AUTONOMOUS AUDIO GENERATION WITH VARIATIONAL AUTOENCODERS USING THE FSDD DATASET

TECHNICAL REPORT

Mónica A. Ayala Marrero

TC3002B: Desarrollo de aplicaciones avanzadas de ciencias computacionales
Monterrey Institute of Technology and Higher Education
Queretaro, Mexico.
a01707439@tec.mx

May 1, 2024

ABSTRACT

This project explores autonomous audio generation using Variational Autoencoders (VAEs) applied to the Free Spoken Digit Dataset (FSDD), an auditory equivalent of the MNIST dataset for handwritten digits. Dimensionality reduction techniques were integral in preprocessing the audio data, including the implementation of harmonic representations, adjustments of frame times and hop lengths in the Short-Time Fourier Transform (STFT), trimming of audio, and resampling to 8 kHz from 44 kHz. These methods significantly enhanced the efficiency and effectiveness of the model training.

This report compares two types of VAEs: a standard VAE employing a mean squared error reconstruction loss and a Gaussian mixture VAE. The methodology consists of transforming .wav audio files into STFT spectrograms which are then used to train the models, Initially, the project began with a dataset comprising Taylor Swift's discography; however, due to the complex nature of the audio data, it was pivoted to the simpler FSDD. This adjustment allowed for more controlled generation and reconstruction of audio, facilitating the correct generation of sounds corresponding to each digit. Additionally, the latent space of the VAEs was effectively mapped using both PCA and t-SNE, demonstrating the models' ability to capture meaningful audio representations. The results underscore the potential of VAEs in the field of audio processing and generation, presenting a novel approach to understanding and creating digit-based audio data.

Keywords Variational Autoencoder · Short-Time Fourier Transform · Deep Leanring · Generative models · Autonomous audio generation · Spoken digits

1 Introduction

The digital synthesis and manipulation of audio have profound implications across various fields, from music production to automated speech recognition. Recent advances in deep learning have further broadened these horizons through the development of generative models capable of producing high-quality audio. Among these, Variational Autoencoders (VAEs) have emerged as a particularly promising approach due to their ability to learn complex probability distributions of audio data and generate new samples that are perceptually similar to the original inputs.

This paper focuses on applying VAEs to the Free Spoken Digit Dataset (FSDD), an auditory equivalent to the well-known MNIST dataset for handwritten digits. The FSDD offers a controlled environment to explore and benchmark the capabilities of VAEs in an audio context. By employing dimensionality reduction techniques, such as adjustments in the Short-Time Fourier Transform (STFT) parameters and audio resampling, this study aims to enhance the efficiency of VAEs in processing and generating spoken digit sounds.

2 About the dataset

The Free Spoken Digit Dataset (FSDD) is a straightforward and accessible audio/speech dataset that features recordings of spoken digits in .wav format, sampled at 8kHz. Designed for ease of use and minimal preprocessing, these recordings are meticulously trimmed to ensure there is minimal silence at the beginnings and ends of the files, facilitating various audio processing tasks. The dataset is available in `https://github.com/Jakobovski/free-spoken-digit-dataset/`

- Speakers: 6
- Recordings: 3,000 (50 of each digit per speaker)
- Pronunciations: English

The files within FSDD are systematically named following a specific format: {digitLabel}_{speakerName}_{index}.wav. For example, the file name '7_jackson_32.wav' represents the thirty-second recording of the digit 'seven' by the speaker named Jackson. FSDD is an open dataset, which signifies its ongoing growth as new data contributions are made, the version used can be found in `https://zenodo.org/records/1342401` and is identified as version 1.0.8. All six speakers are male and have different accents (French, German, American, Greek and Belgian).

2.1 Obtaining new data for the dataset

To obtain new recordings and more data contributors can record the spoken digits in intervals, ensuring each number is clearly pronounced. This recording should then be processed to split the continuous audio into individual files for each digit, adhering to the dataset's naming convention and format.

For testing new data was generated using Audacity and changing the settings to 8kHz. Scripts to facilitate this are provided in the github repository of the project `https://github.com/monica-ayala/AudioGeneration_MNIST`. After obtaining the new recordings the magnitude was amplified using librosa, a python library, to ensure that the audio could be clearly heard.

3 Audio representation

Audio representation is fundamental in processing and analyzing audio signals effectively. Various techniques have been developed to transform raw audio data into formats that are more manageable and informative for specific applications. For this project we had to explore different audio representations in order to chose the one that best fit our model, some of them are: Short-Time Fourier Transform (STFT) spectrograms, Mel-spectrograms, Mel Frequency Cepstral Coefficients (MFCCs), and MIDI files.

3.1 Short-Time Fourier Transform (STFT) Spectrograms

STFT divides a longer time signal into shorter segments of equal length and computes the Fourier transform separately on each segment. This approach captures both the temporal and frequency information, which is crucial for many audio analysis tasks. While STFT provide a detailed representation of the signal's frequency and time information, essential for detecting changes in sound properties over time, the resolution is limited by the size of the window. This creates a compromise between time and frequency resolution—larger windows provide better frequency resolution but worse time resolution, and vice versa.

They are also computational intensive when having large sample rates and window sizes of the fourier transform or small number of samples between succesive frames, as the time dimension and frequency dimension depend on these parameters but create another compromise between quality audio and computational time.

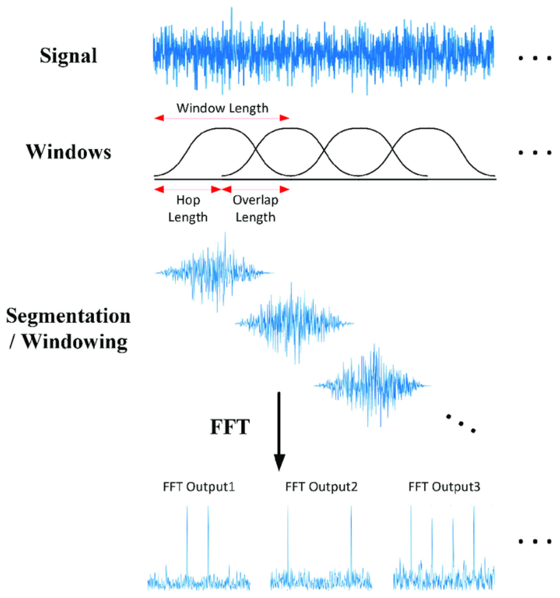

Figure 1: Computation of a Short-Time Fourier Transfrom Spectrogram

3.2 Mel-Spectrograms

This technique involves converting the frequency scale of STFT spectrograms from the linear scale to the Mel scale, which is designed to mimic the human ear's response more closely. While they are better at representing sound the way humans perceive it, making it particularly useful in speech and music recognition tasks, for audio generation they are tricky to work with. The Mel scale conversion losses some of the key data and if passing from spectrogram representation into audio is hard, mel-spectrograms are harder to re-transform into .wav files and result in lower-sounding audio.

