RegressionMLP

October 20, 2024

[1]:
```
import numpy as np

# Load the dataset
data = np.genfromtxt('/home/darksst/Desktop/Fall24/StatisticalDecisionTheory/
↪Data/gt_data/gt_2015.csv', skip_header=1, delimiter=',', usecols=(0, 3, 8,␣
↪9, 10))

# Clean the data (remove rows with missing values)
data_clean = data[~np.isnan(data).any(axis=1)]

#The response matrix
Y = data_clean[:, 3].reshape(-1, 1)  # Reshape Y to make it a column vector

#The feature matrix
X = data_clean[:, [0, 1]]

# Print the shapes of X and Y to verify
print("Shape of X:", X.shape)
print("Shape of Y:", Y.shape)
```

```
Shape of X: (7384, 2)
Shape of Y: (7384, 1)
```

[2]:
```
input_size = X.shape[1]
hidden = 2
output = 1

bound = 1/np.sqrt(2)

#Weights and Biases for the hidden layer
W1 = np.random.uniform(low=-bound, high=bound, size=(input_size, hidden))
b1 = np.random.uniform(low=-bound, high=bound, size=(1, hidden))

#Weights and Biases for the output layer
W2 = np.random.uniform(low=-bound, high=bound, size=(hidden, output))
b2 = np.random.uniform(low=-bound, high=bound, size=(1, output))

# Print initialized parameters to verify
```

```python
print("W1 (hidden layer weights):\n", W1)
print("b1 (hidden layer biases):\n", b1)
print("W2 (output layer weights):\n", W2)
print("b2 (output layer biases):\n", b2)
```

```
W1 (hidden layer weights):
 [[-0.63055488  0.38743156]
 [ 0.16238567  0.15180489]]
b1 (hidden layer biases):
 [[ 0.61807912 -0.22531297]]
W2 (output layer weights):
 [[-0.3759852 ]
 [-0.37538897]]
b2 (output layer biases):
 [[-0.18014044]]
```

[3]:

```python
# ReLU activation function
def relu(Z):
    return np.maximum(0, Z)

# Forward propagation function
def forward_propagation(X, W1, b1, W2, b2):
    # Step 1: Compute the pre-activation for the hidden layer
    Z1 = np.dot(X, W1) + b1  # Pre-activation for hidden layer

    # Step 2: Apply ReLU activation function for the hidden layer
    A1 = relu(Z1)  # Activation for hidden layer

    # Step 3: Compute the pre-activation for the output layer (no activation)
    Z2 = np.dot(A1, W2) + b2  # Pre-activation for output layer (regression)

    # Output of the forward propagation (Z2 is the predicted CO)
    return Z2, A1  # Return both Z2 (output) and A1 (hidden layer activation)


Z2, A1 = forward_propagation(X, W1, b1, W2, b2)

# Print the output (Z2) and hidden layer activations (A1)
print("Output of the network (predicted CO values):\n", Z2)
print("Activations of the hidden layer (A1):\n", A1)
```

```
Output of the network (predicted CO values):
 [[-0.5238261 ]
 [-0.49878208]
 [-0.56898997]
 …
 [-1.08163183]
 [-1.17825172]
```

```
 [-1.19471848]]
Activations of the hidden layer (A1):
 [[0.         0.91554545]
 [0.23807225 0.61038021]
 [0.4708415  0.5642683 ]
 …
 [0.         2.40148608]
 [0.         2.65887216]
 [0.         2.70273802]]
```

[4]:
```python
# Mean Squared Error (MSE) loss function
def compute_loss(Y, Y_hat):
    m = Y.shape[0]  # Number of samples
    loss = (1/m) * np.sum((Y_hat - Y)**2)  # MSE formula
    return loss

# Perform forward propagation to get predicted CO values
Y_hat, _ = forward_propagation(X, W1, b1, W2, b2)

# Compute the loss (MSE)
loss = compute_loss(Y, Y_hat)

# Print the loss
print("Mean Squared Error (MSE) Loss:", loss)
```

```
Mean Squared Error (MSE) Loss: 39.629756089767056
```

