Neural Networks- Fall 2023

Cracow University of Technology

Laboratory No. 4:

1) Implementation of backward pass of a two-layer neural network.

2) Neural Network Training with Regularization

Task 1

```
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Import necessary libraries

```
# Generate XOR dataset
np.random.seed(0)
X = np.random.randn(500, 2)
y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)
y = np.where(y, 1, 0)
```

Here, an XOR dataset with 500 samples and 2 features is generated. The labels (y) are determined by the XOR function applied to the features

```
# Convert binary labels to one-hot encoding
y_one_hot = np.column_stack([(1 - y), y])
```

Converts binary labels to single-code encoding using column_stack.

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot,
test_size=0.2, random_state=42)
```

Splits the dataset into training and testing sets (80% training, 20% testing) using train_test_split.

```
# Define the architecture of the neural network
input_size = X_train.shape[1]
hidden_size = 2
output_size = 2  # Two classes for one-hot encoding
# Initialize weights using Xavier initialization
weights_input_hidden = np.random.randn(input_size, hidden_size) /
np.sqrt(input_size)
biases_hidden = np.zeros((1, hidden_size))
weights_hidden_output = np.random.randn(hidden_size, output_size) /
np.sqrt(hidden_size)
biases_output = np.zeros((1, output_size))
```

Defines the architecture of the neural network and initialises the weights.

```
# Define the Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Define the derivative of the Sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
```

In my previous code, I have mistakenly used relu instead of sigmoid, here is the fixed version with sigmoid function.

```
# Define binary cross-entropy loss
def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    loss = -(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 -
y_pred))
    return np.mean(loss)
```

This code defines the binary_cross_entropy function, which computes the binary cross entropy loss between true labels (y_true) and predicted probabilities (y_pred).

```
# Forward pass
def forward_pass(X, weights_input_hidden, biases_hidden,
weights_hidden_output, biases_output):
    # Input to hidden layer
    hidden_input = np.dot(X, weights_input_hidden) + biases_hidden
    hidden_output = sigmoid(hidden_input)

    # Hidden to output layer
    output_input = np.dot(hidden_output, weights_hidden_output) +
biases_output
    output = sigmoid(output_input)

    return output, hidden_output
```

The forward_pass function computes the forward pass of a neural network with one hidden layer and sigmoidal activation. It takes input features (X), applies weights and offsets to compute the output of the hidden layer, applies sigmoidal activation, and then computes the output of the output layer with sigmoidal activation. The final results are returned for further use in the training process.

```
# Backward pass (Backpropagation)
```

```
def backward_pass(X, y, output, hidden_output, weights_hidden_output,
weights_input_hidden, biases_output, biases_hidden, learning_rate):
    # Calculate the gradients
    output_error = y - output
    output_delta = output_error * output * (1 - output)  # Derivative
of the sigmoid function

    hidden_error = output_delta.dot(weights_hidden_output.T)
    hidden_delta = hidden_error * sigmoid_derivative(hidden_output)

    # Update weights and biases
    weights_hidden_output += hidden_output.T.dot(output_delta) *
learning_rate
    biases_output += np.sum(output_delta, axis=0, keepdims=True) *
learning_rate

    weights_input_hidden += X.T.dot(hidden_delta) * learning_rate
    biases_hidden += np.sum(hidden_delta, axis=0, keepdims=True) *
learning_rate
```

This code defines the backward_pass function that implements the backward propagation algorithm, which is used to update the weights and biases of the neural network during training. The backward propagation algorithm calculates gradients with respect to the network parameters and adjusts these parameters to minimise the error.

```
# Training the neural network
epochs = 10000
learning_rate = 0.01

for epoch in range(epochs):
    # Forward pass
    output, hidden_output = forward_pass(X_train, weights_input_hidden,
biases_hidden, weights_hidden_output, biases_output)

    # Backward pass
    backward_pass(X_train, y_train, output, hidden_output,
weights_hidden_output, weights_input_hidden, biases_output,
biases_hidden, learning_rate)

    # Print the loss every 1000 epochs
    if epoch % 1000 == 0:
        loss = binary_cross_entropy(y_train, output)
        print(f'Epoch {epoch}, Loss: {loss}')
```

Trains a neural network by iteratively performing forward and backward passes for a given number of epochs. Outputs losses every 1000 epochs

```
# Testing the trained neural network
output_test, _ = forward_pass(X_test, weights_input_hidden,
biases_hidden, weights_hidden_output, biases_output)

# Calculate binary cross-entropy loss on the testing set
loss_test = binary_cross_entropy(y_test, output_test)

# Print the testing loss
print(f'Testing Loss: {loss_test}')
```

Tests the trained neural network on the testing set and prints the testing loss.

```
# Plot the training set
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)')

# Plot the testing set
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 XOR dataset, differentiating between training and testing sets and classes using different colours and markers.

Task 2

```
def initialize_weights(input_size, hidden_size, output_size):
    W1 = np.random.randn(input_size, hidden_size) * 0.01
    b1 = np.zeros((1, hidden_size))
    W2 = np.random.randn(hidden_size, output_size) * 0.01
    b2 = np.zeros((1, output_size))
    return W1, b1, W2, b2
```

Initializes the weights (W1, W2) and biases (b1, b2) of the neural network with small random values.

```
def forward_pass(X, W1, b1, W2, b2):
    hidden_input = np.dot(X, W1) + b1
    hidden_output = relu(hidden_input)
    output_input = np.dot(hidden_output, W2) + b2
    predicted_output = softmax(output_input)
    return hidden_input, hidden_output, output_input, predicted_output
```

Computes the forward propagation of the neural network by determining the input and output of the hidden layer using ReLU activation, and setting the input and predicted output of the output layer using Softmax activation.

```
def backward_pass(X, y, hidden_input, hidden_output, output_input,
predicted_output, W1, W2):
    grad_output_input = derivative_categorical_cross_entropy_loss(y,
predicted_output)
    grad_W2 = np.dot(hidden_output.T, grad_output_input)
    grad_b2 = np.sum(grad_output_input, axis=0, keepdims=True)
    grad_hidden_input = np.dot(grad_output_input, W2.T) *
derivative_relu(hidden_input)
    grad_W1 = np.dot(X.T, grad_hidden_input)
    grad_b1 = np.sum(grad_hidden_input, axis=0, keepdims=True)
    return grad_W1, grad_b1, grad_W2, grad_b2
```

Implements backpropagation to compute gradients and outputs gradients for weights and offsets in both layers.

```
def update_weights(W1, b1, W2, b2, grad_W1, grad_b1, grad_W2, grad_b2,
learning_rate):
    W1 -= learning_rate * grad_W1
    b1 -= learning_rate * grad_b1
    W2 -= learning_rate * grad_W2
    b2 -= learning_rate * grad_b2
    return W1, b1, W2, b2
```

Updating weights and offsets using gradients and learning rate

```
def calculate_accuracy(y_true, y_pred):
    correct_predictions = np.sum(np.argmax(y_true, axis=1) ==
np.argmax(y_pred, axis=1))
    total_samples = y_true.shape[0]
    accuracy = correct_predictions / total_samples
    return accuracy
```

Calculates model accuracy based on true and predicted labels.

```
def relu(x):
    return np.maximum(0, x)

def derivative_relu(x):
    return np.where(x > 0, 1, 0)

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

Implements the ReLU and softmax activation functions and their derivatives.

```
def categorical_cross_entropy_loss(y_true, y_pred):
    eps = 1e-15
    y_pred = np.clip(y_pred, eps, 1 - eps)
    loss = -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]
    return loss

def derivative_categorical_cross_entropy_loss(y_true, y_pred):
    return (y_pred - y_true) / y_true.shape[0]

def L1_reg(lambda_, W1, W2):
    return lambda_ * (np.sum(np.abs(W1)) + np.sum(np.abs(W2)))

def derivative_L1_reg(lambda_, W):
    return lambda_ * np.sign(W)

def L2_reg(lambda_, W1, W2):
    return lambda_ * (np.sum(np.square(W1)) + np.sum(np.square(W2)))

def derivative_L2_reg(lambda_, W):
    return lambda_ * 2 * W
```

Defines the categorical cross-entropy loss function and its derivative.
Implements the regularization conditions L1 and L2 and their derivatives.

```
# Load the iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Normalize the input data
X = X / np.max(X)

# One-hot encode the target variable
encoder = OneHotEncoder()
y = encoder.fit_transform(y.reshape(-1, 1)).toarray()
```

Loads the Iris dataset and normalises the input features.
One-hot encodes the target variable.

```
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.25, random_state=42)

# Define the neural network architecture
input_size = X_train.shape[1]
hidden_size = 32
output_size = y_train.shape[1]

# Initialize weights and biases
W1, b1, W2, b2 = initialize_weights(input_size, hidden_size,
output_size)

# Define the hyperparameters
learning_rate = 0.1
num_epochs = 1000
batch_size = 16
l1_lambda = 0.001
l2_lambda = 0.001

# Training the neural network
loss_history = []

for epoch in range(num_epochs):
    epoch_loss = 0.0  # Variable to store the cumulative loss for the
epoch
    for i in range(0, X_train.shape[0], batch_size):
        # Forward pass
```

```
        batch_X = X_train[i:i + batch_size]
        batch_y = y_train[i:i + batch_size]

        hidden_input, hidden_output, output_input, predicted_output =
forward_pass(batch_X, W1, b1, W2, b2)

        # Calculate loss with L1 and L2 regularization
        loss = categorical_cross_entropy_loss(batch_y,
predicted_output)
        l1_term = L1_reg(l1_lambda, W1, W2)
        l2_term = L2_reg(l2_lambda, W1, W2)
        total_loss = loss + l1_term + l2_term

        epoch_loss += total_loss  # Accumulate the loss per mini-batch

        # Backward pass
        grad_W1, grad_b1, grad_W2, grad_b2 = backward_pass(batch_X,
batch_y, hidden_input, hidden_output, output_input, predicted_output,
W1, W2)

        # Update weights and biases using gradients and learning rate
        W1, b1, W2, b2 = update_weights(W1, b1, W2, b2, grad_W1,
grad_b1, grad_W2, grad_b2, learning_rate)

    # Calculate average loss for the epoch
    avg_epoch_loss = epoch_loss / (X_train.shape[0] / batch_size)
    loss_history.append(avg_epoch_loss)

# Test the model using the test set
_, _, _, test_predictions = forward_pass(X_test, W1, b1, W2, b2)

# Calculate and print the accuracy
accuracy = calculate_accuracy(y_test, test_predictions)
print(f'Test Accuracy: {accuracy}')

# Plot the training loss over epochs
plt.plot(range(num_epochs), loss_history, label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')
plt.legend()
plt.show()
```

In this section, the code divides the data set into training and test sets, where 80% is allocated to training and 20% to testing. Then, the neural network architecture is defined, specifying the sizes of the input, hidden and output layers. Weights and biases are initialised and hyperparameters are set. A training cycle begins, in which each epoch processes a training set of mini-batches. For each mini-batch in the forward loop, predictions and losses are computed with L1 and L2 regularisation. In the backward pass, gradients are computed and weights and offsets are updated. The training losses are accumulated for each epoch. After training, the model is tested on a separate test set and the training losses are plotted by epoch for clarity.