CS178 Homework 4

Due: Monday November 4th, 2024 (11:59pm)

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 [8]:
```
%pip install torch torchvision torchaudio
```

```
Collecting torch
  Using cached torch-2.5.1-cp310-none-macosx_11_0_arm64.whl.metadata (28 kB)
Collecting torchvision
  Downloading torchvision-0.20.1-cp310-cp310-macosx_11_0_arm64.whl.metadata (6.1 kB)
Collecting torchaudio
  Downloading torchaudio-2.5.1-cp310-cp310-macosx_11_0_arm64.whl.metadata (6.4 kB)
Collecting filelock (from torch)
  Using cached filelock-3.16.1-py3-none-any.whl.metadata (2.9 kB)
Collecting typing-extensions>=4.8.0 (from torch)
  Using cached typing_extensions-4.12.2-py3-none-any.whl.metadata (3.0 kB)
Requirement already satisfied: networkx in /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.1
0/site-packages (from torch) (3.2.1)
Requirement already satisfied: jinja2 in /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/
site-packages (from torch) (3.1.2)
Collecting fsspec (from torch)
  Using cached fsspec-2024.10.0-py3-none-any.whl.metadata (11 kB)
Collecting sympy==1.13.1 (from torch)
  Using cached sympy-1.13.1-py3-none-any.whl.metadata (12 kB)
Collecting mpmath<1.4,>=1.1.0 (from sympy==1.13.1->torch)
  Using cached mpmath-1.3.0-py3-none-any.whl.metadata (8.6 kB)
Requirement already satisfied: numpy in /Users/adityasingh/Library/Python/3.10/lib/python/site-packages (fr
om torchvision) (1.24.1)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /Library/Frameworks/Python.framework/Versions/3.10/
lib/python3.10/site-packages (from torchvision) (9.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /Library/Frameworks/Python.framework/Versions/3.10/lib/py
thon3.10/site-packages (from jinja2->torch) (2.1.3)
Using cached torch-2.5.1-cp310-none-macosx_11_0_arm64.whl (63.9 MB)
Using cached sympy-1.13.1-py3-none-any.whl (6.2 MB)
Downloading torchvision-0.20.1-cp310-cp310-macosx_11_0_arm64.whl (1.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 6.7 MB/s eta 0:00:00a 0:00:01
Downloading torchaudio-2.5.1-cp310-cp310-macosx_11_0_arm64.whl (1.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.8/1.8 MB 54.4 MB/s eta 0:00:00
Using cached typing_extensions-4.12.2-py3-none-any.whl (37 kB)
Using cached filelock-3.16.1-py3-none-any.whl (16 kB)
Using cached fsspec-2024.10.0-py3-none-any.whl (179 kB)
Using cached mpmath-1.3.0-py3-none-any.whl (536 kB)
Installing collected packages: mpmath, typing-extensions, sympy, fsspec, filelock, torch, torchvision, torc
haudio
  Attempting uninstall: typing-extensions
    Found existing installation: typing_extensions 4.5.0
    Uninstalling typing_extensions-4.5.0:
      Successfully uninstalled typing_extensions-4.5.0
```

```
  WARNING: The script isympy is installed in '/Library/Frameworks/Python.framework/Versions/3.10/bin' which
is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-l
ocation.
  WARNING: The scripts convert-caffe2-to-onnx, convert-onnx-to-caffe2, torchfrtrace and torchrun are instal
led in '/Library/Frameworks/Python.framework/Versions/3.10/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-l
ocation.
Successfully installed filelock-3.16.1 fsspec-2024.10.0 mpmath-1.3.0 sympy-1.13.1 torch-2.5.1 torchaudio-2.
5.1 torchvision-0.20.1 typing-extensions-4.12.2

[notice] A new release of pip is available: 24.0 -> 24.3.1
[notice] To update, run: python3 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

In [6]:

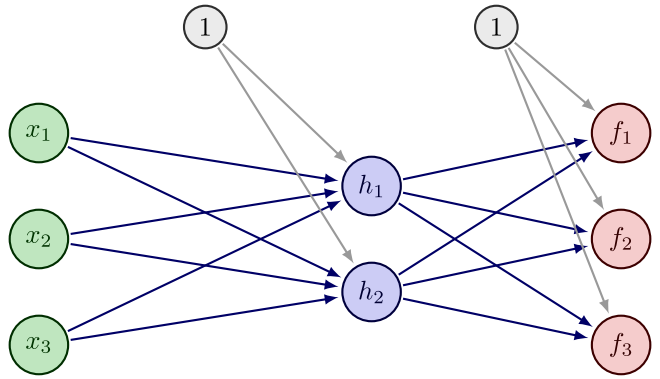
```
import numpy as np
import matplotlib.pyplot as plt

import torch

from IPython import display

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)
```

Out[6]:

```
<torch._C.Generator at 0x1154b5bb0>
```

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 $\mathbf{x}$,

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \begin{bmatrix} 1 & 3 & -2 \end{bmatrix}$$

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 $\mathbf{x}$?

Problem 1.1: Forward Pass Solution

Given Information:

- Input vector:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \begin{bmatrix} 1 & 3 & -2 \end{bmatrix}$$

- Weights for the hidden layer:

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

- Weights for the output layer:

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

Step 1: Compute Hidden Units

The hidden layer has two units h_1 and h_2. These units are calculated using the given weights W and input vector $\mathbf{x}$, followed by applying the ReLU activation function.

Hidden Unit h_1

$$h_1 = w_{01} + w_{11}x_1 + w_{21}x_2 + w_{31}x_3$$
$$h_1 = 1 + (-1) \cdot 1 + 0 \cdot 3 + 5 \cdot (-2)$$
$$h_1 = 1 - 1 + 0 - 10 = -10$$

Applying ReLU:

$$h_1 = \max(0, -10) = 0$$

Hidden Unit h_2

$$h_2 = w_{02} + w_{12}x_1 + w_{22}x_2 + w_{32}x_3$$
$$h_2 = 2 + 1 \cdot 1 + 1 \cdot 3 + 2 \cdot (-2)$$
$$h_2 = 2 + 1 + 3 - 4 = 2$$

Applying ReLU:

$$h_2 = \max(0, 2) = 2$$

Step 2: Compute Outputs

Now we have the hidden units $h_1 = 0$ and $h_2 = 2$. We will use these to compute the outputs f_0, f_1, f_2 using the weights B.

Output f_0

$$f_0 = \beta_{01} + \beta_{11}h_1 + \beta_{21}h_2$$
$$f_0 = 4 + (-1) \cdot 0 + 0 \cdot 2 = 4$$

Output f_1

$$f_1 = \beta_{02} + \beta_{12} h_1 + \beta_{22} h_2$$

$$f_1 = 3 + 0 \cdot 0 + 2 \cdot 2 = 3 + 4 = 7$$

Output f_2

$$f_2 = \beta_{03} + \beta_{13} h_1 + \beta_{23} h_2$$

$$f_2 = 2 + 1 \cdot 0 + 1 \cdot 2 = 2 + 0 + 2 = 4$$

Step 3: Apply Softmax Activation Function

To determine the output class probabilities, we apply the softmax function to f_0, f_1, f_2.

$$\sigma(f_i) = \frac{e^{f_i}}{\sum_{j=0}^{2} e^{f_j}}$$

Compute Exponentials

$$e^{f_0} = e^4, \quad e^{f_1} = e^7, \quad e^{f_2} = e^4$$

Compute Sum of Exponentials

$$\sum_{j=0}^{2} e^{f_j} = e^4 + e^7 + e^4 = 2e^4 + e^7$$

Compute Softmax Probabilities

- For f_0:

$$\sigma(f_0) = \frac{e^4}{2e^4 + e^7}$$

- For f_1:

$$\sigma(f_1) = \frac{e^7}{2e^4 + e^7}$$

- For f_2:

$$\sigma(f_2) = \frac{e^4}{2e^4 + e^7}$$

Step 4: Predicted Class

The predicted class is the one with the highest softmax probability. In this case, f_1 has the highest value, so:

$$\text{Predicted class} = 1$$

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

p0 = e4 / (2 * e4 + e7)
p1 = e7 / (2 * e4 + e7)
p2 = e4 / (2 * e4 + e7)

loss_y1 = -np.log(p1)
loss_y2 = -np.log(p2)
```

```
print(f"Loss when the true label is y = 1: {loss_y1:.4f}")
print(f"Loss when the true label is y = 2: {loss_y2:.4f}")
```

```
Loss when the true label is y = 1: 0.0949
Loss when the true label is y = 2: 3.0949
```

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?

Hidden Layer

Weights:

3×12=36

Biases: 12

Total for hidden layer:

36+12=48

Output Layer

Weights:

12×3=36

Biases:

3

Total for output layer:

36+3=39

Total parameters:

48+39=87

Final Answer:

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 [8]:
```
# 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 [9]:
```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.1, random_state=seed, shuffle=True)
```

In [10]:
```
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 [11]:
```
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_train_subset = X_tr[:m_tr]
    y_train_subset = y_tr[:m_tr]

    mlp = MLPClassifier(hidden_layer_sizes=(64,), activation='relu', solver='sgd',
                        learning_rate_init=0.001, batch_size=256, random_state=seed, max_iter=20)

    mlp.fit(X_train_subset, y_train_subset)

    y_train_pred = mlp.predict(X_train_subset)
    y_test_pred = mlp.predict(X_te)

    tr_err = 1 - accuracy_score(y_train_subset, y_train_pred)
    te_err = 1 - accuracy_score(y_te, y_test_pred)

    tr_err_mlp.append(tr_err)
    te_err_mlp.append(te_err)
print(f'Total elapsed time: {time.time()-tic}')




    ### YOUR CODE ENDS HERE
```

```
Total elapsed time: 0.11368227005004883
Total elapsed time: 0.5100111961364746
Total elapsed time: 2.17391300201416
Total elapsed time: 5.13140606880188
Total elapsed time: 11.393774032592773
Total elapsed time: 27.92508625984192
Total elapsed time: 50.87063503265381
```

In [12]:
```
# 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 [13]:
```
## YOUR CODE HERE (PLOTTING)
plt.figure(figsize=(10, 6))
plt.plot(train_sizes, tr_err_mlp, label='MLP Training Error', color='b', marker='o')
plt.plot(train_sizes, tr_err_lr, label='Logistic Regression Training Error', color='b', linestyle='--')
plt.plot(train_sizes, te_err_mlp, label='MLP Test Error', color='r', marker='o')
plt.plot(train_sizes, te_err_lr, label='Logistic Regression Test Error', color='r', linestyle='--')
plt.xscale('log')
plt.xlabel('Number of Training Data Points (log scale)')
plt.ylabel('Error Rate')
plt.title('Training and Test Error Rates vs. Number of Training Data Points')
plt.legend()
plt.grid(True)

plt.show()
```

In [14]:
```
# DISCUSS
print("\nSummary:\n")
print("As the number of training data points increases, both the training and testing error rates for the M
print("This suggests that more training data is helpful for improving model performance, and it is likely t
```

```
Summary:

As the number of training data points increases, both the training and testing error rates for the MLP mode
l generally decrease.
This suggests that more training data is helpful for improving model performance, and it is likely that per
formance would continue to improve with additional data beyond 63,000 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:

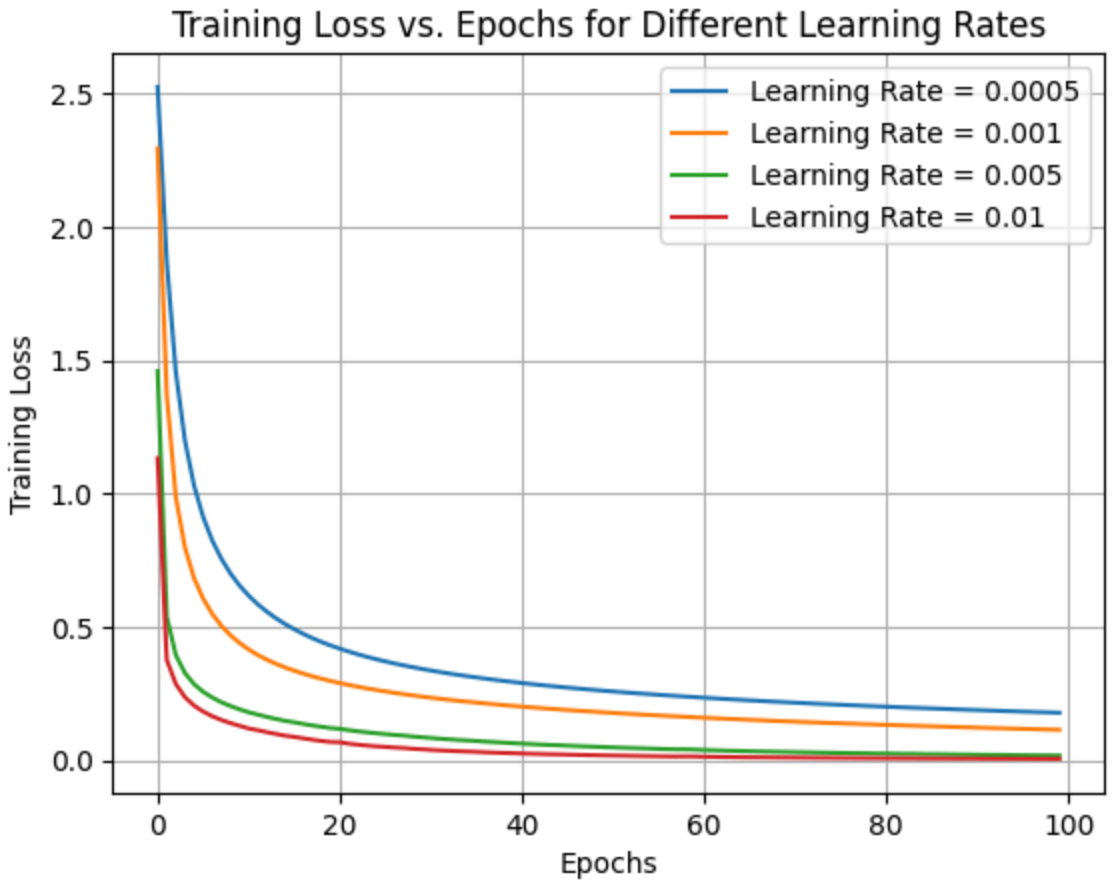
- Train a neural network with the above settings, but vary the learning rate in `lr = [0.0005, 0.001, 0.005, 0.01]`.
- Create a plot showing the training loss as a function of the training epoch (i.e. the x-axis corresponds to training iterations) for each learning rate above. You should have a single 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 [15]:
```
# 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 [16]:
```
learning_rates = [0.0005, 0.001, 0.005, 0.01]

err_curves = []

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

    mlp.fit(X_tr, y_tr)
    plt.plot(mlp.loss_curve_, label=f'Learning Rate = {lr}')

plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.title('Training Loss vs. Epochs for Different Learning Rates')
plt.legend()
plt.grid(True)
plt.show()

print("\nSummary for Problem 2.2:\n")
print("The training loss decreases at different rates depending on the learning rate. Higher learning rates



    ### YOUR CODE ENDS HERE
```



Summary for Problem 2.2:

The training loss decreases at different rates depending on the learning rate. Higher learning rates converge faster but might have higher final loss, while lower learning rates converge more slowly but may achieve lower loss.

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 [17]:
```
hyperparameters = [
    {'hidden_layer_sizes': (128,), 'learning_rate_init': 0.001, 'batch_size': 128, 'activation': 'relu', 'a
    {'hidden_layer_sizes': (64, 64), 'learning_rate_init': 0.005, 'batch_size': 256, 'activation': 'tanh',
    {'hidden_layer_sizes': (32, 32, 32), 'learning_rate_init': 0.01, 'batch_size': 64, 'activation': 'relu
    {'hidden_layer_sizes': (256,), 'learning_rate_init': 0.0005, 'batch_size': 512, 'activation': 'logisti
    {'hidden_layer_sizes': (128, 64), 'learning_rate_init': 0.005, 'batch_size': 256, 'activation': 'relu'
]

results = []

for params in hyperparameters:
    # Initialize the MLPClassifier with the given hyperparameters
    mlp = MLPClassifier(hidden_layer_sizes=params['hidden_layer_sizes'], activation=params['activation'],
                        solver='sgd', learning_rate_init=params['learning_rate_init'], batch_size=params['
                        alpha=params['alpha'], random_state=seed, max_iter=100, n_iter_no_change=100)

    mlp.fit(X_tr, y_tr)
```

```
    y_train_pred = mlp.predict(X_tr)
    y_val_pred = mlp.predict(X_val)

    train_acc = accuracy_score(y_tr, y_train_pred)
    val_acc = accuracy_score(y_val, y_val_pred)
    results.append((params, train_acc, val_acc))

print("\nHyperparameter Tuning Results:\n")
for i, (params, train_acc, val_acc) in enumerate(results):
    print(f"Model {i+1}: {params}")
    print(f"  Training Accuracy: {train_acc:.4f}")
    print(f"  Validation Accuracy: {val_acc:.4f}\n")

best_model = max(results, key=lambda x: x[2])
print("Best Model Hyperparameters:")
print(best_model[0])
print(f"Validation Accuracy: {best_model[2]:.4f}")

final_mlp = MLPClassifier(hidden_layer_sizes=best_model[0]['hidden_layer_sizes'], activation=best_model[0]
                          solver='sgd', learning_rate_init=best_model[0]['learning_rate_init'], batch_size=
                          alpha=best_model[0]['alpha'], random_state=seed, max_iter=100, n_iter_no_change=1
final_mlp.fit(X_tr, y_tr)

y_test_pred = final_mlp.predict(X_te)

test_acc = accuracy_score(y_te, y_test_pred)
print(f"\nFinal Model Test Accuracy: {test_acc:.4f}")
```

```
Hyperparameter Tuning Results:

Model 1: {'hidden_layer_sizes': (128,), 'learning_rate_init': 0.001, 'batch_size': 128, 'activation': 'rel
u', 'alpha': 0.0001}
  Training Accuracy: 0.9909
  Validation Accuracy: 0.9374

Model 2: {'hidden_layer_sizes': (64, 64), 'learning_rate_init': 0.005, 'batch_size': 256, 'activation': 'ta
nh', 'alpha': 0.0001}
  Training Accuracy: 0.9989
  Validation Accuracy: 0.9284

Model 3: {'hidden_layer_sizes': (32, 32, 32), 'learning_rate_init': 0.01, 'batch_size': 64, 'activation':
'relu', 'alpha': 0.001}
  Training Accuracy: 1.0000
  Validation Accuracy: 0.9334

Model 4: {'hidden_layer_sizes': (256,), 'learning_rate_init': 0.0005, 'batch_size': 512, 'activation': 'log
istic', 'alpha': 1e-05}
  Training Accuracy: 0.8585
  Validation Accuracy: 0.8454

Model 5: {'hidden_layer_sizes': (128, 64), 'learning_rate_init': 0.005, 'batch_size': 256, 'activation': 'r
elu', 'alpha': 0.0005}
  Training Accuracy: 1.0000
  Validation Accuracy: 0.9411

Best Model Hyperparameters:
{'hidden_layer_sizes': (128, 64), 'learning_rate_init': 0.005, 'batch_size': 256, 'activation': 'relu', 'al
pha': 0.0005}
Validation Accuracy: 0.9411

Final Model Test Accuracy: 0.9394
```

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 [19]:
```
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 most p
```

```
    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=.001, pl
        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 [20]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

conv_net = myConvNet()
conv_params = conv_net.conv_.weight.numel() + conv_net.conv_.bias.numel()
lin_params = conv_net.lin_.weight.numel() + conv_net.lin_.bias.numel()
```

```
print("\nProblem 3.1: CNN Parameters\n")
print(f"Convolutional Layer Parameters: {conv_params}")
print(f"Fully Connected Layer Parameters: {lin_params}")
print(f"Total Trainable Parameters: {conv_params + lin_params}")
```

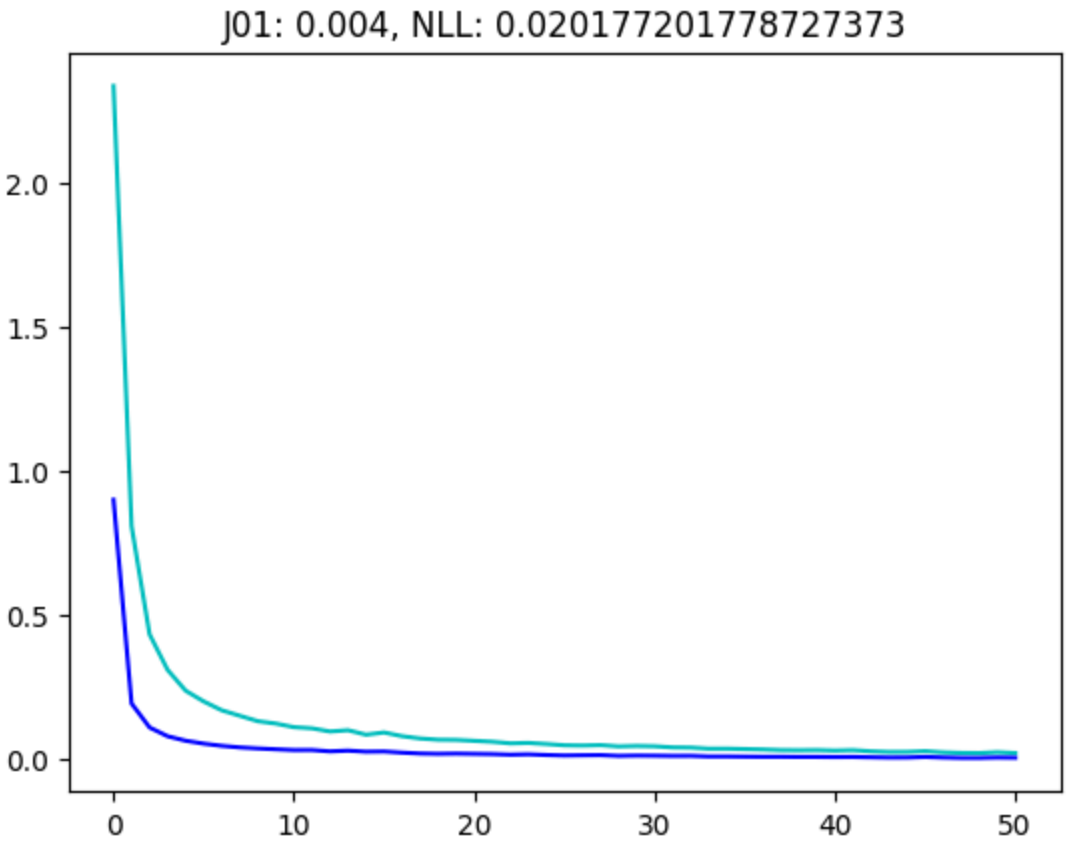
```
Problem 3.1: CNN Parameters

Convolutional Layer Parameters: 416
Fully Connected Layer Parameters: 4010
Total Trainable 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 [21]:
```
conv_net.fit(X_tr, y_tr, batch_size=256, max_iter=50, learning_rate_init=0.001, plot=True)
print("\nSummary for Problem 3.2:\n")
print("The CNN model was trained on 10,000 data points for 50 epochs with a learning rate of 0.001. The rea
```



Summary for Problem 3.2:

The CNN model was trained on 10,000 data points for 50 epochs with a learning rate of 0.001. The real-time plot allows us to monitor the convergence of the loss.

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 [22]:
```
train_error_cnn = conv_net.J01(X_tr, y_tr)
val_error_cnn = conv_net.J01(X_val, y_val)
test_error_cnn = conv_net.J01(X_te, y_te)
```

```
print("\nProblem 3.3: CNN Evaluation and Discussion\n")
print(f"Training Error (CNN): {train_error_cnn:.4f}")
print(f"Validation Error (CNN): {val_error_cnn:.4f}")
print(f"Test Error (CNN): {test_error_cnn:.4f}")

test_error_mlp = 1 - test_acc
print(f"\nComparison with MLP Model:\n")
print(f"Test Error (MLP): {test_error_mlp:.4f}")

print("\nDiscussion:\n")
print("The CNN model achieves different error rates compared to the MLP model. The differences can be attr
```

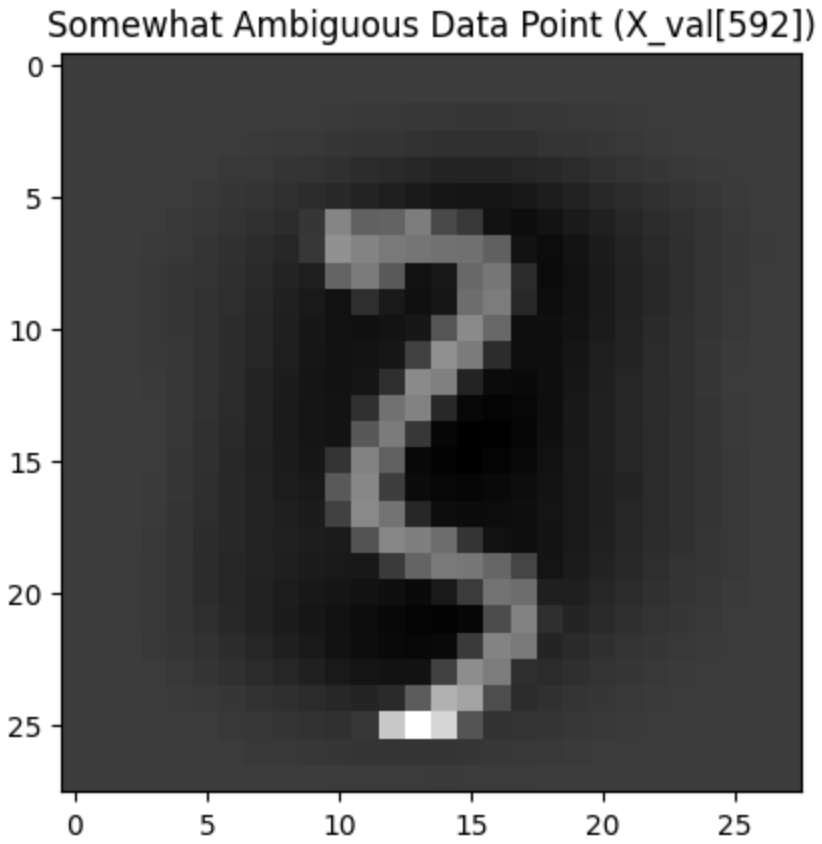
```
Problem 3.3: CNN Evaluation and Discussion

Training Error (CNN): 0.0040
Validation Error (CNN): 0.0268
Test Error (CNN): 0.0323

Comparison with MLP Model:

Test Error (MLP): 0.0606

Discussion:

The CNN model achieves different error rates compared to the MLP model. The differences can be attributed t
o the architectural differences between the models: the CNN is better suited for capturing spatial features
in image data, which likely results in better generalization for the MNIST dataset. The MLP model, on the o
ther hand, does not take advantage of spatial information, which may lead to lower accuracy compared to the
CNN.
```

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 [23]:
```
plt.imshow(X_val[592].reshape(28, 28), cmap='gray')
plt.title('Somewhat Ambiguous Data Point (X_val[592])')
```

```
plt.show()

mlp_probs = final_mlp.predict_proba(X_val[592].reshape(1, -1))
print("\nMLP Class Probabilities:\n")
for i, prob in enumerate(mlp_probs[0]):
    print(f"Class {i}: {prob:.4f}")

X_val_tensor = torch.tensor(X_val[592].reshape(1, 1, 28, 28), dtype=torch.float64)
cnn_probs = conv_net.forward_(X_val_tensor).detach().numpy()[0]
print("\nCNN Class Probabilities:\n")
for i, prob in enumerate(cnn_probs):
    print(f"Class {i}: {prob:.4f}")

print("\nDiscussion on Model Uncertainty:\n")
print("The MLP model and the CNN model may assign different probabilities to each class, reflecting their
print("The CNN model, being more suited to capturing spatial features, may exhibit lower uncertainty for c
```


Somewhat Ambiguous Data Point (X_val[592])
0
5
10
15
20
25
0
5
10
15
20
25

```
MLP Class Probabilities:

Class 0: 0.0019
Class 1: 0.0296
Class 2: 0.1431
Class 3: 0.5384
Class 4: 0.0000
Class 5: 0.0168
Class 6: 0.0001
Class 7: 0.2207
Class 8: 0.0251
Class 9: 0.0244

CNN Class Probabilities:

Class 0: 0.0000
Class 1: 0.0000
Class 2: 0.0152
Class 3: 0.9769
Class 4: 0.0000
Class 5: 0.0000
Class 6: 0.0000
Class 7: 0.0032
Class 8: 0.0046
Class 9: 0.0000

Discussion on Model Uncertainty:

The MLP model and the CNN model may assign different probabilities to each class, reflecting their respecti
ve levels of uncertainty.
The CNN model, being more suited to capturing spatial features, may exhibit lower uncertainty for certain a
mbiguous images compared to the MLP model.
```

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.

N/A