EXPERIMENT-1

TENSORFLOW INSTALLATION AND SET UP

AIM: Tensor flow in python Installation and Environment Setup.

DESCRIPTION:

TensorFlow: TensorFlow is an open-source platform and framework for machine learning, which includes libraries and tools based on Python and Java — designed with the objective of training machine learning and deep learning models on data.TensorFlow was initially created without considering deep learning for large numerical calculations.

TensorFlow supports data in the shape of tensors, which are multidimensional arrays of greater dimensions. Arrays with several dimensions are highly useful for managing enormous volumes of data.

TensorFlow uses the concept of graphs of data flow with nodes and edges. As the implementation method is in tables and graphs, spreading TensorFlow code over a cluster of GPU-equipped machines is more straightforward.

STEPS TO INSTALL:

Step 1: Install Anaconda

The first step is to download and install Anaconda on your Windows machine. You can download the latest version of Anaconda from the official website:
https://www.anaconda.com/products/individual#windows.

Step 2: Create a New Environment

The next step is to create a new environment for Tensorflow. You can use the following command to create a new environment called tensorflow:

```
conda create -n tensorflow python=3.8
```

This command will create a new environment called tensorflow with Python 3.8 installed.

Step 3: Activate the Environment

After creating the environment, you need to activate it using the following command:

```
conda activate tensorflow
```

Step 4: GPU setup

First you have to install the NVIDIA GPU driver if it’s not already installedNext, use conda to install CUDA and cuDNN.

```
conda install -c conda-forge cudatoolkit=11.2 cudnn=8.1.0
```

Step 5: Install Tensorflow

You can install Tensorflow using the following command:

```
pip install tensorflow
```

This command will download and install the latest version of Tensorflow in your environment.

Step 6: Verify the Installation

To verify that Tensorflow is installed correctly, you can open a Python shell and run the following commands:

Verify the CPU setup:

```
python -c "import tensorflow as tf;
print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

If a tensor is returned, you've installed TensorFlow successfully.

Verify the GPU setup:

```
python -c "import tensorflow as tf;
print(tf.config.list_physical_devices('GPU'))"
```

If the output is a list of GPUs, Tensorflow GPU is successfully installed.

CONCLUSION:

By following these steps, we can quickly set up a new environment for Tensorflow and start building machine learning models.

EXPERIMENT-2

KERAS INSTALLATION AND SET UP

AIM: Keras in Python Installation and Environment setup.

DESCRIPTION:

Keras: Keras is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation. It is relatively easy to learn and work with because it provides a python frontend with a high level of abstraction while having the option of multiple back-ends for computation purposes. This makes Keras slower than other deep learning frameworks, but extremely beginner friendly.

TensorFlow has adopted Keras as its official high-level API. Keras is embedded in TensorFlow and can be used to perform deep learning fast as it provides inbuilt modules for all neural network computations.

Keras has features such as :

- It runs smoothly on both CPU and GPU.
- It supports almost all neural network models.
- It is modular in nature, which makes it expressive, flexible, and apt for innovative research.

STEPS TO INSTALL:

Since, Keras is embedded in Tensorflow, we need to have tensorflow installed in our computers before the installation and set up of Keras Library.

Step-1: Install Anaconda Distribution on your windows machine.

Step-2: Create a new environment for Tensorflow and install Tensorflow in it using pip command.

Step-3: Now that we have Tensorflow set up in our system we can install keras using the following command:

```
pip install keras
```

Step-4: You can check the installation of Keras by importing the library. If no error occurs, then the installation is successful.

CONCLUSION:

By following these steps, we can quickly set up a new environment for Keras and start building machine learning models.

EXPERIMENT-3

2 NODE NEURAL NETWORK

AIM: Implement a 2 node neural network model using keras.

DESCRIPTION:

A two-node neural network is one of the simplest forms of neural networks, consisting of only two computational units or "nodes." These nodes typically represent neurons in a biological analogy. One node serves as the input, where data or features are fed into the network, and the other node acts as the output, producing the network's prediction or output.

In this basic setup, each connection between the input and output nodes has a corresponding weight, which determines the strength of the connection. During training, these weights are adjusted iteratively to minimize the difference between the predicted output and the actual output, using a process called backpropagation.

While a two-node neural network may seem rudimentary, it can still be used to perform simple tasks such as binary classification or regression. However, it's limited in its capacity to handle complex datasets or tasks compared to larger, more complex neural network architectures.

CODE IMPLEMENTATION:

```
import tensorflow as tf
from tensorflow.keras import layers, models


# Define the model architecture
model = models.Sequential([
    layers.Dense(2, activation='relu', input_shape=(2,)),
    layers.Dense(1, activation='sigmoid')
])

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

# Print model summary
model.summary()
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 6

 dense_1 (Dense)             (None, 1)                 3

=================================================================
Total params: 9 (36.00 Byte)
```

Trainable params: 9 (36.00 Byte)
Non-trainable params: 0 (0.00 Byte)

EXPERIMENT-4

ACTIVATION FUNCTIONS

AIM: Implement different activation functions to train Neural Network (Binary Step Activation Function, Linear Activation Function and Non-Linear Activation Function).

DESCRIPTION:

Activation Functions: Activation functions are crucial components of neural networks that introduce non-linearity into the model, allowing it to learn complex patterns and relationships in data. They are applied to the output of each neuron in a neural network, transforming the input signal into an output signal that determines whether the neuron should be activated or not.

Here are brief descriptions of some common activation functions used in neural networks:

- **Sigmoid Activation Function:** The sigmoid function squashes the input values between 0 and 1.It's useful in binary classification tasks as it models the probability of the output being in one of the two classes.
- **ReLU (Rectified Linear Unit) Activation Function:** ReLU function returns 0 for negative inputs and the input value for positive inputs. It's computationally efficient and helps mitigate the vanishing gradient problem, making it widely used in deep learning architectures.
- **TanH (Hyperbolic Tangent) Activation Function**: Similar to the sigmoid function, but squashes the input values between -1 and 1. It's often used in hidden layers of neural networks as it tends to make the output more centered around zero, aiding in training.
- **Leaky ReLU Activation Function:** A variant of ReLU where negative inputs result in a small negative slope instead of zero. It addresses the dying ReLU problem where neurons can become inactive during training.
- **Softmax Activation Function:** Softmax function is typically used in the output layer of a neural network for multi-class classification tasks. It normalizes the output into a probability distribution over multiple classes, allowing the model to make predictions for each class.

CODE IMPLEMENTATION:

(1)Binary Step:

```
import numpy as np
import matplotlib.pyplot as plt

# BinaryStep
def binarystep(x):
  data=[1 if value>0 else 0 for value in x]
  return  np.array(data,dtype=float)
```

```
# Derivative of BinaryStep Function
def der_binarystep(x):
  data=[0 for value in x]

return np.array(data, dtype=float)

# Generating data for Graph
x_data = np.linspace(-10,10)
y_data = binarystep(x_data)
dy_data = der_binarystep(x_data)

# Graph
plt.plot(x_data, y_data, x_data, dy_data)
plt.title('BinaryStep Activation Function & Derivative')
plt.legend(['BinaryStep','der_BinaryStep'])
plt.grid()
plt.show()
```

OUTPUT:

(2)**Linear Activation Function:**

```
import numpy as np
import matplotlib.pyplot as plt

# Linear Activation Function
def linear(x):
  data=[value*5 for value in x]
```

```
return np.array(data,dtype=float)
# Derivative Linear Function
def der_linear(x):

  data=[5 for value in x]
  return np.array(data, dtype=float)

# Generating data for Graph
x_data = np.linspace(-10,10)
y_data = linear(x_data)
dy_data = der_linear(x_data)

# Graph
plt.plot(x_data, y_data, x_data, dy_data)
plt.title('Linear Activation Function & Derivative')
plt.legend(['Linear','der_Linear'])
plt.grid()
plt.show()
```

OUTPUT:

(3)**Sigmoid:**

```
import matplotlib.pyplot as plt
import numpy as np

def sigmoid(x):
    s=1/(1+np.exp(-x))
```

```
ds=s*(1-s)
    return s,ds
x=np.arange(-6,6,0.01)
sigmoid(x)
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=3,
label="sigmoid")
ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=3,
label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```

OUTPUT:

(4)**Tanh Activation Function:**

```
import matplotlib.pyplot as plt
import numpy as np
def tanh(x):
    t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
```

```
dt=1-t**2
    return t,dt
z=np.arange(-4,4,0.01)
tanh(z)[0].size,tanh(z)[1].size
# Setup centered axes
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# Create and show plot
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3, label="tanh")
ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3,
label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```

OUTPUT:

(5)**ReLU Activation Function:**

```
import numpy as np
import matplotlib.pyplot as plt
# Rectified Linear Unit (ReLU)
def ReLU(x):
  data = [max(0,value) for value in x]
  return np.array(data, dtype=float)
```

```
# Derivative for ReLU
def der_ReLU(x):
  data = [1 if value>0 else 0 for value in x]
  return np.array(data, dtype=float)

# Generating data for Graph
x_data = np.linspace(-10,10,100)
y_data = ReLU(x_data)
dy_data = der_ReLU(x_data)

# Graph
plt.plot(x_data, y_data, x_data, dy_data)
plt.title('ReLU Activation Function & Derivative')
plt.legend(['ReLU','der_ReLU'])
plt.grid()
plt.show()
```

OUTPUT:

(6)**Leaky ReLU:**

```
import numpy as np
import matplotlib.pyplot as plt

# Leaky Rectified Linear Unit (leaky ReLU) Activation Function
def leaky_ReLU(x):
  data = [max(0.05*value,value) for value in x]
  return np.array(data, dtype=float)
```

```
# Derivative for leaky ReLU
def der_leaky_ReLU(x):
  data = [1 if value>0 else 0.05 for value in x]
  return np.array(data, dtype=float)

# Generating data For Graph
x_data = np.linspace(-10,10,100)
y_data = leaky_ReLU(x_data)
dy_data = der_leaky_ReLU(x_data)

# Graph
plt.plot(x_data, y_data, x_data, dy_data)
plt.title('leaky ReLU Activation Function & Derivative')
plt.legend(['leaky_ReLU','der_leaky_ReLU'])
plt.grid()
plt.show()
```

OUTPUT:

(7)**Softmax:**

```
import numpy as np

def softmax(Z_data):
    exp_Z_data = np.exp(Z_data)
    #print("exp(Z_data) = ",exp_Z_data)

    sum_of_exp_Z_data = np.sum(exp_Z_data)
```

```
#print("sum of exponentials = ", sum_of_exp_Z_data)
prob_dist = [exp_Zi/sum_of_exp_Z_data for exp_Zi in exp_Z_data]

    return np.array(prob_dist, dtype=float)


Z_data = [1.25,2.44,0.78,0.12] #for cat, dog, tiger, none

p_cat = softmax(Z_data)[0]
print("probability of being cat = ",p_cat)
p_dog = softmax(Z_data)[1]
print("probability of being dog = ",p_dog)
p_tiger = softmax(Z_data)[2]
print("probability of being tiger = ",p_tiger)
p_none = softmax(Z_data)[3]
print("probability of being none = ",p_none)
```

OUTPUT:

```
probability of being cat =  0.19101770813831334
probability of being dog =  0.627890718698843
probability of being tiger =  0.11938650086860875
probability of being none =  0.0617050722942348
```

EXPERIMENT-5

LOGIC GATES USING PERCEPTRONS

AIM: (i) Implement AND/OR Gate with 2-bit Binary Inputs using single layer perceptron.

(ii) Implement NOT Gate with 1-bit Binary Inputs using single layer perceptron.

DESCRIPTION:

(i) Implementing AND/OR gates with a single-layer perceptron means teaching the perceptron to behave like these logic gates. You encode the possible input combinations and their corresponding outputs in binary form, then train the perceptron to find a straight line (or plane in higher dimensions) that separates inputs leading to TRUE outputs from those resulting in FALSE outputs. Once trained, the perceptron can correctly classify new input combinations. This approach works well because AND and OR gates have linearly separable behavior, meaning a single-layer perceptron can learn them effectively.

(ii) Implementing a NOT gate with a single-layer perceptron means teaching the perceptron to flip its input: turning 0 into 1 and 1 into 0. This operation is simple and linearly separable, making it suitable for a single-layer perceptron. During training, the perceptron adjusts its weights and bias to correctly map input values to their corresponding inverted outputs. Once trained, the perceptron can accurately mimic the behavior of the NOT gate.

CODE IMPLEMENTATION:

(i)**AND gate:**

```
import numpy as np

# implementing unit Step
def Steps(v):
    if v<=1:
        return 0
    else:
        return 1

# creating Perceptron
def perceptron(x, w, b):
    v = np.dot(w, x) + b
    y = Steps(v)
    return y

def logic_AND(x):
    w = np.array([2, 2])
    b = +1
    return perceptron(x, w, b)
```

```
def perceptron_algorithm(x):
    # y = w1x1 + b
    y = np.dot(W, x) + b
    # y = 1 if Wx+b > 0 else y = 0
    y = activation_function(y)
    return y

# testing the Perceptron Model
p1 = np.array([0, 1])
p2 = np.array([1, 1])
p3 = np.array([0, 0])
p4 = np.array([1, 0])

print("nAND(0, 1) = {}".format( logic_AND(p1) +
perceptron_algorithm(p1)))
print("nAND(1, 1) = {}".format( logic_AND(p2) +
perceptron_algorithm(p2)))
print("nAND(0, 0) = {}".format( logic_AND(p3) +
perceptron_algorithm(p3)))
print("nAND(1, 0) = {}".format( logic_AND(p4) +
perceptron_algorithm(p4)))
```

OUTPUT:

```
AND(0, 1) = 1
AND(1, 1) = 1
AND(0, 0) = 0
AND(1, 0) = 1
```

(i)OR gate:

```
import numpy as np
def Steps(v):
    if v >= 0:
        return 1
    else:
        return 0
def percept(x, w, b):
    v = np.dot(w, x) + b

    y = Steps(v)
    return y

def OR(x):
    w = np.array([2, 2])
    b = -1
    return percept(x, w, b)

pair1 = np.array([0, 0])
```

```
pair2 = np.array([0, 1])
pair3 = np.array([1, 0])
pair4 = np.array([1, 1])
print("OR(0, 0) = {}".format( OR(pair1)))
print("OR(0, 1) = {}".format( OR(pair2)))
print("OR(1, 0) = {}".format( OR(pair3)))
print("OR(1, 1) = {}".format( OR(pair4)))
```

OUPUT:

```
OR(0, 0) = 0
OR(0, 1) = 1
OR(1, 0) = 1
OR(1, 1) = 1
```

(ii)NOT gate:

