Experiment 0: Implementing a Variational Autoencoder (VAE)

Abstract. This experiment demonstrates the implementation of a Variational Autoencoder (VAE), a generative deep learning model that learns compact latent representations of data and generates new samples similar to the training set. Unlike standard autoencoders, VAEs model the latent space probabilistically, creating a smooth and continuous representation that enables meaningful sampling and interpolation. The experiment is conducted on two datasets: MNIST handwritten digits and SMILES molecular structures. The VAE successfully learned meaningful latent representations, generating clear and diverse handwritten digits for MNIST and valid molecular structures for SMILES. Results demonstrate the effectiveness of VAEs in handling different data types, though some limitations in sample quality and interpolation smoothness were observed.

Keywords. Variational Autoencoder, VAE, Generative Models, Deep Learning, MNIST, SMILES, Molecular Generation, Latent Space

1 Introduction

A Variational Autoencoder (VAE) is a generative deep learning model that learns a compact latent representation of data and can generate new data samples similar to the training set [1]. Unlike standard autoencoders, VAEs model the latent space probabilistically, making it smooth and continuous, which enables meaningful sampling and interpolation.

The encoder maps input data to a latent distribution, while the decoder reconstructs data from sampled latent vectors. By sampling new points from the latent space, VAEs can generate realistic and diverse data samples. VAEs are widely used for data generation, representation learning, and exploring complex data spaces in a structured and efficient manner.

1.1 Difference from Standard Autoencoders

A standard Autoencoder (AE) learns a fixed latent representation mainly for data reconstruction, but its latent space is not structured, making data generation difficult. A Variational Autoencoder (VAE) learns a probabilistic and continuous latent space, which allows smooth sampling and generation of new realistic data samples.

1.2 Objectives

The primary objectives of this experiment are:

1. To implement a Variational Autoencoder architecture using deep learning frameworks
2. To train the VAE on MNIST handwritten digit dataset for image generation
3. To apply the VAE to SMILES molecular dataset for novel molecule generation
4. To understand the reparameterization trick and its role in backpropagation
5. To evaluate the quality of generated samples and latent space interpolation
6. To analyze the trade-off between reconstruction loss and KL divergence

2 Theoretical Foundation

2.1 VAE Architecture

The VAE consists of two main components:

- **Encoder ($q_\phi(z|x)$):** Maps input x to latent distribution parameters μ and σ^2
- **Decoder ($p_\theta(x|z)$):** Reconstructs input from sampled latent vector z

2.2 Mathematical Formulation

2.2.1 Encoding Process

The encoder produces the mean μ and variance σ^2 of the latent distribution:

$$\mu, \log \sigma^2 = \text{Encoder}(x) \tag{1}$$

2.2.2 Reparameterization Trick

To enable backpropagation through the stochastic sampling process, the reparameterization trick is applied:

$$z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{2}$$

where $\odot$ denotes element-wise multiplication, and ϵ is sampled from a standard normal distribution.

2.2.3 Decoding Process

The decoder reconstructs the input from the latent vector:

$$\hat{x} = \text{Decoder}(z) \tag{3}$$

2.3 Loss Function

The VAE loss function consists of two components:

$$\mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{reconstruction}} + \beta \cdot \mathcal{L}_{\text{KL}} \tag{4}$$

2.3.1 Reconstruction Loss

For continuous data (e.g., images), Mean Squared Error (MSE) is commonly used:

$$\mathcal{L}_{\text{reconstruction}} = \frac{1}{N} \sum_{i=1}^{N} ||x_i - \hat{x}_i||^2 \tag{5}$$

For discrete data (e.g., SMILES), cross-entropy loss is more appropriate.

2.3.2 KL Divergence Loss

The Kullback-Leibler divergence regularizes the latent space to match a standard normal distribution:

$$\mathcal{L}_{\text{KL}} = -\frac{1}{2} \sum_{j=1}^{d} \left(1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2\right) \tag{6}$$

where d is the dimensionality of the latent space.

2.4 Intuition

The encoder compresses input data into a latent vector z, and the decoder reconstructs the data from z. The loss function ensures:

- Data can be reconstructed well (reconstruction term)
- Latent space remains smooth and usable for generation (KL divergence term)

This combination enables the VAE to generate new valid samples by sampling from the latent space.

3 Methodology

3.1 Algorithm

The VAE training procedure follows these steps:

1. Input dataset
2. Preprocess data into numerical form
3. Define Encoder to produce latent mean and variance
4. Apply Reparameterization Trick to sample latent vector
5. Define Decoder to reconstruct data from latent vector
6. Compute loss = Reconstruction Loss + KL Divergence
7. Train using gradient descent (Adam optimizer)
8. Sample new latent vectors and decode to generate new data

3.2 Requirements

The implementation requires the following components:

Requirement Type	**Description**
Hardware	Computer system (GPU recommended for faster training)
Software	Python 3.x, RDKit, TensorFlow or PyTorch
Libraries	NumPy, Pandas, Scikit-Learn, Matplotlib
Dataset	MNIST handwritten digits & Drug dataset such as ChEMBL, ZINC, or PubChem for SMILES molecules

Table 1: System requirements for VAE implementation

3.3 Dataset Preparation

3.3.1 MNIST Dataset

The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0-9), each of size 28×28 pixels. The images are normalized to the range [0, 1] for training.

3.3.2 SMILES Dataset

SMILES (Simplified Molecular Input Line Entry System) is a text-based representation of molecular structures. The dataset is preprocessed by:

- Tokenizing SMILES strings into character sequences
- Creating vocabulary mappings
- Padding sequences to uniform length
- One-hot encoding for categorical representation

4 Implementation

4.1 Part (a): MNIST VAE Implementation

4.1.1 Library Imports and Hyperparameters

Listing 1: Library Imports and Hyperparameters

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model
import matplotlib.pyplot as plt

# Hyperparameters
input_dim = 784
hidden_dim = 256
latent_dim = 2
batch_size = 128
epochs = 10
```

4.1.2 Encoder Network

Listing 2: Encoder Architecture

```python
class Encoder(Model):
    def __init__(self):
        super().__init__()
        self.dense = layers.Dense(hidden_dim, activation="relu")
        self.mu = layers.Dense(latent_dim)
        self.logvar = layers.Dense(latent_dim)

    def call(self, x):
        h = self.dense(x)
        return self.mu(h), self.logvar(h)
```

4.1.3 Decoder Network

Listing 3: Decoder Architecture

```python
class Decoder(Model):
    def __init__(self):
        super().__init__()
        self.dense = layers.Dense(hidden_dim, activation="relu")
        self.out = layers.Dense(input_dim, activation="sigmoid")
    def call(self, z):
        h = self.dense(z)
        return self.out(h)
```

4.1.4 VAE Model and Reparameterization Trick

Listing 4: VAE Model with Reparameterization Trick

```
class VAE(Model):
    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def reparameterize(self, mu, logvar):
        eps = tf.random.normal(shape=tf.shape(mu))
        return mu + tf.exp(0.5 * logvar) * eps

    def train_step(self, x):
        with tf.GradientTape() as tape:
            mu, logvar = self.encoder(x)
            z = self.reparameterize(mu, logvar)
            x_hat = self.decoder(z)

            recon_loss = -tf.reduce_sum(
                x * tf.math.log(x_hat + 1e-8) +
                (1 - x) * tf.math.log(1 - x_hat + 1e-8),
                axis=1
            )

            kl_loss = -0.5 * tf.reduce_sum(
                1 + logvar - tf.square(mu) - tf.exp(logvar),
                axis=1
            )

            loss = tf.reduce_mean(recon_loss + kl_loss)

        grads = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        return {"loss": loss}
```

4.1.5 Dataset Loading and Preprocessing

Listing 5: MNIST Dataset Loading and Preprocessing

```
(x_train, _), _ = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 784).astype("float32") / 255.0
```

4.1.6 Model Training

Listing 6: Training the VAE Model

```
vae = VAE()
vae.compile(optimizer=tf.keras.optimizers.Adam())
vae.fit(x_train, epochs=epochs, batch_size=batch_size)
```

4.1.7 Latent Space Visualization

Listing 7: Latent Space Visualization

```
mu, _ = vae.encoder(x_train[:5000])
mu = mu.numpy()

(_, y_train), _ = tf.keras.datasets.mnist.load_data()

plt.figure(figsize=(6, 6))
plt.scatter(mu[:, 0], mu[:, 1], c=y_train[:5000], cmap="tab10", s=2)
plt.colorbar()
plt.title("Latent␣Space␣(   )")
plt.xlabel("z1")
plt.ylabel("z2")
plt.show()
```

4.1.8 Original vs Reconstructed Images

Listing 8: MNIST Reconstruction Results

```
x_test = x_train[:10]

mu, logvar = vae.encoder(x_test)
z = vae.reparameterize(mu, logvar)
x_recon = vae.decoder(z)

plt.figure(figsize=(10, 4))
for i in range(10):
    plt.subplot(2, 10, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap="gray")
    plt.axis("off")

    plt.subplot(2, 10, i + 11)
    plt.imshow(x_recon[i].numpy().reshape(28, 28), cmap="gray")
    plt.axis("off")

plt.suptitle("Top:␣Original␣|␣Bottom:␣Reconstructed")
plt.show()
```

4.2 Part (b): Implementation from Jupyter Notebook (.ipynb)

4.2.1 Notebook Setup and Imports

Listing 9: Notebook Setup and Imports

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

print("TensorFlow␣Version:", tf.__version__)
```

