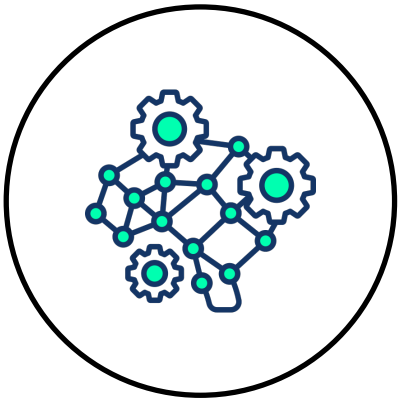
Deep Generative Modeling

University of Manitoba

Assignment 1 (ECE-7650)



Author:
Md. Masud Mazumder
Term: Winter 26
Student ID: 8056333

Instructors:
Ian Jeffrey
Associate Professor
Dept. of Electrical and Computer Engineering
University of Manitoba, Canada

Vahab Khoshdel
Assistant Professor
Dept. of Electrical and Computer Engineering
University of Manitoba, Canada

Code Available: GitHub Repository

Date: February 13, 2026

Declaration

I, Md. Masud Mazumder, attest that the work I am submitting is my own work and that it has not been copied/plagiarized from online or other sources. Any sourced material used for completing this work has been properly cited.

Signature

Md. Masud Mazumder

Contents

List of Figures

A Problem 1: Autoencoder Modifications

A.1 Task 1

Statement: Select a different architecture (e.g., more layers, fully connected layers, anything else you think might be of interest).

Solution Approach: The objective of this task was to modify the baseline autoencoder architecture while keeping all other components unchanged. Therefore, to increase the representational capacity of the model, I chose to add one additional 2D convolution layer with 128 filters and strides = 1 in the encoder, as illustrated in Figure 1.

To preserve architectural symmetry between the encoder and decoder, I also added an additional transposed convolution layer with 128 filters in the decoder, as shown in Figure 2. Adjusting the strides between convolution layers, the reconstructed image size was kept as the original image. All other components, including the latent dimension (2), loss function (binary cross-entropy), optimizer (Adam), and batch size, were kept identical to the baseline implementation to ensure a fair comparison.

```python
latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2D(128, 3, activation="relu", strides=1, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z = layers.Dense(latent_dim, name="z")(x)
encoder = keras.Model(encoder_inputs, z, name="encoder")
encoder.summary()
```

Figure 1: Modified encoder architecture.

```python
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 128, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 128))(x)
x = layers.Conv2DTranspose(128, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=1, padding="same")(x)
decoder_outputs = layers.Conv2DTranspose(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
decoder.summary()
```

Figure 2: Modified decoder architecture.

Rationale: The primary motivation for increasing the network depth was the hypothesis

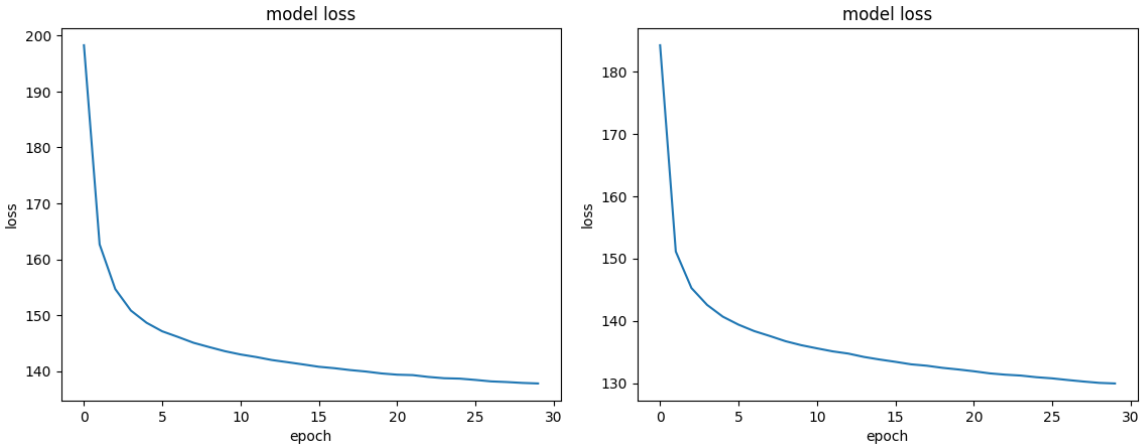
that deeper convolutional neural networks can extract higher-level abstractions. Although MNIST digits are relatively simple, they contain structured patterns such as strokes, curves, and spatial relationships. A deeper architecture may capture these hierarchical features more effectively.

Additionally, increasing the number of filters (particularly introducing 128 channels) enhances the capacity of the model, potentially allowing it to encode more variations in digit structure. On the other hand, maintaining structural symmetry between encoder and decoder often improves reconstruction stability and performance.

Effect on Training: During training, the baseline architecture required approximately 22 minutes, whereas the modified architecture required approximately 70 minutes. This significant increase in training time is expected due to the additional convolutional and transposed convolutional layers with a relatively large number of filters (128), which increase both parameter count and computational complexity.

Although no separate validation set was used to measure generalized performance with the baseline, the training loss curve provides insight into reconstruction quality. While both the baseline and modified architectures exhibited smooth convergence (Figure 3), the baseline model's reconstruction loss started at approximately 198.3 in the first epoch and decreased to 137.8 by the 30th epoch. In contrast, the modified architecture started with a lower initial loss of approximately 184.2 and converged to a lower final loss of 130.0. This improvement was also visible in the display of a grid of sampled digits from the latent space in Figure 4.

Overall, this improvement in final reconstruction loss indicates that the deeper architecture achieved better representational efficiency despite maintaining the same 2D bottleneck.



(a) Baseline model loss curve.

(b) Modified model loss curve.

Figure 3: Loss curve for baseline and modified architectures.

A.2 Task 2

Statement: Use a different latent space. Specifically, we want you to try a latent space size of your choice, but we also want you to try a latent space with dimension 1.

