Q3

In [ ]:
```
import torch
import torch.optim as optim
import torch.nn as nn
from torchvision import datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import statistics
```

Problem Idea

Instead of doing another regression and data problem, I will try something MNIST to see what the intrinsic dimension of the 0-9 digit images are. Before any analysis my guess is 10/11, depending on whether the blank white-space in MNIST images will be treated as part of the latent knowledge (I suspect it will be)

In [ ]:
```
from models import AutoencoderModel
```

Data Preparation

In [ ]:
```
device_name = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [ ]:
```
data = datasets.MNIST(root='./data', download=True)
images = data.data.float().div(255).unsqueeze(1).to(device_name)
```

In [ ]:
```
batch_size = 100
learning_images, validation_images = torch.utils.data.random_split(images, [50000, 10000])
learn_loader = DataLoader(learning_images, batch_size=batch_size, shuffle=True)
validation_loader = DataLoader(validation_images, batch_size=batch_size, shuffle=False)
```

In [44]:
```
test_data = datasets.MNIST(root='./data', train=False, download=True)
test_images = test_data.data.float().div(255).to(device_name)
test_images_loader = DataLoader(test_images, batch_size=1, shuffle=False)
```

Model Training Pre Checks

In [48]:
```
def train_model(model, learn_loader, validation_loader, optimiser, loss_function, num_epochs):
    learning_losses = []
    validation_losses = []

    for epoch in range(num_epochs):
        model.train()
        total_loss = 0.0
        for images in learn_loader:
            images = images.view(-1, 28 * 28).to(device_name)
            optimiser.zero_grad()
            reconstructed_images = model(images)
            loss = loss_function(reconstructed_images, images)
            loss.backward()
            optimiser.step()
            total_loss += loss.item()

        avg_learning_loss = total_loss / len(learn_loader)
        learning_losses.append((epoch + 1, avg_learning_loss))

        # Validation step
        model.eval()
        total_val_loss = 0.0
        with torch.no_grad():
            for val_images in validation_loader:
                val_images = val_images.view(-1, 28 * 28).to(device_name)
                reconstructed_val_images = model(val_images)
                val_loss = loss_function(reconstructed_val_images, val_images)
                total_val_loss += val_loss.item()

        avg_validation_loss = total_val_loss / len(validation_loader)
        validation_losses.append((epoch + 1, avg_validation_loss))

        print(f"Epoch [{epoch + 1}/{num_epochs}]")

    return learning_losses, validation_losses
```

In [55]:
```
model = AutoencoderModel(sizes=[784, 256, 64, 10, 64, 256, 784]).to(device_name)
```

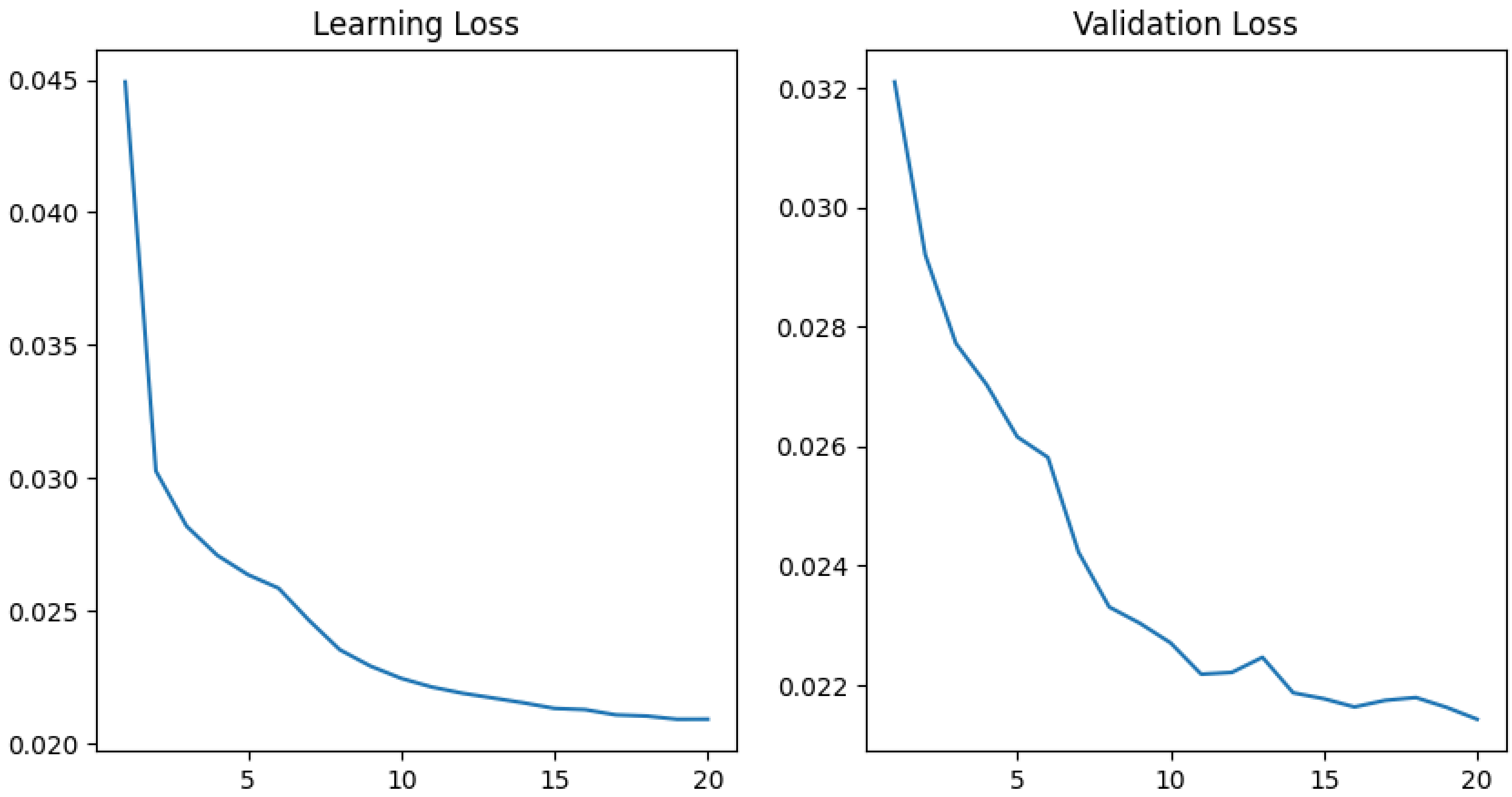
In [56]:
```
learning_rate = 0.005
num_epochs = 20
loss_function = nn.MSELoss()
optimiser = optim.Adam(model.parameters(), lr=learning_rate)
```

In [57]:
```
learning_losses, validation_losses = train_model(model, learn_loader, validation_loader, optimiser, loss_function, num_epochs)
```

```
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```

In [58]:
```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot([x[0] for x in learning_losses], [x[1] for x in learning_losses], label='Learning Loss')
plt.title('Learning Loss')
plt.subplot(1, 2, 2)
plt.plot([x[0] for x in validation_losses], [x[1] for x in validation_losses], label='Validation Loss')
plt.title('Validation Loss')
```

Out[58]:
```
Text(0.5, 1.0, 'Validation Loss')
```

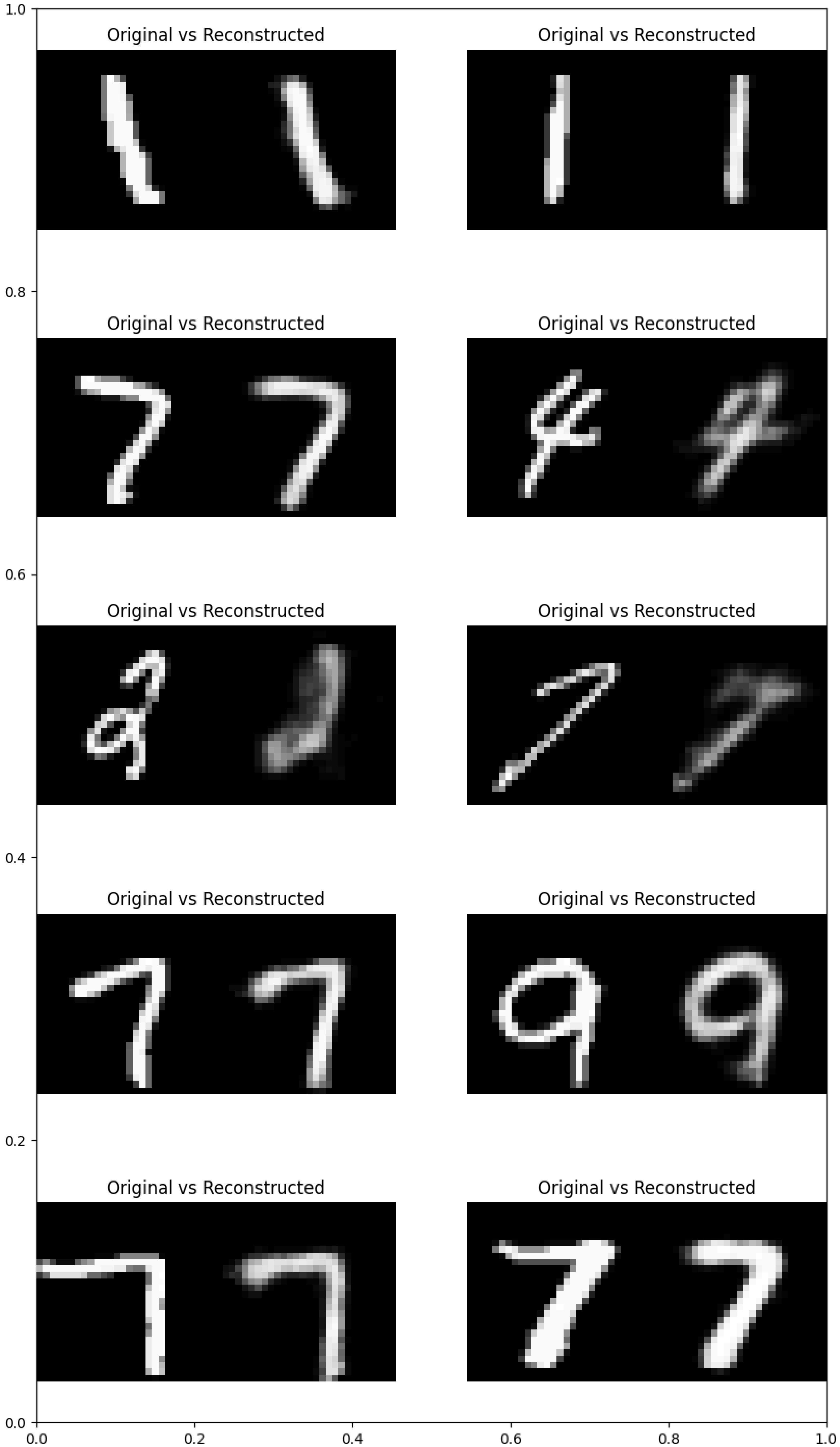


In [59]:
```
print(learning_losses[-1])
print(validation_losses[-1])
```

```
(20, 0.020914565831422805)
(20, 0.021427661776542664)
```

In [31]:
```
# From the above plots, the autoencoder seems to be learning well,
# We can explicitly reconstruct some images to get a visual confirmation

random_indices = torch.randint(0, len(images), (10,))
random_images = images[random_indices]
with torch.no_grad():
    reconstructed_images = model(random_images)

plt.figure(figsize=(10, 18))
plt.subplots_adjust(hspace=0.1)
plt.gcf().patch.set_facecolor('none')
plt.gca().set_facecolor('none')

random_images = random_images.view(-1, 28, 28)
reconstructed_images = reconstructed_images.reshape(random_images.shape)

for i in range(10):
    plt.subplot(5, 2, i + 1)
    stacked_images = torch.hstack((random_images[i].squeeze(), reconstructed_images[i]))
    plt.imshow(stacked_images.cpu(), cmap='gray')
```

```
    plt.title(f"Original vs Reconstructed")
    plt.axis('off')
```

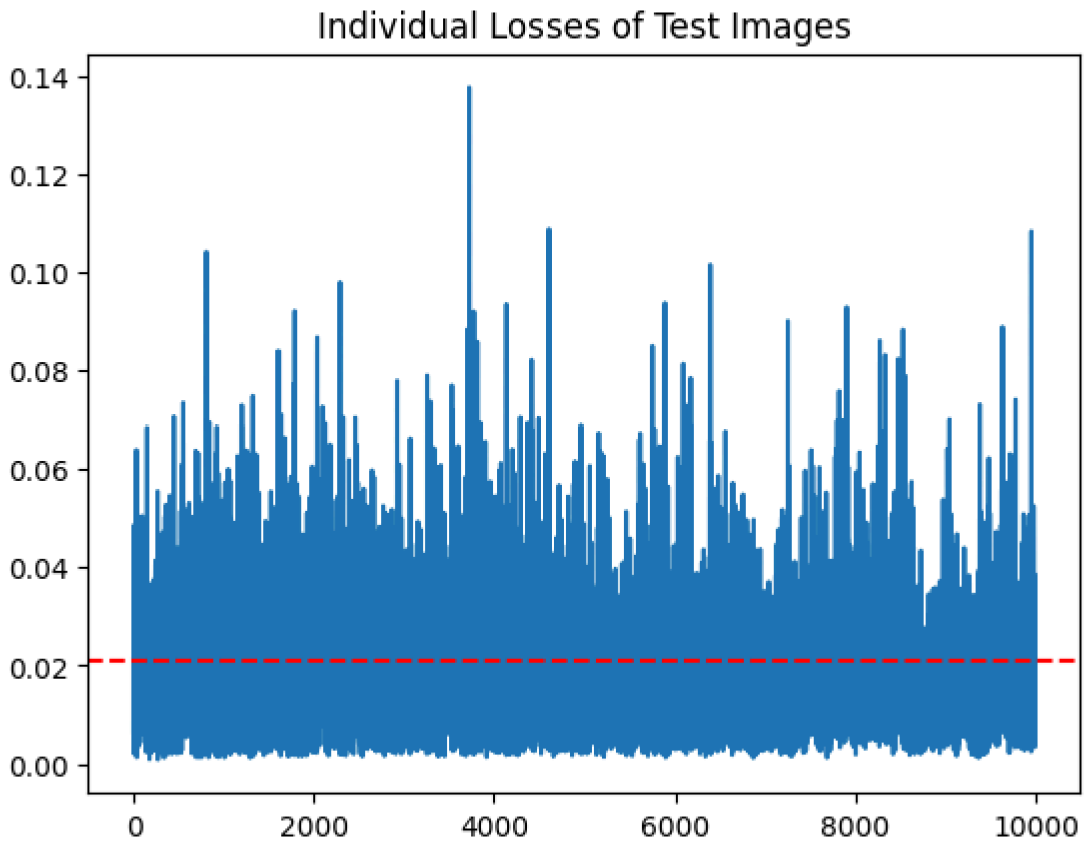


I am not sure how to feel about the above images, so we can do some further analysis. I will plot the images which individually show the worst loss to see if I can intuitively understand what the network is missing

In [36]:
```
model.eval()
individual_losses = []

with torch.no_grad():
    for index, image in enumerate(test_images_loader):
        image = image.view(-1, 28 * 28)
        reconstructed_image = model(image).reshape(28, 28)
        loss = loss_function(reconstructed_image, image.reshape(28, 28))
        individual_losses.append((index, loss.item()))

mean = statistics.mean([x[1] for x in individual_losses])
std = statistics.stdev([x[1] for x in individual_losses])
print(f"Average Loss: {mean:.4f}, Standard Deviation: {std:.4f}")
plt.plot(*zip(*individual_losses), label='Individual Losses')
plt.axhline(mean, color='red', linestyle='--', label='Mean Loss')
plt.title('Individual Losses of Test Images')
```

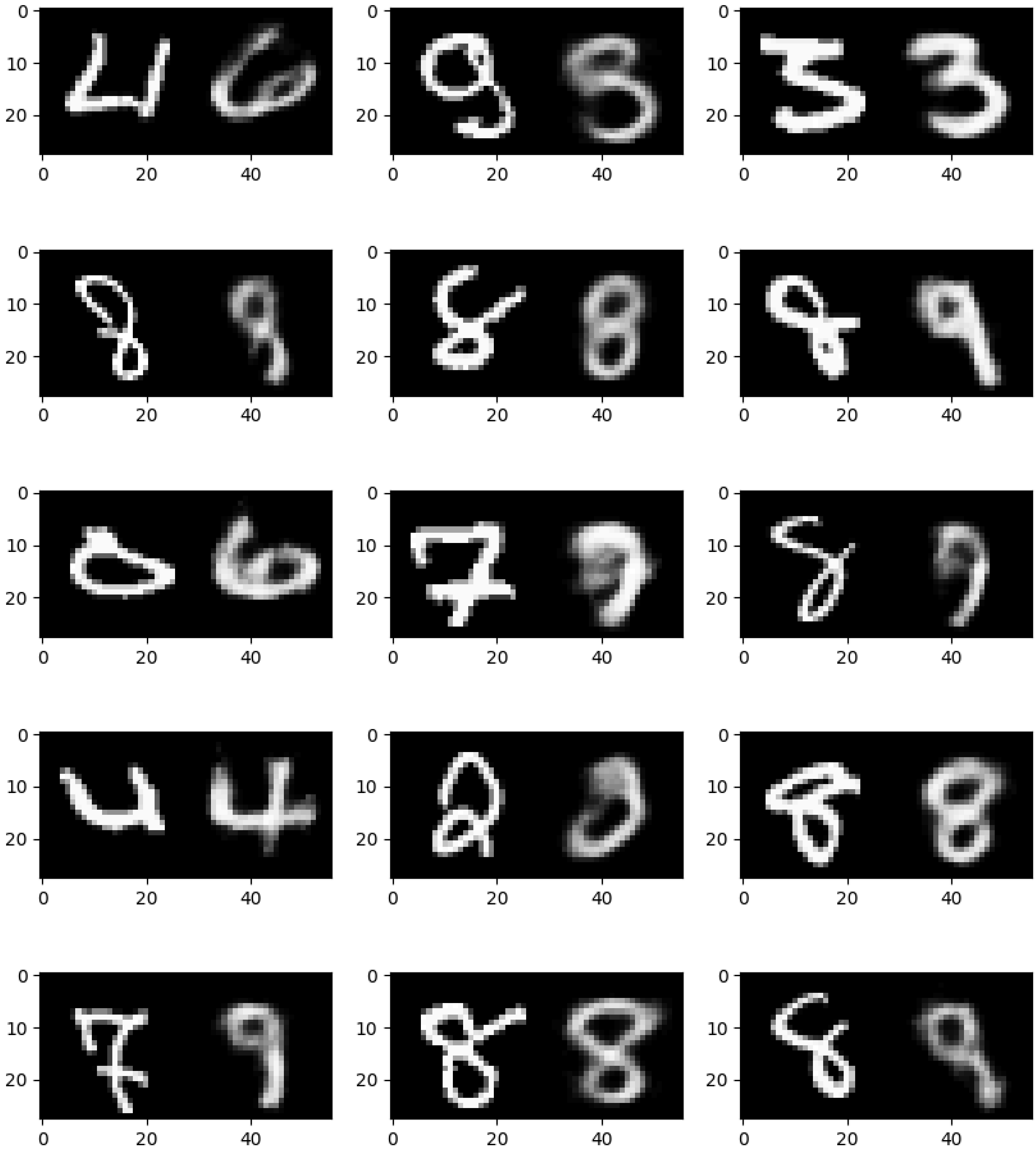
```
Average Loss: 0.0211, Standard Deviation: 0.0129
```

Out[36]:
```
Text(0.5, 1.0, 'Individual Losses of Test Images')
```



In [ ]:
```
loss_benchmark = mean + 3 * std
bad_images = []

with torch.no_grad():
    for index, image in enumerate(test_images_loader):
        image = image.view(-1, 28 * 28)
        reconstructed_image = model(image)

        loss = loss_function(reconstructed_image, image)
        if loss.item() > loss_benchmark:
            bad_images.append(index)

        if len(bad_images) >= 15:
            break
```

In [40]:
```
plt.figure(figsize=(10, 12))
plt.subplots_adjust(hspace=0.1)
num_subplots = 15

for i in range(num_subplots):
    plt.subplot(5, 3, i + 1)
    image = test_images[bad_images[i]]
    reconstructed_image = model(image.view(-1, 28 * 28)).reshape(28, 28).detach()
    stacked_images = torch.hstack((image, reconstructed_image)).cpu()
    plt.imshow(stacked_images.cpu(), cmap='gray')
```



Intuitively I feel like the latent may not be enough to capture the serious features like loops and 4/6 slashes. Could also be a training issue, because I only did 20 epochs. Lets save the model as a base reference

In [78]:
```
torch.save(model.state_dict(), 'model-weights/AutoencoderModel-l10.pth')
```

