FAKULTÄT:

ANGEWANDTE MATHEMATIK, PHYSIK UND ALLGEMEINWISSENSCHAFTEN

Masters in applied mathematics and physics

Master thesis:

SOUND AND TONE REPLICATING AI

Understanding generative neural network architecture

Submitted by:	Rohit Upreti
Matriculation number:	3556227
Program of Study:	Applied Mathematics and Physics
Professor:	Prof. Dr. Elke Wilczok
Primary supervisor:	Prof. Dr. Elke Wilczok
Secondary supervisor:	Prof. Dr. Peter Jonas
Submitted on:	30.09.2023.

Abstract

The human ear can hear and recognize a person's voice once they are familiar with that person. Furthermore, humans can also mimic a person's voice after hearing it. Machines on the other hand follow a set of instructions and need to be trained. There are already several learning methods where machines are used to mimic voices, and in recent years, Text-to-Speech (TTS) technology has been widely employed. In this thesis, a machine learning algorithm and its probabilistic model are discussed. Unlike traditional neural networks, the Autoencoder model employs both an encoder and decoder. During the decoding process, a similar yet different output is generated for a given input. And autoencoders with probabilistic foundation are called Variational autoencoders.

Since audio cannot be directly used as input, several operations are performed on the audio signal. The first step involves converting the audio into a spectrogram, which can be thought of as an image being the audio data. While a digital image typically comprises of three-color bands—Red, Green, and Blue—these images are processed by passing each band through the hidden layers of a neural network. To optimize computational efficiency and reduce processing time, the color bands are converted into grayscale images, where pixels vary in shades of gray between pure white and black. Batches of these grayscale images are then fed into a neural network for training and data generation. The resulting spectrogram is a processed representation that replicates and yet differs from the original audio input.

Index

List of Illustration

List of abbreviation

FSDD-Free spoken Digit Dataset

Hz-Herz

MNIST-Modified National Institute of Standards and Technology

DFT- Discrete Fourier transform

CFT- Continuous Fourier transform

STFT Short time Fourier transform

VAE Variational autoencoder

Introduction:

In May 2023, Apple introduced a feature that can clone a user's voice with just 15 minutes of recorded data (Apple, 2023).Not only Apple, many other companies and individuals can now manipulate audio files to mimic voices (Geekflare, 2023). This thesis presents the architectural framework for building such neural networks, as well as the mathematical methods for data preprocessing. In machine learning, the quality of the outcome depends on the quality of the training and testing data. Depending on the method of learning, data must not only be clean but also identifiable, ensuring that distinct patterns can be recognized and extracted effectively. Furthermore, several types of learning, including supervised, unsupervised, and generative learning, are explored within this context. In this thesis, an autoencoder is trained using a generated set of spectrograms from FSDD (R., 2020), and for comparison, the model is also trained on the MNIST dataset (MNIST, 2019).

This thesis is divided into four sections:

1. Audio processing and spectrogram:

   This section explains the basis of the signal generation, propagation, and some of the features of this audio signal. After this, a short time Fourier transformation(STFT) is done, and this spectrogram is then taken as input for the neural network.

2. Deep Learning:

   This section provides an understanding of deep neural networks and convolutional layers.

3. Autoencoder:

   This section introduces the basic concept of the generative model. Autoencoder itself can be used as generative model but is not termed as generative neural network.

4. Variational Autoencoder:

   Variational Autoencoder are the generative model that combines the autoencoder element and combines a probabilistic model. Here is a working understanding of this model is explained.

The conclusion and discussion are done in the last section of thesis. Python is used as a programming language and scripts are to be found in “Code” section.

1.Audio processing and Spectrogram:

A very simple act of disturbance in the medium causes a tone or sound to generate. When we speak, we disturb the molecules in air and the air molecules behave in a certain manner. It then becomes a wave, has a certain intensity, frequency, and travels with speed of sound. The human ear is sensitive to both Intensity and frequency. This Intensity can be represented and analyzed in both time and frequency domains.

1.1 Time-Frequency domain

Since sound propagates as a wave, it possesses a specific frequency and amplitude. In cases where the wave is long enough, periodicity can be seen, indicating that the wave repeats itself after a certain period with a fundamental frequency. However, it's important to note that most audio samples, including everyday conversations and practical purposes, are non-periodic in nature. Even simple pieces of music exhibit complex frequency spectra upon analysis, even if their intensity is low. While the time-amplitude domain is useful for visualizing the waveform, integral transformations allow the use of the frequency spectrum for comprehensive analysis. The transformation from the time domain to the frequency domain is known as the Fourier transformation (Kulkarni, 2002).

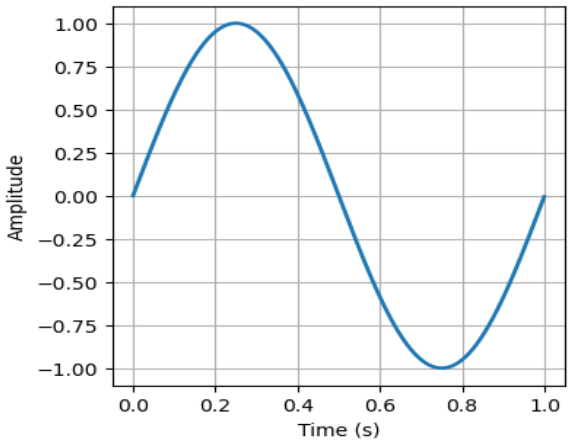


Figure 1 Continuous wave in time domain.

A continuous wave with a period of 1 sec is shown in the figure. It takes one sec to complete its form, thus its frequency is 1Hz. If X(t) is its wave equation, then it is expressed in time domain as,

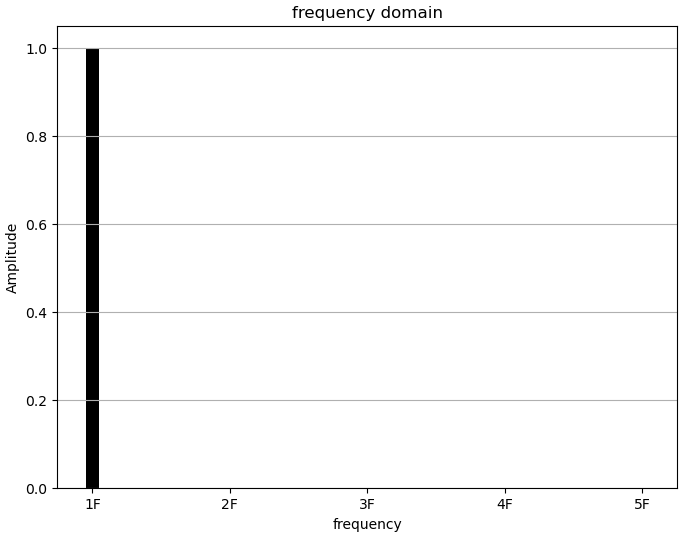


Figure 2 Frequency domain representation.

Above is the schematic representation of the frequency domain at 1 Hz.

The Fourier transform for a continuous wave from time to frequency domain is given by:

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-j2\pi ft}\, dt \qquad 1$$

Where, f is calculated for the given time t.

But the sound waves we encounter in daily life are not periodic. That is why there are different tones and expressions while speaking or in music as well. There might exist a whole set of discretized frequency even for small amount of time Thus a transformation is needed to analyze this discretized frequency and time. This transformation is called Discrete Fourier transformation (DFT).

The Discrete Fourier Transform (DFT) serves as a critical tool in signal processing and data analysis, bridging the gap between continuous signals and their digital counterparts. While the Continuous Fourier Transform (CFT) is designed for continuous-time signals, real-world data is often discrete and sampled. Herein lies the importance of DFT: it allows us to analyze and

manipulate digital signals efficiently. DFT breaks down a discrete signal into its constituent frequency components, revealing hidden patterns and characteristics. With DFT, we can perform tasks like spectral analysis, filtering, and feature extraction, enabling us to extract meaningful insights from real-world data in a way that was simply not feasible with the continuous Fourier Transform (Lyon, 2004).

The mathematical expression for DFT in trigonometric form is,

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot (\cos(2\pi kn/N) - j \cdot \sin(2\pi kn/N)) \qquad 2$$

And in exponential form,

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi nm/N} \qquad 3$$

X[k]: This is the DFT output at frequency bin k. It is the complex amplitude of the corresponding frequency component in the input signal *x[n]*.

k: This is the frequency index or bin. It ranges from 0 to N−1, where N is the total number of samples in the input signal. Each k corresponds to a specific frequency component in the frequency domain.

n: This is the time index. It ranges from 0 to N−1 and represents the discrete time samples of the input signal *x[n]*.

For each frequency bin k, the DFT calculates the contributions of each sample x[n] by multiplying it with the corresponding complex sinusoidal components at the given frequency. These components, represented by cosine and sine functions, capture the real and imaginary parts of the signal's frequency content. Summing up these contributions provides the complex amplitude *X[k]* for each frequency bin, revealing the signal's spectral characteristics, including frequency magnitudes and phases. The DFT is fundamental in applications such as signal processing, audio analysis, and image processing for extracting valuable frequency domain information from discrete data.

1.2 Short Time Fourier transform (STFT)

The Short-Time Fourier Transform (STFT) is an integral transformation method used to extract spectrograms, while the DFT provides an averaged frequency value for a given time. Since audio samples often contain various frequency spectrums, and for practical purposes, using

DFT alone may not suffice. However, DFT still plays a crucial role in the analysis. To better understand this, let's consider a real-life audio sample, such as Neil Armstrong's words spoken on the Moon (NASA/Audio), and apply the DFT to analyze the issue. Below is a time-domain representation of the audio.

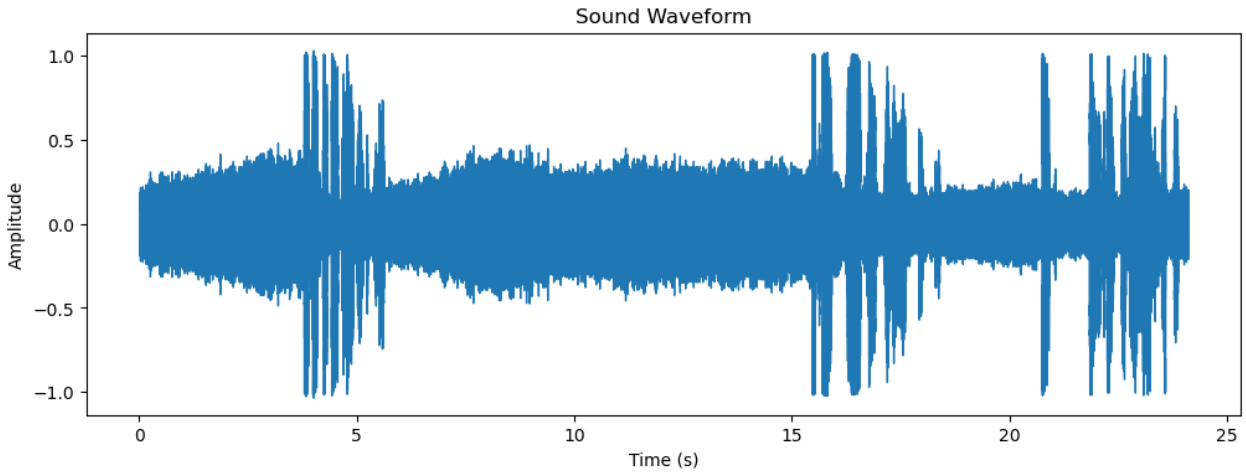


Figure 3 Sound wave of first words spoken on the moon.

As seen in the diagram, the audio signal is not periodic and frequency changes rapidly over time. If the DFT were to be done in such audio signal, then such a frequency spectrum would be obtained.

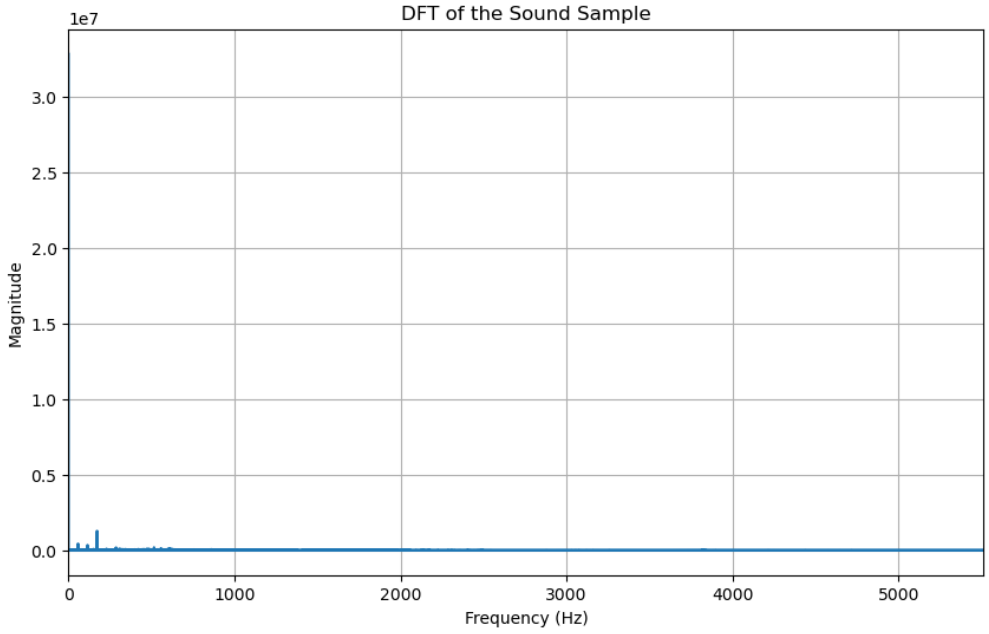


Figure 4 DFT of a non-periodic signal.

As expected, the DFT provides an averaged representation of frequency for a given time window. This characteristic can be problematic when dealing with non-stationary signals, where the frequency content changes rapidly over time. To address this limitation, a window function is introduced in the DFT, which can be thought of as a 'window' sliding over the signal. STFT is sometimes referred to as the Windowed Discrete Fourier Transform because it defines the section by multiplication with input function. Depending on the nature of the input signal and the desired analytical outcome, various window functions, such as Rectangular, Hanning, and Hamming, can be applied to capture the dynamic frequency content more effectively. These window functions help extract time-frequency information, making STFT a valuable tool for analyzing non-stationary signals like speech or music (auidiolabs-erlangen, 2015).

While performing the STFT, a longer audio signal in the time domain is first segmented into smaller sections or windows. Within each of these windows, a window function is applied. The purpose of this window function is to make the signal within each window stationary and to make the entire audio signal appear more periodic. Subsequently, the Discrete Fourier Transform (DFT) is performed on each of these windowed segments. This process results in the final frequency spectrum, providing valuable time-frequency information for the analysis of non-stationary signals like speech or music.

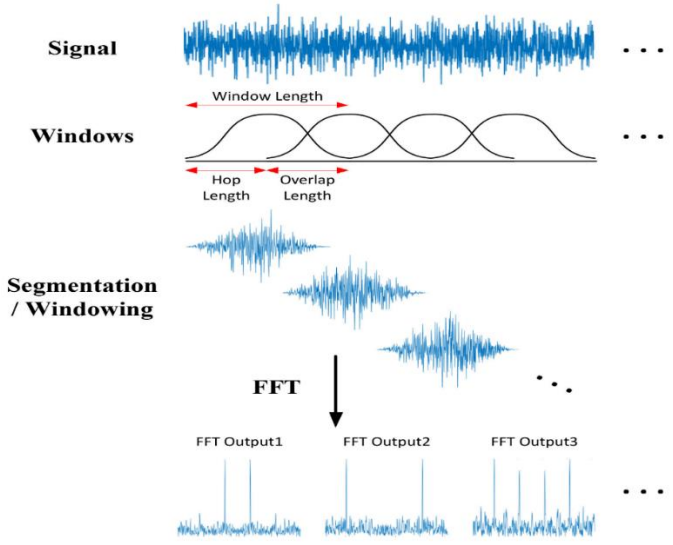


Figure 5 Schematic representation of STFT process (*Appl. Sci.* **2020**, *10*(20)).

Above is a representation of the STFT process. Let *x(t)* represent the input signal and this signal is segmented. The window function is then applied to these segmented signals and frequency in each window is calculated.

The mathematical expressions for the STFT is,.

$X[k,m] = \sum_{n=0}^{N-1} x[n + mH] \cdot w[n] \cdot e^{-j2\pi kn/N}$ 4

X[k, m] represents the STFT coefficient at frequency bin k and time frame m.

x[n] is the input signal.

N is the total number of samples in each window.

w[n] is the window function applied to segment the signal.

H is the hop size, which determines the overlap between consecutive windows (audiolabs-erlangen, 2015)

1.3 Extracting spectrogram

In this section, the Short-Time Fourier Transform (STFT) process is developed, and essential parameters involved are explored. To perform STFT, Powerful libraries Librosa and Matplotlib for efficient signal analysis were imported. When STFT is performed, several crucial parameters come into play. These parameters include the choice of window function, the window's length, frame size, and hop length. Each of these parameters plays a pivotal role in shaping the STFT's outcome and its ability to capture time-frequency information effectively. Below is their brief introduction and their relationship in generating a better spectrogram.

