DEEP LEARNING LAB MANUAL

1.Implementing Matrix Operations Using Numpy

a. Basic Operations (Add, Sub, Mul, Dot, Transpose, Inverse,)
b. Eigen Values And Eigen Vectors
c. Single Value Decomposition

2) Illustrate Bionomial, Piosson And Normal Distribution Using Scipy.Stats Library

3) Implement A Simple Feedforward Neural Network Using A Library Like Tensorflow Or Keras

4) Implement Various Optimizers

a. Implementing Batch Gradient Descent Algorithm

b. Implementing Stochastic Batch Gradient Descent Algorithm(Calculate Gradient And New Weight For Each Sample)

c. Implementing Mini Batch Gradient Descent Algorithm(Calculate Gradient And New Weight For Each Mini Batch)

5.CNN Implementation On MNIST Dataset

6. Apply RNN For Autocompletion

7. Apply RNN For Language Translation

8.Apply LSTM For Autocompletion

Aim:

Implementing Matrix Operations Using Numpy

a. **Basic Operations (Add, Sub, Dot, Transpose, Inverse)**

b. **Eigen Values And Eigen Vectors**

c. **Single Value Decomposition**

Procedure:

1. Import the NumPy Library

- Begin the program by importing the NumPy package using
- import numpy as np
- NumPy provides in-built functions for matrix and numerical computations.

2. Define Matrix A and B

- Create two 3×3 matrices using np.array().
- These matrices are used for performing basic operations like addition, subtraction, and multiplication.

3. Perform Matrix Addition

- Use the function np.add(A, B) to add corresponding elements of the matrices.
- Display the resulting matrix.

4. Perform Matrix Subtraction

- Use np.subtract(A, B) to subtract matrix B from matrix A element-wise.
- Print the output matrix.

5. Perform Matrix Multiplication

- Use np.dot(A, B) to multiply matrix A with matrix B following standard matrix multiplication rules.
- Display the obtained matrix.

6. Find the Transpose of a Matrix

- Define matrix T and use np.transpose(T) to compute its transpose.
- Print both the original and transposed matrices.

7. Calculate the Inverse of a Matrix

- Use np.linalg.inv(T) to find the inverse of matrix T, provided it is non-singular.

- Print the inverse matrix.

8. Calculate the Determinant of a Matrix
   - Use np.linalg.det(T) to compute the determinant of matrix T.
   - Print the determinant value.
9. Compute Eigenvalues and Eigenvectors
   - Define a symmetric matrix X.
   - Use np.linalg.eig(X) to compute eigenvalues and eigenvectors.
   - Display eigenvalues and the corresponding eigenvectors.
10. Perform Singular Value Decomposition (SVD)
    - Define matrix Y and apply np.linalg.svd(Y) to compute:
      - U matrix
      - Sigma (singular values)
      - V^t matrix
    - Display all three components.
11. Calculate the Trace of a Matrix
    - Use np.trace(T) to compute the trace (sum of diagonal elements) of matrix T.
    - Print the trace value.
12. Display All Results
    - Print each matrix operation result in a proper formatted manner..

Program

```
# Matrix operations

import numpy as np

A = np.array([[2,2,2],[2,2,2],[2,2,2]])

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

print("The Given A Matrix is\n", A)

print("The Given B Matrix is \n", B)

#Matrix Addition

C =np.add(A,B)

print("Matrix Addition of A and B is\n",C)

#Matrix Subtraction

D = np.subtract(A,B)

print("Matrix Subtraction of A and B is \n",D)

#Matrix Multilication

E = np.dot(A,B)

print("Matrix Multiplication of A and B is \n", E)

#Matrix Transpose

T = np.array([[1,2,5],[4,3,2],[3,2,4]])

Tran = np.transpose(T)

print("The Original matrix T is \n",T)

print("Transpose of the T matrix is \n ", Tran)

# Matrix Inverse

A_Inv = np.linalg.inv(T)

print("Inverse of the T Matrix\n", A_Inv)


# Determinant of the Given T matrix

A_det = np.linalg.det(T)

    print("Determinant of the T   Matrix is \n", A_det)

    # Eigen Values and Eigen Vectors X=np.array([[8,-6,2],[-6,7,-4],[2,-4,3]])

    print("The Given Matrix X is \n",X) eig_val,eig_vec=np.linalg.eig(X)

    j=[]
```

```
    for i ineig_val:
        j.append(int(i))
    print("eigen values are:\n",j) print("eigen vectors are:\n",eig_vec)
    # Singular Value Decomposition Y=np.array([[3,1,1],[-1,3,1]])
    u,sm,vt=np.linalg.svd(Y)
  print("The SVD values are ")
 print("U value is \n",u)
 print("Summation value is \n",sm)
print("Transpose of V is \n",vt)
print("The Given matrix T is \n",T)
# Trace of the Matrix T
tr=np.trace(T)
print("Trace of the Matrix T is :",tr)
```

AIM:

2) Illustrate Bionomial, Piosson And Normal Distribution Using Scipy.Stats Library

a.Procedure

1. Import Required Library
   Import the binom class from scipy.stats to work with the binomial distribution.
2. Define Number of Trials (n)
   Set the total number of independent trials (e.g., number of coin tosses).
3. Define Probability of Success (p)
   Assign the probability of success for each trial.
   For a fair coin, p = 0.5.
4. Compute the Probability Mass Function (PMF)
   Use binom.pmf(x, n, p) to calculate the probability of getting x successes for all possible values of x from 0 to n.
5. Calculate the Mean
   Compute the mean of the binomial distribution using
   binom.mean(n, p).
6. Calculate the Variance
   Compute the variance using
   binom.var(n, p).
7. Display the Results
   Print the PMF values along with the mean and variance.