Train With Varying Latent Sizes

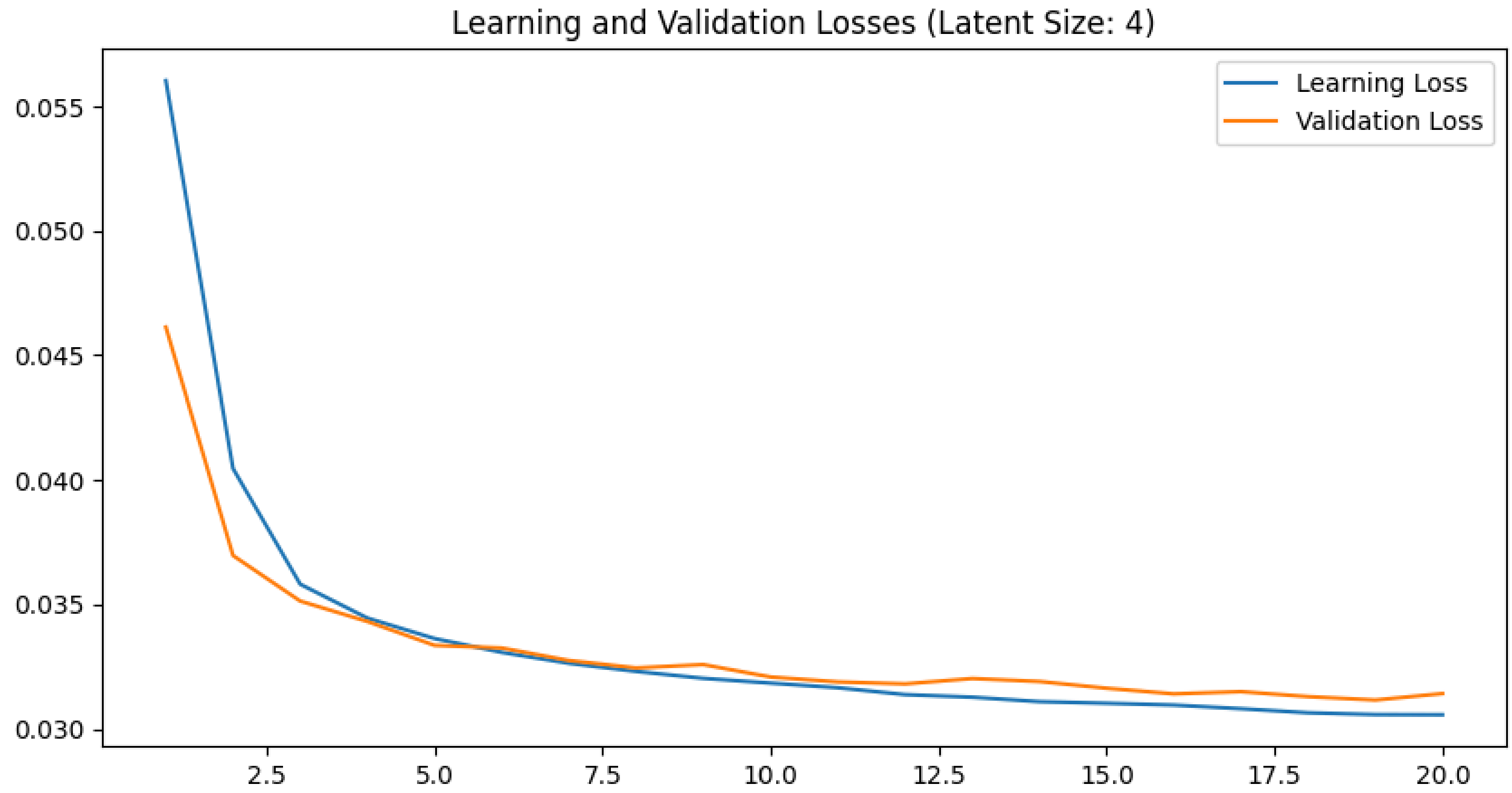
In [49]:
```
num_epochs = 20
learning_rate = 0.005
latent_sizes = [4, 6, 8, 12, 14, 16, 18, 20] # we already have 10
```

In [50]:
```
lowest_loss_reached = []
```

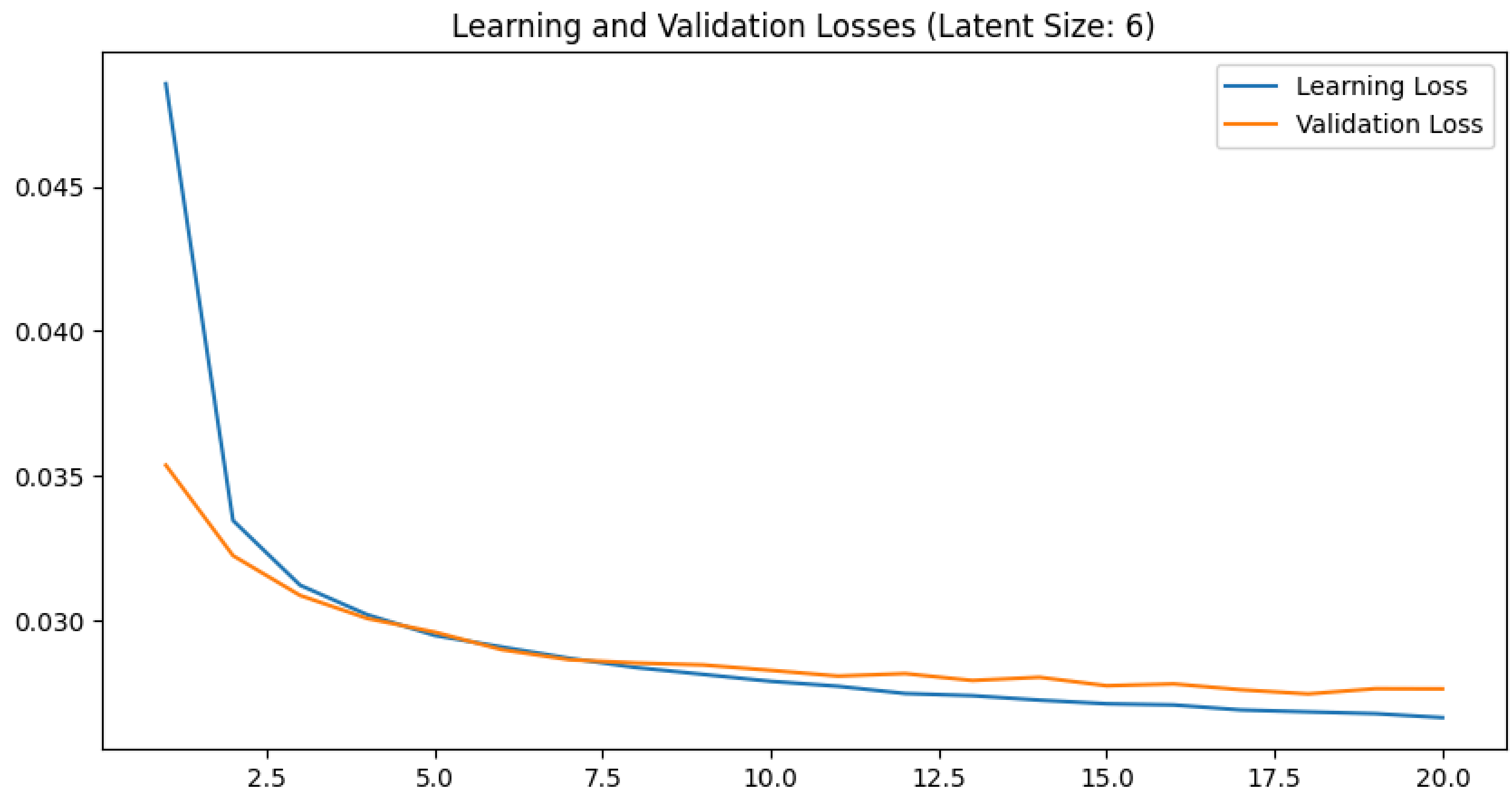
In [51]:
```
for latent_size in latent_sizes:
    print(f"Training model with latent size: {latent_size}")
    model = AutoencoderModel(sizes=[784, 256, 64, latent_size, 64, 256, 784]).to(device_name)
    optimiser = optim.Adam(model.parameters(), lr=learning_rate)

    learning_losses, validation_losses = train_model(model, learn_loader, validation_loader, optimiser, loss_function, num_epc

    # Record the last learning and validation losses for each latent size
    l_loss = learning_losses[-1][1]
    v_loss = validation_losses[-1][1]
    lowest_loss_reached.append((latent_size, l_loss, v_loss))

    plt.figure(figsize=(10, 5))
    plt.plot([x[0] for x in learning_losses], [x[1] for x in learning_losses], label='Learning Loss')
    plt.plot([x[0] for x in validation_losses], [x[1] for x in validation_losses], label='Validation Loss')
    plt.title(f'Learning and Validation Losses (Latent Size: {latent_size})')
    plt.legend()
```

```
    plt.show()

    torch.save(model.state_dict(), f'model-weights/AutoencoderModel-l{latent_size}.pth')
```

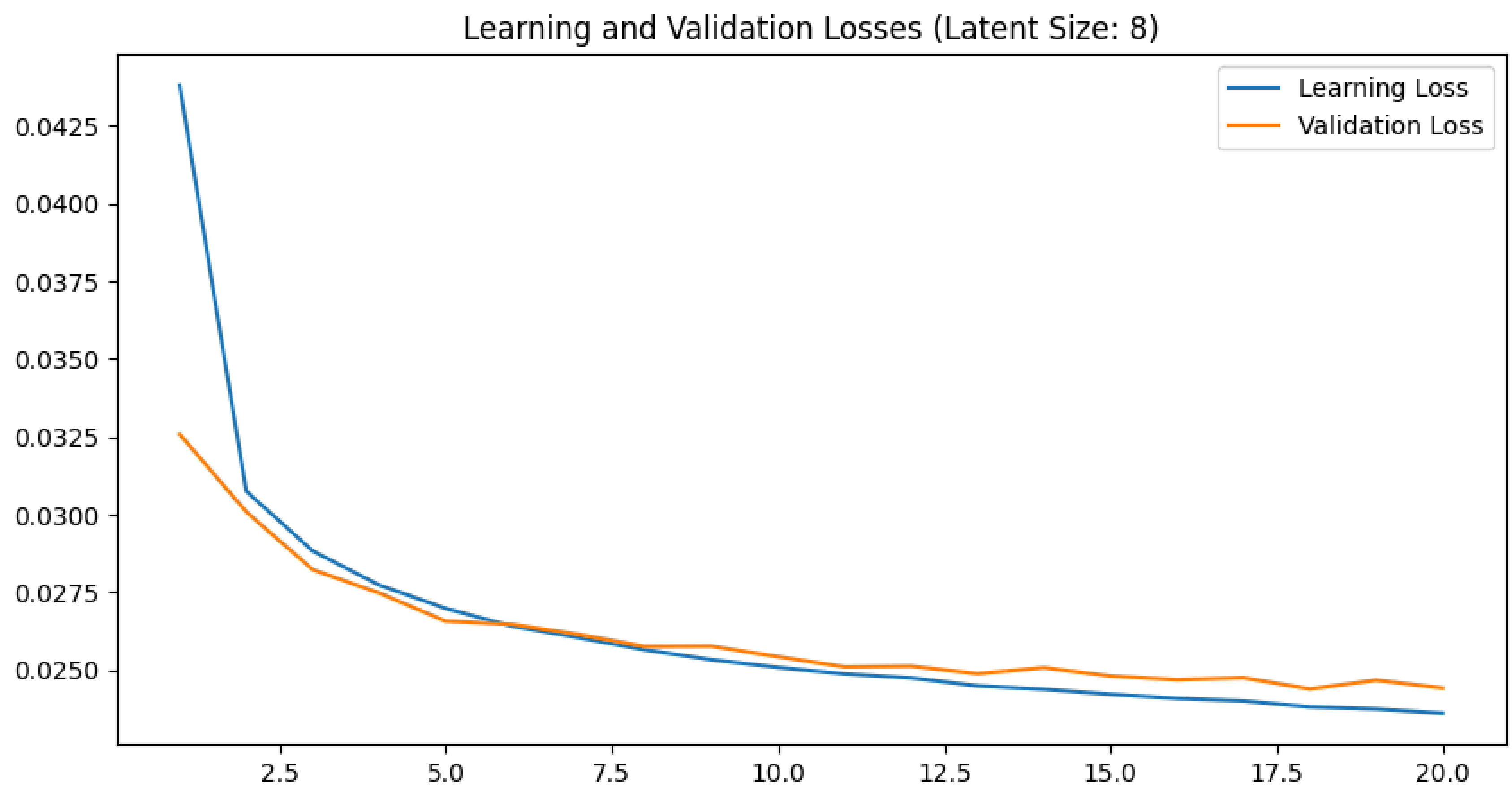
```
Training model with latent size: 4
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



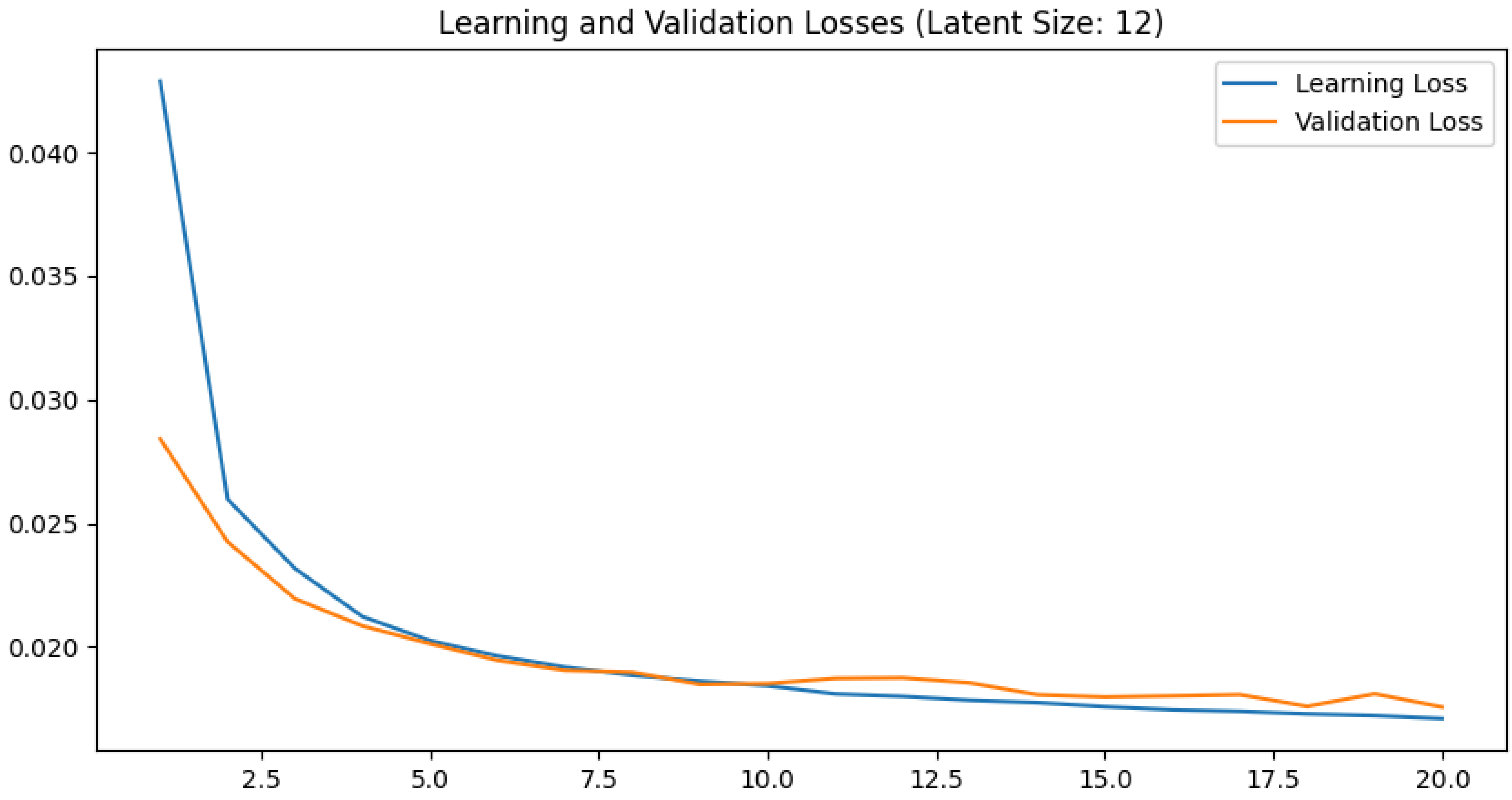

```
Training model with latent size: 6
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



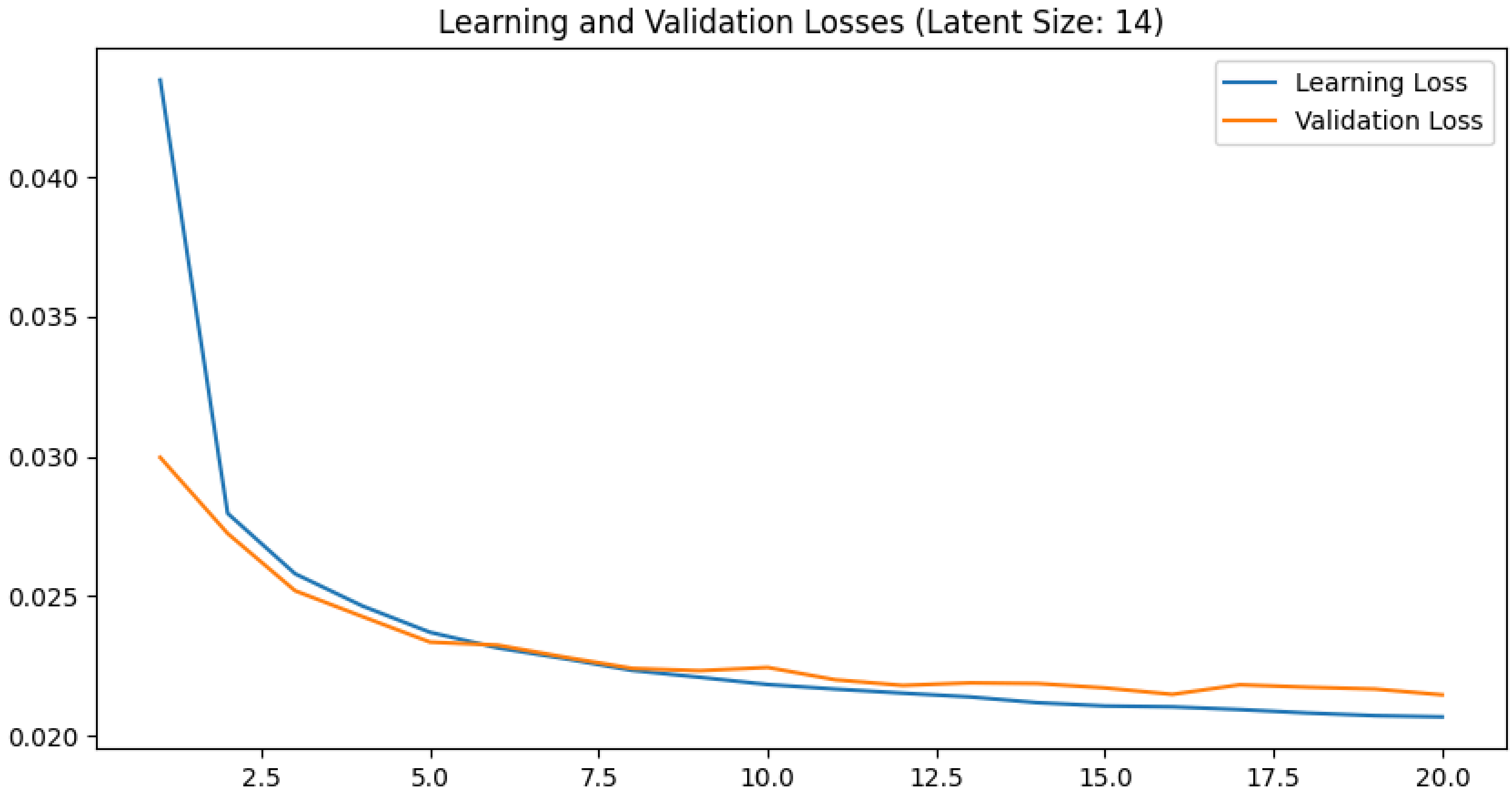

```
Training model with latent size: 8
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



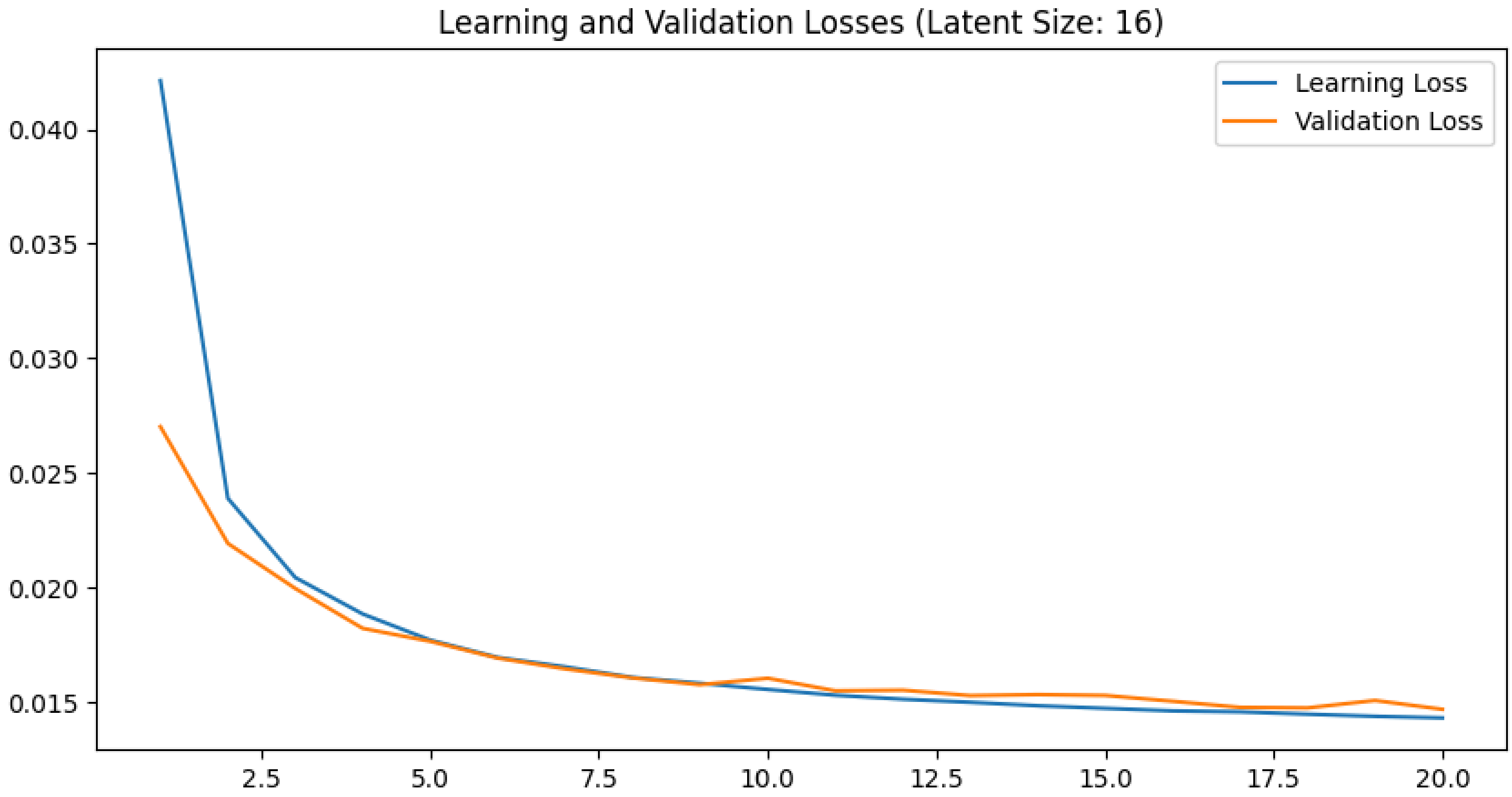
```
Training model with latent size: 12
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