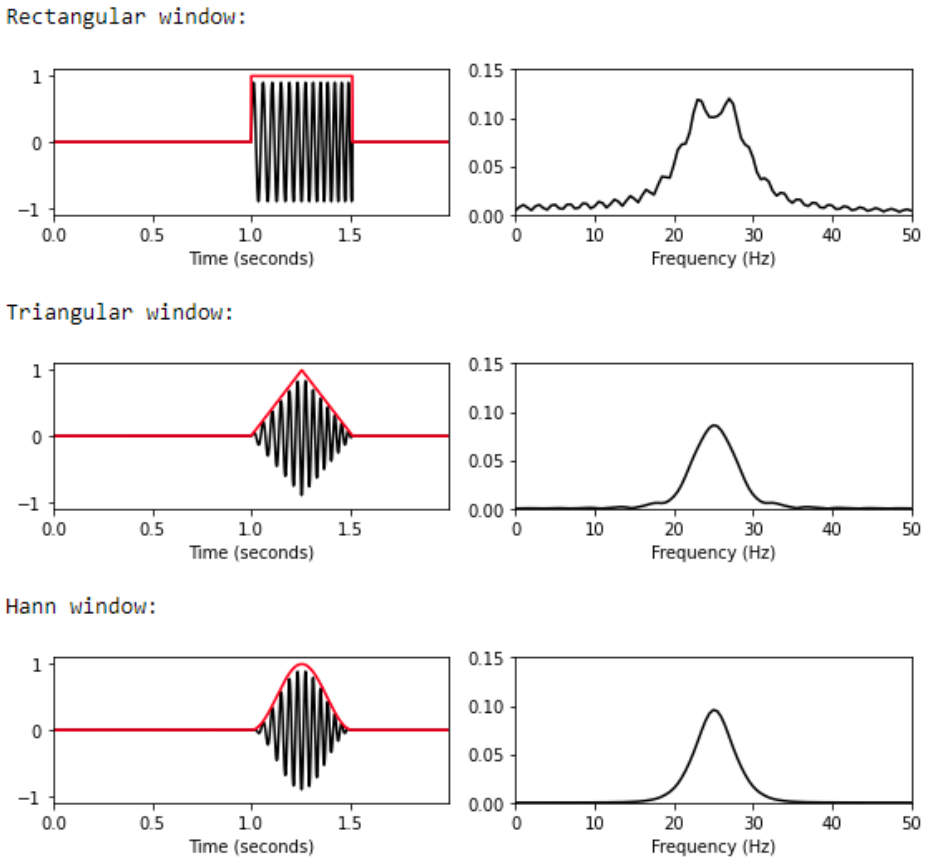


Figure 6 Application of different window function . (audiolabs-erlangen.de).

Figure 6 represents the application of the window function. Defining the window function is the first step in STFT. A window function, in the context of signal processing and spectral analysis, is a mathematical function applied to a segment of a signal to modify its characteristics before further analysis, such as STFT. There are several window functions to choose from, in this thesis Hanning window is chosen.

The Hann window was invented by Julius von Hann, an Austrian meteorologist , around 1900. The primary purpose of the Hanning window, like other window functions, is to shape the signal in the time domain before performing operations such as the Fourier transform. When applied to a signal, the Hanning window reduces the signal's amplitude near its start and end points while preserving its central information. This ensures that the transitions between the window functions are smooth. This reduction in amplitude minimizes the noise that can appear in the frequency domain. To visualize the Hann window, audio sample from section 2.2 is taken.

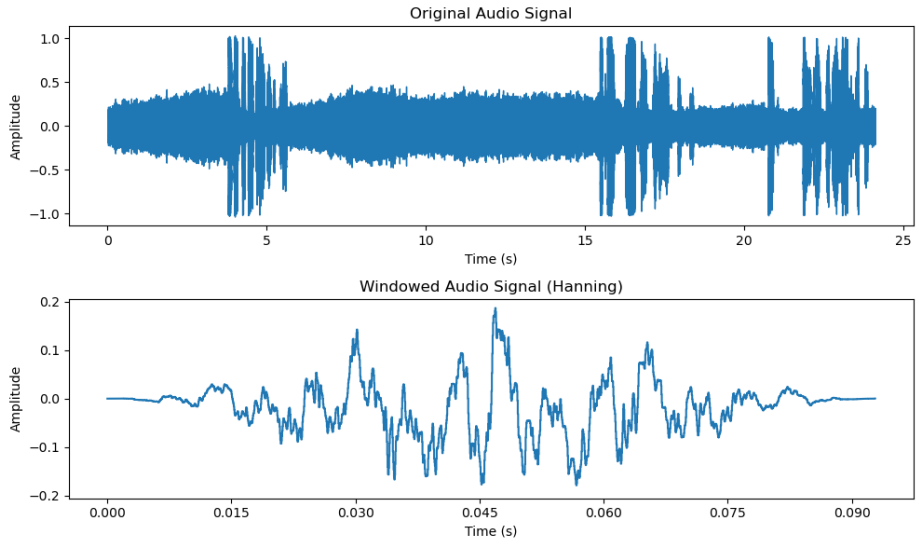


Figure 7 Application of Hanning window in a segment of an audio signal.

Figure 7, representation of the application of the Hanning (Hann) window. This window function is applied to the segmented audio signal and slid across the entire audio signal. Each window has a certain length, win length . Under this window, it encompasses certain samples. These are samples upon which DFT is performed.is known as Frame size ,as this window function is applied to segmented audio signal, it can be segmented in a way that the function is overlapped. This overlapped region is called overlap region and the non-overlapped region is called Hop length. Hop length" is not a region but a specific parameter related to the overlapping of windows. In python, the program treats Hop length and overlap region as equivalent. The python script for the spectrogram extraction can be found in “Codes” section.

Spectrogram is essentially plotting the varying frequency against the time domain. And here comes the question, how the best frequency-time resolution can be obtained. The final spectrum is plotted against the frequency bins and time. And the quality of this spectrum depends upon the parameters chosen, especially Hop length and window length. Larger window length allows to capture more extended time segments of signal and with smaller hop length . Below is the final spectrogram for the words spoken by Niel Armstrong and other examples (Pixabay/effects). Hanning window is taken as the window function.

1. Spectrogram extraction of Neil Armstrong's voice with following parameters:

Hop Length: 512

Frame Size: 2048

Win length: 300

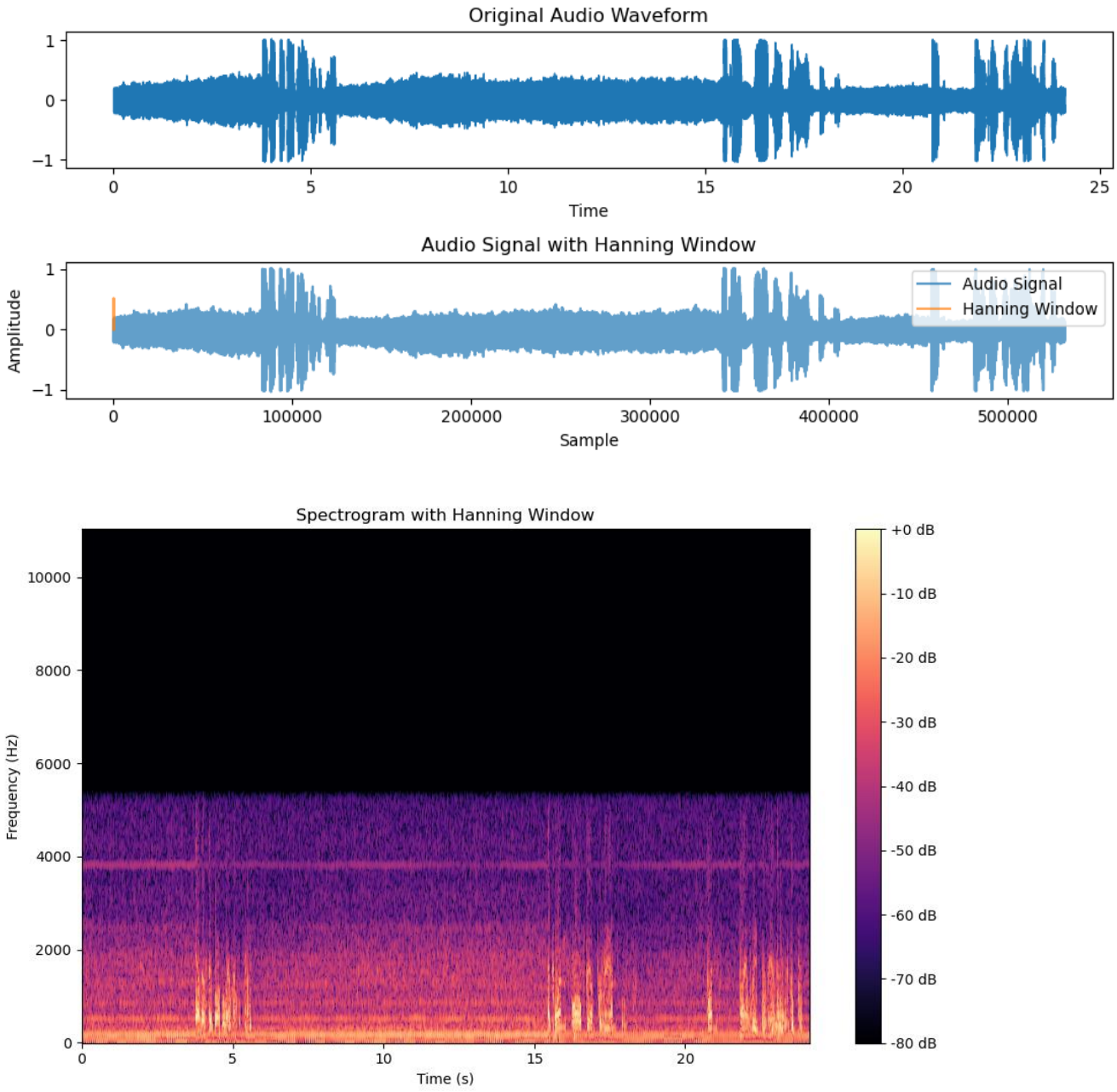


Figure 8 Sound wave and spectrogram of the first words spoken on Moon.

2. Spectrogram of distant ambulance siren

Hop Length: 550

Frame Size: 3500

Win length: 500

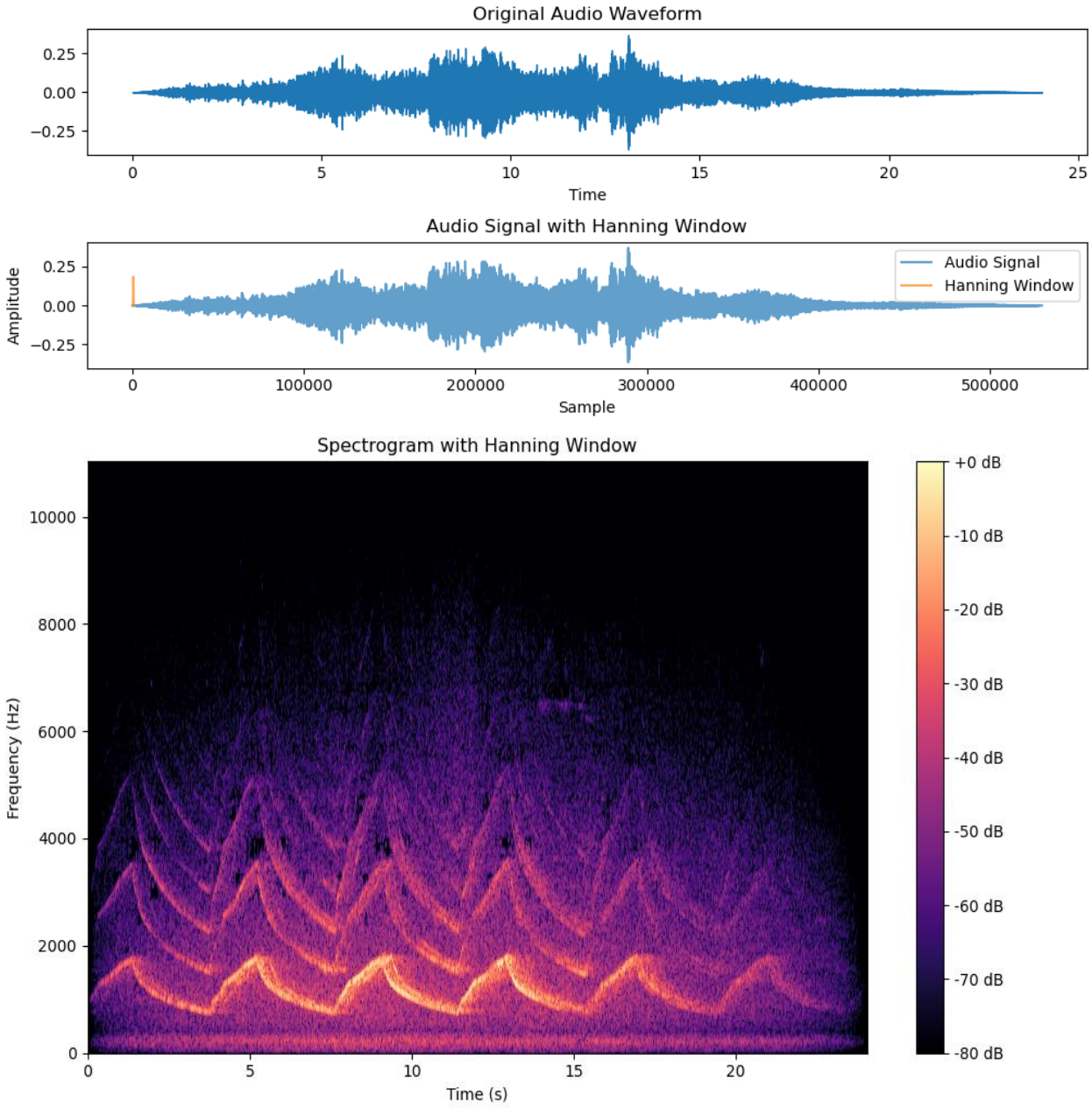


Figure 9 Sound wave and spectrogram of ambulance siren.

3. Spectrogram of Birds chirping

Hop Length: 200

Frame Size: 5500

Win length: 500

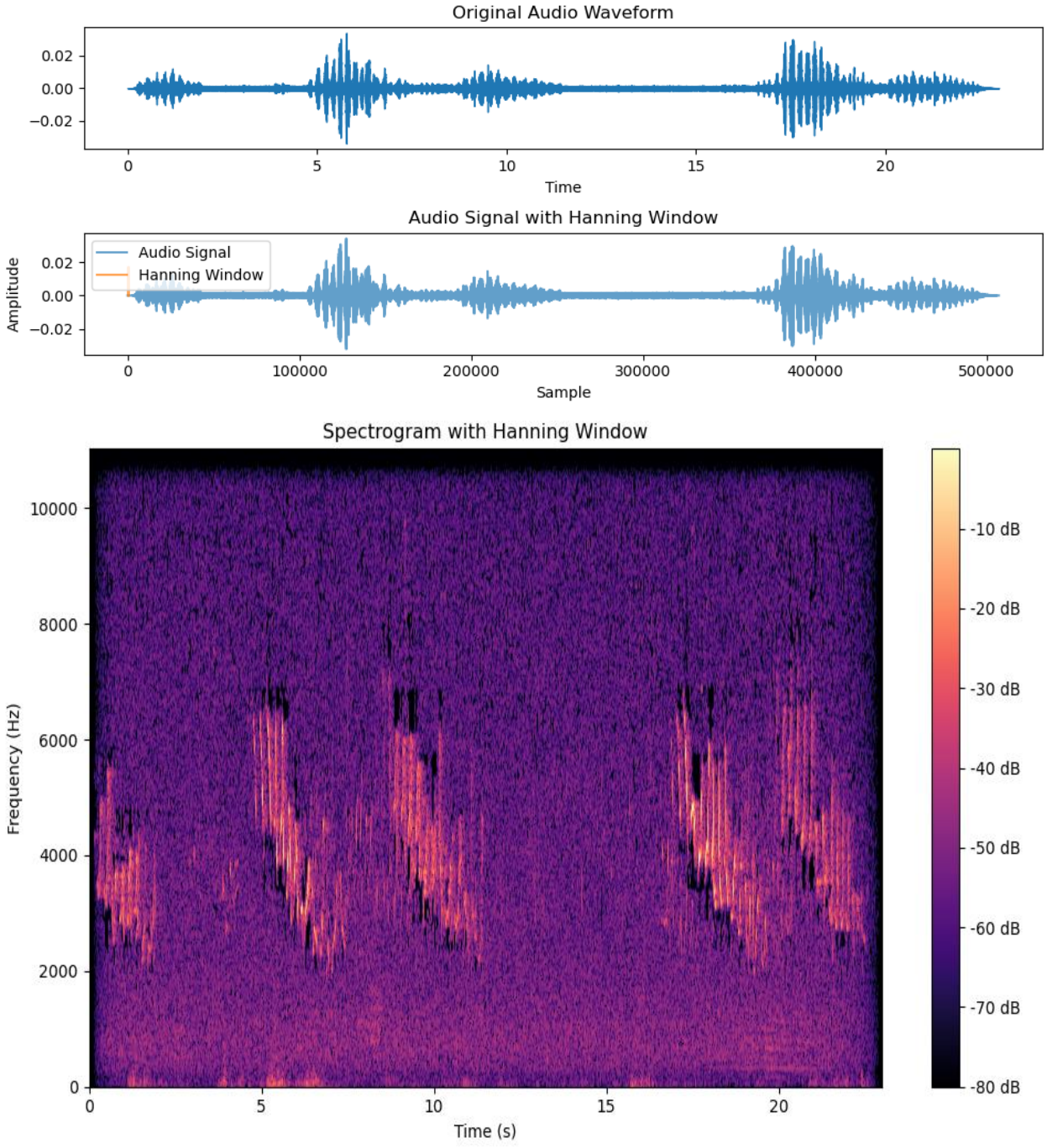


Figure 10 sound wave and spectrogram of Birds chirping.

