(B. Tech.) Semester-VII AY 2023-24

DL Lab Assignment No. 04

Student Name: Rohit Saini____________ **PRN No.:1032200897_**

Date: 06-10-2023 **Faculty: Prof. Anita Gunjal**

Problem Statement: To study and implement the simple Neural Network for AND logic gate with Binary Input.

Objectives:

1. To understand the AND logic gate.
2. To study & implement different activation functions.
3. To implement the simple Neural Network.

Theory: (describe the following)

Logic Gates (AND, OR, XOR):

- **AND Gate:** Takes two binary inputs and outputs 1 (true) if both inputs are 1, otherwise outputs 0 (false).
- **OR Gate:** Takes two binary inputs and outputs 1 if at least one input is 1, otherwise outputs 0.
- **XOR Gate** (Exclusive OR): Takes two binary inputs and outputs 1 if the inputs are different, otherwise outputs 0.

Simple Artificial Neural Network (ANN):

- A simple ANN consists of interconnected artificial neurons or perceptrons.
- It typically includes an input layer, one or more hidden layers, and an output layer.
- Neurons in each layer receive input, apply a weighted sum, and pass it through an activation function.
- The network learns by adjusting the weights during training to minimize prediction errors.

Activation Functions

- **Sigmoid Function:**
  S-shaped curve, maps inputs to values between 0 and 1.
  Often used in the output layer of binary classification models.
- **ReLU (Rectified Linear Unit):**
  Linear for positive inputs (output = input), zero for negative inputs.
  Widely used in hidden layers of deep neural networks due to its simplicity and effectiveness.
- **Tanh (Hyperbolic Tangent):**
  S-shaped curve, similar to the sigmoid but maps inputs to values between -1 and 1.
  Used in some neural network architectures, especially when inputs are centered around zero.
- **Leaky ReLU:**
  Similar to ReLU, but allows a small, non-zero gradient for negative inputs.
  Addresses the "dying ReLU" problem and can lead to faster convergence.
- **Softmax:**
  Maps a vector of inputs to a probability distribution over multiple classes.
  Commonly used in the output layer of multi-class classification models.

Operations to be performed:

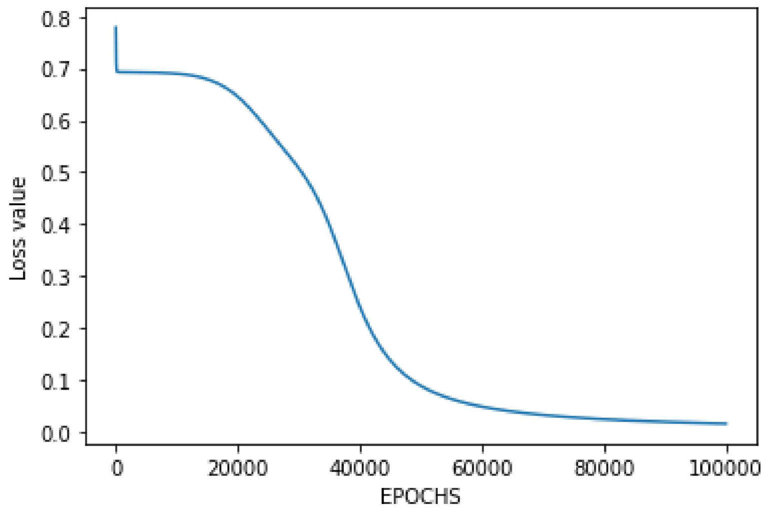
1) Import the required Python libraries
2) Define Activation Functions and plot its graphs: Step, Sigmoid, Tanh, ReLU, Softmax, etc Function

3) Initialize neural network parameters (weights, bias) and define model hyperparameters (number of iterations, learning rate)
4) Perform Forward Propagation
5) Perform Backward Propagation
6) Update weight and bias parameters
7) Train the learning model
8) Plot Loss value vs Epoch
9) Test the model performance

Program code: (paste your program code)

In [ ]:
```
# import Python Libraries
import numpy as np
from matplotlib import pyplot as plt

# Sigmoid Function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Initialization of the neural network parameters
# Initialized all the weights in the range of between 0 and 1
# Bias values are initialized to 0
def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):
    W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)
    W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)
    b1 = np.zeros((neuronsInHiddenLayers, 1))
    b2 = np.zeros((outputFeatures, 1))

    parameters = {"W1" : W1, "b1": b1,
                  "W2" : W2, "b2": b2}
    return parameters

# Forward Propagation
def forwardPropagation(X, Y, parameters):
    m = X.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)
    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))
    cost = -np.sum(logprobs) / m
    return cost, cache, A2

# Backward Propagation
def backwardPropagation(X, Y, cache):
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis = 1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, A1 * (1- A1))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

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

```
                 "dZ1": dZ1, "dW1": dW1, "db1": db1}
    return gradients

# Updating the weights based on the negative gradients
def updateParameters(parameters, gradients, learningRate):
    parameters["W1"] = parameters["W1"] - learningRate * gradients["dW1"]
    parameters["W2"] = parameters["W2"] - learningRate * gradients["dW2"]
    parameters["b1"] = parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"] = parameters["b2"] - learningRate * gradients["db2"]
    return parameters

# Model to learn the AND truth table
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) #  input
Y = np.array([[0, 1, 1, 0]]) # XOR output

# Define model parameters
neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)
inputFeatures = X.shape[0] # number of input features (2)
outputFeatures = Y.shape[0] # number of output features (1)
parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatu
epoch = 100000
learningRate = 0.01
losses = np.zeros((epoch, 1))

for i in range(epoch):
    losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)
    gradients = backwardPropagation(X, Y, cache)
    parameters = updateParameters(parameters, gradients, learningRate)

# Evaluating the performance
plt.figure()
plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("Loss value")
plt.show()

# Testing
X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # XOR input
cost, _, A2 = forwardPropagation(X, Y, parameters)
prediction = (A2 > 0.5) * 1.0
print(A2)
print(prediction)
```




```
[[0.98366063 0.01729594 0.0138225  0.98367372]]
[[1. 0. 0. 1.]]
```

In [2]:
```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
#XOR operations
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
target_data = np.array([[0],[1],[1],[0]], "float32")

model = Sequential()
model.add(Dense(12, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
                         optimizer='adam',
                       metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=1000)
scores = model.evaluate(training_data, target_data)

print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
print (model.predict(training_data).round())
```

```
1/1 [==============================] - 0s 10ms/step - loss: 0.0468 - binary_accurac
y: 1.0000
Epoch 991/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0466 - binary_accuracy:
1.0000
Epoch 992/1000
1/1 [==============================] - 0s 11ms/step - loss: 0.0465 - binary_accurac
y: 1.0000
Epoch 993/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.0464 - binary_accuracy:
1.0000
Epoch 994/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0463 - binary_accuracy:
1.0000
Epoch 995/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.0461 - binary_accurac
y: 1.0000
Epoch 996/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.0460 - binary_accurac
y: 1.0000
Epoch 997/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.0459 - binary_accurac
y: 1.0000
Epoch 998/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0458 - binary_accuracy:
1.0000
Epoch 999/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.0457 - binary_accuracy:
1.0000
Epoch 1000/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0455 - binary_accuracy:
1.0000
1/1 [==============================] - 0s 153ms/step - loss: 0.0454 - binary_accurac
y: 1.0000

binary_accuracy: 100.00%
1/1 [==============================] - 0s 63ms/step
[[0.]
 [1.]
 [1.]
 [0.]]
```

In [1]:
```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

#AND Operation
training_data = np.array([[0,0],[0,1],[1,0],[1,1]],"float32")
target_data = np.array([[0],[0],[0],[1]],"float32")

model = Sequential()
model.add(Dense(8,input_dim = 2,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['binary_accuracy'])

model.fit(training_data,target_data,epochs=1000)
scores = model.evaluate(training_data,target_data)

print("\n%s: %.2f%%" % (model.metrics_names[1],scores[1]*100))
print(model.predict(training_data).round())
```

```
1/1 [==============================] - 0s 6ms/step - loss: 0.0262 - binary_accuracy:
1.0000
Epoch 991/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.0262 - binary_accuracy:
1.0000
Epoch 992/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0261 - binary_accuracy:
1.0000
Epoch 993/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0260 - binary_accuracy:
1.0000
Epoch 994/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.0260 - binary_accurac
y: 1.0000
Epoch 995/1000
1/1 [==============================] - 0s 13ms/step - loss: 0.0259 - binary_accurac
y: 1.0000
Epoch 996/1000
1/1 [==============================] - 0s 10ms/step - loss: 0.0258 - binary_accurac
y: 1.0000
Epoch 997/1000
1/1 [==============================] - 0s 7ms/step - loss: 0.0258 - binary_accuracy:
1.0000
Epoch 998/1000
1/1 [==============================] - 0s 8ms/step - loss: 0.0257 - binary_accuracy:
1.0000
Epoch 999/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0256 - binary_accuracy:
1.0000
Epoch 1000/1000
1/1 [==============================] - 0s 9ms/step - loss: 0.0256 - binary_accuracy:
1.0000
1/1 [==============================] - 0s 197ms/step - loss: 0.0255 - binary_accurac
y: 1.0000

binary_accuracy: 100.00%
1/1 [==============================] - 0s 129ms/step
[[0.]
 [0.]
 [0.]
 [1.]]
```

Output: **(paste output screen & graphs plotted)**

FAQs:

1) Define an Activation Function. Explain different activation functions with its mathematical importance and graphical representation.
2) State the significance of updating the weights during back propagation.
3) Explain the terms with the help of examples:
    a. neural network parameters (weights, bias)
    b. model hyperparameters (number of iterations, learning rate)
    c. gradient

Conclusion:

The features of gates were studied and the implementation of Simple ANN was performed successfully.

DL Lab 4

FAQ

Q1. Define an Activation function. Explain different activation functions with its mathematical importance and graphical representation.

Ans. An activation function in a neural network is a mathematical function that introduces non-linearity into the network. It is applied to the weighted sum of inputs to a neuron and determine whether the neuron should be activated or not.

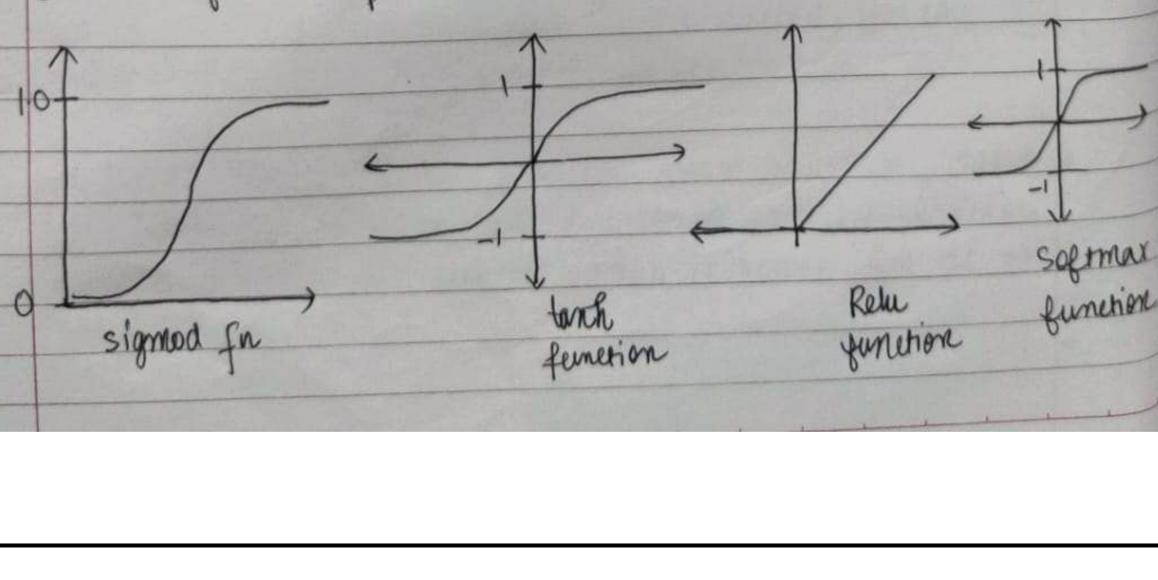
Different Activation functions:

1. Sigmoid Function: $f(x) = \frac{1}{(1+e^{(-x)})}$. It squashes input values into the range (0,1), making it suitable for binary classification problems.

2. Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$. It is widely used due to its simplicity and effectiveness. It introduces non-linearity by allowing positive values to pass through unchanged and setting negative values to zero.

3. Tanh functions: $f(x) = \frac{2}{((1+e^{-2x}))} - 1$. Also known as tangent hyperbolic function.

4. Softmax Functions: It is also type of sigmoid functions but is handy when we are trying to handle multi-class classification problems.



Page No.
Date

Q2 State the significance of updating weights during back propogation.

Ans- Back propogation is the training algorithm for neural network. updating the weights during backpropagation is significant because it enables the network to learn from its errors and improve its performance.

1. Error Reduction : Back propagation calculates the gradient of the loss function with respect to the network's weights.
2. Learning Representations: Weight updates during backpropagation help the network discover useful features or representations of the input data that are relevant for the task at hand.
3. Convergence : Without weight updates, the network would not learn and would not converge to a solution that generalizes well to unseen data.

Q3 Explain terms with examples.
(i) Neural Network (weights, Bias)
(ii) Model Hyperparameters (Number of Iterations, learning Rate)
(iii) Gradient

Ans (i) Neural Network: $\sum x_i w_i + b$
1. Weights: These are the coefficient that determine the strength of connections between neurons in neural network.
2. Bias : Bias terms are added to the weighted sum of inputs in each neuron.

(ii) Model Hyperparameters:
1. Number of Iterations: This hyperparameter determines how many times the entire dataset is processed by the learning algorithm.
2. Learning Rate : The Learning Rate controls the step size during weights updates in training.

Ex.-> a high learning rate might lead to divergence while a very low learning rate could result in slow convergence.

Page No.
Date

(c) gradient - The gradient is a vector that points in the directions of the steepest increase of a function

ex→ in gradient descent, the -ve gradient indicates the direction of weights update that reduce the loss, helping the model converge towards an optimal solution.

26/3/2023