3.3 MIDI Files

MIDI files represent music in terms of digital instructions for music performance, including notes, timing, instrument changes, and other elements of musical expression. They are extremely efficient in terms of storage since they only contain instructions on how to generate sounds, not the sound recordings themselves. Highly versatile for music synthesis. However, MIDI lacks the ability to capture the acoustic nuances of a live performance fully. It is dependent on the synthesis quality of playback devices and may result in synthetic-sounding audio. In the begining when the project was about music generation MIDI files were considered, but as the focus shifted to the FSDD they were discarded.

Short-Time Fourier Transform were the chosen representation for our audio files. Python's librosa library was key in computing the spectrograms and also later transforming them back into audio versions.

4 Preprocessing and Dimensionality Reduction

After selecting the Short-Time Fourier Transform (STFT) Spectrograms as the representation method, a critical step involves adjusting parameters such as n_fft (number of FFT components, the window size of the Fourier transform) and hop_length (the number of samples between successive frames). These parameters are essential for managing the dimensionality of the data, affecting both the time and frequency dimensions.

$$n_{samples} = (duration * sample\_rate)/hop\_lenght + 1 \tag{1}$$

$$f_{bins} = (n\_fft)/2 + 1 \tag{2}$$

The choice of n_fft and hop_length directly influences the resolution and size of the resulting spectrogram, allowing us to balance between temporal resolution and computational efficiency. A larger n_fft can provide a finer frequency resolution but increases the spectrogram's size, whereas a smaller hop_length improves the time resolution by providing more overlapping windows at the expense of increased data redundancy and computational load.

In the end a window frame of 512 and a hop_length of 256 was chosen. Passing these parameters to librosa's method stft allows us to compute the short-time fourier transform and gives us in return an 2D array with complex numbers.

```
stft = librosa.stft(audio, n_fft=512, hop_length=256)
```

During the STFT process, both magnitude and phase information are produced (the real and imaginary part). However, for many applications, only the magnitude is retained, reducing the data to a real-valued representation, which simplifies the model architecture and input features. This omission of phase information simplifies the dataset but poses a significant challenge in audio reconstruction quality, as phase carries crucial information for signal fidelity. We chose to only keep the magnitude information and to transform the amplitude into decibel notation.

To ensure uniformity and facilitate batch processing in neural networks or other algorithms, audio signals are often padded to a consistent length. Padding is typically performed on the right-end of the signal to maintain original content alignment:

```
def padding(audio, num_expected_samples):
    num_missing_samples = num_expected_samples - len(audio)
    return np.pad(audio, (0, num_missing_samples), mode='constant')
```

Normalization of spectrograms is the final step, which involves scaling the amplitude to a common range, enhancing model stability and convergence during training:

```
def normalize_spectrogram(spectrogram, min_val, max_val):
    return (spectrogram - min_val) / (max_val - min_val)
```

In our case the max and min values will be stored in a dictionary for the future reconstruction of the audio.

4.1 Audio reconstruction from spectrogram

To start the reconstruction we need to drop any aditional dimensions added for the benefit of our model (that often has a 3D shape as input) and denormalize the values our model generated with either stored min and max values or with the standard minimum and maximum values for decibels (-80, 0 dB). After denormalization, we transform the decibel magnitude into amplitude using librosa.

The absence of phase information necessitates employing algorithms like Griffin-Lim for phase approximation during the inverse STFT, attempting to reconstruct the original audio signal from the magnitude-only spectrogram. While effective, Griffin-Lim and similar algorithms often introduce artifacts or result in a loss of clarity and detail, underscoring the trade-off between reduced data complexity and audio quality.

```
def reconstruct_audio(stft, output_path, min_val, max_val):
    stft = stft[:, :, 0]
    magnitude_db = inverse_normalize_spectrogram(stft, min_val, max_val)
    magnitude_linear = librosa.db_to_amplitude(magnitude_db)
    y_reconstructed = librosa.istft(magnitude_linear, hop_length=256)

    current_max = np.max(np.abs(y_reconstructed))
    desired_max = 0.9
    if current_max == 0:
        amplification_factor = 1
    else:
        amplification_factor = desired_max / current_max
    amplified_audio = y_reconstructed * amplification_factor
    amplified_audio = np.clip(amplified_audio, -1, 1)
```

The code above is the one used to reconstruct the audio from the information returned by the model, after reconstruction the signal is amplified as many of the database recordings had very small amplitudes and couldn't be easily heard.

5 Variational Autoencoders

A Variational Autoencoder (VAE) is a type of generative model that leverages the principles of deep learning to encode input data into a latent space distribution, unlike traditional autoencoders that compress input into a fixed point. VAEs are characterized by their ability to generate new data points that maintain a similar yet diverse nature compared to the original data. This capability arises from the model's structure, which consists of an encoder that maps input data to a distribution in latent space and a decoder that reconstructs data from sampled points in this space.

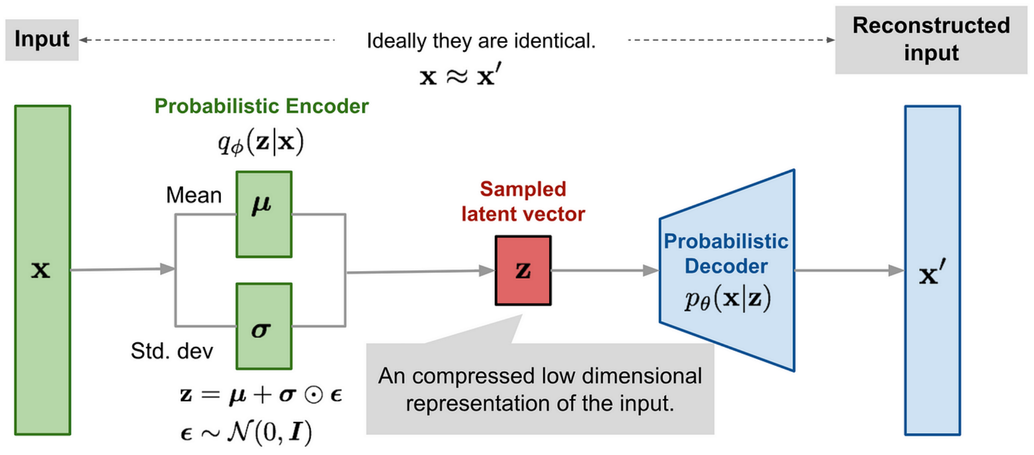
Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
Mean
$\boldsymbol{\mu}$
$\mathbf{x}$
$\boldsymbol{\sigma}$
Std. dev
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

Figure 2: Representation of a variational autoencoder

Training Process:

- **Encoding:** Input data is encoded as a distribution over latent space.
- **Sampling:** A point is sampled from this distribution.
- **Decoding:** The sampled point is decoded, and the reconstruction error is calculated.
- **Backpropagation:** The reconstruction error is backpropagated to train the model.

5.1 The mathematical principles behind VAEs

In a VAE, instead of encoding an input as a fixed point, the encoder models z as a random variable drawn from a distribution $q(z \mid x)$, which is approximated as a Gaussian $\mathcal{N}(\mu, \sigma^2\mathbf{I})$. The decoder then generates data by sampling from $p(x \mid z)$, also typically assumed to be Gaussian, especially for continuous data.

The core of a VAE is variational inference, where the goal is to approximate the true but intractable posterior distribution $p(z \mid x)$ with a variational distribution $q(z \mid x)$. This approximation is facilitated by minimizing the Kullback-Leibler (KL) divergence between $q(z \mid x)$ and $p(z \mid x)$, which quantifies how one probability distribution diverges from a second, expected probability distribution.

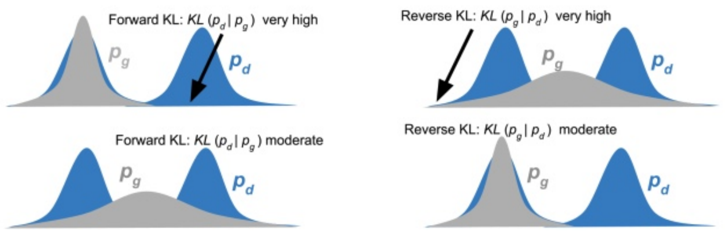

Figure 3: Kullback-Leiber Divergence (Reverse KL is used)

5.1.1 Maximization of ELBO

The objective function in VAEs is the maximization of the Evidence Lower Bound (ELBO) on the marginal likelihood of the observed data:

$$\text{ELBO} = \mathbb{E}_{q(z|x)}[\log p(x \mid z)] - D_{KL}(q(z \mid x) \| p(z))$$

Where:

- $\mathbb{E}_{q(z|x)}[\log p(x \mid z)]$ is the expected log-likelihood of the data, emphasizing the reconstruction accuracy.
- $D_{KL}(q(z \mid x) \| p(z))$ is the KL divergence between the learned latent distribution $q(z \mid x)$ and the prior latent distribution $p(z)$, used as a regularizer.

However, we can also use alternative ways of representing the log-likelihood or reconstruction loss of the data. That's why two specific implementations of VAEs, namely Standard VAE and Gaussian Mixture VAEs, were explored in this project.

5.2 Standard VAEs

The Standard VAE involves mapping the inputs through an encoder to determine the mean μ and log variance $\log \sigma^2$ which describe a normal distribution in the latent space. A point is then sampled from this distribution using the reparameterization trick to ensure that the sampling process is differentiable:

```
def sample_point_from_normal_distribution(args):
    mu, log_variance = args
    epsilon = K.random_normal(shape=K.shape(mu), mean=0., stddev=1.)
    return mu + K.exp(log_variance / 2) * epsilon
\subsection{Gaussian Mixture VAEs}
```

However, the biggest change lies in the reconstruction loss function. When the input data x and its reconstruction $\hat{x}$ (output of the decoder) are assumed to be normally distributed with a constant variance, the MSE effectively becomes a scaled version of the negative log-likelihood of a Gaussian distribution.

This relationship is underpinned by modeling $p(x \mid z)$, the probability of x given the latent variable z, as a Gaussian distribution $\mathcal{N}(\hat{x}, \sigma^2 \mathbf{I})$. Here, σ^2 represents the variance of the Gaussian noise, which, if assumed to be 1, aligns the MSE directly with the negative log-likelihood:

$$-\log p(x \mid z) \propto \frac{1}{2\sigma^2} \sum_i (x_i - \hat{x}_i)^2$$

The standard VAE model uses the MSE directly as a surrogate for the reconstruction term in the Evidence Lower Bound (ELBO). This adaptation simplifies the loss function and offers a more intuitive approach for evaluating the fidelity of the reconstructed audio against the original. The ELBO for our VAE model, considering a simplification where σ^2 is assumed to be 1, is formulated as:

$$\text{ELBO} = -\text{MSE}(x, \hat{x}) - D_{KL}(q(z \mid x) \| p(z))$$

This is not only more intuitive but it also aligns well with our auditory data, where direct comparison between actual and predicted values significantly impacts the quality of the generated output. The subsequent optimization phase is also easier, as the modification of weights between the Kullback-Leiber divergence and the reconstruction loss is also more simple than with the log-likelihood.

5.3 Gaussian Mixture VAEs

Gaussian Mixture VAEs differ primarily in their use of a more complex prior for the latent space, a mixture of Gaussian distributions. This variation aims to capture more intricate relationships and variations within the data. The initial project aiming to generate music based on Taylor Swift's discography used a Gaussian Miture VAE based on the implementation of a GM-VAE for the CelebA dataset found here `https://colab.research.google.com/github/goodboychan/goodboychan.github.io/blob/main/_notebooks/2021-09-14-03-Variational-AutoEncoder-Celeb-A.ipynb`

Prior Distribution: Defined using TensorFlow Probability, the prior is a mixture of Gaussians rather than a single Gaussian distribution. This provides a richer and more flexible representation of latent space.

Instead of MSE, the reconstruction loss in a Gaussian Mixture VAE is calculated as the negative log probability of the decoded distribution given the input as explained in section 5.1.1.

```
def reconstruction_loss(batch_of_images, decoding_dist):
    return -tf.reduce_sum(decoding_dist.log_prob(batch_of_images), axis=0)
```

The inclusion of advanced probabilistic models like Gaussian Mixtures in VAEs enhances the model's ability to capture and recreate complex audio phenomena but introduces computational and mathematical complexity. This is why after changing of dataset into the ligther and more standarized FSDD, the Standard VAE was preferred over the GM-VAE.

6 Model Implementation of Standard VAE

In Figure 4 we find the final architecture of our standard variaitonal autoencoder for audio generation, the latent space dimension being reduced into 128 neurons.

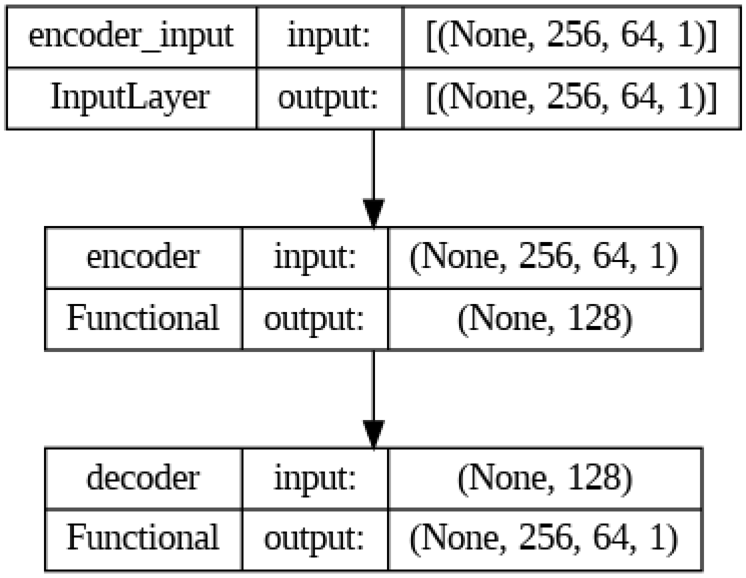

Figure 4: VAE Model

The input shape was able to be reduced into (256, 64, 1) and with the use of the harmonic series it could still be reduced more into a (64, 64, 1) shape. However, our model was able to handle this size of input without much loss to the reconstruction of the audio. The encoder and decoder architectures are the following:

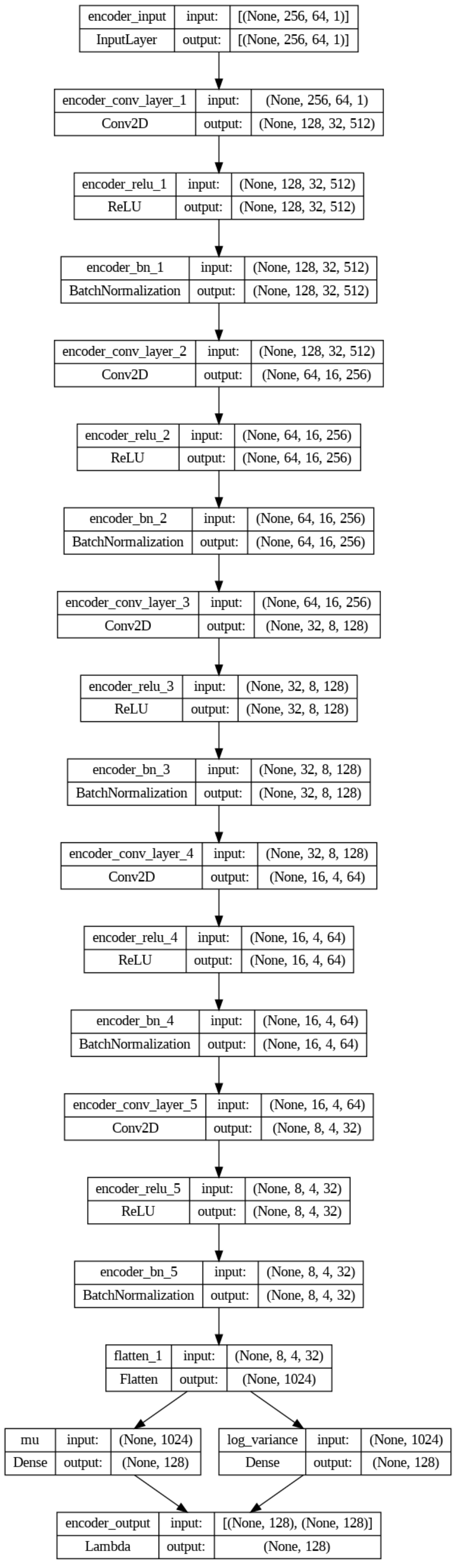
encoder_input | input: | [(None, 256, 64, 1)]
InputLayer | output: | [(None, 256, 64, 1)]
encoder_conv_layer_1 | input: | (None, 256, 64, 1)
Conv2D | output: | (None, 128, 32, 512)
encoder_relu_1 | input: | (None, 128, 32, 512)
ReLU | output: | (None, 128, 32, 512)
encoder_bn_1 | input: | (None, 128, 32, 512)
BatchNormalization | output: | (None, 128, 32, 512)
encoder_conv_layer_2 | input: | (None, 128, 32, 512)
Conv2D | output: | (None, 64, 16, 256)
encoder_relu_2 | input: | (None, 64, 16, 256)
ReLU | output: | (None, 64, 16, 256)
encoder_bn_2 | input: | (None, 64, 16, 256)
BatchNormalization | output: | (None, 64, 16, 256)
encoder_conv_layer_3 | input: | (None, 64, 16, 256)
Conv2D | output: | (None, 32, 8, 128)
encoder_relu_3 | input: | (None, 32, 8, 128)
ReLU | output: | (None, 32, 8, 128)
encoder_bn_3 | input: | (None, 32, 8, 128)
BatchNormalization | output: | (None, 32, 8, 128)
encoder_conv_layer_4 | input: | (None, 32, 8, 128)
Conv2D | output: | (None, 16, 4, 64)
encoder_relu_4 | input: | (None, 16, 4, 64)
ReLU | output: | (None, 16, 4, 64)
encoder_bn_4 | input: | (None, 16, 4, 64)
BatchNormalization | output: | (None, 16, 4, 64)
encoder_conv_layer_5 | input: | (None, 16, 4, 64)
Conv2D | output: | (None, 8, 4, 32)
encoder_relu_5 | input: | (None, 8, 4, 32)
ReLU | output: | (None, 8, 4, 32)
encoder_bn_5 | input: | (None, 8, 4, 32)
BatchNormalization | output: | (None, 8, 4, 32)
flatten_1 | input: | (None, 8, 4, 32)
Flatten | output: | (None, 1024)
mu | input: | (None, 1024)
Dense | output: | (None, 128)
log_variance | input: | (None, 1024)
Dense | output: | (None, 128)
encoder_output | input: | [(None, 128), (None, 128)]
Lambda | output: | (None, 128)

Figure 5: Encoder Architecture

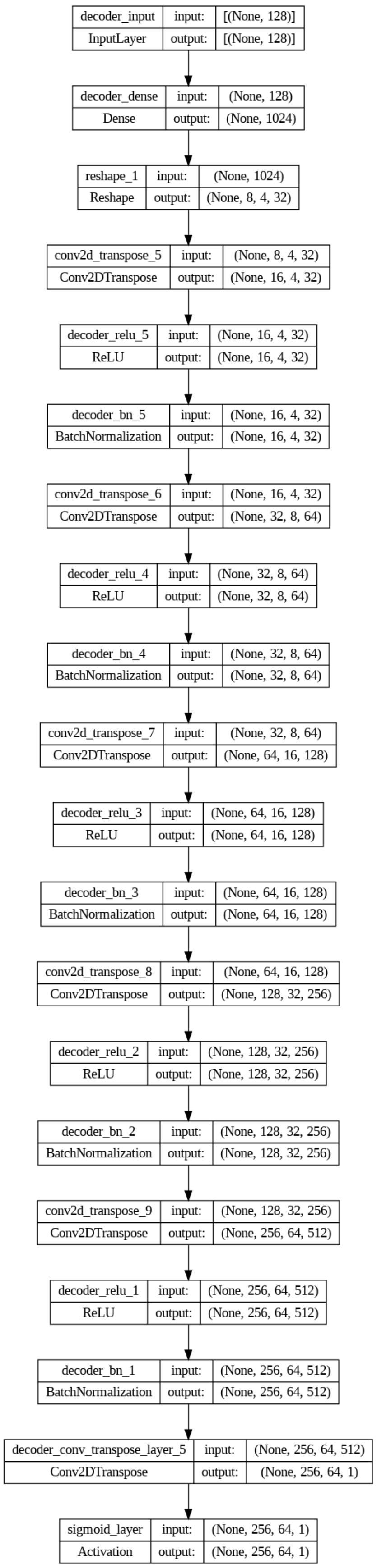
decoder_input | input: | [(None, 128)]
InputLayer | output: | [(None, 128)]
decoder_dense | input: | (None, 128)
Dense | output: | (None, 1024)
reshape_1 | input: | (None, 1024)
Reshape | output: | (None, 8, 4, 32)
conv2d_transpose_5 | input: | (None, 8, 4, 32)
Conv2DTranspose | output: | (None, 16, 4, 32)
decoder_relu_5 | input: | (None, 16, 4, 32)
ReLU | output: | (None, 16, 4, 32)
decoder_bn_5 | input: | (None, 16, 4, 32)
BatchNormalization | output: | (None, 16, 4, 32)
conv2d_transpose_6 | input: | (None, 16, 4, 32)
Conv2DTranspose | output: | (None, 32, 8, 64)
decoder_relu_4 | input: | (None, 32, 8, 64)
ReLU | output: | (None, 32, 8, 64)
decoder_bn_4 | input: | (None, 32, 8, 64)
BatchNormalization | output: | (None, 32, 8, 64)
conv2d_transpose_7 | input: | (None, 32, 8, 64)
Conv2DTranspose | output: | (None, 64, 16, 128)
decoder_relu_3 | input: | (None, 64, 16, 128)
ReLU | output: | (None, 64, 16, 128)
decoder_bn_3 | input: | (None, 64, 16, 128)
BatchNormalization | output: | (None, 64, 16, 128)
conv2d_transpose_8 | input: | (None, 64, 16, 128)
Conv2DTranspose | output: | (None, 128, 32, 256)
decoder_relu_2 | input: | (None, 128, 32, 256)
ReLU | output: | (None, 128, 32, 256)
decoder_bn_2 | input: | (None, 128, 32, 256)
BatchNormalization | output: | (None, 128, 32, 256)
conv2d_transpose_9 | input: | (None, 128, 32, 256)
Conv2DTranspose | output: | (None, 256, 64, 512)
decoder_relu_1 | input: | (None, 256, 64, 512)
ReLU | output: | (None, 256, 64, 512)
decoder_bn_1 | input: | (None, 256, 64, 512)
BatchNormalization | output: | (None, 256, 64, 512)
decoder_conv_transpose_layer_5 | input: | (None, 256, 64, 512)
Conv2DTranspose | output: | (None, 256, 64, 1)
sigmoid_layer | input: | (None, 256, 64, 1)
Activation | output: | (None, 256, 64, 1)

Figure 6: Decoder Architecture

7 Fine-tunning VAE models

Fine-tuning Variational Autoencoders (VAEs) involves adjusting several hyperparameters to optimize the model's performance for specific tasks. Key among these is the balance between the reconstruction loss and the KL divergence term, which significantly impacts the model's behavior and output quality.

