MLE Exercise 6

1. Convolutions of Continuous and Discrete Variables

1(a) Adding two Gaussians

Let
$$X \sim \mathcal{N}(\mu_1, \sigma_1^2), \qquad Y \sim \mathcal{N}(\mu_2, \sigma_2^2), \qquad Z = X + Y,$$
with X and Y *independent*.

Because of independence we can write the pdf of Z as a convolution of pdfs of X and Y:
$$f_Z(z) = (f_X * f_Y)(z) = \int_{-\infty}^{\infty} f_X(x)\, f_Y(z-x)\, dx.$$
This comes from the definition of convolution.

Both f_X and f_Y have the familiar bell-curve form
$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\Big[-\frac{(x-\mu_1)^2}{2\sigma_1^2}\Big], \quad f_Y(y) = \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\Big[-\frac{(y-\mu_2)^2}{2\sigma_2^2}\Big].$$
Substituting $y = z - x$ for the second one gives
$$f_Z(z) \;=\; \frac{1}{2\pi\sigma_1\sigma_2} \int_{-\infty}^{\infty} \exp\Big[-\frac{(x-\mu_1)^2}{2\sigma_1^2} - \frac{(z-x-\mu_2)^2}{2\sigma_2^2}\Big] dx.$$

Completing the square: The exponent is a quadratic in x plus some terms that do *not* depend on x. We collect the quadratic terms and rewrite them as a single perfect square:
$$-\frac{1}{2}\Big[\frac{(x-\mu_1)^2}{\sigma_1^2} + \frac{(z-x-\mu_2)^2}{\sigma_2^2}\Big] \;=\; -\frac{(x-\mu_\star)^2}{2\Sigma^2} \;-\; \frac{\big(z-(\mu_1+\mu_2)\big)^2}{2(\sigma_1^2+\sigma_2^2)},$$
where
$$\Sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2+\sigma_2^2}, \qquad \mu_\star = \Sigma^2\Big(\frac{\mu_1}{\sigma_1^2} + \frac{z-\mu_2}{\sigma_2^2}\Big).$$

The factor $e^{-(x-\mu_\star)^2/(2\Sigma^2)}$ integrates to $\sqrt{2\pi\Sigma^2}$ (the standard Gaussian integral), so we are left with

$$f_Z(z) = \frac{1}{\sqrt{2\pi(\sigma_1^2+\sigma_2^2)}} \exp\Big[-\frac{(z-(\mu_1+\mu_2))^2}{2(\sigma_1^2+\sigma_2^2)}\Big].$$

This *is* a Gaussian pdf, with mean $\mu_1+\mu_2$ and variance $\sigma_1^2+\sigma_2^2$. Therefore the final result:

$$\boxed{Z \sim \mathcal{N}(\mu_1+\mu_2,\ \sigma_1^2+\sigma_2^2)}.$$

1(b) Output size of a 2-D convolution layer

Problem: Given

- input image size (H, W),
- kernel / filter size (K_H, K_W),
- *zero-padding* of p pixels on *each* edge, and
- stride s (same in both spatial dimensions),

derive the height H_{out} and width W_{out} of the resulting feature map.

Answer: Padding inflates the useful image region to $(H+2p)\times(W+2p)$. With stride s, the kernel's top-left corner "jumps" by s pixels each step.

One dimension first. Along, the vertical axis, the kernel can land at positions.

$$0,\ s,\ 2s,\ \ldots,\ \big(H_{\text{eff}}-K_H\big), \quad \text{where } H_{\text{eff}} = H+2p.$$

Counting how many multiples of s fit into that range gives

$$\Big\lfloor \frac{H_{\text{eff}}-K_H}{s} \Big\rfloor + 1.$$

Exactly the same argument applies horizontally.

Final formulae:

$$\boxed{H_{\text{out}} = \Big\lfloor \frac{H+2p-K_H}{s} \Big\rfloor + 1}, \qquad \boxed{W_{\text{out}} = \Big\lfloor \frac{W+2p-K_W}{s} \Big\rfloor + 1}.$$

Special-cases:

- *"Valid" convolution* ($p=0,\ s=1$): $H_{\text{out}} = H-K_H+1$, $W_{\text{out}} = W-K_W+1$.
- *"Same" convolution* (choose $p=\frac{K_H-1}{2}$, K odd, $s=1$): $H_{\text{out}}=H$ etc. Nice consistency check!
- *Down-sampling* ($s>1$) visibly reduces the size.

Machine Learning Essentials SS25 - Exercise Sheet 6

Instructions

- `TODO` 's indicate where you need to complete the implementations.
- You may use external resources, but **write your own solutions**.
- Provide concise, but comprehensible comments to explain what your code does.
- Code that's unnecessarily extensive and/or not well commented will not be scored.

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import torch as tc
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torchsummary import summary

np.random.seed(42)
tc.manual_seed(42)

device = tc.device("cuda" if tc.cuda.is_available() else "cpu")
```

Exercise 2 - CNN Classifier

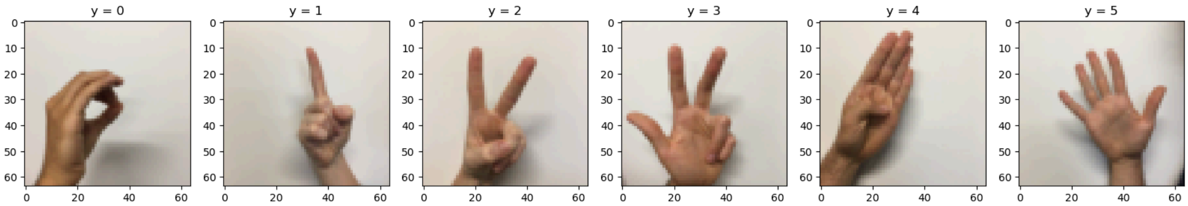
The SIGNS dataset is a collection of 6 signs representing numbers from 0 to 5. We first load the data and have the shapes printed out. The split into train, validation and test set has already been carried out.

In [2]:
```
# Load the dataset
X_train = np.load('sign_data/X_train.npy')
Y_train = np.load('sign_data/Y_train.npy')
X_val = np.load('sign_data/X_val.npy')
Y_val = np.load('sign_data/Y_val.npy')
X_test = np.load('sign_data/X_test.npy')
Y_test = np.load('sign_data/Y_test.npy')

# print the shape of the dataset
print("X_train shape: " + str(X_train.shape))
print("Y_train shape: " + str(Y_train.shape))
print("X_val shape: " + str(X_val.shape))
print("Y_val shape: " + str(Y_val.shape))
print("X_test shape: " + str(X_test.shape))
print("Y_test shape: " + str(Y_test.shape)+"\n")
print("classes: " + str(np.unique(Y_train)))

# check if classes are balanced
print("Counts of classes in Y_train: " + str(np.unique(Y_train, return_count
print("Counts of classes in Y_val: " + str(np.unique(Y_val, return_counts=Tr
print("Counts of classes in Y_test: " + str(np.unique(Y_test, return_counts=
```

```
X_train shape: (960, 64, 64, 3)
Y_train shape: (960,)
X_val shape: (120, 64, 64, 3)
Y_val shape: (120,)
X_test shape: (120, 64, 64, 3)
Y_test shape: (120,)

classes: [0 1 2 3 4 5]
Counts of classes in Y_train: [160 160 160 160 160 160]
Counts of classes in Y_val: [20 20 20 20 20 20]
Counts of classes in Y_test: [20 20 20 20 20 20]
```

The classes are balanced so that accuracy is an appropriate measure for evaluating a classifier. We next visualize an instance of each class.

In [3]:
```python
fig, axs = plt.subplots(1, 6, figsize=(20, 10))
for i in range(6):
    # get indices where the label is i
    idx = np.where(Y_train == i)[0][0]
    axs[i].imshow(X_train[idx])
    axs[i].set_title("y = " + str(i))
```



Pixels in each channel (RGB) of the images take values in the range [0, 255]. However, it is desirable to have absolute values in the range [0, 1] as input for neural network architectures to avoid exploding or vanishing gradient problems. Through the following cell, we apply a simple data scaling procedure: we divide the values of the pixels by 255. As an alternative, you can use the `StandardScaler()` function of the scikit-learn library.

In [4]:
```python
X_train = X_train/255
X_val = X_val/255
X_test = X_test/255
```

Task 1

Use pytorch to build the model. Take a look at the documentation for an introduction, a detailed tutorial, for example for classifiers, can be found here.

Implement the following architecture:

```
 - Conv2d: 4 output channels, 3 by 3 filter size, stride 1,
padding "same"
 - BatchNorm2d: 4 output channels
 - ReLU activation
 - MaxPool2d: 2 by 2 filter size, stride 2, padding 0
 - Conv2d: 8 output channels, 3 by 3 filter size, stride 1,
padding "same"
 - BatchNorm2d: 8 output channels
 - ReLU activation
```

```
- MaxPool2d: Use a 2 by 2 filter size, stride 2, padding 0
- Flatten the previous output
- Linear: 64 output neurons
- ReLu activation function
- Linear: 6 output neurons
- LogSoftmax
```

We use the LogSoftmax here instead of the Softmax for computational reasons. Accordingly, the loss function is not CrossEntropyLoss but NLLLoss. When flattening, be careful not to do it with the batch dimension but only with the height, width and channel dimension.

In [5]:
```python
class CNN_Classifier(nn.Module):
    def __init__(self):
        super().__init__()

        # TODO: Initialize the layers of the CNN

        # First convolutional block
        self.conv1 = nn.Conv2d(3, 4, kernel_size=3, stride=1, padding='same'
        self.bn1 = nn.BatchNorm2d(4)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        # Second convolutional block
        self.conv2 = nn.Conv2d(4, 8, kernel_size=3, stride=1, padding='same'
        self.bn2 = nn.BatchNorm2d(8)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        # Flatten layer
        self.flatten = nn.Flatten()

        # Fully connected layers
        # After two 2x2 max pooling operations, 64x64 -> 32x32 -> 16x16
        # So we have 8 channels * 16 * 16 = 2048 features
        self.fc1 = nn.Linear(8 * 16 * 16, 64)
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(64, 6)
        self.logsoftmax = nn.LogSoftmax(dim=1)


    def forward(self, X):
        # TODO: Implement the forward pass
        # First convolutional block
        X = self.conv1(X)
        X = self.bn1(X)
        X = self.relu1(X)
        X = self.pool1(X)

        # Second convolutional block
        X = self.conv2(X)
        X = self.bn2(X)
        X = self.relu2(X)
        X = self.pool2(X)

        # Flatten and fully connected layers
        X = self.flatten(X)
```

```
        X = self.fc1(X)
        X = self.relu3(X)
        X = self.fc2(X)
        X = self.logsoftmax(X)

        return X
```

To test your model you can foward some random numbers. The shape of the output should be (2, 6).

In [6]:
```
cnn_model = CNN_Classifier()
# dummy sample of batch size 2
X_random = tc.randn(2, 3, 64, 64)
output = cnn_model(X_random)

print("Output shape: " + str(output.shape))
```

```
Output shape: torch.Size([2, 6])
```

torchsummary.summary provides a nice overview of the model and the number of learnable parameters:

In [7]:
```
summary(cnn_model, input_size=(3, 64, 64), device="cpu")
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            112
├─BatchNorm2d: 1-2                       8
├─ReLU: 1-3                              --
├─MaxPool2d: 1-4                         --
├─Conv2d: 1-5                            296
├─BatchNorm2d: 1-6                       16
├─ReLU: 1-7                              --
├─MaxPool2d: 1-8                         --
├─Flatten: 1-9                           --
├─Linear: 1-10                           131,136
├─ReLU: 1-11                             --
├─Linear: 1-12                           390
├─LogSoftmax: 1-13                       --
=================================================================
Total params: 131,958
Trainable params: 131,958
Non-trainable params: 0
=================================================================
```

Out[7]:
```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            112
├─BatchNorm2d: 1-2                       8
├─ReLU: 1-3                              --
├─MaxPool2d: 1-4                         --
├─Conv2d: 1-5                            296
├─BatchNorm2d: 1-6                       16
├─ReLU: 1-7                              --
├─MaxPool2d: 1-8                         --
├─Flatten: 1-9                           --
├─Linear: 1-10                           131,136
├─ReLU: 1-11                             --
├─Linear: 1-12                           390
├─LogSoftmax: 1-13                       --
=================================================================
Total params: 131,958
Trainable params: 131,958
Non-trainable params: 0
=================================================================
```

Task 2

DataLoaders wrap around Datasets to provide efficient data batching, shuffling, and parallel loading during model training or inference. To define a custom dataset we must implement three functions: **init**, **len** and **get_item**. While **len** defines the length of the dataset and thus the number of batches in the dataloader, **get_item** can be used to get a single sample through the index.

In [8]:
```python
class Image_Dataset(Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    def __len__(self):
        # TODO: Return the length of the dataset
        return len(self.X)


    def __getitem__(self, idx):
        # TODO: Return the image as a float tensor and the label as a long t
        # Images need to be transposed from (H, W, C) to (C, H, W) for PyTor
        image = tc.tensor(self.X[idx].transpose(2, 0, 1), dtype=tc.float32)
        label = tc.tensor(self.Y[idx], dtype=tc.long)
        return image, label
```

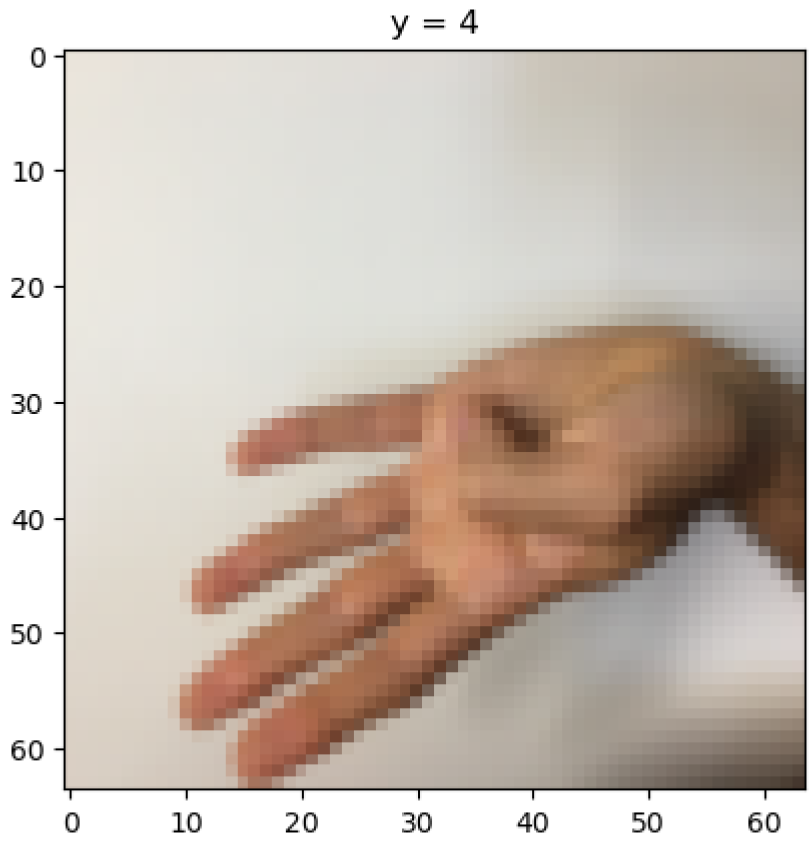
In [9]:
```python
train_batch_size = 64
val_batch_size = len(Y_val)
test_batch_size = len(Y_test)

# TODO: Create the dataset and dataloader
train_dataset = Image_Dataset(X_train, Y_train)
val_dataset = Image_Dataset(X_val, Y_val)
test_dataset = Image_Dataset(X_test, Y_test)

train_loader = DataLoader(train_dataset, batch_size=train_batch_size, shuffl
```

```
val_loader = DataLoader(val_dataset, batch_size=val_batch_size, shuffle=Fals
test_loader = DataLoader(test_dataset, batch_size=test_batch_size, shuffle=F
```

To make sure that everything has worked properly, we take a sample of the data_loader and visualize it.

In [10]:
```
sample_X, sample_Y = next(iter(train_loader))
plt.imshow(sample_X[0].T)
plt.title("y = " + str(int(sample_Y[0].item())))
plt.show()
```

```
/var/folders/gl/m711nqcx42d81f7zr656_v600000gn/T/ipykernel_1541/2461653407.
py:2: UserWarning: The use of `x.T` on tensors of dimension other than 2 to
reverse their shape is deprecated and it will throw an error in a future re
lease. Consider `x.mT` to transpose batches of matrices or `x.permute(*torc
h.arange(x.ndim - 1, -1, -1))` to reverse the dimensions of a tensor. (Trig
gered internally at /Users/runner/work/pytorch/pytorch/pytorch/aten/src/ATe
n/native/TensorShape.cpp:4416.)
  plt.imshow(sample_X[0].T)
```



Task 3

Implement the training loop. Use the negative log-likelihood loss (NLLLoss) and the Adam optimizer. Be sure to zero the gradients after each optimization step to avoid accumulating contributions from previous epochs and batches.

In [13]:
```
def train_cnn(model, train_loader, val_loader, lr, n_epochs, device):
    model = model.to(device)

    # TODO: Initialize the optimizer and loss function
    loss_function = nn.NLLLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
```

```
    train_loss = np.zeros(n_epochs)
    val_loss = np.zeros(n_epochs)
    train_acc = np.zeros(n_epochs)
    val_acc = np.zeros(n_epochs)

    for epoch in range(1, n_epochs + 1):
        model.train()

        epoch_loss = 0

        for X, Y in train_loader:
            X, Y = X.to(device), Y.to(device)
            # TODO: Implement the training loop
            optimizer.zero_grad()
            output = model(X)
            loss = loss_function(output, Y)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()/len(train_loader)

        train_loss[epoch - 1] = epoch_loss
        train_acc[epoch - 1] = (output.argmax(dim=1) == Y).float().mean().it

        model.eval()

        with tc.no_grad():
            X, Y = next(iter(val_loader))
            X, Y = X.to(device), Y.to(device)
            # TODO: Implement the evaluation step
            output = model(X)
            loss = loss_function(output, Y)

            val_loss[epoch - 1] = loss.item()
            val_acc[epoch - 1] = (output.argmax(dim=1) == Y).float().mean().
        print(f"Epoch {epoch}/{n_epochs} - Train Loss: {epoch_loss:.4f}, Tes

    return train_loss, val_loss, train_acc, val_acc
```

In [19]:
```
n_epochs = 50
# TODO: Train the model with different learning rates
learning_rates = [1e-1, 1e-3, 1e-4, 1e-5]
results = {}

for lr in learning_rates:
    print(f"\nTraining with learning rate: {lr}")
    model = CNN_Classifier()
    train_loss, val_loss, train_acc, val_acc = train_cnn(
        model, train_loader, val_loader, lr, n_epochs, device
    )
    results[lr] = {
        'model': model,
        'train_loss': train_loss,
        'val_loss': val_loss,
        'train_acc': train_acc,
        'val_acc': val_acc
    }
```

```
Training with learning rate: 0.1
Epoch 1/50 - Train Loss: 24.1252, Test Loss: 1.8230
Epoch 2/50 - Train Loss: 1.8122, Test Loss: 2.5632
Epoch 3/50 - Train Loss: 1.7964, Test Loss: 1.7932
Epoch 4/50 - Train Loss: 1.7949, Test Loss: 1.7925
Epoch 5/50 - Train Loss: 1.7954, Test Loss: 1.7921
Epoch 6/50 - Train Loss: 1.7937, Test Loss: 1.7921
Epoch 7/50 - Train Loss: 1.7929, Test Loss: 1.7920
Epoch 8/50 - Train Loss: 1.7954, Test Loss: 1.7922
Epoch 9/50 - Train Loss: 1.7939, Test Loss: 1.7926
Epoch 10/50 - Train Loss: 1.7943, Test Loss: 1.7919
Epoch 11/50 - Train Loss: 1.7937, Test Loss: 1.7921
Epoch 12/50 - Train Loss: 1.7970, Test Loss: 1.7923
Epoch 13/50 - Train Loss: 1.7968, Test Loss: 1.7943
Epoch 14/50 - Train Loss: 1.7947, Test Loss: 1.7928
Epoch 15/50 - Train Loss: 1.7987, Test Loss: 1.7925
Epoch 16/50 - Train Loss: 1.7989, Test Loss: 1.7931
Epoch 17/50 - Train Loss: 1.7977, Test Loss: 1.7933
Epoch 18/50 - Train Loss: 1.7964, Test Loss: 1.7941
Epoch 19/50 - Train Loss: 1.7954, Test Loss: 1.7923
Epoch 20/50 - Train Loss: 1.7964, Test Loss: 1.7931
Epoch 21/50 - Train Loss: 1.7951, Test Loss: 1.7925
Epoch 22/50 - Train Loss: 1.7975, Test Loss: 1.7927
Epoch 23/50 - Train Loss: 1.7972, Test Loss: 1.7931
Epoch 24/50 - Train Loss: 1.7990, Test Loss: 1.7928
Epoch 25/50 - Train Loss: 1.7964, Test Loss: 1.7927
Epoch 26/50 - Train Loss: 1.7984, Test Loss: 1.7924
Epoch 27/50 - Train Loss: 1.7955, Test Loss: 1.7934
Epoch 28/50 - Train Loss: 1.7944, Test Loss: 1.7921
Epoch 29/50 - Train Loss: 1.7954, Test Loss: 1.7922
Epoch 30/50 - Train Loss: 1.7959, Test Loss: 1.7926
Epoch 31/50 - Train Loss: 1.7952, Test Loss: 1.7923
Epoch 32/50 - Train Loss: 1.7985, Test Loss: 1.7932
Epoch 33/50 - Train Loss: 1.7981, Test Loss: 1.7926
Epoch 34/50 - Train Loss: 1.7973, Test Loss: 1.7928
Epoch 35/50 - Train Loss: 1.7980, Test Loss: 1.7929
Epoch 36/50 - Train Loss: 1.7945, Test Loss: 1.7925
Epoch 37/50 - Train Loss: 1.7952, Test Loss: 1.7922
Epoch 38/50 - Train Loss: 1.7949, Test Loss: 1.7925
Epoch 39/50 - Train Loss: 1.7963, Test Loss: 1.7920
Epoch 40/50 - Train Loss: 1.7984, Test Loss: 1.7933
Epoch 41/50 - Train Loss: 1.7975, Test Loss: 1.7931
Epoch 42/50 - Train Loss: 1.7964, Test Loss: 1.7931
Epoch 43/50 - Train Loss: 1.8002, Test Loss: 1.7926
Epoch 44/50 - Train Loss: 1.7954, Test Loss: 1.7929
Epoch 45/50 - Train Loss: 1.7953, Test Loss: 1.7922
Epoch 46/50 - Train Loss: 1.7966, Test Loss: 1.7921
Epoch 47/50 - Train Loss: 1.7955, Test Loss: 1.7936
Epoch 48/50 - Train Loss: 1.7958, Test Loss: 1.7926
Epoch 49/50 - Train Loss: 1.7955, Test Loss: 1.7919
Epoch 50/50 - Train Loss: 1.7962, Test Loss: 1.7920

Training with learning rate: 0.001
Epoch 1/50 - Train Loss: 1.5027, Test Loss: 1.7479
Epoch 2/50 - Train Loss: 0.8677, Test Loss: 2.1267
Epoch 3/50 - Train Loss: 0.5151, Test Loss: 1.4809
Epoch 4/50 - Train Loss: 0.3310, Test Loss: 0.8321
Epoch 5/50 - Train Loss: 0.2247, Test Loss: 0.4975
Epoch 6/50 - Train Loss: 0.1585, Test Loss: 0.6596
Epoch 7/50 - Train Loss: 0.1109, Test Loss: 0.4652
Epoch 8/50 - Train Loss: 0.0775, Test Loss: 0.3986
```

```
Epoch 9/50 - Train Loss: 0.0662, Test Loss: 0.3639
Epoch 10/50 - Train Loss: 0.0493, Test Loss: 0.3199
Epoch 11/50 - Train Loss: 0.0333, Test Loss: 0.3288
Epoch 12/50 - Train Loss: 0.0271, Test Loss: 0.3127
Epoch 13/50 - Train Loss: 0.0235, Test Loss: 0.3848
Epoch 14/50 - Train Loss: 0.0193, Test Loss: 0.2970
Epoch 15/50 - Train Loss: 0.0154, Test Loss: 0.2904
Epoch 16/50 - Train Loss: 0.0134, Test Loss: 0.3685
Epoch 17/50 - Train Loss: 0.0115, Test Loss: 0.3187
Epoch 18/50 - Train Loss: 0.0098, Test Loss: 0.3201
Epoch 19/50 - Train Loss: 0.0094, Test Loss: 0.3142
Epoch 20/50 - Train Loss: 0.0081, Test Loss: 0.3089
Epoch 21/50 - Train Loss: 0.0069, Test Loss: 0.3160
Epoch 22/50 - Train Loss: 0.0065, Test Loss: 0.3368
Epoch 23/50 - Train Loss: 0.0059, Test Loss: 0.3100
Epoch 24/50 - Train Loss: 0.0051, Test Loss: 0.3254
Epoch 25/50 - Train Loss: 0.0050, Test Loss: 0.3541
Epoch 26/50 - Train Loss: 0.0044, Test Loss: 0.3155
Epoch 27/50 - Train Loss: 0.0040, Test Loss: 0.3286
Epoch 28/50 - Train Loss: 0.0038, Test Loss: 0.3138
Epoch 29/50 - Train Loss: 0.0035, Test Loss: 0.3391
Epoch 30/50 - Train Loss: 0.0032, Test Loss: 0.3292
Epoch 31/50 - Train Loss: 0.0031, Test Loss: 0.3274
Epoch 32/50 - Train Loss: 0.0027, Test Loss: 0.3349
Epoch 33/50 - Train Loss: 0.0025, Test Loss: 0.3404
Epoch 34/50 - Train Loss: 0.0024, Test Loss: 0.3488
Epoch 35/50 - Train Loss: 0.0022, Test Loss: 0.3329
Epoch 36/50 - Train Loss: 0.0021, Test Loss: 0.3436
Epoch 37/50 - Train Loss: 0.0021, Test Loss: 0.3419
Epoch 38/50 - Train Loss: 0.0020, Test Loss: 0.3432
Epoch 39/50 - Train Loss: 0.0018, Test Loss: 0.3441
Epoch 40/50 - Train Loss: 0.0018, Test Loss: 0.3329
Epoch 41/50 - Train Loss: 0.0017, Test Loss: 0.3477
Epoch 42/50 - Train Loss: 0.0016, Test Loss: 0.3502
Epoch 43/50 - Train Loss: 0.0015, Test Loss: 0.3579
Epoch 44/50 - Train Loss: 0.0014, Test Loss: 0.3431
Epoch 45/50 - Train Loss: 0.0014, Test Loss: 0.3438
Epoch 46/50 - Train Loss: 0.0013, Test Loss: 0.3587
Epoch 47/50 - Train Loss: 0.0013, Test Loss: 0.3469
Epoch 48/50 - Train Loss: 0.0012, Test Loss: 0.3517
Epoch 49/50 - Train Loss: 0.0011, Test Loss: 0.3418
Epoch 50/50 - Train Loss: 0.0011, Test Loss: 0.3639

Training with learning rate: 0.0001
Epoch 1/50 - Train Loss: 1.7334, Test Loss: 1.7869
Epoch 2/50 - Train Loss: 1.5546, Test Loss: 1.7271
Epoch 3/50 - Train Loss: 1.3683, Test Loss: 1.5448
Epoch 4/50 - Train Loss: 1.1951, Test Loss: 1.3224
Epoch 5/50 - Train Loss: 1.0445, Test Loss: 1.1279
Epoch 6/50 - Train Loss: 0.9235, Test Loss: 1.0233
Epoch 7/50 - Train Loss: 0.8341, Test Loss: 0.9288
Epoch 8/50 - Train Loss: 0.7452, Test Loss: 0.8967
Epoch 9/50 - Train Loss: 0.6831, Test Loss: 0.8141
Epoch 10/50 - Train Loss: 0.6313, Test Loss: 0.7902
Epoch 11/50 - Train Loss: 0.5789, Test Loss: 0.7342
Epoch 12/50 - Train Loss: 0.5332, Test Loss: 0.7192
Epoch 13/50 - Train Loss: 0.4916, Test Loss: 0.7040
Epoch 14/50 - Train Loss: 0.4625, Test Loss: 0.6450
Epoch 15/50 - Train Loss: 0.4298, Test Loss: 0.6246
Epoch 16/50 - Train Loss: 0.4055, Test Loss: 0.6364
Epoch 17/50 - Train Loss: 0.3709, Test Loss: 0.6077
```

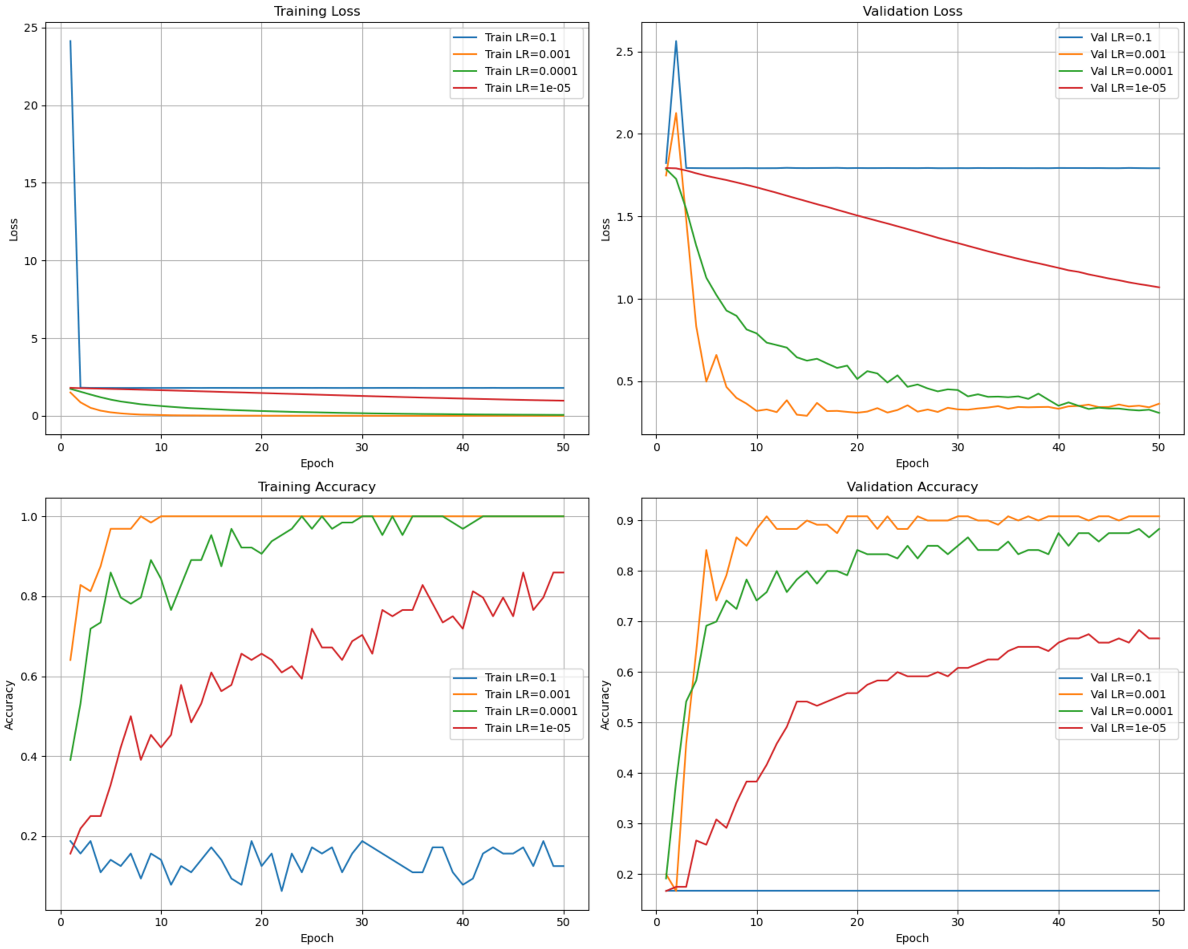
```
Epoch 18/50 - Train Loss: 0.3521, Test Loss: 0.5807
Epoch 19/50 - Train Loss: 0.3291, Test Loss: 0.5946
Epoch 20/50 - Train Loss: 0.3072, Test Loss: 0.5136
Epoch 21/50 - Train Loss: 0.2897, Test Loss: 0.5611
Epoch 22/50 - Train Loss: 0.2726, Test Loss: 0.5471
Epoch 23/50 - Train Loss: 0.2540, Test Loss: 0.4923
Epoch 24/50 - Train Loss: 0.2396, Test Loss: 0.5357
Epoch 25/50 - Train Loss: 0.2292, Test Loss: 0.4661
Epoch 26/50 - Train Loss: 0.2144, Test Loss: 0.4804
Epoch 27/50 - Train Loss: 0.2004, Test Loss: 0.4563
Epoch 28/50 - Train Loss: 0.1872, Test Loss: 0.4384
Epoch 29/50 - Train Loss: 0.1768, Test Loss: 0.4510
Epoch 30/50 - Train Loss: 0.1665, Test Loss: 0.4464
Epoch 31/50 - Train Loss: 0.1553, Test Loss: 0.4082
Epoch 32/50 - Train Loss: 0.1487, Test Loss: 0.4207
Epoch 33/50 - Train Loss: 0.1415, Test Loss: 0.4053
Epoch 34/50 - Train Loss: 0.1312, Test Loss: 0.4066
Epoch 35/50 - Train Loss: 0.1248, Test Loss: 0.4030
Epoch 36/50 - Train Loss: 0.1164, Test Loss: 0.4082
Epoch 37/50 - Train Loss: 0.1119, Test Loss: 0.3930
Epoch 38/50 - Train Loss: 0.1070, Test Loss: 0.4247
Epoch 39/50 - Train Loss: 0.0995, Test Loss: 0.3874
Epoch 40/50 - Train Loss: 0.0951, Test Loss: 0.3513
Epoch 41/50 - Train Loss: 0.0884, Test Loss: 0.3714
Epoch 42/50 - Train Loss: 0.0843, Test Loss: 0.3510
Epoch 43/50 - Train Loss: 0.0810, Test Loss: 0.3316
Epoch 44/50 - Train Loss: 0.0774, Test Loss: 0.3399
Epoch 45/50 - Train Loss: 0.0737, Test Loss: 0.3348
Epoch 46/50 - Train Loss: 0.0697, Test Loss: 0.3343
Epoch 47/50 - Train Loss: 0.0679, Test Loss: 0.3268
Epoch 48/50 - Train Loss: 0.0633, Test Loss: 0.3227
Epoch 49/50 - Train Loss: 0.0600, Test Loss: 0.3273
Epoch 50/50 - Train Loss: 0.0573, Test Loss: 0.3080

Training with learning rate: 1e-05
Epoch 1/50 - Train Loss: 1.8014, Test Loss: 1.7932
Epoch 2/50 - Train Loss: 1.7795, Test Loss: 1.7912
Epoch 3/50 - Train Loss: 1.7627, Test Loss: 1.7782
Epoch 4/50 - Train Loss: 1.7471, Test Loss: 1.7612
Epoch 5/50 - Train Loss: 1.7318, Test Loss: 1.7459
Epoch 6/50 - Train Loss: 1.7161, Test Loss: 1.7328
Epoch 7/50 - Train Loss: 1.6995, Test Loss: 1.7202
Epoch 8/50 - Train Loss: 1.6827, Test Loss: 1.7059
Epoch 9/50 - Train Loss: 1.6662, Test Loss: 1.6912
Epoch 10/50 - Train Loss: 1.6492, Test Loss: 1.6760
Epoch 11/50 - Train Loss: 1.6314, Test Loss: 1.6594
Epoch 12/50 - Train Loss: 1.6137, Test Loss: 1.6429
Epoch 13/50 - Train Loss: 1.5953, Test Loss: 1.6251
Epoch 14/50 - Train Loss: 1.5766, Test Loss: 1.6080
Epoch 15/50 - Train Loss: 1.5594, Test Loss: 1.5910
Epoch 16/50 - Train Loss: 1.5405, Test Loss: 1.5736
Epoch 17/50 - Train Loss: 1.5220, Test Loss: 1.5576
Epoch 18/50 - Train Loss: 1.5039, Test Loss: 1.5398
Epoch 19/50 - Train Loss: 1.4848, Test Loss: 1.5226
Epoch 20/50 - Train Loss: 1.4661, Test Loss: 1.5052
Epoch 21/50 - Train Loss: 1.4479, Test Loss: 1.4891
Epoch 22/50 - Train Loss: 1.4287, Test Loss: 1.4725
Epoch 23/50 - Train Loss: 1.4101, Test Loss: 1.4565
Epoch 24/50 - Train Loss: 1.3911, Test Loss: 1.4397
Epoch 25/50 - Train Loss: 1.3718, Test Loss: 1.4228
Epoch 26/50 - Train Loss: 1.3525, Test Loss: 1.4056
```

```
Epoch 27/50 - Train Loss: 1.3337, Test Loss: 1.3880
Epoch 28/50 - Train Loss: 1.3151, Test Loss: 1.3701
Epoch 29/50 - Train Loss: 1.2963, Test Loss: 1.3535
Epoch 30/50 - Train Loss: 1.2793, Test Loss: 1.3377
Epoch 31/50 - Train Loss: 1.2607, Test Loss: 1.3212
Epoch 32/50 - Train Loss: 1.2428, Test Loss: 1.3046
Epoch 33/50 - Train Loss: 1.2248, Test Loss: 1.2880
Epoch 34/50 - Train Loss: 1.2073, Test Loss: 1.2726
Epoch 35/50 - Train Loss: 1.1909, Test Loss: 1.2575
Epoch 36/50 - Train Loss: 1.1737, Test Loss: 1.2426
Epoch 37/50 - Train Loss: 1.1570, Test Loss: 1.2283
Epoch 38/50 - Train Loss: 1.1415, Test Loss: 1.2152
Epoch 39/50 - Train Loss: 1.1255, Test Loss: 1.2013
Epoch 40/50 - Train Loss: 1.1100, Test Loss: 1.1874
Epoch 41/50 - Train Loss: 1.0952, Test Loss: 1.1728
Epoch 42/50 - Train Loss: 1.0806, Test Loss: 1.1631
Epoch 43/50 - Train Loss: 1.0661, Test Loss: 1.1481
Epoch 44/50 - Train Loss: 1.0515, Test Loss: 1.1359
Epoch 45/50 - Train Loss: 1.0369, Test Loss: 1.1232
Epoch 46/50 - Train Loss: 1.0242, Test Loss: 1.1123
Epoch 47/50 - Train Loss: 1.0107, Test Loss: 1.0999
Epoch 48/50 - Train Loss: 0.9981, Test Loss: 1.0892
Epoch 49/50 - Train Loss: 0.9854, Test Loss: 1.0797
Epoch 50/50 - Train Loss: 0.9729, Test Loss: 1.0696
```

In [20]:

```python
# TODO: Visualize the results
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

for lr in learning_rates:
    epochs = range(1, n_epochs + 1)

    # Plot losses
    axes[0, 0].plot(epochs, results[lr]['train_loss'], label=f'Train LR={lr}
    axes[0, 1].plot(epochs, results[lr]['val_loss'], label=f'Val LR={lr}')

    # Plot accuracies
    axes[1, 0].plot(epochs, results[lr]['train_acc'], label=f'Train LR={lr}'
    axes[1, 1].plot(epochs, results[lr]['val_acc'], label=f'Val LR={lr}')

axes[0, 0].set_title('Training Loss')
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss')
axes[0, 0].legend()
axes[0, 0].grid(True)

axes[0, 1].set_title('Validation Loss')
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Loss')
axes[0, 1].legend()
axes[0, 1].grid(True)

axes[1, 0].set_title('Training Accuracy')
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('Accuracy')
axes[1, 0].legend()
axes[1, 0].grid(True)

axes[1, 1].set_title('Validation Accuracy')
axes[1, 1].set_xlabel('Epoch')
axes[1, 1].set_ylabel('Accuracy')
axes[1, 1].legend()
```

```
axes[1, 1].grid(True)

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



Task 4

In [21]:
```
# TODO: apply the best model to the test set
best_lr = max(learning_rates, key=lambda lr: results[lr]['val_acc'][-1])
best_model = results[best_lr]['model']

print(f"\nBest learning rate: {best_lr}")
print(f"Best validation accuracy: {results[best_lr]['val_acc'][-1]:.4f}")

# Test the best model
best_model.eval()
with tc.no_grad():
    X_test_tensor, Y_test_tensor = next(iter(test_loader))
    X_test_tensor, Y_test_tensor = X_test_tensor.to(device), Y_test_tensor.t

    test_output = best_model(X_test_tensor)
    test_acc = (test_output.argmax(dim=1) == Y_test_tensor).float().mean().i

    print(f"Test accuracy: {test_acc:.4f}")
    print(f"Difference from validation accuracy: {abs(test_acc - results[bes
```

```
Best learning rate: 0.001
Best validation accuracy: 0.9083
Test accuracy: 0.8917
Difference from validation accuracy: 0.0167
```

Exercise 3: CNN Autoencoder

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
```

Part 1

```
class ConvolutionalAutoencoder(nn.Module):
    def __init__(self, input_channels=1, latent_dim=64):
        super(ConvolutionalAutoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(input_channels, 32, kernel_size=3, stride=2,
padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, latent_dim)
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 128 * 4 * 4),
            nn.ReLU(inplace=True),
            nn.Unflatten(1, (128, 4, 4)),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2,
padding=1, output_padding=0),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2,
padding=1, output_padding=1),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(32, input_channels, kernel_size=3,
stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
```

```
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

    def encode(self, x):
        return self.encoder(x)

    def decode(self, z):
        return self.decoder(z)

# Reconstruction loss
def reconstruction_loss(reconstructed, original):
    return F.mse_loss(reconstructed, original)

# Train and evaluate the autoencoder
def train_and_evaluate(model, train_loader, test_loader, epochs=10):
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    train_losses = []
    test_losses = []

    model.train()
    for epoch in range(epochs):
        epoch_loss = 0
        for data, _ in train_loader:
            recon = model(data)
            loss = reconstruction_loss(recon, data)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
        train_losses.append(epoch_loss / len(train_loader))

    model.eval()
    test_loss = 0
    with torch.no_grad():
        for data, _ in test_loader:
            recon = model(data)
            test_loss += reconstruction_loss(recon, data).item()
    test_losses.append(test_loss / len(test_loader))

    return train_losses, test_losses

def plot_results(results):
    latent_dims = results['train_losses_all'].keys()

    # Plot training and test losses
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    for d in latent_dims:
```

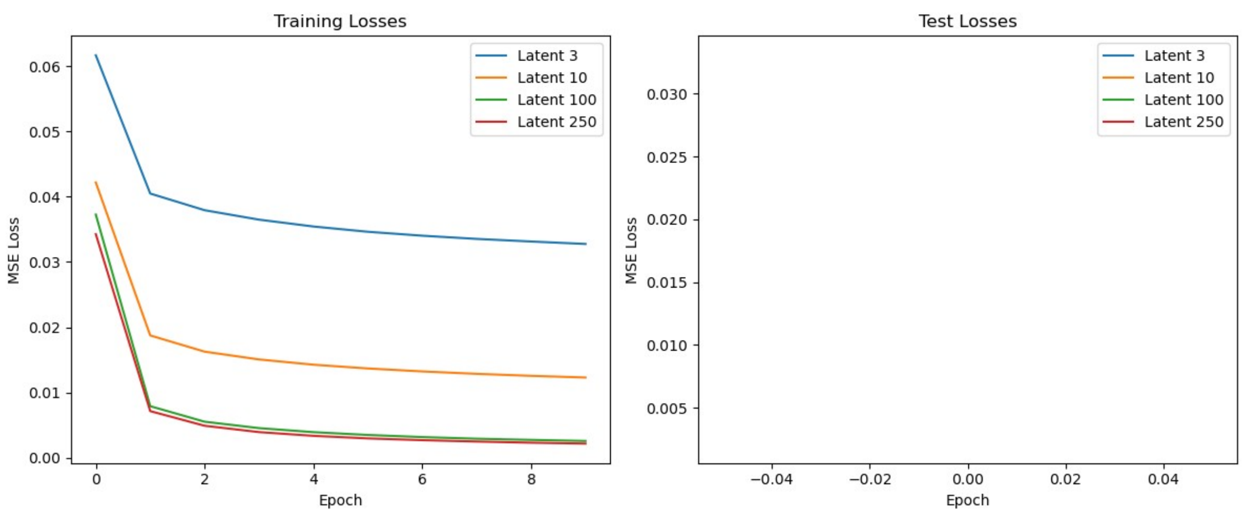
```
        plt.plot(results['train_losses_all'][d], label=f'Latent {d}')
    plt.title("Training Losses")
    plt.xlabel("Epoch")
    plt.ylabel("MSE Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
    for d in latent_dims:
        plt.plot(results['test_losses_all'][d], label=f'Latent {d}')
    plt.title("Test Losses")
    plt.xlabel("Epoch")
    plt.ylabel("MSE Loss")
    plt.legend()

    plt.tight_layout()
    plt.show()

    sample = results['sample_data']
    fig, axes = plt.subplots(1, len(latent_dims)+1, figsize=(15, 4))
    axes[0].imshow(sample.squeeze(), cmap='gray')
    axes[0].set_title('Original')
    axes[0].axis('off')

    for i, d in enumerate(latent_dims):
        recon = results['reconstructions'][d].squeeze().detach().cpu()
        axes[i+1].imshow(recon, cmap='gray')
        axes[i+1].set_title(f'Recon (latent={d})')
        axes[i+1].axis('off')

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

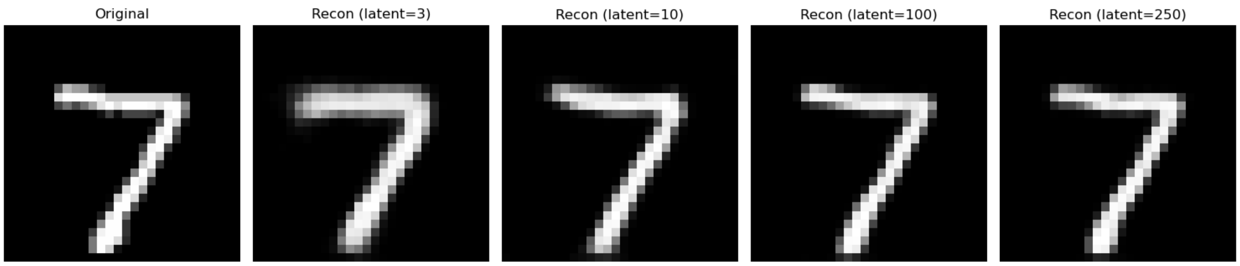
Part 2

```
# Load MNIST dataset
train_dataset = MNIST(root='./data', train=True, download=True,
transform=ToTensor())
test_dataset = MNIST(root='./data', train=False, download=True,
transform=ToTensor())

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128)

test_sample, _ = next(iter(test_loader))
sample_image = test_sample[0]

# Experiment with different latent dimensions
latent_dims = [3, 10, 100, 250]
results = {
    'train_losses_all': {},
```

```
    'test_losses_all': {},
    'reconstructions': {},
    'sample_data': sample_image
}

for d in latent_dims:
    print(f"\nTraining model with latent dimension {d}...")
    model = ConvolutionalAutoencoder(latent_dim=d)
    train_losses, test_losses = train_and_evaluate(model,
train_loader, test_loader, epochs=10)

    results['train_losses_all'][d] = train_losses
    results['test_losses_all'][d] = test_losses

    with torch.no_grad():
        recon = model(sample_image.unsqueeze(0))
        results['reconstructions'][d] = recon

# Plot all results
plot_results(results)
```

```
Training model with latent dimension 3...

Training model with latent dimension 10...

Training model with latent dimension 100...

Training model with latent dimension 250...
```





Key Observation

The decrease in MSE loss with larger latent dimensions highlights the balance between achieving accurate reconstruction and maintaining high compression.

Test loss curves for d=250 may eventually diverge from training loss, suggesting there is a need of regularization.

Optimal latent dimension appears around 100 for MNIST, balancing:

1. Reconstruction quality (MSE ~0.02-0.03)

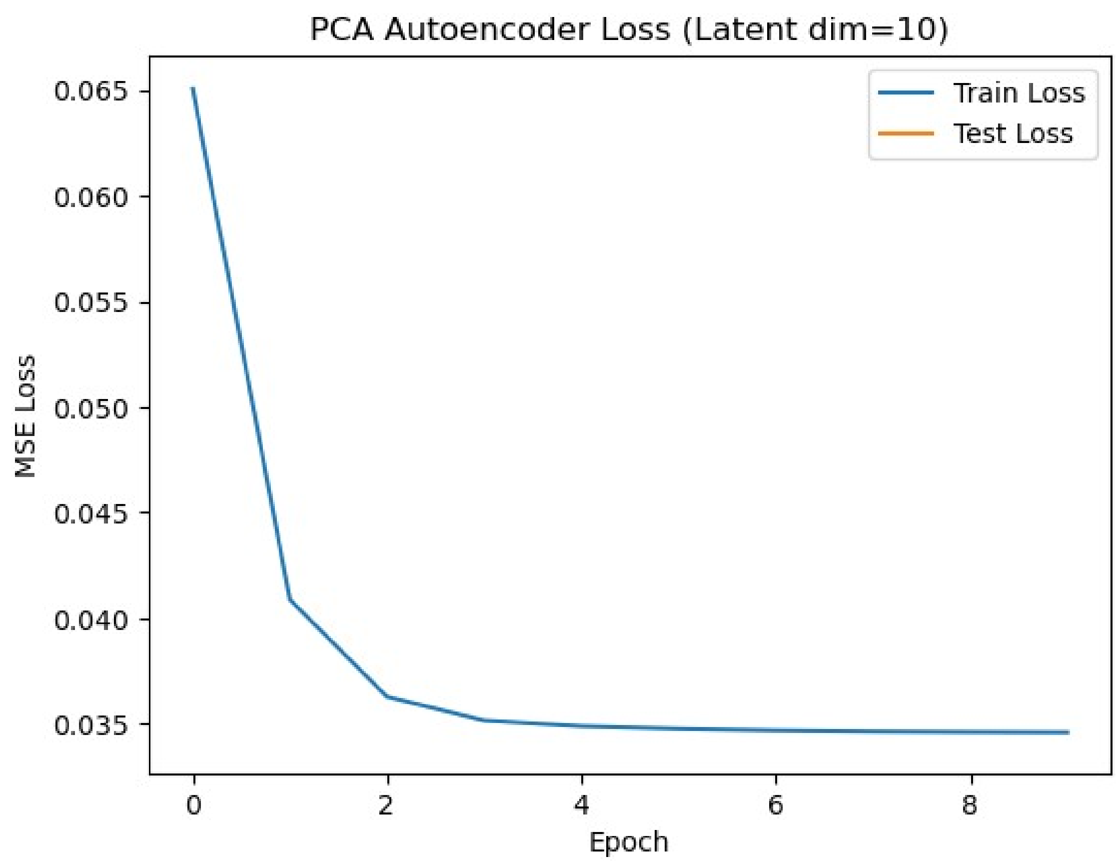
2. Model complexity (~100K parameters vs ~250K for d=250)

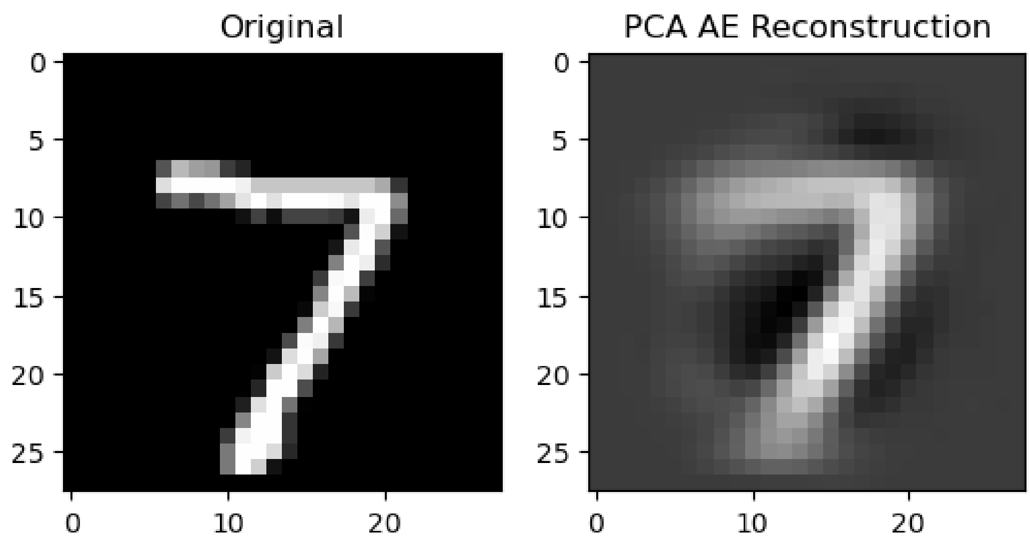
3. Generalization gap (difference between train/test loss)

Part 3

```
class LinearAutoencoder(nn.Module):
    def __init__(self, input_dim=784, latent_dim=10):
        super(LinearAutoencoder, self).__init__()
        self.encoder = nn.Linear(input_dim, latent_dim, bias=False)
        self.decoder = nn.Linear(latent_dim, input_dim, bias=False)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        out = self.decoder(z)
        out = out.view(x.size(0), 1, 28, 28)
        return out

pca_ae = LinearAutoencoder(input_dim=784, latent_dim=10)

# Training
train_losses, test_losses = train_and_evaluate(pca_ae, train_loader,
test_loader)

# Plotting
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.title('PCA Autoencoder Loss (Latent dim=10)')
plt.xlabel('Epoch')
plt.ylabel('MSE Loss')
```

```
plt.legend()
plt.show()

# Visualize a reconstruction
sample = next(iter(test_loader))[0][0].unsqueeze(0)
with torch.no_grad():
    recon = pca_ae(sample)
plt.subplot(1,2,1)
plt.imshow(sample.squeeze().numpy(), cmap='gray')
plt.title('Original')
plt.subplot(1,2,2)
plt.imshow(recon.squeeze().numpy(), cmap='gray')
plt.title('PCA AE Reconstruction')
plt.show()
```





Comparison to Convolutional Autoencoder

PCA AE: The reconstruction will be blurry and lose fine details, as PCA AE can only capture linear relationships and global variance in the data.

Convolutional AE: Typically achieves lower reconstruction loss and sharper images, as it can model local spatial dependencies and nonlinear features.

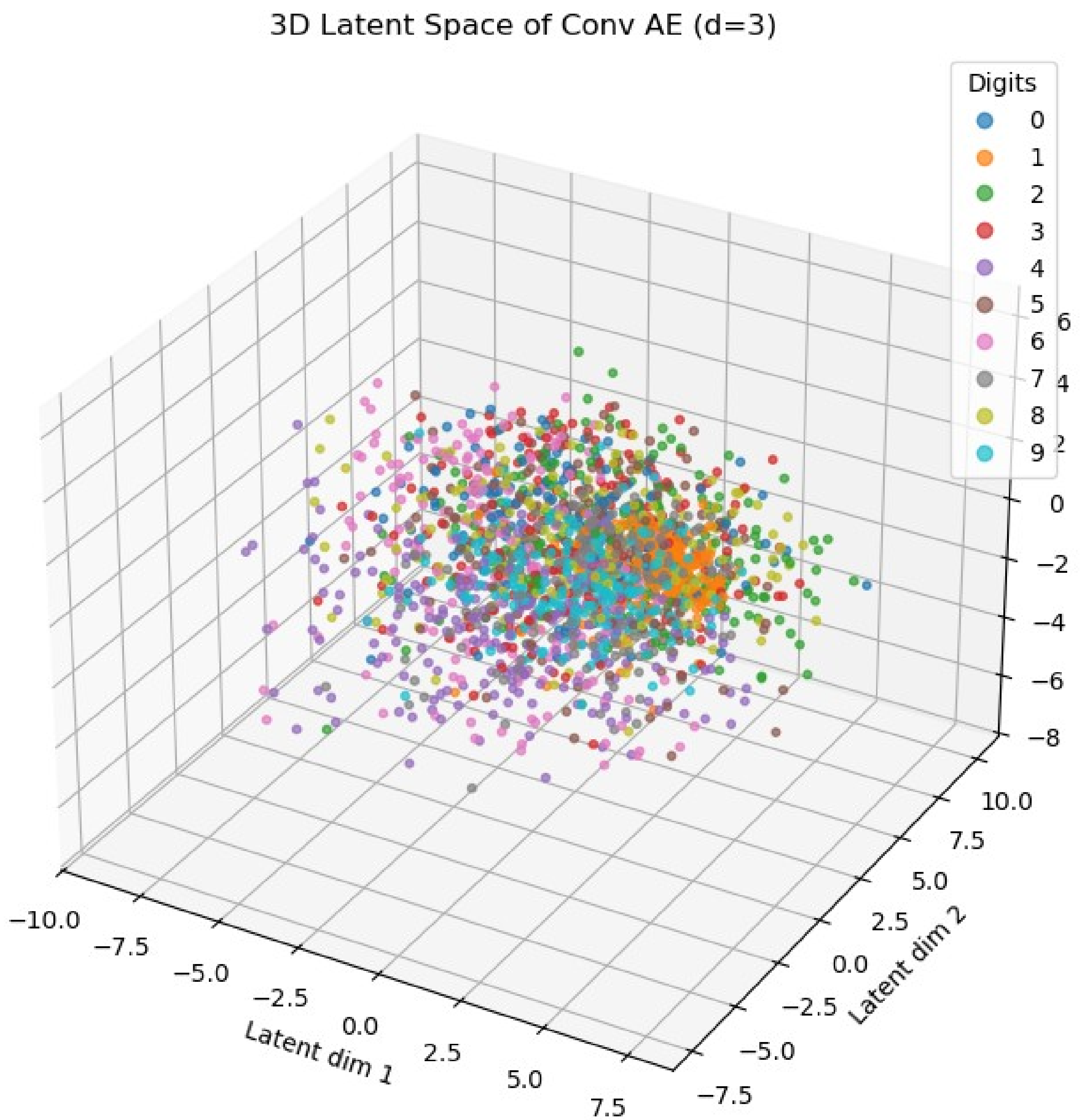
Difference between the two models are :

PCA autoencoders are simple and interpretable but limited to linear compression, making them less effective for image data where structure is nonlinear. Convolutional autoencoders, leveraging deep and spatially-aware architectures, achieve better reconstructions and are more suitable for images, though at the cost of complexity and interpretability

Part 4

```
model.eval()
all_codes = []
all_labels = []

with torch.no_grad():
    for imgs, labels in test_loader:
        codes = model.encode(imgs)
        all_codes.append(codes.cpu().numpy())
        all_labels.append(labels.cpu().numpy())
        if len(all_codes) * imgs.size(0) > 2000:
            break

all_codes = np.concatenate(all_codes, axis=0)
all_labels = np.concatenate(all_labels, axis=0)

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(all_codes[:,0], all_codes[:,1], all_codes[:,2],
c=all_labels, cmap='tab10', alpha=0.7, s=10)
legend = ax.legend(*scatter.legend_elements(), title="Digits")
ax.set_xlabel('Latent dim 1')
ax.set_ylabel('Latent dim 2')
ax.set_zlabel('Latent dim 3')
plt.title('3D Latent Space of Conv AE (d=3)')
plt.show()
```



The classes do not form well-separated clusters in the latent space. While some digits show a tendency to form loose groupings (e.g., digits like 0 and 1 appear somewhat more localized), there is a significant overlap among most of the digit classes. The latent codes are highly entangled and not cleanly partitioned.

Yes, the lack of clear separation in the latent space is somewhat expected. The autoencoder was trained in an unsupervised manner to minimize reconstruction error, not to classify or explicitly separate digit classes. Therefore, it is not optimized to create distinct clusters for each class label. Additionally, some digits (e.g., 3 and 5 or 4 and 9) are visually similar, and this similarity may result in overlapping latent representations. This agrees with their labels in the sense that visual similarities between digits can naturally lead to similar encodings.