The following are examples of spectrogram extraction. Samples of a person's voice, sirens, and natural sounds were used to illustrate the differences in sound characteristics. Although sometimes two sounds may seem similar, analyzing the spectrogram for frequency and time resolution provides a clearer representation of the sound. This concept is at the core of spectrogram extraction, which essentially involves capturing a visual representation of voices, tones, sounds, or any other signals. Parameters are fine-tuned based on the desired outcome. The resulting spectrogram is then used as input for the neural network. During data training, certain transformations are applied, which are discussed in the 'Auto-encoder' section.

2. Deep Learning:

There are several types of learning in the context of machine learning and artificial intelligence. Types of data, features extraction, convolution layers add even more complexity to the model. In this thesis, Spectrogram is taken as the input data and unsupervised learning as type of learning method.

2.1 Deep neural network

A machine can learn from its mistakes. And this process can be termed as Deep Learning. Deep neural network unravels the working structure of Deep learning and is necessary for the generative model as well. Deep neural networks consist of a series of stacked layers. Each of these layers is filled with neurons. And as the model gets trained, it gets updated with some weights and parameters. For multidimensional array of input like image and spectrogram, the input is flattened and stacked in flatten layer as input layer.

The spectrogram obtained from section 3 is in color format. For a 128*128 pixel the input shape is of the form [128 128 3]. The 3 here represents the color channel Red, Green and Blue. Such multidimensional array is flattened, and each neuron is assigned with a certain weight between 0-1.

[128*128*3] = 49152 neurons with some weight assigned to each neuron.

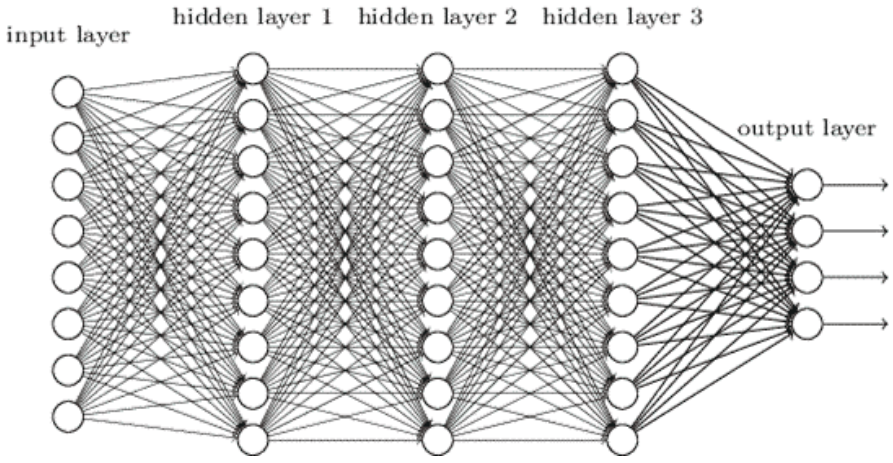


Figure 11 Schematic representation of a Deep neural network (datacareer.de).

Figure 11 is a schematic representation of a deep neural network. The first layer is the input layer. Then follows the hidden layers. There are several types of layers in deep neural networks to choose from. Convolutional layers, Recurrent layer layers, Dense layer , Batch Normalization Layer being some common examples. Every layer has a different feature and function. For e.g., in Dense layer every layer is connected to previous layer so that the weights and parameters can be passed in between layers. Depending on the model layers can be chosen. In this thesis, convolutional layer is taken as layer in neural network The hidden layers are stacked with certain units of neurons. The number of neurons in the layer defines the unit of layer. For e.g., a Dense layer with 200 units contains 200 neurons. And the final layer is the output layer. For a neuron to fire in hidden layer and to continue the neural network, certain minimum condition must be met in these layers. Every neuron from flattened layer carries its weight and travels to neuron in hidden layer. A bias or threshold value is assigned to the neurons in the hidden layer. Only those neurons fire up, where this threshold value is met. This relationship between the weight of incoming neurons and the threshold value can be modeled into a function. This function in neural network is called activation function. Sigmoid, ReLU, Leaky ReLU are some of the activation functions and can be chosen per requirement. In this thesis, sigmoid function, and Rectified Linear unit (ReLU) is used as activation function (Foster, 2019)

2.2 Convolutional Layer

Convolution is a fundamental operation in deep learning used for extracting meaningful features and patterns from input data, such as images. It involves sliding small filters or kernels over the input data, performing local element-wise multiplications, and summing the results to produce feature maps**.**

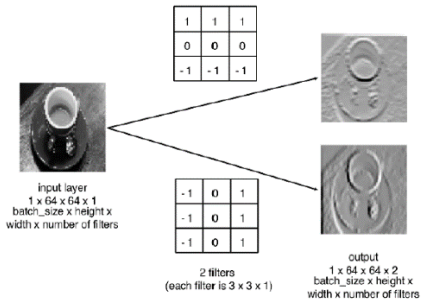


Figure 12 Image convolution with 2 filter (Generative Deep learning, Oreilly).

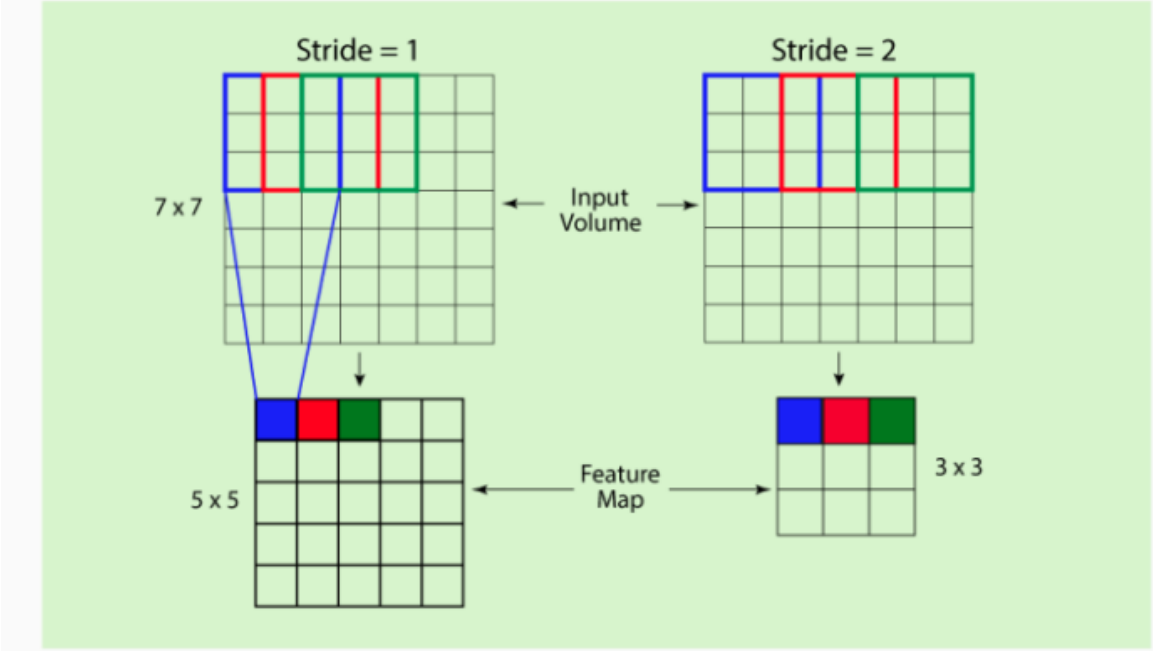


Figure 13 Kernel and striding in a convolution layer (developersbreach).

Figure 12, a filter of size 3*3 is applied two times in an input image. This breaks down the spatial structure of the image. A clear distinction between cup and saucer. Figure 13, how a filter slides across the input. The filter gets applied into the input and moved across the input. This drag of the filter is called stride and can be chosen per requirement. As the filter strides through the input, some kernels may not find input value in the grid. To keep the feature map uniform, a padding function is also used simultaneously .

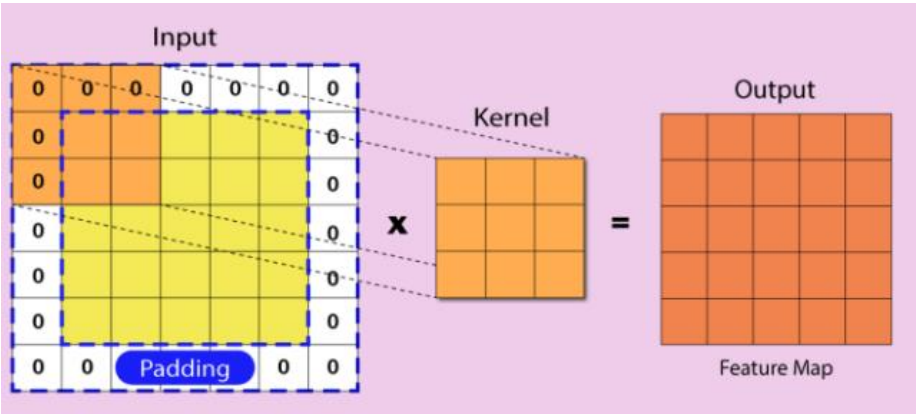


Figure 14 Padding in convolution layer (developersbreach)

The final feature map is then the output, which captures the features and spatial structures along.

When a deep neural network is trained, it gets trained with thousands of different data. In fact, it is recommended to use a wide variety of data to improve the performance of the machine. When convolutional layers are used in the neural network, the size of the kernel remains constant. While the matrix which represents weights or the dimension vector changes and gets updated after every training session (Foster, 2019) (IBM).

3 Autoencoder:

The term "autoencoder" was introduced by Geoffrey E. Hinton and Terrence J. Sejnowski in their 1986 paper titled "Learning and Relearning in Boltzmann Machines" (Hinton, 1986). This paper laid the foundation for the concept of autoencoders, which involve neural networks that encode and decode data, serving as a crucial development in the field of neural networks and unsupervised learning. This section focuses on building an encoder and decoder and bringing these components together.

3.1 Preparing the data.

A machine learning algorithm is only as good as the data used. Data needs to be diverse and robust. Wide range in data avoids the biasness and overfitting in the model. Here the spectrogram is an image, it is insightful to understand it in image terminologies.

3.1.1 Pixel, Resolution and Resizing.

A pixel is the smallest individual unit in a digital image or display. The number of pixels in each dimension is called its Resolution. For generated images in the screen, it is termed Pixel Dimensions. The spectrogram is taken as an image and the resolution being 28*28 = 784 total pixel. But the original image being 690*545 = 376050 pixel. To obtain the desired input size, compression is done in the image. This concept of compression is particularly important, as some of the features may be lost during this process. The science of image compression and resizing is a different topic, but a basic interpolation concept of image is explained in this subsection.

Figure 15 Upscaling of a 28*28(left) image to 690*545-pixel dimension.

Above is an example of image upscaling or increasing the size of pixel in an image. The image on the left is 28*28 pixel and on the right being 690*545 pixel. One can clearly see the difference in the qualities and when a compression is done in an image, one can understand the scale of loss of the information.

1	1	1
1	1	1
1	1	1

1			1			1
1			1			1
1			1			1

Figure 16 Upscaling of a 3*3-pixel image by a factor of 3.

Above is an example of simple upscaling of an image. The pixel value is set at 1 and the imaged is upscaled. As seen in figure above, the points (0,1), (0,2) etc. are unfilled and would be invisible, if not zoomed in. The process of assigning these pixel grids value with a mathematical function is called Interpolation. Common interpolation techniques are Nearest-Neighbor, Bilinear, Bicubic and Lanczos. Depending upon the image and outcome, the technique can be chosen. Interpolation can also be applied to make the pixel

value uniform and smooth. The inverse process is called downsizing. The resolution is decreased, and the pixel grids are closed in together. A distortion or so called “jagged” effect is observed when these pixel grids are closed in together. This effect is called aliasing. After downsizing, pixel value gets smoothed out and this technique is called antialiasing (Savagave, Patil, 2014). (Dey, 2018).

In this thesis, the Autoencoder is trained with two sets of data:

- MNIST
- FSDD

MNIST is a large database of handwritten digits from 0-9. It contains 60,000 training images and 10,000 testing images. The image size is 28*28-pixel, grayscale and anti-aliased. Because of its diversity and size, this database is usually taken to train the deep learning model.

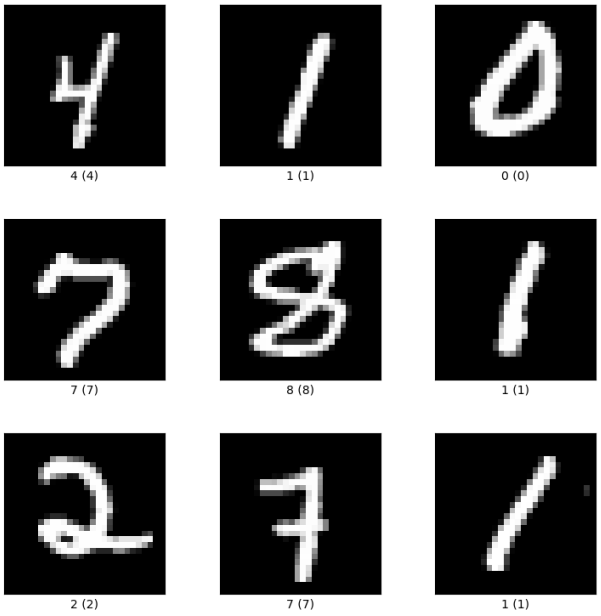
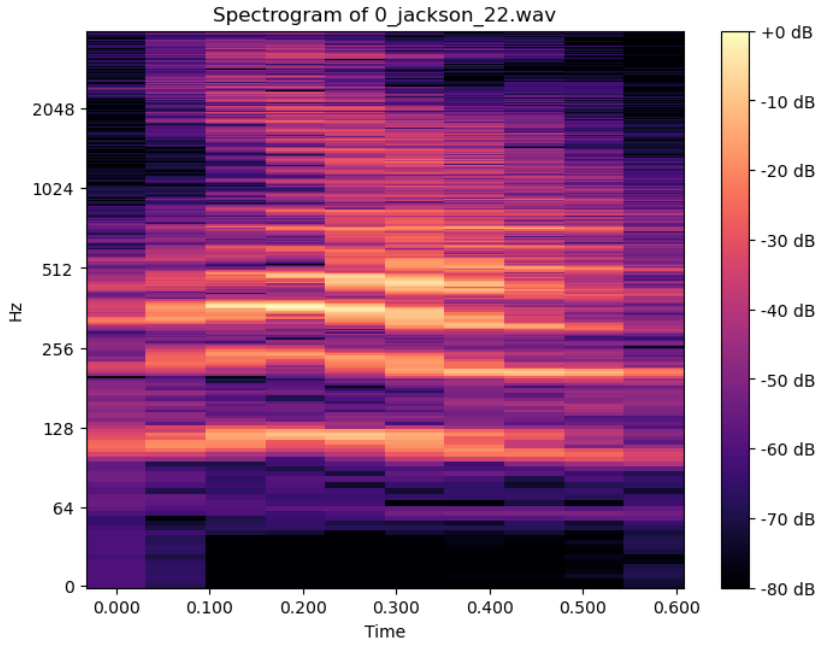
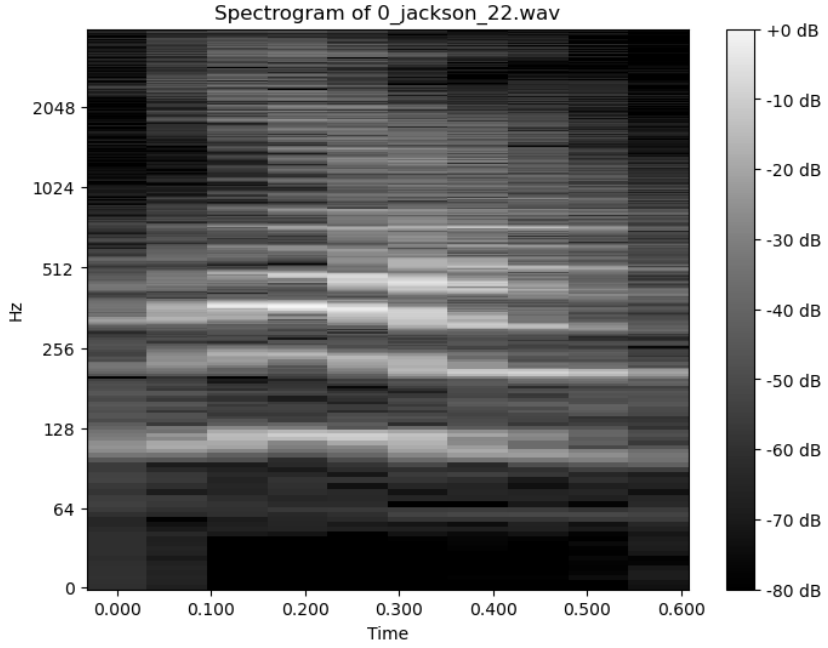


Figure 17 Example of MNIST dataset (mdpi).