```
import numpy as np
# Defining the activation function
def activation_function(y):
    if y > 0:
        y = 1
    elif y <= 0:
        y = 0
    return y
# W = weights of the perceptron model
W = np.array([-1])
# b = bias of the model
b = 1
# Defining the perceptron algorithm
def perceptron_algorithm(x):
    # y = w1x1 + b
    y = np.dot(W, x) + b
    # y = 1 if Wx+b > 0 else y = 0
    y = activation_function(y)
    return y
# Input values to verify the NOT logic
input1 = np.array([0])
input2 = np.array([1])
# Printing the results
print('NOT Logic: \n')
print(f'x = 0 => y = {perceptron_algorithm(input1)}')
print(f'x = 1 => y = {perceptron_algorithm(input2)}')
```

OUTPUT:

```
NOT Logic:
x = 0 => y = 1
x = 1 => y = 0
```

EXPERIMENT-6

XOR PROBLEM USING MULTILAYERED PERCEPTRON

AIM: Implement XOR Problem using Multi-Layered Perceptron.

DESCRIPTION:

To solve the XOR problem with a multi-layered perceptron (MLP), we construct a neural network with an input layer, one or more hidden layers, and an output layer. The hidden layers introduce non-linear transformations, enabling the network to learn complex patterns inherent in XOR logic. During training, we adjust the network's weights and biases using backpropagation and gradient descent. This iterative process minimizes prediction errors by updating parameters to better align predicted outputs with true labels. Through this training, the MLP learns to accurately predict XOR outputs for different input combinations, demonstrating the effectiveness of multi-layered architectures in capturing non-linear relationships.

In summary, the process involves building an MLP architecture with hidden layers to address the non-linear nature of the XOR problem. Through iterative training with backpropagation and gradient descent, the network learns to adjust its parameters, ultimately achieving accurate predictions for XOR inputs. This showcases the power of multi-layered perceptrons in solving complex problems that require capturing non-linear relationships between inputs and outputs.

CODE IMPLEMENTATION:

```
import numpy as np
np.random.seed(0)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(sx):
    return sx * (1 - sx)

# Cost functions.
def cost(predicted, truth):
    return truth - predicted

xor_input = np.array([[0,0], [0,1], [1,0], [1,1]])
xor_output = np.array([[0,1,1,0]]).T

# Lets drop the last row of data and use that as unseen test.
X = xor_input
Y = xor_output

# Define the shape of the weight vector.
num_data, input_dim = X.shape
# Lets set the dimensions for the intermediate layer.
```

```
hidden_dim = 2

W1 = np.random.random((input_dim, hidden_dim))

# Define the shape of the output vector.
output_dim = len(Y.T)
# Initialize weights between the hidden layers and the output
layer.
W2 = np.random.random((hidden_dim, output_dim))

num_epochs = 10000
learning_rate = 1.0

for epoch_n in range(num_epochs):
    layer0 = X
    # Forward propagation.

    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = sigmoid(np.dot(layer1, W2))

    # Back propagation (Y -> layer2)

    # How much did we miss in the predictions?
    layer2_error = cost(layer2, Y)
    # Were we really close? If so, don't change too much.
    layer2_delta = layer2_error * sigmoid_derivative(layer2)

    layer1_error = np.dot(layer2_delta, W2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # update weights
    W2 +=  learning_rate * np.dot(layer1.T, layer2_delta)
    W1 +=  learning_rate * np.dot(layer0.T, layer1_delta)

for x, y in zip(X, Y):
    layer1_prediction = sigmoid(np.dot(W1.T, x)) # Feed the unseen
input into trained W.
    prediction = layer2_prediction = sigmoid(np.dot(W2.T,
layer1_prediction)) # Feed the unseen input into trained W.
    print(int(prediction > 0.5), y)
```

OUTPUT:

```
0 [0]

1 [1]
1 [1]
0 [0]
```

A21126552001

EXPERIMENT-7

THREE LAYER NEURAL NETWORK IN TENSORFLOW

AIM: Implement a simple three layer neural network build in Tensor flow(MNIST Dataset).

Data sizes of the tuples are:

x_train: (60,000 x 28 x 28), y_train: (60,000), x_test: (10,000 x 28 x 28), y_test: (10,000)

DESCRIPTION:

Building a three-layer neural network with TensorFlow for the MNIST dataset involves initial setup, data preprocessing, and model architecture definition. You start by importing TensorFlow and possibly NumPy for data manipulation. Then, you load and preprocess the MNIST dataset, which consists of grayscale images of handwritten digits. Next, you define the neural network architecture, specifying the number of neurons in each layer and selecting appropriate activation functions like ReLU for hidden layers and softmax for the output layer.

Once the architecture is defined, you initialize the network's weights and biases and train the model using optimization algorithms such as stochastic gradient descent or Adam. During training, the model learns from the training dataset, adjusting its parameters to minimize the loss function. Finally, you evaluate the trained model's performance on a separate test dataset to assess its accuracy and generalization ability.

CODE IMPLEMENTATION:

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.datasets import mnist

# i) Function to load data
def load_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train, x_test = x_train / 255.0, x_test / 255.0
    return x_train, y_train, x_test, y_test
# ii) Function to extract training data in batches
def get_batches(x, y, batch_size):
    dataset = tf.data.Dataset.from_tensor_slices((x, y))
    dataset = dataset.shuffle(buffer_size=len(x))
    dataset = dataset.batch(batch_size)
    return dataset

# iii) Initialize variables
epochs = 10
batch_size = 32

# iv) Normalize input images
def normalize_images(x):
    return x / 255.0
```

```
# v) Declare weights
hidden_units = 128
num_classes = 10
initializer = tf.initializers.GlorotUniform()

weights = {
    'hidden': tf.Variable(initializer(shape=(28*28,
hidden_units))),
    'output': tf.Variable(initializer(shape=(hidden_units,
num_classes)))
}

biases = {
    'hidden': tf.Variable(tf.zeros(shape=(hidden_units,))),
    'output': tf.Variable(tf.zeros(shape=(num_classes,)))
}

# vi) Define computations
def forward_pass(x):
    x = tf.reshape(x, shape=(-1, 28*28))
    hidden_layer = tf.nn.relu(tf.matmul(x, weights['hidden']) +
biases['hidden'])
    output_layer = tf.matmul(hidden_layer, weights['output']) +
biases['output']
    return output_layer
# vii) Define loss function
def compute_loss(logits, labels):
    return
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labe
ls=labels, logits=logits))

# viii) Define optimizer
optimizer = optimizers.Adam()
# ix) Training loop
def train_model(model, train_dataset, test_data):
    for epoch in range(epochs):
        for step, (x_batch_train, y_batch_train) in
enumerate(train_dataset):
            with tf.GradientTape() as tape:
                logits = model(x_batch_train)
                loss = compute_loss(logits, y_batch_train)
            gradients = tape.gradient(loss,
model.trainable_variables)
            optimizer.apply_gradients(zip(gradients,
model.trainable_variables))

            if step % 100 == 0:
                test_logits = model(test_data[0])
```

```
test_loss = compute_loss(test_logits, test_data[1])
                test_accuracy =
tf.reduce_mean(tf.cast(tf.equal(tf.argmax(test_logits, axis=1),
test_data[1]), tf.float32))
                print(f"Epoch {epoch + 1}, Step {step}, Loss:
{loss:.4f}, Test Loss: {test_loss:.4f}, Test Accuracy:
{test_accuracy:.4f}")

# Load data
x_train, y_train, x_test, y_test = load_data()

# Initialize model
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(hidden_units, activation='relu'),
    layers.Dense(num_classes)
])

# Compile model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(f
rom_logits=True),
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=epochs)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test,
verbose=2)
print(f'Test accuracy: {test_accuracy}')
```

OUTPUT:

```
11490434/11490434 [==============================] - 0s 0us/step
Epoch 1/10
1875/1875 [==============================] - 11s 6ms/step - loss:
0.2583 - accuracy: 0.9269
Epoch 2/10
1875/1875 [==============================] - 8s 4ms/step - loss:
0.1126 - accuracy: 0.9669
Epoch 3/10
1875/1875 [==============================] - 7s 3ms/step - loss:
0.0770 - accuracy: 0.9772
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss:
0.0587 - accuracy: 0.9817
Epoch 5/10
```

```
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0453 - accuracy: 0.9861
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss:
0.0358 - accuracy: 0.9894
Epoch 7/10
1875/1875 [==============================] - 7s 3ms/step - loss:
0.0286 - accuracy: 0.9914
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss:
0.0233 - accuracy: 0.9927
Epoch 9/10
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0189 - accuracy: 0.9944
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss:
0.0160 - accuracy: 0.9950
313/313 - 1s - loss: 0.0885 - accuracy: 0.9773 - 572ms/epoch -
2ms/step
Test accuracy: 0.9772999882698059
```

EXPERIMENT-8

HOUSE PRICE PREDICTION USING MLP

AIM: Implement House price prediction from kaggle dataset by using Multi Layered Perceptron.

DESCRIPTION:

A multilayer perceptron (MLP) is a type of feedforward artificial neural network.

1.**Architecture:** An MLP consists of fully connected neurons organized into at least three layers:

- Input layer: Receives input data.
- Hidden layer(s): These intermediate layers process information and apply weights to the input.
- Output layer: Produces the final prediction or classification.

2.**Activation Function**: Each neuron in an MLP uses a nonlinear activation function. This nonlinearity allows the network to capture complex relationships in the data. Unlike the original perceptron, which used a Heaviside step function, modern MLPs employ more flexible activation functions.

3.**Purpose:** MLPs are notable for their ability to distinguish data that is not linearly separable. In other words, they can handle complex patterns and perform tasks like classification and regression.

Training: Modern feedforward networks, including MLPs, are trained using the backpropagation method. This involves adjusting the weights based on the error between predicted and actual outputs.

CODE IMPLEMENTATION:

```
import pandas as pd
df = pd.read_csv('/content/housepricedata for DL-8 - housepricedata.csv')
df
```

OUTPUT:

	LotArea	OverallQual	OverallCond	TotalBsmtSF	FullBath	HalfBath	BedroomAbvGr	TotRmsAbvGrd	Fireplaces	GarageArea	AboveMedianPrice	
0	8450	7	5	856	2	1	3	8	0	548	1	
1	9600	6	8	1262	2	0	3	6	1	460	1	
2	11250	7	5	920	2	1	3	6	1	608	1	

	LotArea	OverallQual	OverallCond	TotalBsmtSF	FullBath	HalfBath	BedroomAbvGr	TotRmsAbvGrd	Fireplaces	GarageArea	AboveMedianPrice	
3	9550	7	5	756	1	0	3	7	1	642	0	
4	14260	8	5	1145	2	1	4	9	1	836	1	
...	...	...	...	...	...	...	...	...	...	...	...	...
1455	7917	6	5	953	2	1	3	7	1	460	1	
1456	13175	6	6	1542	2	0	3	7	2	500	1	
1457	9042	7	9	1152	2	0	4	9	2	252	1	
1458	9717	5	6	1078	1	0	2	5	0	240	0	
1459	9937	5	6	1256	1	1	3	6	0	276	0	

1460 rows × 11 columns

```
dataset = df.values
dataset
```

OUTPUT:

```
array([[ 8450,     7,     5, ...,     0,   548,     1],
       [ 9600,     6,     8, ...,     1,   460,     1],
       [11250,     7,     5, ...,     1,   608,     1],
       ...,
       [ 9042,     7,     9, ...,     2,   252,     1],
       [ 9717,     5,     6, ...,     0,   240,     0],
       [ 9937,     5,     6, ...,     0,   276,     0]])
```

```
X = dataset[:,0:10]
Y = dataset[:,10]
from sklearn import preprocessing
min_max_scaler = preprocessing.MinMaxScaler()
X_scale = min_max_scaler.fit_transform(X)
X_scale
```

OUTPUT:

```
array([[0.0334198 , 0.66666667, 0.5       , ..., 0.5       , 0.
,
        0.3864598 ],
       [0.03879502, 0.55555556, 0.875     , ..., 0.33333333,
0.33333333,
        0.32440056],
       [0.04650728, 0.66666667, 0.5       , ..., 0.33333333,
0.33333333,
        0.42877292],
       ...,
       [0.03618687, 0.66666667, 1.        , ..., 0.58333333,
0.66666667,
        0.17771509],
       [0.03934189, 0.44444444, 0.625     , ..., 0.25      , 0.
,
        0.16925247],
       [0.04037019, 0.44444444, 0.625     , ..., 0.33333333, 0.
,
        0.19464034]])
```

```
from sklearn.model_selection import train_test_split
X_train, X_val_and_test, Y_train, Y_val_and_test =
train_test_split(X_scale, Y, test_size=0.3)
X_val, X_test, Y_val, Y_test = train_test_split(X_val_and_test,
Y_val_and_test, test_size=0.5)
print(X_train.shape, X_val.shape, X_test.shape, Y_train.shape,
Y_val.shape, Y_test.shape)

from keras.models import Sequential
from keras.layers import Dense
model = Sequential([
    Dense(32, activation='relu', input_shape=(10,)),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid'),
])
model.compile(optimizer='sgd',
              loss='binary_crossentropy',
              metrics=['accuracy'])
hist = model.fit(X_train, Y_train,
          batch_size=32, epochs=100,
          validation_data=(X_val, Y_val))
```

OUTPUT:

```
Epoch 1/100
32/32 [==============================] - 1s 12ms/step - loss:
0.7031 - accuracy: 0.4501 - val_loss: 0.6969 - val_accuracy:
0.5068
```

```
Epoch 2/100
32/32 [==============================] - 0s 4ms/step - loss:
0.6965 - accuracy: 0.4892 - val_loss: 0.6915 - val_accuracy:
0.5114
Epoch 3/100
32/32 [==============================] - 0s 6ms/step - loss:
0.6916 - accuracy: 0.5000 - val_loss: 0.6870 - val_accuracy:
0.5297
Epoch 4/100
32/32 [==============================] - 0s 4ms/step - loss:
0.6874 - accuracy: 0.5274 - val_loss: 0.6831 - val_accuracy:
0.5616
.................................
.................................
.................................
Epoch 98/100
32/32 [==============================] - 0s 3ms/step - loss:
0.2977 - accuracy: 0.8748 - val_loss: 0.2548 - val_accuracy:
0.9132
Epoch 99/100
32/32 [==============================] - 0s 4ms/step - loss:
0.2977 - accuracy: 0.8796 - val_loss: 0.2519 - val_accuracy:
0.9132
Epoch 100/100
32/32 [==============================] - 0s 4ms/step - loss:
0.2962 - accuracy: 0.8836 - val_loss: 0.2491 - val_accuracy:
0.9178
```

```
model.evaluate(X_test, Y_test)[1]
```

OUTPUT:

```
7/7 [==============================] - 0s 3ms/step - loss: 0.2850
- accuracy: 0.8767
0.8767123222351074
```

```
import matplotlib.pyplot as plt
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper right')
plt.show()
```

OUTPUT:

```
plt.plot(hist.history['accuracy'])
plt.plot(hist.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='lower right')
plt.show()
```

OUTPUT:

EXPERIMENT-9
OPTIMIZERS USING KERAS

AIM: Implement different optimizers on a simple neural network using Keras (Mnist dataset).