b.Procedure

1. Import Required Library
   Import the poisson class from scipy.stats.
2. Define the Mean (λ)
   Set the average number of occurrences per fixed interval (time/space).
3. Define the Random Variable (x)
   Specify the number of occurrences for which the probability is to be calculated.
4. Compute the Probability Mass Function (PMF)
   Use poisson.pmf(x, λ) to calculate the probability of exactly x events occurring.
5. Compute Cumulative Probability (Optional)
   Use poisson.cdf(x, λ) to calculate cumulative probability up to x events.
6. Calculate Probability of More Than x Events (Optional)
   Use
   1 − poisson.cdf(x, λ) to find the probability of more than x events.
7. Display the Results
   Print the computed probabilities clearly.

Find Mean, Variance of the Random Variable X that represents No. of Heads in tossing 2 coins

Solution :

Experiment : Tossing of 2 coins

S : {HH,HT,TH,TT}

X: No .of heads = { 0,1,2}

X	P(x)	X.P(X)	X^2	X^2.P(X)
0	1/4	0	0	0
1	2/4	2/4	1	2/4
2	1/4	2/4	4	1
E(X) = 1			$E(X^2) = 3/2$	

Mean = E(X) = 1

Variance = $E(X^2) - (E(x))^2 = 3/2 - (1)^2 = 1/2$

PROGRAM

```
from scipy.stats import binom # Number of coin tosses
n = 2

# Probability of getting heads (success) p = 0.5   # Assuming a
fair coin
# Calculate the PMF using binomial distribution pmf = [binom.pmf(r, n, p)
for r in range(n + 1)]

# Calculate mean and variance using binomial distribution
mean = binom.mean(n, p)
variance = binom.var(n, p) print("PMF of X:", pmf)
print("Mean of the random variable X:", mean) print("Variance of the random variable X:",
variance)
```

Find Mean, Variance of the Random Variable X that represents No.of Heads in Tossing 3 coins

Solution:

Experiment : Tossing 3 Coins

S = {HHH, HHT, HTT, HTH,TTT, TTH,THH,THT}

X = No. of Heads

= {0,1,2,3}

X	Probabilities P(x)	x.P(x)	X^2	$X^2.P(X)$
0	1/8	0	0	0
1	3/8	3/8	1	3/8
2	3/8	¾	4	3/2
3	1/8	3/8	9	9/8
$E(X) = \sum x.P(X) = 1.5$			$E(X^2) = \sum X^2.P(X) = 3$	

The Mean = E(X) = 1.5

The Variance = $E(X^2) - (E(x))^2$ $=3-(1.5)^2 = 0.75$

Program

```
from scipy.stats import binom
n = 3   # Number of trials (tossing 3 coins)
p = 0.5   # Probability of success (getting heads on a single coin toss)

# Calculate the PMF for 0, 1, and 2 heads print("x \t\t  P(X=x)")
for i in range(0,4): b=binom.pmf(i,n,p)
        print(i,"\t\t",b)

mean = binom.mean(n,p) var =
binom.var(n,p) print("Mean", mean)
print("Variance",var)
```

POISSON DISTRIBUTION ILLUSTRATION

Example:

The mean number of power outages in the city of Brunswick is 4 per year. Find the probability that in a given year,

a.) there are exactly 3 outages,

b.) there are more than 3 outages.

a.) $\mu = 4, \quad x = 3$

$$P(3) = \frac{4^3(2.71828)^{-4}}{3!}$$

$$\approx 0.195$$

b.) $P(\text{more than } 3)$

$$= 1 - P(x \le 3)$$

$$= 1 - [P(3) + P(2) + P(1) + P(0)]$$

$$= 1 - (0.195 + 0.147 + 0.073 + 0.018)$$

$$\approx 0.567$$

```
from scipy.stats import poisson mean = 4
# a) probability of exactly 3 outages x = 3
prob_x = poisson.pmf(x, mean)
print("probability of exactly 3 outages in a year:", prob_x)

 # b) probability of more than 3 outages
y = 3
prob_y = 1 - poisson.cdf(y, mean)
print("probability of more than 3 outages in a year:", prob_y)
```

POISSON DISTRIBUTION ILLUSTRATION

At a small walk-in clinic, an average of five patients arrive at the clinic per hour during opening hours. What is the probability that exactly three patients will arrive in the next hour?

k = 3 patients
λ = 5 patients/hour
e = 2.718

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

$$P(X = 3) = \frac{(2.718^{-5})(5^3)}{3!}$$

$$P(X = 3) = \frac{(0.00674)(125)}{6}$$

$$P(X = 3) = 0.14$$

```
from scipy.stats import poisson

# Average no.of patients arrive per hour
mean = 5

# Probability of exactly 3 patients arrive at next hour
x = 3
prob_x = poisson.pmf(x,mean)
print("Probability of exactly 3 Patients arrive at next hour is :",prob_x)
```

3) Implement A Simple Feedforward Neural Network Using A Library Like Tensorflow Or Keras

AIM: SINGLE LAYER PERCEPTRON ILLUSTRATION

PROCEDURE

Step 1: Import Required Libraries

- Import NumPy for numerical operations.
- Import Keras modules (Sequential, Dense) to build the neural network.