[5]:
```python
# ReLU derivative function
def relu_derivative(Z):
    return Z > 0  # Derivative of ReLU is 1 for Z > 0, and 0 otherwise

# Backpropagation function
def backward_propagation(X, Y, W1, b1, W2, b2, A1, Y_hat):
    m = Y.shape[0]  # Number of samples

    # Step 1: Compute gradients for the output layer
    dZ2 = Y_hat - Y  # Gradient of loss with respect to Z2
    dW2 = (1/m) * np.dot(A1.T, dZ2)  # Gradient of loss with respect to W2
    db2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)  # Gradient of loss with␣
 ↪respect to b2

    # Step 2: Backpropagate into the hidden layer
    dA1 = np.dot(dZ2, W2.T)  # Gradient of loss with respect to A1
    dZ1 = dA1 * relu_derivative(A1)  # Gradient of loss with respect to Z1␣
 ↪(apply ReLU derivative)
    dW1 = (1/m) * np.dot(X.T, dZ1)  # Gradient of loss with respect to W1
    db1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)  # Gradient of loss with␣
 ↪respect to b1
```

```
    return dW1, db1, dW2, db2

# Perform forward propagation to get predicted CO values and hidden layer␣
 ↪activations
Y_hat, A1 = forward_propagation(X, W1, b1, W2, b2)

# Perform backward propagation to compute gradients
dW1, db1, dW2, db2 = backward_propagation(X, Y, W1, b1, W2, b2, A1, Y_hat)

# Print the computed gradients
print("dW1 (gradient of W1):\n", dW1)
print("db1 (gradient of b1):\n", db1)
print("dW2 (gradient of W2):\n", dW2)
print("db2 (gradient of b2):\n", db2)
```

```
dW1 (gradient of W1):
 [[-2.61155242e-02  3.93858315e+01]
 [ 1.32814007e-01  7.80190564e+00]]
db1 (gradient of b1):
 [[0.04239733 2.22126353]]
dW2 (gradient of W2):
 [[ -0.17085583]
 [-42.47115221]]
db2 (gradient of b2):
 [[-5.95287532]]
```

[6]:
```
# Gradient descent update function
def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    # Update weights and biases for the hidden layer
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1

    # Update weights and biases for the output layer
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2

    return W1, b1, W2, b2

# Set the learning rate
alpha = 0.01  # You can adjust this learning rate value if needed

# Perform the parameter update
W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)

# Print updated parameters
print("Updated W1 (hidden layer weights):\n", W1)
```

```python
print("Updated b1 (hidden layer biases):\n", b1)
print("Updated W2 (output layer weights):\n", W2)
print("Updated b2 (output layer biases):\n", b2)
```

```
Updated W1 (hidden layer weights):
 [[-0.63029373 -0.00642675]
 [ 0.16105753  0.07378583]]
Updated b1 (hidden layer biases):
 [[ 0.61765515 -0.24752561]]
Updated W2 (output layer weights):
 [[-0.37427664]
 [ 0.04932255]]
Updated b2 (output layer biases):
 [[-0.12061169]]
```

[7]:

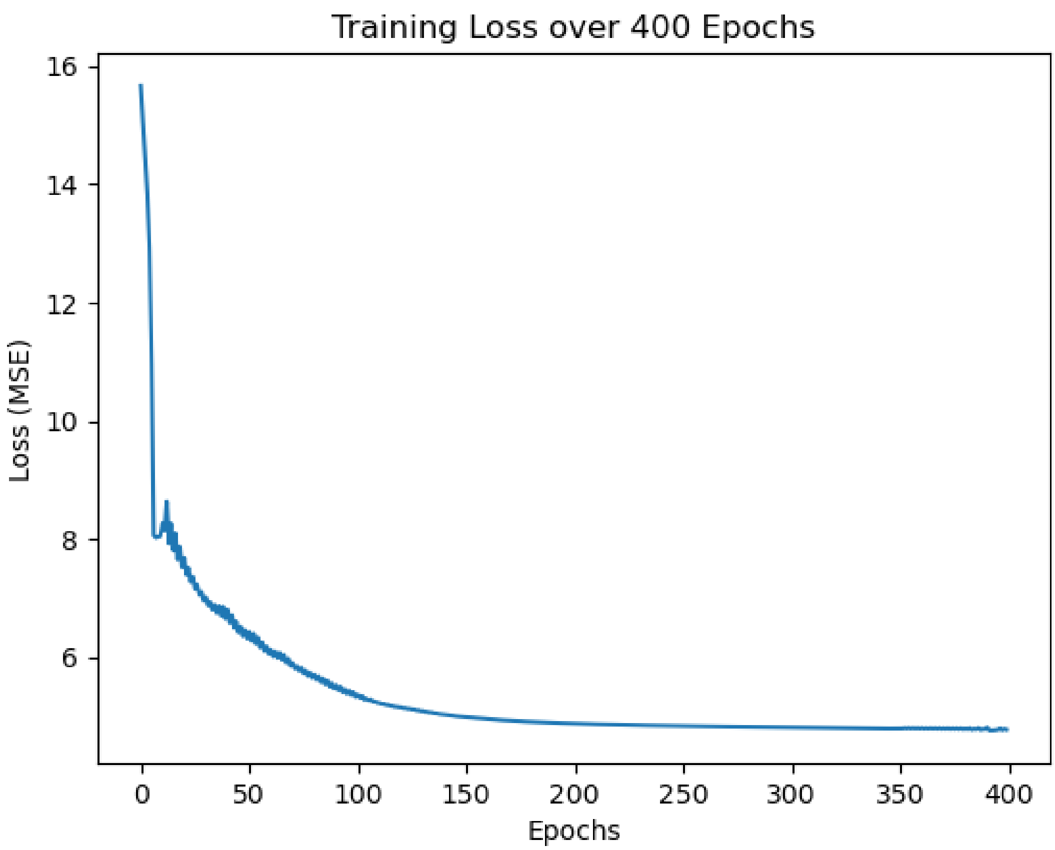
```python
# Training function for multiple epochs
def train(X, Y, W1, b1, W2, b2, epochs, alpha):
    loss_history = []  # To store loss values for each epoch

    for epoch in range(epochs):
        # Step 1: Forward propagation
        Y_hat, A1 = forward_propagation(X, W1, b1, W2, b2)

        # Step 2: Compute the loss (MSE)
        loss = compute_loss(Y, Y_hat)
        loss_history.append(loss)

        # Step 3: Backward propagation
        dW1, db1, dW2, db2 = backward_propagation(X, Y, W1, b1, W2, b2, A1,␣
 ↪Y_hat)

        # Step 4: Update the parameters
        W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2,␣
 ↪alpha)

        # Print the loss every 50 epochs
        if epoch % 50 == 0:
            print(f'Epoch {epoch}/{epochs}, Loss: {loss}')

    return W1, b1, W2, b2, loss_history
```

[8]:

```python
# Set the number of epochs and learning rate
epochs = 400
alpha = 0.03

# Train the model
W1, b1, W2, b2, loss_history = train(X, Y, W1, b1, W2, b2, epochs, alpha)
```

```
# Plot the loss over epochs
import matplotlib.pyplot as plt

plt.plot(range(epochs), loss_history)
plt.xlabel('Epochs')
plt.ylabel('Loss (MSE)')
plt.title('Training Loss over 400 Epochs')
plt.show()
```

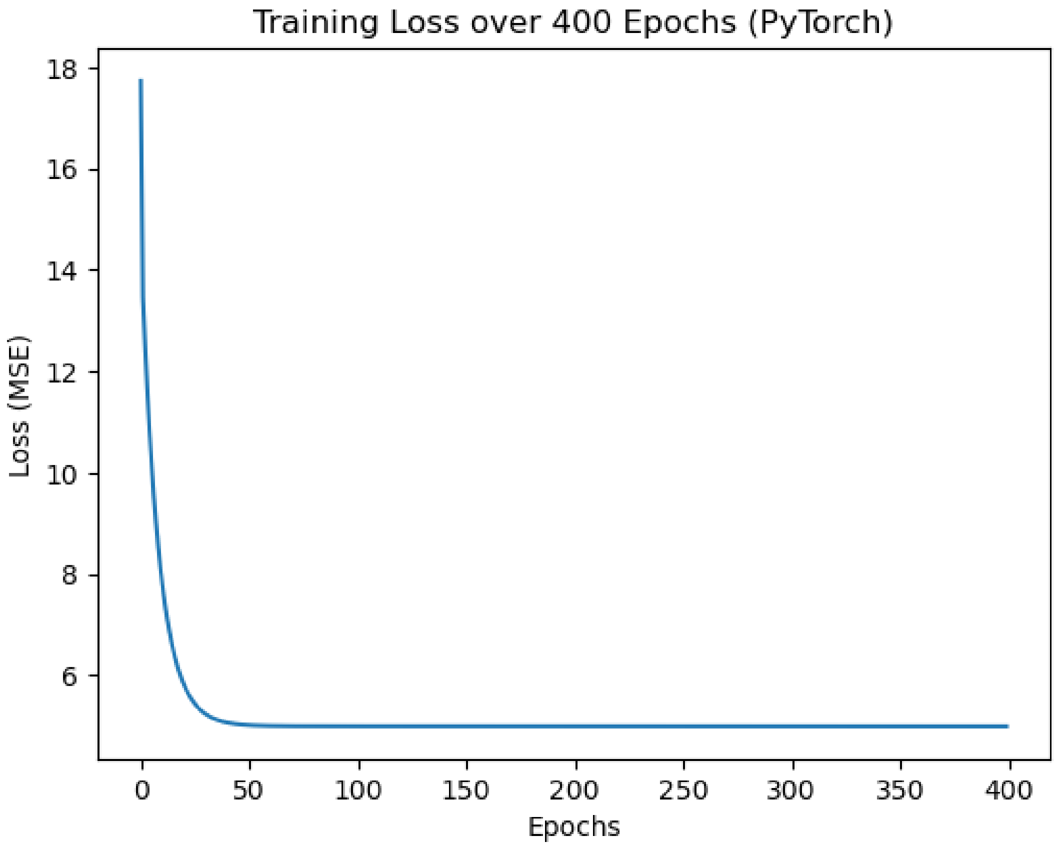
```
Epoch 0/400, Loss: 15.66775683381962
Epoch 50/400, Loss: 6.430319266208102
Epoch 100/400, Loss: 5.367560358904951
Epoch 150/400, Loss: 4.986352346613192
Epoch 200/400, Loss: 4.877048279699598
Epoch 250/400, Loss: 4.836889932618573
Epoch 300/400, Loss: 4.811934859105808
Epoch 350/400, Loss: 4.7956200790935535
```



[9]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# Convert the NumPy arrays to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
Y_tensor = torch.tensor(Y, dtype=torch.float32)

class MLPModel(nn.Module):
    def __init__(self):
        super(MLPModel, self).__init__()
        # Define layers
        self.hidden = nn.Linear(2, 2)  # 2 input features, 2 neurons in the␣
 ↪hidden layer
        self.relu = nn.ReLU()  # ReLU activation for the hidden layer
        self.output = nn.Linear(2, 1)  # 1 output neuron (CO)

    def forward(self, x):
        # Forward pass
        x = self.hidden(x)  # Hidden layer computation
        x = self.relu(x)     # ReLU activation
        x = self.output(x)   # Output layer
        return x

# Initialize the model
model = MLPModel()

# Loss function: Mean Squared Error (MSE)
criterion = nn.MSELoss()

# Optimizer: Stochastic Gradient Descent (SGD)
learning_rate = 0.03
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

# Training loop
def train_model(model, X_tensor, Y_tensor, epochs, optimizer, criterion):
    loss_history = []  # To store loss values at each epoch

    for epoch in range(epochs):
        # Forward pass: Compute predicted Y by passing X to the model
        Y_pred = model(X_tensor)

        # Compute the loss
        loss = criterion(Y_pred, Y_tensor)
```

```
        # Backward pass: Compute gradients
        optimizer.zero_grad()  # Clear previous gradients
        loss.backward()        # Backpropagation

        # Update parameters
        optimizer.step()

        # Store the loss
        loss_history.append(loss.item())

        # Print loss every 50 epochs
        if epoch % 50 == 0:
            print(f'Epoch {epoch}/{epochs}, Loss: {loss.item()}')

    return loss_history

# Train the model for 400 epochs
epochs_400 = 400
loss_history_400 = train_model(model, X_tensor, Y_tensor, epochs_400,␣
 ↪optimizer, criterion)

# Plot the loss over 400 epochs
plt.plot(range(epochs_400), loss_history_400)
plt.xlabel('Epochs')
plt.ylabel('Loss (MSE)')
plt.title('Training Loss over 400 Epochs (PyTorch)')
plt.show()
```

```
Epoch 0/400, Loss: 17.73300552368164
Epoch 50/400, Loss: 5.015661716461182
Epoch 100/400, Loss: 4.994784355163574
Epoch 150/400, Loss: 4.993960380554199
Epoch 200/400, Loss: 4.993415832519531
Epoch 250/400, Loss: 4.993067741394043
Epoch 300/400, Loss: 4.992757797241211
Epoch 350/400, Loss: 4.992537021636963
```


Training Loss over 400 Epochs (PyTorch)
Loss (MSE)
18
16
14
12
10
8
6
0
50
100
150
200
250
300
350
400
Epochs