MLE_sheet06-combined

June 14, 2025

1 Machine Learning Essentials SS25 - Exercise Sheet 6

1.1 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.

[30]:
```
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")
```

1.2 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.

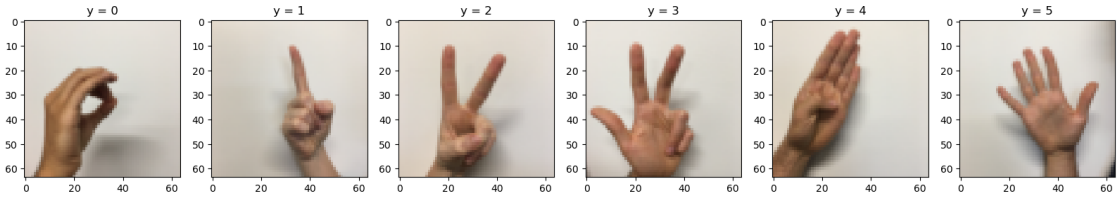
[31]:
```
# Load the dataset
X_train = np.load('sign_data/X_train.npy')
Y_train = np.load('sign_data/Y_train.npy')
X_val = np.load('sign_data/X_val.npy')
Y_val = np.load('sign_data/Y_val.npy')
X_test = np.load('sign_data/X_test.npy')
Y_test = np.load('sign_data/Y_test.npy')

# 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_counts=True)[1]))
print("Counts of classes in Y_val: " + str(np.unique(Y_val,␣
 ↪return_counts=True)[1]))
print("Counts of classes in Y_test: " + str(np.unique(Y_test,␣
 ↪return_counts=True)[1]))
```

```
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.

[32]:
```
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.

[33]:
```
X_train = X_train/255
X_val = X_val/255
X_test = X_test/255
```

1.2.1 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.

[34]:
```
class CNN_Classifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels= 3,out_channels=4,kernel_size=3,␣
↪stride=1,padding="same")
        self.bn1 = nn.BatchNorm2d(4)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride= 2, padding= 0)

        self.conv2 = nn.Conv2d(in_channels=4, out_channels= 8,␣
↪kernel_size=3,stride= 1,padding="same")
        self.bn2 = nn.BatchNorm2d(8)
        self.pool2 = nn.MaxPool2d(kernel_size= 2, stride= 2, padding= 0)


        # TODO: Initialize the layers of the CNN
        self.fc1 = nn.Linear(8 * 16 * 16,64)
```

```
        self.fc2 = nn.Linear(64, 6)

        self.log_softmax = nn.LogSoftmax(dim=1)

    def forward(self, X):
        # TODO: Implement the forward pass
        X = self.conv1(X)
        X = self.bn1(X)
        X = nn.functional.relu(X)
        X = self.pool1(X)

        X = self.conv2(X)
        X = self.bn2(X)
        X = nn.functional.relu(X)
        X = self.pool2(X)

        X = X.view(X.size(0), -1)

        X = self.fc1(X)
        X = nn.functional.relu(X)
        X = self.fc2(X)
        X = self.log_softmax(X)


        return X
```

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

[35]:
```
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:

[36]:
```
summary(cnn_model, input_size=(3, 64, 64), device="cpu")
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 4, 64, 64]             112
       BatchNorm2d-2            [-1, 4, 64, 64]               8
         MaxPool2d-3            [-1, 4, 32, 32]               0
```

```
            Conv2d-4            [-1, 8, 32, 32]             296
       BatchNorm2d-5            [-1, 8, 32, 32]              16
         MaxPool2d-6            [-1, 8, 16, 16]               0
            Linear-7                  [-1, 64]         131,136
            Linear-8                   [-1, 6]             390
        LogSoftmax-9                   [-1, 6]               0
================================================================
Total params: 131,958
Trainable params: 131,958
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 0.42
Params size (MB): 0.50
Estimated Total Size (MB): 0.97
----------------------------------------------------------------
```

1.2.2 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.

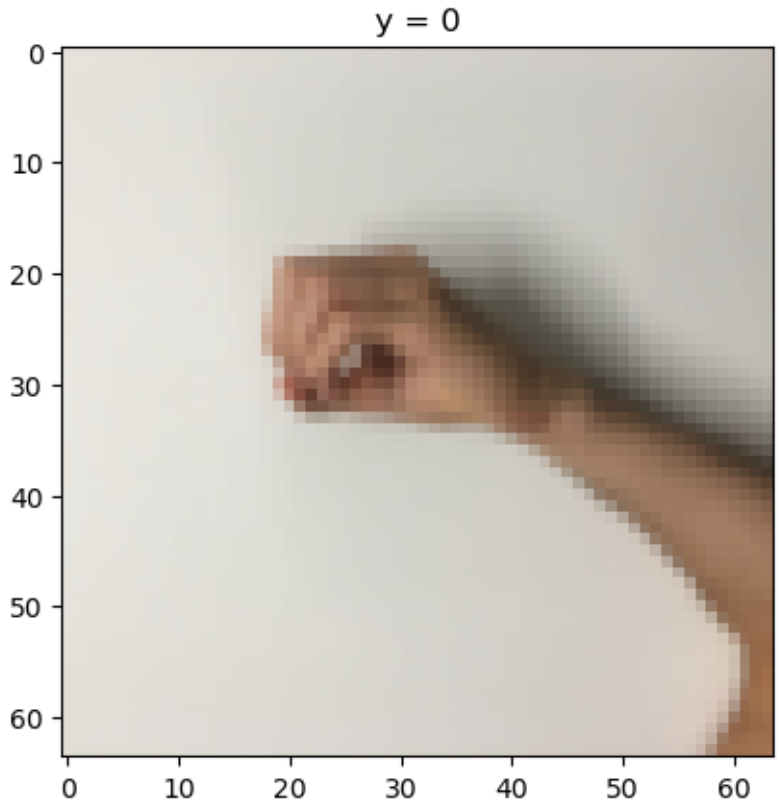
[37]:
```python
class Image_Dataset(Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    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␣
 ↪tensor
        image = tc.tensor(self.X[idx], dtype=tc.float32).permute(2, 0, 1)
        label = tc.tensor(self.Y[idx], dtype=tc.long)
        return image, label
```

[38]:
```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,␣
 ↪shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=val_batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=test_batch_size,␣
 ↪shuffle=False)
```

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

[39]:
```
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()
```



1.2.3 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.

[40]:
```python
def train_cnn(model, train_loader, val_loader, lr, n_epochs, device):
    model = model.to(device)
    model_state = None
    # 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().item()

        model.eval()
        val_epoch_loss = 0
        val_correct = 0
        val_total = 0

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

                val_epoch_loss += loss.item() / len(val_loader)
                val_correct += (output.argmax(dim=1) == Y).sum().item()
                val_total += Y.size(0)
```

```
        val_loss[epoch - 1] = val_epoch_loss
        val_acc[epoch - 1] = val_correct / val_total
        print(f"Epoch {epoch}/{n_epochs} - Train Loss: {epoch_loss:.4f}, Val␣
 ↪Loss: {val_epoch_loss:.4f}, "
              f"Train Acc: {train_acc[epoch - 1] * 100:.2f}%, Val Acc:␣
 ↪{val_acc[epoch - 1] * 100:.2f}%")

    return train_loss, val_loss, train_acc, val_acc
```

[41]:
```
n_epochs = 50
# TODO: Train the model with different learning rates
lr = 0.001
device = tc.device("cuda" if tc.cuda.is_available() else "cpu")
cnn_model = CNN_Classifier()
train_loss, val_loss, train_acc, val_acc = train_cnn(cnn_model, train_loader,␣
 ↪val_loader, lr, n_epochs, device)
```

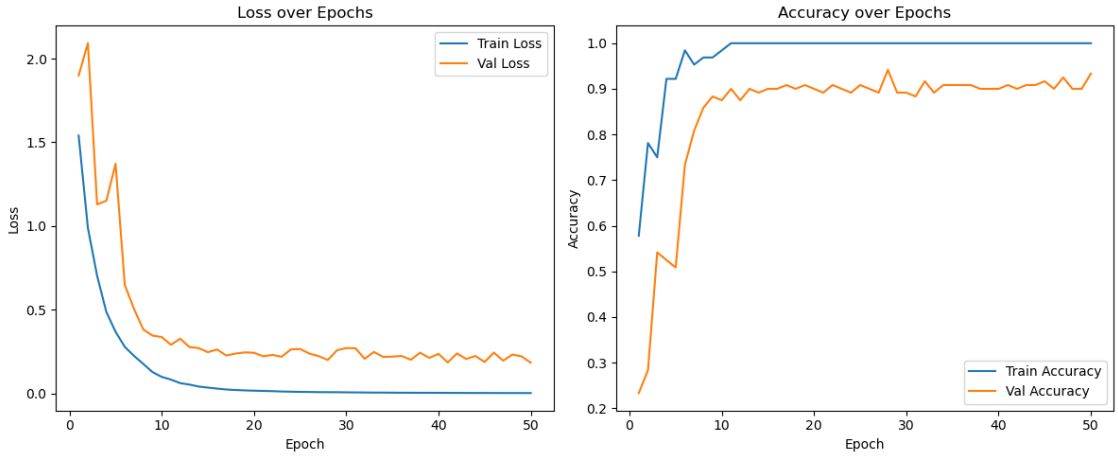
```
Epoch 1/50 - Train Loss: 1.5400, Val Loss: 1.8987, Train Acc: 57.81%, Val Acc:
23.33%
Epoch 2/50 - Train Loss: 0.9890, Val Loss: 2.0921, Train Acc: 78.12%, Val Acc:
28.33%
Epoch 3/50 - Train Loss: 0.7024, Val Loss: 1.1290, Train Acc: 75.00%, Val Acc:
54.17%
Epoch 4/50 - Train Loss: 0.4865, Val Loss: 1.1508, Train Acc: 92.19%, Val Acc:
52.50%
Epoch 5/50 - Train Loss: 0.3678, Val Loss: 1.3717, Train Acc: 92.19%, Val Acc:
50.83%
Epoch 6/50 - Train Loss: 0.2778, Val Loss: 0.6476, Train Acc: 98.44%, Val Acc:
73.33%
Epoch 7/50 - Train Loss: 0.2237, Val Loss: 0.5045, Train Acc: 95.31%, Val Acc:
80.83%
Epoch 8/50 - Train Loss: 0.1761, Val Loss: 0.3821, Train Acc: 96.88%, Val Acc:
85.83%
Epoch 9/50 - Train Loss: 0.1271, Val Loss: 0.3452, Train Acc: 96.88%, Val Acc:
88.33%
Epoch 10/50 - Train Loss: 0.0984, Val Loss: 0.3368, Train Acc: 98.44%, Val Acc:
87.50%
Epoch 11/50 - Train Loss: 0.0823, Val Loss: 0.2899, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 12/50 - Train Loss: 0.0611, Val Loss: 0.3267, Train Acc: 100.00%, Val Acc:
87.50%
Epoch 13/50 - Train Loss: 0.0525, Val Loss: 0.2770, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 14/50 - Train Loss: 0.0407, Val Loss: 0.2704, Train Acc: 100.00%, Val Acc:
89.17%
Epoch 15/50 - Train Loss: 0.0345, Val Loss: 0.2460, Train Acc: 100.00%, Val Acc:
90.00%
```

```
Epoch 16/50 - Train Loss: 0.0283, Val Loss: 0.2622, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 17/50 - Train Loss: 0.0231, Val Loss: 0.2261, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 18/50 - Train Loss: 0.0198, Val Loss: 0.2377, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 19/50 - Train Loss: 0.0176, Val Loss: 0.2448, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 20/50 - Train Loss: 0.0159, Val Loss: 0.2425, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 21/50 - Train Loss: 0.0144, Val Loss: 0.2214, Train Acc: 100.00%, Val Acc:
89.17%
Epoch 22/50 - Train Loss: 0.0129, Val Loss: 0.2297, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 23/50 - Train Loss: 0.0110, Val Loss: 0.2186, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 24/50 - Train Loss: 0.0097, Val Loss: 0.2626, Train Acc: 100.00%, Val Acc:
89.17%
Epoch 25/50 - Train Loss: 0.0088, Val Loss: 0.2651, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 26/50 - Train Loss: 0.0081, Val Loss: 0.2376, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 27/50 - Train Loss: 0.0072, Val Loss: 0.2227, Train Acc: 100.00%, Val Acc:
89.17%
Epoch 28/50 - Train Loss: 0.0069, Val Loss: 0.1994, Train Acc: 100.00%, Val Acc:
94.17%
Epoch 29/50 - Train Loss: 0.0067, Val Loss: 0.2574, Train Acc: 100.00%, Val Acc:
89.17%
Epoch 30/50 - Train Loss: 0.0060, Val Loss: 0.2707, Train Acc: 100.00%, Val Acc:
89.17%
Epoch 31/50 - Train Loss: 0.0056, Val Loss: 0.2692, Train Acc: 100.00%, Val Acc:
88.33%
Epoch 32/50 - Train Loss: 0.0050, Val Loss: 0.2065, Train Acc: 100.00%, Val Acc:
91.67%
Epoch 33/50 - Train Loss: 0.0046, Val Loss: 0.2473, Train Acc: 100.00%, Val Acc:
89.17%
Epoch 34/50 - Train Loss: 0.0044, Val Loss: 0.2176, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 35/50 - Train Loss: 0.0040, Val Loss: 0.2195, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 36/50 - Train Loss: 0.0036, Val Loss: 0.2235, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 37/50 - Train Loss: 0.0035, Val Loss: 0.2005, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 38/50 - Train Loss: 0.0032, Val Loss: 0.2429, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 39/50 - Train Loss: 0.0031, Val Loss: 0.2118, Train Acc: 100.00%, Val Acc:
90.00%
```

```
Epoch 40/50 - Train Loss: 0.0030, Val Loss: 0.2360, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 41/50 - Train Loss: 0.0027, Val Loss: 0.1844, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 42/50 - Train Loss: 0.0027, Val Loss: 0.2384, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 43/50 - Train Loss: 0.0024, Val Loss: 0.2054, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 44/50 - Train Loss: 0.0023, Val Loss: 0.2233, Train Acc: 100.00%, Val Acc:
90.83%
Epoch 45/50 - Train Loss: 0.0022, Val Loss: 0.1870, Train Acc: 100.00%, Val Acc:
91.67%
Epoch 46/50 - Train Loss: 0.0020, Val Loss: 0.2436, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 47/50 - Train Loss: 0.0020, Val Loss: 0.1952, Train Acc: 100.00%, Val Acc:
92.50%
Epoch 48/50 - Train Loss: 0.0019, Val Loss: 0.2315, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 49/50 - Train Loss: 0.0018, Val Loss: 0.2212, Train Acc: 100.00%, Val Acc:
90.00%
Epoch 50/50 - Train Loss: 0.0017, Val Loss: 0.1830, Train Acc: 100.00%, Val Acc:
93.33%
```

[42]:
```python
# TODO: Visualize the results

epochs = np.arange(1, n_epochs + 1)

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

# Loss plot
plt.subplot(1, 2, 1)
plt.plot(epochs, train_loss, label="Train Loss")
plt.plot(epochs, val_loss, label="Val Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss over Epochs")
plt.legend()

# Accuracy plot
plt.subplot(1, 2, 2)
plt.plot(epochs, train_acc, label="Train Accuracy")
plt.plot(epochs, val_acc, label="Val Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Accuracy over Epochs")
plt.legend()
```

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



1.2.4 Task 4

[43]:
```
# TODO: apply the best model to the test set
def evaluate_model(model, test_loader, device):
    model.eval()
    model = model.to(device)

    correct = 0
    total = 0

    with tc.no_grad():
        for X, Y in test_loader:
            X, Y = X.to(device), Y.to(device)
            output = model(X)
            predictions = output.argmax(dim=1)
            correct += (predictions == Y).sum().item()
            total += Y.size(0)

    test_accuracy = correct / total
    print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
    return test_accuracy

test_accuracy = evaluate_model(cnn_model, test_loader, device)
```

```
Test Accuracy: 95.83%
```

1.3 Exercise 3 - CNN Autoencoder

In the next task we want to build an autoencoder. It consists of an encoder, which transforms the data into a low-dimensional code, and a decoder, which reconstructs the original data.

We use the Fashion MNIST dataset, which consists of 28x28 grayscale images. There are 10 classes, each representing different items of clothing. The data can be conveniently downloaded, separated and transformed with torchvision. As we will not be tuning any hyperparameters, we do not need a validation set.

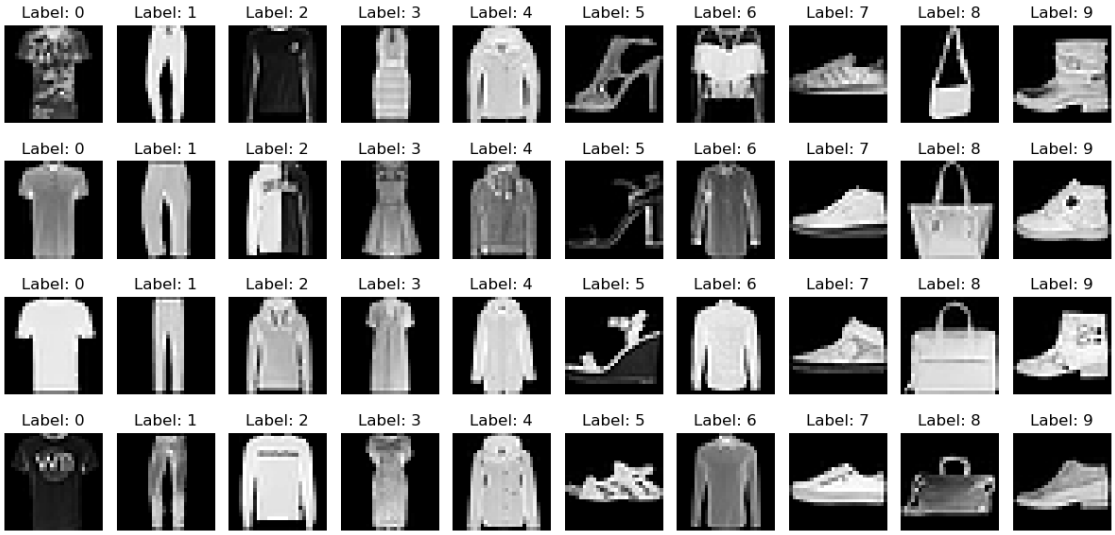
[44]:
```
from torchvision import datasets, transforms
import numpy as np
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from torch import nn

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert PIL image to tensor (range [0, 1])
    transforms.Normalize((0.5,), (0.5,))  # Normalize to range [-1, 1]
])

# Load FashionMNIST train and test sets
train_data = datasets.FashionMNIST(
    root='./fashion_mnist',          # Download path
    train=True,            # Load training set
    download=True,         # Download if not already present
    transform=transform    # Apply transformations
)

test_data = datasets.FashionMNIST(
    root='./fashion_mnist',
    train=False,           # Load test set
    download=True,
    transform=transform
)

classes = train_data.targets.unique()

# we only need a subset that consists of 1000 samples of each class for the␣
 ↪train set
# and 10 samples of each class for the test set
indices_train = []
indices_test = []
for i in range(len(classes)):
    indices_train += list(np.where(train_data.targets == classes[i])[0][:1000])
    indices_test += list(np.where(test_data.targets == classes[i])[0][:10])
```

```
train_data.data = train_data.data[indices_train]
train_data.targets = train_data.targets[indices_train]
test_data.data = test_data.data[indices_test]
test_data.targets = test_data.targets[indices_test]

# Create DataLoaders
train_batch_size = 256
test_batch_size = len(test_data)
train_loader = DataLoader(train_data, batch_size=train_batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=test_batch_size, shuffle=False)

# Print the shape of the dataset
print("train images shape: " + str(train_data.data.shape))
print("train labels shape: " + str(train_data.targets.shape))
print("test images shape: " + str(test_data.data.shape))
print("test labels shape: " + str(test_data.targets.shape))
print("classes: " + str(classes))

# check if classes are balanced
print("Counts of classes in train set: " + str(train_data.targets.
 ↪unique(return_counts=True)[1]))
print("Counts of classes in test set: " + str(test_data.targets.
 ↪unique(return_counts=True)[1]))
```

```
train images shape: torch.Size([10000, 28, 28])
train labels shape: torch.Size([10000])
test images shape: torch.Size([100, 28, 28])
test labels shape: torch.Size([100])
classes: tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Counts of classes in train set: tensor([1000, 1000, 1000, 1000, 1000, 1000,
1000, 1000, 1000, 1000])
Counts of classes in test set: tensor([10, 10, 10, 10, 10, 10, 10, 10, 10, 10])
```

[45]:
```
# Get a batch of data
data_iter = iter(train_loader)
images, labels = next(data_iter)

# Visualize a batch of images
fig, axes = plt.subplots(4, 10, figsize=(12, 6))
for i in range(4):
    for j in range(10):
        # find the ith image of class j
        idx = np.where(labels == j)[0][i]
        axes[i, j].imshow(images[idx].squeeze(), cmap='gray')
        axes[i, j].set_title(f"Label: {j}")
        axes[i, j].axis('off')
plt.tight_layout()
```

```
plt.show()
```



1.3.1 Task 1

We compare two architectures: a linear autoencoder and a CNN autoencoder. The latter typically consists of convolutional layers for the encoder and transposed convolutional layers for the decoder. In addition, fully connected layers can bring the feature to the desired code dimension (also called latent dimension). Implement the following architecture:

Encoder: - Conv2d: 16 output channels, 3 by 3 filter size, stride 1, padding "same" - ReLU activation - MaxPool2d: 2 by 2 filter, stride 1, padding 0 - Conv2d: 32 output channels, 3 by 3, stride 1, padding "same" - ReLU activation - MaxPool2d: 2 by 2 filter, stride 1, padding 0 - Conv2d: 64 output channel, 3 by 3 filter size, stride 1, padding "same" - ReLU activation - MaxPool2d: 2 by 2 filter, stride 1, padding 0 - Flatten the previous output - Linear: *<latent dimension>* output neurons

Decoder: - Linear: 64x3x3 = 576 output neurons - Unflatten the previous output to shape (64, 3, 3) - ConvTranspose2d: 32 output channels, 3 by 3 filter size, stride 2, padding 0, output padding 0 - ReLU activation - ConvTranspose2d: 16 output channels, 3 by 3 filter size, stride 2, padding 1, output padding 1 - ReLU activation - ConvTranspose2d: 1 output channel, 3 by 3 filter size, stride 2, padding 1, output padding 1

You might need to infer the input dimension of the linear layer in the encoder.

[46]:
```
class Conv_AE(nn.Module):
    def __init__(self, latent_dim):
        super(Conv_AE, self).__init__()

        # TODO: Initialize the layers of the encoder
        self.encoder = nn.Sequential(
```

```
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1,␣
↪padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=1, padding=0),
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1,␣
↪padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=1, padding=0),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1,␣
↪padding="same"),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=1, padding=0),
            nn.Flatten(),
            nn.Linear(in_features=64 * 25 * 25, out_features=latent_dim) # we␣
↪have 3 maxpool2d layers, each reducing the spatial dimensions by 2, so the␣
↪final size is 64 * 25 * 25
        )

        # TODO: Initialize the layers of the decoder
        self.decoder = nn.Sequential(
            nn.Linear(in_features=latent_dim, out_features=64 * 3 * 3),
            nn.Unflatten(1, (64, 3, 3)),
            nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=3,␣
↪stride=2, padding=0, output_padding=0),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=32, out_channels=16, kernel_size=3,␣
↪stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=16, out_channels=1, kernel_size=3,␣
↪stride=2, padding=1, output_padding=1)
        )

    def forward(self, X):
        # TODO: Implement the forward pass
        encoded = self.encoder(X)
        decoded = self.decoder(encoded)
        return decoded
```

We check again whether the model is working properly.

[47]:
```
import torch as tc
from torchsummary import summary

conv_ae_model = Conv_AE(2)
X_random = tc.randn(2, 1, 28, 28)
reconstructed, latent = conv_ae_model(X_random)
```

```
print("Reconstructed shape: " + str(reconstructed.shape), "\n", "Latent shape:␣
 ↪" + str(latent.shape))

summary(conv_ae_model, input_size=(1, 28, 28), device="cpu")
```

```
Reconstructed shape: torch.Size([1, 28, 28])
 Latent shape: torch.Size([1, 28, 28])
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 28, 28]             160
              ReLU-2           [-1, 16, 28, 28]               0
         MaxPool2d-3           [-1, 16, 27, 27]               0
            Conv2d-4           [-1, 32, 27, 27]           4,640
              ReLU-5           [-1, 32, 27, 27]               0
         MaxPool2d-6           [-1, 32, 26, 26]               0
            Conv2d-7           [-1, 64, 26, 26]          18,496
              ReLU-8           [-1, 64, 26, 26]               0
         MaxPool2d-9           [-1, 64, 25, 25]               0
           Flatten-10                [-1, 40000]               0
           Linear-11                    [-1, 2]          80,002
           Linear-12                  [-1, 576]           1,728
        Unflatten-13             [-1, 64, 3, 3]               0
  ConvTranspose2d-14             [-1, 32, 7, 7]          18,464
             ReLU-15             [-1, 32, 7, 7]               0
  ConvTranspose2d-16           [-1, 16, 14, 14]           4,624
             ReLU-17           [-1, 16, 14, 14]               0
  ConvTranspose2d-18            [-1, 1, 28, 28]             145
================================================================
Total params: 128,259
Trainable params: 128,259
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 2.16
Params size (MB): 0.49
Estimated Total Size (MB): 2.65
----------------------------------------------------------------
```

1.3.2 Task 2

The training of an autoencoder compares the original input to the reconstruction, usually by means of the mean squared error.

[48]:
```
def train_ae(model, train_loader, test_loader, lr, n_epochs, device):
    model = model.to(device)

    # TODO: Initialize the loss function
```

```
    loss_function = nn.MSELoss()
    optimizer = tc.optim.Adam(model.parameters(), lr=lr)

    train_loss = np.zeros(n_epochs)
    test_loss = np.zeros(n_epochs)

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

        epoch_loss = 0

        for X, _ in train_loader:
            X = X.to(device)
            # TODO: Implement the training loop
            optimizer.zero_grad()
            reconstructed = model(X)
            loss = loss_function(reconstructed, X)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()/len(train_loader)

        train_loss[epoch - 1] = epoch_loss

        model.eval()

        with tc.no_grad():
            X, _ = next(iter(test_loader))
            X = X.to(device)
            # TODO: Implement the evaluation step
            reconstructed = model(X)
            loss = loss_function(reconstructed, X)

            test_loss[epoch - 1] = loss.item()
        print(f"Epoch {epoch}/{n_epochs} - Train Loss: {epoch_loss:.4f}, Test␣
 ↪Loss: {loss.item():.4f}")

    return train_loss, test_loss
```

[49]:
```
n_epochs = 60
lr = 1e-3

# TODO: Train the convolutional autoencoder model with different latent␣
 ↪dimensions
latent_dims = [3, 100, 100, 150]
train_losses = []
test_losses = []
device = tc.device("cuda" if tc.cuda.is_available() else "cpu")
```

```
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=len(test_data), shuffle=False)

for latent_dim in latent_dims:
    print(f"Training Conv_AE with latent dimension: {latent_dim}")
    conv_ae_model = Conv_AE(latent_dim)
    train_loss, test_loss = train_ae(conv_ae_model, train_loader, test_loader,␣
 ↪lr, n_epochs, device)
    train_losses.append(train_loss)
    test_losses.append(test_loss)
```

```
Training Conv_AE with latent dimension: 3
Epoch 1/60 - Train Loss: 0.3054, Test Loss: 0.1680
Epoch 2/60 - Train Loss: 0.1612, Test Loss: 0.1479
Epoch 3/60 - Train Loss: 0.1480, Test Loss: 0.1382
Epoch 4/60 - Train Loss: 0.1402, Test Loss: 0.1320
Epoch 5/60 - Train Loss: 0.1348, Test Loss: 0.1269
Epoch 6/60 - Train Loss: 0.1307, Test Loss: 0.1239
Epoch 7/60 - Train Loss: 0.1270, Test Loss: 0.1203
Epoch 8/60 - Train Loss: 0.1233, Test Loss: 0.1176
Epoch 9/60 - Train Loss: 0.1197, Test Loss: 0.1139
Epoch 10/60 - Train Loss: 0.1164, Test Loss: 0.1114
Epoch 11/60 - Train Loss: 0.1137, Test Loss: 0.1085
Epoch 12/60 - Train Loss: 0.1114, Test Loss: 0.1059
Epoch 13/60 - Train Loss: 0.1098, Test Loss: 0.1062
Epoch 14/60 - Train Loss: 0.1085, Test Loss: 0.1055
Epoch 15/60 - Train Loss: 0.1077, Test Loss: 0.1045
Epoch 16/60 - Train Loss: 0.1064, Test Loss: 0.1032
Epoch 17/60 - Train Loss: 0.1053, Test Loss: 0.1019
Epoch 18/60 - Train Loss: 0.1046, Test Loss: 0.1031
Epoch 19/60 - Train Loss: 0.1040, Test Loss: 0.1022
Epoch 20/60 - Train Loss: 0.1036, Test Loss: 0.1018
Epoch 21/60 - Train Loss: 0.1026, Test Loss: 0.1000
Epoch 22/60 - Train Loss: 0.1023, Test Loss: 0.0998
Epoch 23/60 - Train Loss: 0.1018, Test Loss: 0.1014
Epoch 24/60 - Train Loss: 0.1017, Test Loss: 0.1005
Epoch 25/60 - Train Loss: 0.1012, Test Loss: 0.0992
Epoch 26/60 - Train Loss: 0.1006, Test Loss: 0.0994
Epoch 27/60 - Train Loss: 0.0999, Test Loss: 0.0989
Epoch 28/60 - Train Loss: 0.0997, Test Loss: 0.0985
Epoch 29/60 - Train Loss: 0.0994, Test Loss: 0.0992
Epoch 30/60 - Train Loss: 0.0994, Test Loss: 0.0992
Epoch 31/60 - Train Loss: 0.0989, Test Loss: 0.0985
Epoch 32/60 - Train Loss: 0.0987, Test Loss: 0.0978
Epoch 33/60 - Train Loss: 0.0985, Test Loss: 0.0987
Epoch 34/60 - Train Loss: 0.0983, Test Loss: 0.0987
Epoch 35/60 - Train Loss: 0.0980, Test Loss: 0.0980
Epoch 36/60 - Train Loss: 0.0978, Test Loss: 0.0978
```

```
Epoch 37/60 - Train Loss: 0.0978, Test Loss: 0.0979
Epoch 38/60 - Train Loss: 0.0975, Test Loss: 0.0973
Epoch 39/60 - Train Loss: 0.0972, Test Loss: 0.0974
Epoch 40/60 - Train Loss: 0.0969, Test Loss: 0.0979
Epoch 41/60 - Train Loss: 0.0971, Test Loss: 0.0977
Epoch 42/60 - Train Loss: 0.0968, Test Loss: 0.0972
Epoch 43/60 - Train Loss: 0.0966, Test Loss: 0.0979
Epoch 44/60 - Train Loss: 0.0962, Test Loss: 0.0968
Epoch 45/60 - Train Loss: 0.0961, Test Loss: 0.0962
Epoch 46/60 - Train Loss: 0.0960, Test Loss: 0.0975
Epoch 47/60 - Train Loss: 0.0961, Test Loss: 0.0965
Epoch 48/60 - Train Loss: 0.0961, Test Loss: 0.0962
Epoch 49/60 - Train Loss: 0.0959, Test Loss: 0.0956
Epoch 50/60 - Train Loss: 0.0955, Test Loss: 0.0970
Epoch 51/60 - Train Loss: 0.0953, Test Loss: 0.0958
Epoch 52/60 - Train Loss: 0.0952, Test Loss: 0.0958
Epoch 53/60 - Train Loss: 0.0956, Test Loss: 0.0962
Epoch 54/60 - Train Loss: 0.0952, Test Loss: 0.0965
Epoch 55/60 - Train Loss: 0.0949, Test Loss: 0.0972
Epoch 56/60 - Train Loss: 0.0949, Test Loss: 0.0961
Epoch 57/60 - Train Loss: 0.0948, Test Loss: 0.0949
Epoch 58/60 - Train Loss: 0.0947, Test Loss: 0.0965
Epoch 59/60 - Train Loss: 0.0948, Test Loss: 0.0956
Epoch 60/60 - Train Loss: 0.0947, Test Loss: 0.0958
Training Conv_AE with latent dimension: 100
Epoch 1/60 - Train Loss: 0.2071, Test Loss: 0.0907
Epoch 2/60 - Train Loss: 0.0762, Test Loss: 0.0650
Epoch 3/60 - Train Loss: 0.0588, Test Loss: 0.0573
Epoch 4/60 - Train Loss: 0.0512, Test Loss: 0.0466
Epoch 5/60 - Train Loss: 0.0452, Test Loss: 0.0428
Epoch 6/60 - Train Loss: 0.0415, Test Loss: 0.0388
Epoch 7/60 - Train Loss: 0.0386, Test Loss: 0.0365
Epoch 8/60 - Train Loss: 0.0362, Test Loss: 0.0344
Epoch 9/60 - Train Loss: 0.0343, Test Loss: 0.0329
Epoch 10/60 - Train Loss: 0.0327, Test Loss: 0.0315
Epoch 11/60 - Train Loss: 0.0313, Test Loss: 0.0308
Epoch 12/60 - Train Loss: 0.0302, Test Loss: 0.0297
Epoch 13/60 - Train Loss: 0.0293, Test Loss: 0.0286
Epoch 14/60 - Train Loss: 0.0282, Test Loss: 0.0277
Epoch 15/60 - Train Loss: 0.0271, Test Loss: 0.0270
Epoch 16/60 - Train Loss: 0.0267, Test Loss: 0.0270
Epoch 17/60 - Train Loss: 0.0261, Test Loss: 0.0264
Epoch 18/60 - Train Loss: 0.0252, Test Loss: 0.0250
Epoch 19/60 - Train Loss: 0.0247, Test Loss: 0.0251
Epoch 20/60 - Train Loss: 0.0244, Test Loss: 0.0250
Epoch 21/60 - Train Loss: 0.0239, Test Loss: 0.0241
Epoch 22/60 - Train Loss: 0.0234, Test Loss: 0.0236
Epoch 23/60 - Train Loss: 0.0229, Test Loss: 0.0232
```

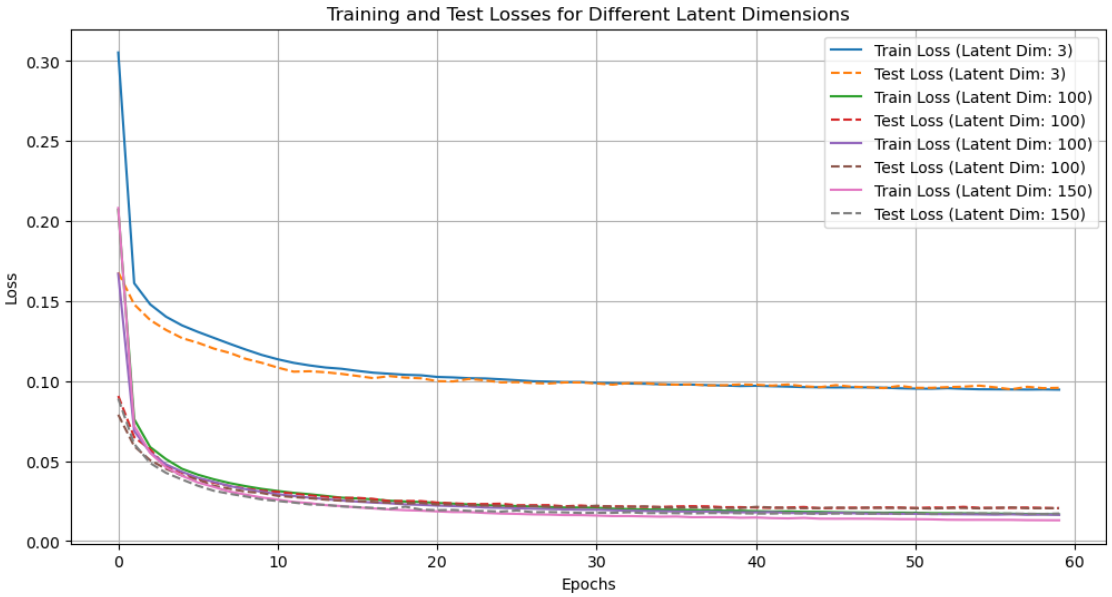
```
Epoch 24/60 - Train Loss: 0.0225, Test Loss: 0.0232
Epoch 25/60 - Train Loss: 0.0221, Test Loss: 0.0234
Epoch 26/60 - Train Loss: 0.0219, Test Loss: 0.0224
Epoch 27/60 - Train Loss: 0.0216, Test Loss: 0.0224
Epoch 28/60 - Train Loss: 0.0213, Test Loss: 0.0225
Epoch 29/60 - Train Loss: 0.0210, Test Loss: 0.0217
Epoch 30/60 - Train Loss: 0.0210, Test Loss: 0.0222
Epoch 31/60 - Train Loss: 0.0208, Test Loss: 0.0217
Epoch 32/60 - Train Loss: 0.0203, Test Loss: 0.0217
Epoch 33/60 - Train Loss: 0.0200, Test Loss: 0.0217
Epoch 34/60 - Train Loss: 0.0198, Test Loss: 0.0214
Epoch 35/60 - Train Loss: 0.0197, Test Loss: 0.0214
Epoch 36/60 - Train Loss: 0.0196, Test Loss: 0.0217
Epoch 37/60 - Train Loss: 0.0195, Test Loss: 0.0218
Epoch 38/60 - Train Loss: 0.0193, Test Loss: 0.0217
Epoch 39/60 - Train Loss: 0.0190, Test Loss: 0.0212
Epoch 40/60 - Train Loss: 0.0190, Test Loss: 0.0212
Epoch 41/60 - Train Loss: 0.0187, Test Loss: 0.0210
Epoch 42/60 - Train Loss: 0.0185, Test Loss: 0.0208
Epoch 43/60 - Train Loss: 0.0186, Test Loss: 0.0210
Epoch 44/60 - Train Loss: 0.0183, Test Loss: 0.0213
Epoch 45/60 - Train Loss: 0.0183, Test Loss: 0.0206
Epoch 46/60 - Train Loss: 0.0181, Test Loss: 0.0207
Epoch 47/60 - Train Loss: 0.0179, Test Loss: 0.0209
Epoch 48/60 - Train Loss: 0.0178, Test Loss: 0.0208
Epoch 49/60 - Train Loss: 0.0178, Test Loss: 0.0211
Epoch 50/60 - Train Loss: 0.0178, Test Loss: 0.0210
Epoch 51/60 - Train Loss: 0.0177, Test Loss: 0.0207
Epoch 52/60 - Train Loss: 0.0175, Test Loss: 0.0210
Epoch 53/60 - Train Loss: 0.0174, Test Loss: 0.0209
Epoch 54/60 - Train Loss: 0.0174, Test Loss: 0.0214
Epoch 55/60 - Train Loss: 0.0172, Test Loss: 0.0208
Epoch 56/60 - Train Loss: 0.0171, Test Loss: 0.0208
Epoch 57/60 - Train Loss: 0.0171, Test Loss: 0.0208
Epoch 58/60 - Train Loss: 0.0169, Test Loss: 0.0210
Epoch 59/60 - Train Loss: 0.0169, Test Loss: 0.0208
Epoch 60/60 - Train Loss: 0.0167, Test Loss: 0.0206
Training Conv_AE with latent dimension: 100
Epoch 1/60 - Train Loss: 0.1672, Test Loss: 0.0790
Epoch 2/60 - Train Loss: 0.0688, Test Loss: 0.0593
Epoch 3/60 - Train Loss: 0.0554, Test Loss: 0.0506
Epoch 4/60 - Train Loss: 0.0479, Test Loss: 0.0451
Epoch 5/60 - Train Loss: 0.0433, Test Loss: 0.0410
Epoch 6/60 - Train Loss: 0.0396, Test Loss: 0.0383
Epoch 7/60 - Train Loss: 0.0370, Test Loss: 0.0346
Epoch 8/60 - Train Loss: 0.0344, Test Loss: 0.0327
Epoch 9/60 - Train Loss: 0.0324, Test Loss: 0.0309
Epoch 10/60 - Train Loss: 0.0308, Test Loss: 0.0301
```

```
Epoch 11/60 - Train Loss: 0.0293, Test Loss: 0.0283
Epoch 12/60 - Train Loss: 0.0282, Test Loss: 0.0275
Epoch 13/60 - Train Loss: 0.0272, Test Loss: 0.0268
Epoch 14/60 - Train Loss: 0.0263, Test Loss: 0.0259
Epoch 15/60 - Train Loss: 0.0254, Test Loss: 0.0252
Epoch 16/60 - Train Loss: 0.0247, Test Loss: 0.0250
Epoch 17/60 - Train Loss: 0.0241, Test Loss: 0.0245
Epoch 18/60 - Train Loss: 0.0236, Test Loss: 0.0240
Epoch 19/60 - Train Loss: 0.0229, Test Loss: 0.0235
Epoch 20/60 - Train Loss: 0.0226, Test Loss: 0.0238
Epoch 21/60 - Train Loss: 0.0222, Test Loss: 0.0228
Epoch 22/60 - Train Loss: 0.0219, Test Loss: 0.0229
Epoch 23/60 - Train Loss: 0.0216, Test Loss: 0.0224
Epoch 24/60 - Train Loss: 0.0210, Test Loss: 0.0222
Epoch 25/60 - Train Loss: 0.0208, Test Loss: 0.0221
Epoch 26/60 - Train Loss: 0.0205, Test Loss: 0.0219
Epoch 27/60 - Train Loss: 0.0203, Test Loss: 0.0220
Epoch 28/60 - Train Loss: 0.0201, Test Loss: 0.0216
Epoch 29/60 - Train Loss: 0.0198, Test Loss: 0.0215
Epoch 30/60 - Train Loss: 0.0196, Test Loss: 0.0214
Epoch 31/60 - Train Loss: 0.0196, Test Loss: 0.0221
Epoch 32/60 - Train Loss: 0.0193, Test Loss: 0.0214
Epoch 33/60 - Train Loss: 0.0190, Test Loss: 0.0214
Epoch 34/60 - Train Loss: 0.0189, Test Loss: 0.0217
Epoch 35/60 - Train Loss: 0.0187, Test Loss: 0.0212
Epoch 36/60 - Train Loss: 0.0185, Test Loss: 0.0209
Epoch 37/60 - Train Loss: 0.0184, Test Loss: 0.0210
Epoch 38/60 - Train Loss: 0.0185, Test Loss: 0.0208
Epoch 39/60 - Train Loss: 0.0181, Test Loss: 0.0207
Epoch 40/60 - Train Loss: 0.0180, Test Loss: 0.0206
Epoch 41/60 - Train Loss: 0.0180, Test Loss: 0.0214
Epoch 42/60 - Train Loss: 0.0181, Test Loss: 0.0210
Epoch 43/60 - Train Loss: 0.0176, Test Loss: 0.0207
Epoch 44/60 - Train Loss: 0.0175, Test Loss: 0.0206
Epoch 45/60 - Train Loss: 0.0174, Test Loss: 0.0207
Epoch 46/60 - Train Loss: 0.0177, Test Loss: 0.0210
Epoch 47/60 - Train Loss: 0.0175, Test Loss: 0.0206
Epoch 48/60 - Train Loss: 0.0172, Test Loss: 0.0209
Epoch 49/60 - Train Loss: 0.0172, Test Loss: 0.0207
Epoch 50/60 - Train Loss: 0.0171, Test Loss: 0.0208
Epoch 51/60 - Train Loss: 0.0171, Test Loss: 0.0206
Epoch 52/60 - Train Loss: 0.0169, Test Loss: 0.0203
Epoch 53/60 - Train Loss: 0.0169, Test Loss: 0.0207
Epoch 54/60 - Train Loss: 0.0168, Test Loss: 0.0207
Epoch 55/60 - Train Loss: 0.0167, Test Loss: 0.0205
Epoch 56/60 - Train Loss: 0.0165, Test Loss: 0.0207
Epoch 57/60 - Train Loss: 0.0168, Test Loss: 0.0210
Epoch 58/60 - Train Loss: 0.0165, Test Loss: 0.0206
```

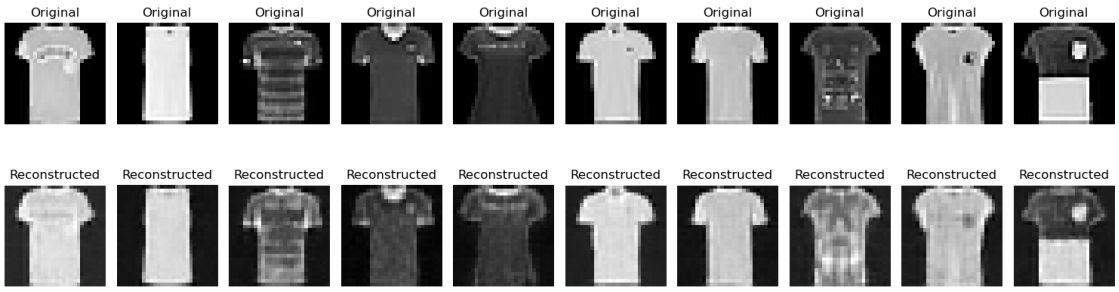
```
Epoch 59/60 - Train Loss: 0.0166, Test Loss: 0.0206
Epoch 60/60 - Train Loss: 0.0164, Test Loss: 0.0205
Training Conv_AE with latent dimension: 150
Epoch 1/60 - Train Loss: 0.2082, Test Loss: 0.0891
Epoch 2/60 - Train Loss: 0.0719, Test Loss: 0.0607
Epoch 3/60 - Train Loss: 0.0551, Test Loss: 0.0488
Epoch 4/60 - Train Loss: 0.0461, Test Loss: 0.0426
Epoch 5/60 - Train Loss: 0.0408, Test Loss: 0.0385
Epoch 6/60 - Train Loss: 0.0366, Test Loss: 0.0346
Epoch 7/60 - Train Loss: 0.0338, Test Loss: 0.0314
Epoch 8/60 - Train Loss: 0.0309, Test Loss: 0.0295
Epoch 9/60 - Train Loss: 0.0291, Test Loss: 0.0279
Epoch 10/60 - Train Loss: 0.0272, Test Loss: 0.0261
Epoch 11/60 - Train Loss: 0.0260, Test Loss: 0.0251
Epoch 12/60 - Train Loss: 0.0245, Test Loss: 0.0241
Epoch 13/60 - Train Loss: 0.0238, Test Loss: 0.0229
Epoch 14/60 - Train Loss: 0.0225, Test Loss: 0.0227
Epoch 15/60 - Train Loss: 0.0219, Test Loss: 0.0217
Epoch 16/60 - Train Loss: 0.0211, Test Loss: 0.0212
Epoch 17/60 - Train Loss: 0.0204, Test Loss: 0.0208
Epoch 18/60 - Train Loss: 0.0198, Test Loss: 0.0201
Epoch 19/60 - Train Loss: 0.0193, Test Loss: 0.0215
Epoch 20/60 - Train Loss: 0.0190, Test Loss: 0.0197
Epoch 21/60 - Train Loss: 0.0186, Test Loss: 0.0193
Epoch 22/60 - Train Loss: 0.0182, Test Loss: 0.0195
Epoch 23/60 - Train Loss: 0.0182, Test Loss: 0.0189
Epoch 24/60 - Train Loss: 0.0176, Test Loss: 0.0188
Epoch 25/60 - Train Loss: 0.0172, Test Loss: 0.0184
Epoch 26/60 - Train Loss: 0.0170, Test Loss: 0.0191
Epoch 27/60 - Train Loss: 0.0167, Test Loss: 0.0181
Epoch 28/60 - Train Loss: 0.0166, Test Loss: 0.0181
Epoch 29/60 - Train Loss: 0.0163, Test Loss: 0.0180
Epoch 30/60 - Train Loss: 0.0162, Test Loss: 0.0176
Epoch 31/60 - Train Loss: 0.0160, Test Loss: 0.0177
Epoch 32/60 - Train Loss: 0.0157, Test Loss: 0.0177
Epoch 33/60 - Train Loss: 0.0157, Test Loss: 0.0180
Epoch 34/60 - Train Loss: 0.0155, Test Loss: 0.0175
Epoch 35/60 - Train Loss: 0.0153, Test Loss: 0.0176
Epoch 36/60 - Train Loss: 0.0154, Test Loss: 0.0174
Epoch 37/60 - Train Loss: 0.0150, Test Loss: 0.0173
Epoch 38/60 - Train Loss: 0.0150, Test Loss: 0.0175
Epoch 39/60 - Train Loss: 0.0149, Test Loss: 0.0175
Epoch 40/60 - Train Loss: 0.0146, Test Loss: 0.0174
Epoch 41/60 - Train Loss: 0.0147, Test Loss: 0.0172
Epoch 42/60 - Train Loss: 0.0144, Test Loss: 0.0172
Epoch 43/60 - Train Loss: 0.0142, Test Loss: 0.0175
Epoch 44/60 - Train Loss: 0.0145, Test Loss: 0.0173
Epoch 45/60 - Train Loss: 0.0140, Test Loss: 0.0170
```

```
Epoch 46/60 - Train Loss: 0.0140, Test Loss: 0.0173
Epoch 47/60 - Train Loss: 0.0140, Test Loss: 0.0173
Epoch 48/60 - Train Loss: 0.0140, Test Loss: 0.0171
Epoch 49/60 - Train Loss: 0.0139, Test Loss: 0.0173
Epoch 50/60 - Train Loss: 0.0137, Test Loss: 0.0170
Epoch 51/60 - Train Loss: 0.0137, Test Loss: 0.0174
Epoch 52/60 - Train Loss: 0.0136, Test Loss: 0.0170
Epoch 53/60 - Train Loss: 0.0134, Test Loss: 0.0170
Epoch 54/60 - Train Loss: 0.0133, Test Loss: 0.0174
Epoch 55/60 - Train Loss: 0.0133, Test Loss: 0.0173
Epoch 56/60 - Train Loss: 0.0133, Test Loss: 0.0173
Epoch 57/60 - Train Loss: 0.0133, Test Loss: 0.0172
Epoch 58/60 - Train Loss: 0.0131, Test Loss: 0.0171
Epoch 59/60 - Train Loss: 0.0131, Test Loss: 0.0169
Epoch 60/60 - Train Loss: 0.0130, Test Loss: 0.0172
```

[50]:
```python
# TODO: Visualize the results
plt.figure(figsize=(12, 6))
for i, latent_dim in enumerate(latent_dims):
    plt.plot(train_losses[i], label=f'Train Loss (Latent Dim: {latent_dim})')
    plt.plot(test_losses[i], label=f'Test Loss (Latent Dim: {latent_dim})',␣
 ↪linestyle='--')
plt.title('Training and Test Losses for Different Latent Dimensions')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```



[51]:
```python
#TODO: Visualize the reconstructed images and the original images
def visualize_reconstruction(model, data_loader, device):
    model.eval()
    with tc.no_grad():
        X, _ = next(iter(data_loader))
        X = X.to(device)
        reconstructed = model(X)

    # Plot original and reconstructed images
    fig, axs = plt.subplots(2, 10, figsize=(15, 5))
    for i in range(10):
        axs[0, i].imshow(X[i].cpu().squeeze(), cmap='gray')
        axs[0, i].set_title("Original")
        axs[0, i].axis('off')

        axs[1, i].imshow(reconstructed[i].cpu().squeeze(), cmap='gray')
        axs[1, i].set_title("Reconstructed")
        axs[1, i].axis('off')

    plt.tight_layout()
    plt.show()
visualize_reconstruction(conv_ae_model, test_loader, device)
```



1.3.3 Task 3

A linear autoencoder aims to represent the data $X \in \mathbb{R}^{n \times m}$ in a new basis using only $d < m$ directions. The objective is to minimize the squared error between X and $D(E(X))$ where $E : \mathbb{R}^m \to \mathbb{R}^d$ is the encoder and $D : \mathbb{R}^d \to \mathbb{R}^m$ is the decoder:

$$\|D(E(X)) - X\|_2^2$$

In geometric terms, we want to find d axes along which most of the variance occurs which is exactly what Principal Component Analysis does. The optimal weights of a linear autoencoder with code dimension d thus span the same space as the first d principal components.

The PCA autoencoder just consists of 1 linear layer for the encoder and 1 linear layer for the decoder. The bias is necessary to subtract the mean value. Since we are dealing with three-dimensional images, we also have to flatten (when encoding) or unflatten (when decoding) the input.

[52]:
```
class PCA_AE(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(PCA_AE, self).__init__()

        # TODO: Initialize the layers of the autoencoder
        self.encoder = nn.Sequential(nn.Linear(input_dim, latent_dim))

        self.decoder = nn.Sequential(nn.Linear(latent_dim, input_dim))

    def forward(self, X):
        # TODO: Implement the forward call
        encoded = self.encoder(X)
        decoded = self.decoder(encoded)
        return decoded
```

For the training we can use the same function as for the convolutional autoencoder.

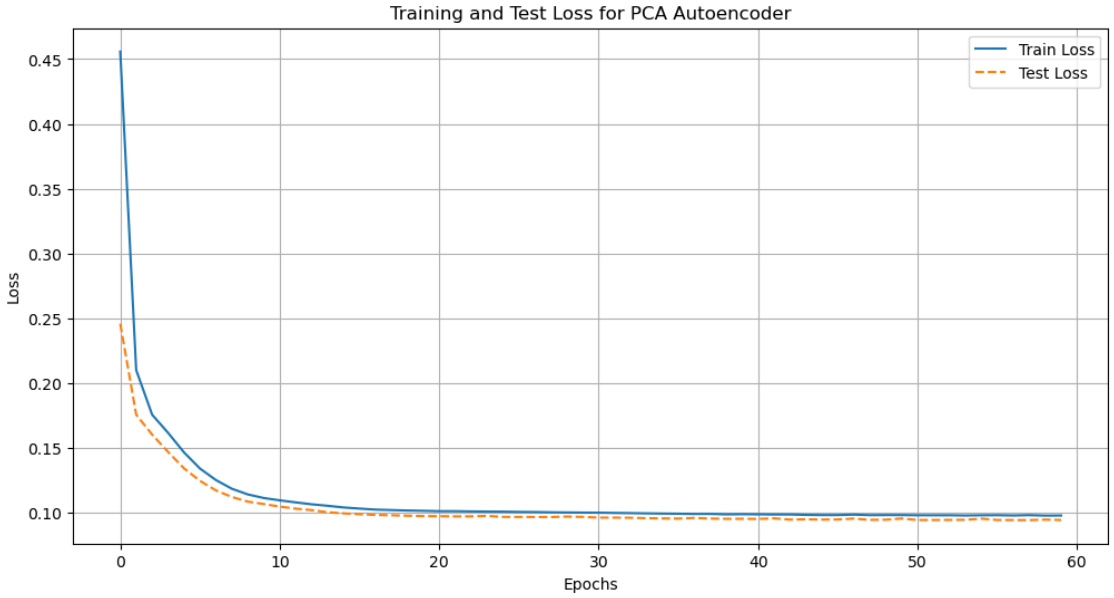
[53]:
```
n_epochs = 60
lr = 1e-3
# TODO: Train the PCA autoencoder model with a latent dimension of 10
latent_dim = 10
pca_ae_model = PCA_AE(input_dim=28*28, latent_dim=latent_dim)

pca_ae_model = pca_ae_model.to(device)

loss_function = nn.MSELoss()
optimizer = tc.optim.Adam(pca_ae_model.parameters(), lr=lr)

train_loss = np.zeros(n_epochs)
test_loss = np.zeros(n_epochs)

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

    epoch_loss = 0

    for X, _ in train_loader:
        X = X.to(device)
        X = X.view(X.size(0), -1)
        optimizer.zero_grad()
        reconstructed = pca_ae_model(X)
        loss = loss_function(reconstructed, X)
        loss.backward()
```

```
        optimizer.step()
        reconstructed = reconstructed.view(X.size(0), 1, 28, 28)

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

    train_loss[epoch - 1] = epoch_loss

    pca_ae_model.eval()

    with tc.no_grad():
        X, _ = next(iter(test_loader))
        X = X.view(X.size(0), -1)
        X = X.to(device)
        reconstructed = pca_ae_model(X)
        loss = loss_function(reconstructed, X)
        reconstructed = reconstructed.view(X.size(0), 1, 28, 28)


        test_loss[epoch - 1] = loss.item()
    print(f"Epoch {epoch}/{n_epochs} - Train Loss: {epoch_loss:.4f}, Test Loss:␣
↪{loss.item():.4f}")
```

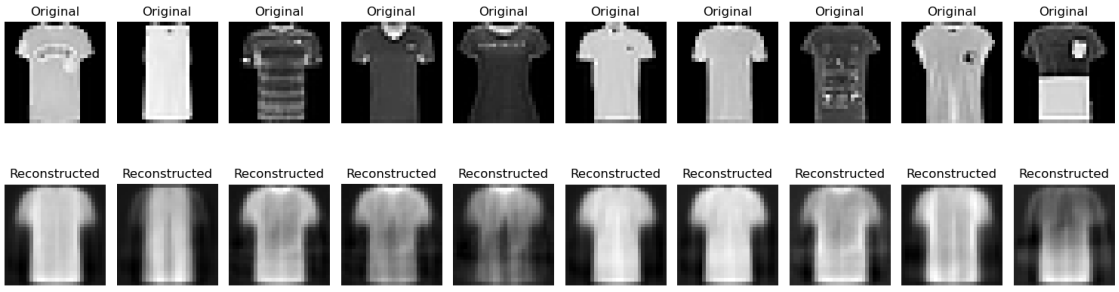
```
Epoch 1/60 - Train Loss: 0.4558, Test Loss: 0.2459
Epoch 2/60 - Train Loss: 0.2102, Test Loss: 0.1758
Epoch 3/60 - Train Loss: 0.1758, Test Loss: 0.1605
Epoch 4/60 - Train Loss: 0.1618, Test Loss: 0.1470
Epoch 5/60 - Train Loss: 0.1467, Test Loss: 0.1344
Epoch 6/60 - Train Loss: 0.1343, Test Loss: 0.1248
Epoch 7/60 - Train Loss: 0.1254, Test Loss: 0.1174
Epoch 8/60 - Train Loss: 0.1187, Test Loss: 0.1123
Epoch 9/60 - Train Loss: 0.1143, Test Loss: 0.1088
Epoch 10/60 - Train Loss: 0.1115, Test Loss: 0.1069
Epoch 11/60 - Train Loss: 0.1097, Test Loss: 0.1049
Epoch 12/60 - Train Loss: 0.1082, Test Loss: 0.1033
Epoch 13/60 - Train Loss: 0.1067, Test Loss: 0.1022
Epoch 14/60 - Train Loss: 0.1056, Test Loss: 0.1006
Epoch 15/60 - Train Loss: 0.1042, Test Loss: 0.0997
Epoch 16/60 - Train Loss: 0.1034, Test Loss: 0.0991
Epoch 17/60 - Train Loss: 0.1026, Test Loss: 0.0985
Epoch 18/60 - Train Loss: 0.1022, Test Loss: 0.0982
Epoch 19/60 - Train Loss: 0.1019, Test Loss: 0.0979
Epoch 20/60 - Train Loss: 0.1017, Test Loss: 0.0976
Epoch 21/60 - Train Loss: 0.1014, Test Loss: 0.0975
Epoch 22/60 - Train Loss: 0.1014, Test Loss: 0.0974
Epoch 23/60 - Train Loss: 0.1012, Test Loss: 0.0974
Epoch 24/60 - Train Loss: 0.1011, Test Loss: 0.0977
Epoch 25/60 - Train Loss: 0.1010, Test Loss: 0.0970
Epoch 26/60 - Train Loss: 0.1008, Test Loss: 0.0970
```

```
Epoch 27/60 - Train Loss: 0.1007, Test Loss: 0.0969
Epoch 28/60 - Train Loss: 0.1005, Test Loss: 0.0968
Epoch 29/60 - Train Loss: 0.1004, Test Loss: 0.0972
Epoch 30/60 - Train Loss: 0.1003, Test Loss: 0.0970
Epoch 31/60 - Train Loss: 0.1002, Test Loss: 0.0964
Epoch 32/60 - Train Loss: 0.1000, Test Loss: 0.0964
Epoch 33/60 - Train Loss: 0.0999, Test Loss: 0.0963
Epoch 34/60 - Train Loss: 0.0997, Test Loss: 0.0960
Epoch 35/60 - Train Loss: 0.0995, Test Loss: 0.0958
Epoch 36/60 - Train Loss: 0.0994, Test Loss: 0.0957
Epoch 37/60 - Train Loss: 0.0992, Test Loss: 0.0962
Epoch 38/60 - Train Loss: 0.0993, Test Loss: 0.0957
Epoch 39/60 - Train Loss: 0.0989, Test Loss: 0.0955
Epoch 40/60 - Train Loss: 0.0990, Test Loss: 0.0955
Epoch 41/60 - Train Loss: 0.0989, Test Loss: 0.0954
Epoch 42/60 - Train Loss: 0.0988, Test Loss: 0.0959
Epoch 43/60 - Train Loss: 0.0988, Test Loss: 0.0949
Epoch 44/60 - Train Loss: 0.0985, Test Loss: 0.0951
Epoch 45/60 - Train Loss: 0.0984, Test Loss: 0.0949
Epoch 46/60 - Train Loss: 0.0984, Test Loss: 0.0950
Epoch 47/60 - Train Loss: 0.0987, Test Loss: 0.0957
Epoch 48/60 - Train Loss: 0.0983, Test Loss: 0.0947
Epoch 49/60 - Train Loss: 0.0984, Test Loss: 0.0948
Epoch 50/60 - Train Loss: 0.0984, Test Loss: 0.0957
Epoch 51/60 - Train Loss: 0.0982, Test Loss: 0.0946
Epoch 52/60 - Train Loss: 0.0982, Test Loss: 0.0946
Epoch 53/60 - Train Loss: 0.0982, Test Loss: 0.0946
Epoch 54/60 - Train Loss: 0.0981, Test Loss: 0.0947
Epoch 55/60 - Train Loss: 0.0982, Test Loss: 0.0956
Epoch 56/60 - Train Loss: 0.0983, Test Loss: 0.0946
Epoch 57/60 - Train Loss: 0.0981, Test Loss: 0.0945
Epoch 58/60 - Train Loss: 0.0983, Test Loss: 0.0945
Epoch 59/60 - Train Loss: 0.0980, Test Loss: 0.0949
Epoch 60/60 - Train Loss: 0.0980, Test Loss: 0.0946
```

[54]:
```python
# TODO: Visualize the training and test loss
plt.figure(figsize=(12, 6))
plt.plot(train_loss, label='Train Loss')
plt.plot(test_loss, label='Test Loss', linestyle='--')
plt.title('Training and Test Loss for PCA Autoencoder')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```



[59]:
```python
# TODO: Visualize the reconstructed images and the original images
def visualize_pca_reconstruction(model, data_loader, device):
    model.eval()
    with tc.no_grad():
        X, _ = next(iter(data_loader))
        X = X.view(X.size(0), -1)
        X = X.to(device)
        reconstructed = model(X)
        reconstructed = reconstructed.view(X.size(0), 1, 28, 28)

    fig, axs = plt.subplots(2, 10, figsize=(15, 5))
    for i in range(10):
        axs[0, i].imshow(X[i].cpu().view(28, 28), cmap='gray')
        axs[0, i].set_title("Original")
        axs[0, i].axis('off')

        axs[1, i].imshow(reconstructed[i].cpu().squeeze(), cmap='gray')
        axs[1, i].set_title("Reconstructed")
        axs[1, i].axis('off')

    plt.tight_layout()
    plt.show()

visualize_pca_reconstruction(pca_ae_model, test_loader, device)
```

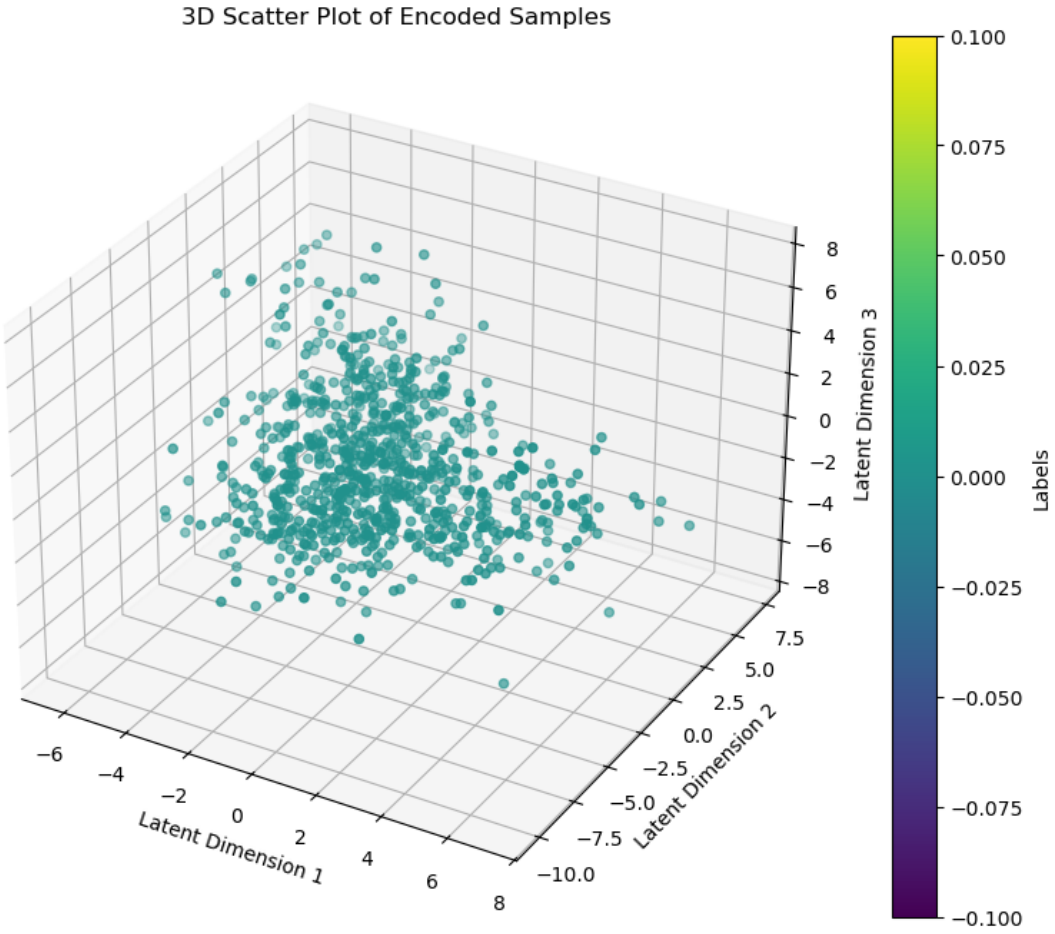


1.3.4 Task 4

[56]:
```
# TODO: Choose the convolutional autoencoder with latent dimension 3 and encode␣
↪800 samples
def encode_samples(model, data_loader, device, num_samples=800):
    model.eval()
    encoded_samples = []
    with tc.no_grad():
        for X, _ in data_loader:
            X = X.to(device)
            encoded = model.encoder(X)
            encoded_samples.append(encoded.cpu().numpy())
            if len(encoded_samples) * X.size(0) >= num_samples:
                break
    return np.concatenate(encoded_samples, axis=0)[:num_samples]

encoded_samples = encode_samples(conv_ae_model, train_loader, device,␣
↪num_samples=800)
```

[ ]:
```
# TODO: Make a 3D scatter plot of the code colored by the labels
def plot_3d_scatter(encoded_samples, labels):
    from mpl_toolkits.mplot3d import Axes3D
    import matplotlib.pyplot as plt

    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    scatter = ax.scatter(encoded_samples[:, 0], encoded_samples[:, 1],␣
↪encoded_samples[:, 2], c=labels, cmap='viridis', s=20)
    plt.colorbar(scatter, ax=ax, label='Labels')

    ax.set_xlabel('Latent Dimension 1')
    ax.set_ylabel('Latent Dimension 2')
    ax.set_zlabel('Latent Dimension 3')
    plt.title('3D Scatter Plot of Encoded Samples')
    plt.show()
```

```
plot_3d_scatter(encoded_samples, train_data.targets.numpy()[:800])
```



1.3.5 Obeservation

From the 3d scatter plot we can observe that the colors of the data points are close to each other, which indicates the separation is not well done. Moreover, the reconstructed images from the cell above can also inform us of this.

To mend this, we may add more latent dimensions to our PCA autodecoder.