Step 2: Prepare the Dataset

- Generate random input values consisting of two numbers per sample.
- Calculate the sum of the two numbers as the target output.
- This forms a supervised learning dataset.

Step 3: Initialize the Neural Network Model

- Create a Sequential model.
- Add layers to the model in a feedforward manner.

Step 4: Add Network Layers

- Add a hidden layer with multiple neurons and ReLU activation.
- Add an output layer with one neuron to predict the sum.

Step 5: Compile the Model

- Select Mean Squared Error (MSE) as the loss function.
- Choose an optimizer such as Stochastic Gradient Descent (SGD).
- Compilation prepares the model for training.

Step 6: Train the Model

- Train the model using the training data.
- Run training for a fixed number of epochs.
- During training, the model adjusts its weights to minimize the loss.

Step 7: Test the Model

- Provide new input values that were not used during training.
- Use the trained model to predict the sum of the test inputs.

Step 8: Display and Analyze Results

- Print the predicted outputs.
- Compare predicted sums with actual sums to verify accuracy.

Algorithm: Single Layer Perceptron (SLP)

1. **Start**
2. **Initialize parameters**
    1. Initialize weight vector **w** with small random values.
    2. Initialize bias **b**.
    3. Set learning rate **η**.
    4. Set number of epochs **N**.
3. **Provide training dataset**
    1. Input vectors $x = [x_1, x_2, \ldots x_n]$
    2. Target outputs y
4. **For each epoch (1 to N), do:**

    a. For each training sample:

    1. Compute net input

$$net = \sum_{i=1}^{n} w_i x_i + b$$

    1. Apply activation function (step/sign function):

$$\hat{y} = \begin{cases} 1, & if\, net \geq 0 \\ 0, & if\, net < 0 \end{cases}$$

    1. Calculate error

$$error = y - \hat{y}$$

    1. Update weights and bias

$$w_i = w_i + \eta \times error \times x_i$$
$$b = b + \eta \times error$$

Repeat until convergence or maximum epochs reached
Output final weights and bias
Stop

Program:

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# Generate training data
x = np.random.random((1000, 2))
y = np.sum(x, axis=1)

# Display first 100 training samples
for i in range(100):
   print("sum of", x[i], "is", y[i])

# Build the model
model = Sequential()
model.add(Dense(25, activation="relu", input_dim=2))
model.add(Dense(1))

# Compile the model
model.compile(loss='mse', optimizer='sgd')
```

```
# Train the model
model.fit(x, y, epochs=30)

# Test data
x_test = np.array([[0.2, 0.1],
                   [0.4, 0.3],
                   [0.6, 0.5]])

# Predict output
y_test = model.predict(x_test)

# Display results
for i in range(len(y_test)):
    print(f"sum of {x_test[i]} is {y_test[i]}")
```

4) Implement Various Optimizers

a. Implementing Batch Gradient Descent Algorithm
b. Implementing Stochastic Batch Gradient Descent Algorithm(Calculate Gradient And New Weight For Each Sample)
c. Implementing Mini Batch Gradient Descent Algorithm(Calculate Gradient And New Weight For Each Mini Batch)

Implementing Batch Gradient Descent Algorithm

EPOCH-1						
X	Y	W	S	A	O	Gradient
1	2	4	4	4	4	2
2	4	4	8	8	8	8
3	6	4	12	12	12	18
4	8	4	16	16	16	32
Gradient 15				new weight 2.5		

EPOCH-2						
X	Y	W	S	A	O	Gradient
1	2	2.5	2.5	2.5	2.5	0.5
2	4	2.5	5	5	5	2
3	6	2.5	7.5	7.5	7.5	4.5
4	8	2.5	10	10	10	8
Gradient 3.75				new weight 2.125		

EPOCH-3						
X	Y	W	S	A	O	Gradient
1	2	2.125	2.125	2.125	2.125	0.125
2	4	2.125	4.25	4.25	4.25	0.5
3	6	2.125	6.375	6.375	6.375	1.125
4	8	2.125	8.5	8.5	8.5	2
Gradient 0.9375						

EPOCH-4						
X	Y	W	S	A	O	Gradient
1	2	2.03125	2.03125	2.03125	2.03125	0.03125
2	4	2.03125	4.0625	4.0625	4.0625	0.125
3	6	2.03125	6.09375	6.09375	6.09375	0.28125
4	8	2.03125	8.125	8.125	8.125	0.5
				new weight 2.00781		

X(Input)	Y(Output)	S(Summation)	A(Activation)	O(Predicted Output)
S= WX	A=F(S) =S O=A	Gradient = (O-Y)*X,	New Weight = Old_weight – 0.1*Gradient	

Column	Meaning
X	Input value
Y	Actual output
W	Current weight
S	Summation: $S = W \times X$
A	Activation: $A = S$ (Identity function)
O	Predicted Output: $O = A$
Gradient	$(O - Y) \times X$

Algorithm

Initialize
Initialize the weight parameters θ(randomly or with predefined values).
Repeat for each epoch (from 1 to E):
Initialize gradient accumulator:

$$G = 0$$

For all training samples $i = 1$to n:
Compute model output:

$$\hat{y}_i = f(x_i, \theta)$$

Compute error:

$$e_i = \hat{y}_i - y_i$$

Compute gradient of loss w.r.t. parameters:

$$G = G + \nabla_\theta L(\hat{y}_i, y_i)$$

Compute average gradient:

$$G = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L(\hat{y}_i, y_i)$$

Update parameters:

$$\theta = \theta - \eta \cdot G$$

Stop
After completing all epochs or when convergence is achieved.

Procedure

Import the required library
Import the NumPy library to perform numerical and matrix operations.
Define the Perceptron class
Create a class named Perceptron to implement the learning model.
Initialize the model
Set the learning rate and number of epochs.

Initialize the weight with a predefined value.
Define the activation function
Use an identity activation function so that the output is linear.
Define the prediction method
Compute the weighted sum of input features and weights.
Apply the activation function to get the predicted output.
Start the training process
Repeat the training process for the given number of epochs.
Initialize gradient accumulator
At the beginning of each epoch, set the accumulated gradient to zero.
Compute predictions and errors
For each training sample:
Predict the output using the current weight.
Calculate the error as the difference between predicted output and actual output.
Accumulate gradients
Multiply the error with the input value.
Add this value to the accumulated gradient.
Calculate batch gradient
Divide the accumulated gradient by the total number of training samples.
Update the weight
Adjust the weight using the batch gradient descent rule:
w=w−η×gradientw = w - \eta \times \text{gradient}w=w−η×gradient
Display updated weight
Print the updated weight after each epoch.
Repeat
Continue steps 7 to 12 until all epochs are completed.
Test the trained model
Provide test input values to the trained model.
Predict and display the output for each test input.

Program

```
# Batch Gradient Descent Perceptron
import numpy as np

class Perceptron:

    # Constructor to initialize parameters
    def __init__(self, input_size, learning_rate=0.1, epochs=4):
        self.learning_rate = learning_rate   # Step size for weight update
        self.epochs = epochs                 # Number of training iterations
        self.weights = np.array([[4]])       # Initial weight (given)

    # Identity activation function (Linear activation)
    def activation_function(self, x):
        return x

    # Function to predict output for given input
    def predict(self, x):
        z = np.dot(self.weights, x)          # Weighted sum (w · x)
        a = self.activation_function(z)      # Apply activation function
```

```
        return a

    # Training the perceptron using Batch Gradient Descent
    def train(self, X, y):

        # Loop over epochs
        for epoch in range(self.epochs):
            ind_grad = 0   # Variable to accumulate gradients of all samples

            # Loop through all training samples
            for i in range(len(X)):
                y_pred = self.predict(X[i])   # Predicted output
                error = y_pred - y[i]         # Error (prediction − actual)
                ind_grad += (error * X[i])    # Accumulate gradient

            # Compute average gradient (Batch Gradient)
            grad = ind_grad / len(X)

            # Update weights using gradient descent rule
            self.weights = self.weights - self.learning_rate * grad

            # Display updated weight after each epoch
            print("After", epoch, "epochs, the updated weight is", self.weights)


# ----------------------------
# Example usage
# ----------------------------

# Input values (X)
X = np.array([[1], [2], [3], [4]])

# Target output values (y)
y = np.array([2, 4, 6, 8])

# Create Perceptron object
perceptron = Perceptron(input_size=1)

# Train the model
perceptron.train(X, y)

# ----------------------------
# Testing the trained model
# ----------------------------
test_data = np.array([[1], [2], [3], [4]])

# Predict output for test inputs
for i in range(len(test_data)):
    prediction = perceptron.predict(test_data[i])
    print(f"Input: {test_data[i]}, Prediction: {prediction}")
```

output:

After 0 epochs, the updated weight is [[2.5]]
After 1 epochs, the updated weight is [[2.125]]
After 2 epochs, the updated weight is [[2.03125]]
After 3 epochs, the updated weight is [[2.0078125]]
Input: [1], Prediction: [2.0078125]
Input: [2], Prediction: [4.015625]
Input: [3], Prediction: [6.0234375]
Input: [4], Prediction: [8.03125]

4b)Implementing Stochastic Batch Gradient Descent Algorithm(Calculate Gradient and New Weight for each sample)

EPOCH-1							
X	Y	W	S	A	O	Gradient	New_weight
1	2	4	4	4	4	2	3.8
2	4	3.8	7.6	7.6	7.6	7.2	3.08
3	6	3.08	9.24	9.24	9.24	9.72	2.108
4	8	2.108	8.432	8.432	8.432	1.728	1.G352
EPOCH-2							
X	Y	W	S	A	O	Gradient	New_weight
1	2	1.9352	1.9352	1.9352	1.9352	-0.0648	1.G4168
2	4	1.94168	3.88336	3.88336	3.8834	-0.23328	1.G65008
3	6	1.965008	5.895024	5.895024	5.895	-0.314928	1.GG65008
4	8	1.996501	7.986003	7.986003	7.986	-0.055987	2.0020GG52
EPOCH-3							
X	Y	W	S	A	O	Gradient	New_weight
1	2	2.0021	2.0021	2.0021	2.0021	0.0020995	2.00188G57
2	4	2.00189	4.003779	4.003779	4.0038	0.0075583	2.00113374
3	6	2.001134	6.003401	6.003401	6.0034	0.0102037	2.00011337
4	8	2.000113	8.000453	8.000453	8.0005	0.001814	1.GGGG31G8
EPOCH-4							
X	Y	W	S	A	O	Gradient	New_weight
1	2	1.999932	1.999932	1.999932	1.9999	-6.8E-05	1.GGGG3878
2	4	1.999939	3.999878	3.999878	3.9999	-0.000245	1.GGGG6327
3	6	1.999963	5.99989	5.99989	5.9999	-0.000331	1.GGGGG633
4	8	1.999996	7.999985	7.999985	8	-5.88E-05	2.0000022

Procedure:

Step 1: Initialize Parameters

- Choose learning rate η
- Choose number of epochs
- Initialize weight randomly (here it is fixed as):

$$w = 4$$

Step 2: Activation Function
This perceptron uses Identity activation:

$$f(z) = z$$

So output is simply:

$$\hat{y} = w \cdot x$$

Step 3: Prediction
For each input sample:

$$\hat{y} = w \cdot x$$

Step 4: Compute Error
Error is difference between predicted and actual output:

$$error = \hat{y} - y$$

Step 5: Compute Gradient
Gradient for one sample:

$$grad = error \cdot x$$

Step 6: Update Weight (SGD Rule)
Weight update formula:

$$w = w - \eta \cdot grad$$

Or:

$$w = w - \eta(error \cdot x)$$

Step 7: Repeat for All Samples and Epochs

- Loop over all samples (stochastic update)
- Repeat for given epochs

Step 8: Testing
After training, predict outputs for test inputs.

Program:

```
#stochastic
import numpy as np
class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=4):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.array([[4]])  # Initial weight

    def activation_function(self, x):
        return x  # Identity activation

    def predict(self, x):
        z = np.dot(self.weights, x)
        return self.activation_function(z)

    def train(self, X, y):
        for epoch in range(self.epochs):
            for i in range(len(X)):
                y_pred = self.predict(X[i])
                error = y_pred - y[i]
                grad = error * X[i]
                self.weights = self.weights - self.learning_rate * grad

                print("In epoch", epoch+1, "after", i+1,
                      "sample, the updated weight is", self.weights)

# Training data
X = np.array([[1], [2], [3], [4]])
y = np.array([2, 4, 6, 8])

# Create and train perceptron
perceptron = Perceptron(input_size=1)
perceptron.train(X, y)

# Testing
test_data = np.array([[1], [2], [3], [4]])
for i in range(len(test_data)):
    prediction = perceptron.predict(test_data[i])
    print(f'Input: {test_data[i]}, Prediction: {prediction}")
```

output:

```
In epoch 1 after 1 sample, the updated weight is [[3.8]]
In epoch 1 after 2 sample, the updated weight is [[3.08]]
In epoch 1 after 3 sample, the updated weight is [[2.108]]
In epoch 1 after 4 sample, the updated weight is [[1.9352]]
In epoch 2 after 1 sample, the updated weight is [[1.94168]]
In epoch 2 after 2 sample, the updated weight is [[1.965008]]
```

In epoch 2 after 3 sample, the updated weight is [[1.9965008]]
In epoch 2 after 4 sample, the updated weight is [[2.00209952]]
In epoch 3 after 1 sample, the updated weight is [[2.00188957]]
In epoch 3 after 2 sample, the updated weight is [[2.00113374]]
In epoch 3 after 3 sample, the updated weight is [[2.00011337]]
In epoch 3 after 4 sample, the updated weight is [[1.99993198]]
In epoch 4 after 1 sample, the updated weight is [[1.99993878]]
In epoch 4 after 2 sample, the updated weight is [[1.99996327]]
In epoch 4 after 3 sample, the updated weight is [[1.99999633]]
In epoch 4 after 4 sample, the updated weight is [[2.0000022]]
Input: [1], Prediction: [2.0000022]
Input: [2], Prediction: [4.00000441]
Input: [3], Prediction: [6.00000661]
Input: [4], Prediction: [8.00000882]

4c) Implementing Mini Batch Gradient Descent Algorithm(Calculate Gradient and New Weight for each Mini Batch)

EPOCH-1						
X	Y	W	S	A	O	GR
1	2	4	4	4	4	2
2	4	4	8	8	8	8

s= wx	a=f(s) =s	o=a	Gradient	5	nw	3.5

EPOCH-2						
X	Y	W	S	A	O	GR
1	2	1.625	1.625	1.625	1.625	-0.375
2	4	1.625	3.25	3.25	3.25	-1.5

s= wx	a=f(s) =s	o=a	Gradient	-0.938	nw	1.7188

EPOCH-3						
X	Y	W	S	A	O	GR
1	2	2.07031	2.07031	2.07031	2.07031	0.07031
2	4	2.07031	4.14063	4.14063	4.14063	0.28125

s= wx	a=f(s) =s	o=a	Gradient	0.1758	nw	2.0527

EPOCH-1						
X	Y	W	S	A	O	GR
3	6	3.5	10.5	10.5	10.5	13.5
4	8	3.5	14	14	14	24

s= wx	a=f(s) =s	o=a	Gradient	18.75	nw	1.625

EPOCH-2						
X	Y	W	S	A	O	GR
3	6	1.71875	5.15625	5.15625	5.15625	-2.5313
4	8	1.71875	6.875	6.875	6.875	-4.5

s= wx	a=f(s) =s	o=a	Gradient	-3.516	nw	2.0703

EPOCH-3						
X	Y	W	S	A	O	GR
3	6	2.05273	6.1582	6.1582	6.1582	0.47461
4	8	2.05273	8.21094	8.21094	8.21094	0.84375

s= wx	a=f(s) =s	o=a	Gradient	0.6592	nw	1.9868

Procedure:

Step 1: Initialize Parameters

- Learning rate $\eta = 0.1$
- Epochs $E = 4$
- Batch size $B = 2$
- Initial weight:

$$w = 4$$

Step 2: Activation Function
Identity activation is used:

$$f(z) = z$$

So prediction is:

$$\hat{y} = w \cdot x$$

Step 3: Divide Data into Mini-Batches
Total samples:

$$N = 4$$

Batch size:

$$B = 2$$

Number of batches:

$$bs = \lceil N/B \rceil = \lceil 4/2 \rceil = 2$$

So batches are:

- Batch 1 → $(1, 2), (2, 4)$
- Batch 2 → $(3, 6), (4, 8)$

Step 4: Forward Pass (Prediction)
For each input x_i:

$$\hat{y}_i = w \cdot x_i$$

Step 5: Compute Error
For each sample:

$$error_i = \hat{y}_i - y_i$$

Step 6: Compute Batch Gradient
Sum gradients for all samples in batch:
Average gradient:

$$grad = \frac{ind\_grad}{B}$$

Step 7: Update Weight After Each Batch
Mini-batch update rule:

$$w = w - \eta \cdot grad$$

Step 8: Repeat for All Epochs

- Loop over epochs
- Loop over all batches
- Update weights batch-wise

Step 9: Testing
After training, predict outputs:

$$\hat{y} = w \cdot x$$

Program:

```
import numpy as np
import math

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=4):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.array([[4]])
        self.batch_size = 2

    def activation_function(self, x):
        return x

    def predict(self, x):
        z = np.dot(self.weights, x)
        a = self.activation_function(z)
        return a

    def train(self, X, y):
        bs = math.ceil(len(X) / self.batch_size)

        for epoch in range(self.epochs):
            for j in range(bs):
                ind_grad = 0
                start = self.batch_size * j
                end = self.batch_size * j + self.batch_size

                for i in range(start, end):
                    if i < len(X):
```

```
                y_pred = self.predict(X[i])
                error = y_pred - y[i]
                ind_grad = ind_grad + (error * X[i])

            grad = ind_grad / self.batch_size
            self.weights = self.weights - self.learning_rate * grad

            print("Epoch", epoch + 1,
                "After batch", j + 1,
                "the new weight is", self.weights)


X = np.array([[1], [2], [3], [4]])
y = np.array([2, 4, 6, 8])

perceptron = Perceptron(input_size=1)
perceptron.train(X, y)

test_data = np.array([[1], [2], [3], [4]])
for i in range(len(test_data)):
   prediction = perceptron.predict(test_data[i])
   print(f"Input: {test_data[i]}, Prediction: {prediction}")
```

output:

```
Epoch 1 After batch 1 the new weight is [[3.5]]
Epoch 1 After batch 2 the new weight is [[1.625]]
Epoch 2 After batch 1 the new weight is [[1.71875]]
Epoch 2 After batch 2 the new weight is [[2.0703125]]
Epoch 3 After batch 1 the new weight is [[2.05273438]]
Epoch 3 After batch 2 the new weight is [[1.98681641]]
Epoch 4 After batch 1 the new weight is [[1.9901123]]
Epoch 4 After batch 2 the new weight is [[2.00247192]]
Input: [1], Prediction: [2.00247192]
Input: [2], Prediction: [4.00494385]
Input: [3], Prediction: [6.00741577]
Input: [4], Prediction: [8.0098877]
```

5)**CNN Implementation on MNIST Dataset**

Procedure:

Step 1: Import Required Libraries

Import TensorFlow/Keras modules
Import MNIST dataset
Import CNN layers like Conv2D, MaxPooling, Dense, Flatten

Step 2: Load MNIST Dataset

Load handwritten digit images using:

$$mnist.load\_data(\quad)$$

Dataset contains:
60,000 training images
10,000 testing images

Step 3: Reshape the Input Data

CNN requires 4D input:

$$(s, a, m, ples \;\; height \;\; width \;\; channels)$$

So images are reshaped to:

$$(28, \;\; , 28 \;\; 1)$$

Step 4: Normalize the Dataset

Pixel values range from 0 to 255
Normalize them into range 0 to 1:

$$X = X/255$$

This improves training speed and accuracy.

Step 5: Convert Output Labels to One-Hot Encoding

Digits (0–9) are converted into categorical form:

$$to\_categorical(y, 10)$$

Example:

$$3 \rightarrow [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$$

Step 6: Build the CNN Model

CNN architecture used:
Conv2D Layer (Feature extraction)
MaxPooling Layer (Downsampling)
Conv2D Layers (Deep feature learning)
Flatten Layer (Convert 2D → 1D)
Dense Layer (Classification learning)
Softmax Output Layer (Digit prediction)

Step 7: Compile the Model

Model is compiled using:

Loss function:

$$categorical\_crossentropy$$

Optimizer:

$$Adam$$

Metric:

$$Accuracy$$

Step 8: Train the Model

Train CNN using training data:

Epochs = 20

Batch size = 128

Step 9: Evaluate the Model

Test model performance on test dataset:

$$model.evaluate(\ \ )$$

Step 10: Display Final Loss and Accuracy

Print final results:

Loss value

Accuracy percentage

Program:

```
import numpy as np
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
from keras.utils import to_categorical, plot_model
from keras import backend as K

# Step 1: Load MNIST Dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

print("Training data shape:", x_train.shape)

# Step 2: Define Image Size
img_rows, img_cols = 28, 28

# Step 3: Reshape Data for CNN
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# Step 4: Normalize Pixel Values
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

```
# Step 5: Convert Labels to One-Hot Encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Step 6: Build CNN Model
model = Sequential()

# Convolution Layer 1
model.add(Conv2D(32, kernel_size=(3,3),
                 activation='relu',
                 input_shape=input_shape))

# Max Pooling Layer
model.add(MaxPooling2D(pool_size=(2,2)))

# Convolution Layer 2
model.add(Conv2D(32, kernel_size=(3,3),
                 activation='relu'))

# Convolution Layer 3
model.add(Conv2D(32, kernel_size=(3,3),
                 activation='relu'))

# Flatten Layer
model.add(Flatten())

# Fully Connected Dense Layer
model.add(Dense(32, activation='relu'))

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

# Step 7: Model Summary
model.summary()

# Step 8: Plot Model Architecture
plot_model(model, to_file="cnn-mnist.png", show_shapes=True)

# Step 9: Compile Model
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# Step 10: Train Model
model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)

# Step 11: Evaluate Model
score = model.evaluate(x_test, y_test, verbose=0)
```

```
print("Loss =", score[0])
print("Accuracy =", score[1])
```

output:

loss= 0.04582029581069946
accuracy= 0.9908999800682068

6)AutoCompletion Using RNN

To implement Auto-Completion using Recurrent Neural Network (RNN) for character-level prediction and text generation by training the model on sample text and predicting the next sequence of characters**.**

Procedure:

1. Import Required Libraries
   Import numpy, tensorflow, and Keras modules for model creation and data processing.
2. Prepare Sample Text
   Define a sample text string to be used for character-level training.
3. Create Character Vocabulary
   Extract unique characters from the text and create:
   - Character to index mapping
   - Index to character mapping
4. Prepare Training Sequences
   - Define sequence length
   - Create sequences of characters
   - Label each sequence with the next character
5. Convert Data to Numerical Format
   - Convert sequences and labels into numpy arrays
   - Apply one-hot encoding for input and output data
6. Build RNN Model
   - Use SimpleRNN layer with 50 neurons
   - Add Dense layer with softmax activation
   - Compile model using adam optimizer and categorical crossentropy loss
7. Train the Model
   - Fit the model using training data
   - Run for 100 epochs to learn character patterns
8. Define Auto-Completion Function
   - Take input starting text
   - Predict next characters iteratively
   - Append predicted characters to form completed text
9. Test Auto-Completion
   - Call autocomplete function with sample input
   - Display generated completed text
10. Observe Output
    - Check whether model generates meaningful continuation
    - Verify auto-completion behavior

Program:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

# Sample text
text = "Deep learning models are powerful"

# Create character vocabulary
chars = sorted(list(set(text)))
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = {i: char for i, char in enumerate(chars)}

# Sequence length
seq_length = 4
sequences = []
labels = []

# Prepare training data
for i in range(len(text) - seq_length):
    seq = text[i:i+seq_length]
    label = text[i+seq_length]

    sequences.append([char_to_index[c] for c in seq])
    labels.append(char_to_index[label])

X = np.array(sequences)
y = np.array(labels)
```

```
# One-hot encoding
X = tf.one_hot(X, len(chars))
y = tf.one_hot(y, len(chars))

# Build RNN model
model = Sequential()
model.add(SimpleRNN(50, input_shape=(seq_length, len(chars)),
activation='relu'))
model.add(Dense(len(chars), activation='softmax'))

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

# Train model
model.fit(X, y, epochs=100, verbose=1)

# ----------------------------
# Auto-completion function
# ----------------------------
def autocomplete(start_text, num_chars):
    generated = start_text

    for _ in range(num_chars):
        input_seq = generated[-seq_length:]
        input_index = np.array([[char_to_index[c] for c in input_seq]])
        input_onehot = tf.one_hot(input_index, len(chars))
```

```
        prediction = model.predict(input_onehot, verbose=0)
        next_char = index_to_char[np.argmax(prediction)]

        generated += next_char

    print("Auto-completed Text:")
    print(generated)

