CS273A 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.

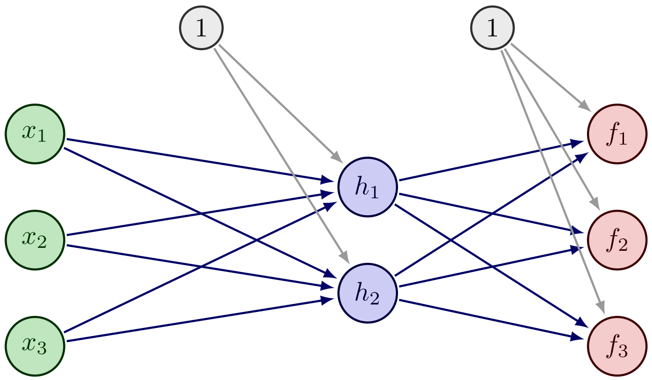
In [5]:

```
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);
```

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 $\mathbf{x}$?

We can compute each hidden unit as

$$h_i = \text{ReLU}(w_{0i} + w_{1i}x_1 + w_{2i}x_2 + w_{3i}x_3)$$

Thus we have

$$h_1 = \text{ReLU}(1 + (-1 \times 1) + (0 \times 3) + (5 \times -2)).$$

$$= \text{ReLU}(1 - 1 + 0 - 10) = \text{ReLU}(-10) = 0.$$

$$h_2 = \text{ReLU}(2 + (1 \times 1) + (1 \times 3) + (2 \times -2)).$$

$$= \text{ReLU}(2 + 1 + 3 - 4) = \text{ReLU}(2) = 2.$$

Thus, the hidden layer output is:

$$\mathbf{h} = \begin{bmatrix} h_1 & h_2 \end{bmatrix} = \begin{bmatrix} 0 & 2 \end{bmatrix}$$

Each output unit (f_j) is calculated using the softmax function, based on the linear response:

$$r_j = \beta_{0j} + \beta_{1j}h_1 + \beta_{2j}h_2$$

Thus we can obtain the response of the output layer as

$$r_0 = 4 + (-1 \times 0) + (0 \times 2) = 4$$

$$r_1 = 3 + (0 \times 0) + (2 \times 2) = 3 + 4 = 7$$

$$r_2 = 2 + (1 \times 0) + (1 \times 2) = 2 + 2 = 4$$

The softmax for each class (f_j) is:

$$f_j = \frac{e^{r_j}}{\sum_{k=0}^{2} e^{r_k}}$$

Calculating $e^{r_0} = e^4$, $e^{r_1} = e^7$, $e^{r_2} = e^4$, we get:

$$\sum_{k=0}^{2} e^{r_k} = e^4 + e^7 + e^4 = 2e^4 + e^7$$

Thus, the output probabilities are $f_0 = \frac{e^4}{2e^4+e^7}$, $f_1 = \frac{e^7}{2e^4+e^7}$, $f_2 = \frac{e^4}{2e^4+e^7}$, respectively.

The network predicts the class with the highest probability, which is class f_1 (corresponding to $r_1 = 7$). Therefore, the network predicts **class 1** for the input $\mathbf{x}$.

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 calculating the log-loss (cross-entropy loss), for a given predicted probability f_j and true label y, the cross-entropy loss is defined as:

$$\text{Loss} = -\log(f_y)$$

where f_y is the network's predicted probability for the true class y.

For the case where the true label is (y = 1), the log-loss is:

$$\text{Loss} = -\log(f_1) = -\log\left(\frac{e^7}{2 \cdot e^4 + e^7}\right)$$

For the case where the true label is (y = 2), the log-loss is:

$$\text{Loss} = -\log(f_2) = -\log\left(\frac{e^4}{2 \cdot e^4 + e^7}\right)$$

In [13]:

```
# We can use the following code to further compute the numerical values
# Compute e^4 and e^7
e4 = np.exp(4)
e7 = np.exp(7)
```

```
# Calculate softmax probabilities
f0 = e4 / (2 * e4 + e7)
f1 = e7 / (2 * e4 + e7)
f2 = e4 / (2 * e4 + e7)

# Compute log-loss
loss_y1 = -np.log(f1)  # Loss when y=1
loss_y2 = -np.log(f2)  # Loss when y=2

print("Loss 1:",loss_y1, "\nLoss 2:",loss_y2)
```

```
Loss 1: 0.09492295642096098
Loss 2: 3.094922956420961
```

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?

To determine the total number of parameters in a neural network, we need to account for both the weights and the biases in each layer: With 3 inputs (x_1, x_2, x_3) and 12 hidden nodes, each hidden node will have a weight for each input, plus an additional bias term. Thus the number of weights for hidden layer is $3 \times 12 = 36$, and the number of biases for hidden layer is 12. So, the total number of parameters for the input-to-hidden layer is $36 + 12 = 48$.

With 12 hidden nodes connecting to 3 output nodes, each output node will have a weight from each hidden node, plus an additional bias term. Thus the number of weights for hidden layer is $12 \times 3 = 36$, and the number of biases for hidden layer is 3. So, the total number of parameters for the hidden-to-output layer is $36 + 3 = 39$.

Thus the number of total parameters of the new network is $48 + 39 = 87$.

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 [19]:
```
# 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 [20]:
```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.1, random_state=seed, shuffle=True)
```

In [21]:
```
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 [23]:
```
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
    # Create the MLP model with specified hyperparameters
    model = MLPClassifier(hidden_layer_sizes=(64,),
```

```python
                        activation='relu',
                        solver='sgd',
                        learning_rate_init=0.001,
                        batch_size=256,
                        random_state=seed,
                        max_iter=1000)

    # Train the model using the first m_tr feature vectors
    model.fit(X_tr[:m_tr], y_tr[:m_tr])

    # Compute training error (1 - accuracy)
    y_tr_pred = model.predict(X_tr[:m_tr])
    tr_err = 1 - accuracy_score(y_tr[:m_tr], y_tr_pred)
    tr_err_mlp.append(tr_err)

    # Compute testing error
    y_te_pred = model.predict(X_te)  # Assuming X_te and y_te are your test data and labels
    te_err = 1 - accuracy_score(y_te, y_te_pred)
    te_err_mlp.append(te_err)

    # print(f'Total elapsed time: {time.time()-tic}')
print(f'Total elapsed time: {time.time()-tic}')

    ### YOUR CODE ENDS HERE
```

```
Total elapsed time: 499.5516428947449
```

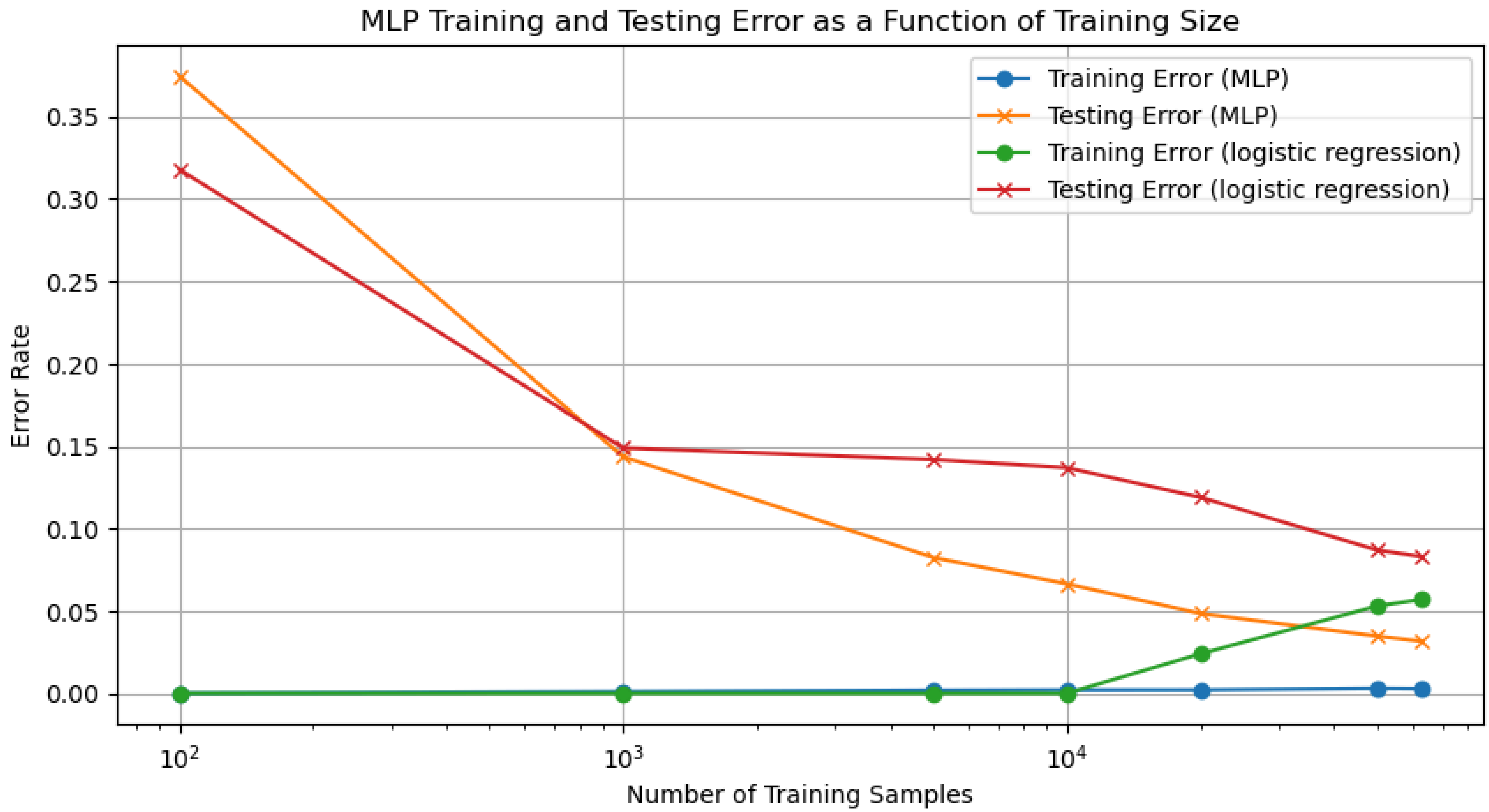
In [24]:

```python
# 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 [25]:

```python
## YOUR CODE HERE (PLOTTING)
# Plot the training and testing errors
plt.figure(figsize=(10, 5))
plt.plot(train_sizes, tr_err_mlp, label='Training Error (MLP)', marker='o')
plt.plot(train_sizes, te_err_mlp, label='Testing Error (MLP)', marker='x')
plt.plot(train_sizes, tr_err_lr, label='Training Error (logistic regression)', marker='o')
plt.plot(train_sizes, te_err_lr, label='Testing Error (logistic regression)', marker='x')
plt.xscale('log')
plt.xlabel('Number of Training Samples')
plt.ylabel('Error Rate')
```

```python
plt.title('MLP Training and Testing Error as a Function of Training Size')
plt.legend()
plt.grid()
plt.show()
```



DISCUSS: The plot illustrates that the MLP model achieves lower training and testing errors compared to logistic regression across various training sizes. This suggests that the MLP is better suited to capturing the complexities in the data. Given the performance observed with 63,000 examples, it is likely that additional data could further enhance the model's performance. In the plot, the training error decreases significantly as the amount of training data increases, indicating that the MLP model learns more effectively with larger datasets. The testing error also shows a downward trend, but it appears to level off after around 1000 examples, suggesting diminishing returns for model performance with additional data beyond 63,000 examples; therefore, while more data may still provide some improvements, it is likely that the impact would be minimal compared to the gains observed from the initial increases in training size.

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:

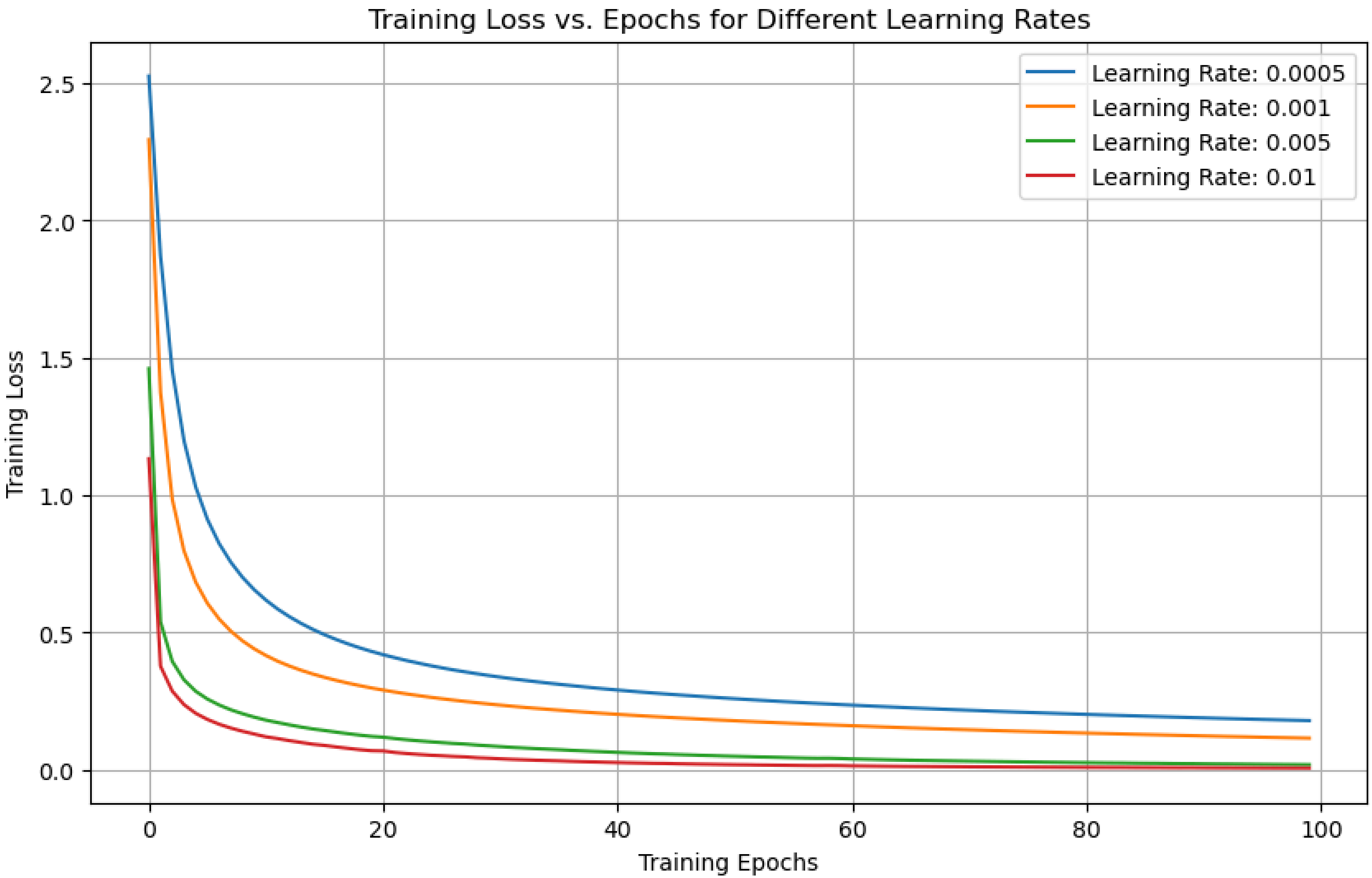
- Train a neural network with the above settings, but vary the learning rate in `lr = [0.0005, 0.001, 0.005, 0.01]`.
- Create a plot showing the training loss as a function of the training epoch (i.e. the x-axis corresponds to training iterations) for each learning rate above. You should have a single 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 [29]:
```python
# 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 [30]:
```python
learning_rates = [0.0005, 0.001, 0.005, 0.01]

err_curves = []

for lr in learning_rates:
    ### YOUR CODE STARTS HERE
    # Iterate over the different learning rates
    model = 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)

    # Fit the model on the training data
    model.fit(X_tr, y_tr)

    # Store the loss curve for this learning rate
    err_curves.append(model.loss_curve_)

# Plot the training loss curves for each learning rate
plt.figure(figsize=(10, 6))

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

# Customize the plot
plt.xlabel('Training Epochs')
plt.ylabel('Training Loss')
```

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

    ### YOUR CODE ENDS HERE
```



DISCUSS: In the plot we can observe that higher learning rates lead to faster convergence, with a more rapid decrease in training loss compared to lower learning rates. Notably, there are no significant fluctuations in the loss values, indicating stable training throughout the epochs. Ultimately, the higher learning rates achieve lower final training loss, suggesting that they facilitate faster learning and potentially better model performance for this specific task / dataset.

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 [33]:
```python
# Define the different hyperparameters to try
learning_rates = [0.001, 0.005, 0.01]
hidden_layer_sizes = [(64,), (128,), (256,)]
regularization_strengths = [0.0001, 0.001, 0.01]
activation_functions = ['relu', 'tanh']
batch_sizes = [64, 128, 256]
```

In [ ]:
```python
# To store results
results = []
for lr in learning_rates:
    for hidden_layers in hidden_layer_sizes:
        for reg in regularization_strengths:
            for activation in activation_functions:
                for batch_size in batch_sizes:
                    print("New Setting")
                    # Define and train the MLPClassifier
                    model = MLPClassifier(hidden_layer_sizes=hidden_layers,
                                          learning_rate_init=lr,
                                          alpha=reg,
                                          activation=activation,
                                          batch_size=batch_size,
                                          max_iter=100,
                                          random_state=seed)

                    # Train the model on the training data
                    model.fit(X_tr, y_tr)

                    # Evaluate on training and validation sets
                    train_acc = accuracy_score(y_tr, model.predict(X_tr))
                    val_acc = accuracy_score(y_val, model.predict(X_val))

                    # Store the results
                    results.append({
                        "learning_rate": lr,
                        "hidden_layers": hidden_layers,
                        "regularization": reg,
                        "activation": activation,
                        "batch_size": batch_size,
                        "train_accuracy": train_acc,
```

```
                        "val_accuracy": val_acc
                    })
```

In [35]:
```
# Sort results by validation accuracy
best_model = sorted(results, key=lambda x: x['val_accuracy'], reverse=True)[0]

print("Best Model Configuration:")
print(best_model)

# Train on the best configuration
final_model = MLPClassifier(
    hidden_layer_sizes=best_model["hidden_layers"],
    learning_rate_init=best_model["learning_rate"],
    alpha=best_model["regularization"],
    activation=best_model["activation"],
    batch_size=best_model["batch_size"],
    max_iter=100,
    random_state=seed
)

# Fit the final model on the entire training set and evaluate on the test set
final_model.fit(X_tr, y_tr)
test_accuracy = accuracy_score(y_te, final_model.predict(X_te))

print("Final Model Test Accuracy:", test_accuracy)
```

```
Best Model Configuration:
{'learning_rate': 0.005, 'hidden_layers': (256,), 'regularization': 0.001, 'activation': 'relu', 'batch_size': 256,
'train_accuracy': 1.0, 'val_accuracy': 0.9549}
Final Model Test Accuracy: 0.9524285714285714
```

In [98]:
```
import pandas as pd
results_df = pd.DataFrame(results)
pd.set_option('display.max_rows', None)
results_df
```

Out[98]:

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
0	0.001	(64,)	0.0001	relu	64	1.0000	0.9488
1	0.001	(64,)	0.0001	relu	128	1.0000	0.9477
2	0.001	(64,)	0.0001	relu	256	1.0000	0.9456
3	0.001	(64,)	0.0001	tanh	64	1.0000	0.9301
4	0.001	(64,)	0.0001	tanh	128	1.0000	0.9286
5	0.001	(64,)	0.0001	tanh	256	1.0000	0.9276
6	0.001	(64,)	0.0010	relu	64	1.0000	0.9500
7	0.001	(64,)	0.0010	relu	128	1.0000	0.9483
8	0.001	(64,)	0.0010	relu	256	1.0000	0.9461
9	0.001	(64,)	0.0010	tanh	64	1.0000	0.9336
10	0.001	(64,)	0.0010	tanh	128	1.0000	0.9302
11	0.001	(64,)	0.0010	tanh	256	1.0000	0.9285
12	0.001	(64,)	0.0100	relu	64	1.0000	0.9509
13	0.001	(64,)	0.0100	relu	128	1.0000	0.9506
14	0.001	(64,)	0.0100	relu	256	1.0000	0.9477
15	0.001	(64,)	0.0100	tanh	64	1.0000	0.9369
16	0.001	(64,)	0.0100	tanh	128	1.0000	0.9318
17	0.001	(64,)	0.0100	tanh	256	1.0000	0.9303
18	0.001	(128,)	0.0001	relu	64	1.0000	0.9508
19	0.001	(128,)	0.0001	relu	128	1.0000	0.9491
20	0.001	(128,)	0.0001	relu	256	1.0000	0.9473
21	0.001	(128,)	0.0001	tanh	64	1.0000	0.9356
22	0.001	(128,)	0.0001	tanh	128	1.0000	0.9331

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
23	0.001	(128,)	0.0001	tanh	256	1.0000	0.9326
24	0.001	(128,)	0.0010	relu	64	1.0000	0.9516
25	0.001	(128,)	0.0010	relu	128	1.0000	0.9509
26	0.001	(128,)	0.0010	relu	256	1.0000	0.9481
27	0.001	(128,)	0.0010	tanh	64	1.0000	0.9371
28	0.001	(128,)	0.0010	tanh	128	1.0000	0.9346
29	0.001	(128,)	0.0010	tanh	256	1.0000	0.9335
30	0.001	(128,)	0.0100	relu	64	1.0000	0.9542
31	0.001	(128,)	0.0100	relu	128	1.0000	0.9517
32	0.001	(128,)	0.0100	relu	256	1.0000	0.9504
33	0.001	(128,)	0.0100	tanh	64	1.0000	0.9425
34	0.001	(128,)	0.0100	tanh	128	1.0000	0.9396
35	0.001	(128,)	0.0100	tanh	256	1.0000	0.9367
36	0.001	(256,)	0.0001	relu	64	1.0000	0.9536
37	0.001	(256,)	0.0001	relu	128	1.0000	0.9532
38	0.001	(256,)	0.0001	relu	256	1.0000	0.9508
39	0.001	(256,)	0.0001	tanh	64	1.0000	0.9403
40	0.001	(256,)	0.0001	tanh	128	1.0000	0.9370
41	0.001	(256,)	0.0001	tanh	256	1.0000	0.9362
42	0.001	(256,)	0.0010	relu	64	1.0000	0.9542
43	0.001	(256,)	0.0010	relu	128	1.0000	0.9542
44	0.001	(256,)	0.0010	relu	256	1.0000	0.9523
45	0.001	(256,)	0.0010	tanh	64	1.0000	0.9437

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
46	0.001	(256,)	0.0010	tanh	128	1.0000	0.9394
47	0.001	(256,)	0.0010	tanh	256	1.0000	0.9372
48	0.001	(256,)	0.0100	relu	64	1.0000	0.9546
49	0.001	(256,)	0.0100	relu	128	1.0000	0.9538
50	0.001	(256,)	0.0100	relu	256	1.0000	0.9532
51	0.001	(256,)	0.0100	tanh	64	1.0000	0.9467
52	0.001	(256,)	0.0100	tanh	128	1.0000	0.9444
53	0.001	(256,)	0.0100	tanh	256	1.0000	0.9424
54	0.005	(64,)	0.0001	relu	64	0.9993	0.9441
55	0.005	(64,)	0.0001	relu	128	0.9983	0.9429
56	0.005	(64,)	0.0001	relu	256	1.0000	0.9497
57	0.005	(64,)	0.0001	tanh	64	1.0000	0.9341
58	0.005	(64,)	0.0001	tanh	128	1.0000	0.9304
59	0.005	(64,)	0.0001	tanh	256	1.0000	0.9282
60	0.005	(64,)	0.0010	relu	64	0.9957	0.9439
61	0.005	(64,)	0.0010	relu	128	0.9990	0.9443
62	0.005	(64,)	0.0010	relu	256	1.0000	0.9498
63	0.005	(64,)	0.0010	tanh	64	0.9976	0.9261
64	0.005	(64,)	0.0010	tanh	128	1.0000	0.9316
65	0.005	(64,)	0.0010	tanh	256	1.0000	0.9303
66	0.005	(64,)	0.0100	relu	64	1.0000	0.9495
67	0.005	(64,)	0.0100	relu	128	1.0000	0.9468
68	0.005	(64,)	0.0100	relu	256	0.9864	0.9390

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
69	0.005	(64,)	0.0100	tanh	64	0.9989	0.9307
70	0.005	(64,)	0.0100	tanh	128	0.9996	0.9309
71	0.005	(64,)	0.0100	tanh	256	0.9967	0.9248
72	0.005	(128,)	0.0001	relu	64	0.9967	0.9470
73	0.005	(128,)	0.0001	relu	128	1.0000	0.9512
74	0.005	(128,)	0.0001	relu	256	1.0000	0.9504
75	0.005	(128,)	0.0001	tanh	64	0.9999	0.9414
76	0.005	(128,)	0.0001	tanh	128	1.0000	0.9389
77	0.005	(128,)	0.0001	tanh	256	1.0000	0.9380
78	0.005	(128,)	0.0010	relu	64	0.9936	0.9436
79	0.005	(128,)	0.0010	relu	128	1.0000	0.9514
80	0.005	(128,)	0.0010	relu	256	1.0000	0.9505
81	0.005	(128,)	0.0010	tanh	64	0.9999	0.9302
82	0.005	(128,)	0.0010	tanh	128	1.0000	0.9409
83	0.005	(128,)	0.0010	tanh	256	1.0000	0.9400
84	0.005	(128,)	0.0100	relu	64	0.9836	0.9348
85	0.005	(128,)	0.0100	relu	128	0.9986	0.9469
86	0.005	(128,)	0.0100	relu	256	1.0000	0.9470
87	0.005	(128,)	0.0100	tanh	64	0.9628	0.9179
88	0.005	(128,)	0.0100	tanh	128	1.0000	0.9408
89	0.005	(128,)	0.0100	tanh	256	1.0000	0.9401
90	0.005	(256,)	0.0001	relu	64	0.9971	0.9504
91	0.005	(256,)	0.0001	relu	128	0.9969	0.9441

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
92	0.005	(256,)	0.0001	relu	256	1.0000	0.9532
93	0.005	(256,)	0.0001	tanh	64	1.0000	0.9448
94	0.005	(256,)	0.0001	tanh	128	1.0000	0.9429
95	0.005	(256,)	0.0001	tanh	256	1.0000	0.9405
96	0.005	(256,)	0.0010	relu	64	0.9872	0.9380
97	0.005	(256,)	0.0010	relu	128	0.9917	0.9457
98	0.005	(256,)	0.0010	relu	256	1.0000	0.9549
99	0.005	(256,)	0.0010	tanh	64	1.0000	0.9396
100	0.005	(256,)	0.0010	tanh	128	1.0000	0.9440
101	0.005	(256,)	0.0010	tanh	256	1.0000	0.9426
102	0.005	(256,)	0.0100	relu	64	0.9928	0.9468
103	0.005	(256,)	0.0100	relu	128	0.9940	0.9434
104	0.005	(256,)	0.0100	relu	256	1.0000	0.9491
105	0.005	(256,)	0.0100	tanh	64	0.9871	0.9282
106	0.005	(256,)	0.0100	tanh	128	1.0000	0.9448
107	0.005	(256,)	0.0100	tanh	256	1.0000	0.9436
108	0.010	(64,)	0.0001	relu	64	0.9939	0.9405
109	0.010	(64,)	0.0001	relu	128	0.9949	0.9418
110	0.010	(64,)	0.0001	relu	256	0.9933	0.9404
111	0.010	(64,)	0.0001	tanh	64	0.9818	0.9167
112	0.010	(64,)	0.0001	tanh	128	0.9999	0.9290
113	0.010	(64,)	0.0001	tanh	256	0.9977	0.9176
114	0.010	(64,)	0.0010	relu	64	0.9926	0.9410

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
115	0.010	(64,)	0.0010	relu	128	0.9906	0.9374
116	0.010	(64,)	0.0010	relu	256	1.0000	0.9485
117	0.010	(64,)	0.0010	tanh	64	0.9822	0.9165
118	0.010	(64,)	0.0010	tanh	128	0.9848	0.9122
119	0.010	(64,)	0.0010	tanh	256	1.0000	0.9289
120	0.010	(64,)	0.0100	relu	64	0.9857	0.9342
121	0.010	(64,)	0.0100	relu	128	0.9815	0.9305
122	0.010	(64,)	0.0100	relu	256	0.9935	0.9423
123	0.010	(64,)	0.0100	tanh	64	0.9703	0.9150
124	0.010	(64,)	0.0100	tanh	128	0.9883	0.9227
125	0.010	(64,)	0.0100	tanh	256	0.9969	0.9199
126	0.010	(128,)	0.0001	relu	64	0.9950	0.9466
127	0.010	(128,)	0.0001	relu	128	0.9972	0.9458
128	0.010	(128,)	0.0001	relu	256	0.9953	0.9416
129	0.010	(128,)	0.0001	tanh	64	0.9915	0.9194
130	0.010	(128,)	0.0001	tanh	128	1.0000	0.9394
131	0.010	(128,)	0.0001	tanh	256	1.0000	0.9382
132	0.010	(128,)	0.0010	relu	64	0.9964	0.9471
133	0.010	(128,)	0.0010	relu	128	0.9884	0.9406
134	0.010	(128,)	0.0010	relu	256	0.9901	0.9374
135	0.010	(128,)	0.0010	tanh	64	0.9742	0.9164
136	0.010	(128,)	0.0010	tanh	128	0.9993	0.9301
137	0.010	(128,)	0.0010	tanh	256	1.0000	0.9396

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
138	0.010	(128,)	0.0100	relu	64	0.9703	0.9247
139	0.010	(128,)	0.0100	relu	128	0.9816	0.9353
140	0.010	(128,)	0.0100	relu	256	0.9854	0.9334
141	0.010	(128,)	0.0100	tanh	64	0.9640	0.9151
142	0.010	(128,)	0.0100	tanh	128	0.9928	0.9278
143	0.010	(128,)	0.0100	tanh	256	0.9998	0.9318
144	0.010	(256,)	0.0001	relu	64	0.9972	0.9483
145	0.010	(256,)	0.0001	relu	128	0.9912	0.9379
146	0.010	(256,)	0.0001	relu	256	0.9817	0.9351
147	0.010	(256,)	0.0001	tanh	64	0.9854	0.9202
148	0.010	(256,)	0.0001	tanh	128	1.0000	0.9368
149	0.010	(256,)	0.0001	tanh	256	1.0000	0.9414
150	0.010	(256,)	0.0010	relu	64	0.9914	0.9473
151	0.010	(256,)	0.0010	relu	128	0.9933	0.9467
152	0.010	(256,)	0.0010	relu	256	0.9904	0.9384
153	0.010	(256,)	0.0010	tanh	64	0.9800	0.9183
154	0.010	(256,)	0.0010	tanh	128	0.9997	0.9287
155	0.010	(256,)	0.0010	tanh	256	1.0000	0.9424
156	0.010	(256,)	0.0100	relu	64	0.9799	0.9359
157	0.010	(256,)	0.0100	relu	128	0.9951	0.9500
158	0.010	(256,)	0.0100	relu	256	0.9869	0.9377
159	0.010	(256,)	0.0100	tanh	64	0.9653	0.9110
160	0.010	(256,)	0.0100	tanh	128	0.9810	0.9225

	learning_rate	hidden_layers	regularization	activation	batch_size	train_accuracy	val_accuracy
161	0.010	(256,)	0.0100	tanh	256	1.0000	0.9406

DISCUSS: Best Model Configuration: {'learning_rate': 0.005, 'hidden_layers': (256,), 'regularization': 0.001, 'activation': 'relu', 'batch_size': 256, 'train_accuracy': 1.0, 'val_accuracy': 0.9549}

Final Model Test Accuracy: 0.9524285714285714

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 [40]:
```
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.
```

```python
    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 probable c

    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, plot=False)
        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 [68]:

```
# Instantiate the model
model = myConvNet()

# Access the parameters for the convolutional layer
conv_weights = model.conv_._parameters['weight']  # Shape: [16, 1, 5, 5]
conv_bias = model.conv_._parameters['bias']       # Shape: [16]

# Calculate number of parameters in conv layer
conv_weights_count = conv_weights.numel()  # total elements in the weight tensor
conv_bias_count = conv_bias.numel()        # total elements in the bias tensor
total_conv_params = conv_weights_count + conv_bias_count

# Access the parameters for the linear layer
lin_weights = model.lin_._parameters['weight']   # Shape: [10, 400]
lin_bias = model.lin_._parameters['bias']        # Shape: [10]

# Calculate number of parameters in linear layer
lin_weights_count = lin_weights.numel()   # total elements in the weight tensor
lin_bias_count = lin_bias.numel()         # total elements in the bias tensor
total_lin_params = lin_weights_count + lin_bias_count

# Total trainable parameters
total_params = total_conv_params + total_lin_params

# Output the counts
print("Convolutional Layer Parameters:")
print(f"  Weights: {conv_weights_count}, Biases: {conv_bias_count}, Total: {total_conv_params}")
```

```python
print("Fully Connected Layer Parameters:")
print(f"  Weights: {lin_weights_count}, Biases: {lin_bias_count}, Total: {total_lin_params}")
print(f"Total Trainable Parameters in CNN Model: {total_params}")
```

```
Convolutional Layer Parameters:
  Weights: 400, Biases: 16, Total: 416
Fully Connected Layer Parameters:
  Weights: 4000, Biases: 10, Total: 4010
Total Trainable Parameters in CNN Model: 4426
```

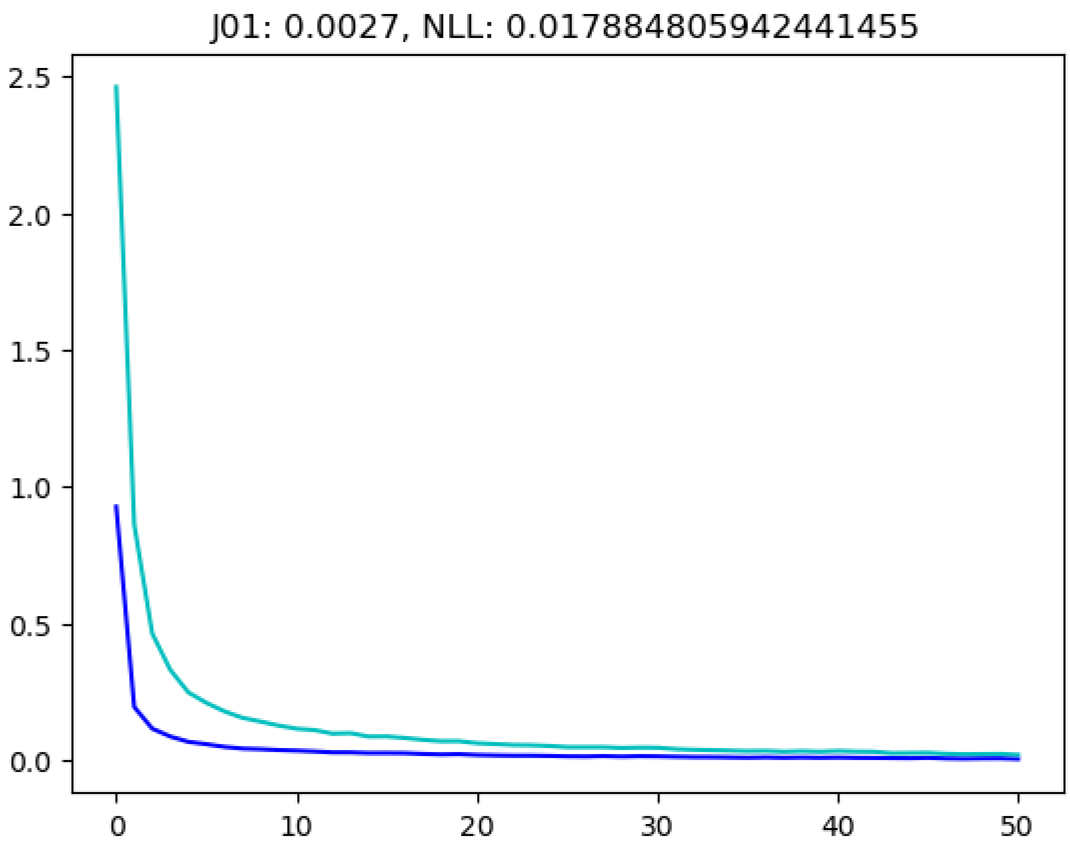
Problem 3.2: Training the model (10 points)

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 [71]:

```python
# Define and initialize the model
model = myConvNet()

# Train the model on X_tr and y_tr for 50 epochs with learning rate 0.001
model.fit(X_tr, y_tr, batch_size=256, max_iter=50, learning_rate_init=0.001, plot=True)
```



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 [105…

```
# Evaluate on the training set
cnn_train_error = accuracy_score(y_te, model.predict(X_te))
mlp_train_error = accuracy_score(y_te, final_model.predict(X_te))

# Evaluate on the validation set
cnn_val_error = accuracy_score(y_val, model.predict(X_val))
```

```
mlp_val_error = accuracy_score(y_val, final_model.predict(X_val))

# Evaluate on the test set
cnn_test_error = accuracy_score(y_te, model.predict(X_te))
mlp_test_error = accuracy_score(y_te, final_model.predict(X_te))

# Print the results
print("CNN training accuracy:",cnn_train_error, "MLP training accuracy:",mlp_train_error)
print("CNN validation accuracy:",cnn_val_error, "MLP validation accuracy:",mlp_val_error)
print("CNN test accuracy:",cnn_test_error, "MLP test accuracy:",mlp_test_error)
```

```
CNN training accuracy: 0.9692857142857143 MLP training accuracy: 0.9524285714285714
CNN validation accuracy: 0.9727 MLP validation accuracy: 0.9549
CNN test accuracy: 0.9692857142857143 MLP test accuracy: 0.9524285714285714
```

DISCUSS: In Problem 2.3, after tuning various hyperparameters for the MLP, the model's performance was likely improved; however, it still did not match the effectiveness of the CNN. It can be observed that the training, validation and test accuracy scores of CNN are all higher than that of MLP. The observed differences in performance between the CNN and the MLP reinforce the notion that architecture plays a crucial role in the model's ability to learn from data. The CNN's ability to extract and generalize features more effectively than the MLP explains its superior performance across training, validation, and test datasets. Further tuning and experimentation with the MLP may yield improvements, but it is clear that convolutional architectures have a significant advantage in this context.

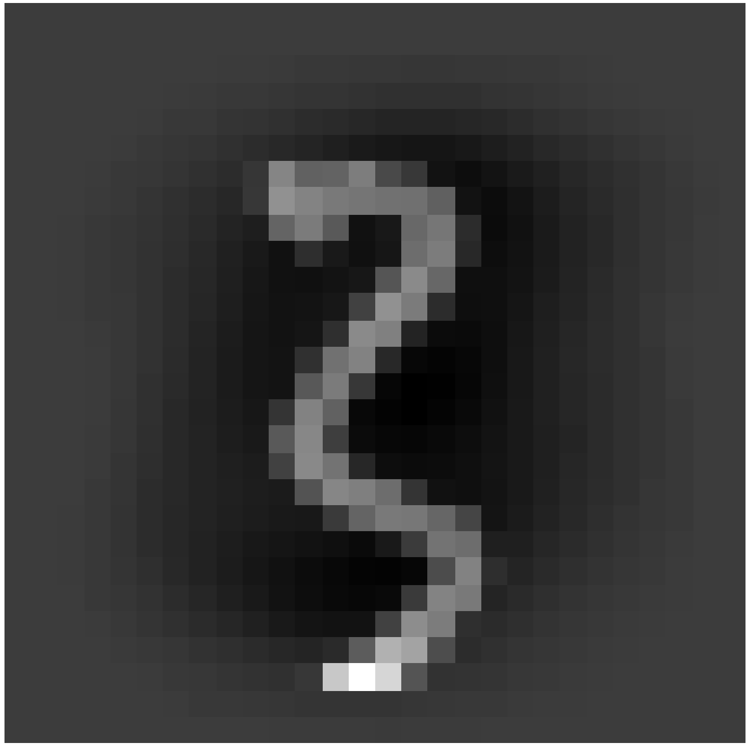
Problem 3.4: Comparing Predictions (5 points)

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 [108…

```
# Display the original image of the data point (assuming X_val is normalized and in 28x28 shape)
plt.imshow(X_val[592].reshape(28, 28), cmap='gray')
plt.title('Data Point X_val[592]')
plt.axis('off')
plt.show()
```

Data Point X_val[592]



In [118…

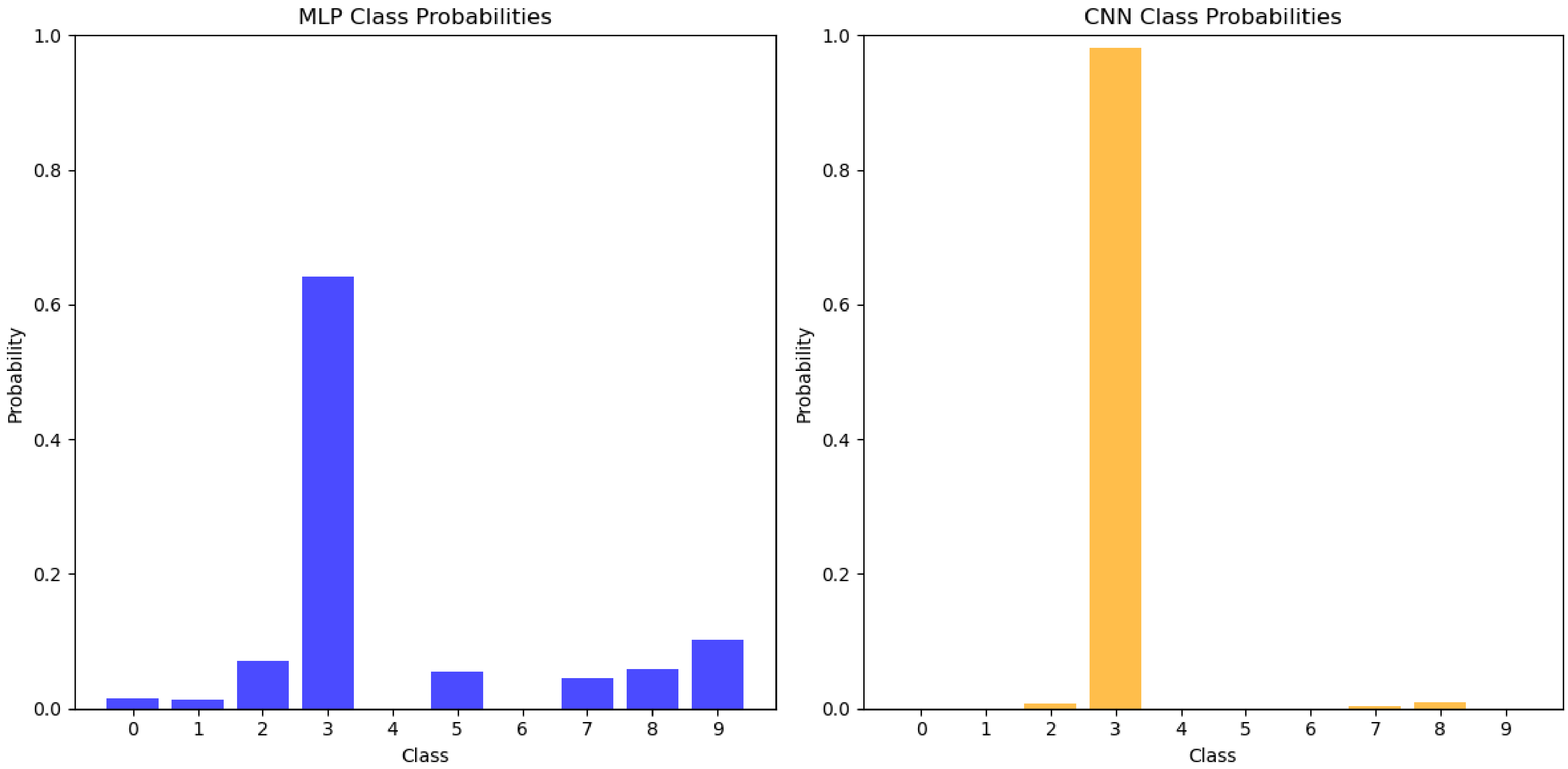
```
# Predict class probabilities with MLPClassifier
mlp_probabilities = final_model.predict_proba(X_val[592].reshape(1, -1))

# Display the probabilities
print("MLP Class Probabilities:", mlp_probabilities)

# Predict class probabilities with the CNN model
input_tensor = torch.tensor(X_val[592].reshape(1, 1, 28, 28), dtype=torch.float64)  # Use float64
cnn_probabilities = model.forward_(input_tensor).detach().numpy()


# Display the probabilities
print("CNN Class Probabilities:", cnn_probabilities)
```

```
MLP Class Probabilities: [[1.45533210e-02 1.25133678e-02 7.04971105e-02 6.41063555e-01
  2.64480871e-08 5.41761914e-02 1.70858028e-05 4.57209902e-02
  5.93498331e-02 1.02108519e-01]]
CNN Class Probabilities: [[3.17587422e-05 3.78560678e-05 7.48372576e-03 9.81449588e-01
  1.58121291e-06 3.02124551e-05 4.58477871e-05 2.46252247e-03
  8.43084107e-03 2.60663799e-05]]
```

In [122…

```python
mlp_predicted_class = np.argmax(mlp_probabilities)
cnn_predicted_class = np.argmax(cnn_probabilities)

print("MLP Predicted Class:", mlp_predicted_class)
print("CNN Predicted Class:", cnn_predicted_class)

# Compare uncertainty by plotting the probabilities of all classes
classes = np.arange(10)
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.bar(classes, mlp_probabilities.flatten(), color='blue', alpha=0.7)
plt.xticks(classes)
plt.title('MLP Class Probabilities')
plt.xlabel('Class')
plt.ylabel('Probability')
plt.ylim(0, 1)

plt.subplot(1, 2, 2)
plt.bar(classes, cnn_probabilities.flatten(), color='orange', alpha=0.7)
plt.xticks(classes)
plt.title('CNN Class Probabilities')
plt.xlabel('Class')
plt.ylabel('Probability')
plt.ylim(0, 1)

plt.tight_layout()
plt.show()
```

```
MLP Predicted Class: 3
CNN Predicted Class: 3
```



DISCUSS: The results are plausible. For the MLP model, the probabilities of classified as 2, 5, and 9 are non-negligible, because the features of the data points corresponding to these classes are similar. By visualizing the classification probabilities output by the two models, it is evident that the uncertainty of the CNN model is lower because its probability distribution is steeper (i.e., the peak of one category is more pronounced while the probabilities of other categories are lower), indicating a higher level of reliability in its predictions. On the other hand, MLP probabilities are more evenly distributed among several classes, it suggests higher uncertainty.

Generally, CNNs tend to perform better on image data due to their ability to capture spatial hierarchies and local patterns. The CNN shows lower uncertainty (more concentrated probability distribution) compared to the MLP for the same ambiguous data point, it could imply that the CNN is leveraging its architecture to better understand the image features.

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.

This assignment was completed independently by me.