SOFT COMPUTING ASSIGNMENT -7

Perumalla Dharan
AP21110010201

1. Write a Python program to train a Back Propagation Neural Network (BPNN) for classifying whether a student passes or fails using a dataset of students' course marks. Assume necessary parameters.

```
import numpy as np
import pandas as pd

# Load the dataset
train_path = r'E:\SRM\Soft Computing\Lab 7 - 25th Sept\training_dataset_students(1000).csv'
train_df = pd.read_csv(train_path)
test_path = r'E:\SRM\Soft Computing\Lab 7 - 25th Sept\students_testing.csv'
test_df = pd.read_csv(test_path)

X_train = train_df[['c1', 'c2', 'c3', 'c4', 'c5', 'c6']].values
y_train = train_df[['result']].values

X_test = test_df[['c1', 'c2', 'c3', 'c4', 'c5', 'c6']].values
y_test = test_df[['result']].values

class MLP:
    def __init__(self, input_size, hidden_size, learning_rate=0.01, iterations=10000):
        self.learning_rate = learning_rate
        self.iterations = iterations
        self.weights_input_hidden = np.random.uniform(-0.5, 0.5, (input_size, hidden_size))
```

```python
        self.bias_hidden = np.random.uniform(-0.5, 0.5, (hidden_size,))
        self.weights_hidden_output = np.random.uniform(-0.5, 0.5, (hidden_size, 1))
        self.bias_output = np.random.uniform(-0.5, 0.5, (1,))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

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

    def predict(self, X):
        hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        hidden_output = self.sigmoid(hidden_input)
        final_input = np.dot(hidden_output, self.weights_hidden_output) + self.bias_output
        final_output = self.sigmoid(final_input)
        return np.round(final_output)

    def train(self, X, y):
        for epoch in range(self.iterations):
            for i in range(len(X)):
                # Forward pass
                hidden_input = np.dot(X[i], self.weights_input_hidden) + self.bias_hidden
                hidden_output = self.sigmoid(hidden_input)
                final_input = np.dot(hidden_output, self.weights_hidden_output) + self.bias_output
                final_output = self.sigmoid(final_input)

                #Finding gradiants and errors
                output_error = (y[i] - final_output)
```

```
                                output_gradient  =  output_error  *
 self.sigmoid_derivative(final_output)

                                                  hidden_error   =
 output_gradient.dot(self.weights_hidden_output.T)
                                hidden_gradient  =  hidden_error  *
 self.sigmoid_derivative(hidden_output)

                # Update weights and biases
                #updating weights (V) and bias
                 self.weights_hidden_output += self.learning_rate *
 hidden_output[:, None] * output_gradient
                          self.bias_output  +=  self.learning_rate  *
 output_gradient

                #updating weights (W) and bias
                 self.weights_input_hidden += self.learning_rate *
 X[i][:, None] * hidden_gradient
                self.bias_hidden += self.learning_rate


# Model configuration
input_size = X_train.shape[1]
hidden_size = 6

mlp     =     MLP(input_size=input_size,     hidden_size=hidden_size,
 learning_rate=0.01, iterations=1000)
mlp.train(X_train, y_train)

# Test predictions and accuracy
predictions = mlp.predict(X_test)
accuracy = np.mean(predictions == y_test)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Test Accuracy: 54.55%
```