Generative AI Theory & Applications
Report 1: Variational Autoencoders 1

Syed Abraham Ahmed

February 7, 2025

Contents

1 Variational Autoencoders

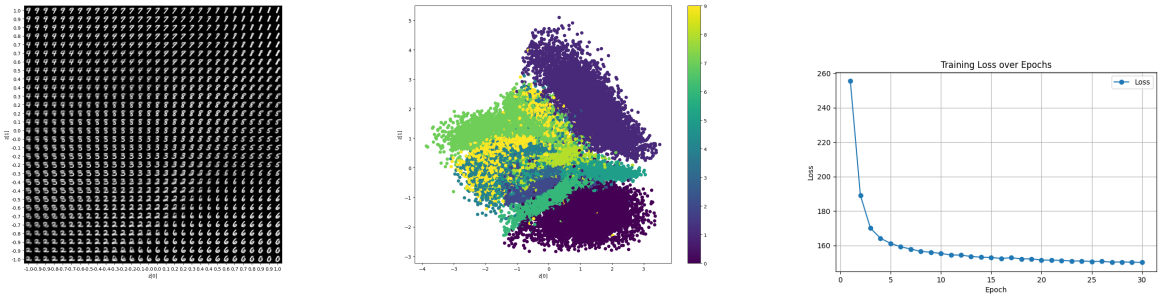
Our task is to modify a baseline variational auto-encoder setup in different ways and compare the outcomes to an original. Specifically, we will:

- Experiment with a different architecture (e.g., adding more layers or switching to fully connected layers).
- Use a different latent space (we will try at least one dimension other than 2, including a 1D latent space).
- Adopt a different loss function.

Through these modifications, we hope to observe how each change affects the variational autoencoders' ability to learn meaningful representations, reconstruct inputs accurately, and generalize.

We use the original variational auto-encoder from Keras example [1]. The encoder starts with a 28×28×1 input for the grayscale image. 2 convolutional layers are applied (32 and 64 filters) with ReLU activation functions to reduce spatial dimensions while extracting features. A flatten layer turns these features into a 1D vector, and 2 dense layers (16 units each) compute the latent mean and log variance, which a sampling layer uses to produce the latent vector z.

The decoder reverses this process. A dense layer expands z into a 7×7×64 volume, which is up-sampled by two transposed convolution layers with ReLU activation. A final transposed convolution layer with a sigmoid activation generates the 28×28 grayscale image.



(a) Latent Space Plot (b) Test Data in Latent Space Plot (c) Loss over Epoch Plot

Figure 1: Original Output

After 30 epochs, the trained VAE has learned a latent space where digits cluster according to their visual similarity, however some classes overlap when their shapes are easy to confuse (3s and 5s). This overlap comes up because the VAE is modeling a continuous distribution of digits rather than learning a strict classification boundary. We observe a smooth transition across the latent space, showing how similar points produce visually similar digits and confirming that the model treats digit generation as a distribution mapping rather than a one-to-one function. It is worth noting an official train time of **2m 41.4s** with a final loss of **151.5002**

1.1 Architecture

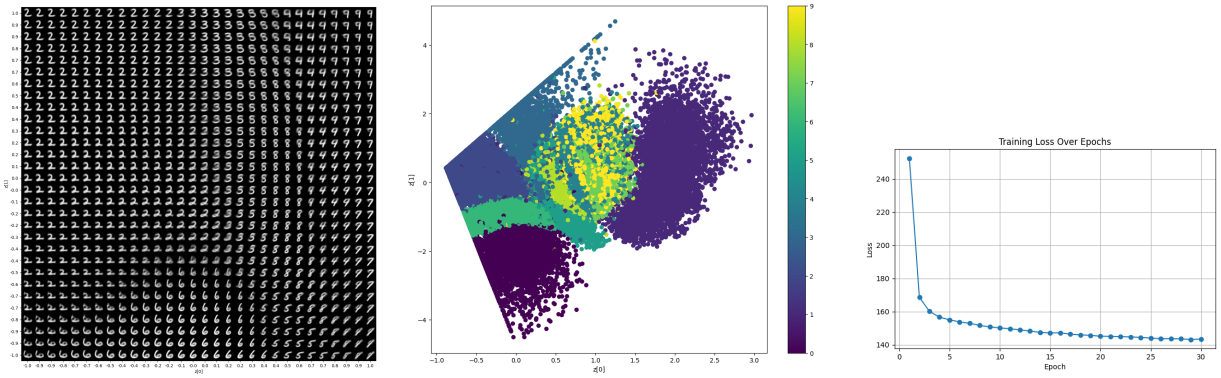
An addition of an extra convolutional layer in both the encoder and decoder was considered. A third convolutional layer in the encoder was added with 128 filters, 2 strides, and ReLU activation before flattening. A third transposed layer in the decoder was added to upsample the latent vector back to the original input dimensions. We adjust the kernel size to 4 and include padding of 1 to adjust for left over pixels to reach a specific output shape of 28x28x1 in the final layer.

```
#ENCODER
# --- Third Conv2D layer ---
x = layers.Conv2D(128, 3, activation="relu", strides=2, padding="same")(x)

#DECODER
# --- Third Conv2DTranspose layer ---
x = layers.Conv2DTranspose(128, 4, activation="relu", strides=2, padding="same", output_padding=1)(x)
```

Listing 1: Addition of Convolution Layer in Encoder and Decoder (transpose)

The reasoning for this change in the architecture is that the additional layer in the encoder and decoder hopefully allows the network to learn higher-level features, potentially improving reconstruction quality.



(a) Latent Space Plot (b) Test Data in Latent Space Plot (c) Loss over Epoch Plot

Figure 2: Architecture Output

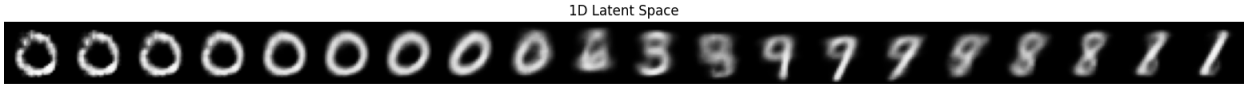
Training time increased to a value of **3m 48.5s** and a final loss value of **141.4165** due to the larger number of parameters to process. Reconstruction quality improved for more complex digits, which was reflected in a drop in the validation loss. An observation to note is that by appearance, it seems as if 2 dimensions creates a "congested" field of plots within our latent space plot. This may be further discussed ahead even when we consider a different architecture for "better" feature extraction (which is evident here in comparison to the original latent space plot)

1.2 Latent Space

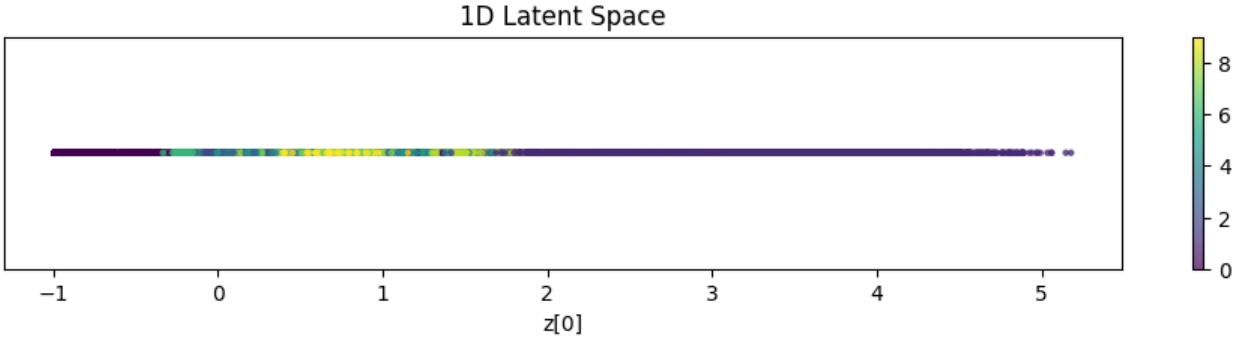
All latent space adjustments were done to latent_dim variable defined in the encoder code block. We also consider our original architecture instead of our additional layers in the encoder and decoder.

1.2.1 Dimension 1

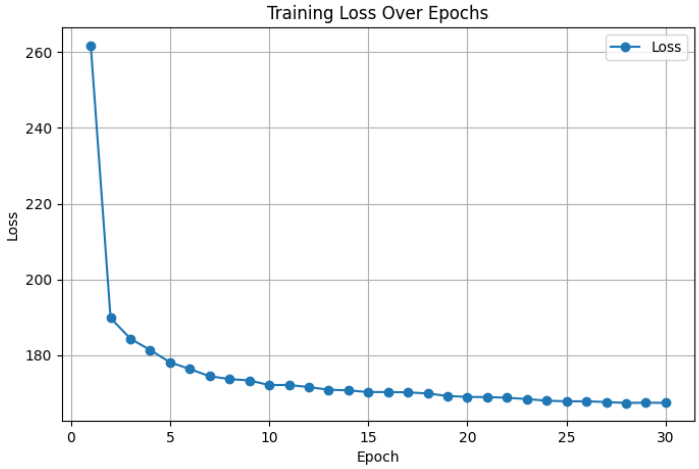
A 1D latent space shows that the VAE could still capture meaningful generative structure in a strictly linear manifold. However there are some considerations discussed further below.



(a) Latent Space Plot



(b) Test Data in Latent Space Plot



(c) Loss over Epoch Plot

Figure 3: 1D Output

The total time of training was **2m 43.5155s** and the final loss value was **163.2497**. The VAE is able to distinguish certain characteristics, however the overlap among digits increased, indicating the model was forced to "cram" digit features into a more confined space, which may suggest the reason why our final loss increased.

1.2.2 Dimension 10

Plots can't be directly visualized in a 10D space, but applying a projection of a t-SNE [2] and using the provided python code for our application [3], we can see a 10D space yielded a higher capacity to learn our distribution model. A python function was created and used in Appendix B to show how the VAE fitted the MNIST digits into the 10D latent space. 10 Dimensions was chosen because the goal is to learn each "digit distribution" among each axis.

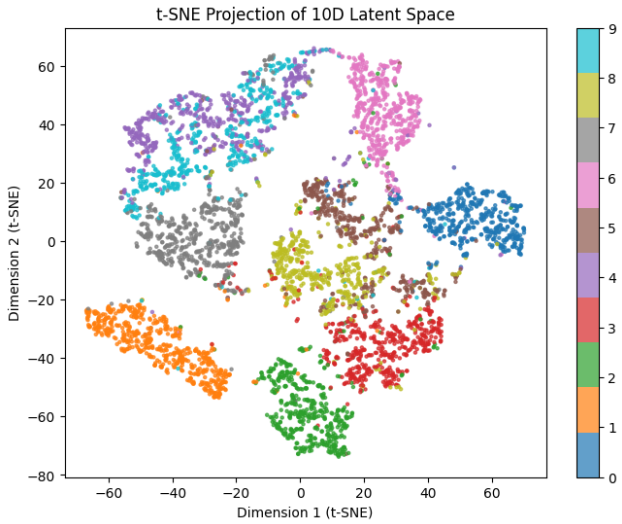


Figure 4: Latent Space Plot 10D Projection

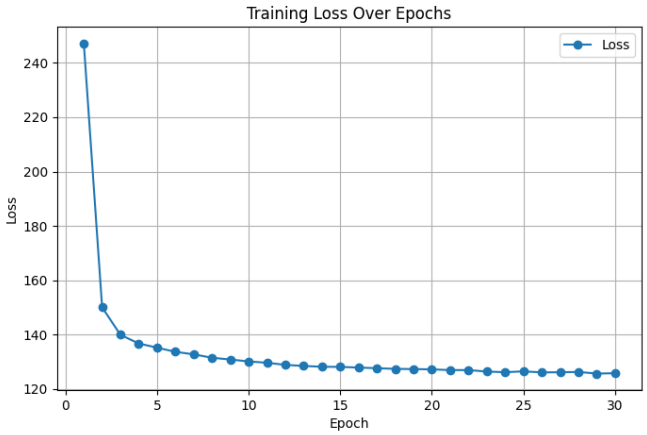


Figure 5: Loss over Epoch Plot

The total training time nearly remained the same being **2m 43.1245s** with a better loss value of **124.3071**. The model has more degrees of freedom to separate classes more clearly, which is evident in the t-SNE visualization in comparison to the restrictive nature of the previous 2D/1D manifold examples.

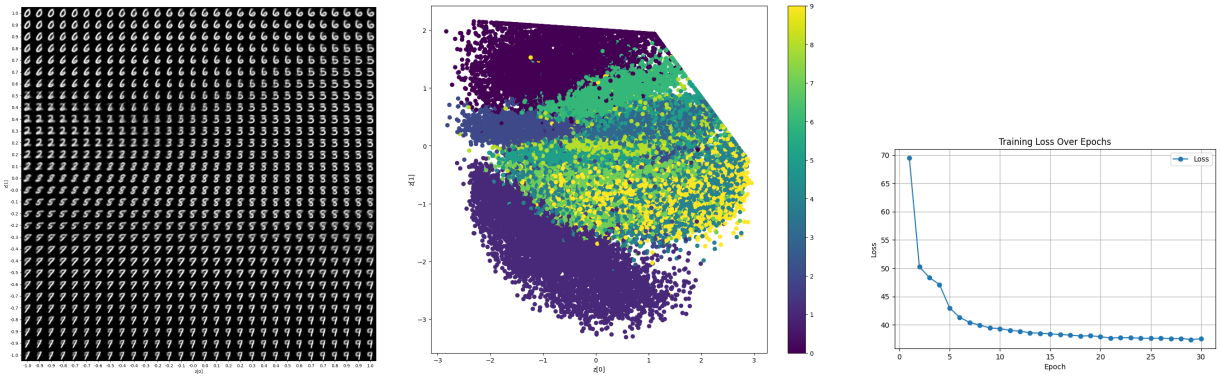
While we allocated 10 latent dimensions, its possible that not all of them were utilized equally and we could confirm this by checking the mean, variance, and or KL-divergence breakdowns per dimension.

1.3 Loss Function

We replaced the original binary cross-entropy loss with a mean squared error term for reconstruction, keeping the KL-divergence term as is. Architecture and latent space dimension of 2 is kept in accordance to our original model and output.

```
with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            reconstruction_loss = ops.mean(
                ops.sum(
                    keras.losses.mean_squared_error(data, reconstruction),
                    axis=(1, 2),
                )
            )
            kl_loss = -0.5 * (1 + z_log_var - ops.square(z_mean) - ops.exp(z_log_var))
            kl_loss = ops.mean(ops.sum(kl_loss, axis=1))
            total_loss = reconstruction_loss + kl_loss
```

With this adjustment of our reconstruction loss, we get the following output.



(a) Latent Space Plot (b) Test Data in Latent Space Plot (c) Loss over Epoch Plot

Figure 6: Loss Function (MSE) Output

The total time it took to train was **2m 47.0015s** with a final loss value of **37.5024**. An initial observation is that the loss over epoch is much lower in comparison to the original. Using a 2D latent space means the model has fewer parameters to learn in comparison to 10D or higher which may imply the training time was expected

Mean squared error (MSE) can sometimes converge more smoothly when the latent space is low-dimensional, as the model has fewer degrees of freedom to represent. In our case, the reconstruction quality, judging by visual inspection of the latent space plots appears comparable to the original binary cross-entropy (BCE) model. However, this alone doesn't imply that MSE or BCE is better for reconstruction, especially since we haven't used numerical metrics to evaluate image quality. It does seem that the latent space learned to separate very few different digit features and cluster them meaningfully.

2 Example Applications

In this section, we build on the autoencoder (AE) developed in Problem 1. Rather than modifying its structure or loss function further, we focus on evaluating how well the trained model performs on 2 tasks. Specifically, we will apply:

- **Inpainting:** Removing patches of an image and attempting to reconstruct them.
- **Generation:** Sampling from the latent space to produce entirely new outputs.

We will use the 10D model. Arguably, the 10D model performed "best" out of all models in Problem 1. The model having more degrees of freedom to distribute classes may lead to better results on inpainting.

2.1 Inpainting

3 images were selected from the MNIST test set, each originally 28×28 pixels. We artificially "masked" them by zeroing out a 10×10 patch near the center of the image (rows 10–20, columns 10–20). The resulting masked images were then fed into our trained VAE to see how well it could reconstruct the missing patch.

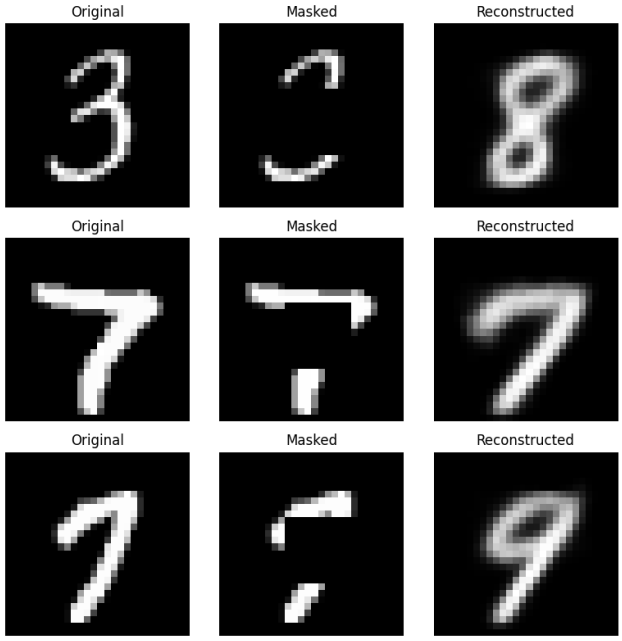


Figure 7: Inpainted Outputs

The VAE often recreates a plausible digit, but not always the correct one. The masked 3 is reconstructed as an 8, suggesting that once a key portion of the top is removed, the model sees enough partial overlap in shape to fill in something resembling an 8. The masked 7 in the second row is reasonably reconstructed back to 7, whereas the third example is misinterpreted as a 9 (could be argued that the original is a 1, but respectfully not a 9). These results support the idea that while the 10D VAE can inpaint missing patches in a consistent manner compared to a 2D VAE, it does not strictly restore the original class if the remaining pixels are ambiguous or uncertain. The network learns a continuous manifold of numbers, so it fills in missing areas with whatever it deems most likely.

2.2 Generation

A key advantage of Variational Autoencoders (VAEs) is their ability to generate new samples simply by providing randomly sampled latent vectors to the decoder.

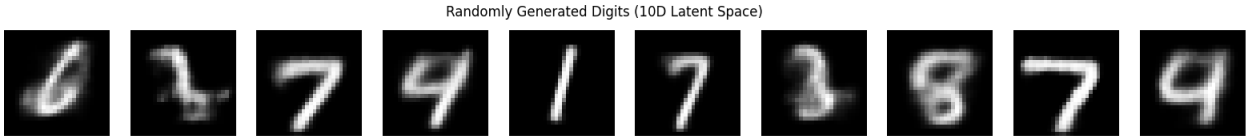


Figure 8: Generated Outputs

By sampling from a 10-dimensional standard normal distribution and passing these samples through the trained decoder, we obtain entirely novel digits that did not exist in the original dataset. Because VAEs model a continuous latent space, small changes in the latent coordinates typically lead to smooth variations in the generated images. We saw before that digits morph into one another as we move around our latent space, reflecting the learned "manifold" of digit shapes. Randomly sampling from this manifold is more likely to produce recognizable digits than random sampling in the original pixel space.

3 Additional Discussion

3.1 Deciding a latent space dimension

One approach is to start with a moderate dimension (3 or 4) and adjust from there. If the latent space is too small, blurry or constrained reconstructions may pop up; if it's too large, the model might not fully use all dimensions or overfit. In a VAE, examining the KL divergence can help reveal whether extra dimensions remain underused, or if the model still needs more capacity. With regard to the complexity of the data in question, it may guide how many dimensions are needed. In the mean while, validation loss and reconstruction quality can support such claims.

3.2 Consideration on using the training set for testing

Using the training set for tasks like inpainting would yield somewhat better results, since the model has already "seen" and learned from those samples. Testing on unseen data is basically needed to verify and to ensure the autoencoder is capturing the broader data structure rather than overfitting.

4 Appendix

4.1 Appendix A: Epoch Loss Function Plot

```python
import matplotlib.pyplot as plt

def plot_loss(loss_values):
    epochs = list(range(1, 31))

    plt.figure(figsize=(8, 5))
    plt.plot(epochs, loss_values, marker='o', label='Loss')
    plt.title("Training Loss Over Epochs")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.grid(True)
    plt.legend()
    plt.show()

plot_loss(epoch_loss)
```

4.2 Appendix B: t-SNE Function Plot

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

def plot_label_clusters_10d(vae, data, labels, num_samples=5000):
    # Subsample cause MNIST is large
    data = data[:num_samples]
    labels = labels[:num_samples]

    # Encode images into 10D latent means
    z_mean, _, _ = vae.encoder.predict(data, verbose=0)  # shape: (num_samples, 10)

    # Use t-SNE to project down to 2D for visualization
    z_2d = TSNE(n_components=2, learning_rate='auto', init='pca').fit_transform(z_mean)

    # Create 2D scatter plot
    plt.figure(figsize=(8, 6))
    scatter = plt.scatter(z_2d[:, 0], z_2d[:, 1], c=labels, s=5, alpha=0.7, cmap='tab10')
    plt.colorbar(scatter, ticks=range(10))
    plt.title("t-SNE Projection of 10D Latent Space")
    plt.xlabel("Dimension 1 (t-SNE)")
    plt.ylabel("Dimension 2 (t-SNE)")
    plt.show()

(x_train, y_train), _ = keras.datasets.mnist.load_data()
x_train = np.expand_dims(x_train, -1).astype("float32") / 255
plot_label_clusters_10d(vae, x_train, y_train, num_samples=5000)
```

4.3 Appendix C: Inpainting Block

```
import random
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_test = np.expand_dims(x_test, -1).astype("float32") / 255.0

# 3 random test images
random.seed(471)
indices = random.sample(range(len(x_test)), 3)
original_images = x_test[indices].copy()

# remove a chunk of pixels
inpainted_images = original_images.copy()
for i in range(len(inpainted_images)):
    inpainted_images[i, 10:20, 10:20, 0] = 0.0

# pass masked images through VAE
z_mean, z_log_var, z = vae.encoder(inpainted_images)
reconstructed_images = vae.decoder(z)

fig, axes = plt.subplots(3, 3, figsize=(8, 8))
# First image
axes[0, 0].imshow(original_images[0].reshape(28, 28), cmap="gray")
axes[0, 0].set_title("Original")
axes[0, 0].axis("off")

axes[0, 1].imshow(inpainted_images[0].reshape(28, 28), cmap="gray")
axes[0, 1].set_title("Masked")
axes[0, 1].axis("off")

axes[0, 2].imshow(reconstructed_images[0].numpy().reshape(28, 28), cmap="gray")
axes[0, 2].set_title("Reconstructed")
axes[0, 2].axis("off")

# Second image
axes[1, 0].imshow(original_images[1].reshape(28, 28), cmap="gray")
axes[1, 0].set_title("Original")
axes[1, 0].axis("off")

axes[1, 1].imshow(inpainted_images[1].reshape(28, 28), cmap="gray")
axes[1, 1].set_title("Masked")
axes[1, 1].axis("off")

axes[1, 2].imshow(reconstructed_images[1].numpy().reshape(28, 28), cmap="gray")
axes[1, 2].set_title("Reconstructed")
axes[1, 2].axis("off")

# Third image
axes[2, 0].imshow(original_images[2].reshape(28, 28), cmap="gray")
axes[2, 0].set_title("Original")
axes[2, 0].axis("off")

axes[2, 1].imshow(inpainted_images[2].reshape(28, 28), cmap="gray")
axes[2, 1].set_title("Masked")
axes[2, 1].axis("off")

axes[2, 2].imshow(reconstructed_images[2].numpy().reshape(28, 28), cmap="gray")
axes[2, 2].set_title("Reconstructed")
axes[2, 2].axis("off")

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

5 References

[1] F.Chollet, Variational AutoEncoder Example, Keras, 2024.DIRECT LINK
[2] Wikipedia, t-SNE. DIRECT LINK
[3] builtin, Using T-SNE in Python. DIRECT LINK