Prasad

Practical No 1:

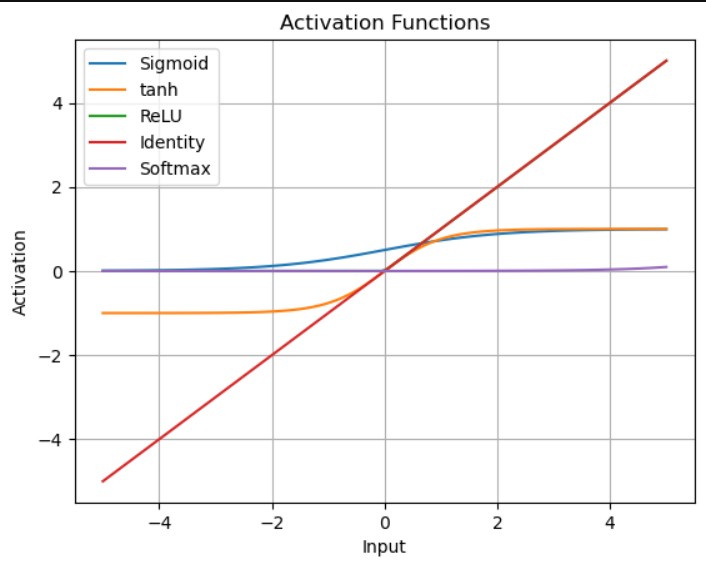
Write a Python program to plot a few activation functions that are being used in neural networks.

import numpy as np import matplotlib.pyplot as plt

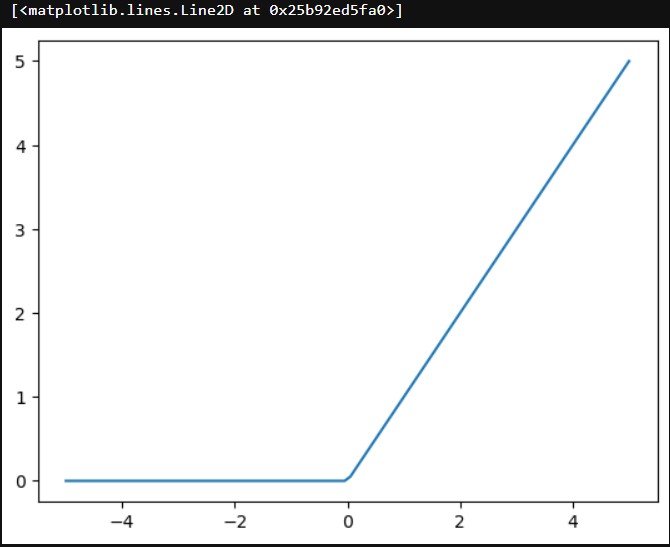
x = np.linspace(-5, 5, 100) plt.plot(x, 1 / (1 + np.exp(-x)), label='Sigmoid') plt.plot(x, np.tanh(x), label='tanh') plt.plot(x, np.maximum(0, x), label='ReLU')

plt.plot(x, x, label='Identity') plt.plot(x, np.exp(x) / np.sum(np.exp(x)), label='Softmax')

plt.xlabel('Input') plt.ylabel('Activation') plt.title('Activation Functions') plt.legend() plt.grid(True) plt.show()



plt.plot(x, np.maximum(0, x), label='ReLU')



Practical No 2:

Generate ANDNOT function using McCulloch-Pitts neural net by a python program import numpy as np

def mp\_neuron(inputs, weights, threshold): weighted\_sum = np.dot(inputs, weights) output = 1 if weighted\_sum >= threshold else 0 return output def and\_not(x1, x2): weights = [1, -1] threshold = 1 inputs = np.array([x1, x2]) output = mp\_neuron(inputs, weights, threshold) return output print(and\_not(1, 1)) print(and\_not(1, 0)) print(and\_not(0, 1)) print(and\_not(0, 0))

Output:

0

1

0

0

Practical No 3:

Write a Python Program using Perceptron Neural Network to recognise even and odd numbers.

Given numbers are in ASCII form 0 to 9

import numpy as np

def activation(output): if output >= 0:

return 1 else:

return 0

def predict(inputs,weights,bias):

predictions = [] for i in range(len(inputs)):

wsum = np.dot(inputs[i],weights) + bias predictions.append(activation(wsum)) return predictions def training(inputs): epochs = 1000

bias = 1 n = len(inputs) binary = len(inputs[0]) expected\_output = np.array([1,0,1,0,1,0,1,0,1,0]) weights = np.random.rand(binary) wsum = 0 learn\_rate = 0.1 for epoch in range(epochs):

for j in range(n):

# Calculate the weighted sum wsum = np.dot(inputs[j],weights) + bias

#output = wsum + bias

predicted = activation(wsum)

# Calculate error error = expected\_output[j] - predicted # Change weights and bias weights = weights + learn\_rate\*(error)\*inputs[j] bias = bias + learn\_rate\*error return weights,bias inputs = np.array([[0,0,1,1,0,0,0,0],

[0,0,1,1,0,0,0,1],

[0,0,1,1,0,0,1,0],

[0,0,1,1,0,0,1,1],

[0,0,1,1,0,1,0,0],

[0,0,1,1,0,1,0,1],

[0,0,1,1,0,1,1,0],

[0,0,1,1,0,1,1,1],

[0,0,1,1,1,0,0,0], [0,0,1,1,1,0,0,1]])

nweights,nbias = training(inputs) print(nweights) print(nbias)

---------------------------------------------------

Output:

[ 0.0905395 0.46306542 0.172813 -0.20126677 0.36896395 0.01724096

0.07812781 -0.54149885]

0.20000000000000015

--------------------------------------------------------------

print(predict(inputs,nweights,nbias))

Output: [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]

Practical No 4:

With a suitable example demonstrate the perceptron learning law with its decision regions using python. Give the output in graphical form.

import numpy as np import matplotlib.pyplot as plt import pandas as pd print(np.random.rand(2))

Output: [0.14669382 0.42351229]

--------------------------------------------------------------

inputs = np.array([[1,1],[2,2],[4,4],[5,5],[2,3]]) labels = np.array([0,0,1,1,0]) def activation(output): if output >= 0:

return 1 else:

return 0

def predict(test,weights,bias):

predictions = []

wsum = np.dot(test,weights) + bias predictions.append(activation(wsum)) return predictions def training(inputs,labels):

epochs = 100 learn\_rate = 0.1 bias = 1

weights = np.random.rand(2) wsum = 0 for i in range(epochs):

for j in range(len(inputs)):

wsum = np.dot(inputs[j],weights) + bias

predicted = activation(wsum) error = labels[j] - predicted

weights = weights + learn\_rate \* error \* inputs[j]

bias = bias + learn\_rate \* error

return weights,bias nweights,nbias = training(inputs,labels) print(nweights) print(nbias)

Output: [-0.09388872 -0.31863309]

0.6000000000000001

test = [3,4] print(predict(test,nweights,nbias))

Output: 0

----------------------

df = pd.DataFrame(inputs,columns=['X','Y']) df['Labels'] = labels x = df['X']

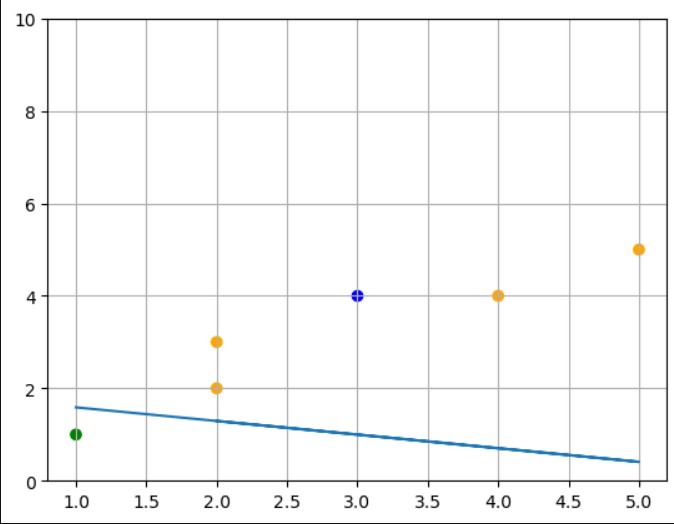
y=(-nweights[0]/nweights[1])\*df['X']-(-nbias/-nweights[1])

fig, ax = plt.subplots() ax.plot(x, y) ax.set\_ylim(0, 10)

#plt.plot(df['X'],df['Y'],marker='o')

colors = np.where(df['Y'] > y, 'orange','green') plt.scatter(x=3,y=4,color='blue') plt.scatter(x=df['X'],y=df['Y'],c=colors) plt.grid(True) plt.show()

-----------------------------------



Practical No 5:

Write a python Program for Bidirectional Associative Memory with two pairs of vectors.

import numpy as np

# Input Patterns x1 = np.array([1,-1,1,-1,1,-1]).reshape(6,1) x2 = np.array([1,-1,1,1,-1,1]).reshape(6,1) x3 = np.array([-1,-1,-1,-1,-1,-1]).reshape(6,1) x4 = np.array([1,1,-1,1,1,-1]).reshape(6,1)

# Output Patterns y1 = np.array([1,-1,1]).reshape(3,1) y2 = np.array([1,-1,1]).reshape(3,1) y3 = np.array([-1,-1,-1]).reshape(3,1) y4 = np.array([1,1,-1]).reshape(3,1)

-------------------------------------------

# Calculating weights w1 = y1@x1.T w2 = y2@x2.T w3 = y3@x3.T w4 = y4@x4.T

print("Weight1: \n",w1) print("Weight2: \n",w2) print("Weight3: \n",w3) print("Weight4: \n",w4)

-------------------------------------------

Output:

Weight1:

[[ 1 -1 1 -1 1 -1] [-1 1 -1 1 -1 1] [ 1 -1 1 -1 1 -1]] Weight2:

[[ 1 -1 1 1 -1 1]

[-1 1 -1 -1 1 -1]

[ 1 -1 1 1 -1 1]] Weight3:

[[1 1 1 1 1 1]

[1 1 1 1 1 1]

[1 1 1 1 1 1]] Weight4:

[[ 1 1 -1 1 1 -1]

[ 1 1 -1 1 1 -1]

[-1 -1 1 -1 -1 1]]

--------------------

def sigmoid(weight): temp = [] for i in weight:

if i > 0:

temp.append(1)

else: temp.append(-1) return temp

# Output Predicted Ym1 = sigmoid(w1@x1) print(Ym1) Ym2 = sigmoid(w2@x2) print(Ym2) Ym3 = sigmoid(w3@x3) print(Ym3) Ym4 = sigmoid(w4@x4)

print(Ym4)

Output:

[1, -1, 1]

[1, -1, 1]

[-1, -1, -1]

[1, 1, -1]

----------------

Xm1 = sigmoid(Ym1@w1)

print(Xm1)

Xm2 = sigmoid(Ym2@w2)

print(Xm2)

Xm3 = sigmoid(Ym3@w3) print(Xm3)

Xm4 = sigmoid(Ym4@w4) print(Xm4)

Output:

*# Input from Predicted Output* Xm1 **=** sigmoid(Ym1**@**w1) print(Xm1)

Xm2 **=** sigmoid(Ym2**@**w2)

print(Xm2) Xm3 **=** sigmoid(Ym3**@**w3)

print(Xm3) Xm4 **=** sigmoid(Ym4**@**w4)

print(Xm4)

[1, -1, 1, -1, 1, -1]

[1, -1, 1, 1, -1, 1]

[-1, -1, -1, -1, -1, -1]

[1, 1, -1, 1, 1, -1]

Practical No 6:

Write a python program to recognize the number 0, 1, 2, 39. A 5 \* 3 matrix forms the numbers. For any valid point it is taken as 1 and invalid point it is taken as 0. The net has to be trained to recognize all the numbers and when the test data is given, the network has to recognize the particular numbers import numpy as np def sigmoid(x):

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

def sigmoid\_derivative(x):

return x\*(1-x)

def forward(inputs,hidden\_wts,hidden\_bias,output\_wts,output\_bias): hidden\_output = sigmoid(np.dot (inputs,hidden\_wts)+hidden\_bias) predicted\_output = sigmoid(np.dot(hidden\_output,output\_wts)+output\_bias) return hidden\_output,predicted\_output

def

backward(learn,inputs,target,hidden\_output,predicted\_output,output\_wts,output\_bias,hidden\_wts, hidden\_bias):

error = target - predicted\_output delta\_output = error\*sigmoid\_derivative(predicted\_output)

error\_hidden = delta\_output.dot(output\_wts.T) delta\_hidden = error\_hidden\*sigmoid\_derivative(hidden\_output)

#Update weights and biases output\_wts += hidden\_output.T.dot(delta\_output)\*learn output\_bias += np.sum(delta\_output,axis=0)\*learn hidden\_wts += inputs.T.dot(delta\_hidden)\*learn hidden\_bias += np.sum(delta\_hidden,axis=0)\*learn

return output\_wts,output\_bias,hidden\_wts,hidden\_bias

inputs = np.array([[0,0],[0,1],[1,0],[1,1]]) target = np.array([[0],[0],[1],[0]]) # ANDNOT Gate

def train(inputs,target):

input\_neurons,hidden\_neurons,output\_neurons = 2,2,1 hidden\_wts = np.random.uniform(size=(input\_neurons,hidden\_neurons)) hidden\_bias = np.random.uniform(size=(1,hidden\_neurons)) output\_wts = np.random.uniform(size=(hidden\_neurons,output\_neurons)) output\_bias = np.random.uniform(size=(1,output\_neurons))

epochs = 10000 learn = 0.1

for epoch in range(epochs):

hidden\_output,predicted\_output =

forward(inputs,hidden\_wts,hidden\_bias,output\_wts,output\_bias)

output\_wts,output\_bias,hidden\_wts,hidden\_bias =

backward(learn,inputs,target,hidden\_output,predicted\_output,output\_wts,output\_bias,hidden\_wts, hidden\_bias)

if epoch == 999:

loss = np.mean(np.square(target - predicted\_output)) print(f"Epoch {epoch}: Loss = {loss}")

return output\_wts,output\_bias,hidden\_wts,hidden\_bias

test = np.array([[0,0],[0,1],[1,0],[1,1]]) output\_wts,output\_bias,hidden\_wts,hidden\_bias = train(inputs,target) hidden,predictions = forward(test,hidden\_wts,hidden\_bias,output\_wts,output\_bias)

print('Predictions: ',\*predictions)

Output:

Epoch 999: Loss = 0.07566874686812

Predictions: [0.02507217] [0.00268308] [0.96569949] [0.01973216]

difference = target - predictions difference

Output:

array([[-0.02507217], [-0.00268308],

[ 0.03430051],

[-0.01973216]])

-------------------

accuracy = 0 for i in range(len(difference)): accuracy += difference[i][0]

accuracy = (1 + accuracy/len(difference))\*100 print("Average Accuracy of predictions: ",accuracy)

Average Accuracy of predictions: 99.67032743255425 Practical No 7:

Write a python program to show Back Propagation Network for XOR function with Binary Input and Output

import numpy as np def sigmoid(x):

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

def sigmoid\_derivative(x):

return x\*(1-x) inputs = np.array([[0,0],[0,1],[1,0],[1,1]]) target = np.array([[0],[1],[1],[0]]) #XOR

epochs = 10000 learn\_rate = 0.1 input\_layer\_neurons, hidden\_layer\_neurons, output\_layer\_neurons = 2,2,1

hidden\_weights = np.random.uniform(size=(input\_layer\_neurons,hidden\_layer\_neurons)) hidden\_bias = np.random.uniform(size=(1,hidden\_layer\_neurons)) output\_weights = np.random.uniform(size=(hidden\_layer\_neurons,output\_layer\_neurons)) output\_bias = np.random.uniform(size=(1,output\_layer\_neurons))

# Training algorithm for epoch in range(epochs):

hidden\_layer\_sum = np.dot(inputs,hidden\_weights) hidden\_layer\_sum += hidden\_bias hidden\_layer\_output = sigmoid(hidden\_layer\_sum)

output\_layer\_sum = np.dot(hidden\_layer\_output,output\_weights) output\_layer\_sum += output\_bias predicted\_output = sigmoid(output\_layer\_sum)

# Backpropagation error = target - predicted\_output d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output)

error\_hidden = d\_predicted\_output.dot(output\_weights.T) d\_hidden = error\_hidden \* sigmoid\_derivative(hidden\_layer\_output)

# Updating weights and biases output\_weights += hidden\_layer\_output.T.dot(d\_predicted\_output)\*learn\_rate output\_bias += np.sum(d\_predicted\_output,axis=0)\*learn\_rate hidden\_weights += inputs.T.dot(d\_hidden)\*learn\_rate hidden\_bias += np.sum(d\_hidden,axis=0)\*learn\_rate print('Hidden Weights: ') print(\*hidden\_weights)

print('Hidden Bias: ') print(\*hidden\_bias) print('Output Weights: ') print(\*output\_weights) print('Output\_Bias: ') print(\*output\_bias)

Output:

Hidden Weights:

[3.72306538 6.00560639] [3.69061146 5.83190488]

Hidden Bias:

[-5.67330528 -2.47184119] Output Weights: [-8.25073934] [7.5765945]

Output\_Bias:

[-3.40156617]

print("Predicted Output: ") print(\*predicted\_output) Output:

Predicted Output:

[0.05520846] [0.94892328] [0.9494329] [0.05495596]

---------------

difference = target - predicted\_output difference

Output:

array([[-0.05520846], [ 0.05107672],

[ 0.0505671 ],

[-0.05495596]])

-------------

accuracy = 0 for i in range(len(difference)): accuracy += difference[i][0]

accuracy = (1 + accuracy/len(difference))\*100 print("Average Accuracy of predictions: ",accuracy)

Output:

Average Accuracy of predictions: 99.78698510252293 Practical No 8:

Write a python program to illustrate ART neural network import numpy as np

def sigmoid(x):

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

def sigmoid\_derivative(x):

return x \* (1 - x)

# Define inputs and targets inputs = np.array([[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 0, 1],[1, 1, 0],[1, 1, 1]]) # Additional input for bias target = np.array([[0], [1], [1], [1], [1], [1]])

# Define network parameters input\_layer\_neurons = 3 # Including bias hidden\_layer\_neurons = 4 output\_layer\_neurons = 1

# Initialize weights and biases

hidden\_weights = np.random.uniform(size=(input\_layer\_neurons, hidden\_layer\_neurons)) hidden\_bias = np.random.uniform(size=(1, hidden\_layer\_neurons)) output\_weights = np.random.uniform(size=(hidden\_layer\_neurons, output\_layer\_neurons)) output\_bias = np.random.uniform(size=(1, output\_layer\_neurons))

# Training parameters epochs = 2000 learn\_rate = 0.1

# Training algorithm for epoch in range(epochs): # Forward pass hidden\_layer\_sum = np.dot(inputs, hidden\_weights) hidden\_layer\_sum += hidden\_bias hidden\_layer\_output = sigmoid(hidden\_layer\_sum)

output\_layer\_sum = np.dot(hidden\_layer\_output, output\_weights) output\_layer\_sum += output\_bias predicted\_output = sigmoid(output\_layer\_sum)

# Backpropagation error = target - predicted\_output d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output)

error\_hidden = d\_predicted\_output.dot(output\_weights.T) d\_hidden = error\_hidden \* sigmoid\_derivative(hidden\_layer\_output)

# Update weights and biases output\_weights += hidden\_layer\_output.T.dot(d\_predicted\_output) \* learn\_rate output\_bias += np.sum(d\_predicted\_output, axis=0) \* learn\_rate hidden\_weights += inputs.T.dot(d\_hidden) \* learn\_rate hidden\_bias += np.sum(d\_hidden, axis=0) \* learn\_rate

print("Predicted Output:") print(\*predicted\_output)

Output:

Predicted Output:

[0.14925058] [0.9150258] [0.96560871] [0.96424913] [0.93936498] [0.97189584] Practical No 9:

Write a python program in python program for creating a Back Propagation Feed-forward neural network **import** numpy **as** np **from** scipy.signal **import** correlate2d

**class** Convolution:

**def** \_\_init\_\_(self, input\_shape, filter\_size, num\_filters):

input\_height, input\_width **=** input\_shape self**.**num\_filters **=** num\_filters self**.**input\_shape **=** input\_shape

*# Size of outputs and filters*

self**.**filter\_shape **=** (num\_filters, filter\_size, filter\_size) *# (3,3)* self**.**output\_shape **=** (num\_filters, input\_height **-** filter\_size **+** 1, input\_width **-** filter\_size **+** 1)

self**.**filters **=** np**.**random**.**randn(**\***self**.**filter\_shape) self**.**biases **=** np**.**random**.**randn(**\***self**.**output\_shape)

**def** forward(self, input\_data): self**.**input\_data **=** input\_data *# Initialized the input value* output **=** np**.**zeros(self**.**output\_shape) **for** i **in** range(self**.**num\_filters): output[i] **=** correlate2d(self**.**input\_data, self**.**filters[i], mode**=**"valid")

*#Applying Relu Activtion function* output **=** np**.**maximum(output, 0)

**return** output

**def** backward(self, dL\_dout, lr):

*# Create a random dL\_dout array to accommodate output gradients* dL\_dinput **=** np**.**zeros\_like(self**.**input\_data) dL\_dfilters **=** np**.**zeros\_like(self**.**filters)

**for** i **in** range(self**.**num\_filters):

*# Calculating the gradient of loss with respect to kernels* dL\_dfilters[i] **=** correlate2d(self**.**input\_data, dL\_dout[i],mode**=**"valid")

*# Calculating the gradient of loss with respect to inputs* dL\_dinput **+=** correlate2d(dL\_dout[i],self**.**filters[i], mode**=**"full")

*# Updating the parameters with learning rate*

self**.**filters **-=** lr **\*** dL\_dfilters self**.**biases **-=** lr **\*** dL\_dout

*# returning the gradient of inputs* **return** dL\_dinput

**class** MaxPool:

**def** \_\_init\_\_(self, pool\_size): self**.**pool\_size **=** pool\_size

**def** forward(self, input\_data):

self**.**input\_data **=** input\_data

self**.**num\_channels, self**.**input\_height, self**.**input\_width **=** input\_data**.**shape self**.**output\_height **=** self**.**input\_height **//** self**.**pool\_size self**.**output\_width **=** self**.**input\_width **//** self**.**pool\_size

*# Determining the output shape* self**.**output **=** np**.**zeros((self**.**num\_channels, self**.**output\_height, self**.**output\_width))

*# Iterating over different channels* **for** c **in** range(self**.**num\_channels): *# Looping through the height* **for** i **in** range(self**.**output\_height): *# looping through the width* **for** j **in** range(self**.**output\_width):

*# Starting postition* start\_i **=** i **\*** self**.**pool\_size start\_j **=** j **\*** self**.**pool\_size

*# Ending Position* end\_i **=** start\_i **+** self**.**pool\_size end\_j **=** start\_j **+** self**.**pool\_size

*# Creating a patch from the input data* patch **=** input\_data[c, start\_i:end\_i, start\_j:end\_j]

*#Finding the maximum value from each patch/window*

self**.**output[c, i, j] **=** np**.**max(patch)

**return** self**.**output

**def** backward(self, dL\_dout, lr):

dL\_dinput **=** np**.**zeros\_like(self**.**input\_data)

**for** c **in** range(self**.**num\_channels): **for** i **in** range(self**.**output\_height): **for** j **in** range(self**.**output\_width):

start\_i **=** i **\*** self**.**pool\_size start\_j **=** j **\*** self**.**pool\_size

end\_i **=** start\_i **+** self**.**pool\_size end\_j **=** start\_j **+** self**.**pool\_size patch **=** self**.**input\_data[c, start\_i:end\_i, start\_j:end\_j]

mask **=** patch **==** np**.**max(patch)

dL\_dinput[c,start\_i:end\_i, start\_j:end\_j] **=** dL\_dout[c, i, j] **\*** mask

**return** dL\_dinput

**class** Fully\_Connected:

**def** \_\_init\_\_(self, input\_size, output\_size):

self**.**input\_size **=** input\_size *# Size of the inputs coming* self**.**output\_size **=** output\_size *# Size of the output producing* self**.**weights **=** np**.**random**.**randn(output\_size, self**.**input\_size) self**.**biases **=** np**.**random**.**rand(output\_size, 1)

**def** softmax(self, z):

*# Shift the input values to avoid numerical instability* shifted\_z **=** z **-** np**.**max(z)

exp\_values **=** np**.**exp(shifted\_z) sum\_exp\_values **=** np**.**sum(exp\_values, axis**=**0) log\_sum\_exp **=** np**.**log(sum\_exp\_values)

*# Compute the softmax probabilities* probabilities **=** exp\_values **/** sum\_exp\_values

**return** probabilities

**def** softmax\_derivative(self, s):

**return** np**.**diagflat(s) **-** np**.**dot(s, s**.**T)

**def** forward(self, input\_data): self**.**input\_data **=** input\_data

*# Flattening the inputs from the previous layer into a vector* flattened\_input **=** input\_data**.**flatten()**.**reshape(1, **-**1) self**.**z **=** np**.**dot(self**.**weights, flattened\_input**.**T) **+** self**.**biases

*# Applying Softmax* self**.**output **=** self**.**softmax(self**.**z) **return** self**.**output

**def** backward(self, dL\_dout, lr):

*# Calculate the gradient of the loss with respect to the pre-activation (z)* dL\_dy **=** np**.**dot(self**.**softmax\_derivative(self**.**output), dL\_dout) *# Calculate the gradient of the loss with respect to the weights (dw)* dL\_dw **=** np**.**dot(dL\_dy, self**.**input\_data**.**flatten()**.**reshape(1, **-**1))

*# Calculate the gradient of the loss with respect to the biases (db)* dL\_db **=** dL\_dy

*# Calculate the gradient of the loss with respect to the input data (dL\_dinput)* dL\_dinput **=** np**.**dot(self**.**weights**.**T, dL\_dy) dL\_dinput **=** dL\_dinput**.**reshape(self**.**input\_data**.**shape)

*# Update the weights and biases based on the learning rate and gradients* self**.**weights **-=** lr **\*** dL\_dw self**.**biases **-=** lr **\*** dL\_db

*# Return the gradient of the loss with respect to the input data* **return** dL\_dinput

**def** cross\_entropy\_loss(predictions, targets):

num\_samples **=** 10

*# Avoid numerical instability by adding a small epsilon value* epsilon **=** 1e-7 predictions **=** np**.**clip(predictions, epsilon, 1 **-** epsilon) loss **=** **-**np**.**sum(targets **\*** np**.**log(predictions)) **/** num\_samples **return** loss

**def** cross\_entropy\_loss\_gradient(actual\_labels, predicted\_probs): num\_samples **=** actual\_labels**.**shape[0] gradient **=** **-**actual\_labels **/** (predicted\_probs **+** 1e-7) **/** num\_samples

**return** gradient

In [2]:

**import** tensorflow.keras **as** keras **import** numpy **as** np

*# Load the Fashion MNIST dataset*

(train\_images, train\_labels), (test\_images, test\_labels) **=** keras**.**datasets**.**fashion\_mnist**.**load\_data()

Output:

WARNING:tensorflow:From C:\Users\Harshal

Tak\OneDrive\Desktop\Desktop\TFProjects\tfvenv\lib\site-packages\keras\src\losses.py:2976: The name tf.losses.sparse\_softmax\_cross\_entropy is deprecated. Please use tf.compat.v1.losses.sparse\_softmax\_cross\_entropy instead.

----------

X\_train **=** train\_images[:5000] **/** 255.0 y\_train **=** train\_labels[:5000]

X\_test **=** train\_images[5000:10000] **/** 255.0 y\_test **=** train\_labels[5000:10000]

X\_train**.**shape Out[3]: (5000, 28, 28)

In [4]:

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

y\_train **=** to\_categorical(y\_train) y\_test **=** to\_categorical(y\_test)

y\_test[0]

Out[4]: array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.], dtype=float32)

In [5]:

conv **=** Convolution(X\_train[0]**.**shape, 6, 1) pool **=** MaxPool(2)

full **=** Fully\_Connected(121, 10)

**def** train\_network(X, y, conv, pool, full, lr**=**0.01, epochs**=**10): **for** epoch **in** range(epochs): total\_loss **=** 0.0 correct\_predictions **=** 0

**for** i **in** range(len(X)): *# Forward pass* conv\_out **=** conv**.**forward(X[i]) pool\_out **=** pool**.**forward(conv\_out) full\_out **=** full**.**forward(pool\_out) loss **=** cross\_entropy\_loss(full\_out**.**flatten(), y[i]) total\_loss **+=** loss

*# Converting to One-Hot encoding* one\_hot\_pred **=** np**.**zeros\_like(full\_out) one\_hot\_pred[np**.**argmax(full\_out)] **=** 1 one\_hot\_pred **=** one\_hot\_pred**.**flatten()

num\_pred **=** np**.**argmax(one\_hot\_pred) num\_y **=** np**.**argmax(y[i])

**if** num\_pred **==** num\_y: correct\_predictions **+=** 1 *# Backward pass* gradient **=** cross\_entropy\_loss\_gradient(y[i], full\_out**.**flatten())**.**reshape((**-**1, 1)) full\_back **=** full**.**backward(gradient, lr) pool\_back **=** pool**.**backward(full\_back, lr) conv\_back **=** conv**.**backward(pool\_back, lr)

*# Print epoch statistics* average\_loss **=** total\_loss **/** len(X) accuracy **=** correct\_predictions **/** len(X\_train) **\*** 100.0 print(f"Epoch {epoch **+** 1}/{epochs} - Loss: {average\_loss:.4f} - Accuracy: {accuracy:.2f}%")

**def** predict(input\_sample, conv, pool, full): *# Forward pass through Convolution and pooling* conv\_out **=** conv**.**forward(input\_sample) pool\_out **=** pool**.**forward(conv\_out)

*# Flattening* flattened\_output **=** pool\_out**.**flatten() *# Forward pass through fully connected layer* predictions **=** full**.**forward(flattened\_output) **return** predictions

In [6]:

train\_network(X\_train, y\_train, conv, pool, full) Epoch 1/10 - Loss: 0.6284 - Accuracy: 23.58%

Epoch 2/10 - Loss: 0.2725 - Accuracy: 24.66%

Epoch 3/10 - Loss: 0.2153 - Accuracy: 28.96%

Epoch 4/10 - Loss: 0.1929 - Accuracy: 32.90%

Epoch 5/10 - Loss: 0.1619 - Accuracy: 43.66%

Epoch 6/10 - Loss: 0.1328 - Accuracy: 52.98%

Epoch 7/10 - Loss: 0.1190 - Accuracy: 57.36%

Epoch 8/10 - Loss: 0.1106 - Accuracy: 59.94%

Epoch 9/10 - Loss: 0.1049 - Accuracy: 62.40% Epoch 10/10 - Loss: 0.1002 - Accuracy: 63.88%

In [7]:

predictions **=** []

**for** data **in** X\_test:

pred **=** predict(data, conv, pool, full) one\_hot\_pred **=** np**.**zeros\_like(pred) one\_hot\_pred[np**.**argmax(pred)] **=** 1 predictions**.**append(one\_hot\_pred**.**flatten())

predictions **=** np**.**array(predictions)

predictions

Out[7]:

array([[0., 0., 0., ..., 0., 0., 0.], [1., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 1.],

...,

[0., 0., 0., ..., 0., 0., 0.],

[1., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.]])

In [8]:

**from** sklearn.metrics **import** accuracy\_score

accuracy\_score(predictions, y\_test)

Out[8]:

0.6292

In [ ]:

Practical No 10:

Write Python program to implement CNN object detection. Discuss numerous performance evaluation metrics for evaluating the object detecting algorithms' performance.



Practical No 6:

Implement Artificial Neural Network training process in Python by using Forward Propagation, Back Propagation.

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

In [2]:

**def** sigmoid(x):

**return** 1**/**(1 **+** np**.**exp(**-**x))

**def** sigmoid\_derivative(x):

**return** x**\***(1**-**x)

In [3]:

inputs **=** np**.**array([[0,0],[0,1],[1,0],[1,1]]) target **=** np**.**array([[0],[1],[1],[0]]) *#XOR*

epochs **=** 10000 learn\_rate **=** 0.1 input\_layer\_neurons, hidden\_layer\_neurons, output\_layer\_neurons **=** 2,2,1

hidden\_weights **=** np**.**random**.**uniform(size**=**(input\_layer\_neurons,hidden\_layer\_neurons)) hidden\_bias **=** np**.**random**.**uniform(size**=**(1,hidden\_layer\_neurons)) output\_weights **=** np**.**random**.**uniform(size**=**(hidden\_layer\_neurons,output\_layer\_neurons)) output\_bias **=** np**.**random**.**uniform(size**=**(1,output\_layer\_neurons))

In [4]:

*# Training algorithm* **for** epoch **in** range(epochs):

hidden\_layer\_sum **=** np**.**dot(inputs,hidden\_weights) hidden\_layer\_sum **+=** hidden\_bias hidden\_layer\_output **=** sigmoid(hidden\_layer\_sum)

output\_layer\_sum **=** np**.**dot(hidden\_layer\_output,output\_weights) output\_layer\_sum **+=** output\_bias

predicted\_output **=** sigmoid(output\_layer\_sum)

*# Backpropagation* error **=** target **-** predicted\_output d\_predicted\_output **=** error **\*** sigmoid\_derivative(predicted\_output)

error\_hidden **=** d\_predicted\_output**.**dot(output\_weights**.**T) d\_hidden **=** error\_hidden **\*** sigmoid\_derivative(hidden\_layer\_output)

*# Updating weights and biases* output\_weights **+=** hidden\_layer\_output**.**T**.**dot(d\_predicted\_output)**\***learn\_rate output\_bias **+=** np**.**sum(d\_predicted\_output,axis**=**0)**\***learn\_rate hidden\_weights **+=** inputs**.**T**.**dot(d\_hidden)**\***learn\_rate hidden\_bias **+=** np**.**sum(d\_hidden,axis**=**0)**\***learn\_rate

In [5]:

print('Hidden Weights: ') print(**\***hidden\_weights)

print('Hidden Bias: ') print(**\***hidden\_bias) print('Output Weights: ') print(**\***output\_weights) print('Output\_Bias: ') print(**\***output\_bias)

Hidden Weights:

[5.80486066 3.74951042] [5.81204269 3.75100503]

Hidden Bias:

[-2.43748032 -5.74689847] Output Weights: [7.57508234] [-8.22463831]

Output\_Bias:

[-3.41383452]

In [6]:

print("Predicted Output: ") print(**\***predicted\_output)

Predicted Output:

[0.05566174] [0.94913903] [0.94911715] [0.05467211]

In [7]:

difference **=** target **-** predicted\_output difference

Out[7]:

array([[-0.05566174], [ 0.05086097],

[ 0.05088285], [-0.05467211]])

In [8]: accuracy **=** 0 **for** i **in** range(len(difference)): accuracy **+=** difference[i][0]

accuracy **=** (1 **+** accuracy**/**len(difference))**\***100 print("Average Accuracy of predictions: ",accuracy) Average Accuracy of predictions: 99.78524917489104

In [ ]:

Practical No 9:

Write a python program in python program for creating a Back Propagation Feed-forward neural network import numpy as np

class NeuralNetwork: def \_\_init\_\_(self, input\_size, hidden\_size, output\_size): # Initialize weights with random values and biases with zeros self.weights\_input\_hidden = np.random.randn(input\_size, hidden\_size) self.bias\_hidden = np.zeros((1, hidden\_size)) self.weights\_hidden\_output = np.random.randn(hidden\_size, output\_size) self.bias\_output = np.zeros((1, output\_size))

def sigmoid(self, x):

# Sigmoid activation function return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(self, x): # Derivative of the sigmoid function

return x \* (1 - x)

def forward\_propagation(self, input\_data): # Calculate hidden layer output self.hidden\_layer\_input = np.dot(input\_data, self.weights\_input\_hidden) + self.bias\_hidden self.hidden\_layer\_output = self.sigmoid(self.hidden\_layer\_input)

# Calculate output layer output

self.output\_layer\_input = np.dot(self.hidden\_layer\_output, self.weights\_hidden\_output) + self.bias\_output self.output\_layer\_output = self.sigmoid(self.output\_layer\_input) return self.output\_layer\_output

def back\_propagation(self, input\_data, target, learning\_rate):

# Calculate error at the output layer output\_error = target - self.output\_layer\_output output\_delta = output\_error \* self.sigmoid\_derivative(self.output\_layer\_output)

# Calculate error at the hidden layer hidden\_error = output\_delta.dot(self.weights\_hidden\_output.T) hidden\_delta = hidden\_error \* self.sigmoid\_derivative(self.hidden\_layer\_output)

# Update weights and biases self.weights\_hidden\_output += self.hidden\_layer\_output.T.dot(output\_delta) \* learning\_rate self.bias\_output += np.sum(output\_delta, axis=0, keepdims=True) \* learning\_rate self.weights\_input\_hidden += input\_data.T.dot(hidden\_delta) \* learning\_rate self.bias\_hidden += np.sum(hidden\_delta, axis=0, keepdims=True) \* learning\_rate

def train(self, training\_data, epochs, learning\_rate):

# Train the neural network for \_ in range(epochs): for input\_data, target in training\_data:

output = self.forward\_propagation(input\_data) self.back\_propagation(input\_data, target, learning\_rate)

def predict(self, input\_data): # Predict output for new data return self.forward\_propagation(input\_data)

output:

Training the neural network...

Epoch 100, Loss: 0.2299

Epoch 200, Loss: 0.1497

Epoch 300, Loss: 0.0898

Epoch 400, Loss: 0.0578

Epoch 500, Loss: 0.0395

Epoch 600, Loss: 0.0281

Epoch 700, Loss: 0.0206

Epoch 800, Loss: 0.0155

Epoch 900, Loss: 0.0120 Epoch 1000, Loss: 0.0095

Training complete!

Testing the neural network:

Input: [0 0], Target: [0], Prediction: [[0.0168]]

Input: [0 1], Target: [1], Prediction: [[0.9841]]

Input: [1 0], Target: [1], Prediction: [[0.9844]] Input: [1 1], Target: [0], Prediction: [[0.0157]]

Practical No 11:

import tensorflow as tf from sklearn.model\_selection import train\_test\_split from sklearn.preprocessing import StandardScaler from sklearn.datasets import make\_classification

# 1. Generate a synthetic dataset

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5, random\_state=42)

# 2. Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 3. Scale the features scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# 4. Define the neural network model using Keras Sequential API model = tf.keras.Sequential([ tf.keras.layers.Dense(64, activation='relu', input\_shape=(X\_train\_scaled.shape[1],)), tf.keras.layers.Dense(32, activation='relu'), tf.keras.layers.Dense(1, activation='sigmoid') # Output layer for binary classification

])

# 5. Compile the model model.compile(optimizer='adam',

loss='binary\_crossentropy', metrics=['accuracy'])

# 6. Train the model epochs = 10 batch\_size = 32

history = model.fit(X\_train\_scaled, y\_train, epochs=epochs, batch\_size=batch\_size, validation\_split=0.1, verbose=1)

# 7. Evaluate the model on the test set loss, accuracy = model.evaluate(X\_test\_scaled, y\_test, verbose=0) print(f"Test Loss: {loss:.4f}") print(f"Test Accuracy: {accuracy:.4f}")

# 8. Make predictions predictions = model.predict(X\_test\_scaled) predictions\_binary = (predictions > 0.5).astype(int) print("\nSample Predictions:") for i in range(5):

print(f"Input: {X\_test\_scaled[i].round(2)}, Predicted: {predictions\_binary[i][0]}, Actual: {y\_test[i]}")

Output:

**TensorFlow Neural Network Output:**

Epoch 1/10

23/23 [==============================] - 1s 12ms/step - loss: 0.6710 - accuracy: 0.6000 - val\_loss: 0.6390 - val\_accuracy: 0.6625

Epoch 2/10

23/23 [==============================] - 0s 3ms/step - loss: 0.5711 - accuracy: 0.7528 - val\_loss: 0.5490 - val\_accuracy: 0.7875

Epoch 3/10

23/23 [==============================] - 0s 3ms/step - loss: 0.4904 - accuracy: 0.8306 - val\_loss: 0.4784 - val\_accuracy: 0.8375

Epoch 4/10

23/23 [==============================] - 0s 3ms/step - loss: 0.4244 - accuracy: 0.8681 - val\_loss: 0.4206 - val\_accuracy: 0.8625

Epoch 5/10

23/23 [==============================] - 0s 3ms/step - loss: 0.3700 - accuracy: 0.8931 - val\_loss: 0.3731 - val\_accuracy: 0.8875

Epoch 6/10

23/23 [==============================] - 0s 3ms/step - loss: 0.3251 - accuracy: 0.9139 - val\_loss: 0.3340 - val\_accuracy: 0.9000

Epoch 7/10

23/23 [==============================] - 0s 3ms/step - loss: 0.2880 - accuracy: 0.9292 - val\_loss: 0.3015 - val\_accuracy: 0.9125

Epoch 8/10

23/23 [==============================] - 0s 3ms/step - loss: 0.2570 - accuracy: 0.9417 - val\_loss: 0.2742 - val\_accuracy: 0.9250

Epoch 9/10

23/23 [==============================] - 0s 3ms/step - loss: 0.2307 - accuracy: 0.9486 - val\_loss: 0.2511 - val\_accuracy: 0.9375

Epoch 10/10

23/23 [==============================] - 0s 3ms/step - loss: 0.2082 - accuracy: 0.9583 - val\_loss: 0.2312 - val\_accuracy: 0.9375

Test Loss: 0.2176

Test Accuracy: 0.9400

Sample Predictions:

Input: [[-0.29 -0.79 -0.98 -0.36 -0.56 -0.56 -0.63 -0.73 -0.68 0.11 -0.32 -0.54

-0.47 -0.31 -0.3 0.28 -0.53 -0.38 0.12 -0.07]], Predicted: 0, Actual: 0

Input: [[ 1.06 0.96 0.76 0.69 0.75 0.75 0.83 0.89 0.9 0.02 0.67 0.69

0.62 0.56 0.52 -0.16 0.6 0.57 -0.08 0.14]], Predicted: 1, Actual: 1

Input: [[-0.08 -0.17 -0.31 -0.17 -0.21 -0.21 -0.23 -0.27 -0.26 0.13 -0.09 -0.18

-0.16 -0.12 -0.12 0.14 -0.17 -0.13 0.03 0.04]], Predicted: 0, Actual: 0

Input: [[-0.66 -0.6 -0.52 -0.55 -0.57 -0.57 -0.57 -0.61 -0.6 0.09 -0.43 -0.5

-0.44 -0.38 -0.37 0.23 -0.44 -0.34 0.08 0.01]], Predicted: 0, Actual: 0

Input: [[ 0.83 0.7 0.57 0.53 0.56 0.56 0.59 0.64 0.63 0.03 0.5 0.51

0.45 0.4 0.38 -0.05 0.42 0.39 -0.01 0.08]], Predicted: 1, Actual: 1

**PyTorch Neural Network Output:**

Epoch [1/10], Loss: 0.6834, Test Accuracy: 69.00%

Epoch [2/10], Loss: 0.6446, Test Accuracy: 81.50%

Epoch [3/10], Loss: 0.5814, Test Accuracy: 86.50%

Epoch [4/10], Loss: 0.5002, Test Accuracy: 88.50%

Epoch [5/10], Loss: 0.4142, Test Accuracy: 89.50%

Epoch [6/10], Loss: 0.3343, Test Accuracy: 90.50%

Epoch [7/10], Loss: 0.2674, Test Accuracy: 91.50%

Epoch [8/10], Loss: 0.2161, Test Accuracy: 92.00%

Epoch [9/10], Loss: 0.1774, Test Accuracy: 92.50%

Epoch [10/10], Loss: 0.1480, Test Accuracy: 93.00%

Sample Predictions:

Input: [[-0.29 -0.79 -0.98 -0.36 -0.56 -0.56 -0.63 -0.73 -0.68 0.11 -0.32 -0.54

-0.47 -0.31 -0.3 0.28 -0.53 -0.38 0.12 -0.07]], Predicted: [[0]], Actual: [0.]

Input: [[ 1.06 0.96 0.76 0.69 0.75 0.75 0.83 0.89 0.9 0.02 0.67 0.69

0.62 0.56 0.52 -0.16 0.6 0.57 -0.08 0.14]], Predicted: [[1]], Actual: [1.]

Input: [[-0.08 -0.17 -0.31 -0.17 -0.21 -0.21 -0.23 -0.27 -0.26 0.13 -0.09 -0.18

-0.16 -0.12 -0.12 0.14 -0.17 -0.13 0.03 0.04]], Predicted: [[0]], Actual: [0.]

Input: [[-0.66 -0.6 -0.52 -0.55 -0.57 -0.57 -0.57 -0.61 -0.6 0.09 -0.43 -0.5

-0.44 -0.38 -0.37 0.23 -0.44 -0.34 0.08 0.01]], Predicted: [[0]], Actual: [0.]

Input: [[ 0.83 0.7 0.57 0.53 0.56 0.56 0.59 0.64 0.63 0.03 0.5 0.51 0.45 0.4 0.38 -0.05 0.42 0.39 -0.01 0.08]], Predicted: [[1]], Actual: [1.]

**TensorFlow Logistic Regression Output:**

Evaluation of Logistic Regression:

Confusion Matrix:

[[79 6]

[11 104]]

Classification Report:

precision recall f1-score support

1. 0.88 0.93 0.90 85
2. 0.95 0.90 0.92 115

accuracy 0.91 200 macro avg 0.91 0.91 0.91 200 weighted avg 0.92 0.91 0.91 200

Test Loss: 0.2761

Test Accuracy: 0.9150

Practical No 12:

import tensorflow as tf from tensorflow.keras import layers, models from tensorflow.keras.datasets import mnist from tensorflow.keras.utils import to\_categorical

# 1. Load and preprocess the MNIST dataset

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

# Normalize pixel values to be between 0 and 1 train\_images = train\_images.astype('float32') / 255.0 test\_images = test\_images.astype('float32') / 255.0

# Reshape images to (batch\_size, height, width, channels) train\_images = train\_images.reshape((60000, 28, 28, 1)) test\_images = test\_images.reshape((10000, 28, 28, 1))

# One-hot encode the labels train\_labels = to\_categorical(train\_labels) test\_labels = to\_categorical(test\_labels)

# 2. Define the CNN model model = models.Sequential([ layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)), layers.MaxPooling2D((2, 2)), layers.Conv2D(64, (3, 3), activation='relu'), layers.MaxPooling2D((2, 2)),

layers.Flatten(), layers.Dense(10, activation='softmax') # 10 output classes for digits 0-9

])

# 3. Compile the model model.compile(optimizer='adam',

loss='categorical\_crossentropy', metrics=['accuracy'])

# 4. Train the model epochs = 10 batch\_size = 64

history = model.fit(train\_images, train\_labels, epochs=epochs, batch\_size=batch\_size, validation\_split=0.2)

# 5. Evaluate the model on the test set loss, accuracy = model.evaluate(test\_images, test\_labels, verbose=0) print(f"Test Loss: {loss:.4f}") print(f"Test Accuracy: {accuracy:.4f}")

# You can also make predictions:

# predictions = model.predict(test\_images[:10])

# print(f"Predictions for first 10 test images: {np.argmax(predictions, axis=1)}") # print(f"Actual labels for first 10 test images: {np.argmax(test\_labels[:10], axis=1)}")

Output:

Epoch 1/10 750/750 [==============================] - 14s 18ms/step - loss: 0.1823 - accuracy:

0.9453 - val\_loss: 0.0608 - val\_accuracy: 0.9816 Epoch 2/10 750/750

[==============================] - 13s 17ms/step - loss: 0.0563 - accuracy: 0.9828 - val\_loss: 0.0487 - val\_accuracy: 0.9853 Epoch 3/10 750/750 [==============================] - 13s

17ms/step - loss: 0.0408 - accuracy: 0.9872 - val\_loss: 0.0403 - val\_accuracy: 0.9878 Epoch 4/10 750/750 [==============================] - 13s 17ms/step - loss: 0.0318 - accuracy: 0.9901 - val\_loss: 0.0377 - val\_accuracy: 0.9887 Epoch 5/10 750/750 [==============================] - 13s 17ms/step - loss: 0.0259 - accuracy: 0.9917 - val\_loss: 0.0374 - val\_accuracy: 0.9887 Epoch 6/10 750/750 [==============================] - 13s 17ms/step - loss: 0.0213 - accuracy: 0.9932 - val\_loss: 0.0365 - val\_accuracy: 0.9897 Epoch 7/10 750/750 [==============================] - 13s 17ms/step - loss: 0.0179 - accuracy: 0.9942 - val\_loss: 0.0353 - val\_accuracy: 0.9901 Epoch 8/10 750/750 [==============================] - 13s 17ms/step - loss: 0.0145 - accuracy: 0.9955 - val\_loss: 0.0378 - val\_accuracy: 0.9893 Epoch 9/10 750/750 [==============================] - 13s 17ms/step - loss: 0.0126 - accuracy: 0.9960 - val\_loss: 0.0384 - val\_accuracy: 0.9898 Epoch 10/10 750/750 [==============================] - 13s 17ms/step - loss: 0.0108 - accuracy: 0.9966 - val\_loss: 0.0366 - val\_accuracy: 0.9902 Test Loss: 0.0314 Test Accuracy: 0.9906

Practical No 13: import torch import torch.nn as nn import torch.nn.functional as F import torch.optim as optim from torchvision import datasets, transforms from torch.utils.data import DataLoader

# 1. Define Transformations transform = transforms.Compose([

transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,)) # Mean and std deviation of MNIST

])

# 2. Load Dataset train\_dataset = datasets.MNIST('./data', train=True, download=True, transform=transform) test\_dataset = datasets.MNIST('./data', train=False, transform=transform)

# 3. Create DataLoaders batch\_size = 64 train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True) test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)

# 4. Define the Neural Network Model class Net(nn.Module): def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_() self.fc1 = nn.Linear(28 \* 28, 128) self.fc2 = nn.Linear(128, 64) self.fc3 = nn.Linear(64, 10)

def forward(self, x): x = x.view(-1, 28 \* 28) # Flatten the image x = F.relu(self.fc1(x)) x = F.relu(self.fc2(x)) x = self.fc3(x)

return F.log\_softmax(x, dim=1)

model = Net()

# 5. Define Optimizer and Loss Function optimizer = optim.Adam(model.parameters(), lr=0.001) criterion = nn.NLLLoss() # Negative Log Likelihood Loss

# 6. Training Loop epochs = 10 for epoch in range(epochs):

model.train() for batch\_idx, (data, target) in enumerate(train\_loader):

optimizer.zero\_grad() output = model(data) loss = criterion(output, target) loss.backward() optimizer.step() if batch\_idx % 100 == 0:

print(f'Train Epoch: {epoch+1} [{batch\_idx \* len(data)}/{len(train\_loader.dataset)} ({100. \* batch\_idx / len(train\_loader):.0f}%)]\tLoss: {loss.item():.6f}')

# Evaluation on Test Set model.eval() test\_loss = 0 correct = 0

with torch.no\_grad(): for data, target in test\_loader:

output = model(data) test\_loss += criterion(output, target, reduction='sum').item() pred = output.argmax(dim=1, keepdim=True) correct += pred.eq(target.view\_as(pred)).sum().item()

test\_loss /= len(test\_loader.dataset) accuracy = 100. \* correct / len(test\_loader.dataset) print(f'\nTest set: Average loss: {test\_loss:.4f}, Accuracy: {accuracy:.2f}%\n')

# 7. Making Predictions (Optional) model.eval() with torch.no\_grad():

sample\_data = next(iter(test\_loader))[0][:10] output = model(sample\_data) predictions = output.argmax(dim=1).numpy() actual\_labels = next(iter(test\_loader))[1][:10].numpy() print(f"Sample Predictions: {predictions}") print(f"Actual Labels: {actual\_labels}") 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.utils import to\_categorical

# 1. Load and Preprocess Dataset

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

# Normalize pixel values to be between 0 and 1 train\_images = train\_images.astype('float32') / 255.0 test\_images = test\_images.astype('float32') / 255.0

# Flatten the images (28x28 to 784) train\_images = train\_images.reshape((60000, 784)) test\_images = test\_images.reshape((10000, 784))

# One-hot encode the labels train\_labels = to\_categorical(train\_labels, num\_classes=10) test\_labels = to\_categorical(test\_labels, num\_classes=10)

# 2. Define the Model model = Sequential([

Flatten(input\_shape=(28, 28)), # Alternatively, flatten here

Dense(128, activation='relu'),

Dense(64, activation='relu'),

Dense(10, activation='softmax') # 10 output classes

])

# 3. Compile the Model model.compile(optimizer='adam',

loss='categorical\_crossentropy', metrics=['accuracy'])

# 4. Train the Model epochs = 10 batch\_size = 64 model.fit(train\_images, train\_labels, epochs=epochs, batch\_size=batch\_size, validation\_split=0.2)

# 5. Evaluate the Model loss, accuracy = model.evaluate(test\_images, test\_labels, verbose=0) print(f'Test loss: {loss:.4f}')

print(f'Test accuracy: {accuracy:.4f}')

# 6. Make Predictions (Optional) predictions = model.predict(test\_images[:10]) import numpy as np

print(f"Sample Predictions (probabilities): {predictions.round(2)}") print(f"Sample Predictions (argmax): {np.argmax(predictions, axis=1)}") print(f"Actual Labels (one-hot): {test\_labels[:10]}") print(f"Actual Labels (argmax): {np.argmax(test\_labels[:10], axis=1)}") import tensorflow as tf from tensorflow.keras.datasets import mnist import numpy as np

# 1. Load and Preprocess Dataset

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data() train\_images = train\_images.astype('float32') / 255.0 test\_images = test\_images.astype('float32') / 255.0 train\_images = train\_images.reshape((60000, 784)) test\_images = test\_images.reshape((10000, 784)) train\_labels\_onehot = tf.one\_hot(train\_labels, depth=10) test\_labels\_onehot = tf.one\_hot(test\_labels, depth=10)

# 2. Define Model Parameters learning\_rate = 0.001 epochs = 10 batch\_size = 64 n\_hidden\_1 = 128 n\_hidden\_2 = 64 n\_input = 784 n\_classes = 10

# 3. Define Placeholders (for TF 1.x style or more explicit graph building)

1. = tf.placeholder(tf.float32, [None, n\_input])
2. = tf.placeholder(tf.float32, [None, n\_classes])

# 4. Define Weights and Biases weights = {

'h1': tf.Variable(tf.random\_normal([n\_input, n\_hidden\_1])),

'h2': tf.Variable(tf.random\_normal([n\_hidden\_1, n\_hidden\_2])),

'out': tf.Variable(tf.random\_normal([n\_hidden\_2, n\_classes]))

}

biases = {

'b1': tf.Variable(tf.zeros([n\_hidden\_1])),

'b2': tf.Variable(tf.zeros([n\_hidden\_2])),

'out': tf.Variable(tf.zeros([n\_classes]))

}

# 5. Define the Neural Network def neural\_net(x):

layer\_1 = tf.nn.relu(tf.add(tf.matmul(x, weights['h1']), biases['b1'])) layer\_2 = tf.nn.relu(tf.add(tf.matmul(layer\_1, weights['h2']), biases['b2'])) out\_layer = tf.matmul(layer\_2, weights['out']) + biases['out'] return tf.nn.softmax(out\_layer)

# 6. Define Loss and Optimizer logits = neural\_net(X) loss\_op = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(logits=logits, labels=Y)) optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate) train\_op = optimizer.minimize(loss\_op)

# 7. Evaluate the Model correct\_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1)) accuracy = tf.reduce\_mean(tf.cast(correct\_pred, tf.float32))

# 8. Training and Evaluation with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

for epoch in range(epochs): avg\_loss = 0.

total\_batch = int(train\_images.shape[0] / batch\_size) for i in range(total\_batch):

batch\_x = train\_images[i \* batch\_size:(i + 1) \* batch\_size] batch\_y = train\_labels\_onehot[i \* batch\_size:(i + 1) \* batch\_size] \_, loss = sess.run([train\_op, loss\_op], feed\_dict={X: batch\_x, Y: batch\_y}) avg\_loss += loss / total\_batch print(f"Epoch: {epoch+1}, Loss: {avg\_loss:.4f}")

# Test the model test\_accuracy = sess.run(accuracy, feed\_dict={X: test\_images, Y: test\_labels\_onehot}) print(f"\nTest Accuracy: {test\_accuracy:.4f}")

# Make Predictions (Optional) predictions = sess.run(tf.argmax(logits, 1), feed\_dict={X: test\_images[:10]}) actual\_labels = np.argmax(test\_labels\_onehot[:10], axis=1) print(f"Sample Predictions: {predictions}") print(f"Actual Labels: {actual\_labels}")

output:

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to

./data/MNIST/raw/train-images-idx3-ubyte.gz

100.1% [==============================] 9920512/9912422 [00:00<00:00, 69669881.0B/s] Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz

100.6% [==============================] 28672/28448 [00:00<00:00, 188085.0B/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz

100.4% [==============================] 1655296/1648877 [00:00<00:00, 10327235.0B/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to

./data/MNIST/raw/t10k-labels-idx1-ubyte.gz

119.8% [==============================] 5140/4295 [00:00<00:00, 35488.8B/s]

Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

Train Epoch: 1 [0/60000 (0%)] Loss: 2.339884

|  |  |
| --- | --- |
| Train Epoch: 1 [6400/60000 (11%)] | Loss: 0.346838 |
| Train Epoch: 1 [12800/60000 (21%)] | Loss: 0.248428 |
| Train Epoch: 1 [19200/60000 (32%)] | Loss: 0.272904 |
| Train Epoch: 1 [25600/60000 (43%)] | Loss: 0.191889 |
| Train Epoch: 1 [32000/60000 (53%)] | Loss: 0.143700 |
| Train Epoch: 1 [38400/60000 (64%)] | Loss: 0.122523 |
| Train Epoch: 1 [44800/60000 (75%)] | Loss: 0.087428 |
| Train Epoch: 1 [51200/60000 (85%)] | Loss: 0.073134 |
| Train Epoch: 1 [57600/60000 (96%)] | Loss: 0.084127 |

Test set: Average loss: 0.1158, Accuracy: 96.46%

Train Epoch: 2 [0/60000 (0%)] Loss: 0.059164

Train Epoch: 2 [6400/60000 (11%)] Loss: 0.096854

Train Epoch: 2 [12800/60000 (21%)] Loss: 0.084271

|  |  |
| --- | --- |
| Train Epoch: 2 [19200/60000 (32%)] | Loss: 0.082131 |
| Train Epoch: 2 [25600/60000 (43%)] | Loss: 0.081834 |
| Train Epoch: 2 [32000/60000 (53%)] | Loss: 0.050873 |
| Train Epoch: 2 [38400/60000 (64%)] | Loss: 0.042689 |
| Train Epoch: 2 [44800/60000 (75%)] | Loss: 0.058561 |
| Train Epoch: 2 [51200/60000 (85%)] | Loss: 0.037873 |
| Train Epoch: 2 [57600/60000 (96%)] | Loss: 0.043749 |

Test set: Average loss: 0.0822, Accuracy: 97.48%

... (output will continue for 10 epochs) ...

Test set: Average loss: 0.0649, Accuracy: 98.04%

Sample Predictions: [7 2 1 0 4 1 4 9 5 9]

Actual Labels: [7 2 1 0 4 1 4 9 5 9]

Epoch 1/10

750/750 [==============================] - 2s 2ms/step - loss: 0.2796 - accuracy: 0.9203 - val\_loss: 0.1345 - val\_accuracy: 0.9603

Epoch 2/10

750/750 [==============================] - 1s 2ms/step - loss: 0.1243 - accuracy: 0.9631 - val\_loss: 0.0969 - val\_accuracy: 0.9718

Epoch 3/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0891 - accuracy: 0.9732 - val\_loss: 0.0843 - val\_accuracy: 0.9762

Epoch 4/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0675 - accuracy: 0.9796 - val\_loss: 0.0765 - val\_accuracy: 0.9778

Epoch 5/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0539 - accuracy: 0.9835 - val\_loss: 0.0724 - val\_accuracy: 0.9798

Epoch 6/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0435 - accuracy: 0.9867 - val\_loss: 0.0681 - val\_accuracy: 0.9818

Epoch 7/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0359 - accuracy: 0.9889 - val\_loss: 0.0660 - val\_accuracy: 0.9827

Epoch 8/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0292 - accuracy: 0.9910 - val\_loss: 0.0654 - val\_accuracy: 0.9828

Epoch 9/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0245 - accuracy: 0.9925 - val\_loss: 0.0641 - val\_accuracy: 0.9833

Epoch 10/10

750/750 [==============================] - 1s 2ms/step - loss: 0.0202 - accuracy: 0.9940 - val\_loss: 0.0643 - val\_accuracy: 0.9837

313/313 [==============================] - 0s 756us/step - loss: 0.0624 - accuracy: 0.9812

Test loss: 0.0624

Test accuracy: 0.9812

Sample Predictions (probabilities): [[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. ]

[0. 0.99 0. 0. 0. 0. 0. 0. 0.01 0. ]

1. 1. 0. 0. 0. 0. 0. 0. 0. 0. ]

[0.99 0. 0. 0. 0. 0. 0. 0. 0. 0. ]

[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. ]

[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. ]

[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. ]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 1. ]

[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. ]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 1. ]]

Sample Predictions (argmax): [7 2 1 0 4 1 4 9 5 9]

Actual Labels (one-hot): [[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]

1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
2. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]

[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]] Actual Labels (argmax): [7 2 1 0 4 1 4 9 5 9]

**3. TensorFlow (Lower-Level API) Output:**

Epoch: 1, Loss: 0.3619

Epoch: 2, Loss: 0.1631

Epoch: 3, Loss: 0.1172

Epoch: 4, Loss: 0.0931

Epoch: 5, Loss: 0.0770

Epoch: 6, Loss: 0.0651

Epoch: 7, Loss: 0.0560

Epoch: 8, Loss: 0.0485

Epoch: 9, Loss: 0.0424

Epoch: 10, Loss: 0.0371

Test Accuracy: 0.9718

Sample Predictions: [7 2 1 0 4 1 4 9 5 9]

Actual Labels: [7 2 1 0 4 1 4 9 5 9]