FSDD is the dataset of spoken digits. It contains 3,000 recordings, 50 of each digit from 6 different speakers. Spectrograms from this recording are taken to train the model.





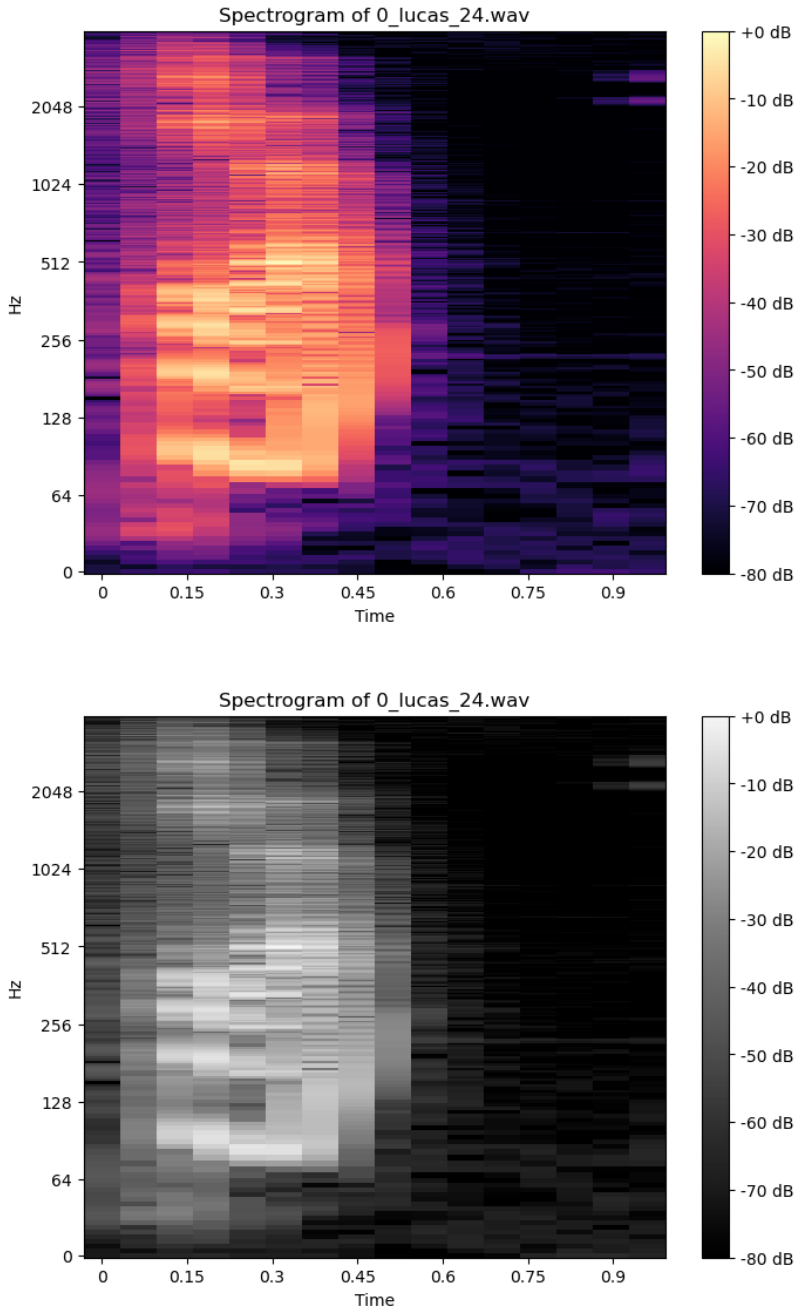


Figure 18 Comparison between the RGB and Grayscale spectrogram of word zero from different users.

Above are the examples of the spectrograms that shall be taken as input for the model. Speaker “Jackson” and “Lucas” have spoken the same word “Zero” and the change in frequency spectrum can be seen in both cases. With 4 other speakers, such changes in spectrum makes FSDD, a good source for spectrogram extraction.

3.2 Building and training the model

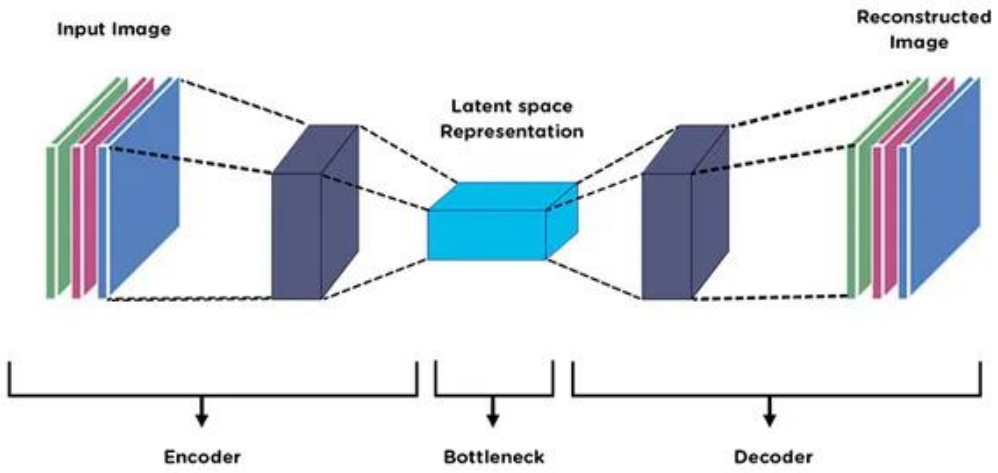


Figure 19 Architecture of an autoencoder model (Researchgate).

Building an autoencoder model involves constructing a neural network architecture that consists of an encoder, a decoder, and a latent space representation. The input image is passed through the layers in neural network and the essential features of the inputs are extracted. This final output of encoder is then encoded in the latent space. The dimensionality of the latent space can be chosen according to the model. Higher dimensionality can improve the model but can make model sizable. This encoded extracted feature is then passed through the decoder layer. This layer is the mirror image of the neural network in encoder layer. As the model gets trained it saves and updates the weights and the parameter of the model. To make the model stable, these weights and parameters should not fluctuate wildly. To ensure this stability, Batch normalization technique is applied. The final output takes this weight and parameter and generates a similar yet different image.

Autoencoder aims to learn efficient representation of the input data in an unsupervised manner. A learning rate is then set for the model during training the model. In this model, this rate is still fixed throughout the process. During each training iteration(epoch), the model makes predictions on the training data. A mathematical function quantifies this error between the prediction made by the model and the actual target values on the training data set. This function is generally termed as loss function or cost function. In this thesis, MSE(Mean squared error) is taken as the loss function. (Foster, 2019)

Loss functions are of critical importance since they evaluate the model and need to be minimized. Values of weights and parameters need to be adjusted so that the loss function can be minimized (Nielsen, 2015) To achieve these best values, an optimization algorithm is used. Gradient descent ,Parameter update and stochastic descent are some of the common algorithms . In this thesis Adam is taken as the optimization algorithm.

Adam (Adaptive Moment Estimation) is an optimization is a stochastic gradient descent method that updates the learning rate adaptively. It is built into the Keras library as one of the available optimization algorithms for training neural networks (Kingma & Ba, 2014)

3.2.1 encoder and decoder

Model: "encoder"

Layer (type)	Output Shape	Param #
encoder_input (InputLayer)	[(None, 28, 28, 1)]	0
encoder_conv_layer_1 (Conv 2D)	(None, 28, 28, 32)	320
encoder_relu_1 (ReLU)	(None, 28, 28, 32)	0
encoder_bn_1 (BatchNormali zation)	(None, 28, 28, 32)	128
encoder_conv_layer_2 (Conv 2D)	(None, 14, 14, 64)	18496
encoder_relu_2 (ReLU)	(None, 14, 14, 64)	0
encoder_bn_2 (BatchNormali zation)	(None, 14, 14, 64)	256
encoder_conv_layer_3 (Conv 2D)	(None, 7, 7, 64)	36928
encoder_relu_3 (ReLU)	(None, 7, 7, 64)	0
encoder_bn_3 (BatchNormali zation)	(None, 7, 7, 64)	256
encoder_conv_layer_4 (Conv 2D)	(None, 7, 7, 64)	36928
encoder_relu_4 (ReLU)	(None, 7, 7, 64)	0
encoder_bn_4 (BatchNormali zation)	(None, 7, 7, 64)	256
flatten (Flatten)	(None, 3136)	0
encoder_output (Dense)	(None, 2)	6274

Total params: 99842 (390.01 KB)
Trainable params: 99394 (388.26 KB)
Non-trainable params: 448 (1.75 KB)

Figure 20 encoder summary.

To build the neural network, Python is taken as the programing language. With Keras and TensorFlow as library and backend API.

Figure 20 shows the encoder architecture. The first layer is the input layer, and this layer is passed through four convolution layers in sequence. Each sequence captures the feature and passes on to another layer. The last convolutional layer is then flattened and connected to a Dense layer of size 2. This represents the two-dimensional latent space.

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   [(None, 2)]               0

decoder_dense (Dense)        (None, 3136)              9408

reshape (Reshape)            (None, 7, 7, 64)          0

decoder_conv_transpose_lay   (None, 7, 7, 64)          36928
er_1 (Conv2DTranspose)

decoder_relu_1 (ReLU)        (None, 7, 7, 64)          0

decoder_bn_1 (BatchNormali   (None, 7, 7, 64)          256
zation)

decoder_conv_transpose_lay   (None, 14, 14, 64)        36928
er_2 (Conv2DTranspose)

decoder_relu_2 (ReLU)        (None, 14, 14, 64)        0

decoder_bn_2 (BatchNormali   (None, 14, 14, 64)        256
zation)

decoder_conv_transpose_lay   (None, 28, 28, 64)        36928
er_3 (Conv2DTranspose)

decoder_relu_3 (ReLU)        (None, 28, 28, 64)        0

decoder_bn_3 (BatchNormali   (None, 28, 28, 64)        256
zation)

decoder_conv_transpose_lay   (None, 28, 28, 1)         577
er_4 (Conv2DTranspose)

sigmoid_layer (Activation)   (None, 28, 28, 1)         0

=================================================================
Total params: 121537 (474.75 KB)
Trainable params: 121153 (473.25 KB)
Non-trainable params: 384 (1.50 KB)
_________________________________________________________________
```

Figure 21 decoder summary.

The first layer in the decoder is the Dense layer of size 2. This layer is reshaped and passed through four transposed convolution layers. In each transition, the model uses the updated weights and parameters to increase the spatial resolution and finally an image.

Finally, these sections are merged to form an “autoencoder” model. In Keras and TensorFlow, it can be done by defining the model with encoder input and decoder output.

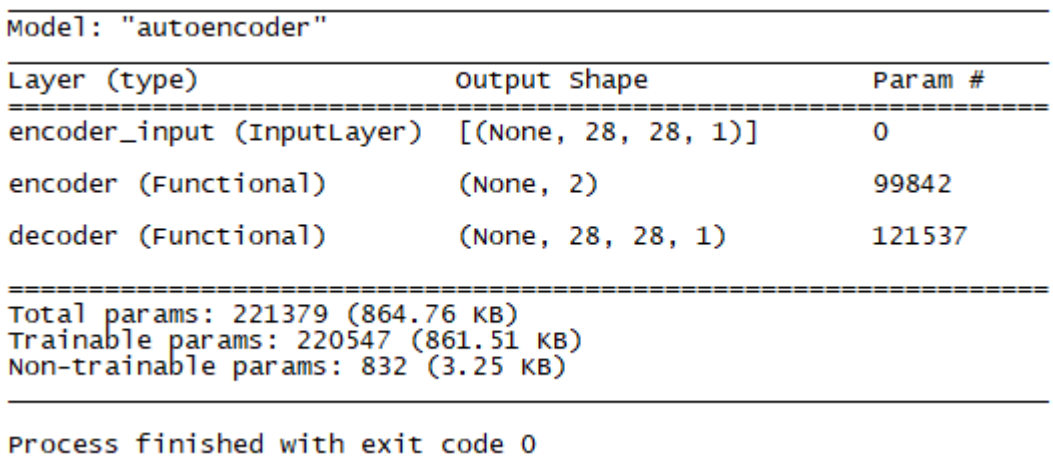


Figure 22 autoencoder summary.

This final autoencoder model is then compiled with an optimizer along with learning rate to keep the loss function minimum. Adam is taken as the optimizer with a learning rate of 0.0001, to keep the model stable.

3.2.2 Training the model with MNIST dataset.

After the model compilation, training of the model is done. MNIST dataset can be imported from the tensorflow. keras.dasets. Training data and testing data in MNIST are fixed and predefined, 60,000 and 10,000 respectively.

When working with large sets of data, the data is divided into batches. In this training session, the data is divided into 32 batches. When the model goes through a batch and updates the weights and parameters, it’s called one Iteration. And one complete pass through of the training set is called an epoch or number of training steps. Batch size and epoch number are critical hyperparameter for a neural network. Smaller batch sizes can lead to more frequent weight updates and faster convergence but may require more iterations. Larger batch sizes can train the data fast but may result in less weight updates. Similarly, training for less epochs or too many epochs can result in underfitting and overfitting of the model. With MNIST data set, only training data is taken without testing, as it is a generative model.

For training, following values were taken for:

BATCHLEARNING_RATE = 0.0001

BATCH_SIZE = 32

EPOCHS = 100

The autoencoder was then trained with 10,000 training data,

autoencoder = train(X_train[:10000], LEARNING_RATE, BATCH_SIZE, EPOCHS)

```
Epoch 97/100
313/313 [==============================] - 68s 217ms/step - loss: 0.0366
Epoch 98/100
313/313 [==============================] - 71s 228ms/step - loss: 0.0365
Epoch 99/100
313/313 [==============================] - 72s 231ms/step - loss: 0.0365
Epoch 100/100
313/313 [==============================] - 71s 226ms/step - loss: 0.0364

