3. Assignment 3 part 1

File Edit View Insert Runtime Tools Help Cell Toolbox Widgets

Comment Share

Implement a convolution-free digit classifier using a neural network. You can use available modules in Scikit-Learn or TensorFlow to program the network.

Do not use convolution architectures and pre-trained models.

Generate new training images by flipping the white and black pixels. Then create the foreground which was originally black with one lesser white and the background which was originally white with one lesser black. Now prepare batches which have equal number of original and the generated inverted images.

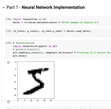
Plot the training curve on a graph step by step experiment.

Part 1 - Neural Network Implementation

```
[1]: import numpy as np
     mnist = np.load('mnist.npz')  # first download mnist of digits 0-9

[2]: x_train, y_train, x_test, y_test = mnist['x_train'], mnist['y_train'],
     mnist['x_test'], mnist['y_test']

[3]: # Visualization
     # Plot dataset
     n_to_show = 10
     indexes = np.random.choice(len(y_train), size=n_to_show)  # choosing n_to_show samples
     print(indexes)
     fig = plt.figure(figure_to_show, (n_to_show, 1))
     for i in range(n_to_show):
         ax = fig.add_subplot(1, n_to_show, i+1)
         ax.imshow(x_train[indexes[i]], cmap='Greys')
         ax.set_xticks([])
         ax.set_yticks([])
         ax.set_ylabel('%d' % y_train[indexes[i]])
         ax.set_xlabel('%d' % indexes[i])
     plt.show()
```




```
[4]: # Normalize
     x_train = x_train / 255.
     x_test = x_test / 255.
     print(x_train[0].shape, x_train[0].max(), x_train[0].min())
     print(x_test[0].shape, x_test[0].max(), x_test[0].min())
     # Flatten
     x_train = x_train.reshape(-1, 784)
     x_test = x_test.reshape(-1, 784)
     print(x_train[0].shape, x_train[0].max(), x_train[0].min())
     print(x_test[0].shape, x_test[0].max(), x_test[0].min())
     # Binarization
     x_train = (x_train > 0.5).astype(np.int)
     x_test = (x_test > 0.5).astype(np.int)
     print(x_train[0].shape, x_train[0].max(), x_train[0].min())
     print(x_test[0].shape, x_test[0].max(), x_test[0].min())
     # Check
     print('Train dataset shape', x_train.shape, y_train.shape)
     print('Test dataset shape', x_test.shape, y_test.shape)
     # Visualization
     # Plot dataset
     n_to_show = 10
     indexes = np.random.choice(len(y_train), size=n_to_show)  # choosing n_to_show samples
     print(indexes)
     fig = plt.figure(figure_to_show, (n_to_show, 1))
     for i in range(n_to_show):
         ax = fig.add_subplot(1, n_to_show, i+1)
         ax.imshow(x_train[indexes[i]], cmap='Greys')
         ax.set_xticks([])
         ax.set_yticks([])
         ax.set_ylabel('%d' % y_train[indexes[i]])
         ax.set_xlabel('%d' % indexes[i])
     plt.show()
```

```
(28, 28) 255.0 0.0
(28, 28) 255.0 0.0
(784,) 1.0 0.0
(784,) 1.0 0.0
(784,) 1 0
(784,) 1 0
Train dataset shape (60000, 784) (60000,)
Test dataset shape (10000, 784) (10000,)
[ 5 0 4 1 9 2 1 3 1 4]
```

```
[5]: def one_hot_encode(labels, num_classes):
         # Initialize the one-hot-encoded array
         one_hot = np.zeros((len(labels), num_classes))
         # Set the appropriate elements to 1
         for i, label in enumerate(labels):
             one_hot[i, label] = 1
         return one_hot
     # Test the function
     print("One-hot encoding for digits 0, 1, 2:")
     print(one_hot_encode(np.array([0, 1, 2]), 10))
```

```
One-hot encoding for digits 0, 1, 2:
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
```

```
[6]: # Apply one-hot encoding to the training and test labels
     y_train = one_hot_encode(y_train, 10)
     y_test = one_hot_encode(y_test, 10)
     # Print the shapes to verify
     print("Shape of y_train:", y_train.shape)
     print("Shape of y_test:", y_test.shape)
     # Print a few samples to see the output
     print("Sample of y_train:")
     print(y_train[:5])
     print("Sample of y_test:")
     print(y_test[:5])
```

```
Shape of y_train: (60000, 10)
Shape of y_test: (10000, 10)
Sample of y_train:
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
Sample of y_test:
[[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

