Generating images with variational autoencoders

- Sampling from a latent space of images to create entirely new images or edit existing ones
  - The techniques are not specific to images.

Sampling from latent spaces of images

- The key idea
  - Develop a low-dimensional latent space of representations where any point can be mapped to a realistic-looking image.
- Generator or decoder
  - The module takes as input a latent point and outputs an image.

  No description has been provided for this image
- Two popular approaches: variational autoencoders (VAEs) and generative adversarial networks (GANs)

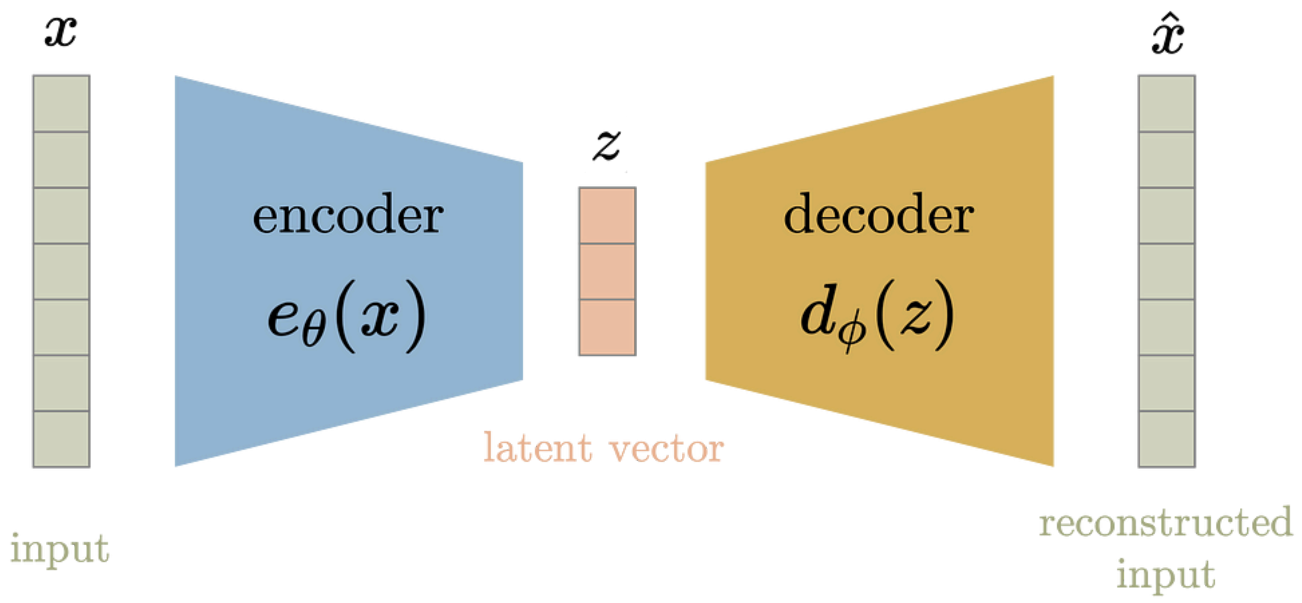
  No description has been provided for this image
- The concept vector
  - The idea of a concept vector is very similar to that of word embeddings.
  - Given a latent space of representations, or an embedding space, certain directions in the space may encode interesting axes of variation in the original data.

  No description has been provided for this image

Autoencoder

An **autoencoder** is a neural network that learns to reconstruct its input.

- Basic idea:
  - Input (x) → **Encoder** → latent vector (z) → **Decoder** → reconstructed (\hat{x})
  - Training objective: make (\hat{x}) as close as possible to (x) by minimizing a reconstruction loss.
- Architecture:
  - **Encoder**: progressively compresses the input into a low-dimensional latent vector (z).
  - **Decoder**: reconstructs the input from (z).
  - The bottleneck in the middle forces the model to keep only the most important features.
- Loss function:
  - Common choices are **Mean Squared Error (MSE)** or **Binary Cross-Entropy (BCE)** between input and output.
  - Here we use MNIST images in the ([0, 1]) range and apply BCE loss.



$$loss = \|x - \hat{x}\|_2 = \|x - d_\phi(z)\|_2 = \|x - d_\phi(e_\theta(x))\|_2$$