Process finished with exit code 0
```

Figure 23 Training summary.

Training is undeniably an important part of the autoencoder model. The more the model gets trained, the better the output. Keras and TensorFlow are powerful and versatile deep learning frameworks that provide a high degree of flexibility and capabilities. Such features come at the cost of increased complexity and computational demands. To accelerate the training process and for better results, GPUs and TPUs are recommended. Newer versions of Keras and TensorFlow supply more features, Bug Fixes, Stability and are more compatible to third-party libraries and tools.

3.3 Reconstruction and Analysis

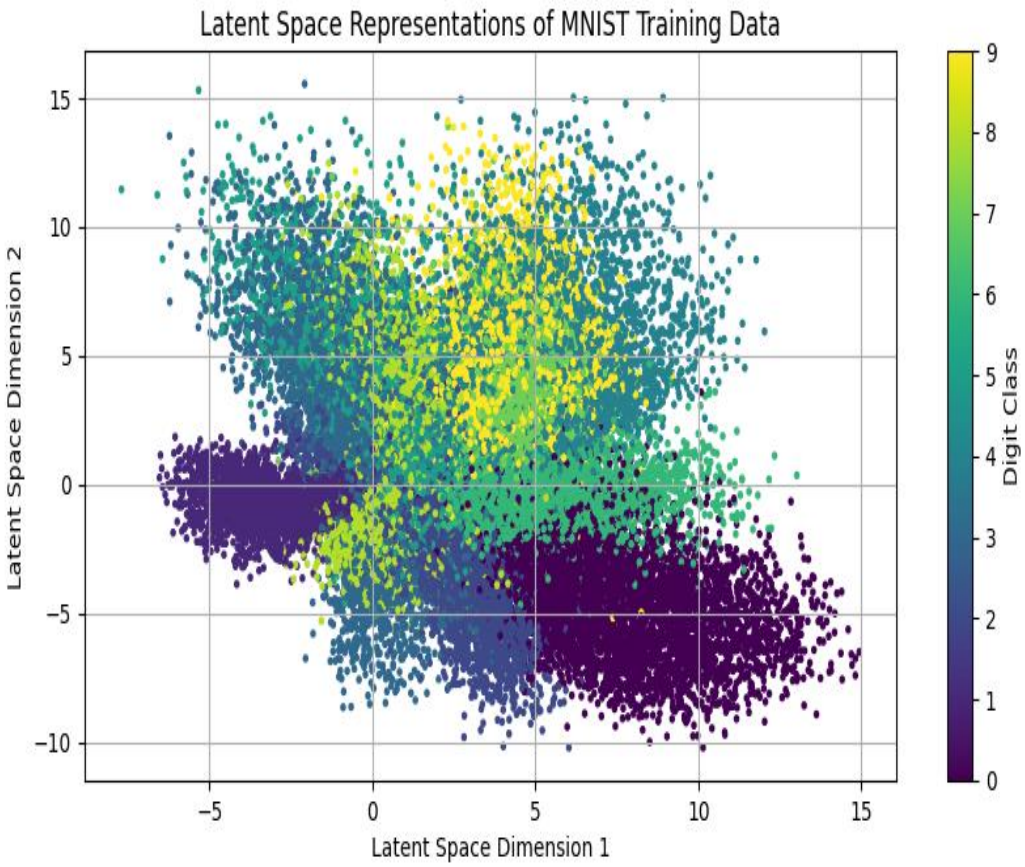


Figure 24 Latent space representation with digit classification 0-9.

Above is the latent space representation of the model. The model is trained with 10,000 inputs, i.e., number of encoded and decoded information.

Autoencoder architecture has one to one relationship between the input and output. Reconstruction is done by a prediction function from Keras. When the specific input is chosen for the reconstruction, it is like pointing at a color in the given spectrum. Let us use the number 7 as the input, which lies in greenish spectrum. The decoder now takes a point from this spectrum and uses its weight and parameters to generate the output. Which is a generated image of the input. The input index is taken from the training sets. Below are the examples of various inputs and their generated images.

- First 5 sample input

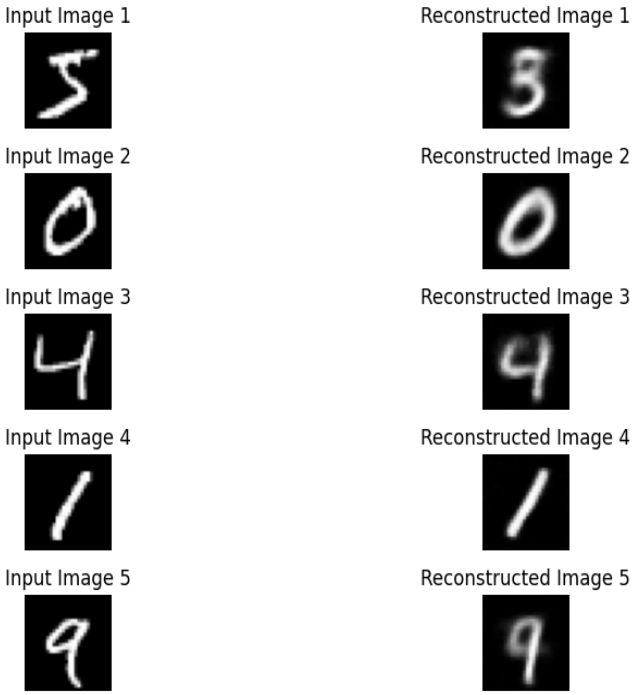


Figure 25 First 5 input sample and their generated images.

- Randon input samples.

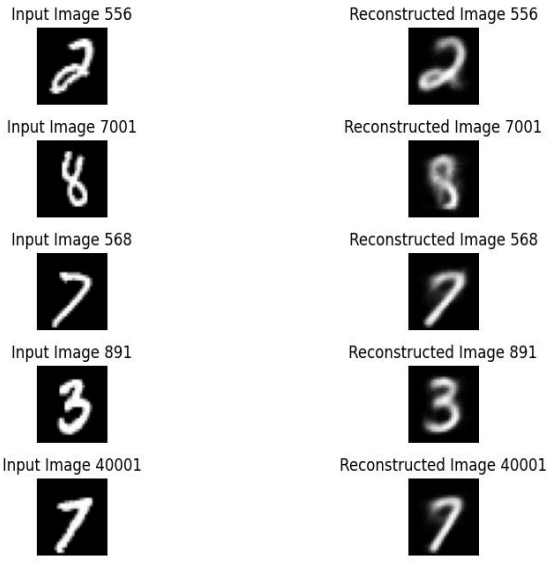


Figure 26 Generated images for random input.

3.4 Training with spectrograms.

Training with the spectrograms is rather a complicated process. Unlike MNIST dataset, where there are only numbers. With the spectrograms, the whole image is pixelated. When the input is passed through the convolution layer in the autoencoder model. It gets filtered through the layer and the features are extracted. But what are these features in audio samples? The pixels are assigned some value in spectrogram. The distribution of these values can be understood as the feature of an audio. The spatial structures along the pixel intensity are responsible for change in tone and pitch.

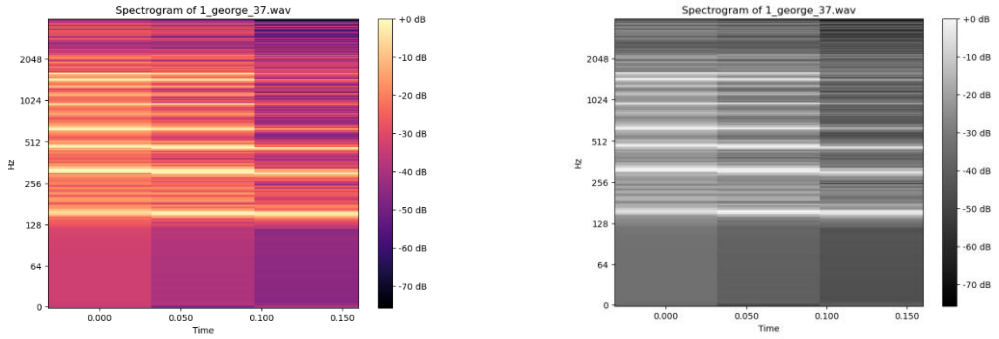


Figure 27 RGB and Grayscale image of word "One".

Above is a conversion of RGB into Grayscale image. The image conveys the information about the frequency distribution, and it does so by assigning the highest neuron value to higher frequency and lower neuron value to lower frequency. The image is then normalized by Min-Max Scaling (Sebastian, 2014),

$X(normalized) = \frac{X - Xmin}{Xmax - Xmin}$,

X (max) and *x(min)* 1 and 0 set respectively.

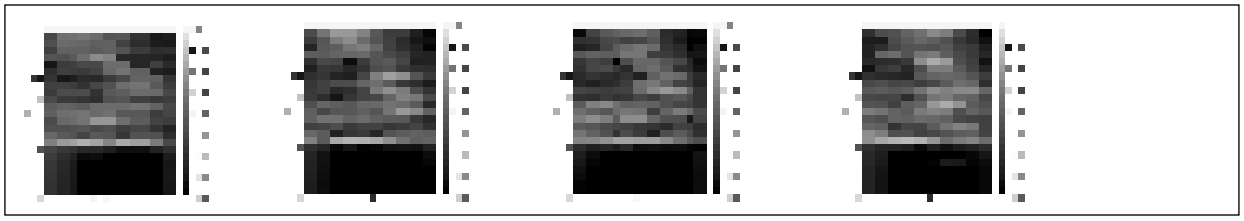


Figure 28 Compressed spectrogram in 28*28-pixel dimension.

A total of 800 samples were taken from the FSDD data set. Figure 27 represents the first five compressed and normalized audio samples before training. The training process in this thesis is performed in steps, started with 1400 and increased 100 at a time to avoid the overfitting of the data. The following parameters were used:

Learning rate=0.0005

Batch size=10

Epochs=1400.

[0, 1, 2] indices from X-train:

Original:

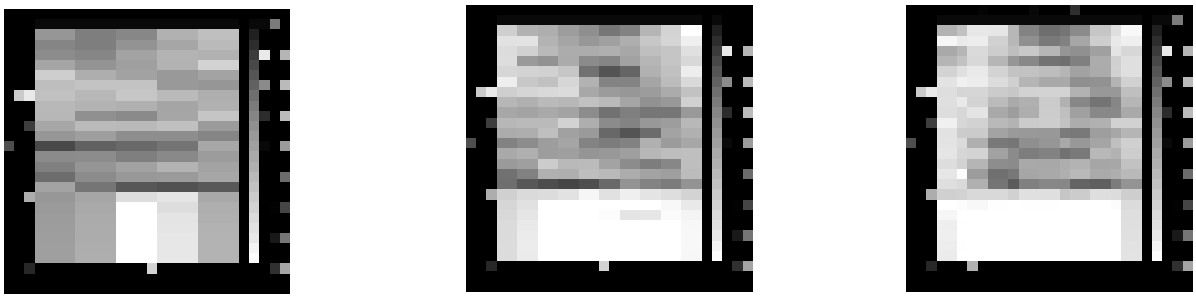


Figure 29 [0, 1, 2] ordered input from X-train.

Reconstructed:

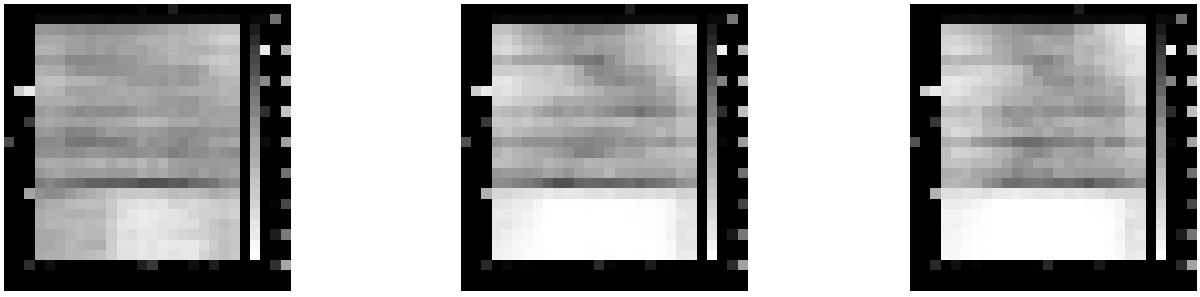


Figure 30 Reconstructed image from [0, 1, 2] input.

Epochs=1600

[1, 2, 3] indices from X-train:

Original:

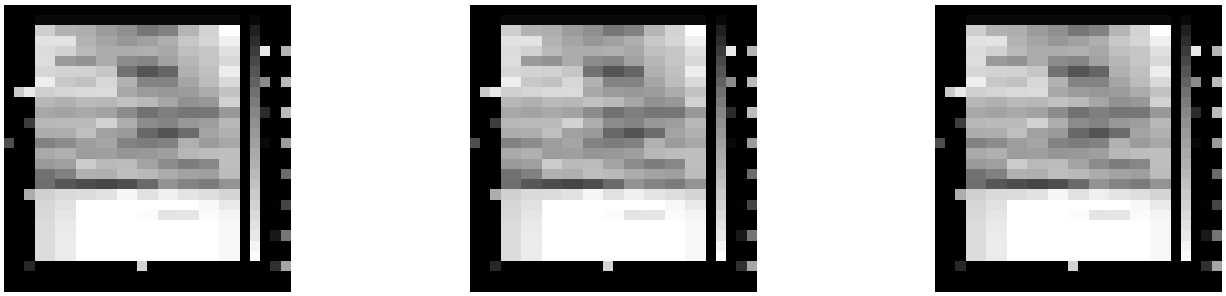


Figure 31 [1, 2, 3] ordered input from X-train.

Reconstructed:

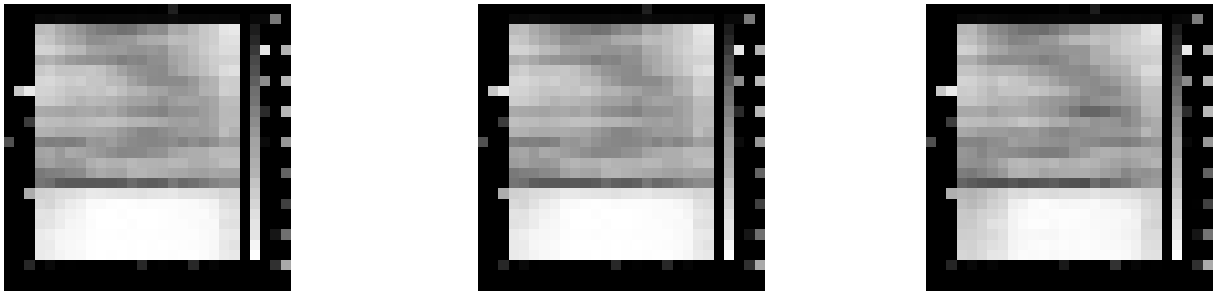


Figure 32 Reconstructed image from [1, 2, 3] input.

Epochs=1700

[2,3,4] indices from X-train

Original

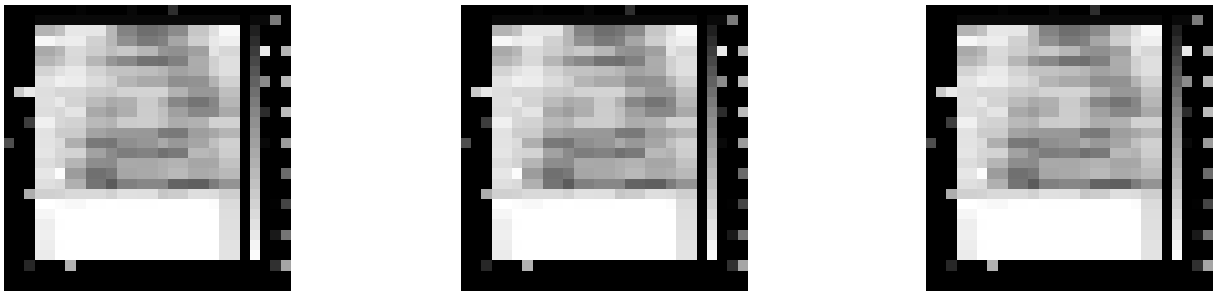


Figure 33 [2, 3, 4] ordered input from X-train.

Reconstructed:

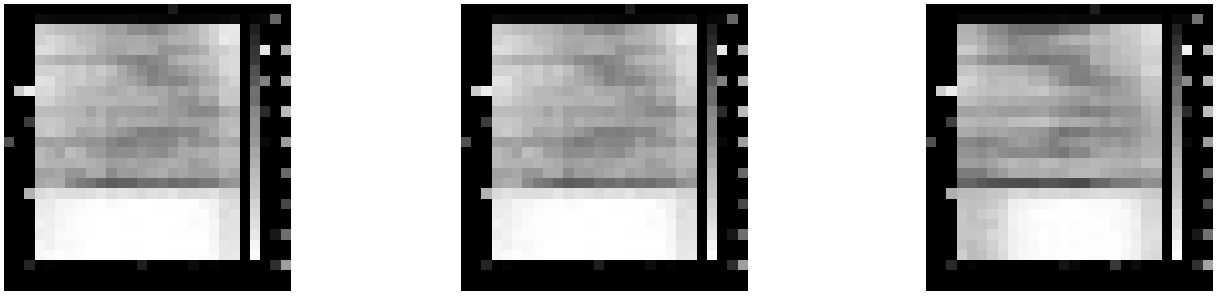


Figure 34 Reconstructed image from [2, 3, 4] input.

Epochs:2000

[5,6,7] indices from X-train

Original:

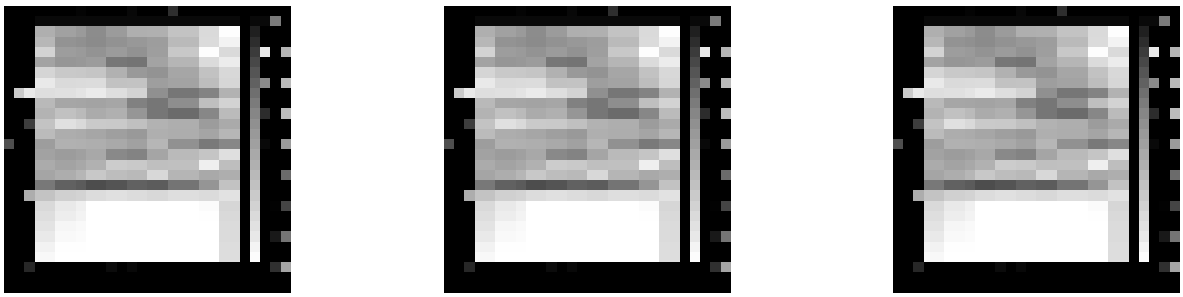


Figure 35 [5, 6, 7] ordered input from X-train.

Reconstructed:

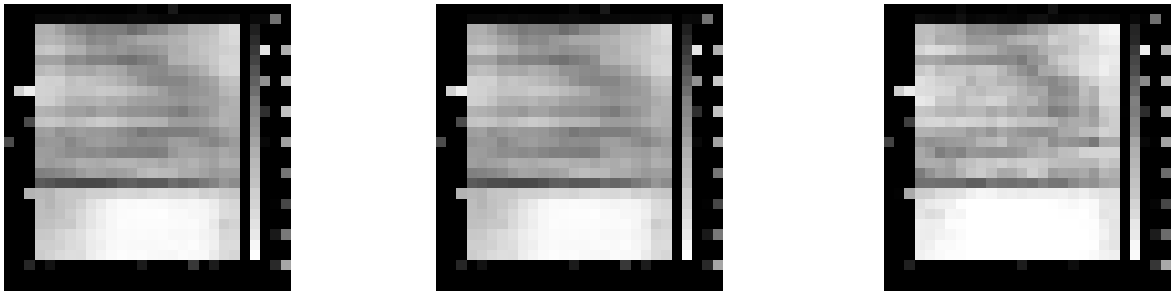


Figure 36 Reconstructed image from [5, 6, 7] input.

Epochs:2100

[12,13,14] indices from X-train

Original:

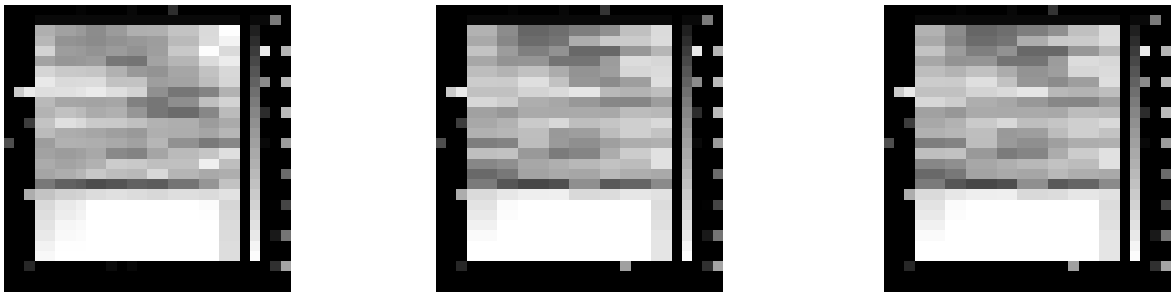


Figure 37 [12, 13, 14] ordered input from X-train.

Reconstructed:

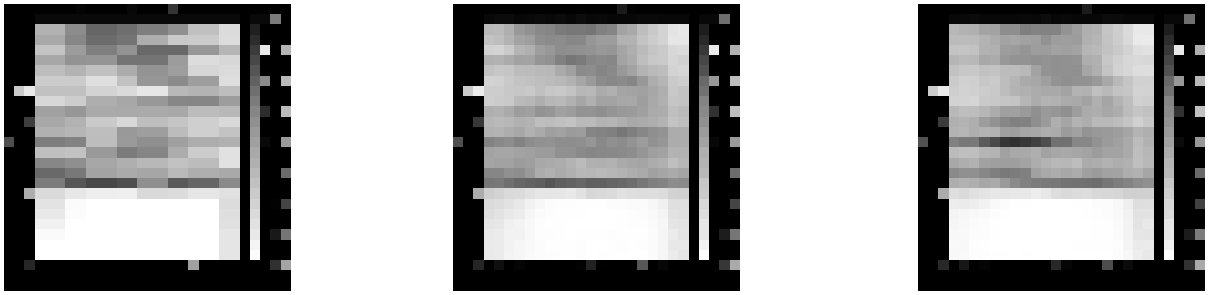


Figure 38 Reconstructed image from [12, 13, 14] input.

Epochs:2500

[15,16,17] indices from X-train

Original:

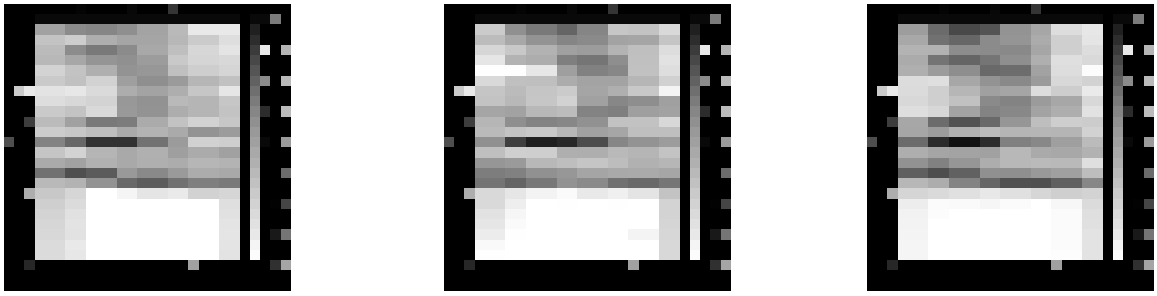


Figure 39 [15, 16, 17] ordered input from X-train.

Reconstructed:

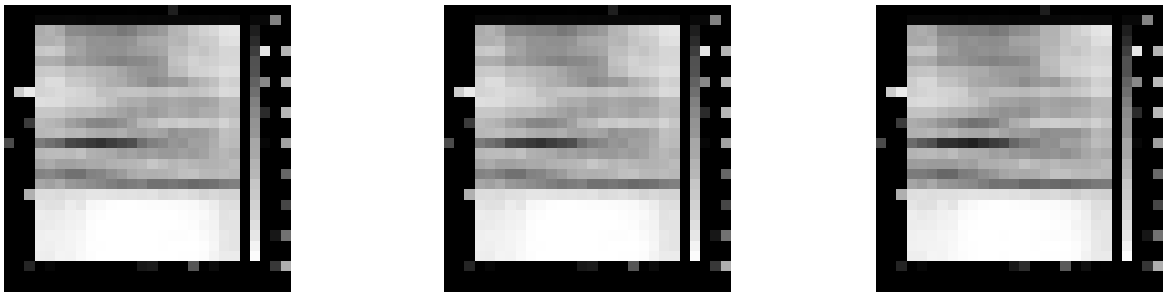


Figure 40 Reconstructed image from [15, 16, 17] input.

And the Latent Space representation,

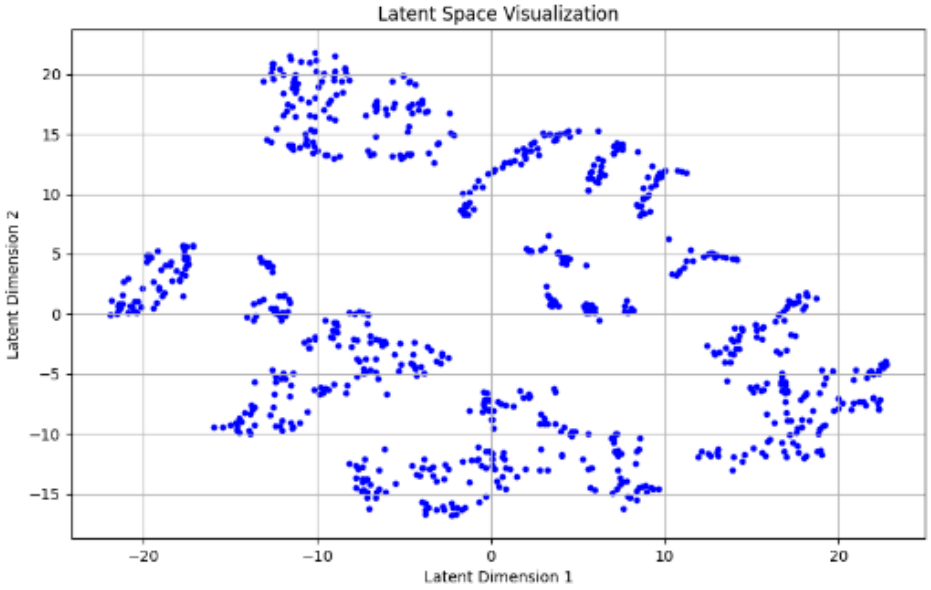


Figure 41 Latent space representation of spectrogram data.

Figure 41 is the latent space representation of the model with a total of 800 encoded samples. Every dot is the input in latent space. The final image is generated with the weights and the parameters. These images need to be further processed into meaningful sound waves. This transformation is discussed in inverse short time Fourier transform.

3.5. Inverse short time Fourier transform

The reconstruction from spectrogram involves one more step than from MNIST's. The final generated image is a spectrogram image. But it needs to be converted into an audio wave form. This can be done by inverse short time Fourier transform (audiolabs-erlangen, 2015)

The inverse short time Fourier transform is the process of reconstructing the time domain signal. The frequency-time domain *X[k,m]* is deconstructed by the same window function with the same parameters.

From equation 3,

$$X[k,m] = \sum_{n=0}^{N-1} x[n+mH] \cdot w[n] \cdot e^{-j2\pi kn/N} \ldots .5$$

Inverting the window function,

$$X[n+mH] = \sum_{k=0}^{N-1} x[k,m] \cdot w[n] \cdot e^{j2\pi kn/N}$$

Expressing only in terms of *x[n]*,

$$X[n] = \sum m(\sum_{k=0}^{N-1} x[k,m] \cdot w[n-mH] \cdot e^{j2\pi kn/N}) \quad \ldots\ldots\ldots 6$$

Which is the final processed audio signal as output,

In this thesis, for comparison ,four ISTFT is done,

1. With 28*28-pixel image.

The 12th reconstructed image after 2100 epochs is taken as an example,

Hop length:54.

Win length:100

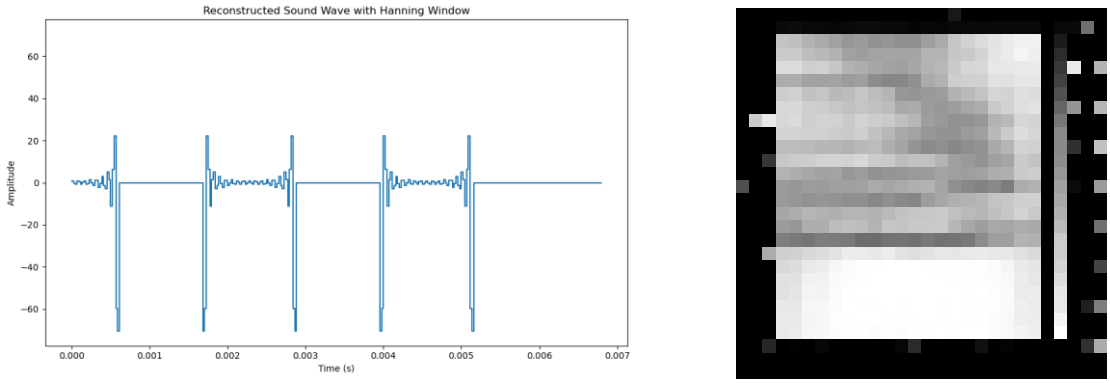


Figure 42 ISTFT from 28*28-pixel image.

2. With 690*545 pixels image

The same image is then enlarged and the ISTFT is performed.

Hop length:450.

Win length:1088.

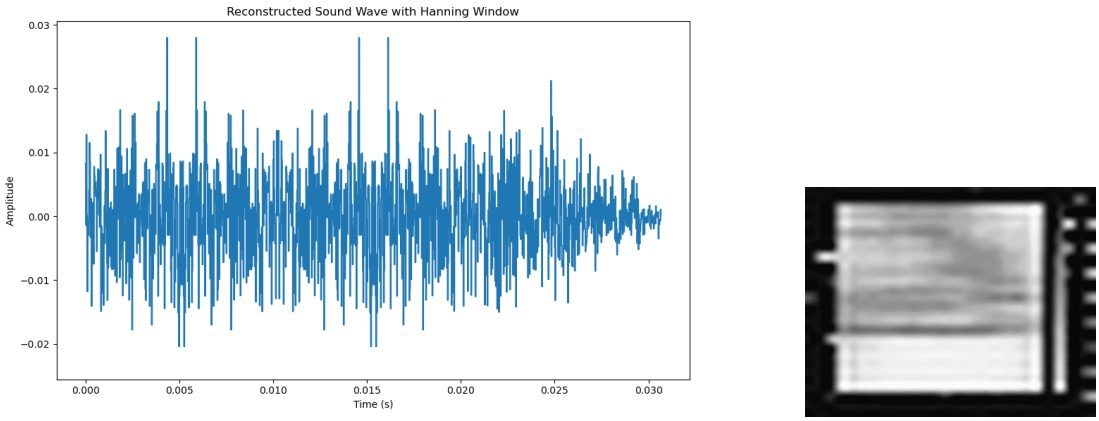


Figure 43 ISTFT from 690*545 pixels image.

3. With the same image 690*545 after 2500 epochs

Hop length:450.

Win length:1088

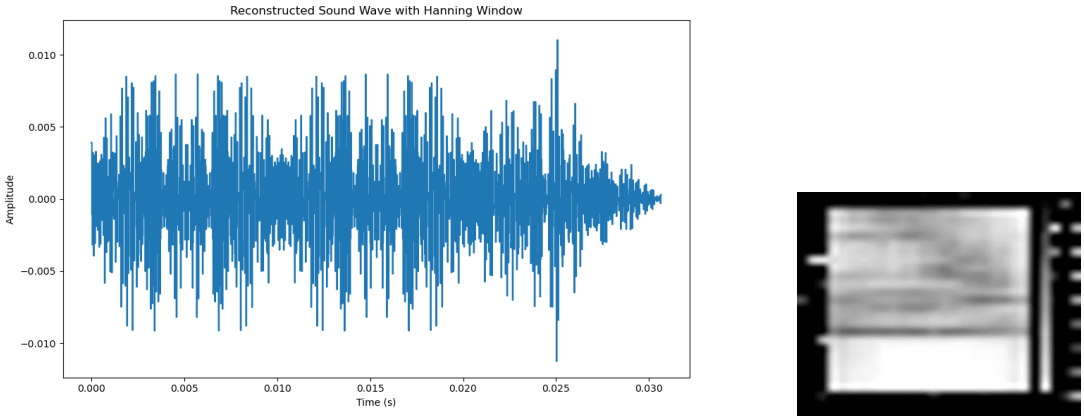


Figure 44 ISTFT from 690*545 pixels image.

4. With the original 12th sample from recording sample.

Hop length:450.

Win length:1088.

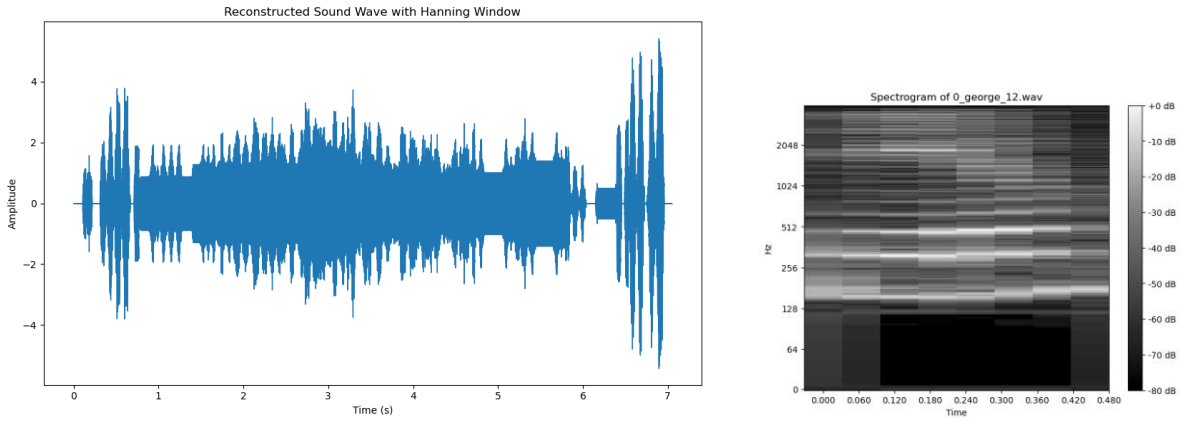


Figure 45 ISTFT from 690*545 pixel of the original image.

Four ISTFT were done for the comparison between the image quality Better the resolution, the window function can encompass more samples and better audio signal can be generated.

Autoencoder model was used to generate the model. The input was changed from normal image to Spectrogram. As seen in the ISTFT figure, the sound wave is not as good as the one from original one. There are several factors that affect and reduce the quality of final audio wave. Data and model used are of great importance, but some other techniques can be employed for better results. This is discussed in the Result and Discussion section.

4 Variational Autoencoder:

The Variational Autoencoder(VAE) is a generative deep learning model, which works on the principles of Autoencoders. The model was first introduced in 2013 by Kingma and Welling in 2013(ref). This section focuses on the working architecture of the model, how it is different from Autoencoders model.

4.1 VAE architecture.

Variational autoencoder like autoencoder is composed of an encoder a decoder and a latent space representation. Unlike traditional autoencoders, VAE model uses a probabilistic approach for method of data encoding Unlike Autoencoders which encodes image as univariate, In VAE each image is mapped to a multivariate normal distribution around a point in the latent space.

The normal distribution, also known as the Gaussian distribution or bell curve, is a fundamental probability distribution in statistics and probability theory. It is characterized by its symmetric, bell-shaped curve when plotted, and it's defined by two parameters: the mean (μ) and the standard deviation (σ). The probability density function (PDF) of the normal distribution is given by the formula,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad 6$$

Here,

f(x) is the probability density function representing the probability of a random variable *x* taking a specific value.

μ and σ are the mean and standard deviation of the data spread (Moore, McCabe, & Craig, 2019)

These multivariate parameters modeled with autoencoder model is the Variational autoencoder model. When a VAE model is trained, the input is passed through the convolutional layers just like Autoencoder. But unlike autoencoder, where the encoder's

output is in the Dense layer, the encoder's output in the VAE is these multivariate ,i.e., mean, and standard variance.

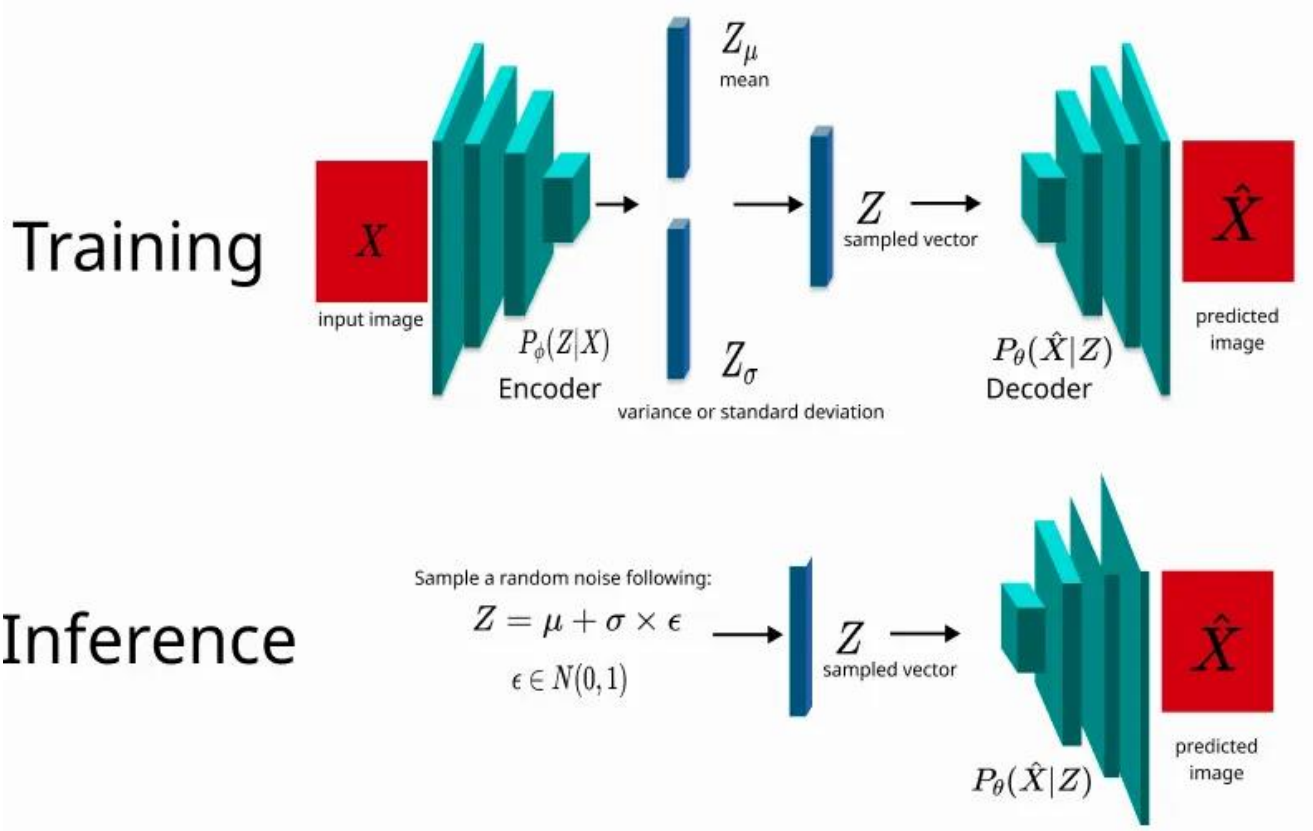


Figure 46 Schematic representation of Variational autoencoder *(towardsdatascience)*.

Above is the VAE model representation. The inference part is what makes this a better, generative, and probabilistic model. The input is flattened and divided into its multivariate format, mean and variance.

The next step is to make a normal distribution function from these multivariate. This distribution must be differentiable with respect to these multivariate. The technique to make such a probability distribution function d with these parameters is called reparameterization. In Keras, this can be done by applying lambda layer. This layer wraps the two parameter and sample a point Z from normal distribution with mean and variance, as

$Z = \mu + \varepsilon . \sigma$

ε is the standard normal distribution

Reparameterization is particularly important because it ensures efficient and differentiable training. It also makes the latent space continuous, which is the key feature for generation in VAE (Foster, 2019)

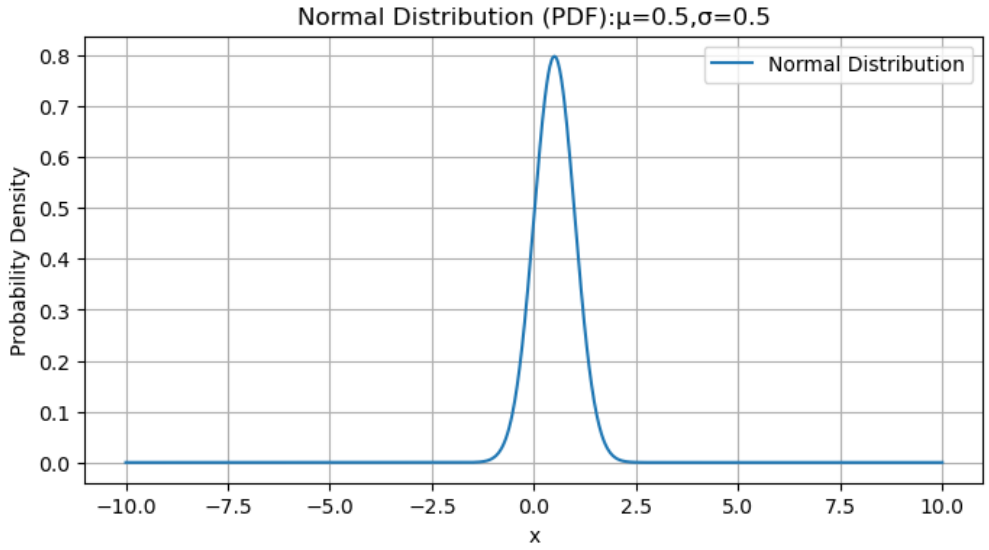


Figure 47 Normal distribution with 0.5 mean and 0.5 Standard deviation.

Above is the normal distribution with mean 0.5 and standard deviation 0.5. This distribution function can be taken as one of the possible distribution functions, which can be made from the multivariate

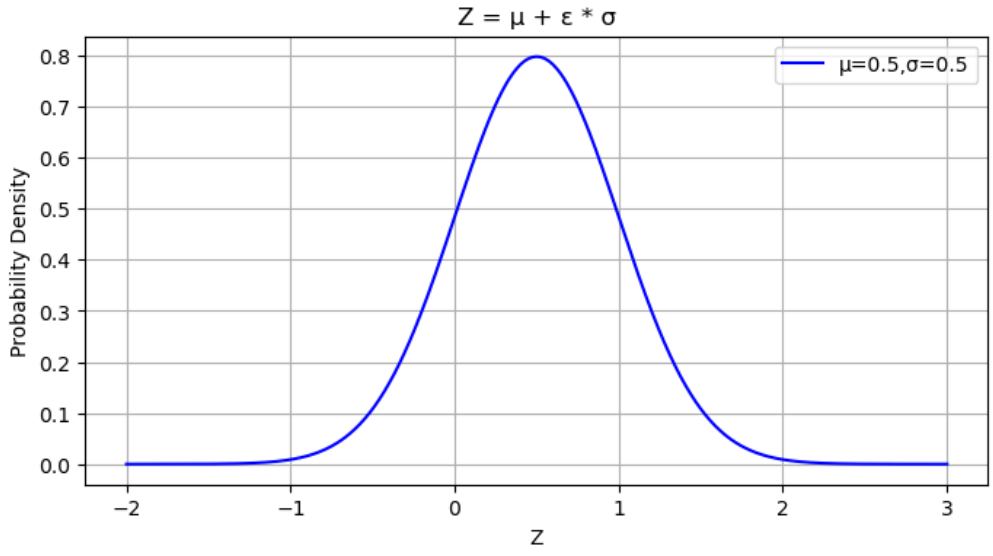


Figure 48 Normal distribution after reparameterization.

Figure shows the normal distribution after reparameterization with the same mean and standard deviation. Every multivariate is wrapped with this layer and transformed into such distribution. This distribution then represents the mapped encoded image to a normal distribution around a point in the latent space.

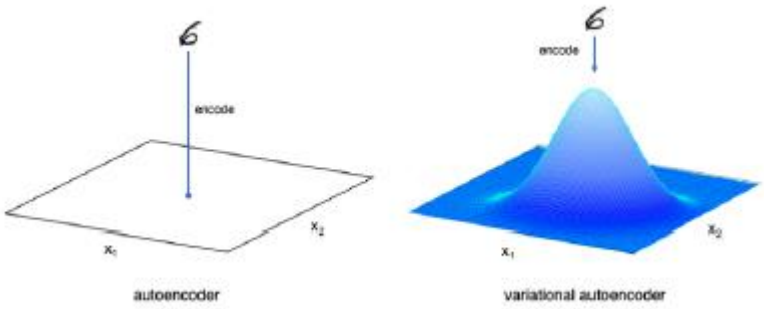


Figure 49 Latent space representation of an input in VAE model (towardsai).

Figure 49 represents the encoding of input from MNIST dataset into the latent space of the VAE model.

4.2 Loss function in VAE

Loss function is VAE plays a vital role in overall model. VAE is a generative deep model. The generated output needs to be meaningful. Loss function in VAE consists of two components: the reconstruction loss and the regularization term.

- Reconstruction loss: This measures the dissimilarity between the input and the data generated by the model. Mean squared error , binary cross-entropy loss can be taken to measure this loss.
