Lab Sheet 7: Variational AutoEncoders (VAEs)

7.1 - Overview of VAEs

Variational Autoencoders (VAEs) are a type of generative models that are explicitly designed to capture the underlying probability distribution of a given dataset and generate novel samples.

Architecture

VAEs utilize an architecture that comprises an encoder-decoder structure. The encoder transforms input data into a latent form, and the decoder aims to reconstruct the original data based on this latent representation. The VAE is programmed to minimize the dissimilarity between the original and reconstructed data, enabling it to comprehend the underlying data distribution and generate new samples that conform to the same distribution.

Encoder and Decoder

A VAE typically has two major components: An encoder connection and a decoder connection. The encoder network transforms the input data into a low-dimensional secret space, often called a “secret code”. Various neural network topologies, such as fully connected or convolutional neural networks, can be investigated for implementing encoder networks. The encoder network produces essential parameters, such as the mean and variance of a Gaussian distribution, necessary for sampling and generating the latent code. Similarly, we can construct the decoder network using various types of neural networks, and its objective is to reconstruct the original data from the provided latent code.

Regularization

In addition to the architectural aspects, researchers apply regularization to the latent code, making it a vital element of VAEs. This regularization prevents overfitting by encouraging a smooth distribution of the latent code rather than simply memorizing the training data. The regularization not only aids in generating new data samples that interpolate smoothly between training data points but also contributes to the VAE's ability to generate novel data resembling the training data.

Applications

VAEs find applications in various domains like density estimation and text generation, as we will see. They could also be used to develop personalized medical treatments for patients, design new materials with unique properties, and in creative AI.

Reference Blogs

(1) An Overview of Variational Autoencoders (VAEs).

(2) What are variational autoencoders and what learning tasks they are used for.

(3) The root: Variational Autoencoders (VAEs)

7.2 - Generating text with VAEs

Problem Statement

The objective of this project is to generate text in the style of Shakespeare using Variational Autoencoders (VAEs). However, the twist is that the generated text should not only mimic the style of Shakespeare but also incorporate a modern twist, blending Shakespearean language with contemporary English.

Data Preparation

The first step involves preparing the Shakespeare dataset. This includes cleaning the text, removing unnecessary characters, and converting the text into a suitable format for training the VAE.

Model Design

Design a VAE with an encoder and decoder network. The encoder network should transform the input data (Shakespeare text) into a latent space representation. The decoder network should then reconstruct the original data from this latent space representation.

Training

Train the VAE on the Shakespeare dataset. The loss function should include a reconstruction term (to ensure the generated text is similar to the input text) and a regularization term (to ensure the latent space has good properties).

Generation

Use the trained VAE to generate new text. Sample from the latent space and pass these samples through the decoder to generate new text.

Modern Twist

Introduce a modern twist to the generated text. This could be done in various ways, such as by incorporating modern slang or references into the Shakespearean text.

7.3 - Python Implementation

You will have to do some brainstorming to adjust the hyperparameters and maybe even the model in this exercise to make it run within constraints of your pc. You have to generate suitable Shakespearean text as the result somehow. If you face any issues **make sure you report them in your final notebook** and what steps were taken to alleviate them.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Lambda
from tensorflow.keras.models import Model
from tensorflow.keras import backend as K
from tensorflow.keras.losses import binary_crossentropy
from nltk.tokenize import word_tokenize
from sklearn.model_selection import train_test_split
```

Step 1: Data Preparation

```
def prepare_data(text, num_words):
    # Tokenize the text into words
    words = word_tokenize(text)
    # Create a dictionary mapping words to indices
    word_to_index = {word: i for i, word in enumerate(set(words))}
    # Convert the words to their corresponding indices
    text_indices = np.array([word_to_index[word] for word in words])
    # Convert indices to one-hot vectors
    text_one_hot = tf.keras.utils.to_categorical(text_indices,
num_classes=num_words)
    return text_one_hot, word_to_index
```

Step 2: Model Design

```
def create_vae(input_dim, latent_dim):
    # Encoder
    inputs = Input(shape=(input_dim,))
    h = Dense(64, activation='relu')(inputs)
    z_mean = Dense(latent_dim)(h)
    z_log_var = Dense(latent_dim)(h)

    # Reparameterization trick
    def sampling(args):
        z_mean, z_log_var = args
```

```
        epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
mean=0., stddev=1.)
        return z_mean + K.exp(z_log_var / 2) * epsilon

    z = Lambda(sampling)([z_mean, z_log_var])

    # Decoder
    decoder_h = Dense(64, activation='relu')
    decoder_mean = Dense(input_dim, activation='sigmoid')
    h_decoded = decoder_h(z)
    x_decoded_mean = decoder_mean(h_decoded)

    # VAE model
    vae = Model(inputs, x_decoded_mean)

    # Loss
    reconstruction_loss = binary_crossentropy(inputs, x_decoded_mean) *
input_dim
    kl_loss = - 0.5 * K.sum(1 + z_log_var - K.square(z_mean) -
K.exp(z_log_var), axis=-1)
    vae_loss = K.mean(reconstruction_loss + kl_loss)

    vae.add_loss(vae_loss)
    return vae
```

Step 3: Training

```
def train_vae(vae, data, epochs=50, batch_size=100):
    vae.compile(optimizer='adam')
    for epoch in range(epochs):
        print(f"Epoch {epoch+1}/{epochs}")
        for i in range(0, len(data), batch_size):
            batch = data[i:i+batch_size]
            loss = vae.train_on_batch(batch, None)
            print(f"  Batch {i//batch_size+1}/{len(data)//batch_size},
Loss: {loss:.4f}", end='\r')
        print()
```

Step 4: Generation