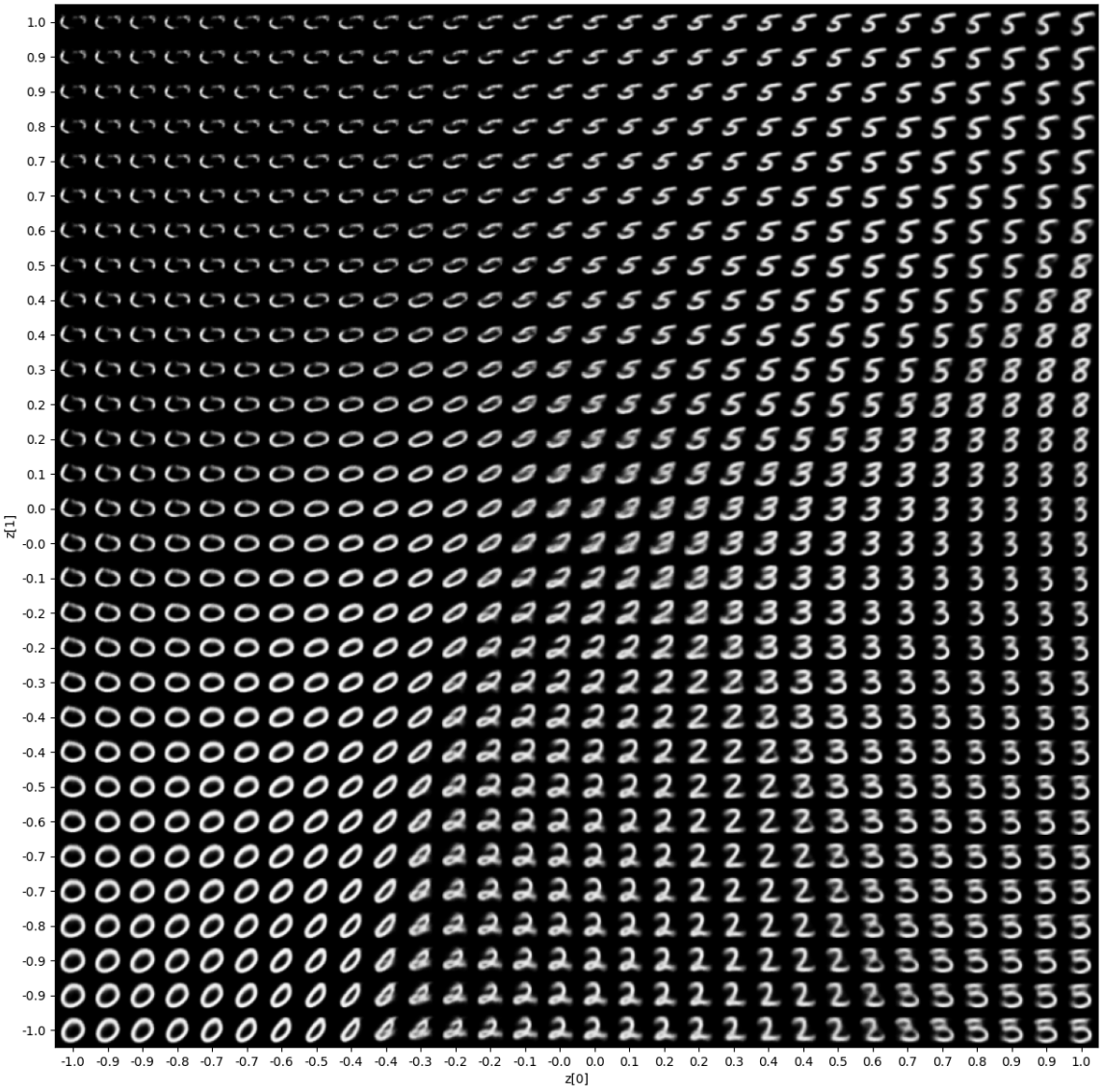


Figure 4: Digit manifold examples for modified architecture.

Solution Approach: In this task, I modified the latent dimension of the baseline autoencoder. The original implementation used a 2-dimensional latent space, while I increased the dimension to 3. The rest of the architecture (convolution layers, loss function, optimizer, batch size, and number of epochs) remained unchanged.

Since the latent space is now 3-dimensional, the original 2D latent visualization was no longer sufficient. Therefore, I visualized the latent structure by:

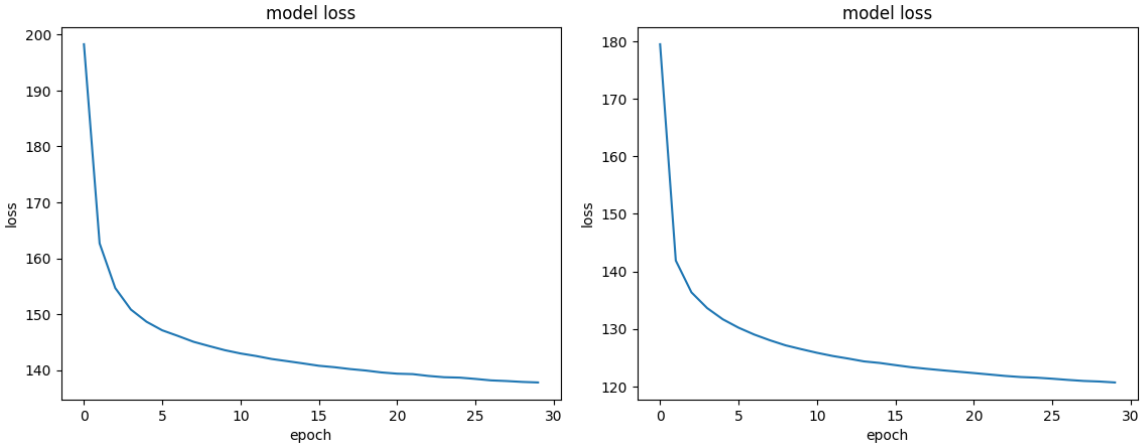
- Plotting 2D projections of the 3D latent space from different views.
- Fixing one latent dimension and varying the other two to observe generated digit manifolds.

Rationale: The motivation behind increasing the latent dimension was to study how additional representational capacity in the bottleneck affects reconstruction and latent structure.

While I could have selected a much higher dimension, choosing 3 provided a balance between improved capacity and interpretability.

Effect on Training: The training configuration remained identical to the baseline model. The total training time was approximately 22 minutes, which is similar to the baseline model.

From the loss curves shown in Figure 5, the 3D latent model achieved slightly better reconstruction performance compared to the 2D baseline. The additional latent dimension reduced compression pressure, enabling the model to encode more information before reconstruction. As a result, the final training loss was lower.



(a) 2D (baseline) model loss curve.

(b) 3D model loss curve.

Figure 5: 2D and 3D model's loss curves.

Latent Space Behavior: When visualizing[1] 2D projections of the 3D latent space (Figure 6), digit clusters appeared more separated compared to the baseline model. However, some overlap remained when observed from certain viewpoints. This is expected because the clusters may be well separated in 3D space but appear overlapping when projected onto 2D planes.

When fixing one latent dimension and varying the other two (Figure 7), some digit manifolds showed smooth and gradual transitions, indicating structured organization in the latent space. However, in other cases, transitions appeared unusual or less smooth. This behavior is reasonable because we are observing only 2D slices of a 3D latent space, and the structure may not be uniformly smooth across all directions.

Overall, this experiment highlights that increasing the latent dimension improves reconstruction quality and produces a more structured latent space without significantly increasing computational cost. But we need to remember that the higher latent dimension results in reduced compression.

[1]The code for visualization was generated with assistance from ChatGPT [1]. The generated code was reviewed, modified, and integrated by me.

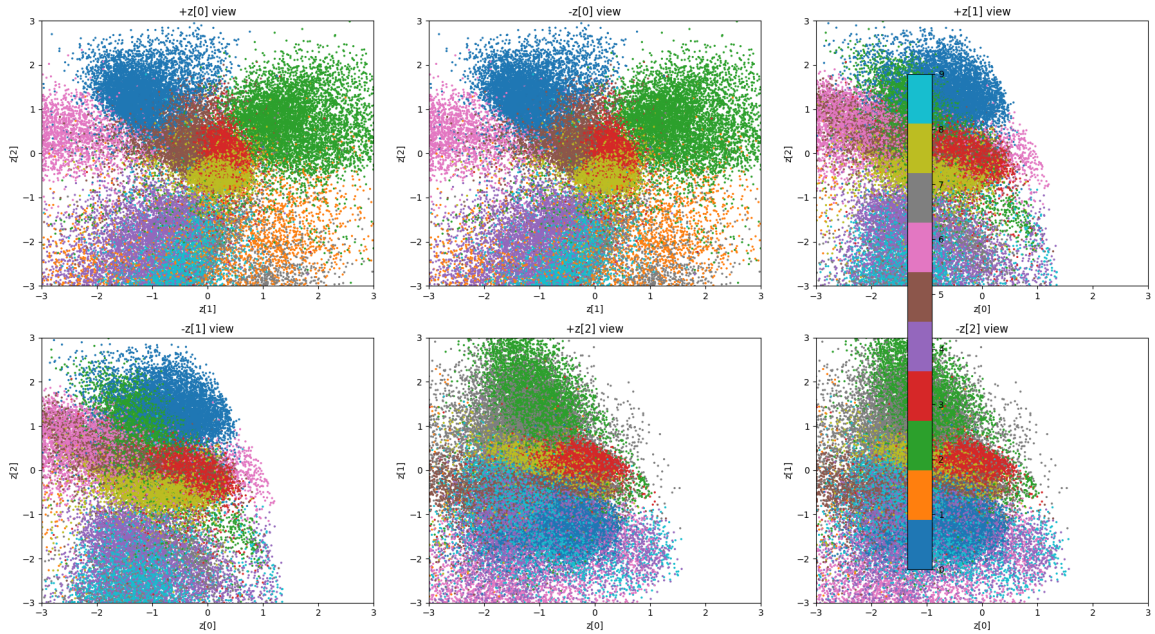
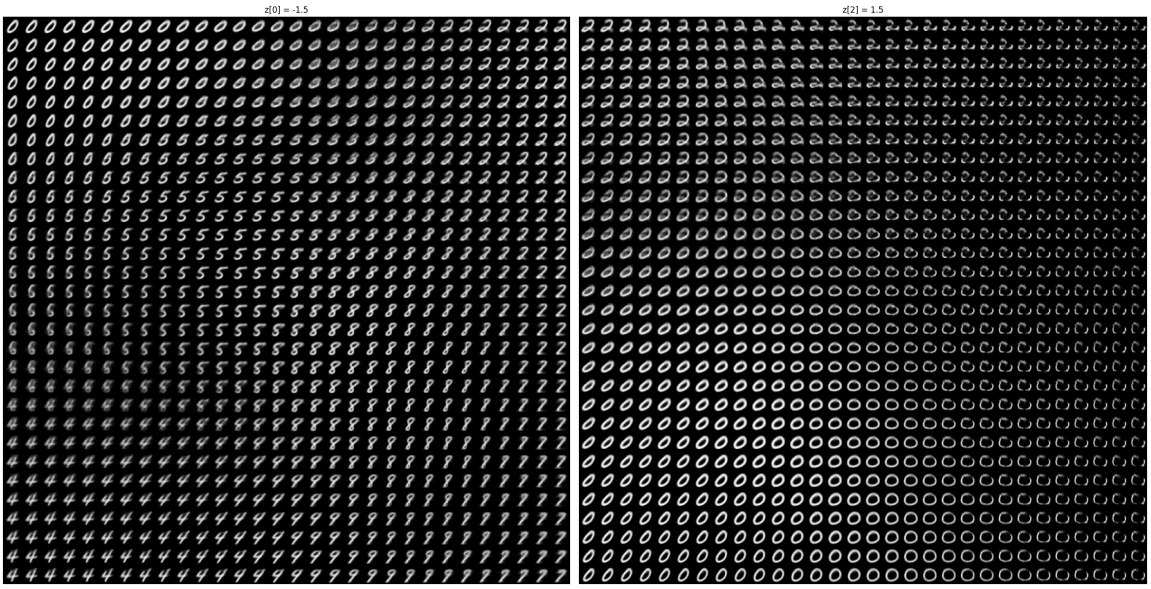


Figure 6: Visualizing 2D projections of the 3D latent space.



(a) Smoother transition.

(b) Unusual transition.

Figure 7: Digit manifold examples

A.3 Task 2: Latent Dimension 1

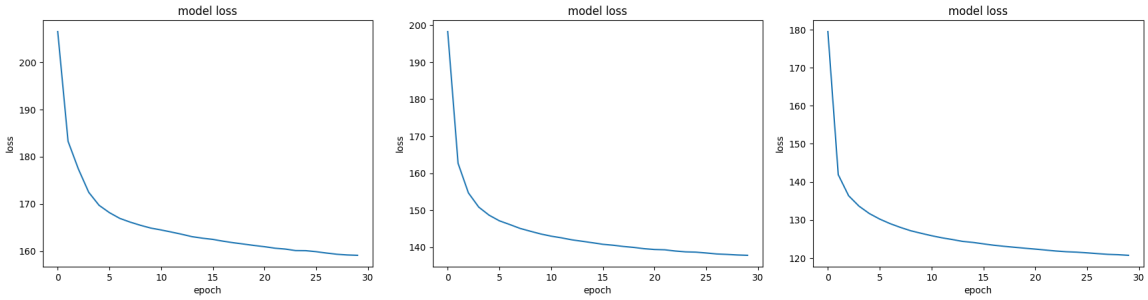
Statement: Use a different loss function.

Solution Approach: In this experiment, I modified the latent dimension of the baseline autoencoder from 2 to 1. The encoder output layer was changed to produce a single latent value, and the decoder input was adjusted accordingly. All other components of the model,

including architecture, loss function, optimizer, batch size, and number of epochs, were kept unchanged to ensure a fair comparison.

Rationale: Reducing the latent dimension to 1 forces extreme compression. The model must encode all structural information of the image into a single scalar value. This somehow maps the most important features into just one single number.

Effect on Training: Training remained stable and took almost an equal amount of time as the baseline model to complete 30 epochs, but the reconstruction loss was higher compared to the 2D and 3D latent models, as shown in Figure 8. This is expected because a 1D latent space imposes a very strong compression constraint. The encoder cannot preserve sufficient structural details, leading to information loss during reconstruction.



(a) 1D latent model loss curve. (b) 2D latent model loss curve. (c) 3D latent model loss curve.

Figure 8: 1D, 2D, and 3D latent model loss comparison.

Latent Space Behavior: When varying the single latent variable across a range (-1 to 1), the generated images (Figure 9) showed a gradual transition between digits and their styles. Some images appeared blurrier. Although the figure does not show any significant holes, there is no meaningful representation in practice when the latent values go beyond -2 and 100, which can be easily interpreted from the latent space cluster in Figure 10. An important observation here is that, although most of the digits overlap in the clustering, the cluster for 0 is the smallest. The reason, I believe, for some 0s is that it is a feature that is different from others.

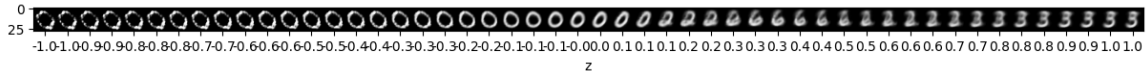


Figure 9: Digit manifold examples for 1D latent space.

Overall, reducing the latent dimension to 1 significantly increases compression but decreases reconstruction quality and representation capacity. While the model can still capture general structural patterns, it struggles to preserve fine details and class separability.

A.4 Task 3

Statement: Use a different loss function.

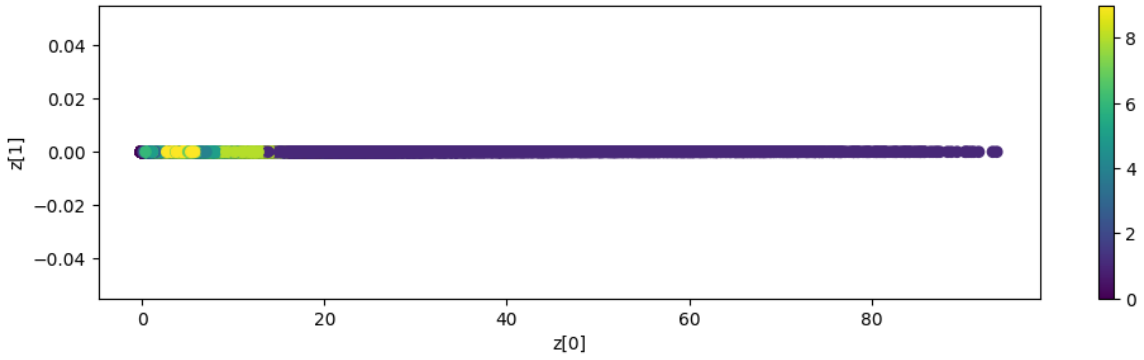


Figure 10: 1D latent space cluster.

Solution Approach: In this task, I modified the reconstruction loss function of the baseline autoencoder. The original implementation used Binary Cross-Entropy (BCE) as the reconstruction loss, and I replaced BCE with Mean Squared Error (MSE). The only change made in the implementation was inside the *train_step()* function, where the reconstruction loss was computed using:

```
keras.losses.mean_squared_error(data, reconstruction)
```

Rationale: Binary Cross-Entropy is commonly used when the input pixels are normalized between 0 and 1 and are interpreted as probabilities. However, MNIST images are grayscale images rather than strictly binary images. Mean Squared Error measures the squared difference between the original and reconstructed pixel values. It treats reconstruction as a regression problem rather than a probabilistic classification problem. The lecture notes also mentioned the MSE loss function.

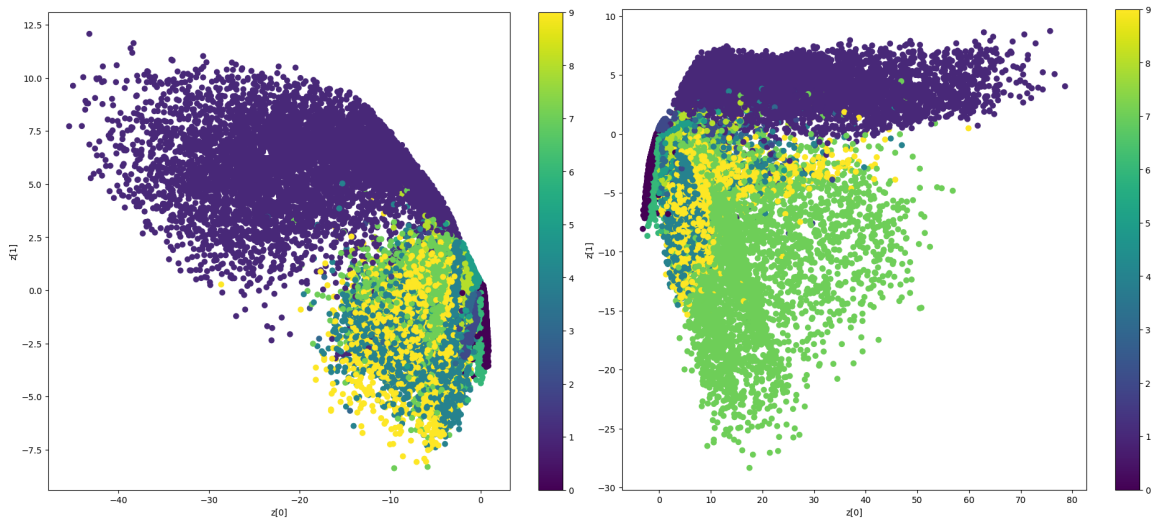
Effect on Training: The training configuration remained identical to the baseline model, and the model showed stable convergence throughout training. We cannot compare the loss value with the baseline model, as BCE and MSE loss measures are different. However, the comparison of the latent space cluster separation in Figure 11 highlights significant improvement over the baseline model using the BCE loss function. This improvement is expected because the MSE loss tends to penalize large pixel deviations heavily.

A.5 Task 4

Statement: Use a different data set, specifically something with more than 1 channel.

Solution Approach: I searched for a colored dataset and, fortunately, found the colorized MNIST dataset. The dataset was originally formatted as (C, H, W), where C is the number of channels. The baseline model expects input in the form (H, W, C). Therefore, a simple transpose operation was required to adjust the shape. Due to memory limitations, I randomly selected 70% of the dataset for training.

In addition, the baseline model was originally configured to accept 1-channel grayscale images. Therefore, I modified the input layer and the final decoder layer to handle 3-channel (RGB) images. The code snippet for loading and preparing the dataset is shown in Figure 12. Figure 13 shows 20 random samples from the dataset.



(a) Baseline model's clustering. (b) Modified model's clustering.

Figure 11: Cluster separation comparison.

```python
# load the dataset and extract the images
ds = load_dataset("pietrolesci/cmnist", "bias_conflicting")
train_splits = [k for k in ds.keys() if k.startswith("train")]
dataset_images = np.concatenate(
    [np.array(ds[split]["pixel_values"]) for split in train_splits],
    axis=0
)
dataset_images = np.transpose(dataset_images, (0, 2, 3, 1))

#select randomly 70% of the data only for training
num_samples = dataset_images.shape[0]
num_train_samples = math.ceil(num_samples * 0.7)
random_indices = np.random.choice(num_samples, num_train_samples, replace=False)
dataset_images = dataset_images[random_indices]
```

Figure 12: Load and prepare dataset code snippet.

Figure 13: Samples from the colorized MNIST dataset.

Rationale: The primary goal was to evaluate how the autoencoder behaves when the input dimensionality increases from 1 channel to 3 channels.

Although the choice of dataset was practical rather than theoretical, using colorized MNIST allows us to isolate the effect of additional channels while keeping the underlying digit structure similar. This makes it easier to interpret the impact of multi-channel inputs on reconstruction and latent space organization.

Effect on Training: Since the input channels increased from 1 to 3, the model processes three times more pixel information. Therefore, a higher computational cost would normally be expected. However, training took approximately 20 minutes, which is slightly less than the baseline training time. This is because the total number of images was lower than the baseline dataset.

The loss curve decreased gradually with minor fluctuations across epochs (Figure 14). This indicates stable convergence. The slight fluctuations may be due to the increased complexity of reconstructing RGB images.

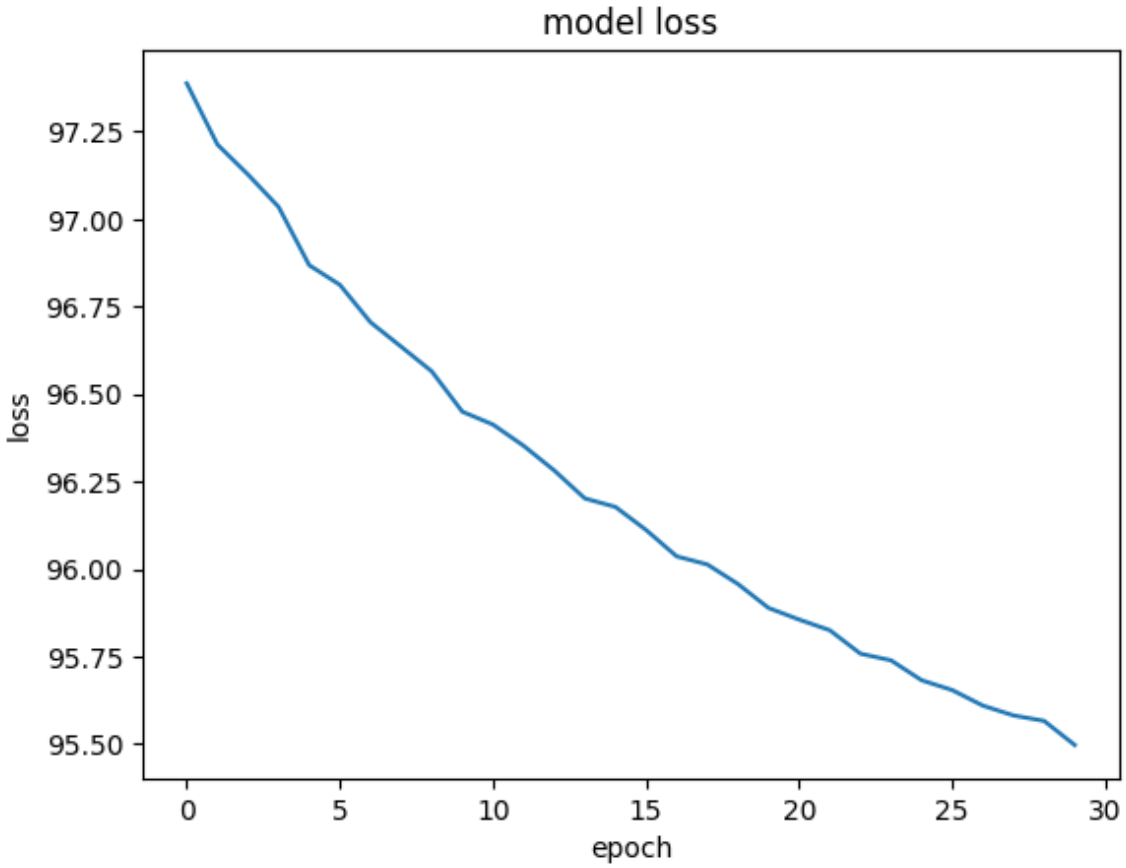


Figure 14: Loss function with colorized MNIST dataset.

An interesting observation is that the latent space clusters are more clearly separated compared to the grayscale baseline (Figure 15). Almost all digit classes formed distinct clusters. This improved separation is strongly influenced by the color information. In the colorized MNIST dataset, many digit classes are associated with specific dominant colors. Therefore, the network may rely heavily on color cues rather than structural digit features. While this improves cluster separation and reduces loss, it introduces a bias that may not generalize well in practical scenarios.

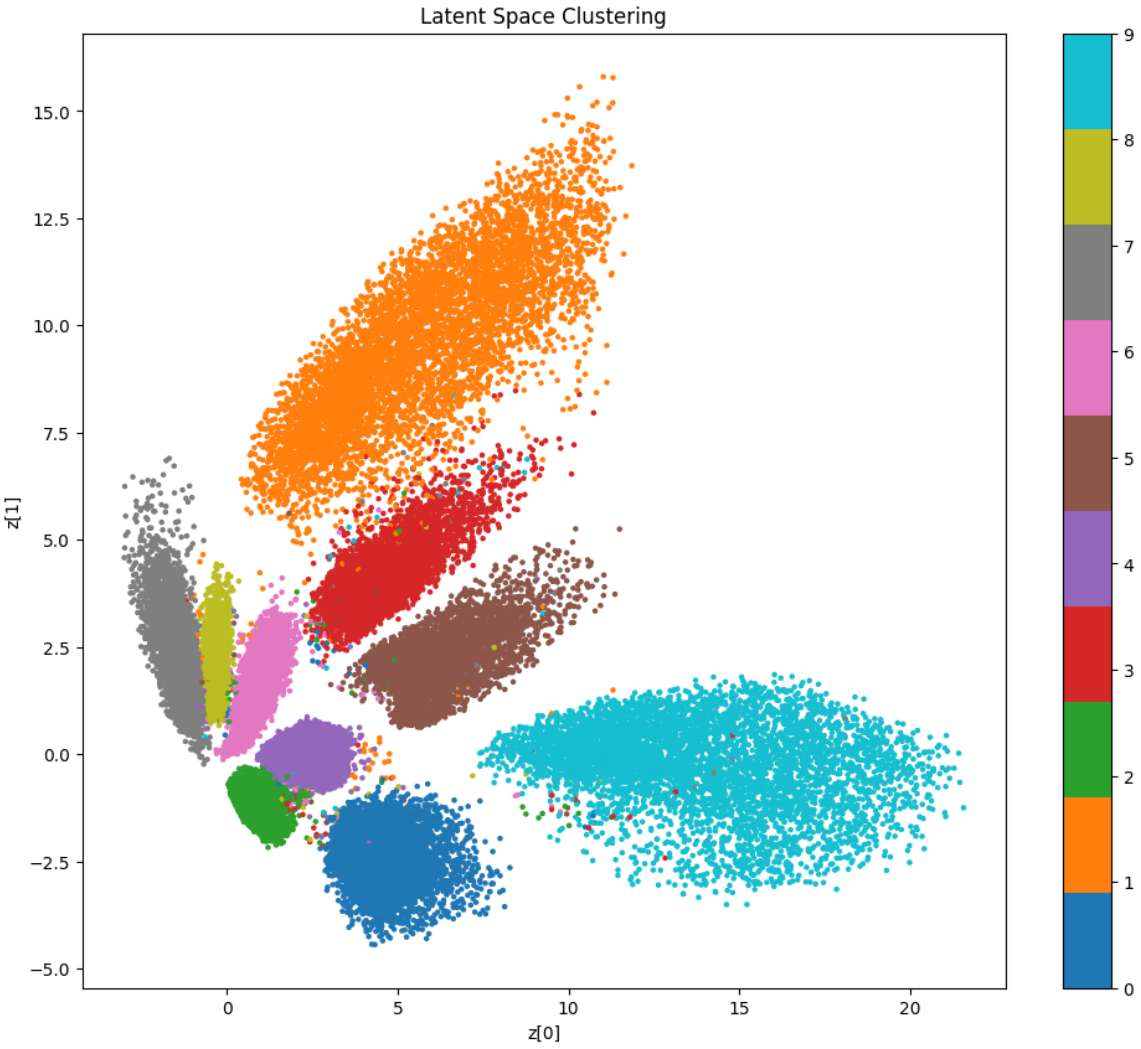


Figure 15: Latent space clusters for the colorized MNIST dataset.

B Problem 2: Autoencoder Application

For this problem, I have chosen the modified model in Problem 1 (Task 1) as the base model. All the experiments were conducted on this model only.

B.1 Task 1

Inpainting: From the test set, select samples and remove chunks of various sizes from the input image. Use the AE and attempt to reconstruct the original image.

Input Generation: For this experiment, I fixed the size of a square-shaped mask. I selected 10 random images from the test dataset to evaluate inpainting. From a random starting position in each image, I set the pixel values inside a square region to 0, thereby applying the mask (Figure 16). The masked images were then passed through the encoder and decoder to reconstruct the complete image. Random selection and reconstruction results are shown in Figure 17.

```python
def mask_image(img, mask_size=12):
    img_masked = img.copy()
    H, W, _ = img.shape

    # Random top-left corner
    x = np.random.randint(0, H - mask_size)
    y = np.random.randint(0, W - mask_size)

    # Apply mask as black square to the image
    img_masked[x:x+mask_size, y:y+mask_size, :] = 0.0

    return img_masked
```

Figure 16: Apply a square-shaped mask randomly.

Result: The 10 inpainting examples are shown in Figure 18, where each example includes the original, masked, and reconstructed images.

For mask size 10, most reconstructions were reasonably accurate, and the missing area was filled in realistically. However, the reconstructed images appear slightly blurrier than the original images. This blurriness occurs because the autoencoder compresses the image into a low-dimensional latent space. So, the encoder cannot perfectly recover the exact pixel-level details. When part of the image is missing, reconstruction becomes more difficult. Instead, it reconstructs the most likely pattern based on the learned data distribution.

```python
num_samples = 10
indices = np.random.choice(len(mnist_test_set), num_samples, replace=False)

originals = mnist_test_set[indices]
masked = np.array([mask_image(img, mask_size=10) for img in originals])

reconstructed = ae.decoder.predict(ae.encoder.predict(masked))
```

Figure 17: Random image selection and image reconstruction from masked images.

When the mask size was increased from 10 to 15 (Figure 19), the reconstruction quality degraded significantly. As a larger portion of the image is removed, more structural information is lost. Consequently, the latent representation formed from the masked image becomes less informative, leading to poorer reconstruction. This behavior is expected because the autoencoder is not explicitly trained for inpainting; it is trained for full-image reconstruction. Therefore, when too much information is missing, the model struggles to infer the correct structure.