- Regularization term: In addition to the reconstruction loss, one extra component is added in VAE. This is known as Kullback-Leibler (KL) divergence.

Kullback-Leibler divergence:

Just like the reconstruction loss, KL divergence is the dissimilarity between two functions. To calculate the KL divergence, a target normal distribution, usually standard, is defined. KL divergence then gives the difference between the transformed normal distribution in latent and this defined target standard normal distribution (MacKay, 2003)

The mathematical notation of the KL divergence between the probability functions is given by:

$$D_{KL}\big(N(\mu,\sigma)\,|\,N(0,1)\big) = \frac{1}{2}(1 + \log(\sigma^2) - \mu^2 - \sigma^2) \ldots\ldots\ldots\ldots\ldots 7$$

Here, $N(\mu,\sigma)\,|\,N(0,1)$ are the transformed and the standard normal distribution respectively, and D_{KL} is the measure of the difference.

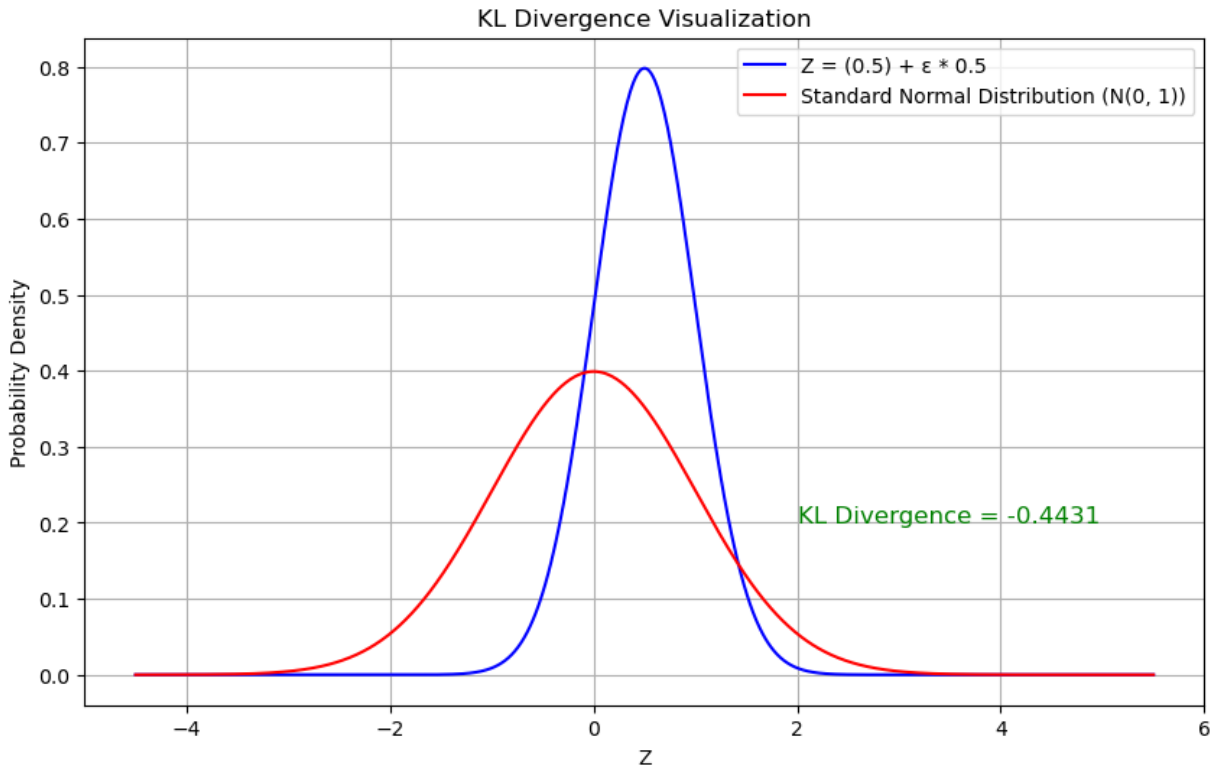


Figure 50 KL divergence between two probability densities.

The application of the KL divergence is shown in figure 50. Red is the standard normal distribution and blue is the transformed normal distribution in the latent space with mean 0.5 and standard deviation 0.5. The KL divergence between the two functions is -0.4431. This divergence imposes a restriction on which sensible parameter sets can be chosen to generate correlated and right data. The latent space of the VAE is smooth, and when any point is chosen in the latent space it generates a new image with the enclosed parameters and at the same keeping the loss function in account.

Due to its probabilistic model and unique loss function, VAEs are better than traditional vanilla autoencoders. Being a generative model, it needs to be trained with a lot of data to generate correct and better images.

5 Result and Discussion:

Generative neural networks like VAE and autoencoders are one of the important deep neural networks in Machine learning. Data can be changed, generated and many other operations can be performed by such generative model.

When the model is trained, it should not be overtrained, the epoch needs to be right enough to avoid the overfitting of the model.

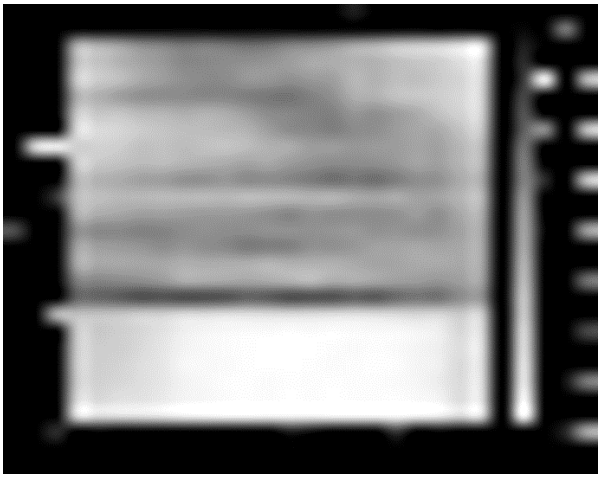
Figure 51 Reconstruction of the first 3 input sample from training set after 10,000 epochs.

Above is the example of the overfitting of the model. The model was trained 10,000 times and the samples were reconstructed. To avoid the over and underfitting of the model, the epochs were gradually increased. Below are the reconstructed audio samples from different epochs for visual comparison. Same sample from the training set is taken as an example. To visualize the difference between and after different epochs, the 5th sample from the training set is taken and the ISTFT is performed. The pixel dimension for the image is enlarged to 690*545.

After 1400 epochs:

Hop length:450.

Win length:1088.



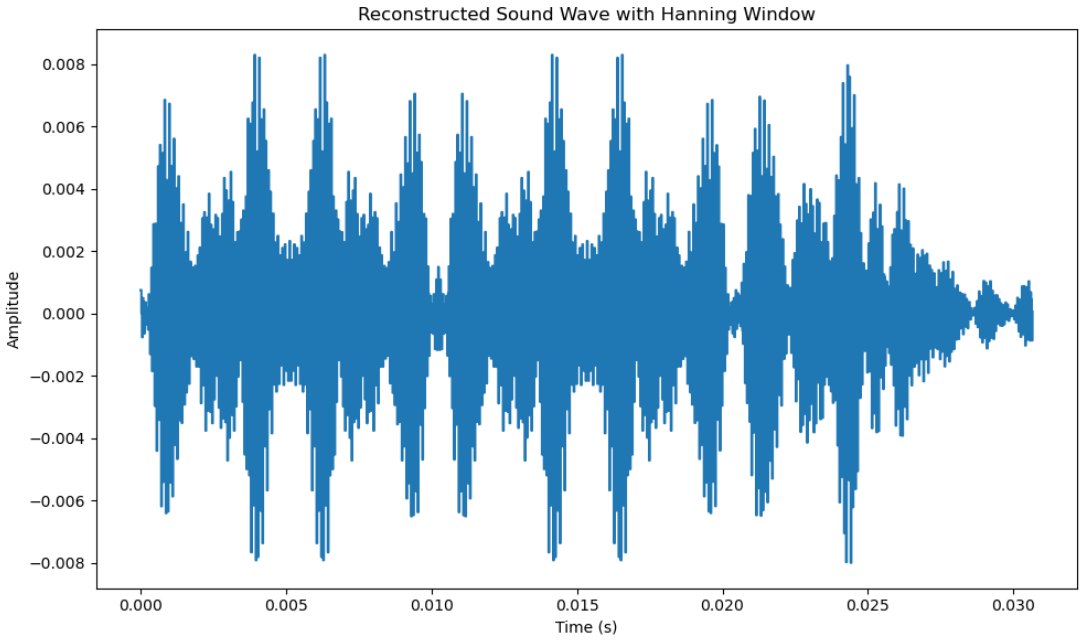


Figure 52 Spectrogram and ISTFT performed after 1400 epochs.

After 1700 epochs:

Hop length:450.

Win length:1088.

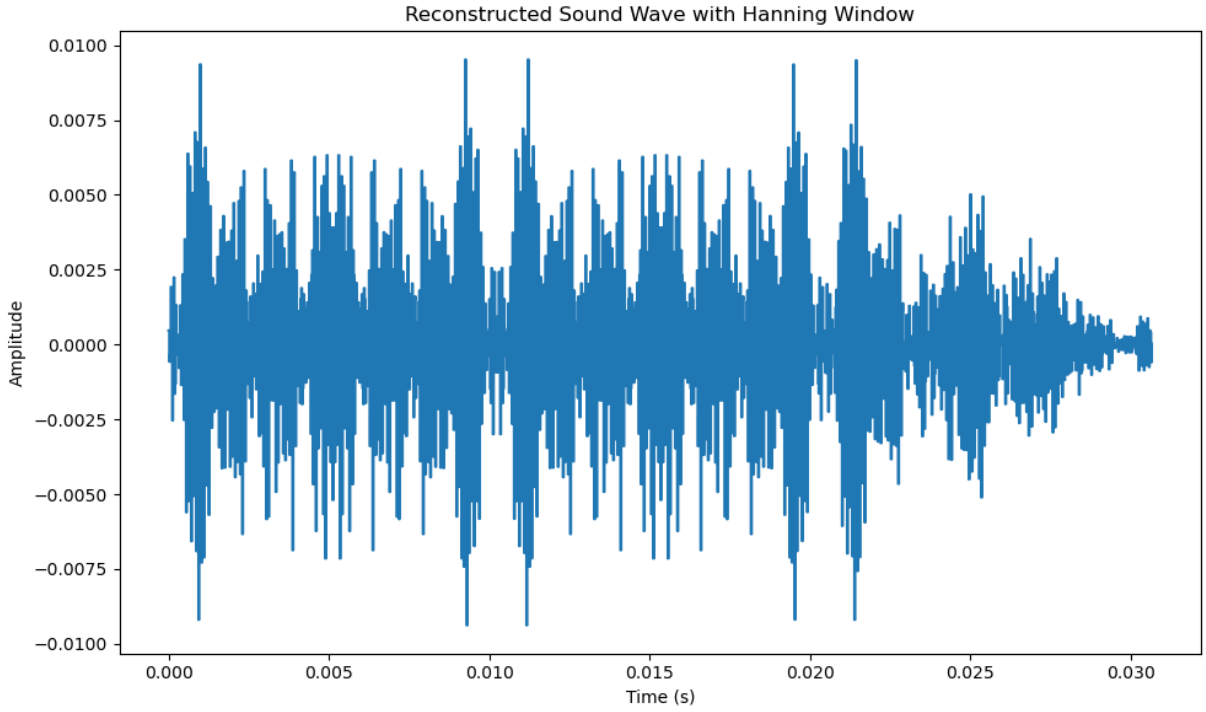
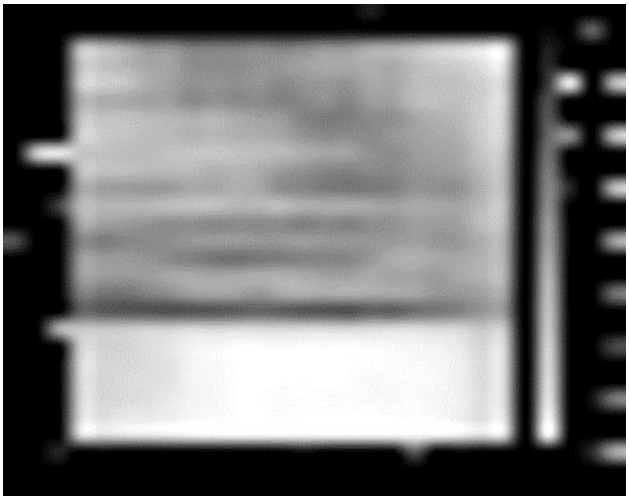


Figure 53 Spectrogram and ISTFT performed after 1700 epochs.

After 1900 epochs:

Hop length:450.

Win length:1088.



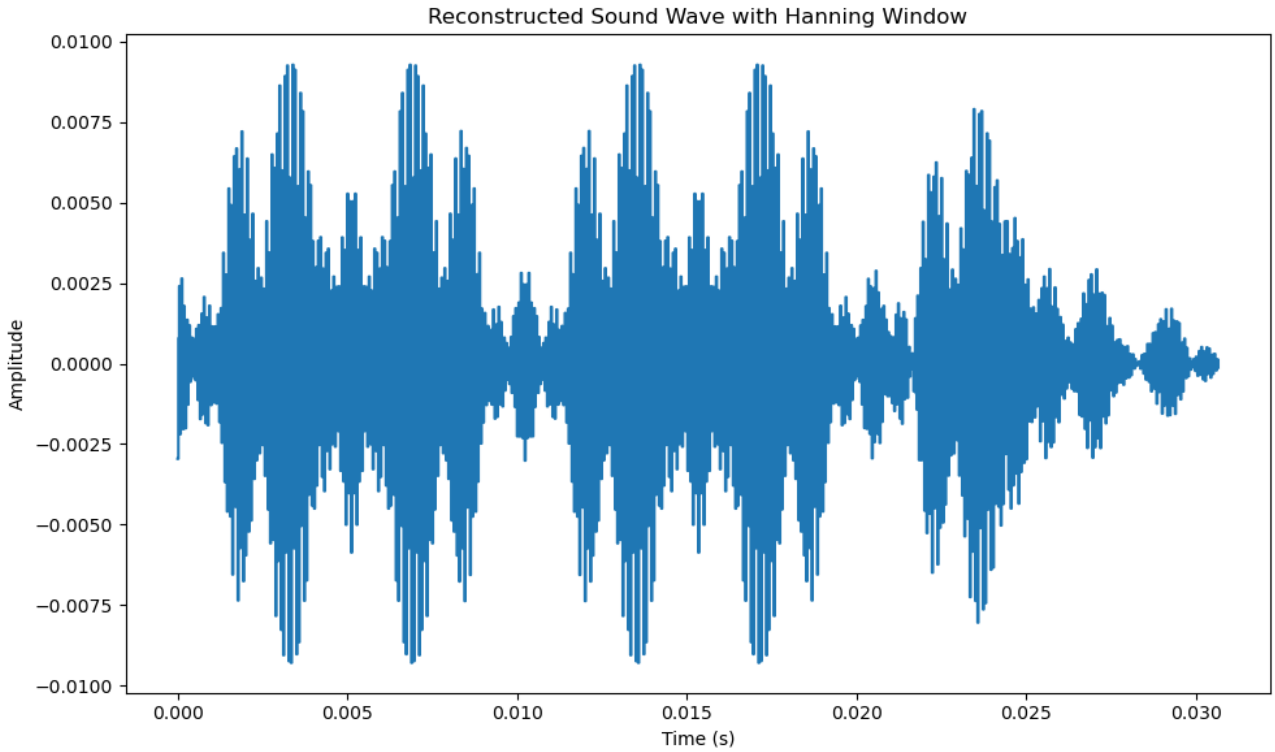


Figure 54 Spectrogram and ISTFT performed after 1900 epochs.

After 2500 epochs:

Hop length:450.

Winlength:1088.

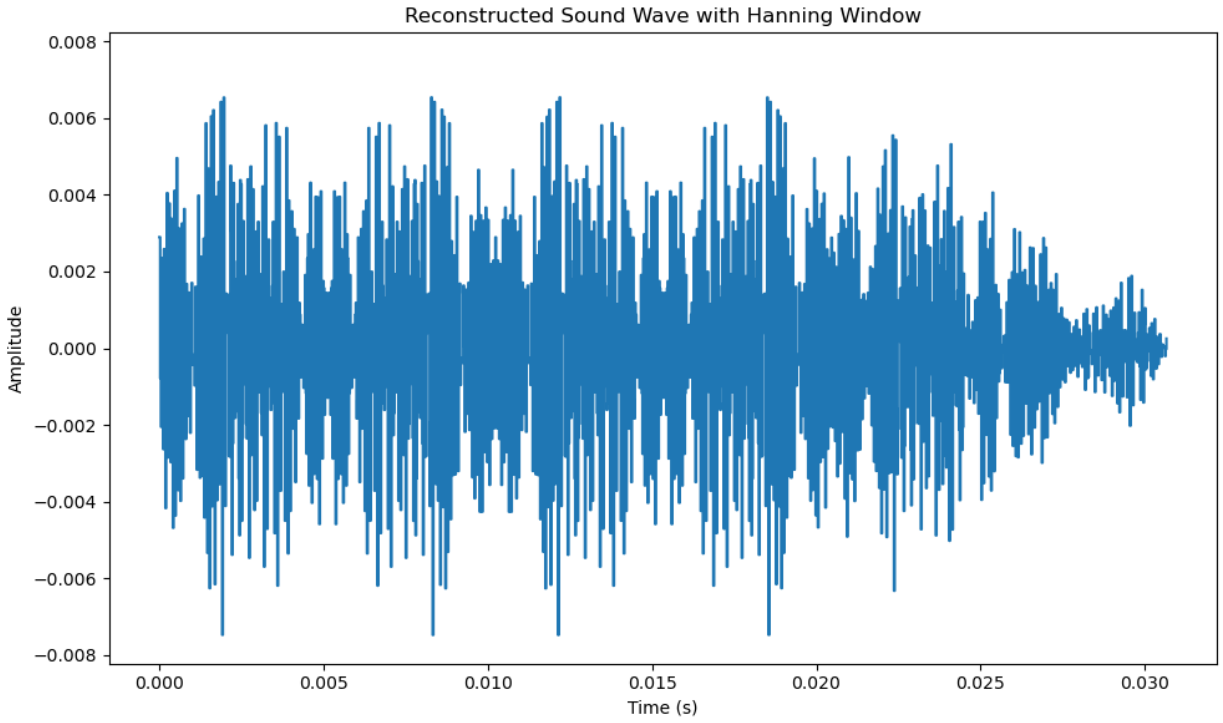


Figure 55 Spectrogram and ISTFT performed after 2500 epochs.

From the above results and ISFT it can be visually seen that the audio wave gets better after each epoch, and it encapsulates many audio samples as well. A final sound wave in audio could not be extracted as the spectrogram was pixelated heavily.

In this thesis the input is prepared from the audio samples and changed to the needs of the deep neural network. Below are some points discussed for the betterment and improvement of the overall output.

1. Data preparation:

Choice and preparation of the data is one of the most, if not the most important process in the Machine learning or neural network training. It is recommended that data needs to be robust, diverse, and large in quantity as the model gets trained with diverse and larger data sets, the problem of overfitting and underfitting can be avoided.

2. Integral transformation:

Integral transformations are the mathematical operations that involve converting a function from one domain to another domain using integral. In this thesis multiple integral transformations are performed, i.e., short time Fourier transformation and Inverse short time Fourier transform. Each transformation depends on the parameters chosen such as Hop length, Win length , Frame size etc. Optimal parameter choice for the transformation function counts as an important step.

3. Deep neural network model:

Deep neural networks are the important aspect for any machine learning project. All the feature extraction, loss function optimization and all the fine parameter tuning is done in this part. Therefore, it is recommended to choose these parameters carefully.

The size of the input sample also plays a vital role in the output of the model. In this thesis, the pixel dimension of the input is set to 28*28. Input of higher pixel dimension can improve the outcome of the model significantly, as there are more parameters for the model to train with. This makes the model more stable and reduces the biasness to a certain extent .But the size of the input comes at the computational price.

Model: "autoencoder"

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_input (Input Layer)  [(None, 690, 545, 3)]     0

 encoder (Functional)        (None, 2)                 3127874

 decoder (Functional)        (None, 692, 548, 1)       4662721

=================================================================
Total params: 7790595 (29.72 MB)
Trainable params: 7789763 (29.72 MB)
Non-trainable params: 832 (3.25 KB)
```