```
def generate_text(vae, word_to_index, seed_word, length):
    index_to_word = {i: word for word, i in word_to_index.items()}
    current_word = seed_word
```

```
    generated_text = [current_word]
    for i in range(length):
        x_pred = np.zeros((1, len(word_to_index)))
        x_pred[0, word_to_index[current_word]] = 1
        next_index = vae.predict(x_pred.reshape(1, -1))[0].argmax()  #
Reshape the input
        next_word = index_to_word[next_index]
        generated_text.append(next_word)
        current_word = next_word
    return ' '.join(generated_text)
```

Step 5: Modern Twist

```
def add_modern_twist(text):

    # You can read about style transfer in text for more complicated
examples
def add_modern_twist(text):
    shakespeare_to_modern = {
        'thou': 'you',
        'thy': 'your',
        'hast': 'have',
        'art': 'are',
        'doth': 'does',
        'hath': 'has',
    }

    # Tokenize the text into words
    words = word_tokenize(text)

    # Replace old English words with their modern equivalents
    modern_words = [shakespeare_to_modern.get(word, word) for word in
words]

    return ' '.join(modern_words)
```

Load your Shakespeare dataset

```
text = open('shakespeare.txt').read()
```

Prepare the data

```
num_words = len(set(word_tokenize(text)))
text_one_hot, word_to_index = prepare_data(text, num_words)
```

```
# Create the VAE
vae = create_vae(num_words, 50)

# Train the VAE
train_vae(vae, text_one_hot)

# Generate new text
generated_text = generate_text(vae, word_to_index, 'the', 100)
```

Add a modern twist

```
altered_text = add_modern_twist(generated_text)

print(altered_text)
```

7.4 - Image generation with VAEs

Problem Statement

Image Generation from a Small Public Dataset. (MNIST)

Objective

The goal of this task is to generate new images that resemble the original dataset using a generative model trained on the relatively small public dataset, MNIST.

Tasks

1. **Data:** We will be using the MNIST dataset.

2. **Preprocessing** Preprocess the images to make them suitable for input into the model. This might involve resizing, normalization, etc.

3. **Model Building:** Design and train a generative model on the collected dataset. The architecture of the model should be designed keeping in mind the size of the dataset.

4. **Image Generation**: Use the trained model to generate new images. The generated images should be visually similar to the images in the original dataset. For instance, if the original dataset contains images of dogs, the generated images should also resemble dogs. The quality of the generated images can be evaluated using human judgment or automated metrics like Inception Score (IS) or Frechet Inception Distance (FID), as we had discussed in previous lab sheets.

7.5 - Python Implementation

```
import numpy as np
from keras.layers import Input, Dense, Lambda
from keras.models import Model
from keras import backend as K
from keras import metrics
from keras.datasets import mnist
```

hyperparameters

```
batch_size = 100
original_dim = 784  # images are 28x28 pixels
latent_dim = 2
intermediate_dim = 256
epochs = 50
epsilon_std = 1.0
```

encoder

```
x = Input(shape=(original_dim,))
h = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim)(h)
z_log_var = Dense(latent_dim)(h)
```

sampling from latent space

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
mean=0., stddev=epsilon_std)
    return z_mean + K.exp(z_log_var / 2) * epsilon

z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_var])
```

decoder

```
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)
```

end-to-end autoencoder

```
vae = Model(x, x_decoded_mean)
```

encoder, from inputs to latent space

```
encoder = Model(x, z_mean)
```

generator, from latent space to reconstructed inputs

```
decoder_input = Input(shape=(latent_dim,))
_h_decoded = decoder_h(decoder_input)
_x_decoded_mean = decoder_mean(_h_decoded)
generator = Model(decoder_input, _x_decoded_mean)
```

loss

```
xent_loss = original_dim * metrics.binary_crossentropy(x, x_decoded_mean)
kl_loss = - 0.5 * K.sum(1 + z_log_var - K.square(z_mean) -
K.exp(z_log_var), axis=-1)
vae_loss = K.mean(xent_loss + kl_loss)
```

```
vae.add_loss(vae_loss)
vae.compile(optimizer='rmsprop')
vae.summary()
```

train

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

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

vae.fit(x_train, shuffle=True, epochs=epochs, batch_size=batch_size,
validation_data=(x_test, None))
```

display a 2D plot of the digit classes in the latent space

```
x_test_encoded = encoder.predict(x_test, batch_size=batch_size)
plt.figure(figsize=(6, 6))
plt.scatter(x_test_encoded[:, 0], x_test_encoded[:, 1], c=y_test)
plt.colorbar()
plt.show()
```

display a 2D manifold of the digits

```
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
grid_x = np.linspace(-15, 15, n)
grid_y = np.linspace(-15, 15, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        x_decoded = generator.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```

Again, your **objective** is to visualize the VAE latent space and to train the model appropriately. You will have to make various tweaks to the model itself and the hyperparameters to achieve this. This is to make sure you thoroughly understand the implementation of this problem statement.