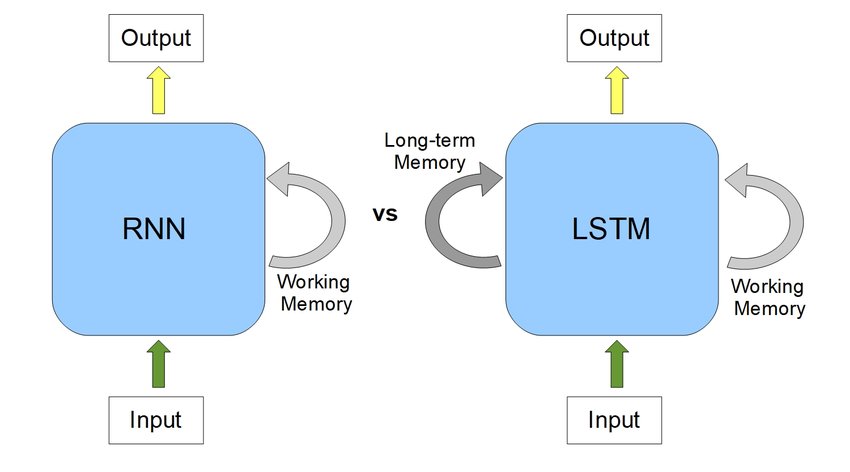
**Practical - 9**

**Statement: Implementation of RNN/LSTM.**

**Theory:** In Neural Networks, we stack up various layers, composed of nodes that contain hidden layers, which are for learning and a dense layer for generating output. But, the central loophole in neural networks is that it does not have memory. Since no memory is associated, it becomes very difficult to work on sequential data like text corpora where we have sentences associated with each other, and even time-series where data is entirely sequential and dynamic.

Here, Recurrent Neural Networks comes to play. RNN addresses the memory issue by giving a feedback mechanism that looks back to the previous output and serves as a kind of memory. Since the previous outputs gained during training leaves a footprint, it is very easy for the model to predict the future tokens (outputs) with help of previous ones.

A sentence or phrase only holds meaning when every word in it is associated with its previous word and the next one. LSTM, short for Long Short Term Memory, as opposed to RNN, extends it by creating both short-term and long-term memory components to efficiently study and learn sequential data. Hence, it’s great for Machine Translation, Speech Recognition, time-series analysis, etc.



**Code:**

data['sentiment'] = ['pos' if (x>3) else 'neg' for x in data['stars']]

data['text'] = data['text'].apply((lambda x: re.sub('[^a-zA-z0-9\s]','',x)))

for idx,row in data.iterrows():

row[0] = row[0].replace('rt',' ')

data['text'] = [x.encode('ascii') for x in data['text']]

tokenizer = Tokenizer(nb\_words=2500, lower=True,split=' ')

tokenizer.fit\_on\_texts(data['text'].values)

#print(tokenizer.word\_index) # To see the dicstionary

X = tokenizer.texts\_to\_sequences(data['text'].values)

X = pad\_sequences(X)

embed\_dim = 128

lstm\_out = 200

batch\_size = 32

model = Sequential()

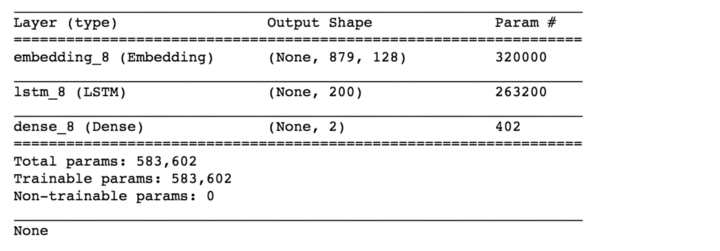
model.add(Embedding(2500, embed\_dim,input\_length = X.shape[1], dropout = 0.2))

model.add(LSTM(lstm\_out, dropout\_U = 0.2, dropout\_W = 0.2))

model.add(Dense(2,activation='softmax'))

model.compile(loss = 'categorical\_crossentropy', optimizer='adam',metrics = ['accuracy'])

print(model.summary())



Y = pd.get\_dummies(data['sentiment']).values

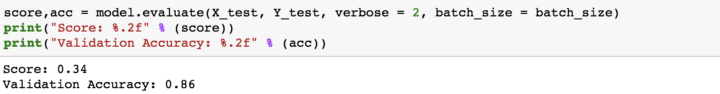
X\_train, X\_valid, Y\_train, Y\_valid = train\_test\_split(X,Y, test\_size = 0.20, random\_state = 36)

#Here we train the Network.

model.fit(X\_train, Y\_train, batch\_size =batch\_size, nb\_epoch = 1, verbose = 5)



**Output:**

****