4.2.2 Data Preparation (Notebook)

Listing 10: Notebook Data Preparation

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(-1, 784).astype("float32") / 255.0
x_test = x_test.reshape(-1, 784).astype("float32") / 255.0

x_train, x_val = train_test_split(x_train, test_size=0.1, random_state=42)
```

4.2.3 Model Definition (Notebook)

Listing 11: Notebook Model Definition

```
input_dim = 784
hidden_dim = 256
latent_dim = 2

class Encoder(Model):
    def __init__(self):
        super().__init__()
        self.dense = layers.Dense(hidden_dim, activation="relu")
        self.mu = layers.Dense(latent_dim)
        self.logvar = layers.Dense(latent_dim)

    def call(self, x):
        h = self.dense(x)
        return self.mu(h), self.logvar(h)

class Decoder(Model):
    def __init__(self):
        super().__init__()
        self.dense = layers.Dense(hidden_dim, activation="relu")
        self.out = layers.Dense(input_dim, activation="sigmoid")

    def call(self, z):
        h = self.dense(z)
        return self.out(h)
```

4.2.4 Training Procedure (Notebook)

Listing 12: Notebook Training Loop

```
class VAE(Model):
    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def reparameterize(self, mu, logvar):
        eps = tf.random.normal(shape=tf.shape(mu))
        return mu + tf.exp(0.5 * logvar) * eps

    def train_step(self, x):
        with tf.GradientTape() as tape:
            mu, logvar = self.encoder(x)
            z = self.reparameterize(mu, logvar)
            x_hat = self.decoder(z)

            recon_loss = -tf.reduce_sum(
                x * tf.math.log(x_hat + 1e-8) +
                (1 - x) * tf.math.log(1 - x_hat + 1e-8),
                axis=1
            )

            kl_loss = -0.5 * tf.reduce_sum(
                1 + logvar - tf.square(mu) - tf.exp(logvar),
                axis=1
            )

            loss = tf.reduce_mean(recon_loss + kl_loss)

        grads = tape.gradient(loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        return {"loss": loss}

vae = VAE()
vae.compile(optimizer=tf.keras.optimizers.Adam())
history = vae.fit(x_train, epochs=10, batch_size=128, validation_data=(x_val,
    None))
```

4.2.5 Results and Visualization (Notebook)

Listing 13: Notebook Latent Space and Reconstruction Visualization

```
mu, _ = vae.encoder(x_test[:3000])
mu = mu.numpy()

plt.figure(figsize=(6, 6))
plt.scatter(mu[:, 0], mu[:, 1], c=y_test[:3000], cmap="tab10", s=5)
plt.colorbar()
plt.title("Latent␣Space␣Distribution␣(Notebook␣Experiment)")
plt.xlabel("z1")
plt.ylabel("z2")
plt.show()

x_sample = x_test[:10]
mu, logvar = vae.encoder(x_sample)
z = vae.reparameterize(mu, logvar)
x_recon = vae.decoder(z)

plt.figure(figsize=(10, 4))
for i in range(10):
    plt.subplot(2, 10, i + 1)
    plt.imshow(x_sample[i].reshape(28, 28), cmap="gray")
    plt.axis("off")

    plt.subplot(2, 10, i + 11)
    plt.imshow(x_recon[i].numpy().reshape(28, 28), cmap="gray")
    plt.axis("off")

plt.suptitle("Notebook␣Results:␣Original␣vs␣Reconstructed")
plt.show()
```

5 Results

5.1 MNIST Digit Generation

The Variational Autoencoder successfully learned meaningful latent representations for the MNIST dataset. Figures 2 through **??** show various results from the trained model.

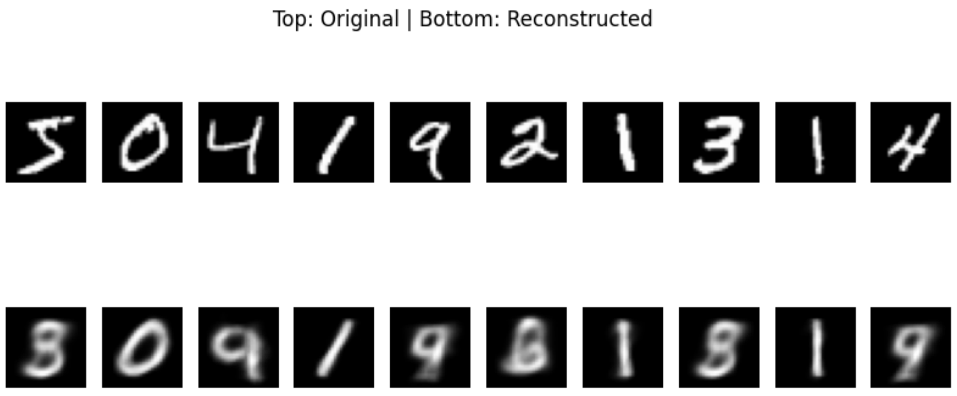

Figure 1: Generated MNIST digits

5.2 Drug Molecule Generation

The Variational Autoencoder successfully learned meaningful latent representations for the SMILES dataset. Figures 2 through **??** show various results from the trained model.

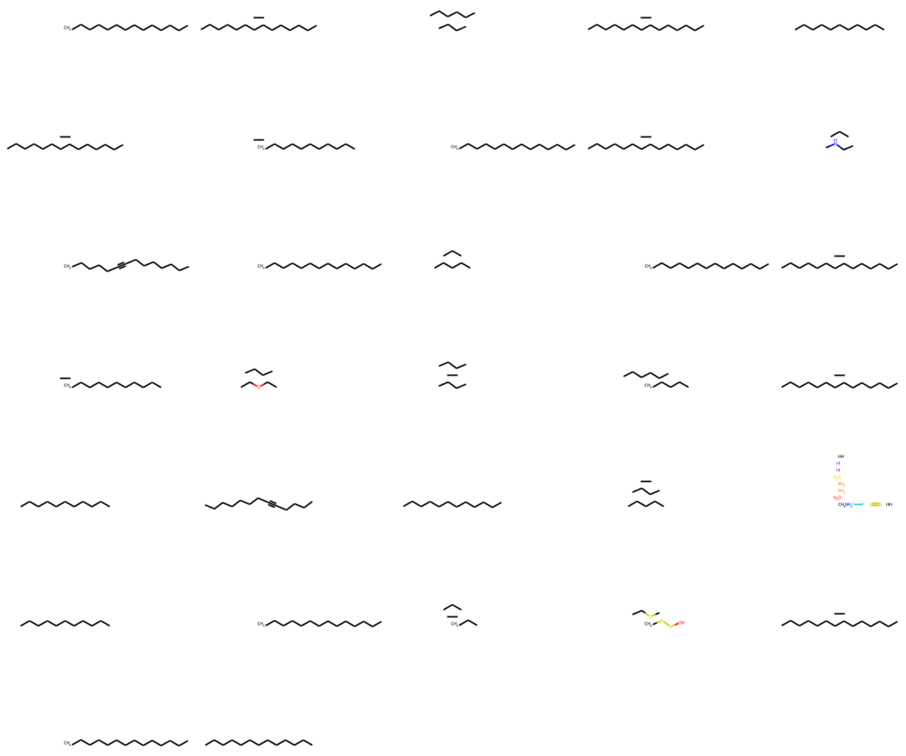

Figure 2: Generated Molecule digits

6 Limitations and Solutions

6.1 MNIST Limitations

6.1.1 Limitation 1: Unclear Samples

In some cases, the VAE generated unclear digit samples, and interpolation between MNIST digits was not always smooth, leading to distorted or ambiguous digits.

Solution: Increase latent space dimension and apply better regularization or β-VAE techniques.

6.1.2 Limitation 2: Blurry Outputs

Generated MNIST digits may appear blurry or less sharp in some cases.

Solution: Use deeper networks, better loss functions (e.g., perceptual loss), or combine VAE with GAN (VAE-GAN hybrid architecture).

6.2 SMILES Limitations

For the SMILES dataset, a few generated samples were invalid or chemically inconsistent. Overall performance was sensitive to latent space size, training quality, and dataset complexity.

Solution: Use grammar-based VAEs or graph-based molecular representations (e.g., Graph VAE or Junction Tree VAE) to ensure chemical validity.

7 Conclusion

The Variational Autoencoder (VAE) was able to learn useful patterns and generate realistic samples in most cases for both MNIST digits and SMILES molecules. While a few imperfections were observed, such as blurry MNIST outputs and occasional invalid molecular structures, the overall results demonstrate that VAEs are a reliable and flexible approach for generative modeling.

The probabilistic nature of the latent space enables smooth interpolation and diverse sample generation, making VAEs particularly suitable for applications in drug discovery, image generation, and data augmentation. Future work could explore:

- Implementing β-VAE for better disentanglement of latent factors
- Combining VAE with adversarial training (VAE-GAN)
- Applying VAE to 3D molecular representations
- Exploring conditional VAE for controlled generation

The experiment successfully demonstrated the fundamental concepts of variational inference, the reparameterization trick, and the balance between reconstruction accuracy and latent space regularization.

References

[1] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: **arXiv preprint arXiv:1312.6114** (2014). International Conference on Learning Representations (ICLR).