In VAEs, the total loss function is a combination of reconstruction loss and KL divergence, which measures the difference between the learned latent distribution and the prior distribution. The weight β applied to the KL divergence term plays a crucial role in this balance:

- High β values increase the weight of the KL term, encouraging the latent space to closely follow the prior distribution, which can lead to less overfitting but potentially poorer reconstruction (underfitting).
- Low β values decrease the influence of the KL term, allowing the model to focus more on minimizing reconstruction error. This can lead to better detail in reconstruction but at the risk of overfitting and a less regularized latent space.

Adjusting β is essential for controlling the trade-off between diversity (variation) and fidelity (accuracy) of the reconstructions.

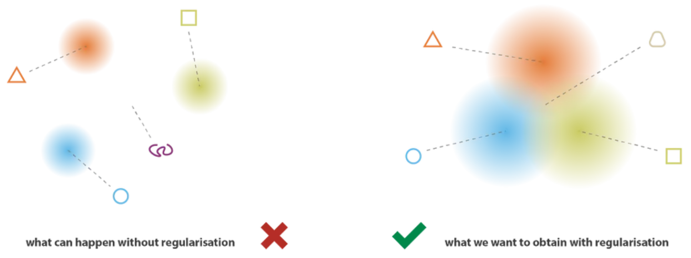
what can happen without regularisation
what we want to obtain with regularisation

Figure 7: Effects of KL Divergence regularization on latent space

Other hyperparameters include the learning rate, number of epochs and the number of latent dimensions. Through trial and eror the following hyperparameters were defined for the best performing model:

```
input_shape = (256, 64, 1)
conv_filters = (512, 256, 128, 64, 32)
conv_kernels = (3, 3, 3, 3, 3)
conv_strides = (2, 2, 2, 2, (2, 1))
latent_space_dim = 128
reconstruction_loss_weight = 1000000
number_of_epochs = 200
learning_rate = 0.0005
```

8 Results and Evaluation

8.1 Reconstruction of new data

After optimizing the model and identifying the optimal weights, we proceeded to evaluate its performance on new audio samples that were not part of the training set. To do this, I created a personal dataset by recording myself pronouncing the digits from 0 to 9, adhering to the same specifications as those used in the Free Spoken Digit Dataset (FSDD). These recordings were then preprocessed in an identical manner to the training data to ensure consistency. Subsequently, I tested the model's ability to reconstruct these new audio inputs, enabling us to assess its generalization capabilities and effectiveness in handling unseen data.

```
def reconstruct(encoder, decoder, batch_of_images):
    latent_representations = encoder.predict(batch_of_images)
    reconstructed_images = decoder.predict(latent_representations)
    return reconstructed_images, latent_representations
```

The above function takes input data and maps it to a latent space by running a prediction in the encoder. Then the resulting output is used to run a prediction on the decoder, taking that latent representation and attempting to reconstruct the original input data from this compressed representation.

The following images show the comparison between the original audios vs those reconstructed by the Variational Autoencoder model. They graph the audio signals and the STFT spectrograms of both audios.

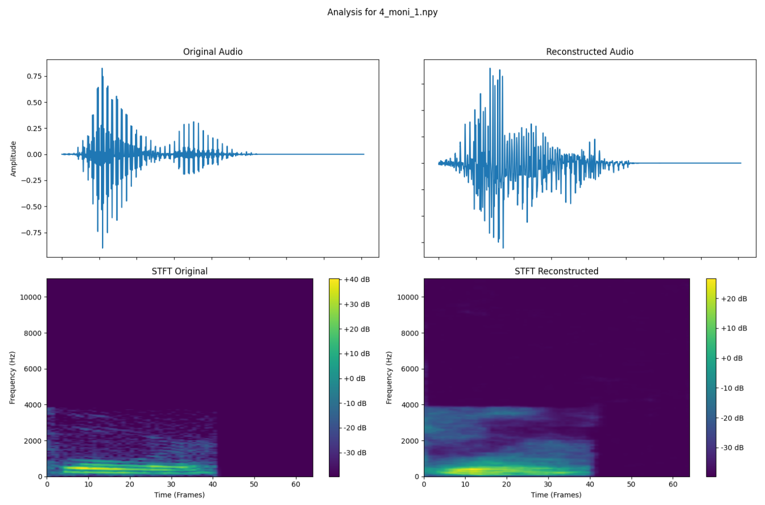

Figure 8: Reconstruction of audio file through the VAE

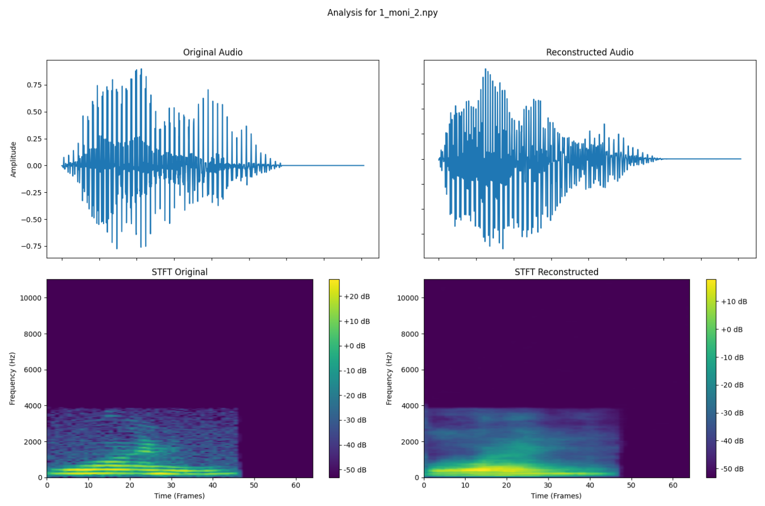

Figure 9: Reconstruction of audio file through the VAE

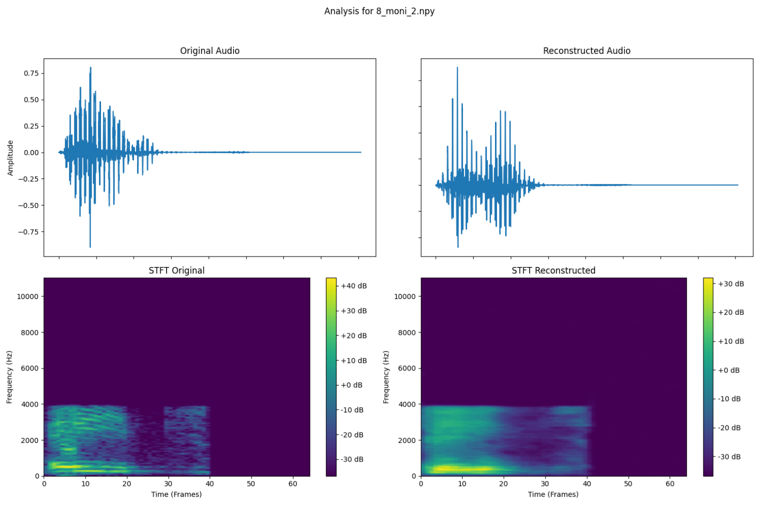

Figure 10: Reconstruction of audio file through the VAE

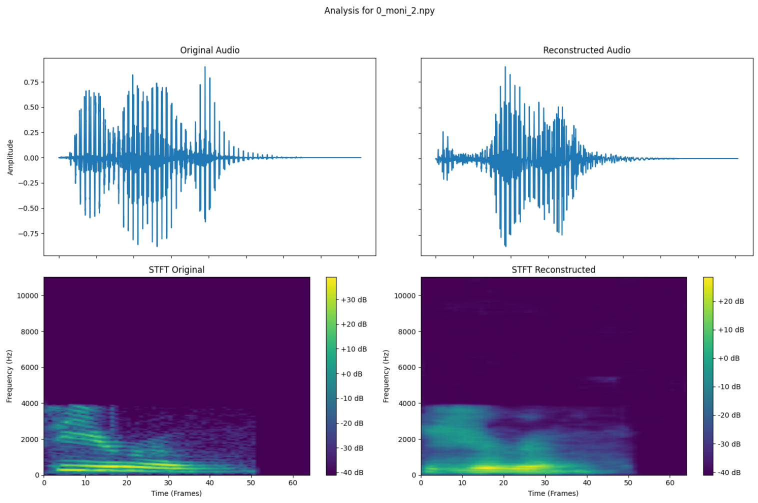

Figure 11: Reconstruction of audio file through the VAE

8.2 Generation of new and specific data

Using VAEs for generating new audio involves manipulating the latent space—a compressed representation of the data learned by the model. Below, I will explain the provided functions and the theory behind generating new audio data with a VAE.

The generate function is designed to create new audio samples from random points in the latent space:

```
def generate(decoder, num_samples, latent_dim=128):
    z_sample = np.random.normal(size=(num_samples, latent_dim))
    generated_data = decoder.predict(z_sample)
    return generated_data
```

Based on how many samples we want to generate, we obtain an array of latent vectors z_sample, each sampled from a standard normal distribution in the latent space. These latent vectors are then used as inputs for the decoder, which tries to reconstruct the audio data from these latent points. The decoder's output is the generated audio data for each of these samples. Then we run a recontrcut from spectrogram function to obtain the real generated audio. However, this would be a random audio generation from the latent space.

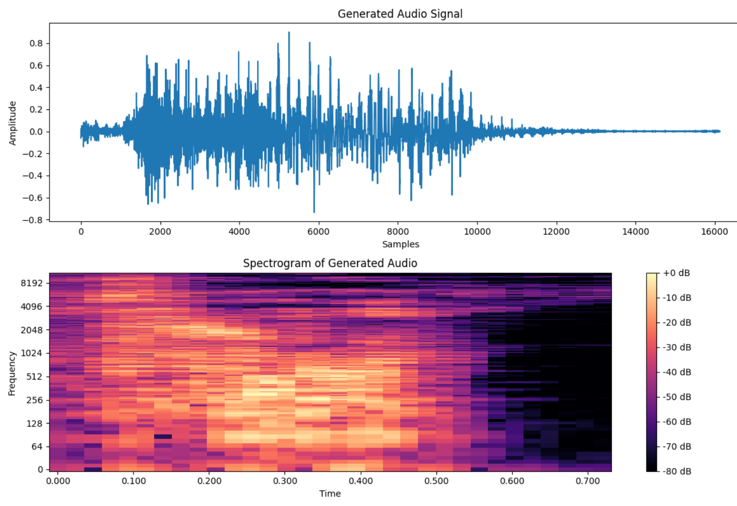

Figure 12: Generation of audio through the VAE model

VAEs have another advantage: the ability to generate specific types of audio, in this case, specific digits, can be enhanced by first finding the mean latent vectors for each class and then sampling from a distribution centered around the mean latent vector of the desired class. This can be done with the following code.

```
def find_class_mean_latent_vectors(encoder, data, labels, num_classes):
    latent_vectors = encoder.predict(data)
    class_mean_vectors = {}
    for i in range(num_classes):
        class_latent_vectors = latent_vectors[labels == i]
        class_mean_vector = np.mean(class_latent_vectors, axis=0)
        class_mean_vectors[i] = class_mean_vector
    return class_mean_vectors

def generate_from_class(decoder, class_mean_vector, num_samples=1, scale=0.5):
    z_sample = np.random.normal(loc=class_mean_vector, scale=scale, size=(num_samples, class_mean_vecto
    generated_data = decoder.predict(z_sample)
    return generated_data
```

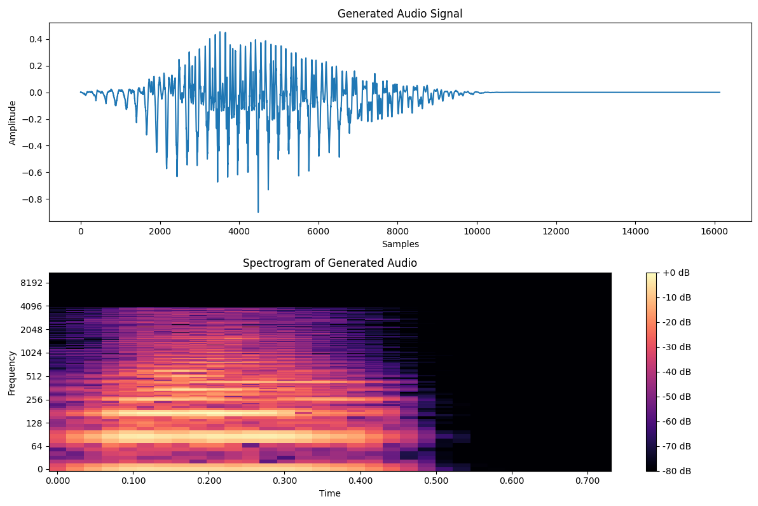

Figure 13: Generation of digit 9

The key theoretical aspect here is that VAEs not only learn an efficient compression of the data into a latent space but also enforce this space to follow a regular, continuous distribution (typically Gaussian). This regularization (enforced by the KL divergence term in the loss function) allows for meaningful interpolation and sampling within the latent space.