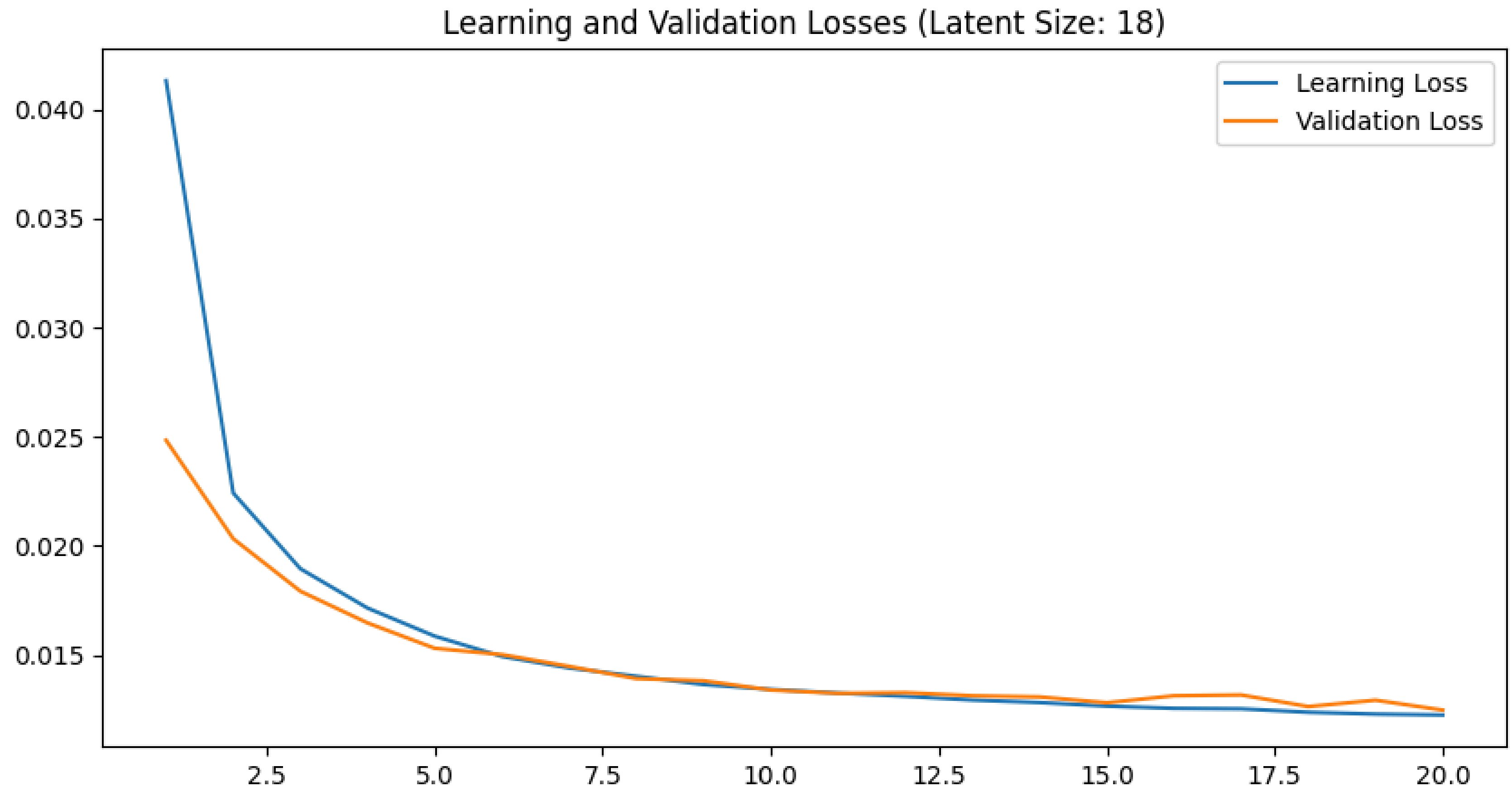

```
Training model with latent size: 14
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



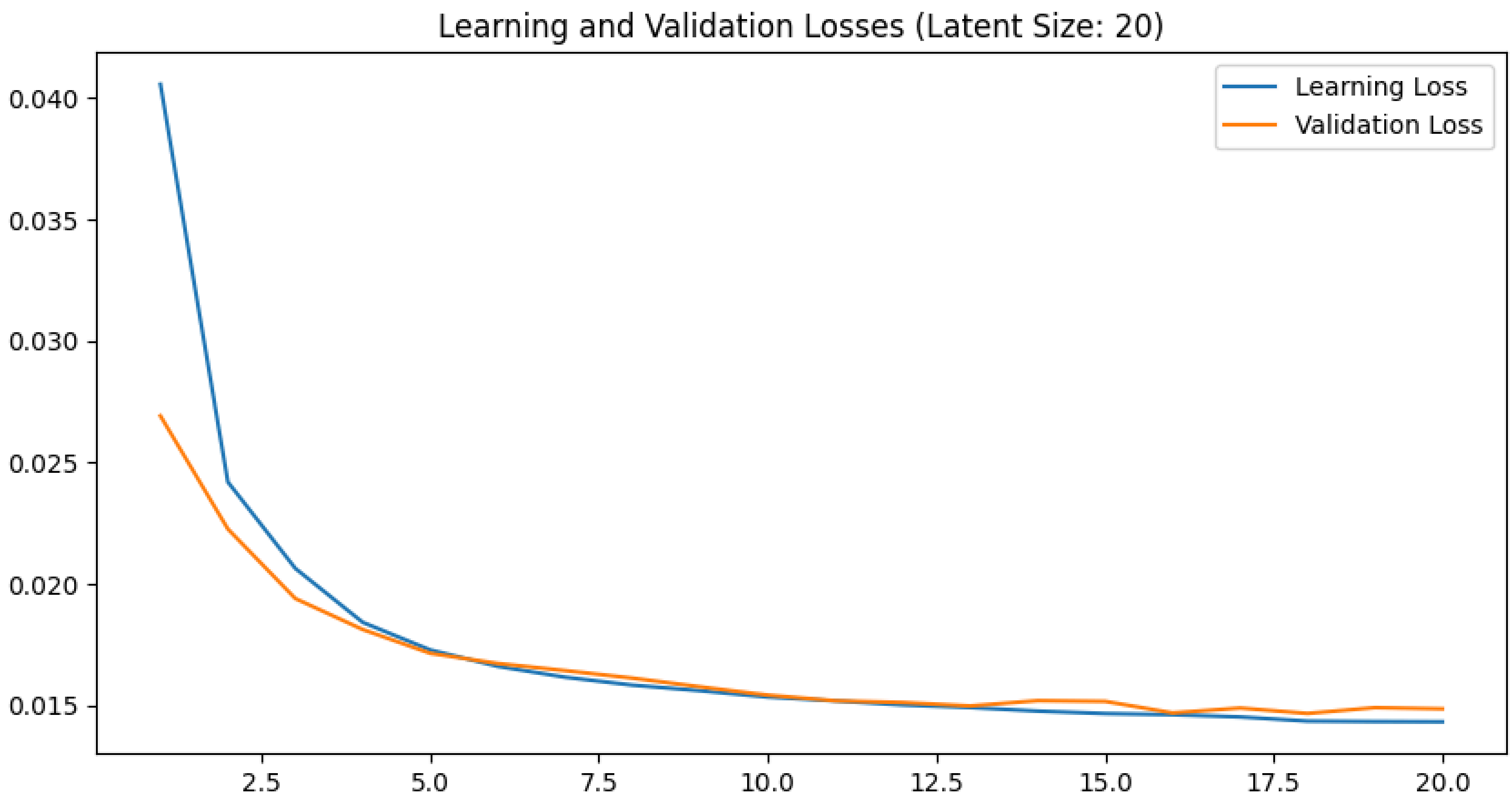

```
Training model with latent size: 16
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



```
Training model with latent size: 18
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



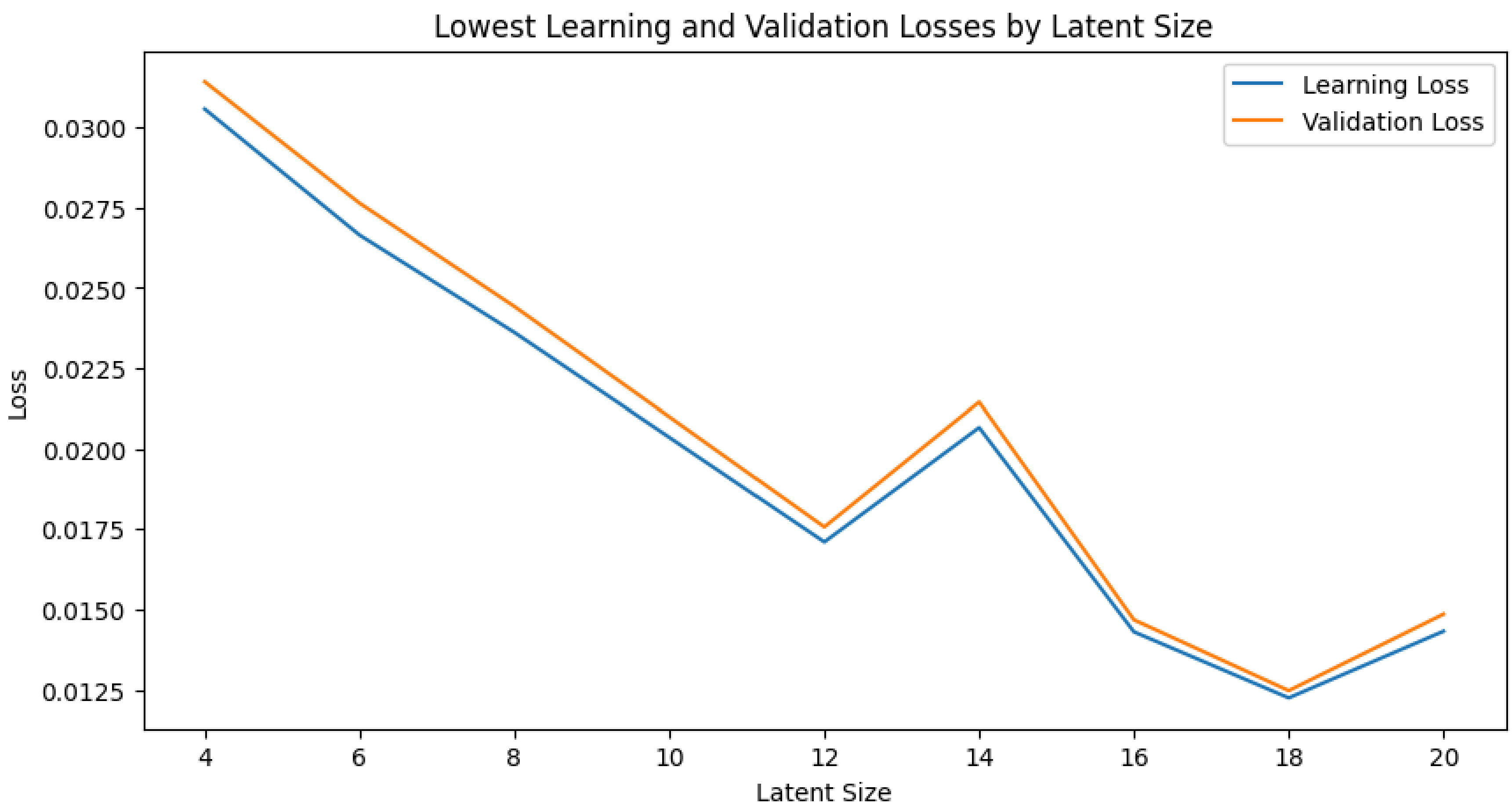

```
Training model with latent size: 20
Epoch [1/20]
Epoch [2/20]
Epoch [3/20]
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]
```



In [52]:

```
# Now We plot the lowest learning and validation losses for each latent size
plt.figure(figsize=(10, 5))
plt.plot([x[0] for x in lowest_loss_reached], [x[1] for x in lowest_loss_reached], label='Learning Loss')
plt.plot([x[0] for x in lowest_loss_reached], [x[2] for x in lowest_loss_reached], label='Validation Loss')
plt.title('Lowest Learning and Validation Losses by Latent Size')
plt.xlabel('Latent Size')
plt.ylabel('Loss')
plt.legend()
```

Out[52]: <matplotlib.legend.Legend at 0x1d159d97b60>



Wow This is a lot more interesting than I anticipated...

In [54]:

```
# write the lowest loss reached to a file
with open('lowest_loss', 'w') as f:
    for latent_size, l_loss, v_loss in lowest_loss_reached:
        f.write(f"Latent Size: {latent_size}, Learning Loss: {l_loss:.4f}, Validation Loss: {v_loss:.4f}\n")
```

Looking Inside the Autoencoder

In [67]:

```
labels = data.targets.to(device_name)
```

In [87]:

```
# Take out 5 images from each class 0-9
counts = [0] * 10
counted_images = [[] for _ in range(10)]

for image, label in zip(images, labels):
    if counts[label] < 5:
        counted_images[label].append(image)
```

```python
        counts[label] += 1
    if all(c >= 5 for c in counts):
        break
```

In [88]:
```python
model = AutoencoderModel(sizes=[784, 256, 64, 10, 64, 256, 784]).to(device_name)
model.load_state_dict(torch.load('model-weights/AutoencoderModel-l10.pth', map_location=device_name), strict=True)
```

Out[88]: <All keys matched successfully>

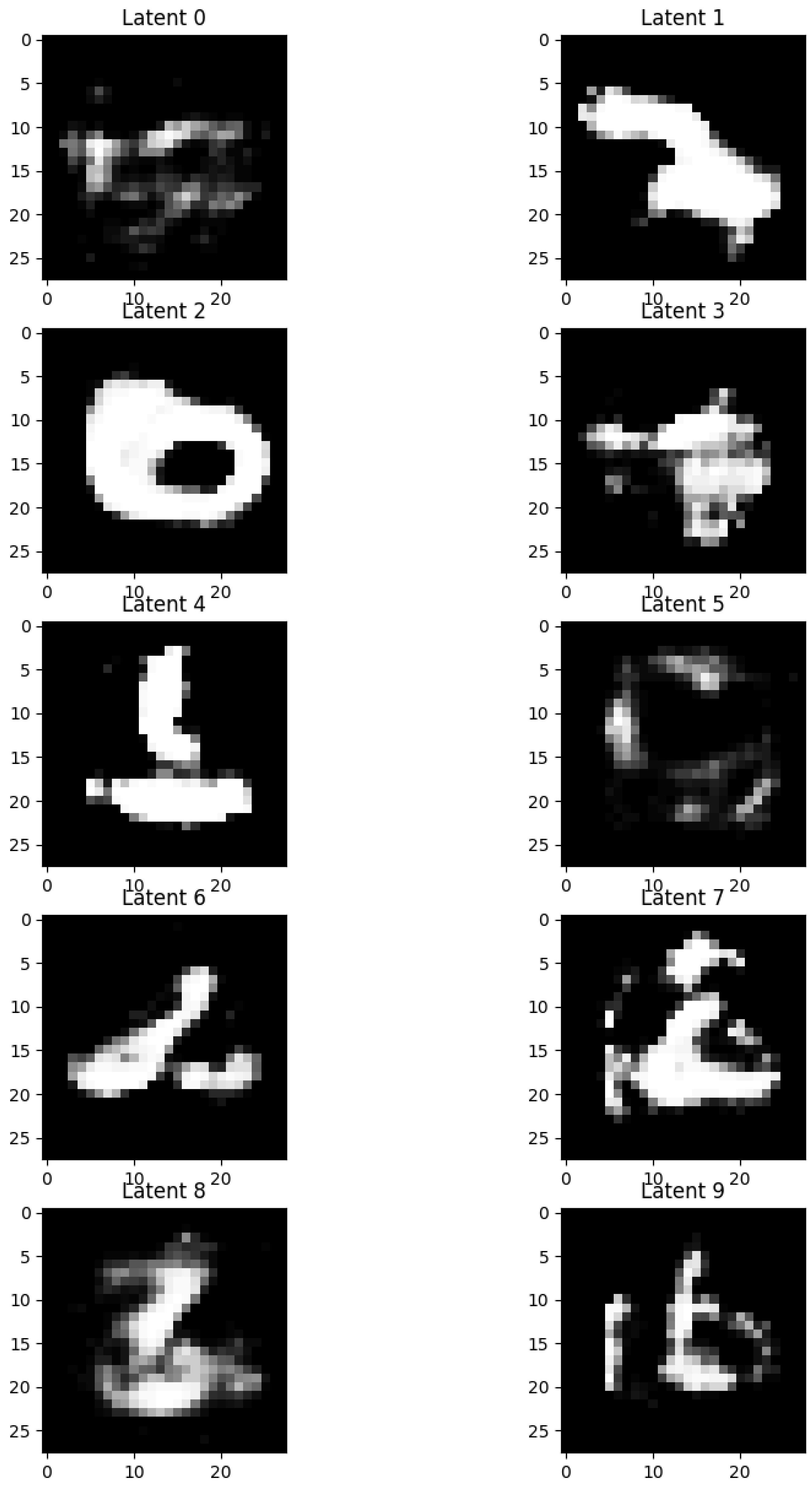
In [120…
```python
# Find the latents of the counted images
counted_latents = []
with torch.no_grad():
    for label in range(10):
        all_images = torch.stack(counted_images[label])
        latents = model.encoder_forward(all_images)
        counted_latents.append(latents)
```

In [104…
```python
# Reconstruct each latent to see what each feature represents. This is a bit of a hack, but it works
def plot_each_latent_individually(model, size):
    one_hot_latents = torch.zeros((size, size)).to(device_name)
    for i in range(size):
        one_hot_latents[i][i] = 10

    reconstructed_images = model.decoder_forward(one_hot_latents).reshape(-1, 28, 28)

    plt.figure(figsize=(10, size // 2 * 3))
    plt.subplots_adjust(hspace=0.2)
    for i in range(size):
        plt.subplot(size // 2, 2, i + 1)
        image = reconstructed_images[i].detach().cpu()
        plt.imshow(image, cmap='gray')
        plt.title(f"Latent {i}")
```

Plot the image and its latent values to see how it coincides with the actual reconstructed images

Sometimes a single latent will try to capture the entire number (for example the latent number 4 out of 4 so we will see a peak in the latent vector for the image of 2...)

In [105…
```python
model = AutoencoderModel(sizes=[784, 256, 64, 10, 64, 256, 784]).to(device_name)
model.load_state_dict(torch.load('model-weights/AutoencoderModel-l10.pth', map_location=device_name), strict=True)
plot_each_latent_individually(model, 10)
```



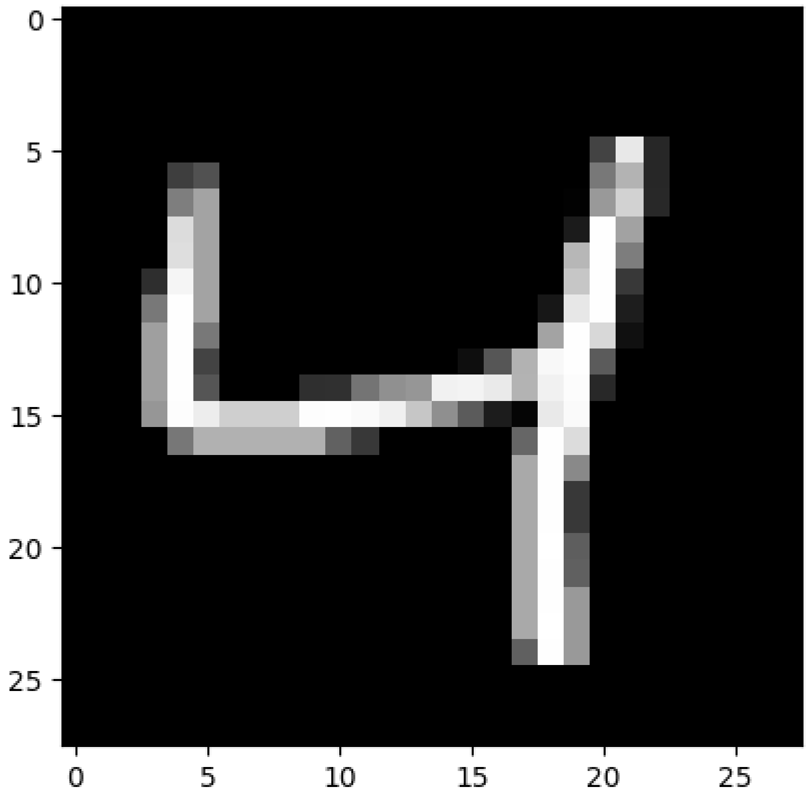
In [106…

```
i = 4
j = 0
print(counted_latents[i][j])
plt.imshow(counted_images[i][j][0].cpu(), cmap='gray')
```

```
tensor([0.0000, 5.1707, 3.3791, 3.7886, 1.1896, 4.5254, 2.3126, 0.0000, 0.0000,
        0.0000], device='cuda:0')
```

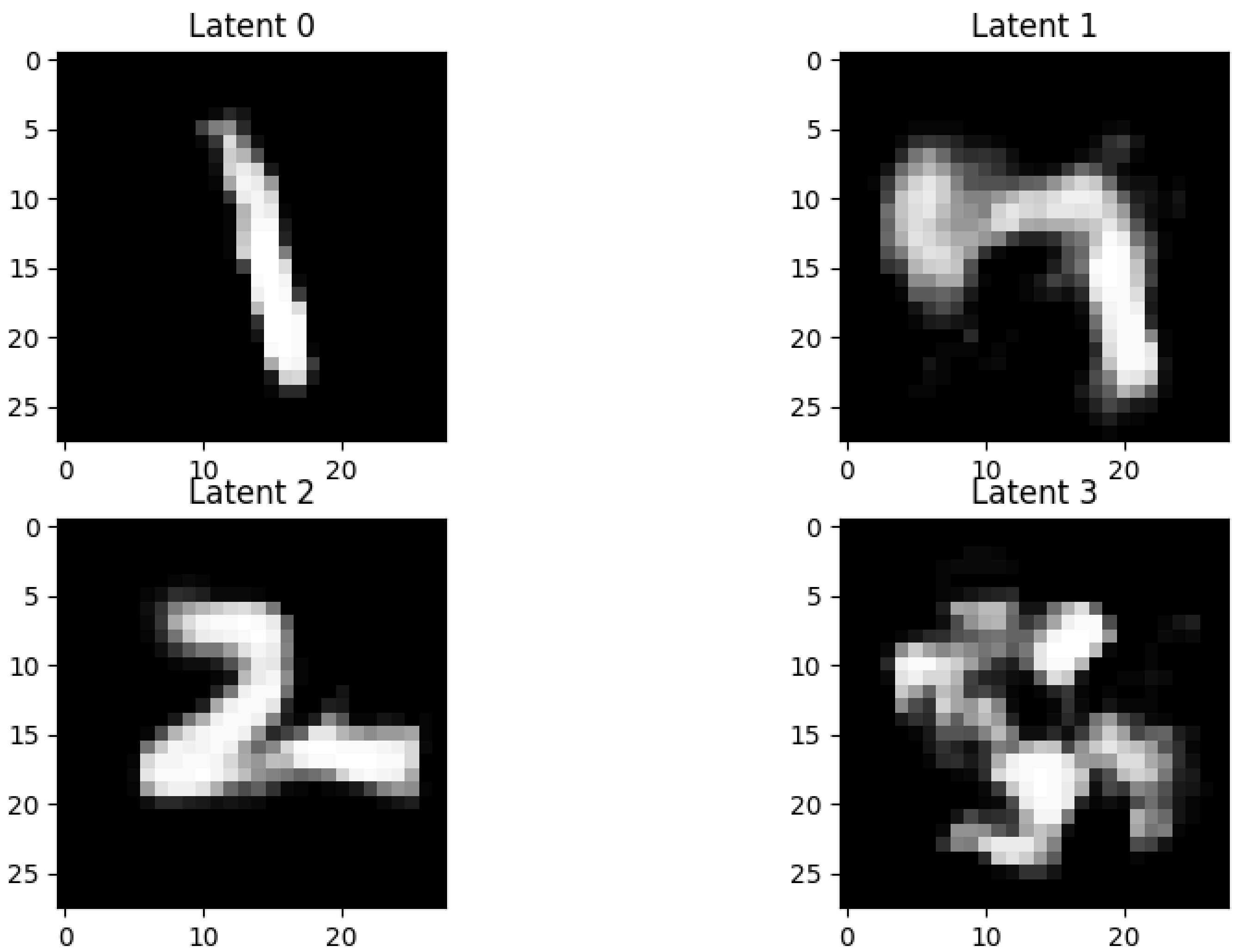
Out[106…

```
<matplotlib.image.AxesImage at 0x1d1519fa840>
```



```
In [107…: ### Try for Models with latent 4 and 12 and 18 as well
```

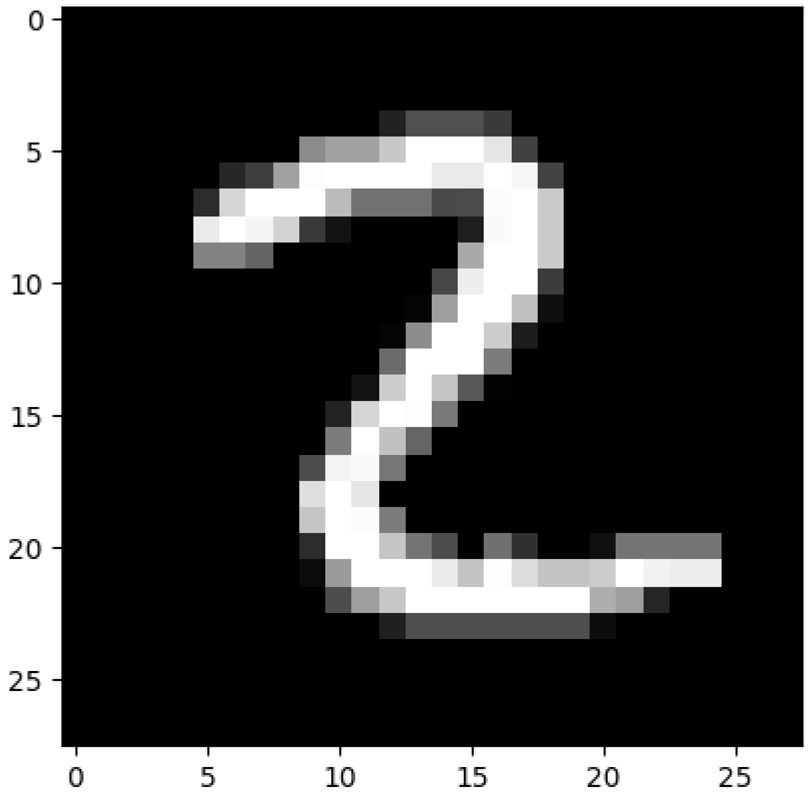
```
In [119…: model = AutoencoderModel(sizes=[784, 256, 64, 4, 64, 256, 784]).to(device_name)
model.load_state_dict(torch.load('model-weights/AutoencoderModel-14.pth', map_location=device_name), strict=True)
plot_each_latent_individually(model, 4)
```




```
In [121…: i = 2
j = 4
print(counted_latents[i][j])
plt.imshow(counted_images[i][j][0].cpu(), cmap='gray')
```

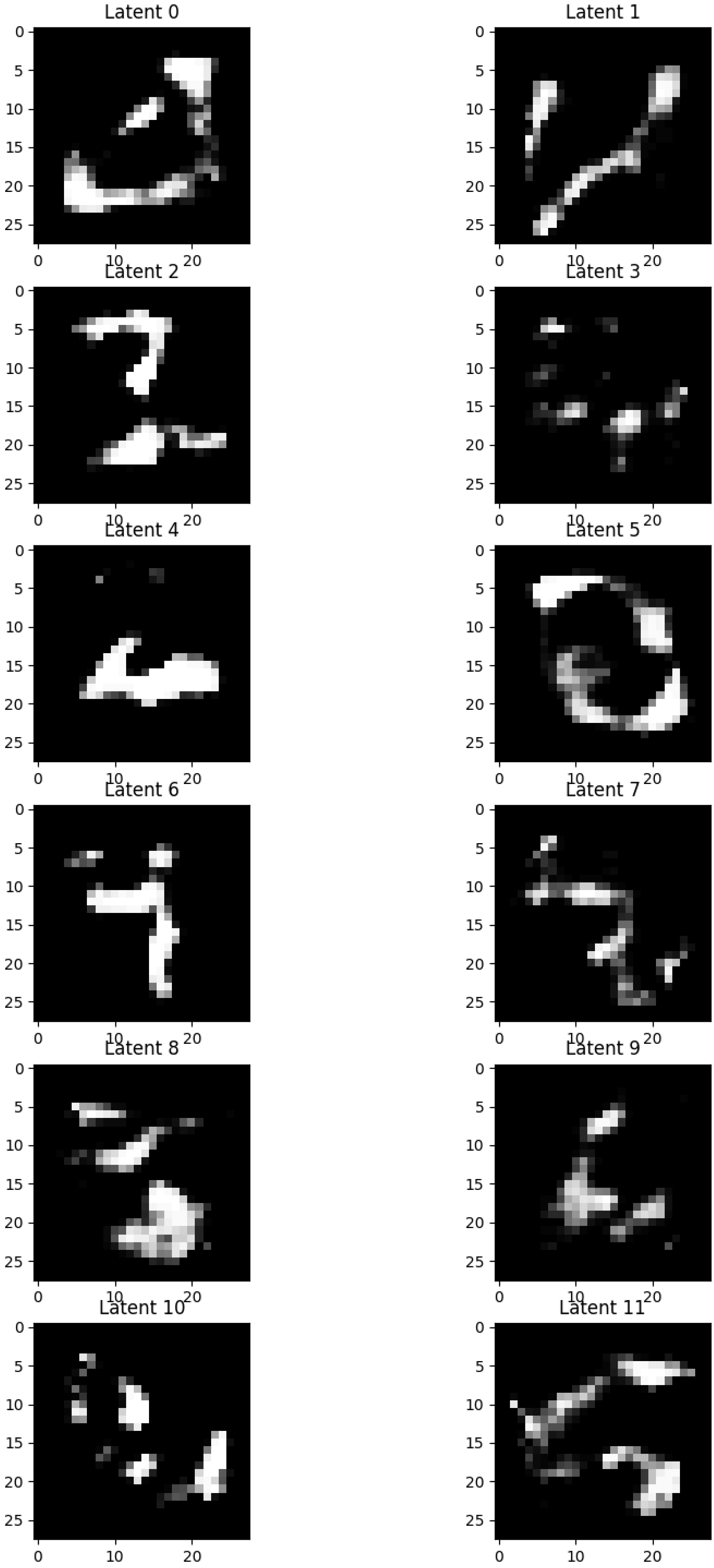
```
tensor([1.9741, 0.0931, 3.1668, 0.0000], device='cuda:0')
```

```
Out[121…: <matplotlib.image.AxesImage at 0x1d17326c830>
```



In [116…

```
model = AutoencoderModel(sizes=[784, 256, 64, 12, 64, 256, 784]).to(device_name)
model.load_state_dict(torch.load('model-weights/AutoencoderModel-l12.pth', map_location=device_name), strict=True)
plot_each_latent_individually(model, 12)
```


Latent 0
Latent 1
Latent 2
Latent 3
Latent 4
Latent 5
Latent 6
Latent 7
Latent 8
Latent 9
Latent 10
Latent 11

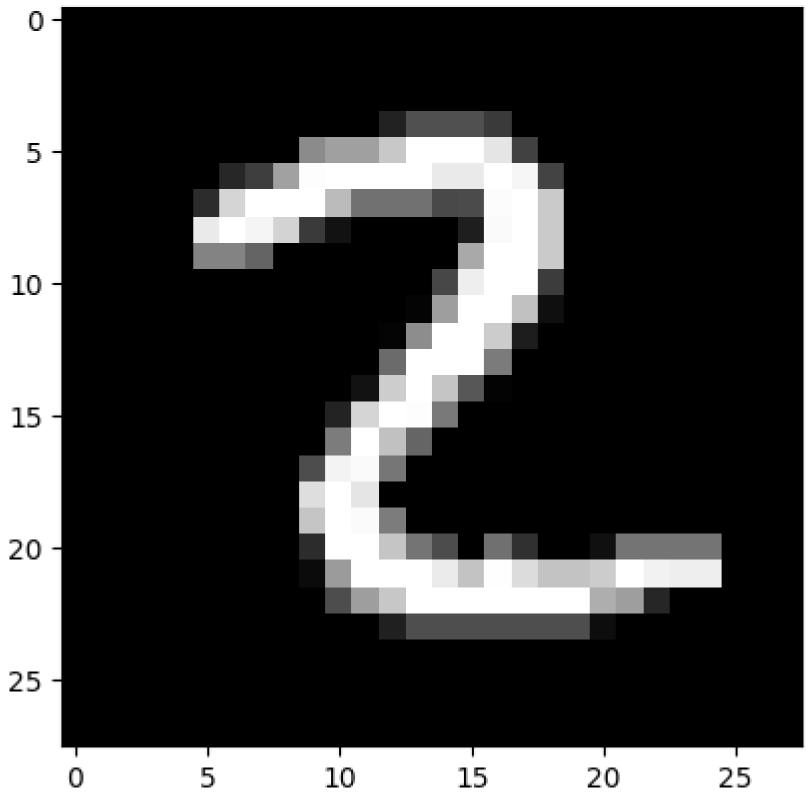
In [112…

```python
i = 2
j = 4
print(counted_latents[i][j])
plt.imshow(counted_images[i][j][0].cpu(), cmap='gray')

# Notice the really large component of the 2nd latent vector, because the second latent (above) literally just captures the im
# of a 2
```

```
tensor([0.0000, 5.2602, 1.1655, 1.6448, 6.0462, 3.9344, 3.5568, 0.0000, 0.0000,
        0.0000], device='cuda:0')
```

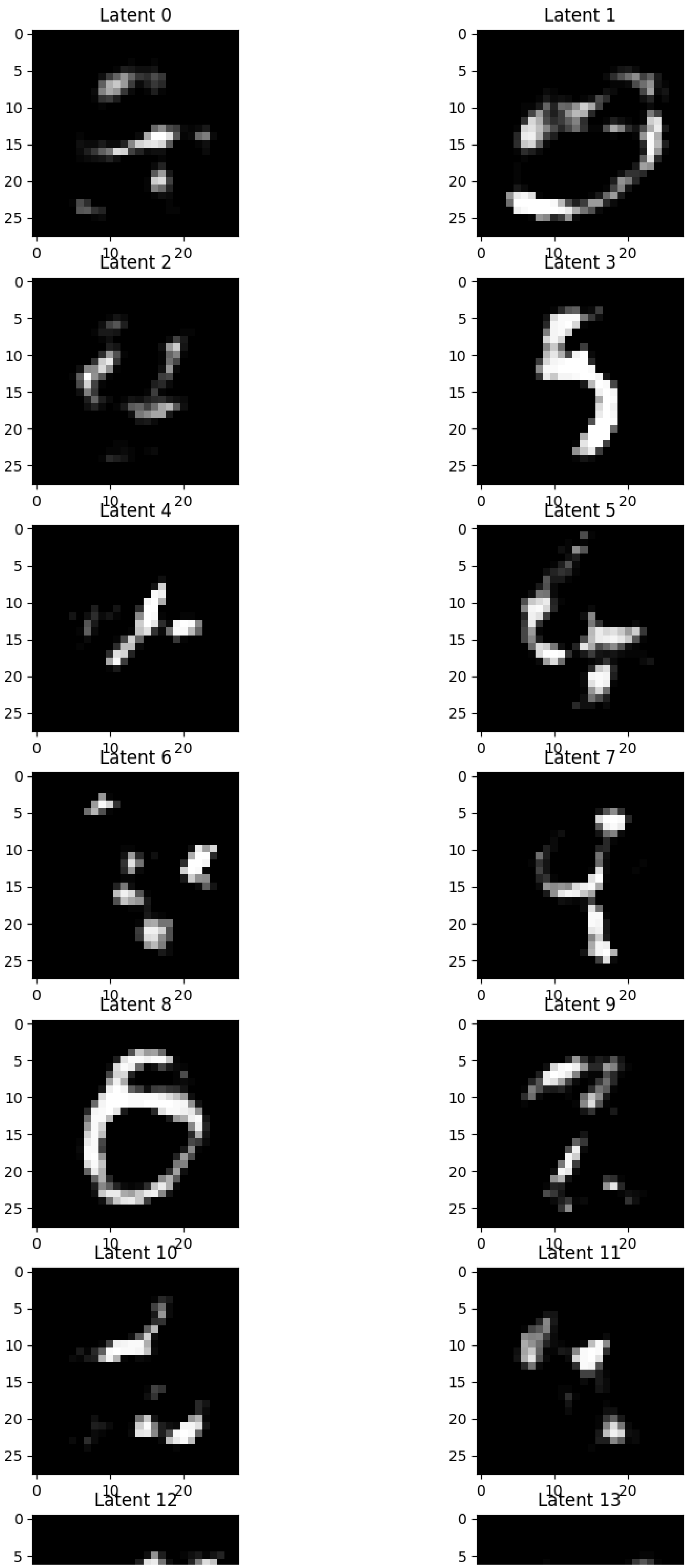
Out[112…

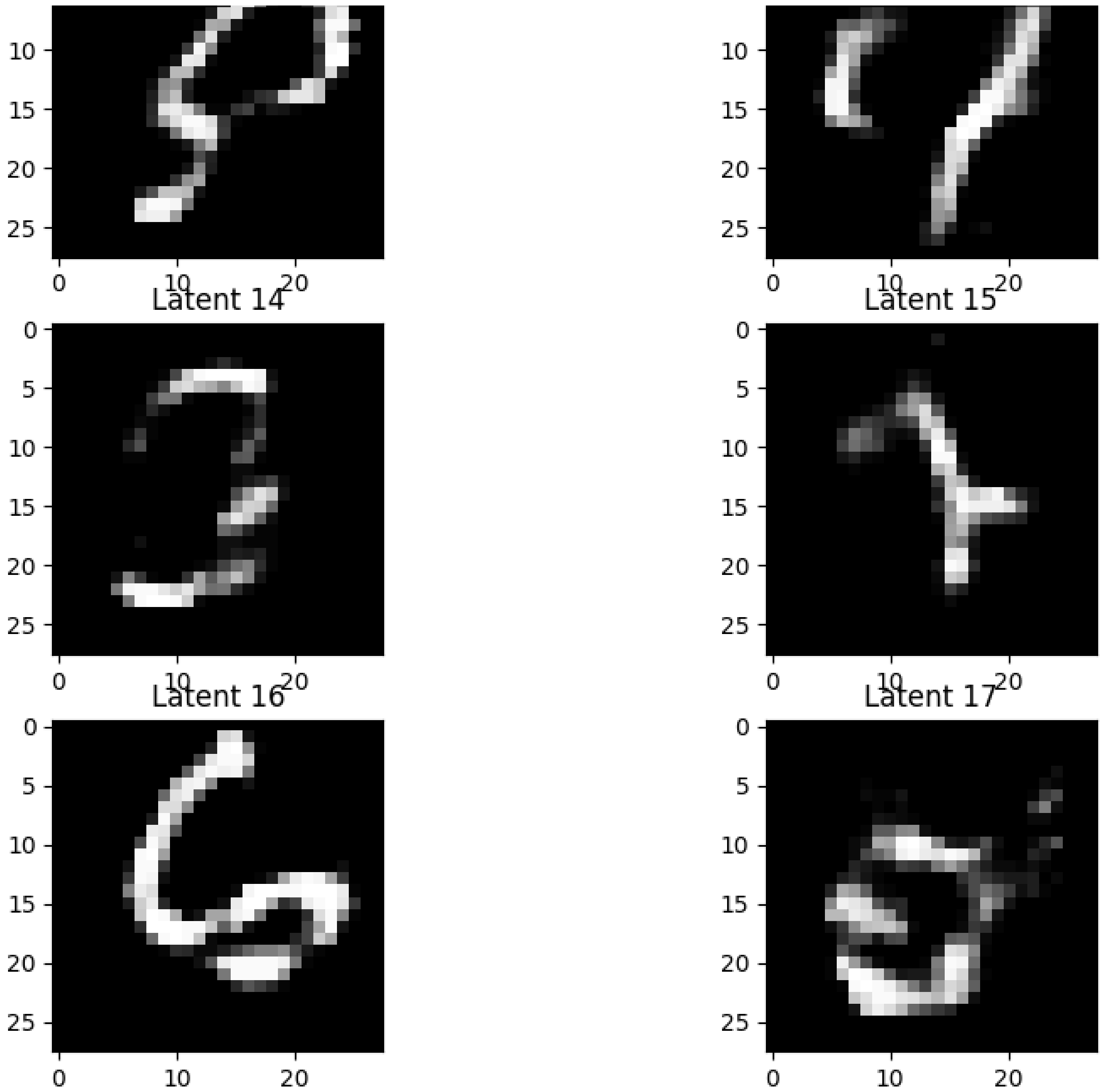
```
<matplotlib.image.AxesImage at 0x1d170fa3fb0>
```



In [113…

```python
model = AutoencoderModel(sizes=[784, 256, 64, 18, 64, 256, 784]).to(device_name)
model.load_state_dict(torch.load('model-weights/AutoencoderModel-l18.pth', map_location=device_name), strict=True)
plot_each_latent_individually(model, 18)
```


Latent 0
Latent 1
Latent 2
Latent 3
Latent 4
Latent 5
Latent 6
Latent 7
Latent 8
Latent 9
Latent 10
Latent 11
Latent 12
Latent 13



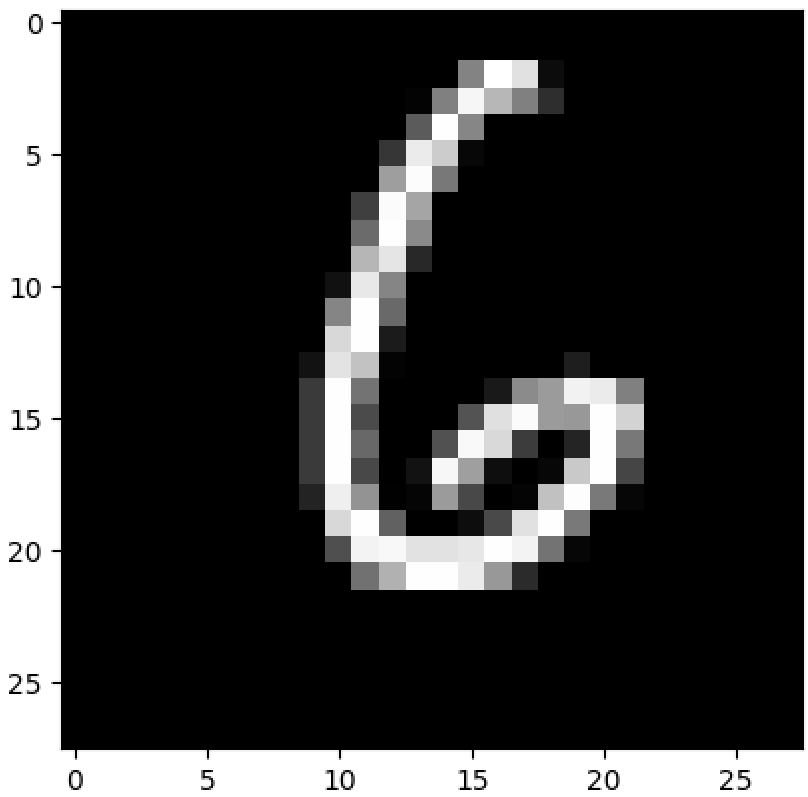
In [115…

```
# Latent8 purely captures the image of a 6, lets see
i = 6
j = 2
print(counted_latents[i][j])
plt.imshow(counted_images[i][j][0].cpu(), cmap='gray')
```

```
tensor([0.0000, 0.5878, 0.0000, 3.0704, 1.1339, 0.0000, 2.5035, 0.7379, 2.8546,
        0.6392, 2.2877, 0.0000, 3.4240, 0.8586, 0.9325, 1.5337, 3.8483, 0.0000],
       device='cuda:0')
```

Out[115…

```
<matplotlib.image.AxesImage at 0x1d151ba6270>
```



In [ ]: