ml2-lab2

August 21, 2024

[1]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import numpy as np

# Load the Iris dataset
df = load_breast_cancer()
X, y = df.data, df.target

# 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)
```

[2]:
```
def initialize_network(input_size, hidden_size, output_size):
    np.random.seed(42)

    # Get user input for weights and biases
    print("Enter initial weight range:")
    weight_min = float(input("Min value: "))
    weight_max = float(input("Max value: "))

    print("Enter initial bias range:")
    bias_min = float(input("Min value: "))
    bias_max = float(input("Max value: "))

    # Initialize weights and biases
    weights_input_hidden = np.random.uniform(weight_min, weight_max,␣
↪(input_size, hidden_size))
    weights_hidden_output = np.random.uniform(weight_min, weight_max,␣
↪(hidden_size, output_size))
    bias_hidden = np.random.uniform(bias_min, bias_max, (1, hidden_size))
    bias_output = np.random.uniform(bias_min, bias_max, (1, output_size))

    return weights_input_hidden, weights_hidden_output, bias_hidden, bias_output

# Initialize the network
input_size = X_train.shape[1]
hidden_size = 5  # You can adjust this
```

```
output_size = 3  # Number of classes in Iris dataset

weights_input_hidden, weights_hidden_output, bias_hidden, bias_output =␣
 ↪initialize_network(input_size, hidden_size, output_size)
```

```
Enter initial weight range:
Min value: 5
Max value: 10
Enter initial bias range:
Min value: 10
Max value: 20
```

[3]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

def one_hot_encode(y, num_classes):
    return np.eye(num_classes)[y]

def feedforward(X, weights_input_hidden, weights_hidden_output, bias_hidden,␣
 ↪bias_output):
    # Hidden layer
    hidden_layer = sigmoid(np.dot(X, weights_input_hidden) + bias_hidden)
    # Output layer
    output_layer = softmax(np.dot(hidden_layer, weights_hidden_output) +␣
 ↪bias_output)
    return hidden_layer, output_layer

def backpropagation(X, y, weights_input_hidden, weights_hidden_output,␣
 ↪bias_hidden, bias_output, learning_rate, epochs):
    for epoch in range(epochs):
        # Feedforward
        hidden_layer, output_layer = feedforward(X, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output)

        # Calculate error
        y_one_hot = one_hot_encode(y, output_size)
        error = y_one_hot - output_layer

        # Backpropagation phase
        d_output = error
```

```
        d_hidden = np.dot(d_output, weights_hidden_output.T) *␣
 ↪sigmoid_derivative(hidden_layer)

        # Update weights and biases
        weights_hidden_output += learning_rate * np.dot(hidden_layer.T,␣
 ↪d_output)
        weights_input_hidden += learning_rate * np.dot(X.T, d_hidden)
        bias_output += learning_rate * np.sum(d_output, axis=0, keepdims=True)
        bias_hidden += learning_rate * np.sum(d_hidden, axis=0, keepdims=True)

        if epoch % 100 == 0:
            loss = -np.mean(y_one_hot * np.log(output_layer + 1e-8))
            print(f"Epoch {epoch}, Loss: {loss:.4f}")

    return weights_input_hidden, weights_hidden_output, bias_hidden, bias_output

# Train the network
learning_rate = 0.01
epochs = 1000

weights_input_hidden, weights_hidden_output, bias_hidden, bias_output =␣
 ↪backpropagation(
    X_train, y_train, weights_input_hidden, weights_hidden_output, bias_hidden,␣
 ↪bias_output, learning_rate, epochs
)

# Evaluate the model
_, y_pred = feedforward(X_test, weights_input_hidden, weights_hidden_output,␣
 ↪bias_hidden, bias_output)
accuracy = np.mean(np.argmax(y_pred, axis=1) == y_test)
print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 0, Loss: 1.9519
Epoch 100, Loss: 2.2241
Epoch 200, Loss: 2.2582
Epoch 300, Loss: 2.2805
Epoch 400, Loss: 2.2804
Epoch 500, Loss: 0.2264
Epoch 600, Loss: 2.2608
Epoch 700, Loss: 1.0188
Epoch 800, Loss: 0.2730
Epoch 900, Loss: 1.8567
Test accuracy: 0.6228
```

[4]:
```
import matplotlib.pyplot as plt
```

```
def train_and_evaluate(X_train, y_train, X_test, y_test, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output, learning_rate, epochs,␣
 ↪iterations):
    mse_history = []
    accuracy_history = []

    for iteration in range(iterations):
        # d) Perform multiple iterations of step c
        # e) Update weights accordingly
        weights_input_hidden, weights_hidden_output, bias_hidden, bias_output =␣
 ↪backpropagation(
            X_train, y_train, weights_input_hidden, weights_hidden_output,␣
 ↪bias_hidden, bias_output, learning_rate, epochs
        )

        # Calculate MSE
        _, y_pred_train = feedforward(X_train, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output)
        y_train_one_hot = one_hot_encode(y_train, output_size)
        mse = np.mean((y_train_one_hot - y_pred_train) ** 2)
        mse_history.append(mse)

        # Calculate accuracy
        _, y_pred_test = feedforward(X_test, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output)
        accuracy = np.mean(np.argmax(y_pred_test, axis=1) == y_test)
        accuracy_history.append(accuracy)

        print(f"Iteration {iteration + 1}, MSE: {mse:.4f}, Accuracy: {accuracy:.
 ↪4f}")

    return mse_history, accuracy_history
```

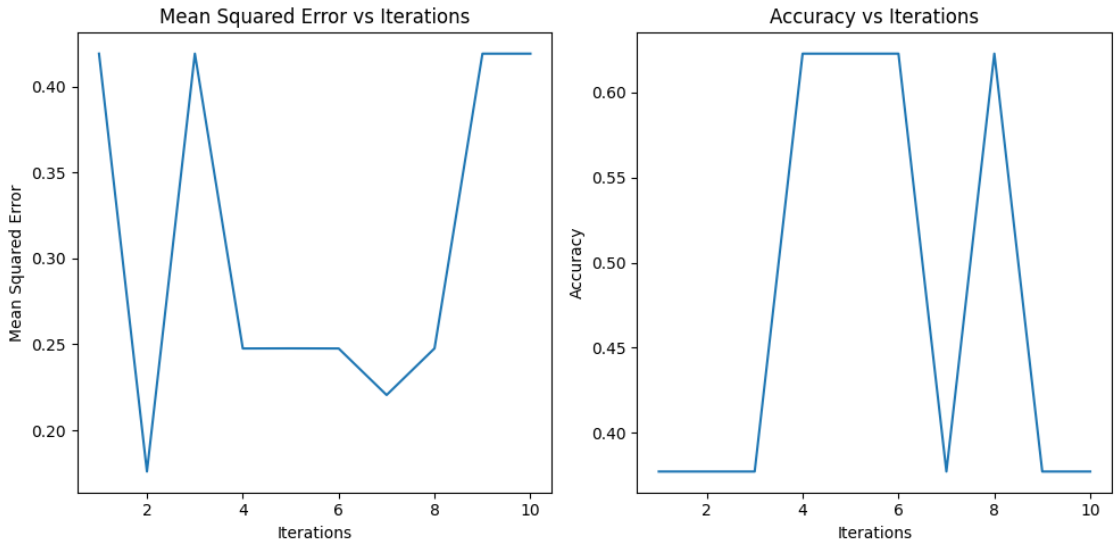
[5]:
```
# Set hyperparameters
learning_rate = 0.01
epochs = 1000
iterations = 10

# Train and evaluate the model
mse_history, accuracy_history = train_and_evaluate(
    X_train, y_train, X_test, y_test, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output, learning_rate, epochs,␣
 ↪iterations
)
```

```python
# f) Plot the mean squared error for each iteration
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(range(1, iterations + 1), mse_history)
plt.title('Mean Squared Error vs Iterations')
plt.xlabel('Iterations')
plt.ylabel('Mean Squared Error')

# g) Plot accuracy for iterations
plt.subplot(1, 2, 2)
plt.plot(range(1, iterations + 1), accuracy_history)
plt.title('Accuracy vs Iterations')
plt.xlabel('Iterations')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()

# Note the results
final_mse = mse_history[-1]
final_accuracy = accuracy_history[-1]
print(f"Final MSE: {final_mse:.4f}")
print(f"Final Accuracy: {final_accuracy:.4f}")
```

```
Epoch 0, Loss: 1.8225
Epoch 100, Loss: 1.7134
Epoch 200, Loss: 2.5033
Epoch 300, Loss: 2.4337
Epoch 400, Loss: 2.9740
Epoch 500, Loss: 0.3310
Epoch 600, Loss: 2.3666
Epoch 700, Loss: 0.2555
Epoch 800, Loss: 1.4936
Epoch 900, Loss: 0.2792
Iteration 1, MSE: 0.4190, Accuracy: 0.3772
Epoch 0, Loss: 3.1598
Epoch 100, Loss: 2.6377
Epoch 200, Loss: 0.2205
Epoch 300, Loss: 2.2806
Epoch 400, Loss: 1.8335
Epoch 500, Loss: 2.2794
Epoch 600, Loss: 0.2566
Epoch 700, Loss: 1.3908
Epoch 800, Loss: 3.1747
Epoch 900, Loss: 0.2246
Iteration 2, MSE: 0.1760, Accuracy: 0.3772
Epoch 0, Loss: 0.2404
```

```
Epoch 100, Loss: 2.2755
Epoch 200, Loss: 1.0218
Epoch 300, Loss: 2.2799
Epoch 400, Loss: 2.2614
Epoch 500, Loss: 3.0573
Epoch 600, Loss: 3.0074
Epoch 700, Loss: 0.3680
Epoch 800, Loss: 1.2894
Epoch 900, Loss: 2.2299
Iteration 3, MSE: 0.4190, Accuracy: 0.3772
Epoch 0, Loss: 2.9167
Epoch 100, Loss: 2.3041
Epoch 200, Loss: 3.1182
Epoch 300, Loss: 0.3641
Epoch 400, Loss: 2.2691
Epoch 500, Loss: 3.0694
Epoch 600, Loss: 2.2298
Epoch 700, Loss: 1.1000
Epoch 800, Loss: 2.2679
Epoch 900, Loss: 3.1101
Iteration 4, MSE: 0.2476, Accuracy: 0.6228
Epoch 0, Loss: 1.1205
Epoch 100, Loss: 2.8528
Epoch 200, Loss: 1.6169
Epoch 300, Loss: 0.2594
Epoch 400, Loss: 1.3079
Epoch 500, Loss: 2.2807
Epoch 600, Loss: 2.2744
Epoch 700, Loss: 2.2700
Epoch 800, Loss: 0.2199
Epoch 900, Loss: 2.2807
Iteration 5, MSE: 0.2476, Accuracy: 0.6228
Epoch 0, Loss: 2.2697
Epoch 100, Loss: 1.1055
Epoch 200, Loss: 0.3637
Epoch 300, Loss: 2.4962
Epoch 400, Loss: 1.1238
Epoch 500, Loss: 0.2549
Epoch 600, Loss: 0.2297
Epoch 700, Loss: 0.2978
Epoch 800, Loss: 0.2311
Epoch 900, Loss: 1.8075
Iteration 6, MSE: 0.2476, Accuracy: 0.6228
Epoch 0, Loss: 1.1203
Epoch 100, Loss: 0.3131
Epoch 200, Loss: 2.2353
Epoch 300, Loss: 3.1201
Epoch 400, Loss: 0.3079
```

```
Epoch 500, Loss: 0.2199
Epoch 600, Loss: 1.0520
Epoch 700, Loss: 0.2653
Epoch 800, Loss: 0.3334
Epoch 900, Loss: 2.1459
Iteration 7, MSE: 0.2205, Accuracy: 0.3772
Epoch 0, Loss: 0.2881
Epoch 100, Loss: 0.3003
Epoch 200, Loss: 2.3826
Epoch 300, Loss: 0.7312
Epoch 400, Loss: 0.2201
Epoch 500, Loss: 0.2594
Epoch 600, Loss: 2.2807
Epoch 700, Loss: 2.0405
Epoch 800, Loss: 0.2472
Epoch 900, Loss: 2.3714
Iteration 8, MSE: 0.2476, Accuracy: 0.6228
Epoch 0, Loss: 2.2586
Epoch 100, Loss: 2.2764
Epoch 200, Loss: 1.1062
Epoch 300, Loss: 2.2753
Epoch 400, Loss: 2.2759
Epoch 500, Loss: 1.8386
Epoch 600, Loss: 2.2807
Epoch 700, Loss: 1.0251
Epoch 800, Loss: 0.2743
Epoch 900, Loss: 2.2713
Iteration 9, MSE: 0.4190, Accuracy: 0.3772
Epoch 0, Loss: 2.4548
Epoch 100, Loss: 2.2770
Epoch 200, Loss: 2.2804
Epoch 300, Loss: 2.3525
Epoch 400, Loss: 1.1901
Epoch 500, Loss: 0.9243
Epoch 600, Loss: 2.2778
Epoch 700, Loss: 0.2204
Epoch 800, Loss: 2.6889
Epoch 900, Loss: 1.2020
Iteration 10, MSE: 0.4190, Accuracy: 0.3772
```



