**EXPERIMENT-4**

**Aim:**

Write a program to implement Artificial Neural Network.

**Platform Used:** Google Colab.

**Theory:**

The term "Artificial Neural Network" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.

Input Layer:

As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:

The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:

The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

**Input:**

import numpy as np

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

input\_layer\_size = 2

hidden\_layer\_size = 1

output\_layer\_size = 1

np.random.seed(0)

weights\_input\_hidden = np.random.randn(input\_layer\_size, hidden\_layer\_size)

biases\_hidden = np.random.randn(hidden\_layer\_size)

weights\_hidden\_output = np.random.randn(hidden\_layer\_size, output\_layer\_size)

biases\_output = np.random.randn(output\_layer\_size)

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

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

epochs = 10000

learning\_rate = 0.1

for epoch in range(epochs):

    # Forward pass

    hidden\_layer\_activation = np.dot(X, weights\_input\_hidden) + biases\_hidden

    hidden\_layer\_output = sigmoid(hidden\_layer\_activation)

    output\_layer\_activation = np.dot(hidden\_layer\_output, weights\_hidden\_output) + biases\_output

    output = sigmoid(output\_layer\_activation)

    # Backward pass

    error = y - output

    output\_gradient = sigmoid\_derivative(output)

    output\_error = error \* output\_gradient

    hidden\_layer\_error = output\_error.dot(weights\_hidden\_output.T)

    hidden\_layer\_gradient = sigmoid\_derivative(hidden\_layer\_output)

    hidden\_layer\_error\_term = hidden\_layer\_error \* hidden\_layer\_gradient

    # Update weights and biases

    weights\_hidden\_output += hidden\_layer\_output.T.dot(output\_error) \* learning\_rate

    biases\_output += np.sum(output\_error) \* learning\_rate

    weights\_input\_hidden += X.T.dot(hidden\_layer\_error\_term) \* learning\_rate

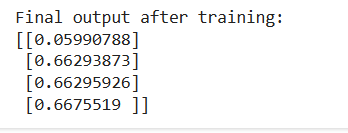
    biases\_hidden += np.sum(hidden\_layer\_error\_term) \* learning\_rate

# Print the final output

print("Final output after training:")

print(output)

**Output:**

****

**EXPERIMENT-5**

**Aim:**

Write a program to implement the CNN.

**Platform Used:** Google Colab.

**Theory:**

CNN stands for Convolutional Neural Network, which is a type of deep neural network commonly used for analyzing visual imagery. CNNs are particularly well-suited for tasks like image recognition and classification.

Key components of a CNN include:

1. **Convolutional layers**: These layers apply convolution operations to the input, using filters (also known as kernels) to extract features from the input data.
2. **Pooling layers**: Pooling layers reduce the spatial dimensions of the input data by down-sampling, which helps in reducing computation and controlling overfitting.
3. **Activation functions**: Typically, ReLU (Rectified Linear Unit) is used as the activation function in CNNs to introduce non-linearity into the network.
4. **Fully connected layers**: These layers connect every neuron in one layer to every neuron in the next layer, similar to a traditional neural network.
5. **Output layer**: The final layer of the CNN produces the output, which could be probabilities for different classes in a classification task or continuous values in a regression task.

CNNs have been highly successful in various applications, including image recognition, object detection, image segmentation, and more. Their ability to automatically learn features from raw data, along with their hierarchical structure, makes them powerful tools for tasks involving visual data.