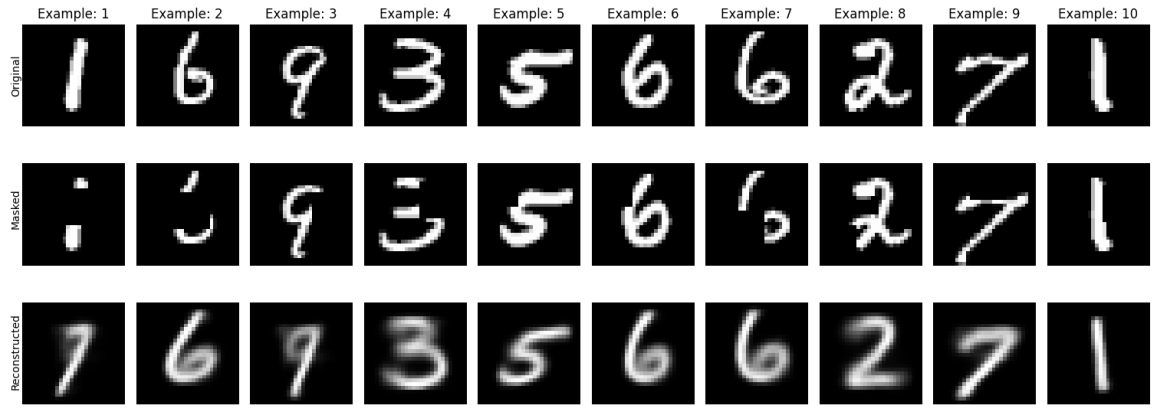


Figure 18: Original, masked, and reconstructed examples with mask size 10.

B.2 Task 2

Denoising: From the test set, select samples and add noise to the data. Use the AE and attempt to reconstruct the original image.

Input Generation: For this experiment, I randomly selected 10 images from the test dataset and added Gaussian noise to them (Figures 20 and 21). Gaussian noise was added by sampling from a normal distribution and adding it to the image pixels. After adding noise, pixel values were clipped to remain within the valid range [0, 1]. Then the noisy images were passed through the encoder and decoder to obtain reconstructed outputs.

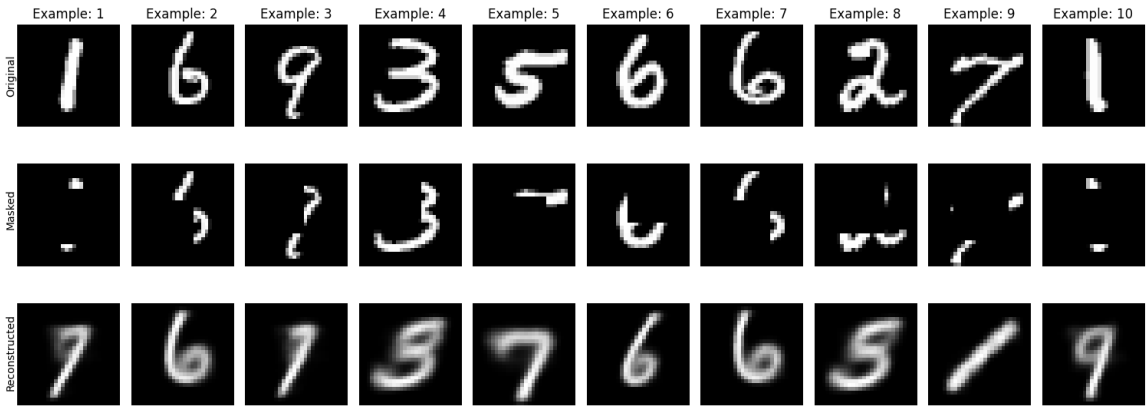


Figure 19: Original, masked, and reconstructed examples with mask size 15.

```python
def add_gaussian_noise(img, sigma=0.1):
    noise = np.random.normal(0, sigma, img.shape)
    noisy_img = img + noise
    noisy_img = np.clip(noisy_img, 0., 1.)
    return noisy_img
```

Figure 20: Gaussian noising function.

```python
num_samples = 10
indices = np.random.choice(len(mnist_test_set), num_samples, replace=False)

originals = mnist_test_set[indices]
noisy = np.array([add_gaussian_noise(img, sigma=0.1) for img in originals])
reconstructed = ae.decoder.predict(ae.encoder.predict(noisy))

plot(originals, noisy, reconstructed, num_samples)
```

Figure 21: Apply denoising with AE.

Result: The reconstructed images were significantly cleaner than the noisy inputs, as shown in Figures 22 and 23. The model successfully removed most of the Gaussian noise

irrespective of the noise quantity added. However, the reconstructed images were slightly smoother compared to the original images. Some fine details were softened, which I believe is the effect of compression into a lower-dimensional latent space. However, from the figures it is clear that although the AEs can remove noise from data, it is hard to reconstruct the original data from the noisy one. I believe this is because the MNIST dataset contains only two colors, and the encoder focuses more on the pixel values than the digit's shape or other features. Denoising in Figure 22 (sigma = 0.1) and 23 (sigma = 0.4) removes noise similarly, but the first one more correctly reconstructs the original digits than the other one, which validates my assumption.

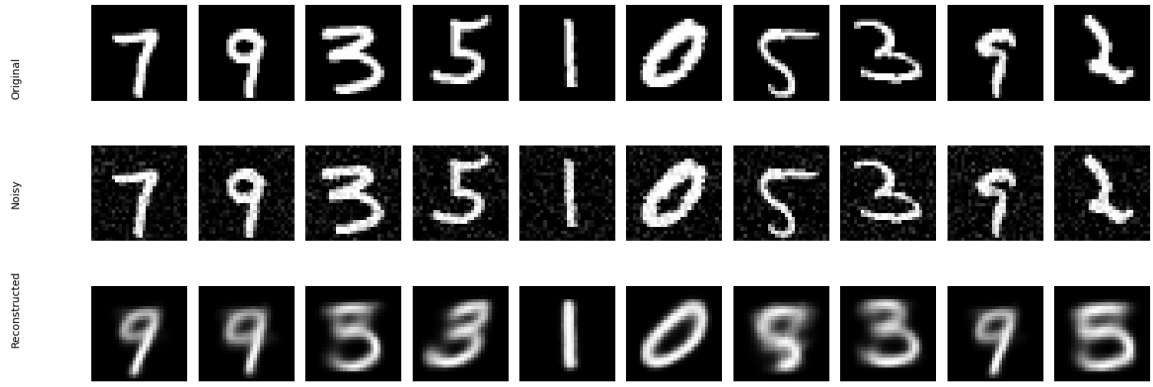


Figure 22: Noising with Sigma = 0.1 and denoising.

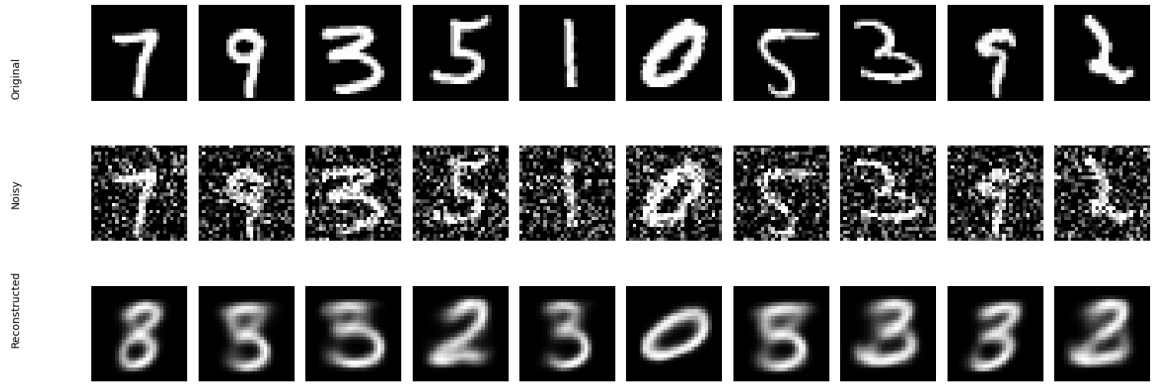


Figure 23: Noising with Sigma = 0.4 and denoising.