9 Exploring the latent space

Exploring the latent space of our Variational Autoencoder allows us to understand how well the VAE has learned to separate and represent different classes (digits 1-9) within its latent space. Techniques like Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) are usually employed as they allow us to visualize the high-dimensional latent space in two or three dimensions.

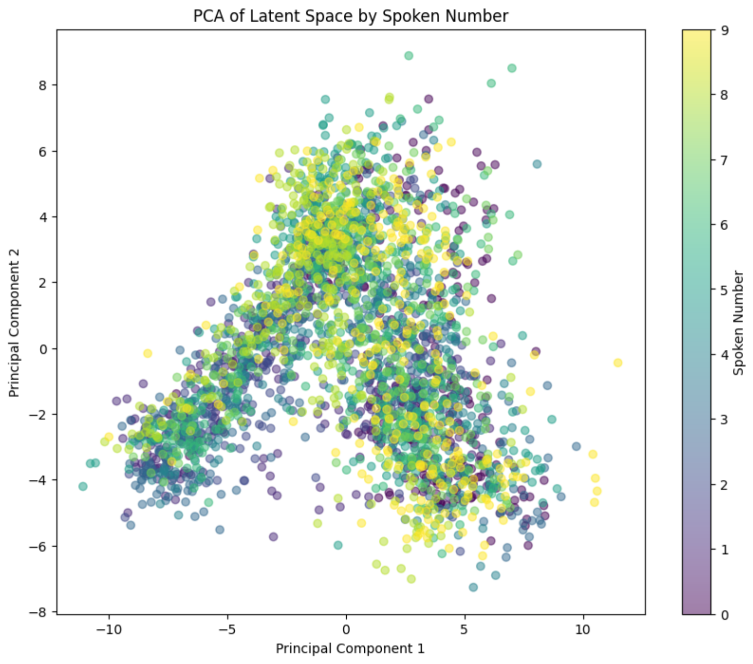

Figure 14: PCA exploration of Latent Space by spoken digit

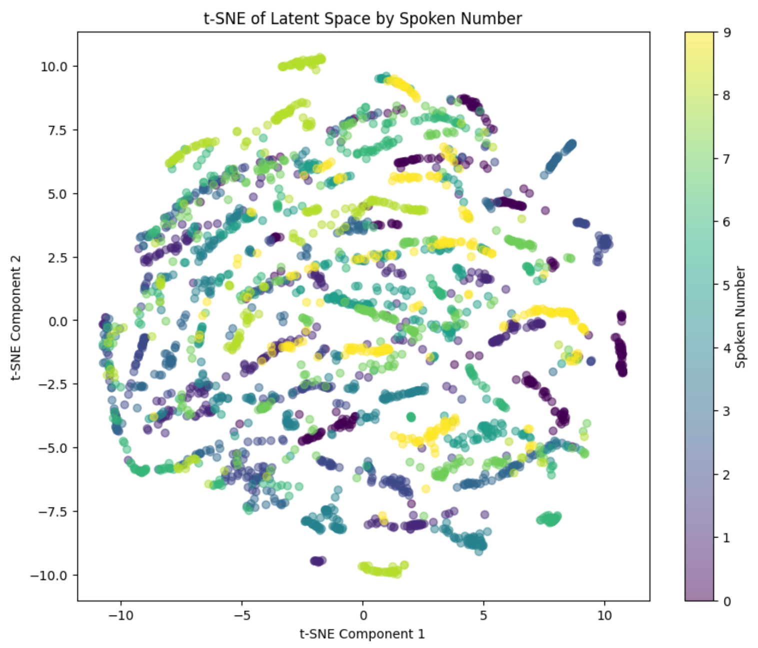

Figure 15: t-SNE exploration of Latent Space by spoken digit

These types of visualizations and other explorations of the latent space can also help in fine-tuning the model by identifying areas where performance might be improved (such as classes tightly grouped together that could be more separate).

10 Conclusion

In conclusion, this project has demonstrated the capabilities of Variational Autoencoders (VAEs) in the domain of audio generation and reconstruction with the Free Spoken Digit Dataset (FSDD). Through the application of dimensionality reduction techniques and preprocessing measures, the study efficiently managed audio data, enabling the effective training of a variational autoencoder. The transition from a more intricate dataset of Taylor Swift's discography to the FSDD was crucial in the success of the project.

The comparative analysis between the standard VAE and the Gaussian mixture VAE revealed distinct advantages in how these models handle audio data. This project is a foundation for my future explorations into more complex audio generation tasks, such as music. In the future I want to implement GM-VAE ir VQ-VAEs for music generation where the audio data presents significantly higher complexity.

11 Final Notes

- **Audio quality:** It is difficult to differentiate the audio samples and the originals but graphical representations such as signals and spectrograms allow us to see the results are satisfactory.
- **Latent Space PCA:** Using Principal Component Analysis means we lose out on some of the variance of the data and the exploration of the latent space is quite incomplete in this study.
- **Harmonic representation:** At one point during the project harmonic representation was explored and still holds great potential for reducing dimensionality while allowing us to have higher sample rates.

References

[1] Anastasia Natsiou and others. An Exploration of the Latent Space of a Convolutional Variational Autoencoder for the Generation of Musical Instrument Tones. In *xAI*, 2023.

[2] JP Briot and F Pachet. Deep learning for music generation: challenges and directions. *Neural Comput & Applic*, 32:981–993, 2020. `https://doi.org/10.1007/s00521-018-3813-6`

[3] P Dhariwal, H Jun, C Payne, JW Kim, A Radford, and I Sutskever. Jukebox: A Generative Model for Music. *ArXiv preprint ArXiv:2005.00341*, 2020. Available at `https://openai.com/research/jukebox`