**Input:**

import numpy as np

class CNN:

    def \_\_init\_\_(self):

        pass

    def convLayer(self, input\_shape, channels, strides, padding, filter\_size):

        pass

    def maxPooling(self, input\_matrix):

        pass

    def flatten(self, input\_matrix):

        pass

    def dropout(self, input\_matrix, dropout\_rate = 0):

        pass

    def convLayer(self, input\_shape, channels, strides, padding, filter\_size):

        height, width = input\_shape

        input\_shape\_with\_channels = (height, width, channels)

        print("Input Shape (with channels):", input\_shape\_with\_channels)

        # for random input and filter matrix

        input\_matrix = np.random.randint(0, 10, size=input\_shape\_with\_channels)

        filter\_matrix = np.random.randint(0, 5, size=filter\_size)

        input\_matrix = np.array([

                    [1, 1, 1, 0, 0],

                    [0, 1, 1, 1, 0],

                    [0, 0, 1, 1, 1],

                    [0, 0, 1, 1, 0],

                    [0, 1, 1, 0, 0]

        ])

        filter\_matrix = np.array([

                    [1, 0, 1],

                    [0, 1, 0],

                    [1, 0, 1]

        ])

        print("\nInput Matrix:")

        print(input\_matrix)

        print("\nFilter Matrix:")

        print(filter\_matrix)

        padding.lower()

        padSize = 0

        if padding == 'same':

            # Calculate padding needed for each dimension

            pad\_height = ((height - 1) \* strides[0] + filter\_size[0] - height) // 2

            pad\_width = ((width - 1) \* strides[1] + filter\_size[1] - width) // 2

            # Apply padding to the input matrix

            input\_matrix = np.pad(input\_matrix, ((pad\_height, pad\_height), (pad\_width, pad\_width),

                                                (0, 0)), mode='constant')

            # Adjust height and width to consider the padding

            height += 2 \* pad\_height

            width += 2 \* pad\_width

        elif padding == 'valid':

            padSize = filter\_size[0] // 2

            print("\nPad Size: ", padSize)

        else:

            return "Invalid Padding!!"

        # output dimension

        conv\_height = (height - filter\_size[0]) // strides[0] + 1

        conv\_width = (width - filter\_size[1]) // strides[1] + 1

        output\_matrix = np.zeros((conv\_height, conv\_width))

        # Convolution Operation

        for i in range(0, height - filter\_size[0] + 1, strides[0]):

            for j in range(0, width - filter\_size[1] + 1, strides[1]):

                receptive\_field = input\_matrix[i:i + filter\_size[0], j:j + filter\_size[1]]

                output\_matrix[i // strides[0], j // strides[1]] = np.sum(receptive\_field \* filter\_matrix)

        return output\_matrix

    def maxPooling(self, input\_matrix, pool\_size, strides\_pooling):

        pool\_height, pool\_width = pool\_size

        stride\_height, stride\_width = strides\_pooling

        pooled\_height = (input\_matrix.shape[0] - pool\_height) // stride\_height + 1

        pooled\_width = (input\_matrix.shape[1] - pool\_width) // stride\_width + 1

        pooled\_matrix = np.zeros((pooled\_height, pooled\_width))

        for i in range(pooled\_height):

            for j in range(pooled\_width):

                patch = input\_matrix[i \* stride\_height: i \* stride\_height + pool\_height,

                                     j \* stride\_width: j \* stride\_width + pool\_width]

                pooled\_matrix[i, j] = np.max(patch)

        return pooled\_matrix

    def flatten(self, input\_matrix):

        return input\_matrix.flatten()

    def dropout(self, input\_matrix, dropout\_rate = 0):

        dropout\_mask = np.random.binomial(1, 1 - dropout\_rate, size=input\_matrix.shape)

        return input\_matrix \* dropout\_mask

input\_shape = (5, 5)

channels = 1

strides = (1, 1)

padding = 'valid'

filter\_size = (3, 3)

cnn\_model = CNN()

conv1 = cnn\_model.convLayer(input\_shape, channels, strides, padding, filter\_size)

conv1

pool\_size = (2, 2)

strides\_pooling = (1, 1)

maxPool = cnn\_model.maxPooling(conv1, pool\_size, strides\_pooling)

maxPool

flattened\_output = cnn\_model.flatten(maxPool)

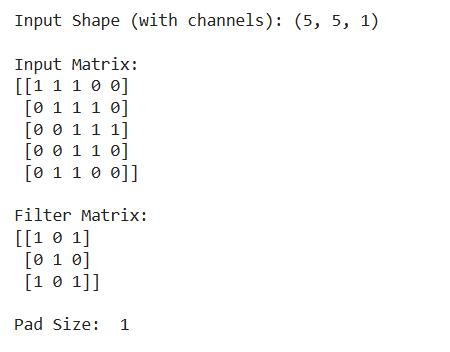
flattened\_output

dropout\_output = cnn\_model.dropout(flattened\_output, 0.3)

dropout\_output

**Output:**

**A number with black text

Description automatically generated with medium confidence**

**A number and number written on a white background

Description automatically generated with medium confidence**



**Experiment -6**

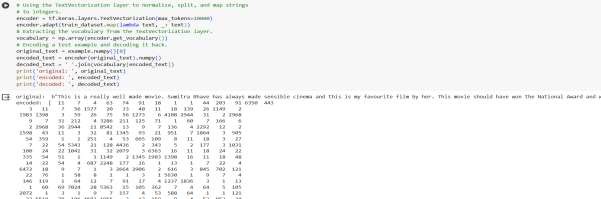
**Aim:**

Implement RNN network.

**Theory:**

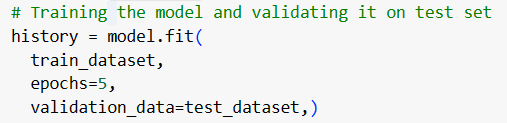
Recurrent Neural Network (RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

**Input:**

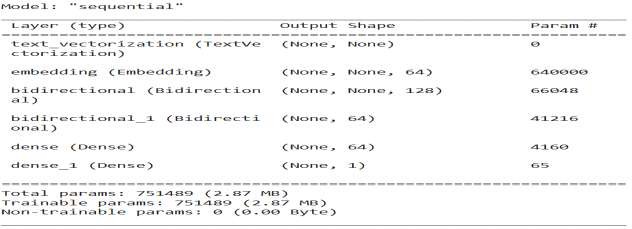


A screen shot of a computer code

Description automatically generated



**Output:**





A graph showing the results of a training and validation accuracy

Description automatically generated

**EXPERIMENT-7**

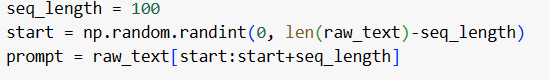
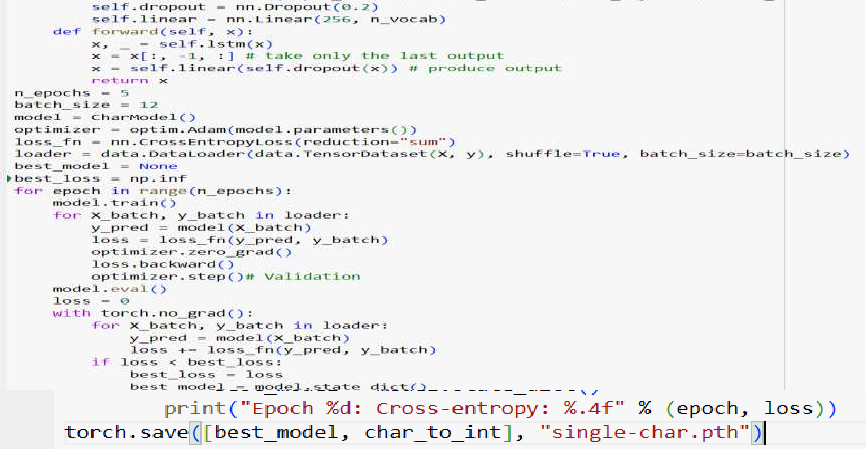
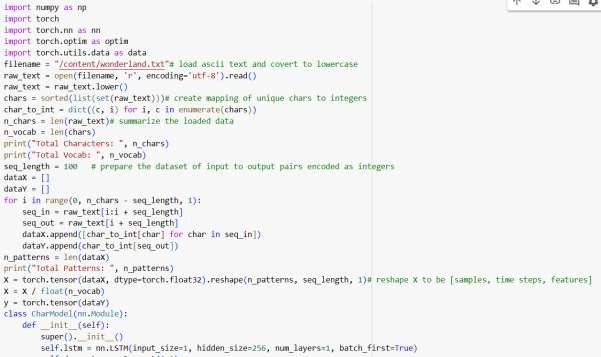
**Aim:** Implement LSTM network.

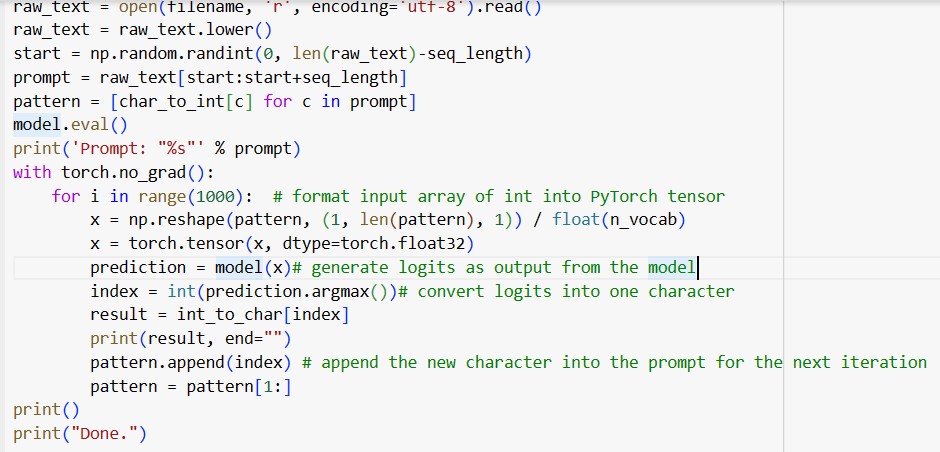
**Platform used:** Colab

**Theory:**

Long Short-Term Memory is an improved version of recurrent neural network designed by Hochreiter & Schmidhuber. LSTM is well-suited for sequence prediction tasks and excels in capturing long-term dependencies. Its applications extend to tasks involving time series and sequences. LSTM’s strength lies in its ability to grasp the order dependence crucial for solving intricate problems, such as machine translation and speech recognition. A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies. LSTMs address this problem by introducing a memory cell, which is a container that can hold information for an extended period. LSTM networks are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, and time series forecasting. LSTMs can also be used in combination with other neural network architectures, such as Convolutional Neural Networks (CNNs) for image and video analysis. The memory cell is controlled by three gates: the input gate, the forget gate, and the output gate. These gates decide what information to add to, remove from, and output from the memory cell. The input gate controls what information is added to the memory cell. The forget gate controls what information is removed from the memory cell. And the output gate controls what information is output from the memory cell. This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.

**Input:**





**Output:**

A screenshot of a computer

Description automatically generated

**Conclusion:**

Successfully implemented the LONG SHORT TERM MEMORY network.

**AMITY SCHOOL OF ENGINEERING & TECHNOLOGY**

AMITY UNIVERSITY CAMPUS, SECTOR-125, NOIDA-201303

A logo of a university

Description automatically generated

**Deep Learning and Neural Network Lab**

PRACTICAL FILE

COURSE CODE: AIML302

Submitted to: Submitted by:

Dr. Pintu Kumar Ram Boddu Asmitha Bhavya

A2305221386

6CSE-6X

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| S.No. | Experiment | Date | Signature |
| 1. | Exploring Kaggle Website and Downloading Dataset to work upon it. |  |  |
| 2. | Write a python program on taking a iris data set from Kaggle and Perform these on that data set 1.missing values 2.outliers 3.reputation |  |  |
| 3. | Program that simulates the operation of an AND and OR gate with adjustable weights |  |  |
| 4. | Implementing Artificial Neural Network. |  |  |
| 5. | Implementing Convolutional Neural Network. |  |  |
| 6. | Implement RNN network |  |  |
| 7. | Implement LSTM network |  |  |
| 8. | Perform the sentiment analysis based on data (user choice data) end to end workflow (loading preprocessing modelling compiling) |  |  |