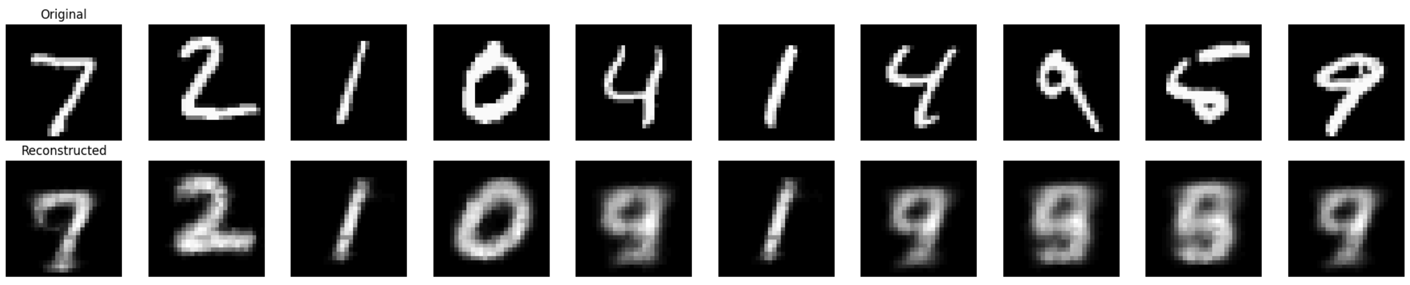
In [1]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Device selection
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# 1. MNIST data loaders
transform = transforms.ToTensor()  # Convert PIL image to tensor in [0, 1]

train_dataset = datasets.MNIST(root="data", train=True, download=True, transform=transform)
test_dataset  = datasets.MNIST(root="data", train=False, download=True, transform=transform)

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


# 2. Convolutional Autoencoder definition
class ConvAutoencoder(nn.Module):
    def __init__(self, latent_dim=16):
        super().__init__()
        self.latent_dim = latent_dim

        # Encoder: 1 x 28 x 28 -> latent_dim
        self.enc_conv1 = nn.Conv2d(1, 16, 3, stride=2, padding=1)   # 1x28x28 -> 16x14x14
        self.enc_conv2 = nn.Conv2d(16, 32, 3, stride=2, padding=1)  # 16x14x14 -> 32x7x7
        self.enc_fc    = nn.Linear(32 * 7 * 7, latent_dim)

        # Decoder: latent_dim -> 1 x 28 x 28
        self.dec_fc      = nn.Linear(latent_dim, 32 * 7 * 7)
        self.dec_deconv1 = nn.ConvTranspose2d(
            32, 16, 3, stride=2, padding=1, output_padding=1
        )  # 32x7x7 -> 16x14x14
        self.dec_deconv2 = nn.ConvTranspose2d(
            16, 1, 3, stride=2, padding=1, output_padding=1
        )  # 16x14x14 -> 1x28x28
    def encode(self, x):
        x = F.relu(self.enc_conv1(x))
        x = F.relu(self.enc_conv2(x))
        x = torch.flatten(x, 1)
        z = self.enc_fc(x)
        return z

    def decode(self, z):
        x = F.relu(self.dec_fc(z))
        x = x.view(-1, 32, 7, 7)
        x = F.relu(self.dec_deconv1(x))
        x = torch.sigmoid(self.dec_deconv2(x))  # output in [0, 1]
        return x

    def forward(self, x):
        z = self.encode(x)
        x_recon = self.decode(z)
        return x_recon


latent_dim = 2
autoencoder = ConvAutoencoder(latent_dim=latent_dim).to(device)

optimizer = torch.optim.Adam(autoencoder.parameters(), lr=1e-3)
# Sum over pixels, average over batch manually
criterion = nn.BCELoss(reduction="sum")

print(autoencoder)
```

```
Using device: cuda
100%|██████████| 9.91M/9.91M [00:00<00:00, 20.0MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 484kB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 4.48MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 13.4MB/s]
ConvAutoencoder(
  (enc_conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (enc_conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (enc_fc): Linear(in_features=1568, out_features=2, bias=True)
  (dec_fc): Linear(in_features=2, out_features=1568, bias=True)
  (dec_deconv1): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), outpu
t_padding=(1, 1))
  (dec_deconv2): ConvTranspose2d(16, 1, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output
_padding=(1, 1))
)
```

In [2]:
```
epochs = 10

for epoch in range(1, epochs + 1):
    autoencoder.train()
    total_loss = 0.0

    for x, _ in train_loader:
        x = x.to(device)

        # Forward pass
        x_recon = autoencoder(x)
        # Normalize by batch size to report mean loss per example
        loss = criterion(x_recon, x) / x.size(0)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * x.size(0)

    avg_loss = total_loss / len(train_loader.dataset)
    print(f"Epoch {epoch}, train loss: {avg_loss:.4f}")
```

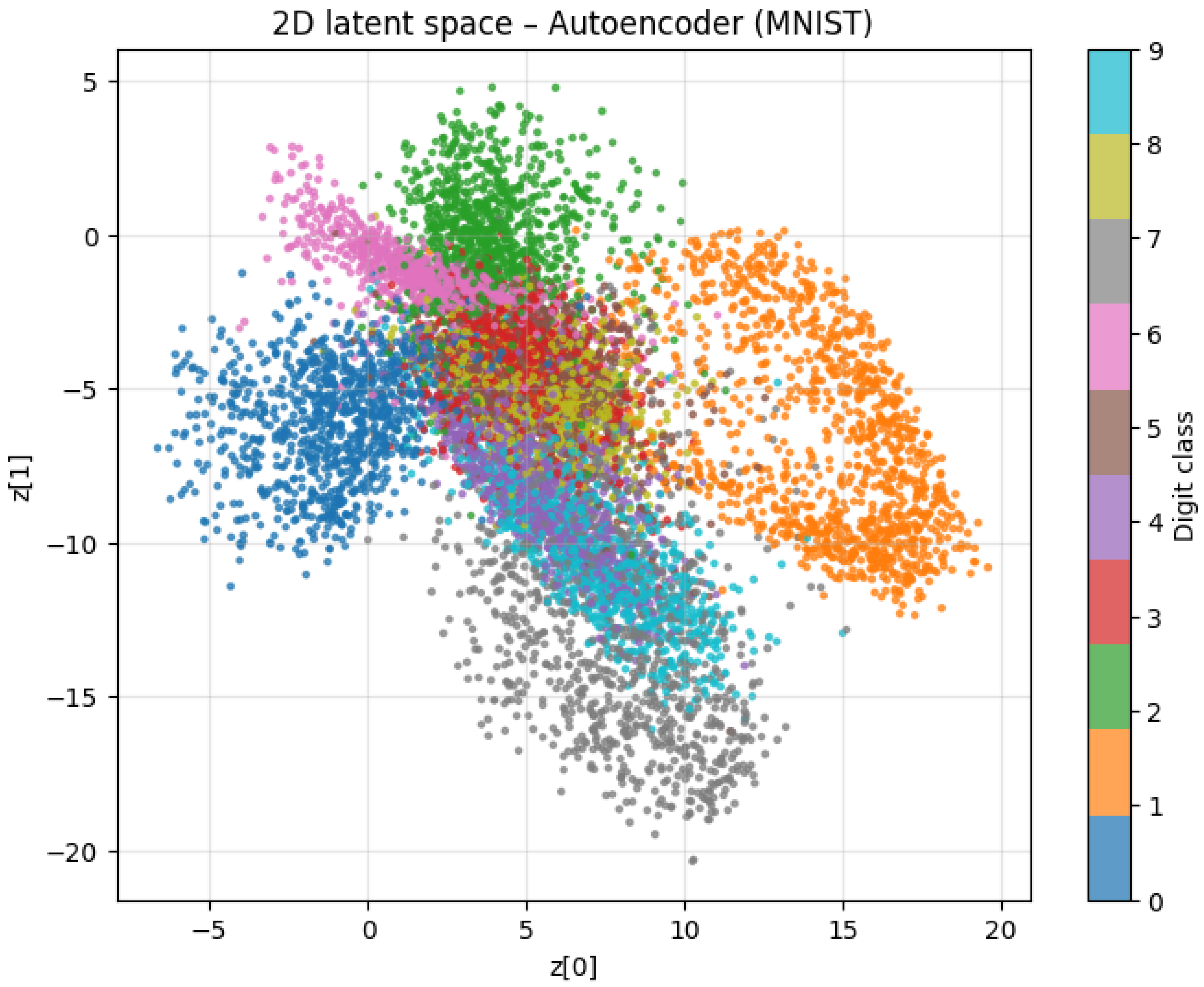
```
Epoch 1, train loss: 217.2682
Epoch 2, train loss: 179.9846