```
[7]: # Define the activation function (sigmoid)
     def sigmoid(x):
         return 1 / (1 + np.exp(-x))
     # Define the derivative of the sigmoid function
     def sigmoid_derivative(x):
         return x * (1 - x)
     # Define the cross-entropy loss function
     def cross_entropy_loss(y_true, y_pred):
         epsilon = 1e-15  # Small value to avoid log(0)
         y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
         return -np.mean(y_true * np.log(y_pred))
     # Initialize weights and biases
     input_size = 784
     hidden_size = 128
     output_size = 10
     np.random.seed(42)
     W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
     b1 = np.random.uniform(-1, 1, (1, hidden_size))
     W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
     b2 = np.random.uniform(-1, 1, (1, output_size))
     # Forward pass
     def forward(x):
         global W1, b1, W2, b2
         z1 = np.dot(x, W1) + b1
         a1 = sigmoid(z1)
         z2 = np.dot(a1, W2) + b2
         a2 = sigmoid(z2)
         return a1, a2
     # Backward pass
     def backward(x, y, a1, a2):
         global W1, b1, W2, b2
         dz2 = a2 - y
         dW2 = np.dot(a1.T, dz2)
         db2 = np.sum(dz2, axis=0, keepdims=True)
         dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
         dW1 = np.dot(x.T, dz1)
         db1 = np.sum(dz1, axis=0, keepdims=True)
         return dW1, db1, dW2, db2
     # Training
     learning_rate = 0.1
     epochs = 5
     batch_size = 32
     for epoch in range(epochs):
         for i in range(0, len(x_train), batch_size):
             x_batch = x_train[i:i+batch_size]
             y_batch = y_train[i:i+batch_size]
             a1, a2 = forward(x_batch)
             dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
             W1 -= learning_rate * dW1
             b1 -= learning_rate * db1
             W2 -= learning_rate * dW2
             b2 -= learning_rate * db2
         a1, a2 = forward(x_train)
         loss = cross_entropy_loss(y_train, a2)
         print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
     # Evaluation
     a1, a2 = forward(x_test)
     predictions = np.argmax(a2, axis=1)
     true_labels = np.argmax(y_test, axis=1)
     accuracy = np.mean(predictions == true_labels)
     print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[8]: # Define the activation function (sigmoid)
     def sigmoid(x):
         return 1 / (1 + np.exp(-x))
     # Define the derivative of the sigmoid function
     def sigmoid_derivative(x):
         return x * (1 - x)
     # Define the cross-entropy loss function
     def cross_entropy_loss(y_true, y_pred):
         epsilon = 1e-15  # Small value to avoid log(0)
         y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
         return -np.mean(y_true * np.log(y_pred))
     # Initialize weights and biases
     input_size = 784
     hidden_size = 128
     output_size = 10
     np.random.seed(42)
     W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
     b1 = np.random.uniform(-1, 1, (1, hidden_size))
     W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
     b2 = np.random.uniform(-1, 1, (1, output_size))
     # Forward pass
     def forward(x):
         global W1, b1, W2, b2
         z1 = np.dot(x, W1) + b1
         a1 = sigmoid(z1)
         z2 = np.dot(a1, W2) + b2
         a2 = sigmoid(z2)
         return a1, a2
     # Backward pass
     def backward(x, y, a1, a2):
         global W1, b1, W2, b2
         dz2 = a2 - y
         dW2 = np.dot(a1.T, dz2)
         db2 = np.sum(dz2, axis=0, keepdims=True)
         dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
         dW1 = np.dot(x.T, dz1)
         db1 = np.sum(dz1, axis=0, keepdims=True)
         return dW1, db1, dW2, db2
     # Training
     learning_rate = 0.1
     epochs = 5
     batch_size = 32
     for epoch in range(epochs):
         for i in range(0, len(x_train), batch_size):
             x_batch = x_train[i:i+batch_size]
             y_batch = y_train[i:i+batch_size]
             a1, a2 = forward(x_batch)
             dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
             W1 -= learning_rate * dW1
             b1 -= learning_rate * db1
             W2 -= learning_rate * dW2
             b2 -= learning_rate * db2
         a1, a2 = forward(x_train)
         loss = cross_entropy_loss(y_train, a2)
         print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
     # Evaluation
     a1, a2 = forward(x_test)
     predictions = np.argmax(a2, axis=1)
     true_labels = np.argmax(y_test, axis=1)
     accuracy = np.mean(predictions == true_labels)
     print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[9]: # Define the activation function (sigmoid)
     def sigmoid(x):
         return 1 / (1 + np.exp(-x))
     # Define the derivative of the sigmoid function
     def sigmoid_derivative(x):
         return x * (1 - x)
     # Define the cross-entropy loss function
     def cross_entropy_loss(y_true, y_pred):
         epsilon = 1e-15  # Small value to avoid log(0)
         y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
         return -np.mean(y_true * np.log(y_pred))
     # Initialize weights and biases
     input_size = 784
     hidden_size = 128
     output_size = 10
     np.random.seed(42)
     W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
     b1 = np.random.uniform(-1, 1, (1, hidden_size))
     W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
     b2 = np.random.uniform(-1, 1, (1, output_size))
     # Forward pass
     def forward(x):
         global W1, b1, W2, b2
         z1 = np.dot(x, W1) + b1
         a1 = sigmoid(z1)
         z2 = np.dot(a1, W2) + b2
         a2 = sigmoid(z2)
         return a1, a2
     # Backward pass
     def backward(x, y, a1, a2):
         global W1, b1, W2, b2
         dz2 = a2 - y
         dW2 = np.dot(a1.T, dz2)
         db2 = np.sum(dz2, axis=0, keepdims=True)
         dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
         dW1 = np.dot(x.T, dz1)
         db1 = np.sum(dz1, axis=0, keepdims=True)
         return dW1, db1, dW2, db2
     # Training
     learning_rate = 0.1
     epochs = 5
     batch_size = 32
     for epoch in range(epochs):
         for i in range(0, len(x_train), batch_size):
             x_batch = x_train[i:i+batch_size]
             y_batch = y_train[i:i+batch_size]
             a1, a2 = forward(x_batch)
             dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
             W1 -= learning_rate * dW1
             b1 -= learning_rate * db1
             W2 -= learning_rate * dW2
             b2 -= learning_rate * db2
         a1, a2 = forward(x_train)
         loss = cross_entropy_loss(y_train, a2)
         print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
     # Evaluation
     a1, a2 = forward(x_test)
     predictions = np.argmax(a2, axis=1)
     true_labels = np.argmax(y_test, axis=1)
     accuracy = np.mean(predictions == true_labels)
     print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[10]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1, W2) + b2
          a2 = sigmoid(z2)
          return a1, a2
      # Backward pass
      def backward(x, y, a1, a2):
          global W1, b1, W2, b2
          dz2 = a2 - y
          dW2 = np.dot(a1.T, dz2)
          db2 = np.sum(dz2, axis=0, keepdims=True)
          dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
          dW1 = np.dot(x.T, dz1)
          db1 = np.sum(dz1, axis=0, keepdims=True)
          return dW1, db1, dW2, db2
      # Training
      learning_rate = 0.1
      epochs = 5
      batch_size = 32
      for epoch in range(epochs):
          for i in range(0, len(x_train), batch_size):
              x_batch = x_train[i:i+batch_size]
              y_batch = y_train[i:i+batch_size]
              a1, a2 = forward(x_batch)
              dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
              W1 -= learning_rate * dW1
              b1 -= learning_rate * db1
              W2 -= learning_rate * dW2
              b2 -= learning_rate * db2
          a1, a2 = forward(x_train)
          loss = cross_entropy_loss(y_train, a2)
          print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
      # Evaluation
      a1, a2 = forward(x_test)
      predictions = np.argmax(a2, axis=1)
      true_labels = np.argmax(y_test, axis=1)
      accuracy = np.mean(predictions == true_labels)
      print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[11]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1, W2) + b2
          a2 = sigmoid(z2)
          return a1, a2
      # Backward pass
      def backward(x, y, a1, a2):
          global W1, b1, W2, b2
          dz2 = a2 - y
          dW2 = np.dot(a1.T, dz2)
          db2 = np.sum(dz2, axis=0, keepdims=True)
          dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
          dW1 = np.dot(x.T, dz1)
          db1 = np.sum(dz1, axis=0, keepdims=True)
          return dW1, db1, dW2, db2
      # Training
      learning_rate = 0.1
      epochs = 5
      batch_size = 32
      for epoch in range(epochs):
          for i in range(0, len(x_train), batch_size):
              x_batch = x_train[i:i+batch_size]
              y_batch = y_train[i:i+batch_size]
              a1, a2 = forward(x_batch)
              dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
              W1 -= learning_rate * dW1
              b1 -= learning_rate * db1
              W2 -= learning_rate * dW2
              b2 -= learning_rate * db2
          a1, a2 = forward(x_train)
          loss = cross_entropy_loss(y_train, a2)
          print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
      # Evaluation
      a1, a2 = forward(x_test)
      predictions = np.argmax(a2, axis=1)
      true_labels = np.argmax(y_test, axis=1)
      accuracy = np.mean(predictions == true_labels)
      print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[12]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1, W2) + b2
          a2 = sigmoid(z2)
          return a1, a2
      # Backward pass
      def backward(x, y, a1, a2):
          global W1, b1, W2, b2
          dz2 = a2 - y
          dW2 = np.dot(a1.T, dz2)
          db2 = np.sum(dz2, axis=0, keepdims=True)
          dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
          dW1 = np.dot(x.T, dz1)
          db1 = np.sum(dz1, axis=0, keepdims=True)
          return dW1, db1, dW2, db2
      # Training
      learning_rate = 0.1
      epochs = 5
      batch_size = 32
      for epoch in range(epochs):
          for i in range(0, len(x_train), batch_size):
              x_batch = x_train[i:i+batch_size]
              y_batch = y_train[i:i+batch_size]
              a1, a2 = forward(x_batch)
              dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
              W1 -= learning_rate * dW1
              b1 -= learning_rate * db1
              W2 -= learning_rate * dW2
              b2 -= learning_rate * db2
          a1, a2 = forward(x_train)
          loss = cross_entropy_loss(y_train, a2)
          print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
      # Evaluation
      a1, a2 = forward(x_test)
      predictions = np.argmax(a2, axis=1)
      true_labels = np.argmax(y_test, axis=1)
      accuracy = np.mean(predictions == true_labels)
      print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[13]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1, W2) + b2
          a2 = sigmoid(z2)
          return a1, a2
      # Backward pass
      def backward(x, y, a1, a2):
          global W1, b1, W2, b2
          dz2 = a2 - y
          dW2 = np.dot(a1.T, dz2)
          db2 = np.sum(dz2, axis=0, keepdims=True)
          dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
          dW1 = np.dot(x.T, dz1)
          db1 = np.sum(dz1, axis=0, keepdims=True)
          return dW1, db1, dW2, db2
      # Training
      learning_rate = 0.1
      epochs = 5
      batch_size = 32
      for epoch in range(epochs):
          for i in range(0, len(x_train), batch_size):
              x_batch = x_train[i:i+batch_size]
              y_batch = y_train[i:i+batch_size]
              a1, a2 = forward(x_batch)
              dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
              W1 -= learning_rate * dW1
              b1 -= learning_rate * db1
              W2 -= learning_rate * dW2
              b2 -= learning_rate * db2
          a1, a2 = forward(x_train)
          loss = cross_entropy_loss(y_train, a2)
          print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
      # Evaluation
      a1, a2 = forward(x_test)
      predictions = np.argmax(a2, axis=1)
      true_labels = np.argmax(y_test, axis=1)
      accuracy = np.mean(predictions == true_labels)
      print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[14]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1, W2) + b2
          a2 = sigmoid(z2)
          return a1, a2
      # Backward pass
      def backward(x, y, a1, a2):
          global W1, b1, W2, b2
          dz2 = a2 - y
          dW2 = np.dot(a1.T, dz2)
          db2 = np.sum(dz2, axis=0, keepdims=True)
          dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
          dW1 = np.dot(x.T, dz1)
          db1 = np.sum(dz1, axis=0, keepdims=True)
          return dW1, db1, dW2, db2
      # Training
      learning_rate = 0.1
      epochs = 5
      batch_size = 32
      for epoch in range(epochs):
          for i in range(0, len(x_train), batch_size):
              x_batch = x_train[i:i+batch_size]
              y_batch = y_train[i:i+batch_size]
              a1, a2 = forward(x_batch)
              dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
              W1 -= learning_rate * dW1
              b1 -= learning_rate * db1
              W2 -= learning_rate * dW2
              b2 -= learning_rate * db2
          a1, a2 = forward(x_train)
          loss = cross_entropy_loss(y_train, a2)
          print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
      # Evaluation
      a1, a2 = forward(x_test)
      predictions = np.argmax(a2, axis=1)
      true_labels = np.argmax(y_test, axis=1)
      accuracy = np.mean(predictions == true_labels)
      print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[15]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1, W2) + b2
          a2 = sigmoid(z2)
          return a1, a2
      # Backward pass
      def backward(x, y, a1, a2):
          global W1, b1, W2, b2
          dz2 = a2 - y
          dW2 = np.dot(a1.T, dz2)
          db2 = np.sum(dz2, axis=0, keepdims=True)
          dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
          dW1 = np.dot(x.T, dz1)
          db1 = np.sum(dz1, axis=0, keepdims=True)
          return dW1, db1, dW2, db2
      # Training
      learning_rate = 0.1
      epochs = 5
      batch_size = 32
      for epoch in range(epochs):
          for i in range(0, len(x_train), batch_size):
              x_batch = x_train[i:i+batch_size]
              y_batch = y_train[i:i+batch_size]
              a1, a2 = forward(x_batch)
              dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
              W1 -= learning_rate * dW1
              b1 -= learning_rate * db1
              W2 -= learning_rate * dW2
              b2 -= learning_rate * db2
          a1, a2 = forward(x_train)
          loss = cross_entropy_loss(y_train, a2)
          print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
      # Evaluation
      a1, a2 = forward(x_test)
      predictions = np.argmax(a2, axis=1)
      true_labels = np.argmax(y_test, axis=1)
      accuracy = np.mean(predictions == true_labels)
      print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

Part 2 - New Images by inverting black and white pixels

```
[16]: # Flipping the colors (0 to 1 and 1 to 0)
      x_train = 1 - x_train  # Inverting the pixels
      x_test = 1 - x_test
      print(x_train[0].shape, x_train[0].max(), x_train[0].min())
      print(x_test[0].shape, x_test[0].max(), x_test[0].min())
      # Check
      print('Train dataset shape', x_train.shape, y_train.shape)
      print('Test dataset shape', x_test.shape, y_test.shape)
      # Visualization
      # Plot dataset
      n_to_show = 10
      indexes = np.random.choice(len(y_train), size=n_to_show)  # choosing n_to_show samples
      print(indexes)
      fig = plt.figure(figure_to_show, (n_to_show, 1))
      for i in range(n_to_show):
          ax = fig.add_subplot(1, n_to_show, i+1)
          ax.imshow(x_train[indexes[i]], cmap='Greys')
          ax.set_xticks([])
          ax.set_yticks([])
          ax.set_ylabel('%d' % y_train[indexes[i]])
          ax.set_xlabel('%d' % indexes[i])
      plt.show()
```

```
(784,) 1 0
(784,) 1 0
Train dataset shape (60000, 784) (60000,)
Test dataset shape (10000, 784) (10000,)
[ 5 0 4 1 9 2 1 3 1 4]
```

```
[17]: # Apply one-hot encoding
      y_train = one_hot_encode(y_train, 10)
      y_test = one_hot_encode(y_test, 10)
      # Print the shapes to verify
      print("Shape of y_train:", y_train.shape)
      print("Shape of y_test:", y_test.shape)
      # Print a few samples to see the output
      print("Sample of y_train:")
      print(y_train[:5])
      print("Sample of y_test:")
      print(y_test[:5])
```

```
Shape of y_train: (60000, 10)
Shape of y_test: (10000, 10)
Sample of y_train:
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
Sample of y_test:
[[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

```
[18]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1, W2) + b2
          a2 = sigmoid(z2)
          return a1, a2
      # Backward pass
      def backward(x, y, a1, a2):
          global W1, b1, W2, b2
          dz2 = a2 - y
          dW2 = np.dot(a1.T, dz2)
          db2 = np.sum(dz2, axis=0, keepdims=True)
          dz1 = np.dot(dz2, W2.T) * sigmoid_derivative(a1)
          dW1 = np.dot(x.T, dz1)
          db1 = np.sum(dz1, axis=0, keepdims=True)
          return dW1, db1, dW2, db2
      # Training
      learning_rate = 0.1
      epochs = 5
      batch_size = 32
      for epoch in range(epochs):
          for i in range(0, len(x_train), batch_size):
              x_batch = x_train[i:i+batch_size]
              y_batch = y_train[i:i+batch_size]
              a1, a2 = forward(x_batch)
              dW1, db1, dW2, db2 = backward(x_batch, y_batch, a1, a2)
              W1 -= learning_rate * dW1
              b1 -= learning_rate * db1
              W2 -= learning_rate * dW2
              b2 -= learning_rate * db2
          a1, a2 = forward(x_train)
          loss = cross_entropy_loss(y_train, a2)
          print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
      # Evaluation
      a1, a2 = forward(x_test)
      predictions = np.argmax(a2, axis=1)
      true_labels = np.argmax(y_test, axis=1)
      accuracy = np.mean(predictions == true_labels)
      print(f"Test accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 0.2000
Epoch 2, Loss: 0.2000
Epoch 3, Loss: 0.2000
Epoch 4, Loss: 0.2000
Epoch 5, Loss: 0.2000
Test accuracy: 0.1135
```

```
[19]: # Define the activation function (sigmoid)
      def sigmoid(x):
          return 1 / (1 + np.exp(-x))
      # Define the derivative of the sigmoid function
      def sigmoid_derivative(x):
          return x * (1 - x)
      # Define the cross-entropy loss function
      def cross_entropy_loss(y_true, y_pred):
          epsilon = 1e-15  # Small value to avoid log(0)
          y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
          return -np.mean(y_true * np.log(y_pred))
      # Initialize weights and biases
      input_size = 784
      hidden_size = 128
      output_size = 10
      np.random.seed(42)
      W1 = np.random.uniform(-1, 1, (input_size, hidden_size))
      b1 = np.random.uniform(-1, 1, (1, hidden_size))
      W2 = np.random.uniform(-1, 1, (hidden_size, output_size))
      b2 = np.random.uniform(-1, 1, (1, output_size))
      # Forward pass
      def forward(x):
          global W1, b1, W2, b2
          z1 = np.dot(x, W1) + b1
          a1 = sigmoid(z1)
          z2 = np.dot(a1
```