Let $L$ be the loss function.

$Z_l$ is the pre-activation at layer $l$

$B_l$ is the parameter matrix (weights and biases) for layer $l$.

Then the gradient of loss with respect to parameters at layer $l$

$$\frac{dL}{dB_l} = \frac{dL}{dZ_l} \cdot \frac{dZ_l}{dB_l}$$

Chain rule for gradients state that

$$\frac{dL}{dB_l} = \frac{dL}{dZ_L} \cdot \frac{dZ_L}{dZ_{L-1}} \cdot \frac{dZ_{L-1}}{dZ_{L-2}} \cdot \ldots \cdot \frac{dZ_{l+1}}{dZ_l} \cdot \frac{dZ_l}{dB_l}$$

Considering all weights are initialized to 0

$\frac{dZ_l}{dB_l} = 0$ when multiplied in the chain rule will result to 0 for all layers following the gradient descent.

For the last layer $L$, the error is directly dependent on the final loss.

Therefore, $\frac{dL}{dB_L} \neq 0$ for the final layer.

The chain rule for gradient descent will not equate to zero as $\frac{dL}{dB_L} \neq 0$.

Therefore, during back propagation from final loss the parameters will update.

# RegressionMLP

October 21, 2024

[1]:
```
import numpy as np

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

data_clean = data[~np.isnan(data).any(axis=1)]

Y = data_clean[:, 3].reshape(-1, 1)
X = data_clean[:, [0, 1]]

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))
```

[3]:
```
def relu(Z):
    return np.maximum(0, Z)

def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1

    A1 = relu(Z1)
```

```
    Z2 = np.dot(A1, W2) + b2

    return Z2, A1

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

[4]:
```
def compute_loss(Y, Y_hat):
    m = Y.shape[0]
    loss = (1/m) * np.sum((Y_hat - Y)**2)
    return loss

Y_hat, _ = forward_propagation(X, W1, b1, W2, b2)

loss = compute_loss(Y, Y_hat)

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

```
Mean Squared Error (MSE) Loss: 66.78648862297081
```

[5]:
```
def relu_derivative(Z):
    return Z > 0

def backward_propagation(X, Y, W1, b1, W2, b2, A1, Y_hat):
    m = Y.shape[0]
    dZ2 = Y_hat - Y
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * relu_derivative(A1)
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)

    return dW1, db1, dW2, db2

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

dW1, db1, dW2, db2 = backward_propagation(X, Y, W1, b1, W2, b2, A1, Y_hat)
```

[6]:
```
def update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2

    return W1, b1, W2, b2
```

