UNSUPERVISED MACHINE LEARNING:

MUSIC GENERATION USING PROBABILISTIC LATENT REPRESENTATIONS (VARIATIONAL AUTOENCODERS)

A029: Rhea Menon & A037: Nidhi Dahiya | Semester II

Code file: Music_Gen.ipynb

Introduction: Generative AI and the rise of digitisation

Over the past decade, AI has exponentially grown in popularity and usage across every domain in the world. Specifically, the term 'Deep Learning' has been labelled to algorithms that can perform extremely complex tasks such as object detection, audio classification, and even language translation. What is unique about them is that the input and output variables are different from each other, and the output provides some additional information about the input itself.

There is a special class of neural networks that not only provide additional information about an input sample, but they generate a whole new sample, whether it be text, images, audio, or video. They are called generative models. What is an input to a neural network, is an output to a generative model.

Problem Statement: The music originality problem

One of the primary issues faced by musicians is the time-consuming nature of composing music, especially for those seeking to create high-quality, original compositions. The challenge of writer's block is also a significant hurdle in the process of making music. There is a multitude of repetitive music that almost sounds the same, and from a psychological point of view, people tend to replicate the music others make at some point in time.

The Solution: Variational Auto-Encoders

Variational autoencoders (VAEs) are a type of generative model that learns a latent representation of input data. They consist of an encoder, which maps input data into a vector onto a latent space, and a decoder, which reconstructs the input data from the latent space.

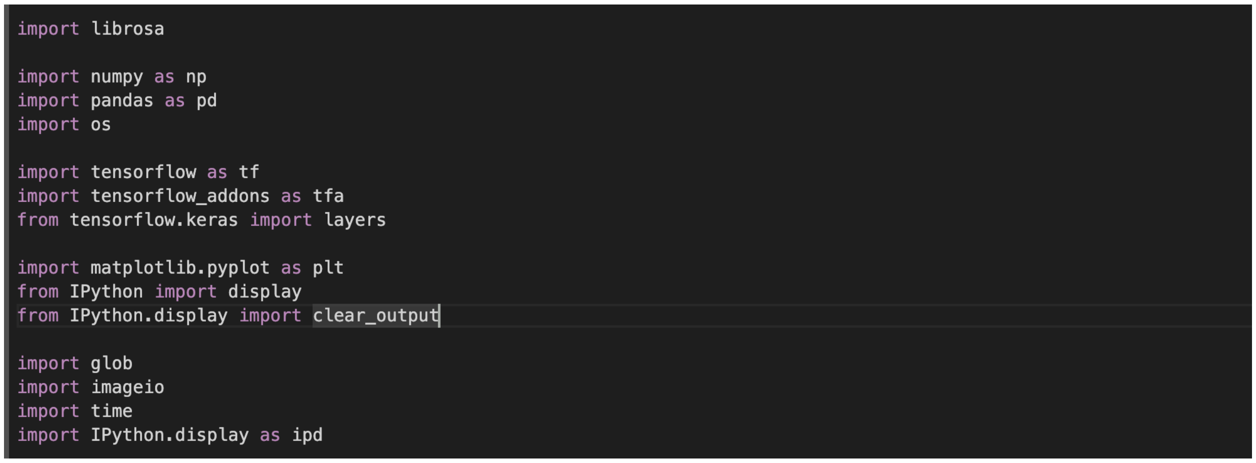
Why VAE?

In theory, there is no significant point of reconstructing an input data back from a latent space representation of the same data. However, for autoencoder structures, we are never concerned with the output itself. The important part is the vector representation of the input data.

VAEs differ from traditional autoencoders by introducing a probabilistic approach to learning latent representations. Instead of learning a deterministic mapping, VAEs learn a distribution over the latent space, typically a Gaussian distribution. This allows VAEs to generate new data points by sampling from the learned distribution in the latent space.

Using VAEs to generate music: The prerequisites

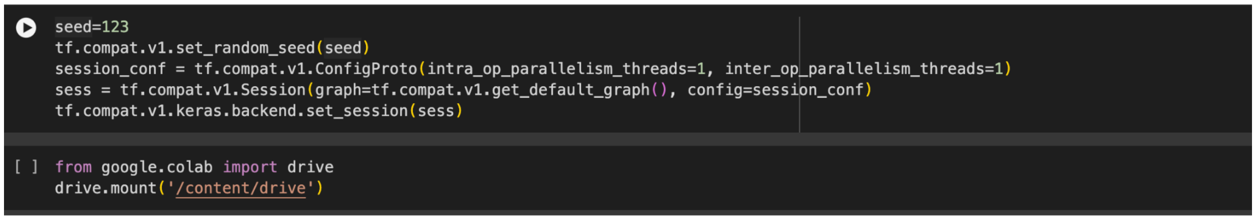
To dive into music generation with VAEs, the initial section of the code imports necessary libraries and sets up configurations for TensorFlow. Libraries such as librosa, numpy, pandas, and tensorflow are imported to handle audio processing, data manipulation, and neural network operations. With these tools, we can preprocess music data and train VAEs to learn the underlying structure of music sequences, enabling the generation of new music compositions.



```
import librosa

import numpy as np
import pandas as pd
import os

import tensorflow as tf
import tensorflow_addons as tfa
from tensorflow.keras import layers

import matplotlib.pyplot as plt
from IPython import display
from IPython.display import clear_output

import glob
import imageio
import time
import IPython.display as ipd
```

TensorFlow configurations are set, including random seeds and session configurations, to ensure reproducibility of results.

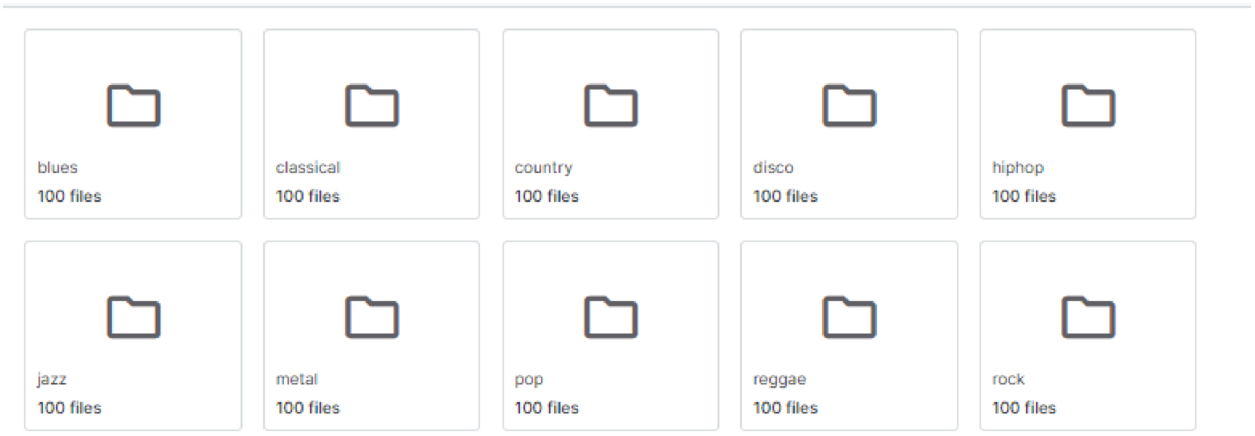
Additionally, Google Drive is mounted using Google Colab to access data stored in Google Drive.



```
seed=123
tf.compat.v1.set_random_seed(seed)
session_conf = tf.compat.v1.ConfigProto(intra_op_parallelism_threads=1, inter_op_parallelism_threads=1)
sess = tf.compat.v1.Session(graph=tf.compat.v1.get_default_graph(), config=session_conf)
tf.compat.v1.keras.backend.set_session(sess)
```

```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

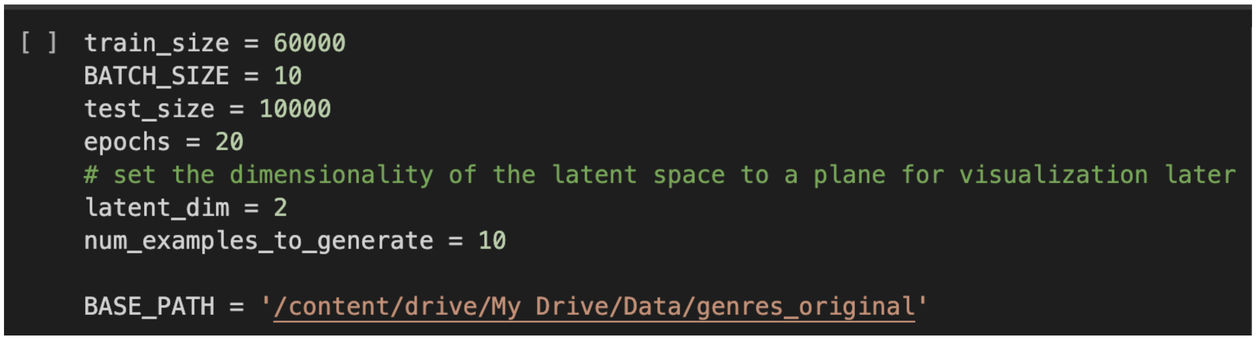
The Dataset



The genres folder consists of further subfolders with 100 files of audios each spanning 30 seconds. The music to be generated by the VAE will work with these samples as the training data.

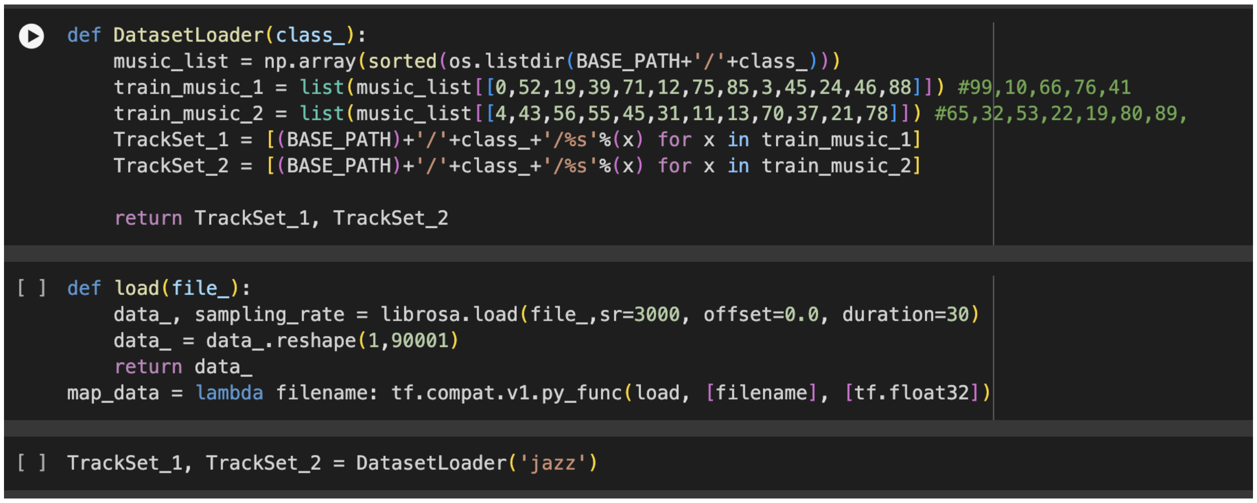
Data preprocessing

The train-test split is a method used in machine learning to divide a dataset into two parts: one for training the model and the other for testing its performance. The training set is used to teach the model, while the testing set evaluates how well it performs on unseen data. This helps assess the model's ability to generalize to new data. Typically, data is randomly split, often with ratios like 70-30 or 80-20 for training and testing, respectively.



```
[ ] train_size = 60000
    BATCH_SIZE = 10
    test_size = 10000
    epochs = 20
    # set the dimensionality of the latent space to a plane for visualization later
    latent_dim = 2
    num_examples_to_generate = 10

    BASE_PATH = '/content/drive/My Drive/Data/genres_original'
```

The code defines variables for train_size, test_size, and BATCH_SIZE. These are used to split the music data into training and testing sets, and to define the batch size for training the model. BATCH_SIZE specifies the number of samples processed by the model during a single training iteration. A smaller batch size might lead to slower convergence but potentially better generalization, while a larger batch size could converge faster but might be more prone to overfitting.



```python
def DatasetLoader(class_):
    music_list = np.array(sorted(os.listdir(BASE_PATH+'/'+class_)))
    train_music_1 = list(music_list[[0,52,19,39,71,12,75,85,3,45,24,46,88]]) #99,10,66,76,41
    train_music_2 = list(music_list[[4,43,56,55,45,31,11,13,70,37,21,78]]) #65,32,53,22,19,80,89,
    TrackSet_1 = [(BASE_PATH)+'/'+class_+'/%s'%(x) for x in train_music_1]
    TrackSet_2 = [(BASE_PATH)+'/'+class_+'/%s'%(x) for x in train_music_2]

    return TrackSet_1, TrackSet_2
```

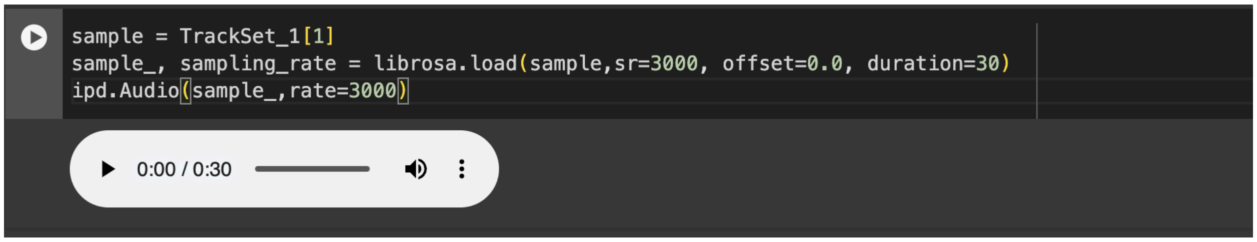
```python
def load(file_):
    data_, sampling_rate = librosa.load(file_,sr=3000, offset=0.0, duration=30)
    data_ = data_.reshape(1,90001)
    return data_
map_data = lambda filename: tf.compat.v1.py_func(load, [filename], [tf.float32])
```

```python
TrackSet_1, TrackSet_2 = DatasetLoader('jazz')
```

The DatasetLoader function takes a class name (class_) as input, indicating the genre of music, in this case, 'jazz'. It constructs two sets of tracks (TrackSet_1 and TrackSet_2), each containing a selection of music tracks from the specified class. These tracks are obtained from a base path (BASE_PATH) and are sorted. The function returns these two sets of tracks.

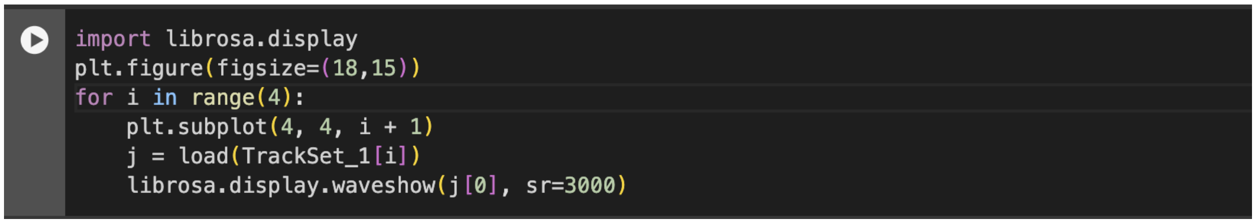
The load function loads audio data from the given file using the librosa library. It loads the data with a specified sampling rate (sr, it determines the rate at which audio data is sampled per second. In this case, sr=3000 means 3000 samples per second. It affects audio fidelity and computational resources) and reshapes it into a 1D array of shape (1, 90001), (this format is suitable for processing and analysis, especially for machine learning tasks, where each row represents an audio sample and each column represents a sample of audio data). The function returns the loaded data and its sampling rate.

“map_data” is a lambda function used to map the load function to each filename. It's intended to be used within TensorFlow's data pipeline (tf.data.Dataset.map) for loading audio data.



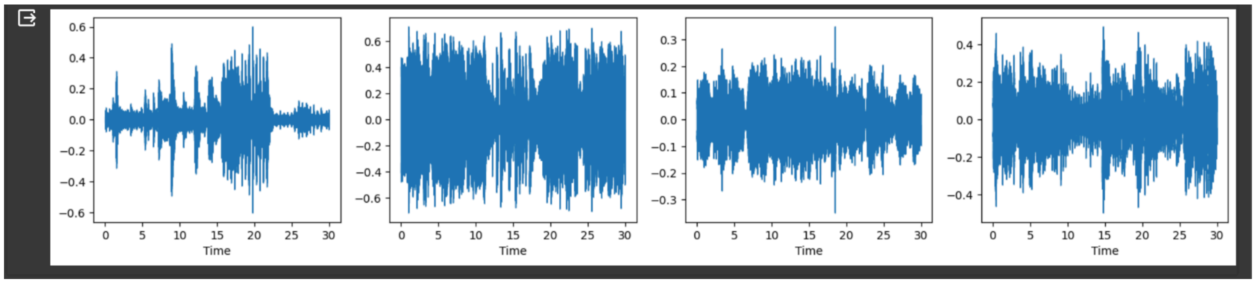
```python
sample = TrackSet_1[1]
sample_, sampling_rate = librosa.load(sample,sr=3000, offset=0.0, duration=30)
ipd.Audio(sample_,rate=3000)
```

This loads a specific audio file from TrackSet_1, processes it using the librosa library, and then plays it using IPython's audio player. It selects a file, loads it with a sampling rate of 3000 samples per second, and then creates an audio player for the loaded audio data, allowing for playback.



```
import librosa.display
plt.figure(figsize=(18,15))
for i in range(4):
    plt.subplot(4, 4, i + 1)
    j = load(TrackSet_1[i])
    librosa.display.waveshow(j[0], sr=3000)
```

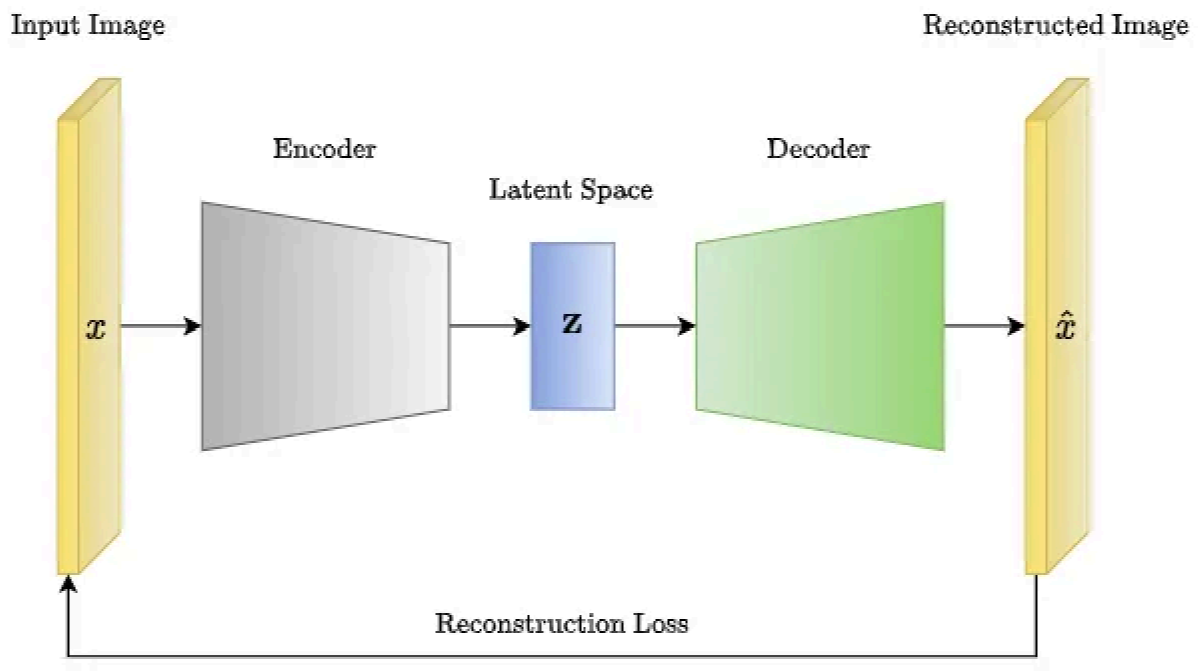
This utilizes Matplotlib and librosa libraries to plot the waveform of four audio samples from the TrackSet_1 dataset. It iterates through each sample, loads it using a predefined function, and then displays its waveform using librosa's waveshow function within Matplotlib subplots. This visualization provides a quick overview of the amplitude and shape of the audio signals, aiding in the analysis of the dataset's audio content.



Encoder and Decoder

The *encoder* consists of a series of neural network layers which extract features from an image and embed or encode them to a low-dimensional latent space. If the encoder consists of a series of Convolution Layers, the resulting architecture is sometimes called CNN-VAE. However, it has been seen that CNN-VAEs for image data outperform standard Multi-Layer Perceptron (MLP) VAEs. Hence, the two terms are used interchangeably. The architecture of the encoder is generally converging, as the latent space is lower-dimensional than the input space.

The *decoder* consists of a series of neural network layers which attempt to recreate the original image from the low-dimensional latent space. The architecture of the decoder is generally diverging. It is not necessary that the encoder and decoder have the same architecture, but in practice, they are usually kept the same. In the case of CNN-VAE, the decoder consists of Convolution Transpose layers which allows us to upsample the input.

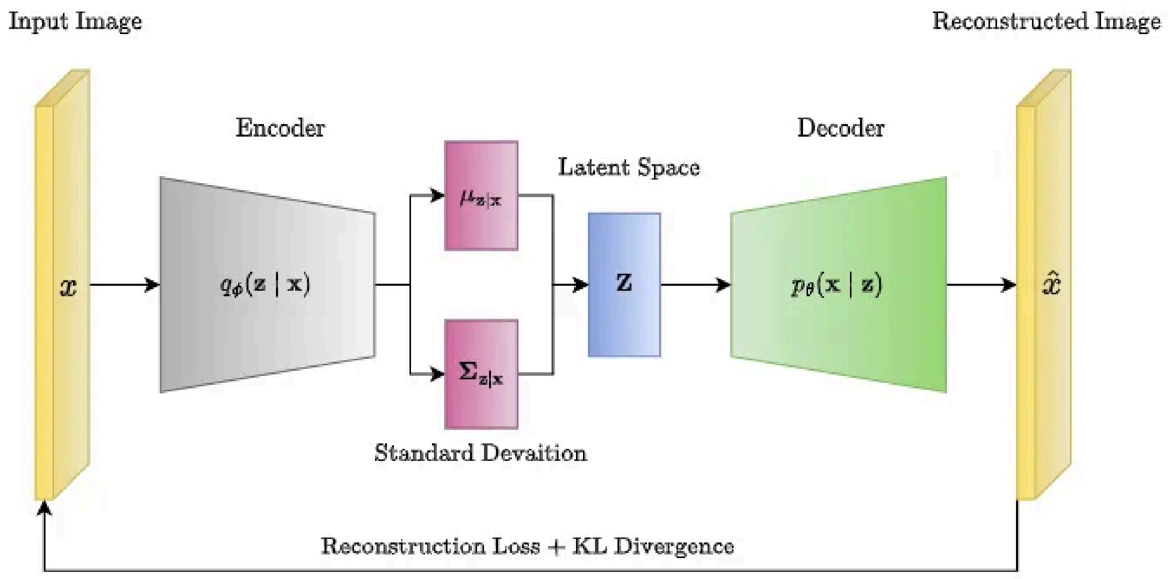


In the most basic form of an *autoencoder*, the encoder and decoder are typically composed of fully connected layers (MLP or CNN). The objective of training an autoencoder is to minimize the difference between the input data and its reconstructed output, which is typically measured using a loss function such as mean squared error or binary cross-entropy.

$$Loss_{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \hat{x_i}\right)^2$$

$$Loss_{BCE} = -\frac{1}{N}\sum_{n=1}^{N}\left[x_n \log \hat{x}_n + (1 - x_n)\log(1 - \hat{x}_n)\right]$$

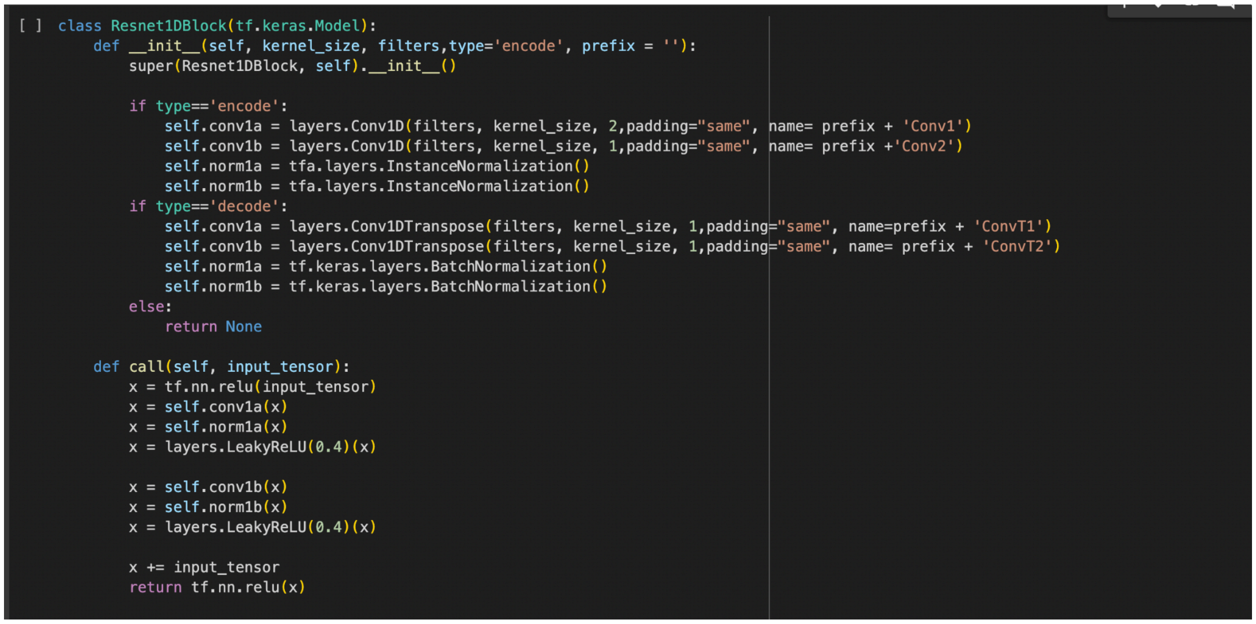
Autoencoders can be used for a variety of tasks, such as denoising, image super-resolution, anomaly detection, and clustering. They can also be stacked to create deeper architectures, such as deep autoencoders or convolutional autoencoders, that are capable of capturing more complex features and patterns in the input data.


Input Image
Reconstructed Image
Encoder
Decoder
Latent Space
x
$q_\phi(\mathbf{z} \mid \mathbf{x})$
$\mu_{\mathbf{z}|\mathbf{x}}$
$\Sigma_{\mathbf{z}|\mathbf{x}}$
$\mathbf{z}$
$p_\theta(\mathbf{x} \mid \mathbf{z})$
$\hat{x}$
Standard Devaition
Reconstruction Loss + KL Divergence

Variational Autoencoders (VAEs) are a type of autoencoder that was introduced to overcome some limitations of traditional AE. VAEs extend the traditional AE architecture by introducing a probabilistic framework for generating the compressed representation of the input data.

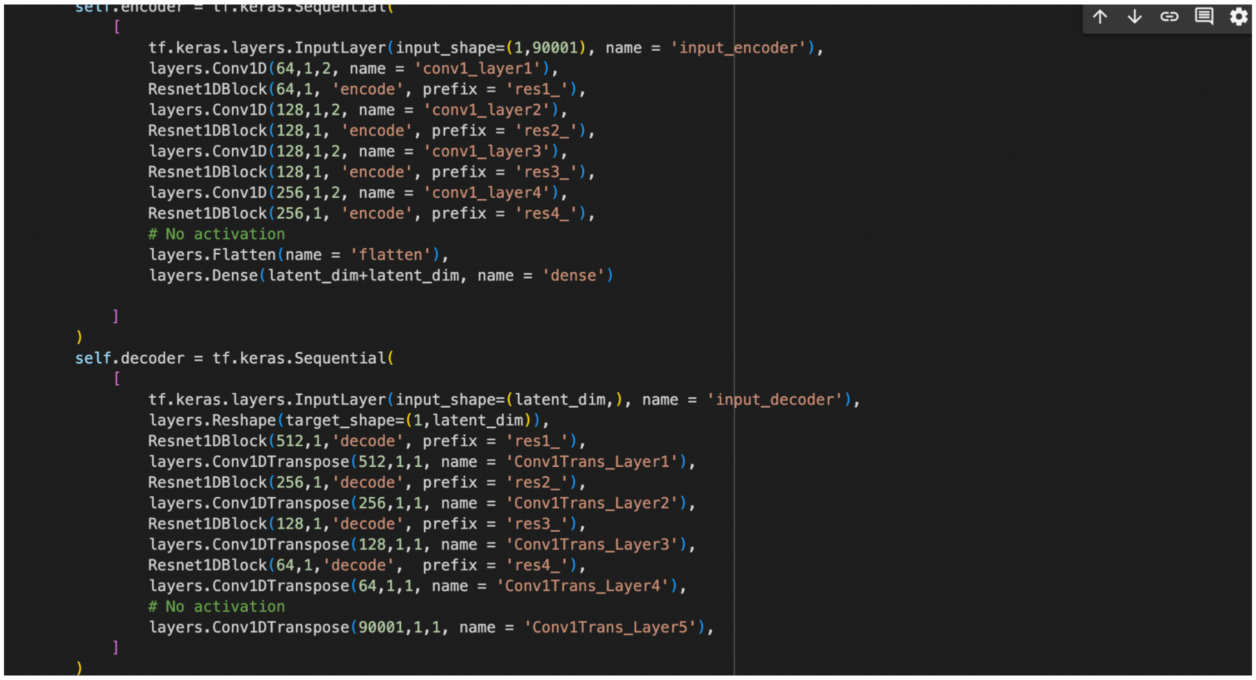
In VAEs, the encoder still maps the input data to a lower-dimensional latent space, but instead of a single point in the latent space, the encoder generates a probability distribution over the latent space. The decoder then samples from this distribution to generate a new data point. This probabilistic approach to encoding the input allows VAEs to learn a more structured and continuous latent space representation, which is useful for generative modeling and data synthesis.

Constructing the network



```
[ ] class Resnet1DBlock(tf.keras.Model):
        def __init__(self, kernel_size, filters,type='encode', prefix = ''):
            super(Resnet1DBlock, self).__init__()

            if type=='encode':
                self.conv1a = layers.Conv1D(filters, kernel_size, 2,padding="same", name= prefix + 'Conv1')
                self.conv1b = layers.Conv1D(filters, kernel_size, 1,padding="same", name= prefix +'Conv2')
                self.norm1a = tfa.layers.InstanceNormalization()
                self.norm1b = tfa.layers.InstanceNormalization()
            if type=='decode':
                self.conv1a = layers.Conv1DTranspose(filters, kernel_size, 1,padding="same", name=prefix + 'ConvT1')
                self.conv1b = layers.Conv1DTranspose(filters, kernel_size, 1,padding="same", name= prefix + 'ConvT2')
                self.norm1a = tf.keras.layers.BatchNormalization()
                self.norm1b = tf.keras.layers.BatchNormalization()
            else:
                return None

        def call(self, input_tensor):
            x = tf.nn.relu(input_tensor)
            x = self.conv1a(x)
            x = self.norm1a(x)
            x = layers.LeakyReLU(0.4)(x)

            x = self.conv1b(x)
            x = self.norm1b(x)
            x = layers.LeakyReLU(0.4)(x)

            x += input_tensor
            return tf.nn.relu(x)
```

This defines a custom TensorFlow Keras model class, Resnet1DBlock, representing a single block of a 1D Residual Neural Network (ResNet) for either encoding or decoding data. Depending on its purpose, it sets up convolutional or transposed convolutional layers followed by normalization layers. During the forward pass, the input passes through these layers and a LeakyReLU activation function, with a residual connection added before returning the output. This block architecture helps in learning complex features and patterns, commonly used in tasks like image and sequence processing.



```
self.encoder = tf.keras.Sequential(
    [
        tf.keras.layers.InputLayer(input_shape=(1,90001), name = 'input_encoder'),
        layers.Conv1D(64,1,2, name = 'conv1_layer1'),
        Resnet1DBlock(64,1, 'encode', prefix = 'res1_'),
        layers.Conv1D(128,1,2, name = 'conv1_layer2'),
        Resnet1DBlock(128,1, 'encode', prefix = 'res2_'),
        layers.Conv1D(128,1,2, name = 'conv1_layer3'),
        Resnet1DBlock(128,1, 'encode', prefix = 'res3_'),
        layers.Conv1D(256,1,2, name = 'conv1_layer4'),
        Resnet1DBlock(256,1, 'encode', prefix = 'res4_'),
        # No activation
        layers.Flatten(name = 'flatten'),
        layers.Dense(latent_dim+latent_dim, name = 'dense')

    ]
)
self.decoder = tf.keras.Sequential(
    [
        tf.keras.layers.InputLayer(input_shape=(latent_dim,), name = 'input_decoder'),
        layers.Reshape(target_shape=(1,latent_dim)),
        Resnet1DBlock(512,1,'decode', prefix = 'res1_'),
        layers.Conv1DTranspose(512,1,1, name = 'Conv1Trans_Layer1'),
        Resnet1DBlock(256,1,'decode', prefix = 'res2_'),
        layers.Conv1DTranspose(256,1,1, name = 'Conv1Trans_Layer2'),
        Resnet1DBlock(128,1,'decode', prefix = 'res3_'),
        layers.Conv1DTranspose(128,1,1, name = 'Conv1Trans_Layer3'),
        Resnet1DBlock(64,1,'decode',  prefix = 'res4_'),
        layers.Conv1DTranspose(64,1,1, name = 'Conv1Trans_Layer4'),
        # No activation
        layers.Conv1DTranspose(90001,1,1, name = 'Conv1Trans_Layer5'),
    ]
)
```

This implements a Convolutional Variational Autoencoder (CVAE) using TensorFlow and Keras. The model comprises an encoder and a decoder, each defined as a sequence of convolutional and transposed convolutional layers, respectively. The encoder compresses input data into a latent space representation, while the decoder reconstructs the original input from the latent representation. Additionally, the model includes methods for sampling from the latent space, encoding input data, and decoding latent representations. The use of TensorFlow @tf.function decorators optimizes the performance of these methods. Overall, this code provides a complete CVAE model capable of learning latent representations and generating new samples.

Convolution (Conv):

- Slides a set of filters over the input data, extracting features.
- Output shape depends on filter size, stride, and padding.
- Encoder in CVAE uses Conv layers for feature extraction.

Normalization/Denormalization:

- Normalization: Scales data to a common range (e.g., 0-1 or mean-variance normalization).

- In CVAE: Instance normalization helps the model focus on local patterns. Batch normalization helps with internal covariate shift during training.
- Denormalization: Reverses the scaling applied during normalization.
- CVAE doesn't explicitly denormalize; it uses activation functions.

Convolution Transpose (ConvT):

- Resamples a feature map, often used for image or sequence reconstruction.
- Output shape depends on filter size, stride, and padding.
- Decoder in CVAE uses ConvT layers for upsampling from latent space.

ReLU (Rectified Linear Unit):

- Activation function that sets negative values to zero and keeps positive values unchanged.
- Helps models learn non-linear relationships between features.
- Used in Resnet1DBlock to introduce non-linearity.

Leaky ReLU:

- Variation of ReLU that allows a small positive slope for negative values.
- Helps prevent the "dying ReLU" problem where gradients become zero, hindering learning.
- Used in Resnet1DBlock to improve gradient flow.

Sequential Network:

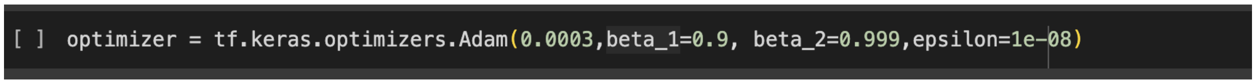
- Stacks layers sequentially, feeding the output of one layer as input to the next.
- Used to build both the encoder and decoder in CVAE.

Resnet1DBlock:

- Residual block architecture that helps with gradient flow in deep networks.
- Adds the original input to the processed output, promoting learning.
- Used in both encoder and decoder for feature extraction and reconstruction.

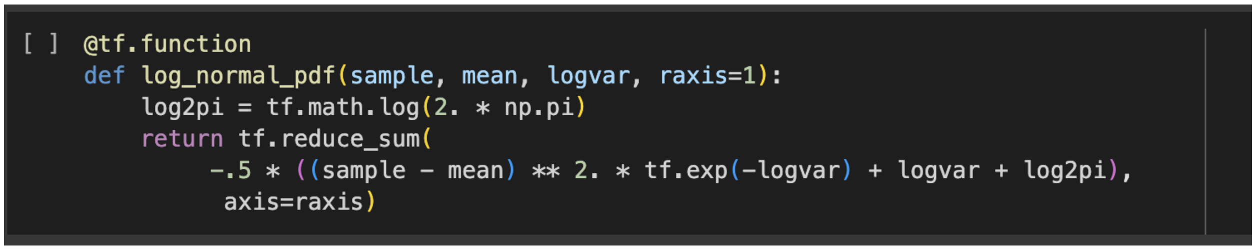
Epsilon (ε) as a Reparameterization Parameter:

- In VAEs, epsilon is a random noise vector added to the mean of the latent distribution.
- This reparameterization trick allows backpropagation through the sampling process.
- The reparameterize function in CVAE generates epsilon and combines it with the mean for sampling.



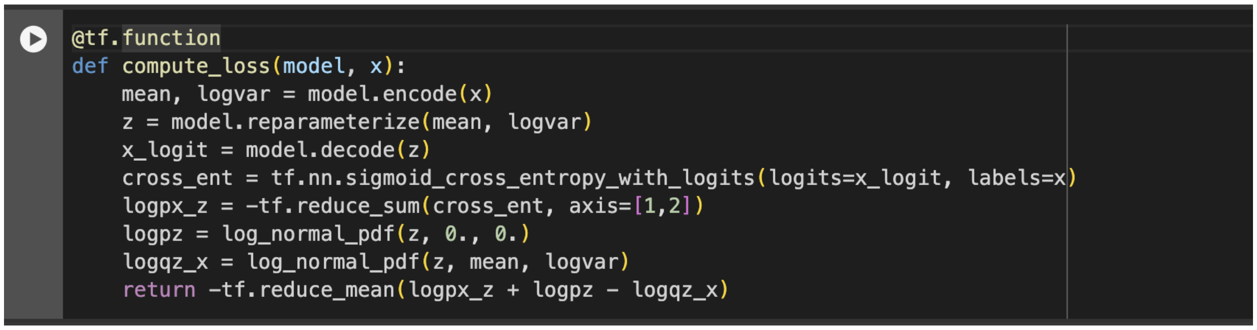
```
optimizer = tf.keras.optimizers.Adam(0.0003,beta_1=0.9, beta_2=0.999,epsilon=1e-08)
```

This initializes an Adam optimizer for training neural networks. The optimizer is configured with a learning rate of 0.0003, first moment decay rate (beta_1) of 0.9, second moment decay rate (beta_2) of 0.999, and epsilon value of 1e-08 for numerical stability. This optimizer is instrumental in iteratively adjusting the model's weights during training to minimize the loss function and enhance the model's performance.



```
@tf.function
def log_normal_pdf(sample, mean, logvar, raxis=1):
    log2pi = tf.math.log(2. * np.pi)
    return tf.reduce_sum(
        -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
         axis=raxis)
```

This TensorFlow function, log_normal_pdf, computes the logarithm of the probability density function (PDF) for a given sample under a multivariate normal distribution. It takes the sample, mean, and log variance as input parameters and returns the logarithm of the PDF. The function utilizes TensorFlow operations to efficiently compute the log PDF, making it suitable for probabilistic modeling tasks like variational inference and generative modeling.



```
@tf.function
def compute_loss(model, x):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logit = model.decode(z)
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1,2])
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)
    return -tf.reduce_mean(logpx_z + logpz - logqz_x)
```

This TensorFlow function, compute_loss, calculates the loss function for a Variational Autoencoder (VAE). It first encodes the input data to obtain the mean and log variance of the latent distribution, then samples latent representations using the reparameterization trick. The function reconstructs the input data from the latent representations and computes the cross-entropy loss between the reconstructed and original data. It also calculates the log probability of the latent representations under the prior and approximate posterior distributions. Finally, it combines these terms to form the total loss, which guides the VAE training process.

Loss function 1: The KL Divergence

The KL divergence tells us how well the probability distribution Q approximates the probability distribution P by calculating the cross-entropy minus the entropy. Intuitively, you can think of that as the statistical measure of how one distribution differs from another. In VAE, let X be the data we want to model, z be latent variable, P(X) be the probability distribution of data, P(z) be the probability distribution of the latent variable and P(X|z) be the distribution of generating data given latent variables.

In the case of variational autoencoders, our objective is to infer P(z) from P(z|X). P(z|X) is the probability distribution that projects our data into latent space. But since we do not have the distribution P(z|X), we estimate it using its simpler estimation Q.

Now while training our VAE, the encoder should try to learn the simpler distribution Q(z|X) such that it is as close as possible to the actual distribution P(z|X). This is where we use KL divergence as a measure of a difference between two probability distributions. The VAE objective function thus includes this KL divergence term that needs to be minimized.

$$D_{\mathrm{KL}}(p(x) \,||\, q(x)) = \sum_{x \in X} p(x) \ln \frac{p(x)}{q(x)}$$

Loss function 2: Reconstruction error

It measures the reconstruction of original input x. This network can be trained by minimizing the reconstruction error, which measures the differences between our original input and the consequent reconstruction.

Mean & Log Variance

In a Variational Autoencoder (VAE), the encode function outputs the latent distribution's mean and log variance rather than a single latent vector. This is an important feature of VAEs, since it relates to how they learn to produce data and execute latent space interpolations.

The reparameterisation trick

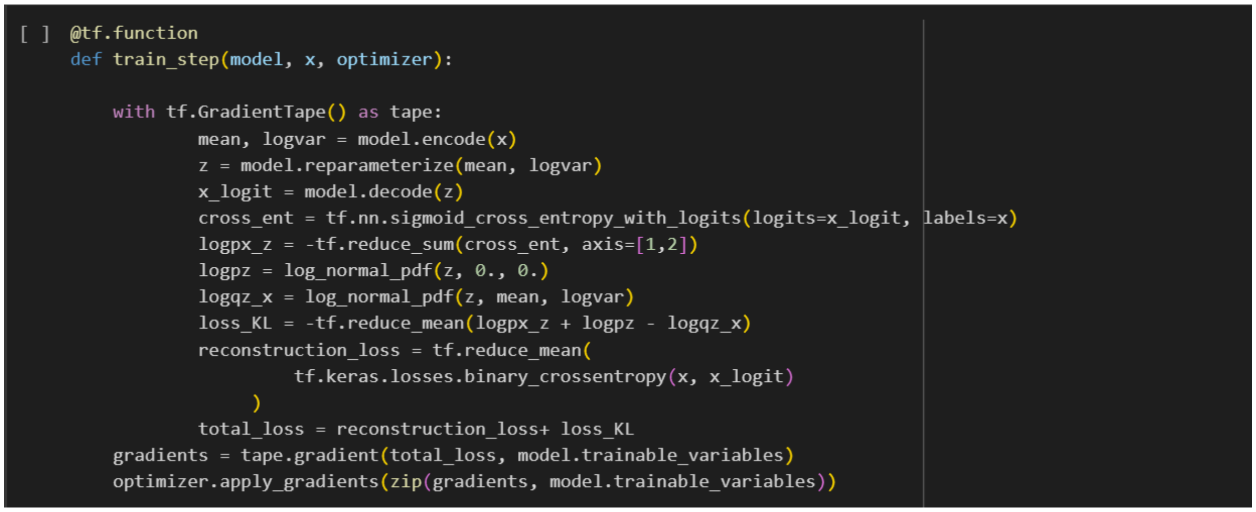
The reparameterization technique is used in Variational Autoencoders (VAEs) to enable efficient and accurate gradient estimation during the training process. The goal of training a VAE is to maximize the evidence lower bound (ELBO), which involves optimizing both the reconstruction loss and the KL divergence.

To generate a sample z for the decoder during training, we can sample from the latent distribution defined by the parameters outputted by the encoder, given an input observation x. However, this sampling operation creates a bottleneck because backpropagation cannot flow through a random node.

To address this, we use a reparameterization trick. In our example, we approximate z using the decoder parameters and another parameter ϵ as follows:

$$z = \mu + \sigma.\epsilon$$

Without the reparameterization trick, sampling from the latent distribution directly would not be differentiable, making it impossible to compute gradients with respect to the parameters of the distribution. By reparameterizing the sampling process, the randomness is moved outside of the network's parameters, allowing gradients to flow through the network and enabling efficient optimization.

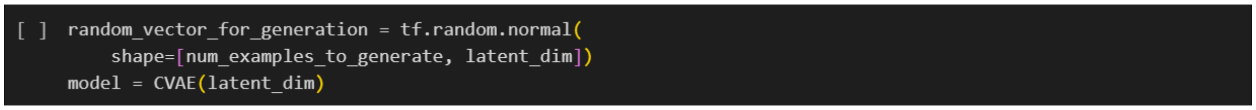


```
[ ] @tf.function
    def train_step(model, x, optimizer):

        with tf.GradientTape() as tape:
                mean, logvar = model.encode(x)
                z = model.reparameterize(mean, logvar)
                x_logit = model.decode(z)
                cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
                logpx_z = -tf.reduce_sum(cross_ent, axis=[1,2])
                logpz = log_normal_pdf(z, 0., 0.)
                logqz_x = log_normal_pdf(z, mean, logvar)
                loss_KL = -tf.reduce_mean(logpx_z + logpz - logqz_x)
                reconstruction_loss = tf.reduce_mean(
                         tf.keras.losses.binary_crossentropy(x, x_logit)
                     )
                total_loss = reconstruction_loss+ loss_KL
        gradients = tape.gradient(total_loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

tf.GradientTape() is a TensorFlow API for automatic differentiation, which is a key technique used in machine learning for optimizing models through techniques like backpropagation. When you perform operations inside a **tf.GradientTape()** context, TensorFlow keeps track of all operations that occur on tensors, allowing it to compute the gradients of a target tensor with respect to some source tensor.

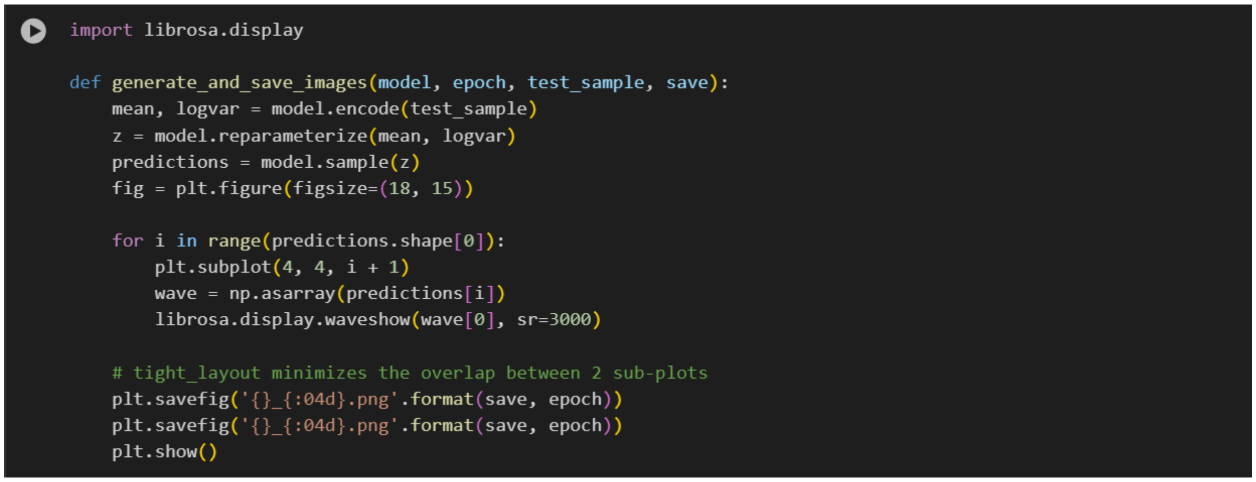
In the above snippet of code, within the Gradient Tape, we first *encode* the input data to a latent representation and then *reparameterize* the representation to sample 'z'. It then returns the *decoded* version of the latent vector as logit_x. Then the *binary cross-entropy* is found between x (the input) and x_logit (the reconstructed input). Then the respective log probabilities for calculating the KL divergence are computed and then the loss function itself is calculated using these previously returned probabilities.

Then, the reconstruction loss between x and logit_x, and the two losses are finally added to give the total loss. Gradient descent is then performed to optimise the parameters of the model.



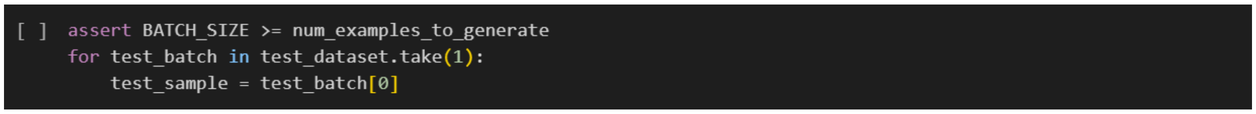
```
random_vector_for_generation = tf.random.normal(
    shape=[num_examples_to_generate, latent_dim])
model = CVAE(latent_dim)
```

Then we created a random vector to generate music. The vector follows a normal distribution and has a dimension of the number of latent dimensions specified at the beginning of the code.



```
import librosa.display

def generate_and_save_images(model, epoch, test_sample, save):
    mean, logvar = model.encode(test_sample)
    z = model.reparameterize(mean, logvar)
    predictions = model.sample(z)
    fig = plt.figure(figsize=(18, 15))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        wave = np.asarray(predictions[i])
        librosa.display.waveshow(wave[0], sr=3000)

    # tight_layout minimizes the overlap between 2 sub-plots
    plt.savefig('{}_{:04d}.png'.format(save, epoch))
    plt.savefig('{}_{:04d}.png'.format(save, epoch))
    plt.show()
```

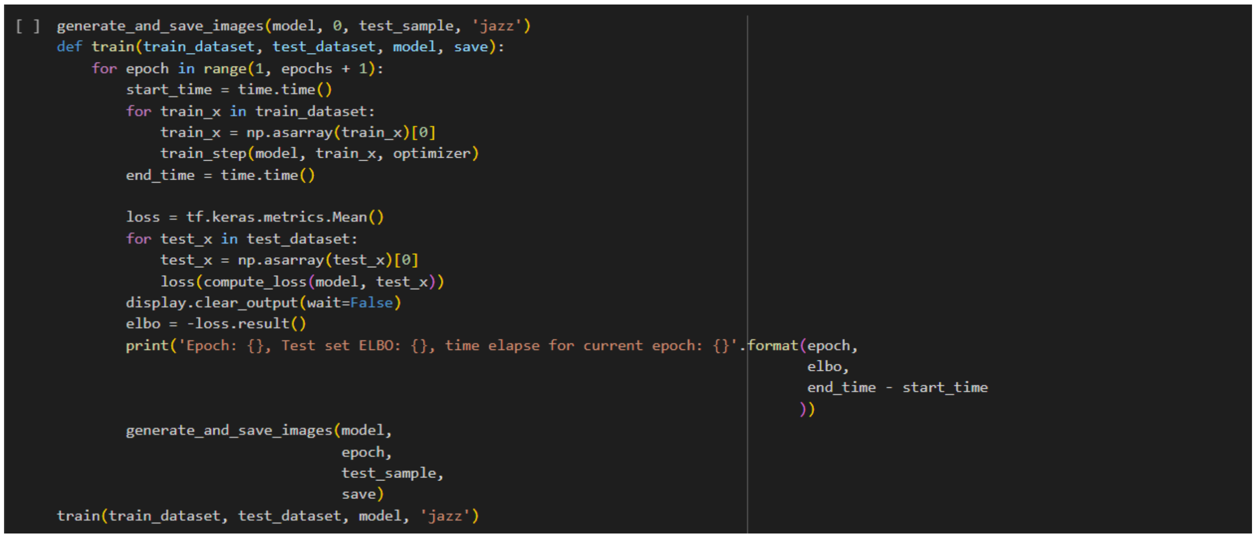
This is a pre-coded function that will be used to generate waveform images of the audio files generated using the VAE network.



```
assert BATCH_SIZE >= num_examples_to_generate
for test_batch in test_dataset.take(1):
    test_sample = test_batch[0]
```

An assertion is a statement that verifies the code's assumptions are true. If this condition isn't met, the program will raise an AssertionError, halting execution.

This ensures that we are generating a feasible number of examples within the batch size limit. It then takes the first sample of the first batch as a test sample.



```
[ ]  generate_and_save_images(model, 0, test_sample, 'jazz')
     def train(train_dataset, test_dataset, model, save):
         for epoch in range(1, epochs + 1):
             start_time = time.time()
             for train_x in train_dataset:
                 train_x = np.asarray(train_x)[0]
                 train_step(model, train_x, optimizer)
             end_time = time.time()

             loss = tf.keras.metrics.Mean()
             for test_x in test_dataset:
                 test_x = np.asarray(test_x)[0]
                 loss(compute_loss(model, test_x))
             display.clear_output(wait=False)
             elbo = -loss.result()
             print('Epoch: {}, Test set ELBO: {}, time elapse for current epoch: {}'.format(epoch,
                                                                                             elbo,
                                                                                             end_time - start_time
                                                                                            ))


             generate_and_save_images(model,
                                      epoch,
                                      test_sample,
                                      save)
     train(train_dataset, test_dataset, model, 'jazz')
```

The main training of the VAE happens in this code snippet. The loop runs through the number of epochs determined at the beginning of the code and for each data in the dataset, it executes the train_step() function explained earlier, which calculates the gradient of the loss function with respect to the parameters and therefore optimises them. The code then computes the mean loss for the test dataset and computes ELBO.

ELBO: Evidence Lower Bound

ELBO (Evidence Lower BOund) plays a crucial role in training Variational Autoencoders (VAEs). It helps the VAE learn a good representation of the data by balancing two competing objectives:

1. Reconstruction Accuracy:

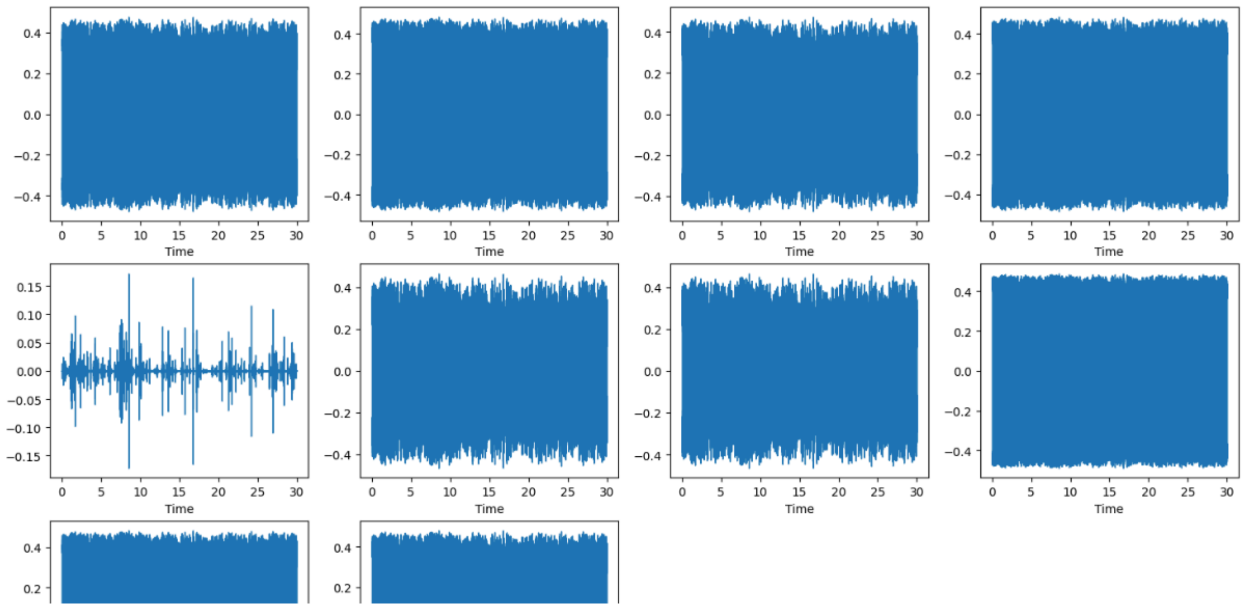
- VAEs aim to reconstruct the input data as accurately as possible. The ELBO includes a term that measures the difference between the original data and the reconstructed data generated by the VAE. This term encourages the VAE to learn a latent representation that captures the essential information for recreating the data.

2. Latent Space Regularization:

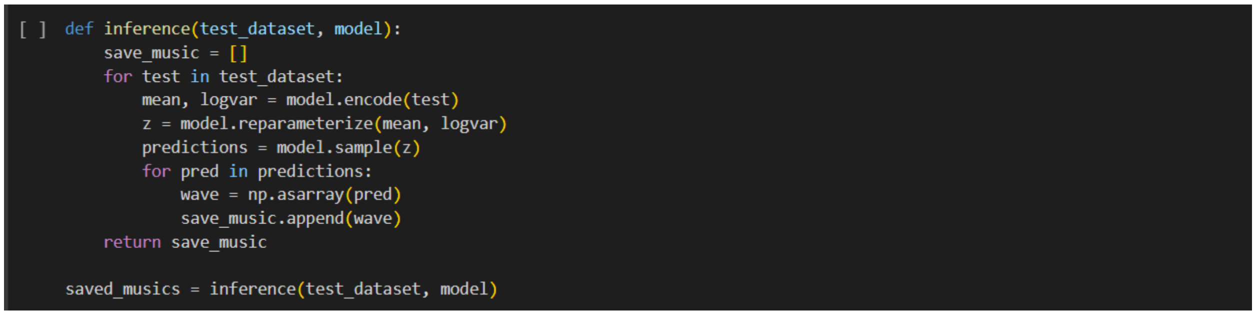
- VAEs also want to avoid overly complex or specific latent representations. The ELBO incorporates another term that measures the Kullback-Leibler (KL) divergence between the approximate posterior distribution (learned by the VAE) of the latent variable and a prior distribution (usually a simple Gaussian).

ELBO provides a lower bound on the log-likelihood of the data. By maximizing the ELBO, the VAE training process indirectly maximizes the likelihood of observing the data, which is essentially what we want the model to learn.

Running the above code generates the following output:

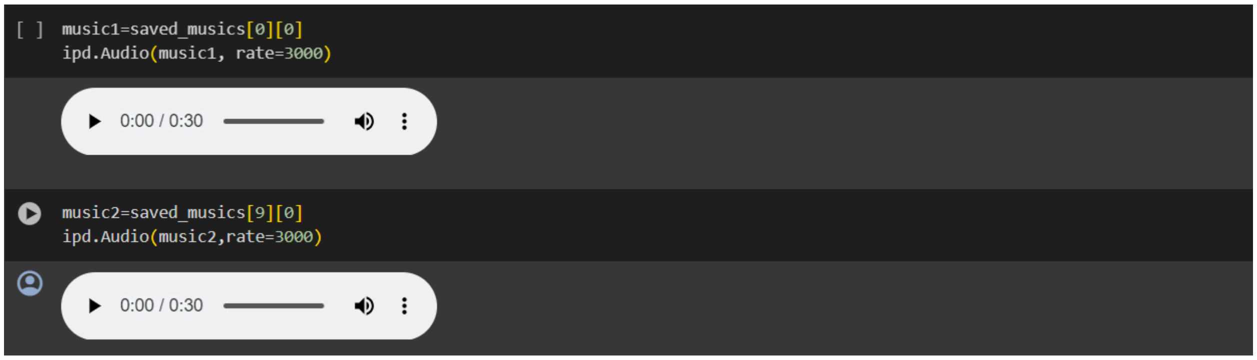


These are the pictorial representations of the waveforms generated for Jazz music by the VAE.



```
def inference(test_dataset, model):
    save_music = []
    for test in test_dataset:
        mean, logvar = model.encode(test)
        z = model.reparameterize(mean, logvar)
        predictions = model.sample(z)
        for pred in predictions:
            wave = np.asarray(pred)
            save_music.append(wave)
    return save_music

saved_musics = inference(test_dataset, model)
```

Now that the model is trained and optimised, we define a final inference() function that will be responsible for generating the music. The same steps are repeated one last time, where the input is encoded to provide the mean and log variance of the latent distribution, the distribution is parameterised to ‘z’, and then decoded to generate completely new music. This generated music is saved to an array.



```
music1=saved_musics[0][0]
ipd.Audio(music1, rate=3000)
```

```
music2=saved_musics[9][0]
ipd.Audio(music2,rate=3000)
```

The generated music can now be viewed by converting it to an audio format using the IPython.display library.

Comparison with other Generative models: GANs

VAEs are generally better suited for this project in regards to genrewise music generation due to the following reasons:

- VAEs inherently model the probability distribution of the data. In the context of music, this means that VAEs can generate diverse outputs by sampling from the learned distribution, resulting in more varied and exploratory compositions.

- VAEs explicitly learn a low-dimensional latent space representation of the data. In music, this latent space can capture meaningful features like rhythm, melody, and harmony, making it easier for users to manipulate and control the generation process.
- VAEs are trained to reconstruct their input data, which means they can generate outputs that are similar to the training data. This is beneficial for generating music that adheres closely to specific styles or genres.

Conclusion

Using VAEs, we were successfully able to generate jazz music using the dataset provided. However, it must be noted that the quality of generated media in VAEs are much lower than that of GANs since VAEs are more concerned with reconstructing the data, whereas GANs have an objective of essentially ‘fooling’ the discriminator network and guarantee hyperrealism. This network is only suitable to generate instrumental music, and other, more complex algorithms can be used to generate complex interleavings of notes and even audio of singers’ vocals. Therefore, for tasks where reconstruction is a priority, VAEs prove to be a solid contender among the plethora of network structures present in Deep Learning.

Inferences

- This particular VAE prioritizes reconstructing the data over generating hyperrealistic outputs. Therefore it may lack the finer details and realism achieved by other methods such as GANs.
- The music generated was purely instrumental, and may require more detailed datasets to learn vocals and background harmonies.
- It is limited in its capabilities to generate complex interleavings of notes and vocal audio.
- This model can only be used if reconstruction is a priority.
- VAEs may not be the ultimate solution for all music generation tasks.