Epoch 3, train loss: 173.4792
Epoch 4, train loss: 170.2837
Epoch 5, train loss: 168.5033
Epoch 6, train loss: 167.1882
Epoch 7, train loss: 166.0776
Epoch 8, train loss: 164.9915
Epoch 9, train loss: 163.7266
Epoch 10, train loss: 162.5282
```

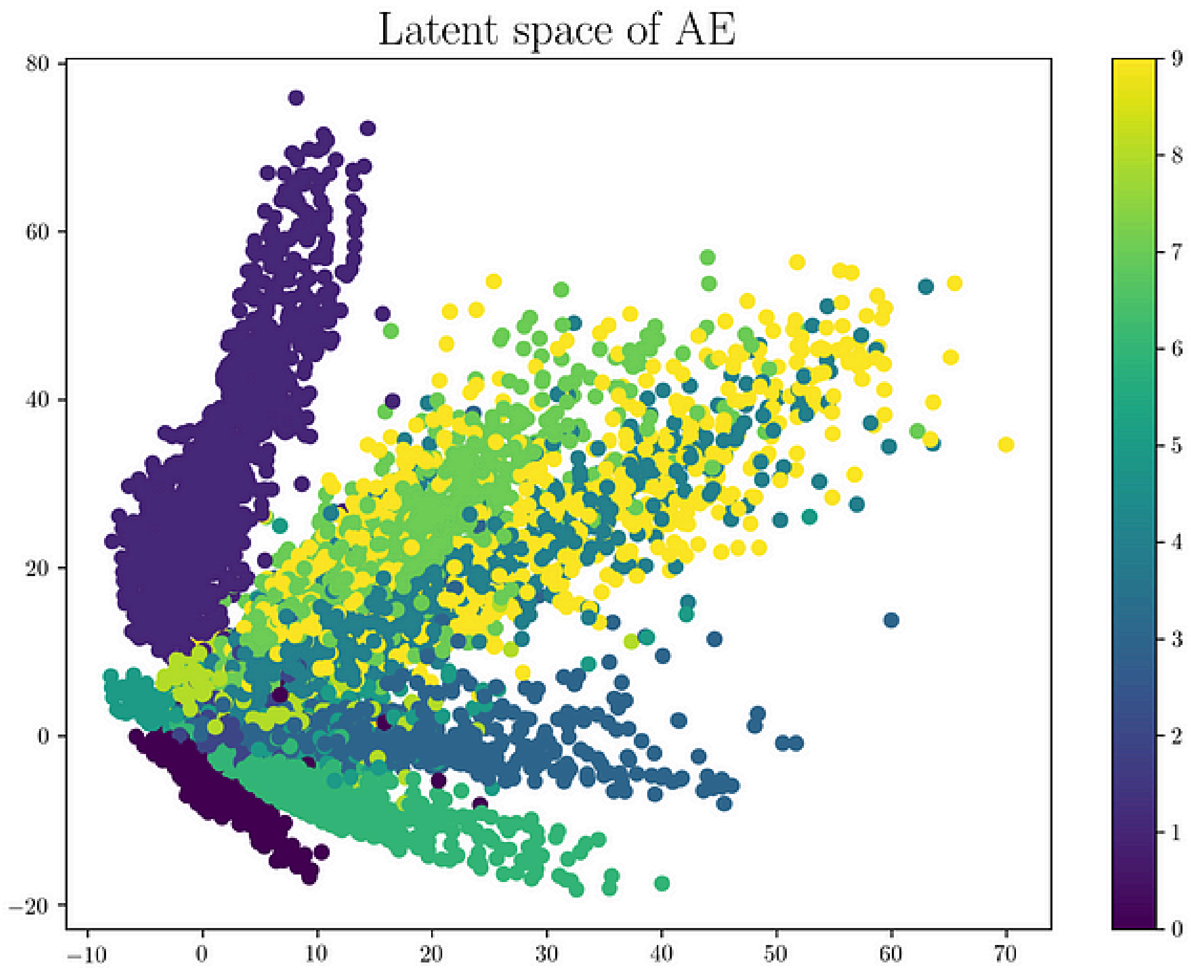
In [3]:
```
import matplotlib.pyplot as plt

autoencoder.eval()
with torch.no_grad():
    x, _ = next(iter(test_loader))
    x = x.to(device)
    x_recon = autoencoder(x)

# Move to CPU for plotting
x = x.cpu().numpy()
x_recon = x_recon.cpu().numpy()

n = 10  # number of examples to display
plt.figure(figsize=(2 * n, 4))

for i in range(n):
    # Original image
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x[i, 0], cmap="gray")
    plt.axis("off")
    if i == 0:
        ax.set_title("Original")

    # Reconstructed image
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(x_recon[i, 0], cmap="gray")
    plt.axis("off")
    if i == 0:
        ax.set_title("Reconstructed")

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



- The latent space generated by training the network on an MNIST dataset
  - The same digits tend to cluster themselves.
  - There are parts of the latent space that doesn't correspond to any data point.
  - AE is mainly used for compression.

In [4]:
```
import matplotlib.pyplot as plt
import numpy as np

autoencoder.eval()

all_z = []
all_labels = []

with torch.no_grad():
    for x, y in test_loader:  # test_loader from the AE section (MNIST test set)
        x = x.to(device)
        z = autoencoder.encode(x)      # shape: [batch_size, 2]
        all_z.append(z.cpu())
        all_labels.append(y)

all_z = torch.cat(all_z, dim=0).numpy()        # shape: [N, 2]
all_labels = torch.cat(all_labels, dim=0).numpy()  # shape: [N]

plt.figure(figsize=(8, 6))
scatter = plt.scatter(all_z[:, 0], all_z[:, 1],
                      c=all_labels, cmap="tab10", s=5, alpha=0.7)
plt.colorbar(scatter, ticks=range(10), label="Digit class")
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.title("2D latent space – Autoencoder (MNIST)")
plt.grid(True, alpha=0.3)
plt.show()
```


2D latent space – Autoencoder (MNIST)
z[0]
z[1]
Digit class
0
1
2
3
4
5
6
7
8
9
−5
0
5
10
15
20
−20
−15
−10



Variational autoencoders

- Autoencoders with Bayesian inference
- A classical autoencoder takes an input, maps it to a latent vector space via an encoder module, and then decodes it back to an output with the same dimensions as the input via a decoder module.

  No description has been provided for this image
- A VAE turns the image into the parameters of a statistical distribution, a mean and a variance, instead of compressing its input image into a fixed code in the latent space.
  - The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input.

  No description has been provided for this image
- How a VAE works:
  - An encoder module turns the input samples `input_img` into two parameters in a latent space of representations, `z_mean` and `z_log_variance`.
  - You randomly sample a point `z` from the latent normal distribution that's assumed to generate the input image, via `z = z_mean + exp(z_log_variance) * epsilon` where `epsilon` is a random tensor of small values.
  - A decoder module maps this point in the latent space back to the original input image.
- Because `epsilon` is random, the process forces the latent space to be continuously meaningful.

- The VAE is trained via two loss functions:
    - a reconstruction loss that forces the decoded samples to match the initial inputs
    - a regularization loss that helps learn well-formed latent spaces and reduce overfitting to the training data
- How it can be implemented:

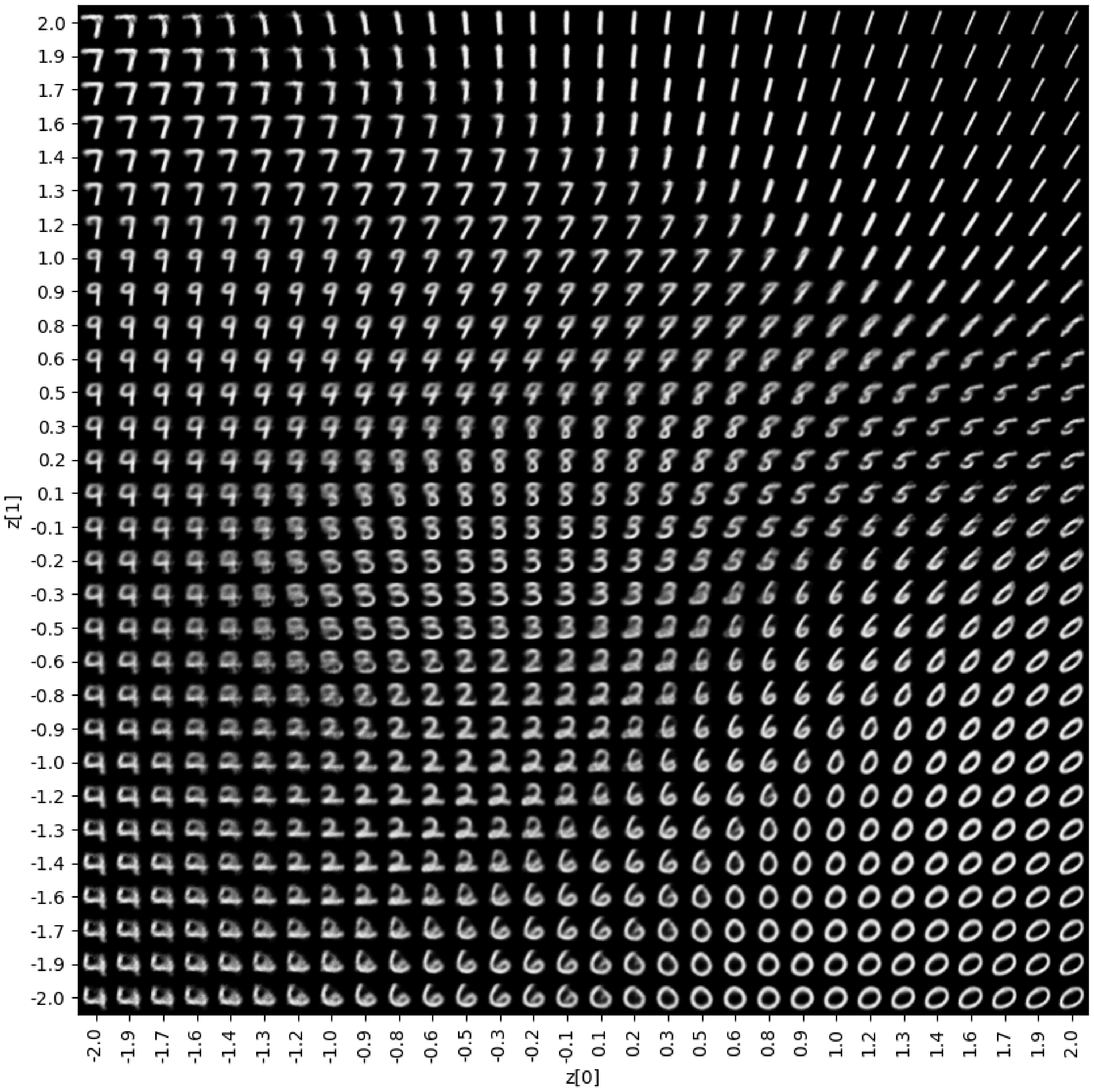
```
z_mean, z_log_variance = encoder(input_img)

z = z_mean + exp(z_log_variance) * epsilon

reconstructed_img = decoder(z)

model = Model(input_img, reconstructed_img)
```

In [5]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

latent_dim = 2


class Encoder(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, stride=2, padding=1)   # 28x28 -> 14x14
        self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)  # 14x14 -> 7x7
        self.fc = nn.Linear(64 * 7 * 7, 16)
        self.fc_mean = nn.Linear(16, latent_dim)
        self.fc_logvar = nn.Linear(16, latent_dim)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc(x))
        z_mean = self.fc_mean(x)
        z_logvar = self.fc_logvar(x)
        return z_mean, z_logvar

class Decoder(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.fc = nn.Linear(latent_dim, 7 * 7 * 64)
        self.deconv1 = nn.ConvTranspose2d(64, 64, 3, stride=2, padding=1, output_padding=1)
        self.deconv2 = nn.ConvTranspose2d(64, 32, 3, stride=2, padding=1, output_padding=1)
        self.out = nn.Conv2d(32, 1, 3, padding=1)

    def forward(self, z):
        x = F.relu(self.fc(z))
        x = x.view(-1, 64, 7, 7)
        x = F.relu(self.deconv1(x))
        x = F.relu(self.deconv2(x))
        x = torch.sigmoid(self.out(x))
```