B.3 Task 3

Anomaly Detection: Choose an image from a different dataset and apply it to the AE. Use the AE and attempt to reconstruct the original image.

Input Generation: For anomaly detection, I selected images from the Fashion-MNIST dataset, which is structurally different from the digit dataset used to train the autoencoder. Since the trained autoencoder expects data formatted as (Batch, H, W, C) and the dataset was in (Batch, H, W) format, I expanded to 1 channel. All images were normalized to the range [0, 1]. The anomaly images were then passed through the encoder and decoder to

obtain reconstructed outputs. Figure 24 shows these steps.

```python
num_samples = 10

(x_fashion, _), _ = keras.datasets.fashion_mnist.load_data()

# Convert to (H, W, C) format as the AE expects and normalize the pixel values to [0, 1]
x_fashion = np.expand_dims(x_fashion, -1)
x_fashion = x_fashion.astype("float32") / 255.0

# Randomly select 10 images from the Fashion dataset
indices = np.random.choice(len(x_fashion), num_samples, replace=False)
anomaly_images = x_fashion[indices]

# Use the AE to reconstruct the selected anomaly images
reconstructed = ae.decoder.predict(ae.encoder.predict(anomaly_images))
```

Figure 24: Anomaly detection dataset generation and image reconstruction.

Result: The reconstructed anomaly images differed significantly from the original anomaly inputs, as shown in Figure 25. While the original dataset/input is of fashion-related images like shoes, t-shirts, pants, etc., the AE reconstructed digits. This behavior is expected because the autoencoder was trained only on the digit dataset. It learns to compress and reconstruct images that follow that specific data distribution. When presented with out-of-distribution samples (e.g., clothing items instead of digits), the encoder fails to generate a meaningful latent representation, leading to poor reconstruction.

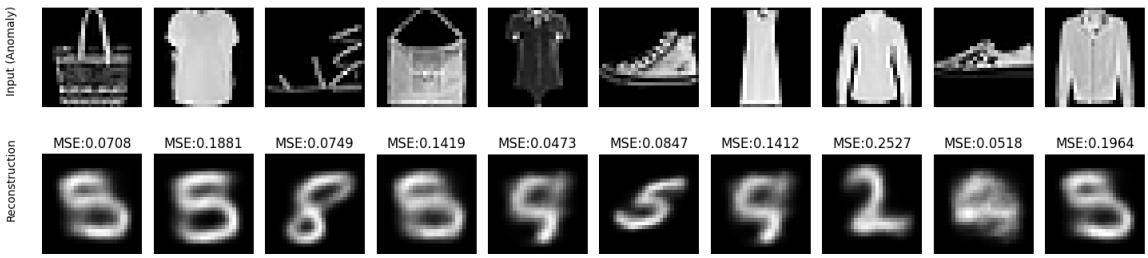


Figure 25: Image reconstruction by anomaly dataset.

I also computed the reconstruction errors as shown in Figure 26 and got significantly higher values per pixel (notice that pixel values are between 0 and 1, and these are grayscale images). Therefore, autoencoders could be used as basic anomaly detectors by measuring reconstruction errors. However, their performance depends strongly on how well the training data represents the target distribution.

B.4 Task 4

Generation: Choose samples from the latent space. Use the decoder to generate new images.

```python
mse = np.mean((anomaly_images - reconstructed) ** 2, axis=(1,2,3))
print("Reconstruction errors:", mse)

# Output:
# Reconstruction errors: [0.06047587 0.20659783 0.11588331 0.09471458
# 0.07613517 0.11381519 0.152023   0.19535057 0.08753812 0.06928855]
```

Figure 26: Reconstruction error calculation.

Input Generation: For this task, I generated new images directly from the latent space using two approaches:

1. **Grid Sampling:** Since the latent dimension is 2, I created a grid of evenly spaced points within a fixed range (-2 to 2) for both latent variables (Figure 27). Each pair of latent coordinates was passed to the decoder to generate an image.

2. **Sampling from the Learned Latent Distribution:** I first encoded the MNIST digit dataset to obtain its latent representations. Then, I computed the mean and standard deviation of these latent vectors. Using these statistics, I sampled new latent vectors from a Gaussian distribution and passed them through the decoder to generate images (Figure 28).

Result: In the grid sampling approach, the generated images (Figure 29) showed smooth transitions between different patterns as the latent variables changed gradually. However, some extreme regions of the latent space produced unrealistic or distorted images. This occurred because the deterministic AE does not regularize the latent space, and some regions contain holes, meaning some regions may not correspond to meaningful training data representations.

In the Gaussian sampling approach, the generated images (Figure 30) appeared more realistic compared to grid sampling. This is because the sampling was done from a learned latent distribution. Although some images are unrealistic, unlike grid sampling, there were no smooth transitions between images. This is expected because the latent vectors were sampled randomly from a distribution rather than arranged in a structured grid.

```python
def generate_from_latent(ae, n=10, scale=2.0):
    grid_x = np.linspace(-scale, scale, n)
    grid_y = np.linspace(-scale, scale, n)

    H, W, C = ae.decoder.output_shape[1:]
    figure = np.zeros((H*n, W*n, C))

    for i, yi in enumerate(grid_y):
        for j, xi in enumerate(grid_x):
            z_sample = np.array([[xi, yi]])
            decoded = ae.decoder.predict(z_sample, verbose=0)[0]

            figure[
                i*H:(i+1)*H,
                j*W:(j+1)*W,
                :
            ] = decoded

    plt.figure(figsize=(10, 10))
    plt.imshow(figure, cmap="gray")
    plt.axis("off")
    plt.title("Generated Samples from Latent Space")
    plt.show()

generate_from_latent(ae, n=15, scale=2)
```

Figure 27: Grid sampling from latent space.

```python
# Compute mean and std of the latent representations of the real data
z_real = encoder.predict(mnist_digits)
mean = np.mean(z_real, axis=0)
std = np.std(z_real, axis=0)

# Sample random points from the latent space and generate 50 images
z_samples = np.random.normal(loc=mean, scale=std, size=(50, z_real.shape[1]))
generated_images = decoder.predict(z_samples)

# Display the generated all 50 images
plt.figure(figsize=(10, 5))
for i in range(50):
    plt.subplot(5, 10, i + 1)
    plt.imshow(generated_images[i].squeeze(), cmap="gray")
    plt.axis("off")
plt.suptitle("Generated Images from Random Latent Samples")
plt.show()
```

Figure 28: Sampling from the learned latent distribution.

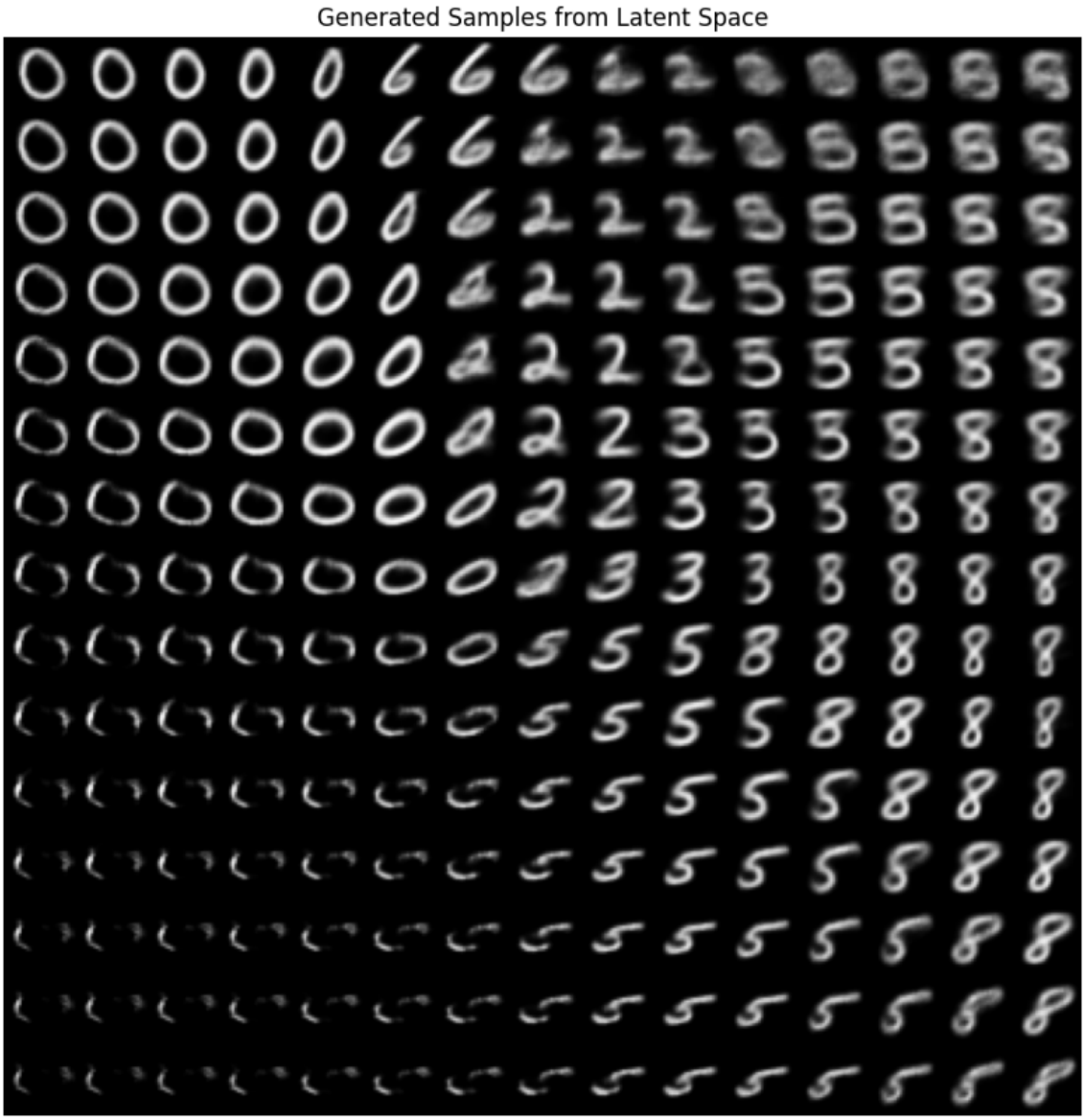


Figure 29: Reconstruction examples from grid sampling.

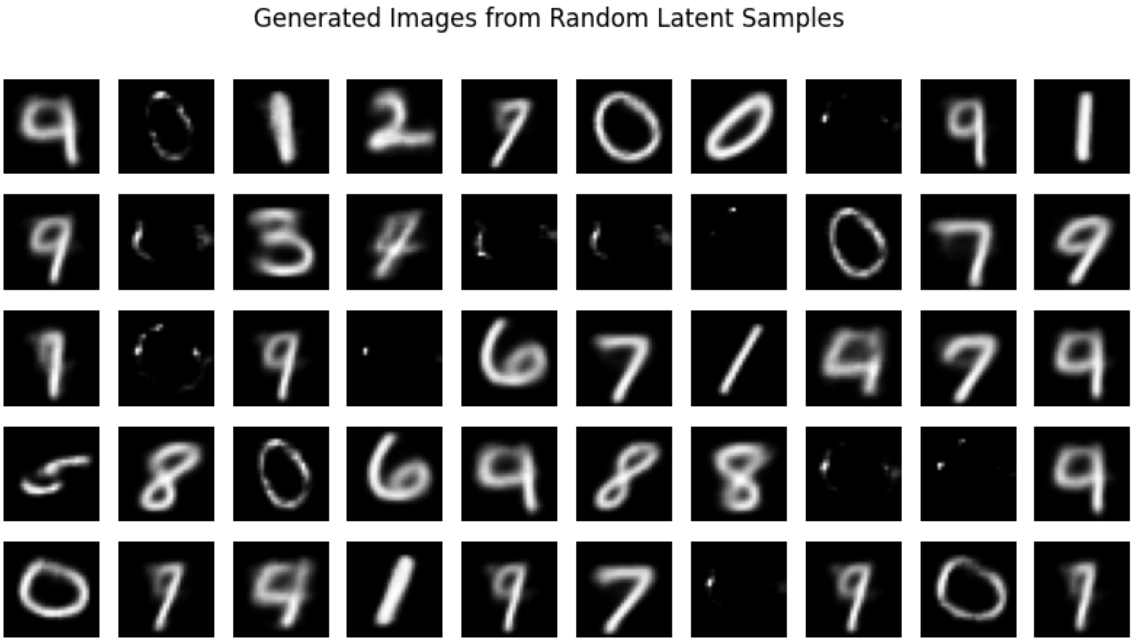


Figure 30: Reconstruction examples from learned latent distribution.

C Additional Discussion Questions

C.1 Question 1

For any given application, how might you go about determining an appropriate latent space size for a given AE model?

Answer: To the best of my knowledge, determining the appropriate latent space size for an autoencoder (AE) is a classic trade-off problem. Too small a latent space size can lose important information, leading to underfitting and significantly hampering the reconstruction of data. On the contrary, too higher a value will also hamper compression of data, and the model learns to pass input straight to the output without learning meaningful structure, leading to overfitting. Therefore, keeping in mind the rule of thumb that the latent dimension should be as low as possible, I would like to go for several approach listed below:

- Start with a small value and gradually increase and test if this size is good enough.
- I can compute reconstruction loss (e.g., MSE) with varying latent sizes on the same dataset. The goal is to find the point where increasing the latent size further hardly reduces the loss.
- Required latent dimension may depend on the complexity of the data. Therefore, I believe a simple grayscale may work well with 2–5 dimensions, while complex images may require many more dimensions.

C.2 Question 2

Your tests for Problem 2 focused on samples from the test set. Would performance change significantly if you took samples from the training set? Why or why not?

Answer: Yes, performance would likely improve slightly if we evaluated the autoencoder using samples from the training set instead of the test set.

An autoencoder is trained to minimize reconstruction error on the training data. Therefore, it has already learned representations that best reconstruct those specific samples. When the same training samples are passed through the model again, reconstruction error is usually lower because the model has effectively "seen" those examples before.

In contrast, test samples were not directly used during optimization. Although they come from the same data distribution, small variations in structure or style may result in slightly higher reconstruction error compared to training samples.

C.3 Question 3

Can residual blocks be included in an autoencoder network? Why or why not?

Answer: Yes, residual blocks can be included in an autoencoder network. They do not change the fundamental structure of the model, which consists of an encoder that compresses the input into a latent representation and a decoder that reconstructs it. Residual connections simply modify how features are learned within the encoder or decoder. They

improve gradient flow and reduce vanishing gradient problems, which is especially helpful for deeper architectures.

However, skip connections (such as encoder–decoder shortcuts) are not ideal when the primary goal is strict data compression. Since skip connections allow feature information to bypass the bottleneck and flow directly from encoder to decoder, they can reduce the importance of the latent representation. This weakens the compression constraint and may result in a less meaningful latent space.

References

[1] OpenAI. Chatgpt. https://chat.openai.com/, 2025. Accessed: 2026-02-13.