Implement the finite words classification system using Back-propagation algorithm

In [1]:
```
import numpy as np
```

In [2]:
```
words = ['clustering', 'classification', 'neuralnetwork', 'supervisedlearning']
labels = np.array([[1, 0],
                   [1, 0],
                   [0, 1],
                   [0, 1]])
```

In [3]:
```
def encode_word(word):
    encoding = np.zeros((26,))
    for char in word:
        if char.isalpha():
            encoding[ord(char) - ord('a')] = 1
    return encoding
```

In [4]:
```
encoded_words = np.array([encode_word(word) for word in words])
```

In [5]:
```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate):
        self.learning_rate = learning_rate
        # Weights initialization
        self.W1 = np.random.rand(input_size, hidden_size)
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.rand(hidden_size, output_size)
        self.b2 = np.zeros((1, output_size))

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

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

    def forward(self, X):
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2)
        return self.a2

    def backward(self, X, y):
        output_error = y - self.a2
        output_delta = output_error * self.sigmoid_derivative(self.a2)

        hidden_error = output_delta.dot(self.W2.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.a1)

        # Update weights and biases
        self.W2 += self.a1.T.dot(output_delta) * self.learning_rate
        self.b2 += np.sum(output_delta, axis=0, keepdims=True) * self.learning_rate
        self.W1 += X.T.dot(hidden_delta) * self.learning_rate
        self.b1 += np.sum(hidden_delta, axis=0, keepdims=True) * self.learning_rate

    def train(self, X, y, epochs):
        for _ in range(epochs):
            self.forward(X)
            self.backward(X, y)

    def predict(self, X):
        output = self.forward(X)
        return np.argmax(output, axis=1)
```

In [6]:
```
input_size = 26
hidden_size = 5
output_size = 2
learning_rate = 0.1
epochs = 10000
```

In [7]:
```
nn = NeuralNetwork(input_size, hidden_size, output_size, learning_rate)
nn.train(encoded_words, labels, epochs)
```

In [8]:
```
test_words = ['cat', 'classification', 'neuralnetwork', 'supervisedlearning']
encoded_test_words = np.array([encode_word(word) for word in test_words])
predictions = nn.predict(encoded_test_words)
```

In [9]:
```
for word, pred in zip(test_words, predictions):
    print(f"Word: {word}, Predicted Class: {pred}")
```

```
Word: cat, Predicted Class: 0
Word: classification, Predicted Class: 0
Word: neuralnetwork, Predicted Class: 1
Word: supervisedlearning, Predicted Class: 1
```

In [ ]: