Problem 3.1: Write down your loss function and describe how it compares to MSE

Loss function: CCE (Categorical Cross-Entropy)

$$L_{\text{CCE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log\left(\hat{y}_{ij}\right)$$

Categorical Cross-Entropy measures how well the predicted probability distribution matches the true distribution. This function is made for categorical classification and is thus better for our MNIST database as it will harshly punish wrong classifications.

Problem 3.2: Compute the gradient of your loss function. Then, describe what needs to change in your approach to implement this new gradient.

Gradient:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

We need to change how we calculate the gradient in our backprop, so `D2 = (-2 / y.shape[0] * (y_onehot - p) * p * (1 - p))` becomes `D2 = (p - y_onehot) / y.shape[0]`.

Problem 3.3: Modify the multilayer perceptron code to implement the new loss function and its gradient for training.

```
class MultiLayerPerceptronMSE:

    def __init__(self, num_features, num_hidden, num_classes,
random_seed=123):

        self.num_classes = num_classes

        rng = np.random.RandomState(random_seed)

        # first layer weights and biases
        self.weight_h = rng.normal(loc=0.0, scale=0.1,
size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # second layer weights and biases
        self.weight_out = rng.normal(loc=0.0, scale=0.1,
size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)

    def int_to_onehot(self, y, num_labels):
```

```python
        ary = np.zeros((y.shape[0], num_labels))
        for i, val in enumerate(y):
            ary[i, val] = 1
        return(ary)

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

    def forward(self, x):
        # enter code here to implement the forward method
        zeta = np.dot(x, self.weight_h.T) + self.bias_h
        h = self.sigmoid(zeta)

        z = np.dot(h, self.weight_out.T) + self.bias_out
        p = self.sigmoid(z)

        return(h, p)

    def predict(self, X):
        _, probs = self.forward(X)
        return np.argmax(probs, axis=1)

    def backward(self, x, h, p, y):

        # encode the labels with one-hot encoding
        y_onehot = self.int_to_onehot(y, self.num_classes)

        # enter code here to implement the backward method
        D2 = (-2 / y.shape[0] * (y_onehot - p) * p * (1 - p))
        d_L__d_w2 = np.dot(D2.T, h)
        d_L__d_b2 = np.dot(D2.T, np.ones_like(y))

        D1 = np.dot(D2, self.weight_out) * h * (1 - h)
        d_L__d_w1 = np.dot(D1.T, x)
        d_L__d_b1 = np.dot(D1.T, np.ones_like(y))

        return(d_L__d_w2, d_L__d_b2, d_L__d_w1, d_L__d_b1)

    def mse_loss(targets, probs, num_labels=10):
        onehot_targets = int_to_onehot(targets, num_labels)
        err = np.mean((onehot_targets - probs)**2)
        return(err)

    def compute_mse_and_acc(self, X, y, num_labels=10):
        _, probs = self.forward(X)
        predicted_labels = np.argmax(probs,axis=1)
        onehot_targets =self.int_to_onehot(y, num_labels)
        loss = np.mean((onehot_targets - probs)**2)
        acc = np.sum(predicted_labels == y)/len(y)
        return(loss, acc)
```

```
    def train(self,X_train, Y_train, X_test, Y_test, num_epochs,
learning_rate=0.1):
        train_losses = []
        test_losses = []
        train_accs = []
        test_accs = []
        for e in range(num_epochs):

            # compute the forward method
            h, p = self.forward(X_train)

            # compute the backward method
            d_L__d_w2, d_L__d_b2, d_L__d_w1, d_L__d_b1 =
self.backward(X_train, h, p, Y_train)

            # update the weights and the biases
            self.weight_out -= learning_rate * d_L__d_w2
            self.bias_out   -= learning_rate * d_L__d_b2
            self.weight_h   -= learning_rate * d_L__d_w1
            self.bias_h     -= learning_rate * d_L__d_b1

            train_loss, train_acc = self.compute_mse_and_acc(X_train,
Y_train)
            train_losses.append(train_loss)
            train_accs.append(train_acc)

            test_loss, test_acc = self.compute_mse_and_acc(X_test,
Y_test)
            test_losses.append(test_loss)
            test_accs.append(test_acc)
        return(train_losses, train_accs, test_losses, test_accs)

class MultiLayerPerceptronCCE:

    def __init__(self, num_features, num_hidden, num_classes,
random_seed=123):
        self.num_classes = num_classes
        rng = np.random.RandomState(random_seed)

        # first layer weights and biases
        self.weight_h = rng.normal(loc=0.0, scale=0.1,
size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # second layer weights and biases
        self.weight_out = rng.normal(loc=0.0, scale=0.1,
size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)
```

```python
    def int_to_onehot(self, y, num_labels):
        ary = np.zeros((y.shape[0], num_labels))
        for i, val in enumerate(y):
            ary[i, val] = 1
        return ary

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

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  #
numerical stability
        return exp_z / np.sum(exp_z, axis=1, keepdims=True)

    def forward(self, x):
        zeta = np.dot(x, self.weight_h.T) + self.bias_h
        h = self.sigmoid(zeta)
        z = np.dot(h, self.weight_out.T) + self.bias_out
        p = self.softmax(z)
        return h, p

    def predict(self, X):
        _, probs = self.forward(X)
        return np.argmax(probs, axis=1)

    def cross_entropy_loss(self, y, p):
        eps = 1e-9
        log_likelihood = -np.log(p[range(len(y)), y] + eps)
        return np.mean(log_likelihood)

    def backward(self, x, h, p, y):
        # one-hot encode labels
        y_onehot = self.int_to_onehot(y, self.num_classes)

        D2 = (p - y_onehot) / y.shape[0]

        d_L__d_w2 = np.dot(D2.T, h)
        d_L__d_b2 = np.sum(D2.T, axis=1)

        D1 = np.dot(D2, self.weight_out) * h * (1 - h)
        d_L__d_w1 = np.dot(D1.T, x)
        d_L__d_b1 = np.sum(D1.T, axis=1)

        return d_L__d_w2, d_L__d_b2, d_L__d_w1, d_L__d_b1

    def compute_loss_and_acc(self, X, y):
        _, probs = self.forward(X)
        loss = self.cross_entropy_loss(y, probs)
        preds = np.argmax(probs, axis=1)
        acc = np.mean(preds == y)
```

```
        return loss, acc

    def train(self, X_train, Y_train, X_test, Y_test, num_epochs,
learning_rate=0.1):
        train_losses, test_losses, train_accs, test_accs = [], [], [],
[]

        for e in range(num_epochs):
            # forward pass
            h, p = self.forward(X_train)

            # backward pass
            d_L__d_w2, d_L__d_b2, d_L__d_w1, d_L__d_b1 =
self.backward(X_train, h, p, Y_train)

            # parameter updates
            self.weight_out -= learning_rate * d_L__d_w2
            self.bias_out   -= learning_rate * d_L__d_b2
            self.weight_h   -= learning_rate * d_L__d_w1
            self.bias_h     -= learning_rate * d_L__d_b1

            # record train/test metrics
            train_loss, train_acc = self.compute_loss_and_acc(X_train,
Y_train)
            test_loss, test_acc   = self.compute_loss_and_acc(X_test,
Y_test)

            train_losses.append(train_loss)
            test_losses.append(test_loss)
            train_accs.append(train_acc)
            test_accs.append(test_acc)

        return train_losses, train_accs, test_losses, test_accs
```

Problem 3.4: Train your model on the MNIST dataset

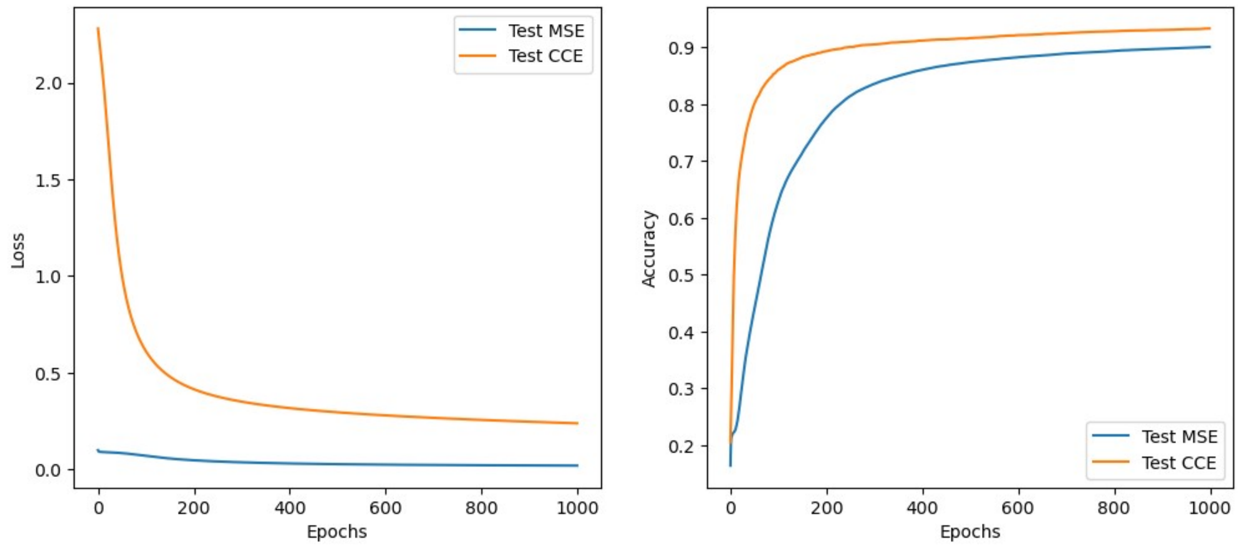
```
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import struct

def read_mnist_images(subset='train'):
    if subset=='train':
        prefix = 'train-'
    else:
        prefix = 't10k-'

    with open(os.path.join('MNIST',prefix+'images.idx3-ubyte'), 'rb')
as f:
```

```
        # unpack header
        _, num_images, num_rows, num_cols = struct.unpack('>IIII', 
f.read(16))

        # read image data
        image_data = f.read(num_images * num_rows * num_cols)
        images = np.frombuffer(image_data, dtype=np.uint8)
        images = images.reshape(num_images, num_rows, num_cols)

    with open(os.path.join('MNIST',prefix+'labels.idx1-ubyte'), 'rb') 
as f:
        # unpack header
        _, num_labels = struct.unpack('>II', f.read(8))

        # read label data
        labels = np.frombuffer(f.read(), dtype=np.uint8)


    return images, labels

train_images, train_labels = read_mnist_images(subset='train')
test_images, test_labels = read_mnist_images(subset='test')

X_train = train_images.reshape(-1, 784) / 255.0
X_test = test_images.reshape(-1, 784) / 255.0
Y_train = train_labels
Y_test = test_labels

mse = MultiLayerPerceptronMSE(784, 50, 10)
train_losses_mse, train_accs_mse, test_losses_mse, test_accs_mse = 
mse.train(X_train, Y_train, X_test, Y_test, 1000, learning_rate = 0.5)

cce = MultiLayerPerceptronCCE(784, 50, 10)
train_losses_cce, train_accs_cce, test_losses_cce, test_accs_cce = 
cce.train(X_train, Y_train, X_test, Y_test, 1000, learning_rate = 0.5)
```

Problem 3.5: Compare and contrast your model results with the model results using MSE. Make plots to compare losses and accuracies. Describe what has changed and what has remained the same.

```
plt.figure(figsize=(12,5))

plt.subplot(1,2,1)
plt.plot(train_losses_mse, label='Test MSE')
plt.plot(test_losses_cce, label='Test CCE')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend()
```

```
plt.subplot(1,2,2)
plt.plot(train_accs_mse, label='Test MSE')
plt.plot(test_accs_cce, label='Test CCE')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```



The CCE model starts and ends with a lot more loss. It does have a pretty sharp convergence, but it's hard to tell if it's relatively steeper than the MSE convergences because the MSE's absolute numbers were so low to begin with.

The accuracy does show a lot more improvement, however, showing that CCE's more sensitive categorical method reuslts in better classification at every stage. CCE also doesn't have the steep convergence at the very beginning followed by the more smooth curve, it's just a smooth curve all the way.

Overall, I'd prefer CCE for this, it is clear that if I continue to train it will continue to have greater accuracy (but they do begin to get close so I assume they'll both converge around the same point).