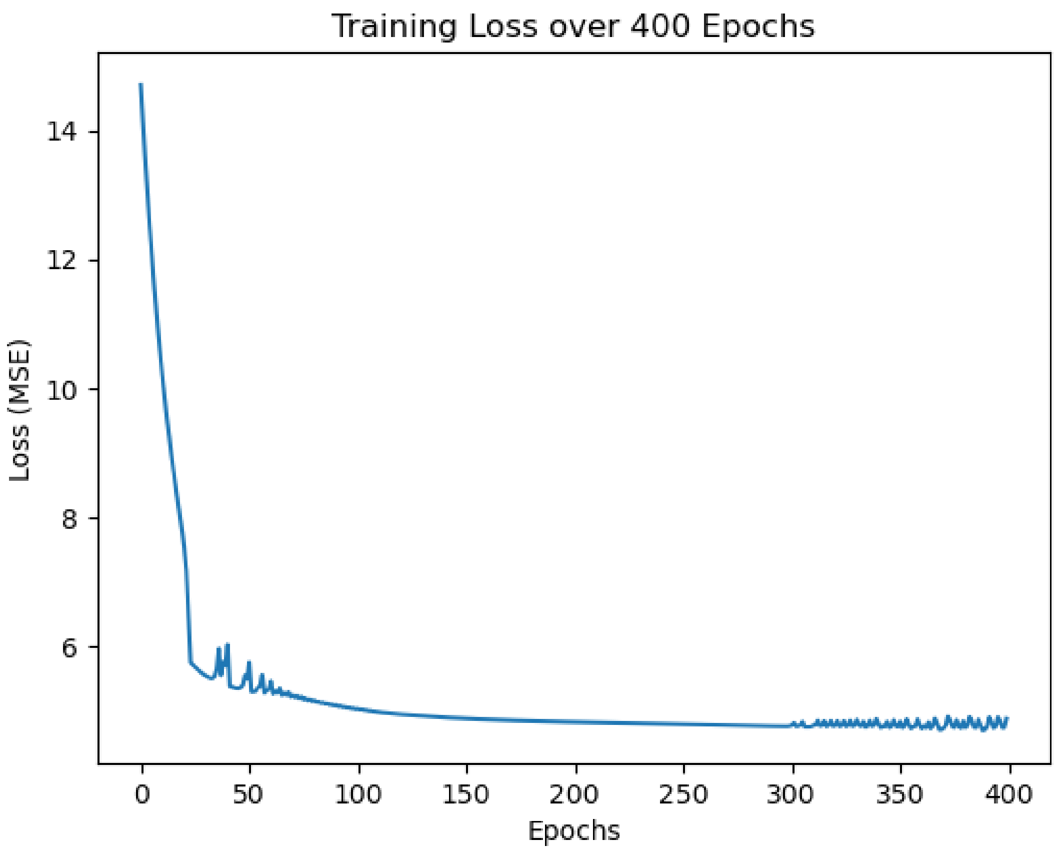
```
alpha = 0.01

W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
```

[7]:
```
def train(X, Y, W1, b1, W2, b2, epochs, alpha):
    loss_history = []

    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]:
```
epochs = 400
alpha = 0.03

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: 14.706488549023492
Epoch 50/400, Loss: 5.748323201440285
Epoch 100/400, Loss: 5.030013933619313
Epoch 150/400, Loss: 4.878225286090602
Epoch 200/400, Loss: 4.824956690483424
```

```
Epoch 250/400, Loss: 4.79124286104354
Epoch 300/400, Loss: 4.772109031222311
Epoch 350/400, Loss: 4.831335614819339
```

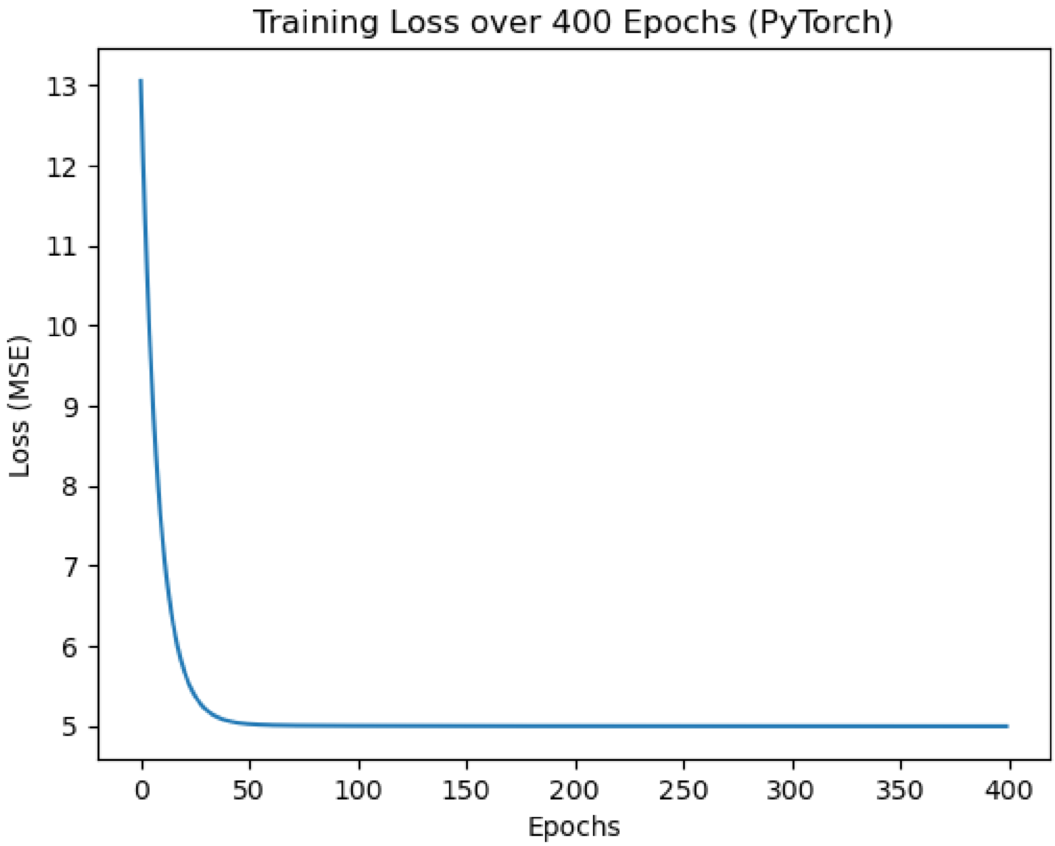



[9]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# 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__()
        self.hidden = nn.Linear(2, 2)
        self.relu = nn.ReLU()
        self.output = nn.Linear(2, 1)
```

```
    def forward(self, x):
        x = self.hidden(x)
        x = self.relu(x)
        x = self.output(x)
        return x
model = MLPModel()
criterion = nn.MSELoss()
learning_rate = 0.03
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

def train_model(model, X_tensor, Y_tensor, epochs, optimizer, criterion):
    loss_history = []

    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()
        loss.backward()

        # 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

epochs_400 = 400
loss_history_400 = train_model(model, X_tensor, Y_tensor, epochs_400,␣
 ↪optimizer, criterion)

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: 13.05589485168457
```

```
Epoch 50/400, Loss: 5.020622253417969
Epoch 100/400, Loss: 4.999037265777588
Epoch 150/400, Loss: 4.996543884277344
Epoch 200/400, Loss: 4.995215892791748
Epoch 250/400, Loss: 4.994339942932129
Epoch 300/400, Loss: 4.993725776672363
Epoch 350/400, Loss: 4.993305206298828
```



# ClassificationMLP

October 21, 2024

[1]:

```
import numpy as np
import torch

data = np.loadtxt('/home/darksst/Desktop/Fall24/StatisticalDecisionTheory/Data/
↪Semion/semeion.data')

X = data[:, :256]  # First 256 columns are the pixel features
Y = data[:, 256:]  # Last 10 columns represent the one-hot encoded digit labels

# Convert one-hot encoded labels to class labels (digits 0 to 9)
Y_labels = np.argmax(Y, axis=1)

# Function to split data into training and testing sets
def split_data(X, Y, test_size=0.2, random_state=None):
    np.random.seed(random_state)
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)  # Shuffle the indices

    test_size = int(test_size * X.shape[0])
    train_indices, test_indices = indices[:-test_size], indices[-test_size:]

    X_train, X_test = X[train_indices], X[test_indices]
    Y_train, Y_test = Y[train_indices], Y[test_indices]

    return X_train, X_test, Y_train, Y_test

X_train, X_test, Y_train, Y_test = split_data(X, Y_labels, test_size=0.2,␣
↪random_state=42)

# Convert NumPy arrays to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
Y_train_tensor = torch.tensor(Y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
Y_test_tensor = torch.tensor(Y_test, dtype=torch.long)
print("Training data shape:", X_train_tensor.shape)
print("Training labels shape:", Y_train_tensor.shape)
print("Test data shape:", X_test_tensor.shape)
```

```
print("Test labels shape:", Y_test_tensor.shape)
```

```
Training data shape: torch.Size([1275, 256])
Training labels shape: torch.Size([1275])
Test data shape: torch.Size([318, 256])
Test labels shape: torch.Size([318])
```

[2]:
```
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

def train_model(model, X_train, Y_train, criterion, optimizer, epochs):
    loss_history = []

    for epoch in range(epochs):
        model.train()

        # Forward pass: Compute predicted Y by passing X to the model
        outputs = model(X_train)

        # Compute the loss (cross-entropy loss)
        loss = criterion(outputs, Y_train)

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

        # Update parameters
        optimizer.step()

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

        # Print loss every 10 epochs
        if (epoch+1) % 100 == 0:
            print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item()}')

    return loss_history

def compute_accuracy(model, X_test, Y_test):
    model.eval()
    with torch.no_grad():
        outputs = model(X_test)
    _, predicted = torch.max(outputs, 1)
    correct = (predicted == Y_test).sum().item()
    total = Y_test.size(0)
    accuracy = correct / total * 100
```

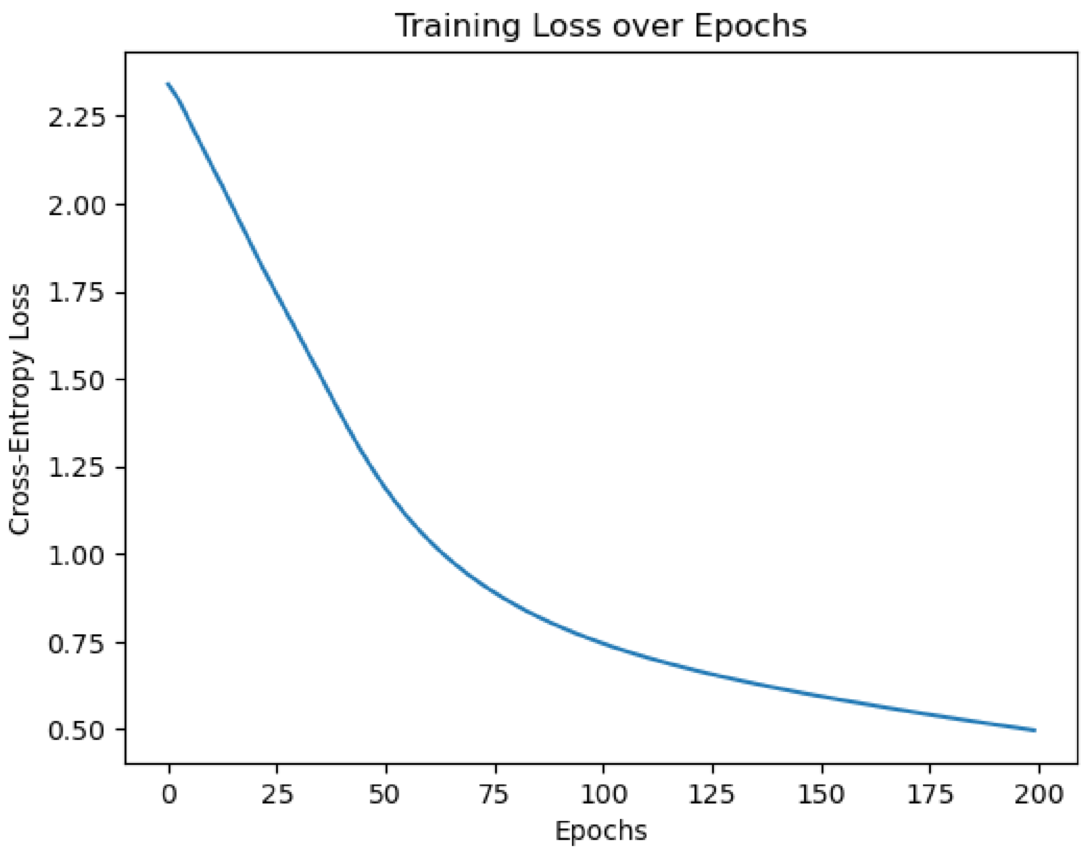
```
    return accuracy

def plot_loss(loss_history, epochs):
    plt.plot(range(epochs), loss_history)
    plt.xlabel('Epochs')
    plt.ylabel('Cross-Entropy Loss')
    plt.title('Training Loss over Epochs')
    plt.show()
```

[3]:

```
#MLP model with 5 neurons in the hidden layer
class MLPModel5(nn.Module):
    def __init__(self):
        super(MLPModel5, self).__init__()
        self.hidden = nn.Linear(256, 5)  # Input: 256 features, Hidden layer: 5␣
 ↪neurons
        self.relu = nn.ReLU()            # ReLU activation for the hidden layer
        self.output = nn.Linear(5, 10)   # Output layer: 10 neurons (for digits␣
 ↪0-9)

    def forward(self, x):
        # Forward pass
        x = self.hidden(x)
        x = self.relu(x)
        x = self.output(x)
        return x

model_5 = MLPModel5()

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_5.parameters(), lr=0.19)

# Set the number of epochs
epochs = 200

# Train the model with 5 neurons in the hidden layer
loss_history_5 = train_model(model_5, X_train_tensor, Y_train_tensor,␣
 ↪criterion, optimizer, epochs)

# Plot the loss
plot_loss(loss_history_5, epochs)

# Compute the accuracy on the test set
test_accuracy_5 = compute_accuracy(model_5, X_test_tensor, Y_test_tensor)
print(f'Accuracy on the test set: {test_accuracy_5:.2f}%')
```

```
Epoch 100/200, Loss: 0.7486768364906311
```

```
Epoch 200/200, Loss: 0.4971608817577362
```

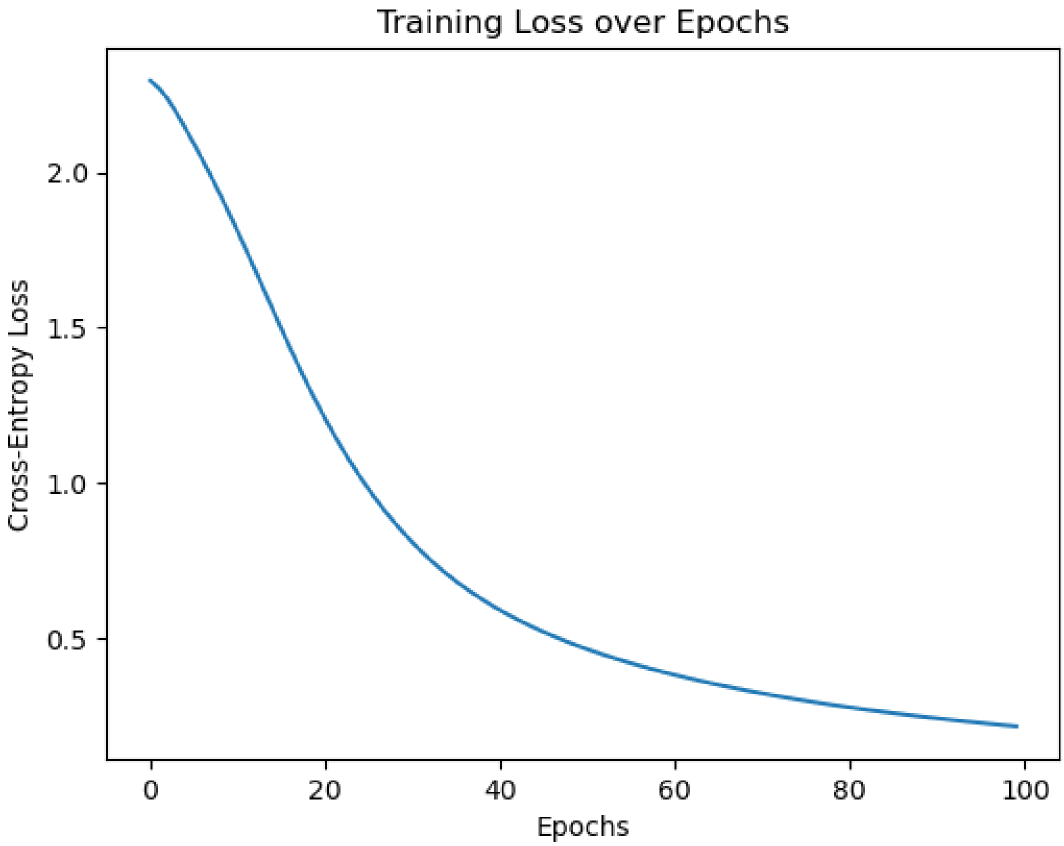



```
Accuracy on the test set: 80.50%
```

[4]:
```python
#MLP model with 10 neurons in the hidden layer
class MLPModel10(nn.Module):
    def __init__(self):
        super(MLPModel10, self).__init__()
        # Define layers
        self.hidden = nn.Linear(256, 10)  # Input: 256 features, Hidden layer:
 ↪10 neurons
        self.relu = nn.ReLU()             # ReLU activation for the hidden layer
        self.output = nn.Linear(10, 10)   # Output layer: 10 neurons (for
 ↪digits 0-9)

    def forward(self, x):
        # Forward pass
        x = self.hidden(x)
        x = self.relu(x)
        x = self.output(x)
```

```
        return x

def reinitialize_parameters(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            nn.init.uniform_(layer.weight, -1/np.sqrt(layer.in_features), 1/np.
 ↪sqrt(layer.in_features))
            nn.init.zeros_(layer.bias)

model_10 = MLPModel10()
reinitialize_parameters(model_10)

# Reinitialize the optimizer for the new model
optimizer = optim.SGD(model_10.parameters(), lr=0.26)

# Set the number of epochs
epochs = 100

# Train the model with 10 neurons in the hidden layer
loss_model2 = train_model(model_10, X_train_tensor, Y_train_tensor, criterion,␣
 ↪optimizer, epochs)

# Plot the loss
plot_loss(loss_model2, epochs)

# Compute the accuracy on the test set
model2_accuracy = compute_accuracy(model_10, X_test_tensor, Y_test_tensor)
print(f'Accuracy on the test set: {model2_accuracy:.2f}%')
```

```
Epoch 100/100, Loss: 0.21584920585155487
```

