Kaliyev Madiyar
Lab Work 3
Report

Part 1

```
class NeuralNetwork:
    def __init__(self, input_size):
        self.input_size = input_size
        self.weights = np.random.randn(input_size + 1)
```

__init__: The constructor initialises a NeuralNetwork object. It takes input_size as a parameter representing the number of input features and initialises the input_size attribute , also it generates random weights using np.random.randn for the input features plus one offset term

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

sigmoid: This method implements a sigmoidal activation function. It takes the value of x and returns the result of applying the sigmoid function.

```
    def forward(self, x):
        z = np.dot(x, self.weights[1:]) + self.weights[0]
        self.pred = self.sigmoid(z)
        return self.pred
```

forward: This method performs a forward pass of the neural network. It takes an input signal x, computes a weighted sum of the inputs plus an offset, applies sigmoidal activation using the sigmoid method, and stores the result in self.pred. It returns the predicted output.

```
    def stochastic_gradient_descent(self, x, y, learning_rate, epochs):
        for epoch in range(epochs):
            for i in range(len(x)):
                prediction = self.forward(x[i])
                error = y[i] - prediction
                self.weights[1:] += learning_rate * error * x[i]
                self.weights[0] += learning_rate * error
```

stochastic_gradient_descent: This method implements stochastic gradient descent. It iterates over epochs and for each epoch it traverses each data point (x[i] and y[i]), computes the prediction, calculates the error and updates the weights using the learning rate.

```
    def batch_gradient_descent(self, x, y, learning_rate, epochs):
        for epoch in range(epochs):
            prediction = self.forward(x)
```

```
            errors = y - prediction
            self.weights[1:] += learning_rate * np.dot(errors, x)
            self.weights[0] += learning_rate * sum(errors)
```

batch_gradient_descent: This method implements batch gradient descent. It iterates over epochs, computes predictions for all data points, computes errors, and updates weights using the learning rate.

```
    def minibatch_gradient_descent(self, x, y, learning_rate, epochs,
batch_size=3):
        N = x.shape[0]
        num_batches = N // batch_size

        for epoch in range(epochs):
            indices = np.random.permutation(N)
            x_shuffled = x[indices]
            y_shuffled = y[indices]

            for i in range(0, N, batch_size):
                x_batch = x_shuffled[i:i + batch_size]
                y_batch = y_shuffled[i:i + batch_size]

                prediction = self.forward(x_batch)
                errors = y_batch - prediction

                self.weights[1:] += learning_rate * np.dot(errors,
x_batch)
                self.weights[0] += learning_rate * sum(errors)
```

minibatch_gradient_descent: This function implements minibatch gradient descent. It divides the dataset into batches, shuffles the data for each epoch, and then updates the weights using each batch.

```
    def predict(self, X):
        predictions = []
        for x in X:
            output = self.forward(x)
            predictions.append(output)
        return predictions
```

predict: This method takes a set of input data X and uses a trained network to make predictions for each input. It returns a list of predictions.

Part 2

```
y_one_hot = np.column_stack([(1 - y), y])
```

Converts binary labels to single-column encoding using np.column_stack.

```
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot,
test_size=0.2, random_state=42)
```

Split the dataset into training and test sets using train_test_split.

```
input_size = X_train.shape[1]
hidden_size = 2
output_size = 2
```

Defines the architecture of a neural network with 2 input features, a hidden layer with 2 blocks and an output layer with 2 blocks.

```
np.random.seed(42)
weights_input_hidden = np.random.randn(input_size, hidden_size)
biases_hidden = np.zeros((1, hidden_size))
weights_hidden_output = np.random.randn(hidden_size, output_size)
biases_output = np.zeros((1, output_size))
```

Initializes the weights and offsets for the neural network.

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

Defines the sigmoid activation function.

```
def categorical_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    loss = -np.sum(y_true * np.log(y_pred), axis=-1)
    return np.mean(loss)
```

The categorical_cross_entropy function computes the cross entropy loss between true class labels (y_true) and predicted probabilities (y_pred). It reduces numerical instability by trimming the predicted values to avoid very small or large logarithmic terms. Losses are computed on an item-by-item basis and then averaged over all items, giving a measure of the discrepancy between predicted and true class probabilities.

```
def forward_pass(X, weights_input_hidden, biases_hidden,
weights_hidden_output, biases_output):
    hidden_input = np.dot(X, weights_input_hidden) + biases_hidden
    hidden_output = sigmoid(hidden_input)
    output_input = np.dot(hidden_output, weights_hidden_output) +
biases_output
    output = sigmoid(output_input)
    return output
```

The forward_pass function calculates the output of the neural network by passing the input data through the hidden layer and the output layer. It uses sigmoidal activation functions for both layers, including weights and biases. The resulting output is the neural network's prediction for the given input data.

```
output_train = forward_pass(X_train, weights_input_hidden,
biases_hidden, weights_hidden_output, biases_output)
loss_train = categorical_cross_entropy(y_train, output_train)
```

Performs a forward pass over the training set and computes categorical cross-entropy loss.

```
output_test = forward_pass(X_test, weights_input_hidden, biases_hidden,
weights_hidden_output, biases_output)
loss_test = categorical_cross_entropy(y_test, output_test)
```

Performs a forward pass over the test set and computes categorical cross-entropy loss.

```
plt.scatter(X_train[y_train[:, 1] == 1, 0], X_train[y_train[:, 1] == 1,
1], c='blue', marker='o', label='Class 1 (Train)')
plt.scatter(X_train[y_train[:, 0] == 1, 0], X_train[y_train[:, 0] == 1,
1], c='red', marker='x', label='Class 0 (Train)')

plt.scatter(X_test[y_test[:, 1] == 1, 0], X_test[y_test[:, 1] == 1, 1],
c='green', marker='o', label='Class 1 (Test)')
plt.scatter(X_test[y_test[:, 0] == 1, 0], X_test[y_test[:, 0] == 1, 1],
c='purple', marker='x', label='Class 0 (Test)')

plt.title('Train-Test Split of XOR Dataset')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(loc='upper right')
plt.show()
```

Plots the training and test sets, distinguishing between classes using different colours.