DESCRIPTION:

Optimizers are algorithms used to adjust the parameters of a neural network during training in order to minimize the error or loss function. Here are brief descriptions of common optimizers:

Stochastic Gradient Descent (SGD): Updates model parameters in the direction opposite to the gradient of the loss function with respect to those parameters. It computes gradients using a single random sample or a small subset (mini-batch) of the training data, making it computationally efficient but potentially noisy.

Adam (Adaptive Moment Estimation): Combines the benefits of two other extensions of SGD, namely AdaGrad and RMSProp. It computes adaptive learning rates for each parameter based on estimates of first and second moments of gradients. Adam is well-suited for a wide range of optimization problems and often converges faster than traditional SGD.

RMSProp (Root Mean Square Propagation): Similar to AdaGrad, RMSProp adapts the learning rate for each parameter based on the magnitude of its recent gradients. However, it addresses the diminishing learning rate problem of AdaGrad by using an exponentially decaying average of squared gradients. This allows RMSProp to handle non-stationary objectives and converge faster in practice, especially for deep neural networks.

CODE IMPLEMENTATION:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adam, RMSprop


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


model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

```
optimizers = {
    "SGD": SGD(),
    "Adam": Adam(),
    "RMSprop": RMSprop()
}

for optimizer_name, optimizer in optimizers.items():
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    print(f"\nTraining with {optimizer_name} optimizer:")
    model.fit(x_train, y_train, epochs=5, batch_size=32, validation_data=(x_test, y_test))


    test_loss, test_acc = model.evaluate(x_test, y_test)
    print(f'Test accuracy with {optimizer_name} optimizer: {test_acc}')
```

OUTPUT:

Training with SGD optimizer:

```
Epoch 1/5
1875/1875 [==============================] - 10s 5ms/step - loss:
0.6493 - accuracy: 0.8357 - val_loss: 0.3585 - val_accuracy:
0.9047
Epoch 2/5
1875/1875 [==============================] - 5s 3ms/step - loss:
0.3348 - accuracy: 0.9079 - val_loss: 0.2924 - val_accuracy:
0.9197
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.2848 - accuracy: 0.9208 - val_loss: 0.2591 - val_accuracy:
0.9279
Epoch 4/5
1875/1875 [==============================] - 5s 3ms/step - loss:
0.2544 - accuracy: 0.9284 - val_loss: 0.2370 - val_accuracy:
0.9343
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.2318 - accuracy: 0.9354 - val_loss: 0.2181 - val_accuracy:
0.9379
313/313 [==============================] - 1s 2ms/step - loss:
0.2181 - accuracy: 0.9379
```

Test accuracy with SGD optimizer: 0.9379000067710876

Training with Adam optimizer:

```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss:
0.1704 - accuracy: 0.9499 - val_loss: 0.1209 - val_accuracy:
0.9635
```

```
Epoch 2/5
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0953 - accuracy: 0.9718 - val_loss: 0.1079 - val_accuracy:
0.9668
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0675 - accuracy: 0.9787 - val_loss: 0.0890 - val_accuracy:
0.9746
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0521 - accuracy: 0.9835 - val_loss: 0.0745 - val_accuracy:
0.9761
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0407 - accuracy: 0.9875 - val_loss: 0.0873 - val_accuracy:
0.9720
313/313 [==============================] - 1s 2ms/step - loss:
0.0873 - accuracy: 0.9720
```

Test accuracy with Adam optimizer: 0.972000002861023

Training with RMSprop optimizer:

```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss:
0.0267 - accuracy: 0.9915 - val_loss: 0.0766 - val_accuracy:
0.9776
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0219 - accuracy: 0.9932 - val_loss: 0.0697 - val_accuracy:
0.9814
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0182 - accuracy: 0.9942 - val_loss: 0.0690 - val_accuracy:
0.9809
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0165 - accuracy: 0.9952 - val_loss: 0.0759 - val_accuracy:
0.9790
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss:
0.0131 - accuracy: 0.9960 - val_loss: 0.0726 - val_accuracy:
0.9809
313/313 [==============================] - 1s 2ms/step - loss:
0.0726 - accuracy: 0.9809
```

Test accuracy with RMSprop optimizer: 0.98089998960495

EXPERIMENT-10

POOLING OPERATIONS IN CNN

AIM: Implement various pooling operations in Convolution Neural Network.

DESCRIPTION:

Pooling operations in Convolutional Neural Networks (CNNs) are used to reduce the spatial dimensions of feature maps while retaining important information. They help in making the learned features more robust and invariant to small transformations like translations, rotations, and scaling. There are several types of pooling operations commonly used in CNNs:

- **Max Pooling:** Max pooling divides the input feature map into non-overlapping rectangular regions and outputs the maximum value from each region.
- **Average Pooling:** Average pooling computes the average value of each non-overlapping rectangular region in the input feature map.
- **Global Average Pooling:** Global average pooling computes the average of each feature map channel across the entire spatial dimensions.
- **Global Max Pooling:** Global max pooling computes the maximum value of each feature map channel across the entire spatial dimensions.
- **Fractional Pooling:** Fractional pooling divides the input feature map into overlapping regions with fractional sizes and computes the average or maximum value within each region.
- **Adaptive Pooling:** Adaptive pooling allows the network to dynamically adapt the output spatial dimensions based on the input size.

CODE IMPLEMENTATION:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
AveragePooling2D
import numpy as np

# Sample input data
input_data = np.random.rand(1, 4, 4, 1).astype(np.float32)

# Max pooling layer
max_pooling_layer = MaxPooling2D(pool_size=(2, 2), strides=2)

# Define a Keras model with the max pooling layer
model_max_pooling = Sequential([
    Conv2D(1, (3, 3), activation='relu', input_shape=(4, 4, 1)),
    max_pooling_layer
])

# Average pooling layer
```

```
average_pooling_layer = AveragePooling2D(pool_size=(2, 2),
strides=2)


# Define a Keras model with the average pooling layer
model_average_pooling = Sequential([
    Conv2D(1, (3, 3), activation='relu', input_shape=(4, 4, 1)),
    average_pooling_layer
])

# Print summary of the models
print("Model with Max Pooling:")
model_max_pooling.summary()

print("\nModel with Average Pooling:")
model_average_pooling.summary()

# Apply the models to input data
output_max_pooling = model_max_pooling.predict(input_data)
output_average_pooling = model_average_pooling.predict(input_data)

print("\nOutput of Max Pooling:")
print(output_max_pooling)

print("\nOutput of Average Pooling:")
print(output_average_pooling)
```

OUTPUT:

```
Model with Max Pooling:
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 2, 2, 1)           10

 max_pooling2d (MaxPooling2  (None, 1, 1, 1)           0
 D)

=================================================================
Total params: 10 (40.00 Byte)
Trainable params: 10 (40.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Model with Average Pooling:
Model: "sequential_1"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_1 (Conv2D)           (None, 2, 2, 1)           10

 average_pooling2d (Average  (None, 1, 1, 1)           0
 Pooling2D)

=================================================================
Total params: 10 (40.00 Byte)


Trainable params: 10 (40.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
1/1 [==============================] - 0s 251ms/step
1/1 [==============================] - 0s 84ms/step
```

Output of Max Pooling:

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

Output of Average Pooling:

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

EXPERIMENT-11

1D CNN FOR HUMAN ACTIVITY RECOGNITION DATASET

AIM: Develop a 1D Convolutional Neural Network for the human activity recognition dataset.

DESCRIPTION:

1D Convolutional Neural Network (CNN):

A 1D CNN is a type of neural network that specializes in processing sequential data. Unlike traditional image-based CNNs that operate on two-dimensional grids, 1D CNNs process data along a single dimension, making them suitable for tasks where the order of elements matters, such as time-series or sensor data analysis. By applying convolutional operations across the sequence, these networks can learn hierarchical representations of features, capturing patterns and dependencies over time.

Human Activity Recognition (HAR) Dataset:

HAR datasets contain sequential sensor data from wearable devices or smartphones, along with activity labels. They're used for tasks like health monitoring and fitness tracking.

Development Process:

Collect and preprocess HAR dataset.

- Design and train a 1D CNN model to capture temporal patterns.
- Evaluate and fine-tune the model's performance.
- Test the model's generalization on unseen data.
- Deploy the model for real-world HAR applications.

CODE IMPLEMENTATION:

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten,
Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load the dataset
x_train= np.genfromtxt('X_train.txt')
x_test= np.genfromtxt('X_test.txt')
y_train= np.genfromtxt('y_train.txt')
y_test=np.genfromtxt('y_test.txt')

# Encode activity labels
label_encoder = LabelEncoder()
#labels_encoded = label_encoder.fit_transform(y_train)
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
```

```
# Split the data for training and validation
X_train, X_val, y_train_encoded, y_val_encoded =
train_test_split(x_train, y_train_encoded, test_size=0.2,
random_state=42)
# Design the 1D CNN model
model = Sequential([
    Conv1D(64, 3, activation='relu',
input_shape=(X_train.shape[1], X_train.shape[2])),
    MaxPooling1D(2),
    Conv1D(128, 3, activation='relu'),
    MaxPooling1D(2),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(6, activation='softmax')  # Assuming 6 classes, change
as per your dataset
])

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

# Print model summary
model.summary()
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 559, 64)           256

 max_pooling1d (MaxPooling1  (None, 279, 64)           0
 D)

 conv1d_1 (Conv1D)           (None, 277, 128)          24704

 max_pooling1d_1 (MaxPoolin  (None, 138, 128)          0
 g1D)

 flatten (Flatten)           (None, 17664)             0

 dense (Dense)               (None, 128)               2261120

 dense_1 (Dense)             (None, 6)                 774

=================================================================
Total params: 2286854 (8.72 MB)
Trainable params: 2286854 (8.72 MB)
Non-trainable params: 0 (0.00 Byte)
```

```
# Train the model
history = model.fit(X_train, y_train_encoded, epochs=10,
batch_size=64, validation_data=(X_val, y_val_encoded))
```

OUTPUT:

```
Epoch 1/10
92/92 [==============================] - 19s 187ms/step - loss:
0.4911 - accuracy: 0.8114 - val_loss: 0.1556 - val_accuracy:
0.9456
Epoch 2/10
92/92 [==============================] - 16s 170ms/step - loss:
0.1167 - accuracy: 0.9599 - val_loss: 0.0889 - val_accuracy:
0.9674
...........

..........

Epoch 8/10
92/92 [==============================] - 16s 170ms/step - loss:
0.0228 - accuracy: 0.9930 - val_loss: 0.0414 - val_accuracy:
0.9884
Epoch 9/10
92/92 [==============================] - 15s 165ms/step - loss:
0.0183 - accuracy: 0.9935 - val_loss: 0.0877 - val_accuracy:
0.9680
Epoch 10/10
92/92 [==============================] - 15s 162ms/step - loss:
0.0218 - accuracy: 0.9922 - val_loss: 0.0689 - val_accuracy:
0.9728
```

```
# Plot training and validation loss
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

OUTPUT:

```
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test Accuracy: {accuracy}')
```

OUTPUT:

```
46/46 [==============================] - 1s 16ms/step - loss:
0.0396 - accuracy: 0.9884
Test Accuracy: 0.9884432554244995
```

EXPERIMENT-12

RECURRENT NEURAL NETWORK

AIM: Implement a Recurrent Neural network from Scratch for a character-level language model.

DESCRIPTION:

Recurrent Neural Network (RNN):

A Recurrent Neural Network (RNN) is a type of neural network architecture designed to handle sequential data by processing it one element at a time while maintaining an internal memory state. Unlike feedforward neural networks, which process input data independently, RNNs have connections that loop back, allowing them to incorporate information from previous time steps into their computations. This recurrent structure makes RNNs well-suited for tasks involving sequential data, such as time-series prediction, natural language processing, and speech recognition.

Developing an RNN for Character Language Model:

To develop an RNN for a character language model, you would design a neural network architecture that takes sequences of characters as input and learns to predict the next character in the sequence. This involves training the RNN on a dataset of text sequences, such as books, articles, or other sources of text data. During training, the RNN learns to capture the underlying patterns and dependencies in the text data, enabling it to generate coherent and realistic text samples.

CODE IMPLEMENTATION:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import random
from typing import Union
from math import ceil
from os import mkdir
EOS = chr(10) # End of sentence

def build_vocabulary() -> list:
    # builds a vocabulary using ASCII characters
    vocabulary = [chr(i) for i in range(10, 128)]
    return vocabulary

def word2index(vocabulary: list, word: str) -> int:
    # returns the index of 'word' in the vocabulary
    return vocabulary.index(word)

def words2onehot(vocabulary: list, words: list) -> np.ndarray:
    # transforms the list of words given as argument into
```

```
# a one-hot matrix representation using the index in the vocabular
y
    n_words = len(words)
    n_voc = len(vocabulary)
    indices = np.array([word2index(vocabulary, word) for word in w
ords])
    a = np.zeros((n_words, n_voc))
    a[np.arange(n_words), indices] = 1
    return a

def sample_word(vocabulary: list, prob: np.ndarray) -> str:
    # sample a word from the vocabulary according to 'prob'
    # probability distribution (the softmax output of our model)
    return np.random.choice(vocabulary, p=prob)


class Model:
    def __init__(self, vocabulary: list = [], a_size: int = 0):
        self.vocab = vocabulary
        self.vocab_size = len(vocabulary)
        self.a_size = a_size
        self.combined_size = self.vocab_size + self.a_size

        # weights and bias used to compute the new a
        # (a = vector that is passes to the next time step)
        self.wa = tf.Variable(tf.random.normal(
            stddev=1.0/(self.combined_size+self.a_size),
            shape=(self.combined_size, self.a_size),
            dtype=tf.double))
        self.ba = tf.Variable(tf.random.normal(
            stddev=1.0/(1+self.a_size),
            shape=(1, self.a_size),
            dtype=tf.double))

        # weights and bias used to compute y (the softmax predicti
ons)
        self.wy = tf.Variable(tf.random.normal(
            stddev=1.0/(self.a_size+self.vocab_size),
            shape=(self.a_size, self.vocab_size),
            dtype=tf.double))
        self.by = tf.Variable(tf.random.normal(
            stddev=1.0/(1+self.vocab_size),
            shape=(1, self.vocab_size),
            dtype=tf.double))

        self.weights = [self.wa, self.ba, self.wy, self.by]
        self.optimizer = tf.keras.optimizers.Adam()

    def __call__(self,
                 a: Union[np.ndarray, tf.Tensor],
```

```
                x: Union[np.ndarray, tf.Tensor],
                y: Union[np.ndarray, tf.Tensor, None] = None) ->
tuple:

        a_new = tf.math.tanh(tf.linalg.matmul(tf.concat([a, x], ax
is=1), self.wa)+self.ba)
        y_logits = tf.linalg.matmul(a_new, self.wy)+self.by
        if y is None:
            # during prediction return softmax probabilities
            return (a_new, tf.nn.softmax(y_logits))
        else:
            # during training return loss
            return (a_new, tf.math.reduce_mean(
                        tf.nn.softmax_cross_entropy_with_logits(y,
y_logits)))

    def fit(self,
            sentences: list,
            batch_size: int = 128,
            epochs: int = 10) -> None:

        n_sent = len(sentences)
        num_batches = ceil(n_sent / batch_size)

        for epoch in range(epochs):

            random.shuffle(sentences)
            start = 0
            batch_idx = 0

            while start < n_sent:

                print('Training model: %05.2f%%' %
                      (100*(epoch*num_batches+batch_idx+1)/(epochs
*num_batches),),
                      end='\r')

                batch_idx += 1
                end = min(start+batch_size, n_sent)
                batch_sent = sentences[start:end]
                start = end
                batch_sent.sort(reverse=True, key=lambda s: len(s)
)

                init_num_words = len(batch_sent)
                a = np.zeros((init_num_words, self.a_size))
                x = np.zeros((init_num_words, self.vocab_size))

                time_steps = len(batch_sent[0])+1
```

```
with tf.GradientTape() as tape:

                    losses = []
                    for t in range(time_steps):
                        words = []
                        for i in range(init_num_words):
                            if t > len(batch_sent[i]):
                                break
                            if t == len(batch_sent[i]):
                                words.append(EOS)
                                break
                            words.append(batch_sent[i][t])

                        y = words2onehot(self.vocab, words)
                        n = y.shape[0]
                        a, loss = self(a[0:n], x[0:n], y)
                        losses.append(loss)
                        x = y

                    loss_value = tf.math.reduce_mean(losses)

                grads = tape.gradient(loss_value, self.weights)
                self.optimizer.apply_gradients(zip(grads, self.wei
ghts))

    def sample(self) -> str:
        # sample a new sentence from the learned model
        sentence = ''
        a = np.zeros((1, self.a_size))
        x = np.zeros((1, self.vocab_size))
        while True:
            a, y_hat = self(a, x)
            word = sample_word(self.vocab, tf.reshape(y_hat, (-1,)
))
            if word == EOS:
                break
            sentence += word
            x = words2onehot(self.vocab, [word])
        return sentence

    def predict_next(self, sentence: str) -> str:
        # predict the next part of the sentence given as parameter
        a = np.zeros((1, self.a_size))
        for word in sentence.strip():
            x = words2onehot(self.vocab, [word])
            a, y_hat = self(a, x)
        s = ''
        while True:
            word = sample_word(self.vocab, tf.reshape(y_hat, (-1,)
))
            if word == EOS:
```

```
break
            s += word
            x = words2onehot(self.vocab, [word])
            a, y_hat = self(a, x)
        return s

    def save(self, name: str) -> None:
        mkdir(f'./{name}')
        with open(f'./{name}/vocabulary.txt', 'w') as f:
            f.write(','.join(self.vocab))
        with open(f'./{name}/a_size.txt', 'w') as f:
            f.write(str(self.a_size))
        np.save(f'./{name}/wa.npy', self.wa.numpy())
        np.save(f'./{name}/ba.npy', self.ba.numpy())
        np.save(f'./{name}/wy.npy', self.wy.numpy())
        np.save(f'./{name}/by.npy', self.by.numpy())

    def load(self, name: str) -> None:
        with open(f'./{name}/vocabulary.txt', 'r') as f:
            self.vocab = f.read().split(',')
        with open(f'./{name}/a_size.txt', 'r') as f:
            self.a_size = int(f.read())

        self.vocab_size = len(self.vocab)
        self.combined_size = self.vocab_size + self.a_size

        self.wa = tf.Variable(np.load(f'./{name}/wa.npy'))
        self.ba = tf.Variable(np.load(f'./{name}/ba.npy'))
        self.wy = tf.Variable(np.load(f'./{name}/wy.npy'))
        self.by = tf.Variable(np.load(f'./{name}/by.npy'))
        self.weights = [self.wa, self.ba, self.wy, self.by]


df = pd.read_csv('../input/million-headlines/abcnews-date-text.csv
')
df
```

OUTPUT:

	publish_date	headline_text
0	20030219	aba decides against community broadcasting lic...
1	20030219	act fire witnesses must be aware of defamation

	publish_date	headline_text
2	20030219	a g calls for infrastructure protection summit
3	20030219	air nz staff in aust strike for pay rise
4	20030219	air nz strike to affect australian travellers
...	...	...
1226253	20201231	what abc readers learned from 2020 looking bac...
1226254	20201231	what are the south african and uk variants of ...
1226255	20201231	what victorias coronavirus restrictions mean f...
1226256	20201231	whats life like as an american doctor during c...
1226257	20201231	womens shed canberra reskilling unemployed pan...

```
vocabulary = build_vocabulary()
sentences = df['headline_text'].values.tolist()
model = Model(vocabulary, 1024)
model.fit(sentences, batch_size=4096, epochs=50)
```

OUTPUT:

```
Training model: 100.00%
```

```
model.save('news_headlines_model')
# model.load('news_headlines_model')
```

```
for i in range(20):
    print(model.sample())
```

```
returls
thees to fover
christs geths hites
words to new grina illedra rouls deating ajustry million at stades
exter tehtal it roitsed remaid fitiftte tolled saseeks sworibit sp
oadaron jail
usee pont weachars
drignt mastle suysay armwya christmast ban brishtire
commar ashamfusting to says racistant as rulests
villand wiotinops
ships
sexicans
mexical gathy dronses to zeares po inquitic and lionst del rings
treberter told jot cerms card prelither istamatations coostared
amation
nrn fruswertion
wayar hompital acoirs natiomitions
greenders tombam murders says
sedera phes abood grate
mccertson awasters ompitions lakes aster fallers tiss timitishance
budgets cobandancy in de pahide investrials word on bust agaits sr
am artions setuttor whinds
clear ranalidn as ralls
```

```
s = 'scientists just discovered'
s += model.predict_next(s)
s
```

OUTPUT:

```
'scientists just discovered brane janled on the thyed for ta the i
bleach hay thdonkkers out of waters marling parinenser mixised dea
th month san mentriats gereralt hirting grefeding accters finices'
```

EXPERIMENT-13

GRU MODEL

AIM: Implement GRU Model from Scratch

DESCRIPTION:

Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture designed to address some of the limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in capturing long-term dependencies in sequential data. Here's a brief description of the GRU model:

Architecture:

The GRU architecture consists of recurrent units with gated mechanisms that control the flow of information within the network. Each unit maintains a hidden state vector that captures relevant information from past time steps and updates it with new information from the current input.

Gating Mechanisms:

GRUs incorporate two gating mechanisms: the update gate and the reset gate. These gates regulate the flow of information through the network, allowing it to selectively update the hidden state based on the input and the current state.

- **Update Gate:** Controls how much of the previous hidden state should be preserved and how much of the new state should be incorporated. It decides whether to update the hidden state with new information or maintain the previous state.
- **Reset Gate**: Determines how much of the previous state should be forgotten or reset. It allows the model to selectively forget irrelevant information from past time steps.

CODE IMPLEMENTATION:

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
# Hyperparameters
sequence_length = 28  # MNIST image height
input_size = 28  # MNIST image width
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 100
num_epochs = 5
learning_rate = 0.001
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
```

```
# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.To
Tensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToT
ensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
# Define GRU model
class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
num_classes):
        super(GRUModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Set initial hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0),
self.hidden_size).to(device)

        # Forward propagate GRU
        out, _ = self.gru(x, h0)  # out: tensor of shape
(batch_size, seq_length, hidden_size)

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out

model = GRUModel(input_size, hidden_size, num_layers,
num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, sequence_length,
input_size).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step,
loss.item()))
```

OUTPUT:

```
Epoch [1/5], Step [100/600], Loss: 0.7451
Epoch [1/5], Step [200/600], Loss: 0.4473
Epoch [1/5], Step [300/600], Loss: 0.2518
Epoch [1/5], Step [400/600], Loss: 0.1746

.........

.........

Epoch [5/5], Step [200/600], Loss: 0.0295
Epoch [5/5], Step [300/600], Loss: 0.0128
Epoch [5/5], Step [400/600], Loss: 0.0728
Epoch [5/5], Step [500/600], Loss: 0.0647
Epoch [5/5], Step [600/600], Loss: 0.0397
```

```
# Test the model
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, sequence_length,
input_size).to(device)
        labels = labels.to(device)
        outputs = model(images)
```

```
_, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {}
%'.format(100 * correct / total))
```

OUTPUT:

Test Accuracy of the model on the 10000 test images: **98.53 %**

EXPERIMENT-14

LSTM MODEL

AIM: Implement LSTM Model from Scratch.

DESCRIPTION:

Long Short-Term Memory (LSTM):

LSTM is a type of recurrent neural network designed to process sequential data by maintaining memory cells and using gates to control the flow of information.

Key Components:

- Memory Cells: Store information over time.
- Gates: Forget, input, and output gates regulate information flow into and out of the cells.
- Cell State Update: Forget gate determines what to discard, input gate decides what to store, and the new cell state combines these.
- Hidden State Update: Output gate controls which parts of the cell state are used to compute the output.
- Activation Functions: Sigmoid functions for gates, hyperbolic tangent for computing cell state.
- Training Mechanism: LSTMs are trained using backpropagation through time (BPTT), often with techniques like gradient clipping and regularization.
- LSTMs are effective for tasks requiring modeling of long-term dependencies in sequential data, such as natural language processing and time-series prediction.

CODE IMPLEMENTATION:

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms


# Hyperparameters
sequence_length = 28  # MNIST image height
input_size = 28  # MNIST image width
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 100
num_epochs = 5
learning_rate = 0.001
```

```
#Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')

# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.To
Tensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToT
ensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
# Define LSTM model
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
num_classes):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Set initial hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0),
self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers, x.size(0),
self.hidden_size).to(device)

        # Forward propagate LSTM
        out, _ = self.lstm(x, (h0, c0))  # out: tensor of shape
(batch_size, seq_length, hidden_size)

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out
```

```
model = LSTMModel(input_size, hidden_size, num_layers,
num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, sequence_length,
input_size).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step,
loss.item()))
```

OUTPUT:

```
Epoch [1/5], Step [100/600], Loss: 0.7166
Epoch [1/5], Step [200/600], Loss: 0.2609
Epoch [1/5], Step [300/600], Loss: 0.2818
Epoch [1/5], Step [400/600], Loss: 0.2170

..........

..........

Epoch [5/5], Step [300/600], Loss: 0.0639
Epoch [5/5], Step [400/600], Loss: 0.0871
Epoch [5/5], Step [500/600], Loss: 0.0207
Epoch [5/5], Step [600/600], Loss: 0.0270
```

```
# Test the model
with torch.no_grad():
    correct = 0
    total = 0
```

```
for images, labels in test_loader:
        images = images.reshape(-1, sequence_length,
input_size).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {}
%'.format(100 * correct / total))
```

OUTPUT:

```
Test Accuracy of the model on the 10000 test images: 98.41 %
```

EXPERIMENT-15

SENTIMENTAL ANALYSIS

AIM: Implement a Sentimental Analysis as a text classification task for large movie review dataset.

DESCRIPTION:

Sentiment analysis, also known as opinion mining, is a natural language processing (NLP) technique that involves analyzing text data to determine the sentiment or emotion expressed within it. The goal of sentiment analysis is to understand the underlying sentiment conveyed by a piece of text and categorize it as positive, negative, or neutral.

- **Text Input:** Sentiment analysis starts with input text data, which can come from various sources such as social media posts, customer reviews, product feedback, news articles, and more.
- **Text Processing**: The input text undergoes preprocessing steps to clean and normalize the data. This typically involves removing punctuation, stopwords, and irrelevant characters, as well as tokenization to split the text into individual words or phrases.
- **Feature Extraction:** After preprocessing, features are extracted from the text data to represent its underlying sentiment. Common techniques for feature extraction include Bag-of-Words, TF-IDF (Term Frequency-Inverse Document Frequency), word embeddings (such as Word2Vec or GloVe), or deep learning-based representations.
- **Sentiment Classification:** Once features are extracted, a machine learning model or algorithm is trained to classify the sentiment of the text. This involves assigning labels such as positive, negative, or neutral to each piece of text based on its underlying sentiment.
- **Model Evaluation:** The trained sentiment analysis model is evaluated using labeled test data to assess its performance. Evaluation metrics such as accuracy, precision, recall, and F1-score are used to measure how well the model predicts sentiment.
- **Application:** Finally, the trained sentiment analysis model can be deployed for real-world applications such as social media monitoring, customer feedback analysis, brand reputation management, market research, and sentiment-aware recommendation systems.

CODE IMPLEMENTATION:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset
df = pd.read_csv('IMDB Dataset.csv')
```

```
# Preprocessing
# Assuming 'review' column contains the text reviews and
'sentiment' column contains the sentiment labels
X = df['review']
y = df['sentiment']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Feature extraction using TF-IDF
tfidf_vectorizer = TfidfVectorizer(max_features=5000)
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)

# Model training
model = LogisticRegression()
model.fit(X_train_tfidf, y_train)

# Predictions
y_pred = model.predict(X_test_tfidf)

# Model evaluation
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print(classification_report(y_test, y_pred))
```

OUTPUT:

```
Accuracy: 0.8959
              precision    recall  f1-score   support

    negative       0.90      0.88      0.89      4961
    positive       0.89      0.91      0.90      5039

    accuracy                           0.90     10000
   macro avg       0.90      0.90      0.90     10000
weighted avg       0.90      0.90      0.90     10000
```