CS178 Homework 4

Instructions

This homework (and subsequent ones) will involve data analysis and reporting on methods and results using Python code. You will submit a **single PDF file** that contains everything to Gradescope. This includes any text you wish to include to describe your results, the complete code snippets of how you attempted each problem, any figures that were generated, and scans of any work on paper that you wish to include. It is important that you include enough detail that we know how you solved the problem, since otherwise we will be unable to grade it.

Your homeworks will be given to you as Jupyter notebooks containing the problem descriptions and some template code that will help you get started. You are encouraged to use these starter Jupyter notebooks to complete your assignment and to write your report. This will help you not only ensure that all of the code for the solutions is included, but also will provide an easy way to export your results to a PDF file (for example, doing *print preview* and *printing to pdf*). I recommend liberal use of Markdown cells to create headers for each problem and sub-problem, explaining your implementation/answers, and including any mathematical equations. For parts of the homework you do on paper, scan it in such that it is legible (there are a number of free Android/iOS scanning apps, if you do not have access to a scanner), and include it as an image in the Jupyter notebook.

Double check that all of your answers are legible on Gradescope, e.g. make sure any text you have written does not get cut off.

If you have any questions/concerns about using Jupyter notebooks, ask us on EdD. If you decide not to use Jupyter notebooks, but go with Microsoft Word or LaTeX to create your PDF file, make sure that all of the answers can be generated from the code snippets included in the document.

Summary of Assignment: 100 total points

- Problem 1: A Small Neural Network (30 points)
  - Problem 1.1: Forward Pass (10 points)
  - Problem 1.2: Evaluate Loss (10 points)
  - Problem 1.3: Network Size (10 points)
- Problem 2: Neural Networks on MNIST (35 points)
  - Problem 2.1: Varying the Amount of Training Data (15 points)
  - Problem 2.3: Optimization Curves (10 points)
  - Problem 2.3: Tuning your Neural Network (10 points)
- Problem 3: Convolutional Networks (30 points)
  - Problem 3.1: Model structure (10 points)
  - Problem 3.2: Training (10 points)
  - Problem 3.3: Evaluation (5 points)
  - Problem 3.4: Comparing predictions (5 points)
- Statement of Collaboration (5 points)

Before we get started, let's import some libraries that you will make use of in this assignment. Make sure that you run the code cell below in order to import these libraries.

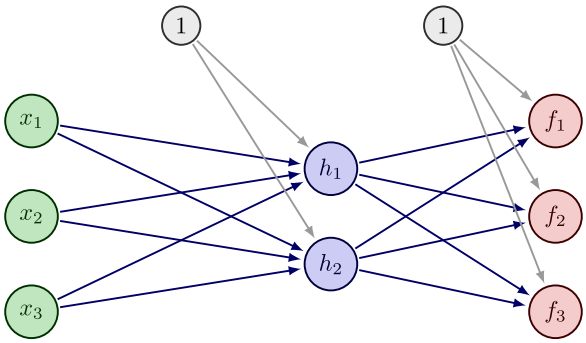
Important: In the code block below, we set `seed=1234`. This is to ensure your code has reproducible results and is important for grading. Do not change this. If you are not using the provided Jupyter notebook, make sure to also set the random seed as below.

Important: Do not change any codes we give you below, except for those waiting for you to complete. This is to ensure your code has reproducible results and is important for grading.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

import torch

from IPython import display
import pandas as pd
from sklearn.datasets import fetch_openml             # common data set access
from sklearn.preprocessing import StandardScaler      # scaling transform
from sklearn.model_selection import train_test_split # validation tools
from sklearn.metrics import accuracy_score

from sklearn.neural_network import MLPClassifier      # scikit's MLP

import warnings
warnings.filterwarnings('ignore')

# Fix the random seed for reproducibility
# !! Important !! : do not change this
seed = 1234
np.random.seed(seed)
torch.manual_seed(seed);
```

Problem 1: A Small Neural Network

Consider the small neural network given in the image below, which will classify a 3-dimensional feature vector $\mathbf{x}$ into one of three classes ($y = 0, 1, 2$):



You are given an input to this network $\mathbf{x}$,

$$\mathbf{x} = [\, x_1 \quad x_2 \quad x_3 \,] = [\, 1 \quad 3 \quad -2 \,]$$

as well as weights W for the hidden layer and weights B for the output layer.

$$W = \begin{bmatrix} w_{01} & w_{11} & w_{21} & w_{31} \\ w_{02} & w_{12} & w_{22} & w_{32} \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 5 \\ 2 & 1 & 1 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} \beta_{01} & \beta_{11} & \beta_{21} \\ \beta_{02} & \beta_{12} & \beta_{22} \\ \beta_{03} & \beta_{13} & \beta_{23} \end{bmatrix} = \begin{bmatrix} 4 & -1 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \end{bmatrix}$$

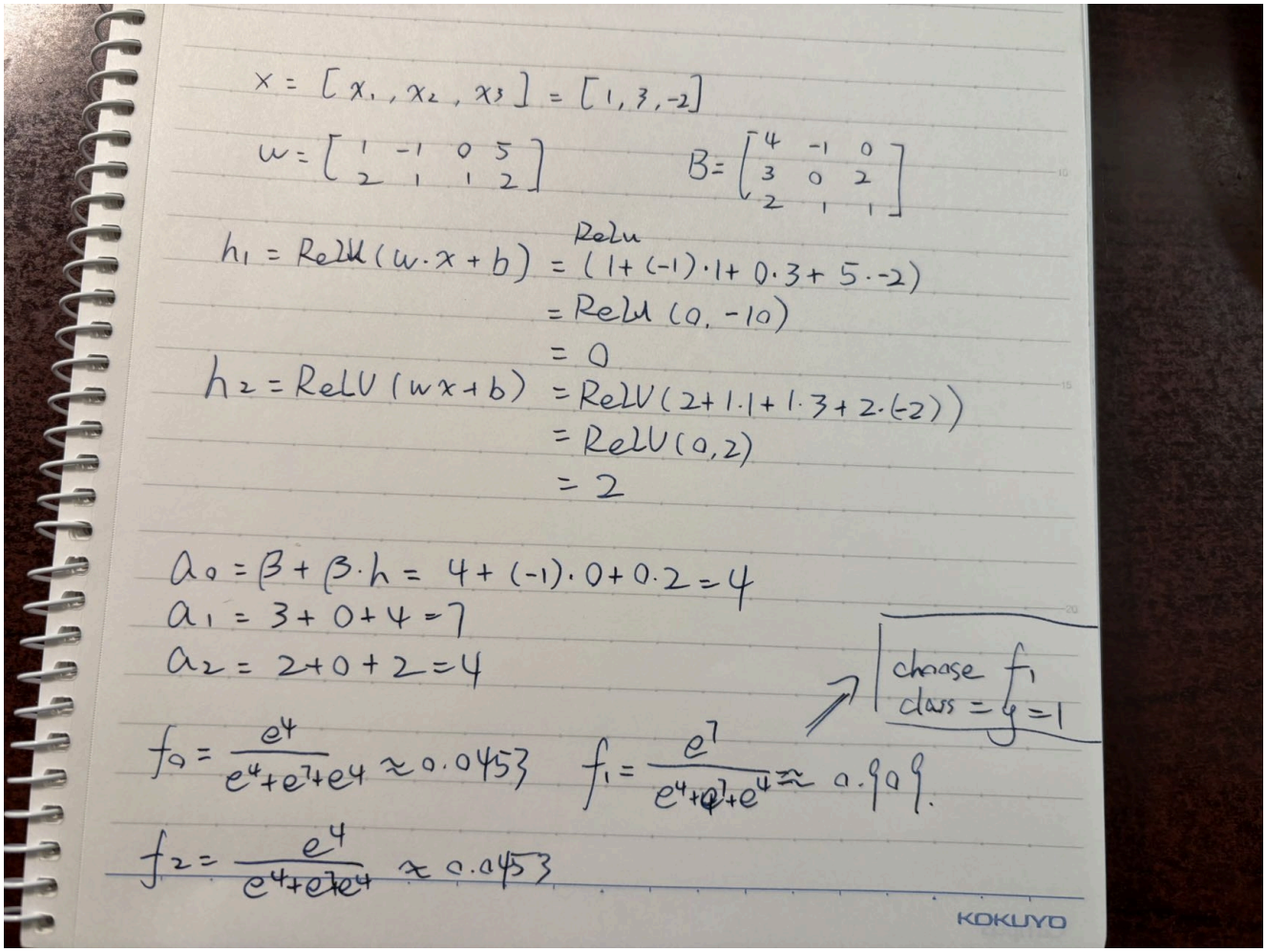
For example, w_{12} is the weight connecting input x_1 to hidden node h_2; w_{01} is the constant (bias) term for h_1, etc.

This network uses the ReLU activation function for the hidden layer, and uses the softmax activation function for the output layer.

Answer the following questions about this network.

Problem 1.1 (10 points): Forward Pass

- Given the inputs and weights above, compute the values of the hidden units h_1, h_2 and the outputs f_0, f_1, f_2. You should do this by hand, i.e. you should not write any code to do the calculation, but feel free to use a calculator to help you do the computations.
- You can optionally use $\LaTeX$ in your answer on the Jupyter notebook. Otherwise, write your answer on paper and include a picture of your answer in this notebook. In order to include an image in Jupyter notebook, save the image in the same directory as the .ipynb file and then write `![caption](image.png)`. Alternatively, you may go to Edit --> Insert Image at the top menu to insert an image into a Markdown cell. **Double check that your image is visible in your PDF submission.**
- What class would the network predict for the input $\mathbf{x}$?



In [9]:
```
print("The network predict for the input x as class y=1)")
```

```
The network predict for the input x as class y=1)
```

Problem 1.2 (10 points): Evaluate Loss

Typically when we train neural networks for classification, we seek to minimize the log-loss function. Note that the output of the log-loss function is always nonnegative (≥ 0), but can be arbitrarily large (you should pause for a second and make sure you understand why this is true).

- Suppose the true label for the input $\mathbf{x}$ is $y = 1$. What would be the value of our loss function based on the network's prediction for $\mathbf{x}$?

- Suppose instead that the true label for the input $\mathbf{x}$ is $y = 2$. What would be the value of our loss function based on the network's prediction for $\mathbf{x}$?

You are free to use numpy / Python to help you calculate this, but don't use any neural network libraries that will automatically calculate the loss for you.

In [3]:
```
e4 = np.exp(4)
e7 = np.exp(7)

denom = 2 * e4 + e7
f = np.array([e4 / denom, e7 / denom, e4 / denom])

print("y = 1: ", end='')
print(round(-np.log(f[1]),3))

print("y = 2: ", end='')
print(round(-np.log(f[2]),3))
```

```
y = 1: 0.095
y = 2: 3.095
```

Problem 1.3 (10 points): Network Size

- Suppose we change our network so that there are 12 hidden nodes instead of 2. How many total parameters (weights and biases) are in our new network?

In [ ]:
```
input to hidden: 3*12+12
hidden to outout: 3*12+3
together: 3*12+12+3*12+3=87
```

Problem 2: Neural Networks on MNIST

In this part of the assignment, you will get some hands-on experience working with neural networks. We will be using the scikit-learn implementation of a multi-layer perceptron (MLP). See here for the corresponding documentation. Although there are specialized Python libraries for neural networks, like TensorFlow and PyTorch, in this problem we'll just use scikit-learn since you're already familiar with it.

Problem 2.0: Setting up the Data

First, we'll load our MNIST dataset and split it into a training set and a testing set. Here you are given code that does this for you, and you only need to run it.

We will use the scikit-learn class `StandardScaler` to standardize both the training and testing features. Notice that we **only** fit the `StandardScaler` on the training data, and *not* the testing data.

In [5]:
```
# Load the features and labels for the MNIST dataset
# This might take a minute to download the images.
X, y = fetch_openml('mnist_784', as_frame=False, return_X_y=True)

# Convert labels to integer data type
y = y.astype(int)
```

In [6]:
```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.1, random_state=seed, shuffle=True)
```

In [7]:
```
scaler = StandardScaler()
scaler.fit(X_tr)
```

```
X_tr = scaler.transform(X_tr)     # We can forget about the original values & work
X_te = scaler.transform(X_te)     #  just with the transformed values from here
```

Problem 2.1: Varying the amount of training data (15 points)

One reason that neural networks have become popular in recent years is that, for many problems, we now have access to very large datasets. Since neural networks are very flexible models, they are often able to take advantage of these large datasets in order to achieve high levels of accuracy. In this problem, you will vary the amount of training data available to a neural network and see what effect this has on the model's performance.

In this problem, you should use the following settings for your network:

- A single hidden layer with 64 hidden nodes
- Use the ReLU activation function
- Train the network using stochastic gradient descent (SGD) and a constant learning rate of 0.001
- Use a batch size of 256
- **Make sure to set `random_state=seed`.**

Your task is to implement the following:

- Train an MLP model (with the above hyperparameter settings) using the first `m_tr` feature vectors in `X_tr`, where `m_tr = [100, 1000, 5000, 10000, 20000, 50000, 63000]`. You should use the `MLPClassifier` class from scikit-learn in your implementation.
- Create a plot of the training error and testing error for your MLP model as a function of the number of training data points. For comparison, also plot the training and test error rates we found when we trained a logistic regression model on MNIST (these values are provided below). Again, be sure to include an x-label, y-label, and legend in your plot and use a log-scale on the x-axis.
- Give a short (one or two sentences) description of what you see in your plot. Do you think that more data (beyond these 63000 examples) would continue to improve the model's performance?

Note that training a neural network with a lot of data can be **a slow process**. Hence, you should be careful to implement your code such that it runs in a reasonable amount of time. One recommendation is to test your code using only a small subset of the given `m_tr` values, and only run your code with the larger values of `m_tr` once you are certain your code is working. (For reference, it took about 20 minutes to train all models on a quad-core desktop with no GPU.)

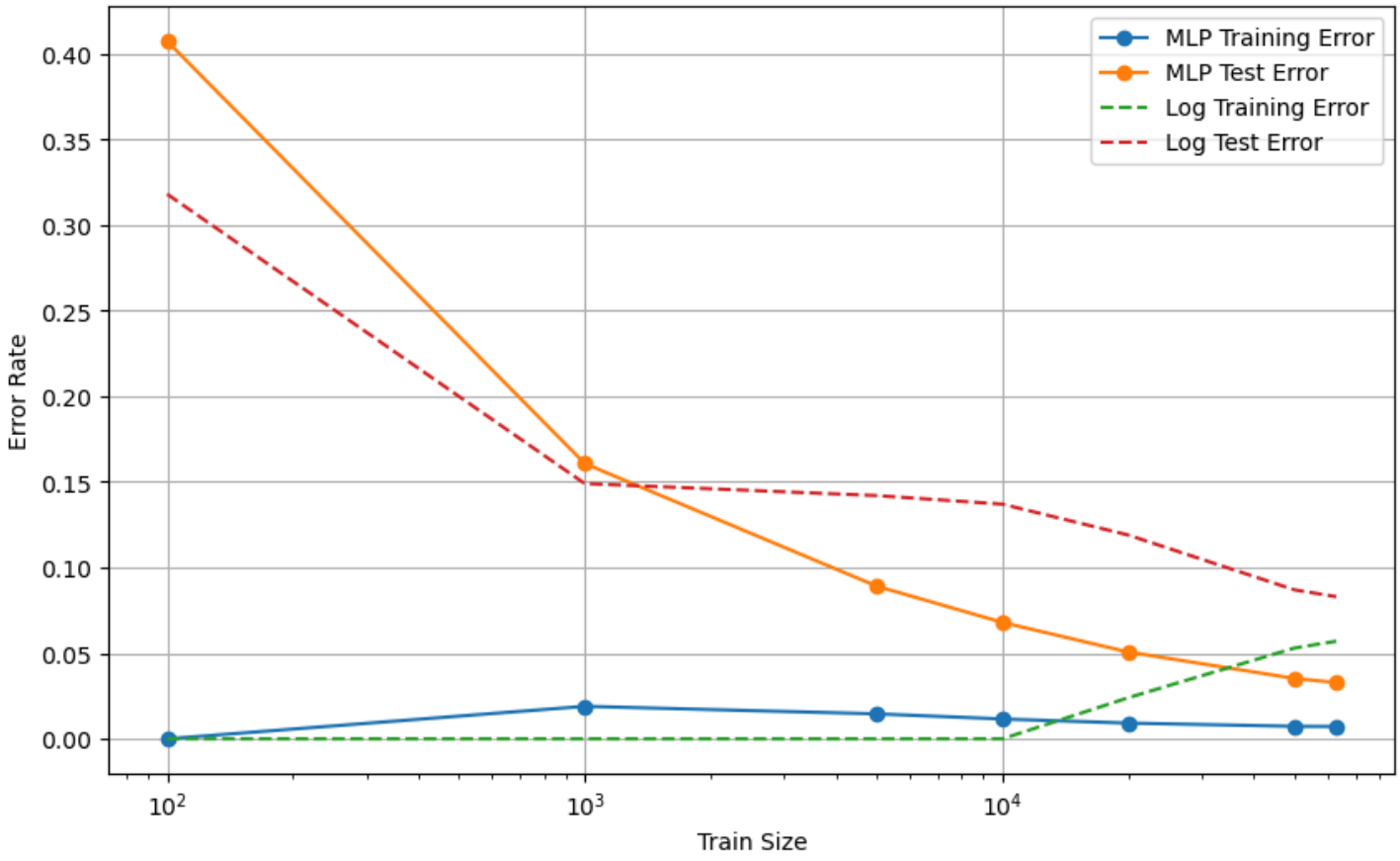
In [10]:
```
import time         # helpful if you want to track execution time
tic = time.time()

train_sizes = [100, 1000, 5000, 10000, 20000, 50000, 63000]
tr_err_mlp = []
te_err_mlp = []
for m_tr in train_sizes:
    ### YOUR CODE STARTS HERE
    X_subset = X_tr[:m_tr]
    y_subset = y_tr[:m_tr]
    learner = MLPClassifier(
                hidden_layer_sizes=(64), activation='relu',
                solver='sgd', learning_rate='constant', learning_rate_init=0.001, batch_size=256,
                random_state=seed
            )

    learner.fit(X_subset,y_subset);

    #loss calculation
    y_train_pred = learner.predict(X_subset)
    y_test_pred = learner.predict(X_te)
    train_error = 1-accuracy_score(y_subset, y_train_pred)
    test_error = 1-accuracy_score(y_te, y_test_pred)
    tr_err_mlp.append(train_error)
```

```
    te_err_mlp.append(test_error)

    print(f'Total elapsed time: {time.time()-tic} for size {m_tr}')


    ### YOUR CODE ENDS HERE
```

```
Total elapsed time: 0.17686939239501953 for size 100
Total elapsed time: 1.2263481616973877 for size 1000
Total elapsed time: 6.065920829772949 for size 5000
Total elapsed time: 16.099424600601196 for size 10000
Total elapsed time: 36.273587226867676 for size 20000
Total elapsed time: 87.34509801864624 for size 50000
Total elapsed time: 151.58473658561707 for size 63000
```

In [13]:
```
# When plotting, use these (rounded) values from the similar logistic regression problem solution:
tr_err_lr = np.array([0.   , 0.   , 0.   , 0.   , 0.024, 0.053, 0.057])
te_err_lr = np.array([0.318, 0.149, 0.142, 0.137, 0.119, 0.087, 0.083])
```

In [15]:
```
## YOUR CODE HERE (PLOTTING)
plt.figure(figsize=(10, 6))
plt.plot(train_sizes, tr_err_mlp, label='MLP Training Error', marker='o')
plt.plot(train_sizes, te_err_mlp, label='MLP Test Error', marker='o')
plt.plot(train_sizes, tr_err_lr, label='Log Training Error', linestyle='--')
plt.plot(train_sizes, te_err_lr, label='Log Test Error', linestyle='--')

plt.xscale('log')
plt.xlabel('Train Size')
plt.ylabel('Error Rate')

plt.legend()
plt.grid(True)
plt.show()
```



In [17]:
```
# DISCUSS
print("""The MLP test error decreases significantly as the training data increases, while the traini
```

```
The MLP test error decreases significantly as the training data increases, while the training error
remains low, approaching zero. Compared to the logistic regression method, MLP demonstrates better g
eneralization performance. However, adding more data may not provide substantial benefits beyond a c
ertain point, as the improvement plateaus around 50,000 examples.
```

Problem 2.2: Optimization Curves (10 points)

One hyperparameter that can have a significant effect on the optimization of your model, and thus its performance, is the learning rate, which controls the step size in (stochastic) gradient descent. In this problem you will vary the learning rate to see what effect this has on how quickly training converges as well as the effect on the performance of your model.

In this problem, you should use the following settings for your network:

- A single hidden layer with 64 hidden nodes
- Use the ReLU activation function
- Train the network using stochastic gradient descent (SGD)
- Use a batch size of 256
- Set `n_iter_no_change=100` and `max_iter=100`. This ensures that all of your networks in this problem will train for 100 epochs (an *epoch* is one full pass over the training data).
- Make sure to set `random_state=seed`.

Your task is to:

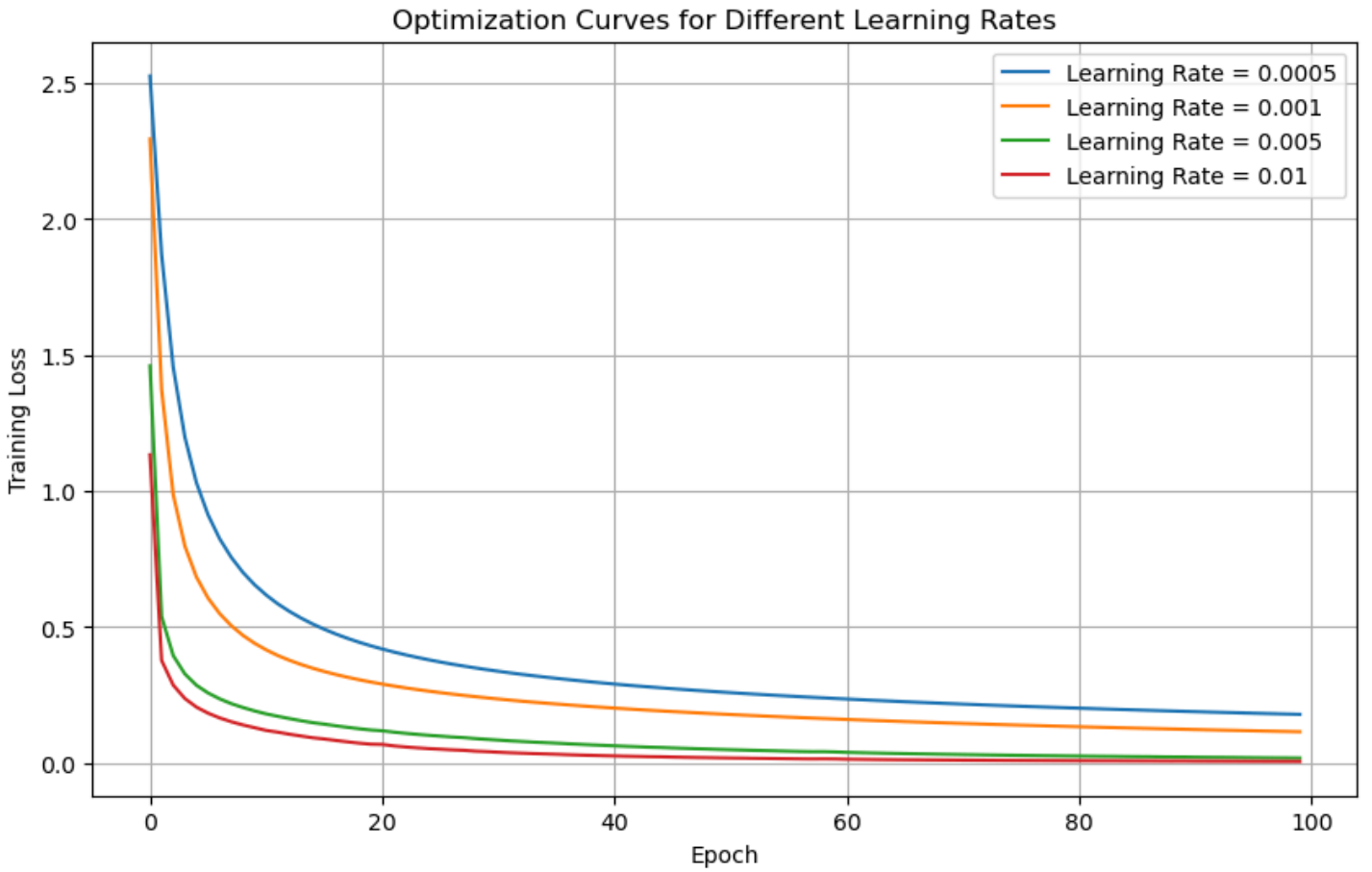
- Train a neural network with the above settings, but vary the learning rate in `lr = [0.0005, 0.001, 0.005, 0.01]`.
- Create a plot showing the training loss as a function of the training epoch (i.e. the x-axis corresponds to training iterations) for each learning rate above. You should have a single plot with four curves. Make sure to include an x-label, a y-label, and a legend in your plot. (Hint: `MLPClassifier` has an attribute `loss_curve_` that you likely find useful.)
- Include a short description of what you see in your plot.

Important: To make your code run faster, you should train all of your networks in this problem on only the first 10,000 images of `X_tr`. In the following cell, you are provided a few lines of code that will create a small training set (with the first 10,000 images in `X_tr`) and a validation set (with the second 10,000 images in `X_tr`). You will use the validation later in Problem 3.3.

In [13]:
```
# Create a smaller training set with the first 10,000 images in X_tr
#   along with a validation set from images 10,000 - 20,000 in X_tr

X_val = X_tr[10000:20000] # Validation set
y_val = y_tr[10000:20000]

X_tr = X_tr[:10000]       # From here on, we will only use these smaller sets,
y_tr = y_tr[:10000]       #  so it's OK to discard the rest of the data
```

In [19]:
```
learning_rates = [0.0005, 0.001, 0.005, 0.01]

err_curves = []

for lr in learning_rates:
    ### YOUR CODE STARTS HERE
    learner = MLPClassifier(hidden_layer_sizes=(64,), activation='relu', solver='sgd',
                        learning_rate_init=lr, batch_size=256,
                        n_iter_no_change=100, max_iter=100,
                        random_state=seed, verbose=False)

    # train model
    learner.fit(X_tr, y_tr)
```

```
    err_curves.append(learner.loss_curve_)

plt.figure(figsize=(10, 6))
for i, lr in enumerate(learning_rates):
    plt.plot(err_curves[i], label=f'Learning Rate = {lr}')

plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Optimization Curves for Different Learning Rates')
plt.legend()
plt.grid(True)
plt.show()

    ### YOUR CODE ENDS HERE
```



In [19]:
```
print("""As the learning rate increases, the training loss decreases more rapidly. The graph compari
```

```
As the learning rate increases, the training loss decreases more rapidly. The graph comparing differ
ent learning rates shows that higher learning rates lead to faster convergence. Additionally, as the
number of epochs increases, the model's performance improves over iterations
```

Problem 2.3: Tuning a Neural Network (10 points)

As you saw in Problem 3.2, there are many hyperparameters of a neural network that can possibly be tuned in order to try to maximize the accuracy of your model. For the final problem of this assignment, it is your job to tune these hyperparameters.

For example, some hyperparameters you might choose to tune are:

- Learning rate
- Depth/width of the hidden layers
- Regularization strength
- Activation functions
- Batch size in stochastic optimization

- etc.

To do this, you should train a network on the training data `X_tr` and evaluate its performance on the validation set `X_val` -- your goal is to achieve the highest possible accuracy on `X_val` by changing the network hyperparameters. **Important: To make your code run faster, you should train all of your networks in this problem on only the first 10,000 images of `X_tr`**. This was already set up for you in Problem 3.2.

Try to find settings that enable you to achieve an error rate smaller than 5% on the validation data. However, tuning neural networks can be difficult; if you cannot achieve this target error rate, be sure to try at least five different neural networks (corresponding to five different settings of the hyperparameters).

In your answer, include a table listing the different hyperparameters that you tried, along with the resulting accuracy on the training and validation sets `X_tr` and `X_val`. Indicate which of these hyperparameter settings you would choose for your final model, and report the accuracy of this final model on the testing set `X_te`.

In [29]:
```
hyperparameters = [
    {'learning_rate_init': 0.01, 'hidden_layer_sizes': (128, 64), 'alpha': 0.01, 'activation': 'relu
    {'learning_rate_init': 0.01, 'hidden_layer_sizes': (128, 64), 'alpha': 0.0001, 'activation': 're
    {'learning_rate_init': 0.001, 'hidden_layer_sizes': (64,), 'alpha': 0.0001, 'activation': 'relu'
    {'learning_rate_init': 0.005, 'hidden_layer_sizes': (128,), 'alpha': 0.001, 'activation': 'relu'
    {'learning_rate_init': 0.001, 'hidden_layer_sizes': (64, 64), 'alpha': 0.0001, 'activation': 'ta
    {'learning_rate_init': 0.0005, 'hidden_layer_sizes': (256,), 'alpha': 0.0001, 'activation': 'log
]
results = []
for idx, params in enumerate(hyperparameters):
    mlp = MLPClassifier(
        learning_rate_init=params['learning_rate_init'],
        hidden_layer_sizes=params['hidden_layer_sizes'],
        alpha=params['alpha'],
        activation=params['activation'],
        batch_size=params['batch_size'],
        solver='sgd',
        max_iter=100,
        random_state=seed
    )

    # train
    mlp.fit(X_tr, y_tr)

    # loss
    train_acc = accuracy_score(y_tr, mlp.predict(X_tr))
    val_acc = accuracy_score(y_val, mlp.predict(X_val))

    results.append({
        'Learning Rate': params['learning_rate_init'],
        'Hidden Layers': params['hidden_layer_sizes'],
        'Alpha': params['alpha'],
        'Activation': params['activation'],
        'Batch Size': params['batch_size'],
        'Train Accuracy': train_acc,
        'Validation Accuracy': val_acc
    })


results_df = pd.DataFrame(results)
print(results_df)
```

```
   Learning Rate Hidden Layers   Alpha Activation  Batch Size  Train Accuracy  \
0         0.0100     (128, 64)  0.0100       relu          64          1.0000
1         0.0100     (128, 64)  0.0001       relu          64          1.0000
2         0.0010         (64,)  0.0001       relu         128          0.9887
3         0.0050        (128,)  0.0010       relu         256          0.9996
4         0.0010      (64, 64)  0.0001       tanh         128          0.9854
5         0.0005        (256,)  0.0001   logistic         128          0.9127

   Validation Accuracy
0               0.9459
1               0.9434
2               0.9338
3               0.9419
4               0.9242
5               0.8957
```

In [15]:
```
# final MLP
X_combined = np.concatenate([X_tr, X_val])
y_combined = np.concatenate([y_tr, y_val])

# train the model
final_mlp = MLPClassifier(
        learning_rate_init=0.01,
        hidden_layer_sizes=(128, 64),
        alpha=0.01,
        activation='relu',
        batch_size=64,
        solver='sgd',
        max_iter=100,
        random_state=seed
    )

final_mlp.fit(X_combined, y_combined)

test_accuracy = accuracy_score(y_te, final_mlp.predict(X_te))

print("""Choose model 0. Parameter as: Learning Rate 0.0100 Hidden Layers (128, 64)  Alpha 0.0100 Ac
print("\nFinal Test Set Accuracy: {:.4f}".format(test_accuracy))
```

```
Choose model 0. Parameter as: Learning Rate 0.0100 Hidden Layers (128, 64)  Alpha 0.0100 Activation
relu Batch Size 64.

Final Test Set Accuracy: 0.9627
```

Problem 3: Torch and Convolutional Networks

In this problem, we will train a small convolutional neural network and compare it to the "standard" MLP model you built in Problem 2. Since `scikit` does not support CNNs, we will implement a simple CNN model using `torch`.

The `torch` library may take a while to install if it is not yet on your system. It should be pre-installed on ICS Jupyter Hub (`hub.ics.uci.edu`) and Google CoLab, if you prefer to use those.

Problem 3.0: Defining the CNN

First, we need to define a CNN model. This is done for you; it consists of one convolutional layer, a pooling layer to down-sample the hidden nodes, and a standard fully-connected or linear layer. It contains methods to calculate the 0/1 loss as well as the negative log-likelihood, and trains using the Adam variant of SGD.

It can (optionally) output a real-time plot of the training process at each epoch, if you would like to assess how it is doing.

In [35]:
```
import torch
torch.set_default_dtype(torch.float64)
```

```
class myConvNet(object):
    def __init__(self):
        # Initialize parameters: assumes data size! 28x28 and 10 classes
        self.conv_ = torch.nn.Conv2d(1, 16, (5,5), stride=2)  # Be careful when declaring sizes;
        self.pool_ = torch.nn.MaxPool2d(3, stride=2)          # inconsistent sizes will give you
        self.lin_ = torch.nn.Linear(400,10)                   # hard-to-read error messages.

    def forward_(self,X):
        """Compute NN forward pass and output class probabilities (as tensor) """
        r1 = self.conv_(X)             # X is (m,1,28,28); R is (m,16,24,24)/2 = (m,16,12,12)
        h1 = torch.relu(r1)            #
        h1_pooled = self.pool_(h1)     # H1 is (m,16,12,12), so H1p is (m,16,10,10)/2 = (m,16,5,5)
        h1_flat = torch.nn.Flatten()(h1_pooled)  # and H1f is (m,400)
        r2 = self.lin_(h1_flat)
        f  = torch.softmax(r2,axis=1)  # Output is (m,10)
        return f

    def parameters(self):
        return list(self.conv_.parameters())+list(self.pool_.parameters())+list(self.lin_.parameters

    def predict(self,X):
        """Compute NN class predictions (as array) """
        m,n = X.shape
        Xtorch = torch.tensor(X).reshape(m,1,int(np.sqrt(n)),int(np.sqrt(n)))
        return self.classes_[np.argmax(self.forward_(Xtorch).detach().numpy(),axis=1)]   # pick the

    def J01(self,X,y):   return (y != self.predict(X)).mean()
    def JNLL_(self,X,y): return -torch.log(self.forward_(X)[range(len(y)),y.astype(int)]).mean()

    def fit(self, X,y, batch_size=256, max_iter=100, learning_rate_init=.005, momentum=0.9, alpha=.0
        self.classes_ = np.unique(y)
        m,n = X.shape
        Xtorch = torch.tensor(X).reshape(m,1,int(np.sqrt(n)),int(np.sqrt(n)))
        self.loss01, self.lossNLL = [self.J01(X,y)], [float(self.JNLL_(Xtorch,y))]

        optimizer = torch.optim.Adam(self.parameters(), lr=learning_rate_init)
        for epoch in range(max_iter):                          # 1 epoch = pass through all data
            pi = np.random.permutation(m)                      # per epoch: permute data order
            for ii,i in enumerate(range(0,m,batch_size)):      # & split into mini-batches
                ivals = pi[i:i+batch_size]
                optimizer.zero_grad()                          # Reset the gradient computation
                Ji = self.JNLL_(Xtorch[ivals,:,:,:],y[ivals])
                Ji.backward()
                optimizer.step()
            self.loss01.append(self.J01(X,y))                  # track 0/1 and NLL losses
            self.lossNLL.append(float(self.JNLL_(Xtorch,y)))

            if plot:                                           # optionally visualize progress
                display.clear_output(wait=True)
                plt.plot(range(epoch+2),self.loss01,'b-',range(epoch+2),self.lossNLL,'c-')
                plt.title(f'J01: {self.loss01[-1]}, NLL: {self.lossNLL[-1]}')
                plt.draw(); plt.pause(.01);
```

Problem 3.1: CNN model structure (10 points)

How many (trainable) parameters are specified in the convolutional network? (If you like, you can count them -- you can access them through each trainable element, e.g., `myConvNet().conv_._parameters['weights']` and `..._parameters['bias']`). List how many from each layer, and the total.

In [43]:
```
learner = myConvNet()


def count_parameters(mod):
    return sum(p.numel() for p in mod.parameters() if p.requires_grad) #filter untrainble params

#conolutional layer
```

```
conv_params = count_parameters(learner.conv_)
print("Convolutional layer:", conv_params)
#pooling layer
pool_params = count_parameters(learner.pool_)
print("Pooling layer:", pool_params)
#fully connected layer
lin_params  = count_parameters(learner.lin_)
print("Fully Connected", lin_params)

print("Total parameters:", conv_params + pool_params + lin_params)
```

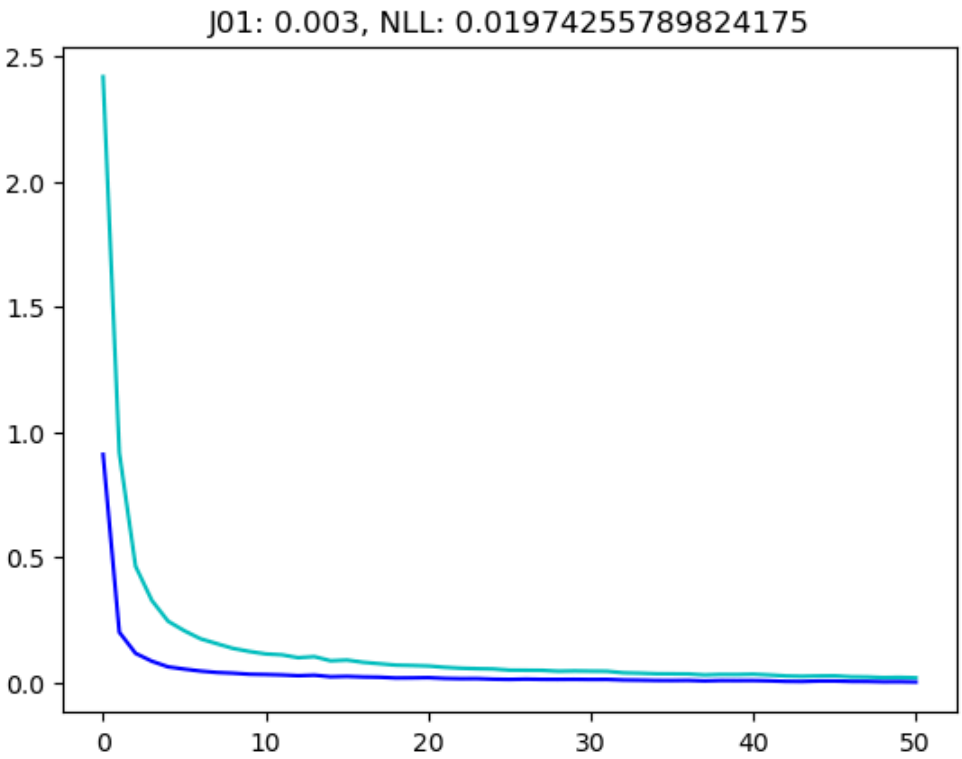
```
Convolutional layer: 416
Pooling layer: 0
Fully Connected 4010
Total parameters: 4426
```

Problem 3.2: Training the model (10 points)

Now train your model on `X_tr`. (Note that this should now include only 10k data points.) If you like you can plot while training to monitor its progress. Train for 50 epochs, using a learning rate of .001.

(Note that my simple training process takes these arguments directly into `fit`, rather than being part of the model properties as is typical in `scikit`.)

In [41]:
```
# Create an instance of the convolutional network
cnn = myConvNet()

# Train the network on X_tr and y_tr for 50 epochs using a learning rate of 0.001.
# The batch size is set to 256, and we use momentum and regularization as specified.
cnn.fit(X_tr, y_tr,
        batch_size=256,
        max_iter=50, #50 epoch?
        learning_rate_init=0.001,
        alpha=0.001,
        plot=True) #see training process
```



Problem 3.3: Evaluation and Discussion (5 points)

Evaluate your CNN model's training, validation, and test error. Compare these to the values you got after optimizing your model's training process in Problem 2.3 (Tuning). Why do you think these differences occur? (Note that your

answer may depend on how well your model in P2.3 did, of course.)

In [49]:
```
# training
train_error_cnn = cnn.loss01[-1]

# validation
y_val_pred = cnn.predict(X_val)
val_error_cnn = 1 - accuracy_score(y_val, y_val_pred)

# test
y_test_pred = cnn.predict(X_te)
test_error_cnn = 1 - accuracy_score(y_te, y_test_pred)

print("CNN Training Error:", train_error_cnn)
print("CNN Validation Error:", val_error_cnn)
print("CNN Test Error:", test_error_cnn)
print(f"Q2.3 MLP model test Error: {1-0.9459}")
```

```
CNN Training Error: 0.003
CNN Validation Error: 0.026499999999999968
CNN Test Error: 0.029571428571428582
Q2.3 MLP model test Error: 0.05410000000000004
```

In [53]:
```
print("""Between two model CNN and MLP, we see that the CNN model achieve a significantly lower test
It might be led by its more advanced inner structure and such ability to capture spatial hierarchies
Compared to MLPs, CNNs utilize a wiser weight sharing and local receptive fields, allowing for more
""")
```

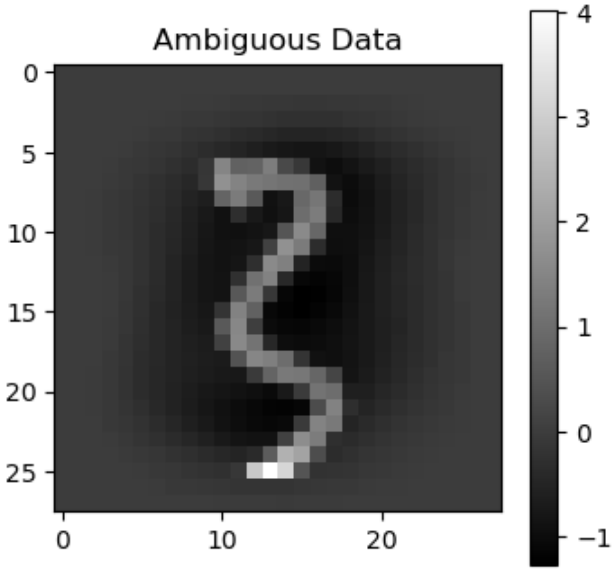
```
Between two model CNN and MLP, we see that the CNN model achieve a significantly lower test error as
0.0296 (compared to 0.0541).
It might be led by its more advanced inner structure and such ability to capture spatial hierarchies
through convolutional layers.
Compared to MLPs, CNNs utilize a wiser weight sharing and local receptive fields, allowing for more
efficient feature extraction and better generalization.
```

Problem 3.4: Comparing Predictions (5 points)

Consider the "somewhat ambiguous" data point `X_val[592]`. Display the data point (it will look a bit weird since it is already normalized). Then, use your trained `MLPClassifier` model to predict the class probabilities. If there are other classes with non-negligible probability, are they plausible? Similarly, find the class probabilities predicted by your CNN model. Compare the two models' uncertainty.

In [59]:
```
# DisplayX_val[592]
plt.figure(figsize=(4,4))
plt.imshow(X_val[592].reshape(28,28), cmap='gray')
plt.title("Ambiguous Data")
plt.colorbar()
plt.show()

# MLP
mlp = MLPClassifier(
        learning_rate_init=0.01,
        hidden_layer_sizes=(128, 64),
        alpha=0.01,
        activation='relu',
        batch_size=64,
        solver='sgd',
        max_iter=100,
        random_state=seed
    )

mlp.fit(X_tr, y_tr)
mlp_probs = mlp.predict_proba(X_val[592].reshape(1, -1))
print("MLP predicted class probabilities:")
for cls, prob in zip(mlp.classes_, mlp_probs[0]):
    print(f"Class {cls}: {prob:.3f}")
```

```
# CNN
tensor = torch.tensor(X_val[592]).reshape(1, 1, 28, 28)
cnn_probs = cnn.forward_(tensor).detach().numpy()
print("\nCNN predicted class probabilities:")
for cls, prob in zip(np.arange(10), cnn_probs[0]):
    print(f"Class {cls}: {prob:.3f}")
```




```
MLP predicted class probabilities:
Class 0: 0.001
Class 1: 0.007
Class 2: 0.013
Class 3: 0.796
Class 4: 0.000
Class 5: 0.011
Class 6: 0.000
Class 7: 0.138
Class 8: 0.012
Class 9: 0.022

CNN predicted class probabilities:
Class 0: 0.000
Class 1: 0.000
Class 2: 0.007
Class 3: 0.992
Class 4: 0.000
Class 5: 0.000
Class 6: 0.000
Class 7: 0.000
Class 8: 0.001
Class 9: 0.000
```

In [61]:
```
print("""Compared to MLP, CNN demonstrates significantly higher certainty in classifying the data pc
Additionally, the MLP assigns a notable probability (0.138) to class 7, suggesting greater uncertair
```

```
Compared to MLP, CNN demonstrates significantly higher certainty in classifying the data point `X_va
l[592]` as class 3 (0.796 vs. 0.992).
Additionally, the MLP assigns a notable probability (0.138) to class 7, suggesting greater uncertain
ty in its classification compared to the CNN.
```

Statement of Collaboration (5 points)

It is **mandatory** to include a Statement of Collaboration in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using EdD) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content before they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to EdD, etc.). Especially after you have started working on the assignment, try to restrict the discussion to EdD as much as possible, so that there is no doubt as to the extent of your collaboration.

In [ ]:
```
Jixuan Luo; discussion 04 used as reference
```