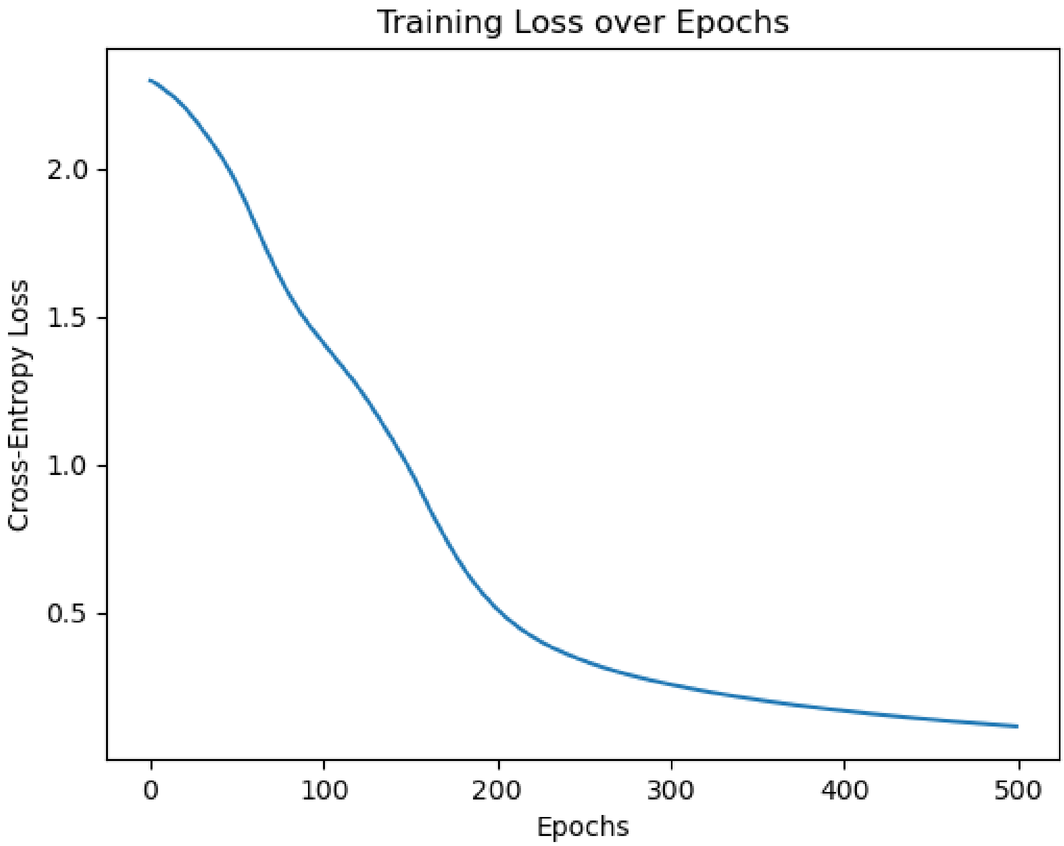



Accuracy on the test set: 90.88%

[19]:
```
# Define the MLP model with 2 hidden layers, each with 5 neurons
class MLPModelTwoHiddenLayers(nn.Module):
    def __init__(self):
        super(MLPModelTwoHiddenLayers, self).__init__()
        # Define layers
        self.hidden1 = nn.Linear(256, 5)  # First hidden layer: 5 neurons
        self.relu1 = nn.ReLU()            # ReLU activation for the first␣
 ↪hidden layer
        self.hidden2 = nn.Linear(5, 5)    # Second hidden layer: 5 neurons
        self.relu2 = nn.ReLU()            # ReLU activation for the second␣
 ↪hidden layer
        self.output = nn.Linear(5, 10)    # Output layer: 10 neurons (for␣
 ↪digits 0-9)

    def forward(self, x):
        # Forward pass
        x = self.hidden1(x)
        x = self.relu1(x)
```

```
        x = self.hidden2(x)
        x = self.relu2(x)
        x = self.output(x)
        return x
def reinitialize_parameters(model):
    for layer in model.children():
        if isinstance(layer, nn.Linear):
            nn.init.uniform_(layer.weight, -1/np.sqrt(layer.in_features), 1/np.
 ↪sqrt(layer.in_features))
            nn.init.zeros_(layer.bias)


model_three = MLPModelTwoHiddenLayers()
reinitialize_parameters(model_three)  # Reinitialize the parameters

# Reinitialize the optimizer for the new model
optimizer = optim.SGD(model_three.parameters(), lr=0.119)

# Set the number of epochs
epochs = 500

# Train the model with 2 hidden layers, each with 5 neurons
loss_model3 = train_model(model_three, X_train_tensor, Y_train_tensor,␣
 ↪criterion, optimizer, epochs)

# Plot the loss
plot_loss(loss_model3, epochs)

# Compute the accuracy on the test set
model3_accuracy = compute_accuracy(model_three, X_test_tensor, Y_test_tensor)
print(f'Accuracy on the test set: {model3_accuracy:.2f}%')
```

```
Epoch 100/500, Loss: 1.4164822101593018
Epoch 200/500, Loss: 0.5161546468734741
Epoch 300/500, Loss: 0.2569941282272339
Epoch 400/500, Loss: 0.1676710695028305
Epoch 500/500, Loss: 0.1146162897348404
```




```
Accuracy on the test set: 87.11%
```

[6]:
```
#MODEL with 5 neurons and 1 hidden layer converges at around 60 epochs
#MODEL with 10 neurons and 1 hidden layer converges at around 30 epochs
#MODEL with 5 neurons at 2 hidden layer converges at arouund 200 epochs
```