# Test auto-completion
autocomplete("Deep", 10)
```

output:

Auto-completed Text:

Deep learning

7)Apply RNN For Language Translation

Aim:

To implement language translation using Recurrent Neural Network (RNN) for translating English sentences into Telugu by training the model on sample bilingual data and predicting corresponding translations.

Procedure

1. Import Required Libraries
    - Import numpy, tensorflow, and keras modules for model building and data processing.
2. Prepare Dataset
    - Define sample English sentences and their Telugu translations.
    - Use small dataset for demonstration of translation.
3. Tokenization (Word Indexing)
    - Create token mappings:
        - English words → numerical indices
        - Telugu words → numerical indices
4. Convert Sentences to Numerical Sequences
    - Convert sentences into sequences using token mappings.
    - Handle words not present in the vocabulary.
5. Apply Padding
    - Use padding to make all sequences of equal length.
    - This ensures uniform input shape for the RNN model.
6. One-Hot Encoding Output
    - Convert target sequences into one-hot encoded format for classification.
7. Build RNN Model
    - Use:
        - Embedding layer (converts words into vector representation)
        - SimpleRNN layer (captures sequence relationships)
        - Dense layer with softmax activation (predicts next word)
    - Compile the model using:
        - Optimizer: Adam
        - Loss Function: Categorical Crossentropy
        - Metric: Accuracy
8. Train the Model
    - Fit the model using training data.
    - Train for sufficient epochs to learn translation patterns.
9. Define Translation Function
    - Accept input English sentence.
    - Convert to sequence and predict Telugu translation.
    - Display translated output.
10. Test Translation
    - Provide sample input (e.g., “good morning”).
    - Observe translated Telugu output.
11. Analyze Results
    - Verify correctness of translation.
    - Evaluate model performance and accuracy.

Program:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Embedding

# --------------------------
# Sample Dataset (English -> Telugu)
# --------------------------
english_sentences = [
   "hello",
   "good morning",
   "thank you",
   "how are you",
   "I am fine"
]

telugu_sentences = [
   "హలో",
   "శుభోదయం",
   "ధన్యవాదాలు",
   "మీరు ఎలా ఉన్నారు",
   "నేను బాగున్నాను"
]

# --------------------------
# Tokenization
# --------------------------
eng_tokens = {word: i for i, word in enumerate(set("
".join(english_sentences).split()))}
tel_tokens = {word: i for i, word in enumerate(set("
".join(telugu_sentences).split()))}

# Convert sentences to sequences
def encode(sentences, tokens):
   return [[tokens[word] for word in sentence.split() if word in tokens] for
sentence in sentences]

X = encode(english_sentences, eng_tokens)
```

```
y = encode(telugu_sentences, tel_tokens)

# Padding sequences
X = tf.keras.preprocessing.sequence.pad_sequences(X, padding='post')
y = tf.keras.preprocessing.sequence.pad_sequences(y, padding='post')

# One-hot encoding output
y = tf.one_hot(y, len(tel_tokens))

# --------------------------
# RNN Model for Translation
# --------------------------
model = Sequential()
model.add(Embedding(len(eng_tokens), 50))
model.add(SimpleRNN(50, return_sequences=True))
model.add(Dense(len(tel_tokens), activation='softmax'))

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

# Train model
model.fit(X, y, epochs=200, verbose=1)

# --------------------------
# Translation Function
# --------------------------
def translate(sentence):
   seq = [eng_tokens[word] for word in sentence.split() if word in
eng_tokens]
   seq = tf.keras.preprocessing.sequence.pad_sequences([seq],
padding='post')

   prediction = model.predict(seq)
   translated = [list(tel_tokens.keys())[np.argmax(p)] for p in prediction[0]]

   print("Translated Text (Telugu):", " ".join(translated))

# Test Translation
translate("good morning")
```

8) AutoCompletion Using LSTM

Aim

To implement Auto-Completion using Long Short-Term Memory (LSTM) for character-level text prediction and sequence generation by training the model on sample data and predicting the next characters.

Procedure

1. Import Required Libraries
    - Import numpy, tensorflow, and keras modules for data processing and model building.
2. Define Sample Text
    - Provide a sample string to be used as training data for character prediction.
3. Create Character Vocabulary
    - Extract unique characters from the text.
    - Create:
        - Character to index mapping
        - Index to character mapping
4. Prepare Training Sequences
    - Define sequence length.
    - Create sequences of characters.
    - Assign labels as the next character for each sequence.
5. Convert Data into Numerical Format
    - Convert sequences and labels into numpy arrays.
    - Use one-hot encoding to represent characters as binary vectors.
6. Build LSTM Model
    - Use LSTM layer with 50 units.
    - Add Dense layer with softmax activation.
    - Compile model using:
        - Optimizer: Adam
        - Loss: Categorical Crossentropy
        - Metrics: Accuracy
7. Train the Model
    - Fit the model with training data.
    - Train for 100 epochs to learn sequence patterns.
8. Define Auto-Completion Function
    - Take an input starting text.
    - Predict next characters iteratively.
    - Append predicted characters to generate completed text.
9. Test Auto-Completion
    - Call the function with sample input.

    - Display the auto-completed generated text.
    - Observe Results
- Verify whether LSTM generates meaningful continuation of text.
- Analyze prediction accuracy and behavior.

Program:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Sample text
text = "Deep learning models are powerful"

# Create character vocabulary
chars = sorted(list(set(text)))
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = {i: char for i, char in enumerate(chars)}

# Sequence length
seq_length = 4
sequences = []
labels = []

# Prepare training data
for i in range(len(text) - seq_length):
    seq = text[i:i+seq_length]
    label = text[i+seq_length]
```

```
    sequences.append([char_to_index[c] for c in seq])
    labels.append(char_to_index[label])

# Convert to numpy arrays
X = np.array(sequences)
y = np.array(labels)

# One-hot encoding
X = tf.one_hot(X, len(chars))
y = tf.one_hot(y, len(chars))

# ----------------------
# LSTM Model
# ----------------------
model = Sequential()
model.add(LSTM(50, input_shape=(seq_length, len(chars)), activation='relu'))
model.add(Dense(len(chars), activation='softmax'))

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

# Train model
model.fit(X, y, epochs=100, verbose=1)

# ----------------------
# Auto-completion function
# ----------------------
def autocomplete(start_text, num_chars):
```

```
    generated = start_text

    for _ in range(num_chars):
        input_seq = generated[-seq_length:]
        input_index = np.array([[char_to_index[c] for c in input_seq]])
        input_onehot = tf.one_hot(input_index, len(chars))

        prediction = model.predict(input_onehot, verbose=0)
        next_char = index_to_char[np.argmax(prediction)]

        generated += next_char

    print("Auto-completed Text:")
    print(generated)

# Test auto-completion
autocomplete("Deep", 10)
```

output:

Translated Text (Telugu): శుభోదయం శుభోదయం