```python
        return x

class VAE(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        recon = self.decoder(z)
        return recon, mu, logvar

# data loader
transform = transforms.ToTensor()

mnist_train = datasets.MNIST(root="data", train=True, download=True, transform=transform)
mnist_test = datasets.MNIST(root="data", train=False, download=True, transform=transform)
mnist_digits = torch.utils.data.ConcatDataset([mnist_train, mnist_test])
loader = DataLoader(mnist_digits, batch_size=128, shuffle=True)

encoder = Encoder(latent_dim).to(device)
decoder = Decoder(latent_dim).to(device)
vae = VAE(encoder, decoder).to(device)

optimizer = torch.optim.Adam(vae.parameters(), lr=1e-3)

def vae_loss(recon_x, x, mu, logvar):
    # BCE loss
    bce = F.binary_cross_entropy(recon_x, x, reduction='sum')
    # KL divergence
    kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return (bce + kld) / x.size(0)

epochs = 30
for epoch in range(1, epochs + 1):
    vae.train()
    total_loss = 0.0
    for x, _ in loader:
        x = x.to(device)
        optimizer.zero_grad()
        recon, mu, logvar = vae(x)
        loss = vae_loss(recon, x, mu, logvar)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * x.size(0)
    avg_loss = total_loss / len(loader.dataset)
    print(f"Epoch {epoch}, loss {avg_loss:.4f}")
```

```
Using device: cuda
Epoch 1, loss 215.6391
Epoch 2, loss 172.1060
Epoch 3, loss 161.9193
Epoch 4, loss 159.4883
Epoch 5, loss 158.1478
Epoch 6, loss 157.0778
Epoch 7, loss 156.2728
Epoch 8, loss 155.5611
Epoch 9, loss 154.9937
Epoch 10, loss 154.4963
Epoch 11, loss 154.0396
Epoch 12, loss 153.6064
Epoch 13, loss 153.2716
Epoch 14, loss 152.9040
Epoch 15, loss 152.5868
Epoch 16, loss 152.3223
Epoch 17, loss 152.0600
Epoch 18, loss 151.8108
Epoch 19, loss 151.5244
Epoch 20, loss 151.3773
Epoch 21, loss 151.1708
Epoch 22, loss 150.9319
Epoch 23, loss 150.7525
Epoch 24, loss 150.5615
Epoch 25, loss 150.4479
Epoch 26, loss 150.2254
Epoch 27, loss 150.0844
Epoch 28, loss 149.9998
Epoch 29, loss 149.8222
Epoch 30, loss 149.6163
```

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_latent_space(decoder, n=30, scale=2.0, digit_size=28):
    decoder.eval()
    figure = np.zeros((digit_size * n, digit_size * n))
    grid_x = np.linspace(-scale, scale, n)
    grid_y = np.linspace(-scale, scale, n)[::-1]

    with torch.no_grad():
        for i, yi in enumerate(grid_y):
            for j, xi in enumerate(grid_x):
                z_sample = torch.tensor([[xi, yi]], dtype=torch.float32, device=device)
                x_decoded = decoder(z_sample)
                digit = x_decoded[0, 0].cpu().numpy()
                figure[
                    i * digit_size : (i + 1) * digit_size,
                    j * digit_size : (j + 1) * digit_size,
                ] = digit

    plt.figure(figsize=(10, 10))
    start_range = digit_size // 2
    end_range = n * digit_size + start_range
    pixel_range = np.arange(start_range, end_range, digit_size)
    sample_range_x = np.round(grid_x, 1)
    sample_range_y = np.round(grid_y, 1)
    plt.xticks(pixel_range, sample_range_x, rotation=90)
    plt.yticks(pixel_range, sample_range_y)
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.imshow(figure, cmap="Greys_r")
    plt.show()

plot_latent_space(decoder)
```



In [ ]: