Batch: B2 Roll No.: 16010121194

Experiment No. 04

Grade: A / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Study and implementation of Delta learning rule

Objective: To write a program to implement Delta learning rule for a given training data.

Expected Outcome of Experiment:
CO3 : To Understand perceptron's and counter propagation networks

Books/ Journals/ Websites referred:

Pre Lab/ Prior Concepts:

Delta learning rule and its application

The Delta Learning Rule, also known as the Widrow-Hoff learning rule or the Least Mean Squares (LMS) algorithm, is a supervised learning algorithm used for training artificial neural networks, particularly single-layer perceptrons. It's a gradient-based learning rule that aims to minimize the error between the network's predicted output and the desired target output. Here's how the Delta Learning Rule works and its applications:

Delta Learning Rule:

1. Initialization:

- Initialize the weights and bias of the neural network to small random values.
- Set the learning rate (η), a small positive constant.

2. Iterative Training:

- For each input-output pair (x, y), perform the following steps:

3. Forward Pass:

- Compute the weighted sum of inputs: z = (w_1 * x_1) + (w_2 * x_2) + ... + (w_n * x_n) + bias.

4. Activation:

- Apply an activation function (usually a step function or sigmoid) to the weighted sum to get the network's output: y_pred = f(z).

5. Error Calculation:

- Calculate the error (the difference between the predicted output and the target output): error = y - y_pred.

6. **Weight** and Bias Update:

- Update the weights and bias using the following formulas:
  - Δw_i = η * error * x_i
  - Δbias = η * error
  - w_i = w_i + Δw_i
  - bias = bias + Δbias

7. Repeat:

- Repeat the above steps for a specified number of iterations or until the error converges to a desired level.

Applications of the Delta Learning Rule:

1. Perceptrons: The Delta Learning Rule is often used to train single-layer perceptrons, which are simple neural networks capable of linear classification. It's suitable for problems where data can be linearly separated, such as binary classification tasks.

2. Adaptive Filtering: The Delta Learning Rule is used in adaptive filtering applications, where the goal is to filter out noise or interference from a signal. It's employed in fields like speech and audio processing, image processing, and telecommunications.

3. Function Approximation: It can be used to approximate functions when the relationship between input and output is not necessarily linear but can be locally approximated as such. This has applications in function interpolation and approximation.

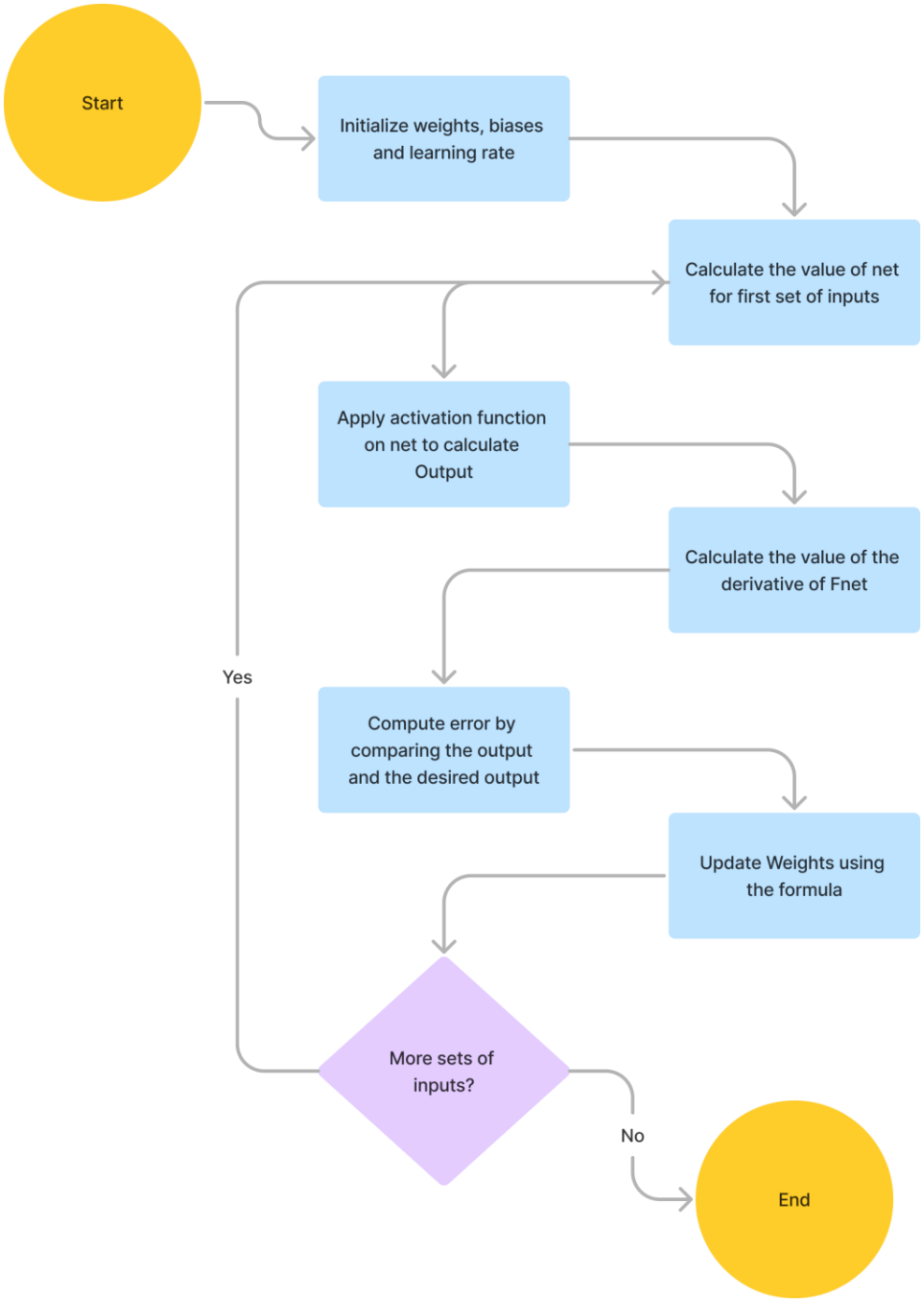
4. Control Systems: In control systems, the Delta Learning Rule can be used for training neural networks to control various processes. For example, it can be used in applications like robotics, where the network learns to control the movement of a robot based on sensor input.

5. Pattern Recognition: The Delta Learning Rule can be applied to pattern recognition tasks, such as handwriting recognition, character recognition, and image classification. In these applications, the network learns to recognize patterns in input data.

6. Time Series Prediction: It can be used for time series prediction tasks, where the goal is to predict future values in a sequence based on historical data.

While the Delta Learning Rule is a straightforward and effective algorithm for certain tasks, it has limitations, such as its inability to handle nonlinearly separable data and its sensitivity to the learning rate parameter. For more complex tasks, multilayer neural networks with backpropagation are often preferred.

Flowchart for Delta Learning Rule:



Implementation Details:

1. **Implement Delta learning rule for the example given in Zurada book and verify the accuracy of the result.**

   Code:

```
import math

#libraries used only for data visualization:
import numpy as np
from tabulate import tabulate

#inputs given in the question
inputs = [[1, -2, 0, -1], [0, 1.5, -0.5, -1], [-1, 1, 0.5, -1]]

#choosing learning rate to be 0.1
learning_rate = 0.1

#desired outputs and initial weights given in the question
desired_output = [-1, -1, 1]
weights = [1, -1, 0, 0.5]

#function defined for calculating activation function
#using bipolar sigmoidal function
def activation_function(net):
    return (1 - math.exp(-net)) / (1 + math.exp(-net))

#creating loop for iteration for each input array
#thus, there will be 3 epochs, because there are 3 input arrays

for j in range(len(inputs)):

    #initializing the value of net to be 0 before every
    #input array's calculations are done
    net = 0

    print("\n\n------------------Epoch ", (j + 1), "------------------")

    #taking dot product of weights array and inputs array to find the value of
fnet
```

```
    for i in range(len(weights)):
        net = net + weights[i] * inputs[j][i]

    #actual output will be the activation function applied on the value of net
    output = activation_function(net)

    #finding the derivative of net
    fnet_derivative = 0.5 * (1 - (output * output))

    #the error is the difference between the desired output and the actual output
    error = (desired_output[j] - output)

    #making a variable for storing the value of error for data visualization
    error_tab = round(error, 3)

    #making a variable for storing the value of old weights for data
visualization
    #using the round function that comes with numpy to show only the first 5
digits after the decimal
    old = list(np.round(weights, 5))

    #declaring a list to store the value of change in weights
    delta_w = []

    #for loop for calculating the value of new weights
    #purpose of for loop is matrix multiplication
    for k in range(len(weights)):

        #storing the value of change in weight given by learning_rate * error *
fnet_derivative * inputs[j][k]
        #for data visualization
        delta_w.append(round(learning_rate * error * fnet_derivative *
inputs[j][k], 5))
        weights[k] += learning_rate * error * fnet_derivative * inputs[j][k]

    #making a variable for storing the value of new weights after this epoch for
data visualization
    new_weights = list(np.round(weights, 3))

    #making a list of numpy arrays to pass in the tabulate function
    #as passing numpy arrays directly to tabulate gives warnings and causes
conflicts
```

```
    table_data = [old, new_weights, delta_w, net, round(fnet_derivative, 3), error_tab]

    #using tabulate functions for data visualization
    headers = ["Old Weights", "New Weights", "Change in Weight", "Net", "FNet", "Error"]
    table = tabulate([table_data], headers=headers, tablefmt="fancy_grid")
    print(table)
```

Output:

```
------------------Epoch  1 ------------------
╒═══════════════════╤═══════════════════════╤═══════════════════════════════════╤═══════╤═══════╤═══════╕
│ Old Weights       │ New Weights           │ Change in Weight                  │   Net │  FNet │ Error │
╞═══════════════════╪═══════════════════════╪═══════════════════════════════════╪═══════╪═══════╪═══════╡
│ [1.0, -1.0, 0.0, 0.5] │ [0.974, -0.948, 0.0, 0.526] │ [-0.02591, 0.05183, -0.0, 0.02591] │   2.5 │  0.14 │ -1.848 │
╘═══════════════════╧═══════════════════════╧═══════════════════════════════════╧═══════╧═══════╧═══════╛

------------------Epoch  2 ------------------
╒═══════════════════════╤═══════════════════════╤═══════════════════════════════════╤═══════════╤═══════╤═══════╕
│ Old Weights           │ New Weights           │ Change in Weight                  │       Net │  FNet │ Error │
╞═══════════════════════╪═══════════════════════╪═══════════════════════════════════╪═══════════╪═══════╪═══════╡
│ [0.97409, -0.94817, 0.0, 0.52591] │ [0.974, -0.956, 0.003, 0.531] │ [-0.0, -0.00817, 0.00272, 0.00545] │ -1.94817 │ 0.218 │ -0.25 │
╘═══════════════════════╧═══════════════════════╧═══════════════════════════════════╧═══════════╧═══════╧═══════╛

------------------Epoch  3 ------------------
╒═══════════════════════╤═══════════════════════╤═══════════════════════════════════╤═══════════╤═══════╤═══════╕
│ Old Weights           │ New Weights           │ Change in Weight                  │       Net │  FNet │ Error │
╞═══════════════════════╪═══════════════════════╪═══════════════════════════════════╪═══════════╪═══════╪═══════╡
│ [0.97409, -0.95634, 0.00272, 0.53136] │ [0.947, -0.93, 0.016, 0.505] │ [-0.02671, 0.02671, 0.01336, -0.02671] │ -2.46043 │ 0.145 │ 1.843 │
╘═══════════════════════╧═══════════════════════╧═══════════════════════════════════╧═══════════╧═══════╧═══════╛
PS C:\Users\uditi>
```

Solution in Zurada:

Step 1 Input is vector $\mathbf{x}_1$, initial weight vector is $\mathbf{w}^1$:

$$net^1 = \mathbf{w}^{1t}\mathbf{x}_1 = 2.5$$
$$o^1 = f(net^1) = 0.848$$
$$f'(net^1) = \frac{1}{2}[1 - (o^1)^2] = 0.140$$
$$\mathbf{w}^2 = c(d_1 - o^1)f'(net^1)\mathbf{x}_1 + \mathbf{w}^1$$
$$= [0.974 \quad -0.948 \quad 0 \quad 0.526]^t$$

Step 2 Input is vector $\mathbf{x}_2$, weight vector is $\mathbf{w}^2$:

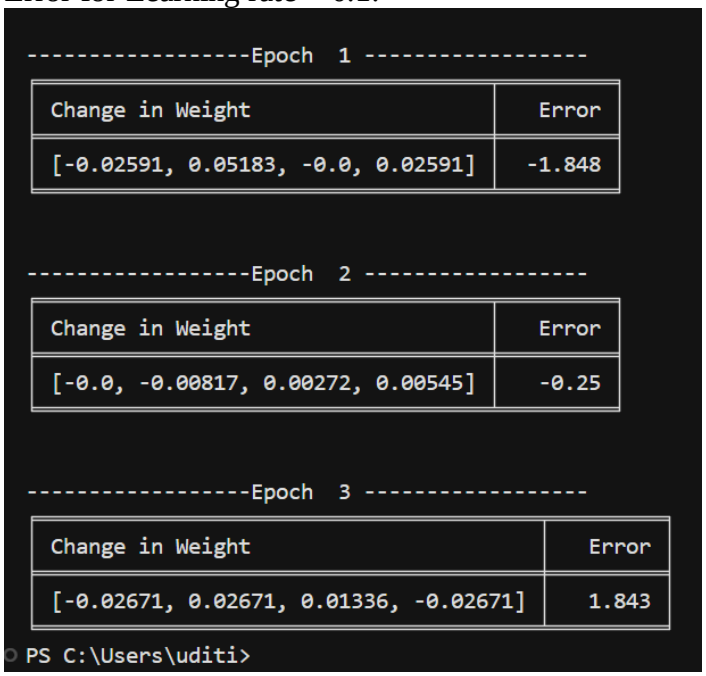
$$net^2 = \mathbf{w}^{2t}\mathbf{x}_2 = -1.948$$
$$o^2 = f(net^2) = -0.75$$
$$f'(net^2) = \frac{1}{2}[1 - (o^2)^2] = 0.218$$
$$\mathbf{w}^3 = c(d_2 - o^2)f'(net^2)\mathbf{x}_2 + \mathbf{w}^2$$
$$= [0.974 \quad -0.956 \quad 0.002 \quad 0.531]^t$$

Step 3 Input is x_3, weight vector is $\mathbf{w}^3$:

$$net^3 = \mathbf{w}^{3t}\mathbf{x}_3 = -2.46$$
$$o^3 = f(net^3) = -0.842$$
$$f'(net^3) = \frac{1}{2}[1 - (o^3)^2] = 0.145$$
$$\mathbf{w}^4 = c(d_3 - o^3)f'(net^3)\mathbf{x}_3 + \mathbf{w}^3$$
$$= [0.947 \quad -0.929 \quad 0.016 \quad 0.505]^t$$

2. Compute the error for different learning rate

Error for Learning rate = 0.1:

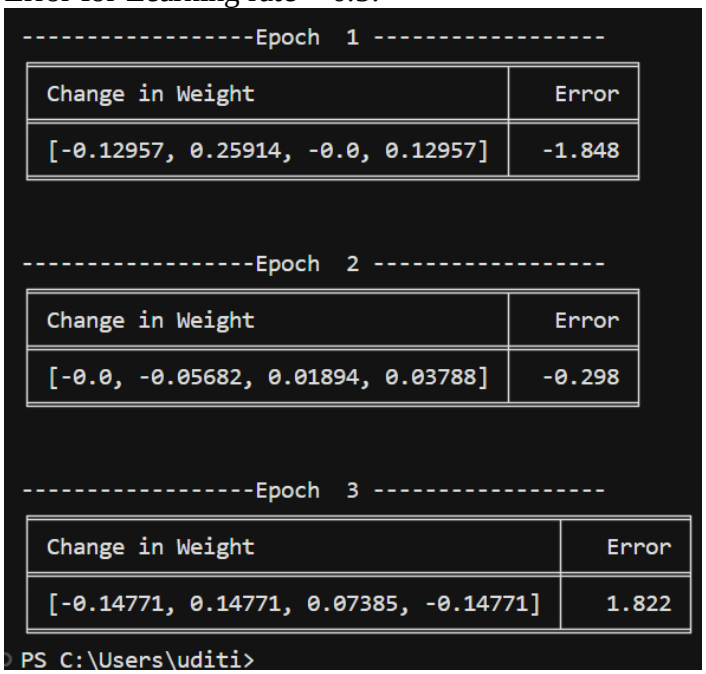


```
------------------Epoch  1 ------------------
| Change in Weight                  | Error |
| [-0.02591, 0.05183, -0.0, 0.02591] | -1.848 |

------------------Epoch  2 ------------------
| Change in Weight                  | Error |
| [-0.0, -0.00817, 0.00272, 0.00545] | -0.25 |

------------------Epoch  3 ------------------
| Change in Weight                  | Error |
| [-0.02671, 0.02671, 0.01336, -0.02671] | 1.843 |
PS C:\Users\uditi>
```

Error for Learning rate = 0.5:

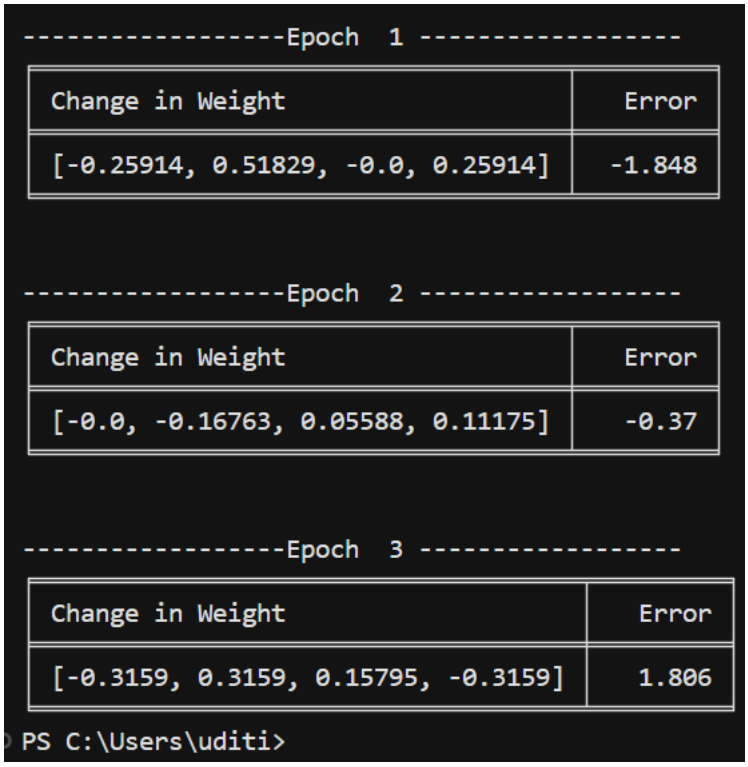


```
------------------Epoch  1 ------------------
| Change in Weight                  | Error |
| [-0.12957, 0.25914, -0.0, 0.12957] | -1.848 |

------------------Epoch  2 ------------------
| Change in Weight                  | Error |
| [-0.0, -0.05682, 0.01894, 0.03788] | -0.298 |

------------------Epoch  3 ------------------
| Change in Weight                  | Error |
| [-0.14771, 0.14771, 0.07385, -0.14771] | 1.822 |
PS C:\Users\uditi>
```

Error for Learning rate = 1:



Conclusion: In this Lab I implemented Delta learning rule using the Perceptron model and studied the changes in weight, error, net, etc. I also observed the errors received for different learning rules.

Post Lab Descriptive Questions :

1. **Comparison of Neural network and algorithmic computation**

Ans)

Dimension	Neural Network	Algorithmic Computation
Problem Types	Complex patterns, NLP, images	Sorting, searching, optimisation
Learning	Data-driven, gradient descent	Rule-based, predefined logic

Generalization	Good at generalizing	Specific, may lack generalization
Data Requirement	Large data, training necessary	Smaller datasets may suffice
Feature Engineering	Automatically learns features	Often requires manual engineering
Parallelism	Parallel processing with GPUs/TPUs	Limited parallelism in some cases
Interpretability	Complex and less interpretable	Transparent and more interpretable
Resource needs	Computationally intensive	Efficient for certain problems

2. Explain different types of supervised techniques.

Ans) Supervised learning is a category of machine learning where the algorithm learns from labeled training data to make predictions or decisions. In supervised learning, each input data point is associated with a corresponding target or label, and the goal is to learn a mapping from inputs to outputs. Here are some common types:

Classification:

- In classification, the goal is to predict a categorical label or class for a given input data point.
- Example: Email spam detection, image classification (identifying objects in images), medical diagnosis (categorizing diseases based on symptoms).

Regression:

- Regression involves predicting a continuous numeric value for an input data point.
- Example: Predicting house prices based on features like square footage, number of bedrooms, etc., stock price prediction, temperature forecasting.

Support Vector Machines (SVM):

- SVM is a classification technique that aims to find a hyperplane that best separates different classes of data.
- It works by maximizing the margin between classes while minimizing classification errors.
- SVM can handle linear and non-linear classification problems.

Decision Trees:

- Decision trees use a tree-like structure to make decisions by repeatedly splitting the data into subsets based on the values of input features.
- They are intuitive and can handle both classification and regression tasks.
- Decision trees are prone to overfitting, which can be mitigated using techniques like pruning.

Random Forest:

- Random Forest is an ensemble technique that combines multiple decision trees to improve prediction accuracy and reduce overfitting.
- Each tree in the forest is trained on a different subset of data and features.
- Random Forest can handle both classification and regression tasks.

Naive Bayes:

- Naive Bayes is a probabilistic classification technique based on Bayes' theorem and the assumption of independence among features.
- It's particularly useful for text classification tasks like spam detection and sentiment analysis.

3. Take any example and show the different steps of Delta Algorithm.

Ans)

The Delta Algorithm is a data differencing algorithm used to efficiently encode the differences between two versions of data. It's commonly used in version control systems and data compression techniques. Let's walk through an example to demonstrate the different steps of the Delta Algorithm.

Suppose we have two strings: `original` and `modified`, and we want to encode the differences between them.

Step 1: Initialization

- Original string: "Hello, world!"
- Modified string: "Hello, universe!"
- Initialize an empty delta string.

Step 2: Match Common Prefix

- Compare the original and modified strings from the beginning to find the longest common prefix.
  - Common prefix: "Hello, "

Step 3: Encode the Common Prefix

- Append "M5,0:Hello, " to the delta string, where "M5,0" indicates "Match 5 characters at position 0."

Step 4: Identify the Differences

- Identify the differences between the original and modified strings after the common prefix.
  - Original suffix: "world!"
  - Modified suffix: "universe!"

Step 5: Encode Differences

- Use the following format to encode differences:
  - "A" for Add: `A<length>:<data>`
  - "D" for Delete: `D<length>`
  - Append the encoded differences to the delta string.

In this case:

- "A5:universe!"

Step 6: Complete the Delta String

- The final delta string is "M5,0:Hello, A5:universe!"

To apply this delta to the original string to reconstruct the modified string:

- Initialize an empty result string.
- Start with the original string: "Hello, world!"
- Apply each operation from the delta string in order:
  - Apply the match operation: "Hello, " (at position 0).
  - Apply the add operation: "universe!"

The result is "Hello, universe!"—the modified string.

The Delta Algorithm is a basic example, and there are more advanced algorithms like VCDIFF that provide more efficient encoding and decoding. However, this example illustrates the fundamental steps of the Delta Algorithm for finding and encoding the differences between two strings.

Date: **Signature of faculty in-charge**