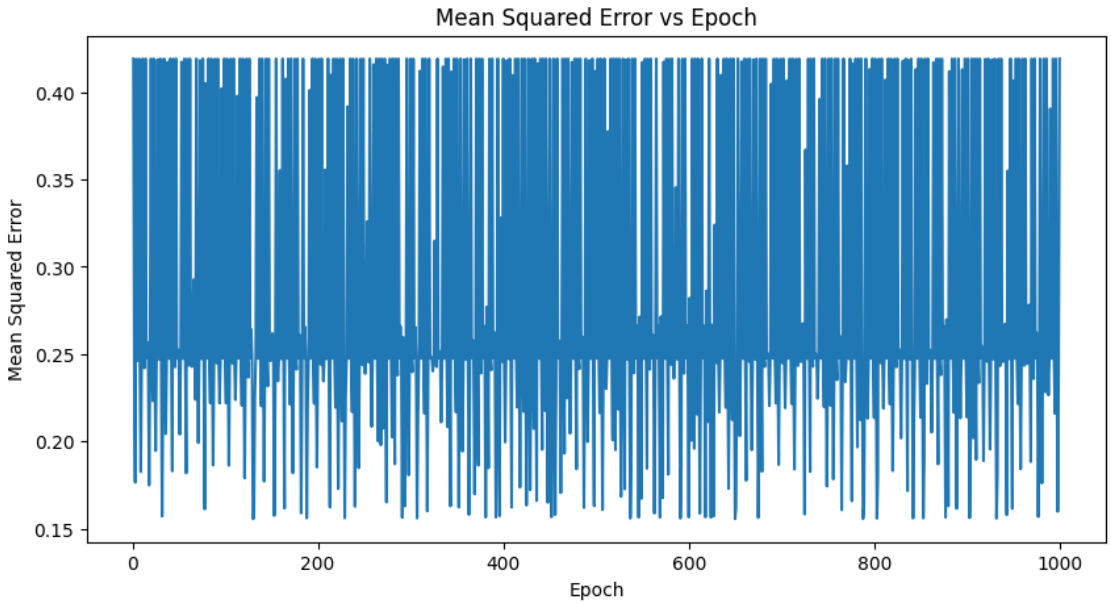

```
Final MSE: 0.4190
Final Accuracy: 0.3772
```

[6]:
```
import matplotlib.pyplot as plt

def train_and_evaluate(X_train, y_train, X_test, y_test, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output, learning_rate, epochs):
    mse_history = []

    for epoch in range(epochs):
        # Feedforward and backpropagation
        hidden_layer, output_layer = feedforward(X_train, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output)

        # Calculate error
        y_train_one_hot = one_hot_encode(y_train, output_size)
        error = y_train_one_hot - output_layer

        # Backpropagation phase
        d_output = error
        d_hidden = np.dot(d_output, weights_hidden_output.T) *␣
 ↪sigmoid_derivative(hidden_layer)

        # Update weights and biases
        weights_hidden_output += learning_rate * np.dot(hidden_layer.T,␣
 ↪d_output)
        weights_input_hidden += learning_rate * np.dot(X_train.T, d_hidden)
        bias_output += learning_rate * np.sum(d_output, axis=0, keepdims=True)
```

```
        bias_hidden += learning_rate * np.sum(d_hidden, axis=0, keepdims=True)

        # Calculate MSE for this epoch
        mse = np.mean((y_train_one_hot - output_layer) ** 2)
        mse_history.append(mse)

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, MSE: {mse:.4f}")

    return weights_input_hidden, weights_hidden_output, bias_hidden,␣
 ↪bias_output, mse_history

# Train the network and get MSE history
learning_rate = 0.01
epochs = 1000

weights_input_hidden, weights_hidden_output, bias_hidden, bias_output,␣
 ↪mse_history = train_and_evaluate(
    X_train, y_train, X_test, y_test, weights_input_hidden,␣
 ↪weights_hidden_output, bias_hidden, bias_output, learning_rate, epochs
)

# Predict for test data
_, y_pred = feedforward(X_test, weights_input_hidden, weights_hidden_output,␣
 ↪bias_hidden, bias_output)
y_pred_classes = np.argmax(y_pred, axis=1)

# Evaluate performance
accuracy = np.mean(y_pred_classes == y_test)
print(f"Test accuracy: {accuracy:.4f}")

# Plot MSE vs Epoch
plt.figure(figsize=(10, 5))
plt.plot(range(1, epochs + 1), mse_history)
plt.title('Mean Squared Error vs Epoch')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error')
plt.show()
```

```
Epoch 0, MSE: 0.4190
Epoch 100, MSE: 0.2218
Epoch 200, MSE: 0.4190
Epoch 300, MSE: 0.2476
Epoch 400, MSE: 0.2476
Epoch 500, MSE: 0.2475
Epoch 600, MSE: 0.2821
Epoch 700, MSE: 0.2449
```

```
Epoch 800, MSE: 0.4190
Epoch 900, MSE: 0.4190
Test accuracy: 0.6228
```



[6]:

COURSE CODE: DJS22DSL501

COURSE NAME: Machine Learning - II **CLASS: AY 2024-25**

LAB EXPERIMENT NO.2

NAME: FIONA HARIA **SAP-ID: 60009220048** **BATCH: D1-2** **ROLL NO: D040**

AIM: Implement backpropagation algorithm from scratch.

THEORY:

In the field of Artificial Neural Networks, the Backpropagation algorithm is a supervised learning approach for multilayer feed-forward networks.

The information processing of one or more brain cells, termed neurons, is the inspiration for feedforward neural networks. A neuron receives input signals through its dendrites, which then transmit the signal to the cell body. The axon transmits the signal to synapses, which are the connections between a cell's axon and the dendrites of other cells.

The backpropagation approach works on the premise of modelling a function by changing the internal weightings of input signals to produce an expected output signal. The system is taught using a supervised learning method, in which the difference between the system's output and a known expected output is supplied to it and utilised to change its internal state.

The backpropagation algorithm, in technical terms, is a method for training the weights in a multilayer feed-forward neural network. As a result, it necessitates the definition of a network structure consisting of one or more levels, each of which is fully connected to the next layer. One input layer, one hidden layer, and one output layer make up a conventional network topology. Backpropagation can be used for both classification and regression problems.

How does backpropagation algorithm work?

As we know in artificial neural networks, training occurs in various steps, from:

- Initialization.
- Forward propagation.
- Error Function.

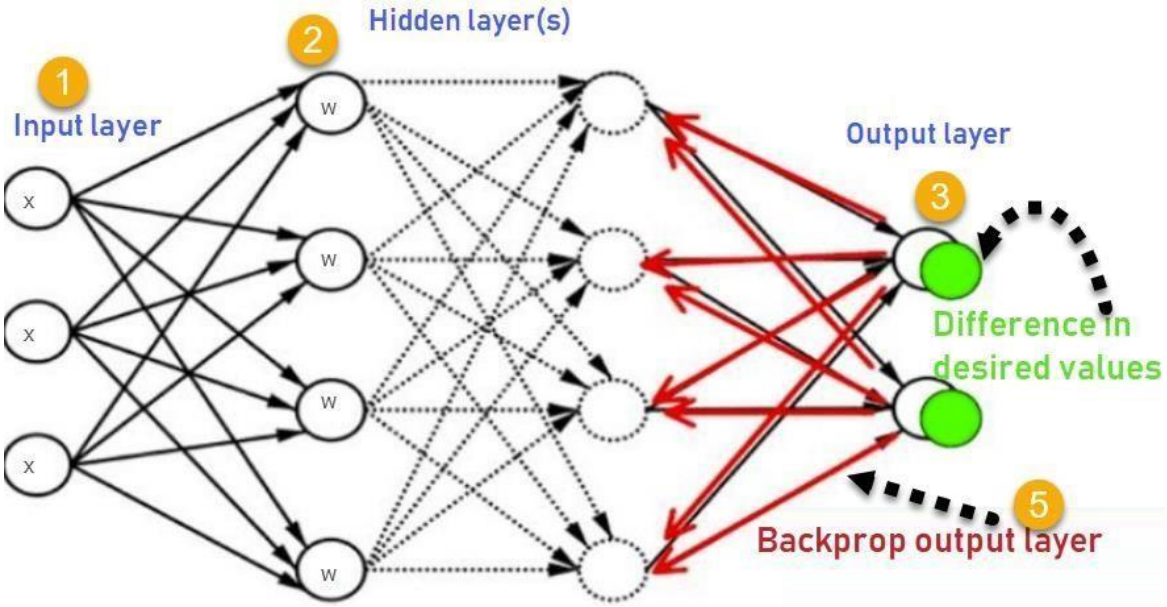
- Backpropagation.
- Weight Update.
- Iteration.

It is the fourth step of the process, a backpropagation algorithm that calculates the gradient of a loss function of the weights in the neural network to ensure the error function is minimum. However, the backpropagation algorithm accomplishes this through a set of Back **Propagation Algorithm Steps**, which involves:

- **Selecting Input & Output:** The first step of the backpropagation algorithm is to choose an input for the process and to set the desired output.
- **Setting Random Weights:** Once the input and output are set, random weights are allocated, as it will be needed to manipulate the input and output values. After this, the output of each neuron is calculated through the forward propagation, which goes through:
    - Input Layer
    - Hidden Layer
    - Output Layer
- **Error Calculation:** This is an important step that calculates the total error by determining how far and suitable the actual output is from the required output. This is done by calculating the errors at the output neuron.
- **Error Minimization:** Based on the observations made in the earlier step, here the focus is on minimizing the error rate to ensure accurate output is delivered.
- **Updating Weights & other Parameters:** If the error rate is high, then parameters (weights and biases) are changed and updated to reduce the rate of error using the delta rule or gradient descent. This is accomplished by assuming a suitable learning rate and propagating backward from the output layer to the previous layer. Acting as an example of dynamic programming, this helps avoid redundant calculations of repeated errors, neurons, and layers.
- **Modeling Prediction Readiness:** Finally, once the error is optimized, the output is tested with some testing inputs to get the desired result.

This process is repeated until the error reduces to a minimum and the desired output is obtained.

Consider the following Back propagation neural network example diagram to understand:



1. Inputs X, arrive through the preconnected path
2. Input is modelled using real weights W. The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs

   $Error_B$= Actual Output – Desired Output
5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased

   Process is repeated until the desired output is achieved.

Need of Backpropagation in Neural Network

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program
- It has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network

- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

Tasks to be performed:

a) **Take the favourable dataset**

b) **Initialize a neural network with random weights.**

c) **Backpropagation Neural Network:**
   i. **Do feedforward propagation**
   ii. **Calculate error**
   iii. **iii. Backpropagation phase**

d) **Perform multiple iterations of step c.**
e) **Update weights accordingly.**

f) **Plot the mean squared error for each iteration**

g) **Similarly plot accuracy for iterations and note the results.**

h) **Predict for test data and evaluate the performance of your backpropagation neural network.**