Figure 56 Autoencoder summary with 690*545-pixel dimension.

Above is the model representation of the autoencoder with the pixel dimension of 690*545 with three color channels in comparison with 28*28 single color channel, this model is about 35 times bigger than that. VAE are even better models than autoencoders, but it requires more computational time and specific hardware's such as GPUs TPUs.

CODES:

Spectrogram extraction:

```
import numpy as np

import matplotlib.pyplot as plt

import librosa

import librosa.display


# Load an audio file (replace with your audio file path)

audio_path = ""

y, sr = librosa.load(audio_path)


# Parameters for the spectrogram

n_fft = 6000  # Size of the FFT window (frame size)

hop_length =  200 # Hop length (overlap)

win_length = 500  # Window length


# Create a Hanning (Hann) window

window = np.hanning(win_length)


# Compute the spectrogram with the Hanning window

D = librosa.stft(y, n_fft=n_fft, hop_length=hop_length, win_length=win_length, window=window)
```

```
# Convert to dB scale
D_db = librosa.amplitude_to_db(np.abs(D), ref=np.max)

# Get the time axis and frequencies
times = librosa.times_like(D_db)
freqs = librosa.fft_frequencies(sr=sr, n_fft=n_fft)

# Plot the original audio waveform
plt.figure(figsize=(10, 4))
plt.subplot(2, 1, 1)
librosa.display.waveshow(y, sr=sr)
plt.title("Original Audio Waveform")

# Plot the Hanning window overlaid on the audio
plt.subplot(2, 1, 2)
plt.plot(np.arange(len(y)), y, label='Audio Signal', alpha=0.7)
plt.plot(np.arange(len(window)) + len(window) // 2, window * np.max(np.abs(y)) / 2, label='Hanning Window', alpha=0.7)
plt.title("Audio Signal with Hanning Window")
plt.xlabel("Sample")
plt.ylabel("Amplitude")
plt.legend()
```

```
plt.tight_layout()


# Plot the spectrogram
plt.figure(figsize=(10, 6))
librosa.display.specshow(D_db, sr=sr, hop_length=hop_length, x_axis='time', y_axis='hz')
plt.colorbar(format="%+2.0f dB")
plt.title("Spectrogram with Hanning Window")
plt.xlabel("Time (s)")
plt.ylabel("Frequency (Hz)")
plt.tight_layout()


# Show all plots
plt.show()


# Print parameters
print(f"Hop Length: {hop_length}")
print(f"Frame Size: {n_fft}")
print(f"Win length: {win_length}")
```

Autoencoder:

```
import os
import pickle
```

```
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Conv2D, ReLU, BatchNormalization, \
    Flatten, Dense, Reshape, Conv2DTranspose, Activation
from tensorflow.keras import backend as K
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import MeanSquaredError
import numpy as np


class Autoencoder:
    """
    Autoencoder represents a Deep Convolutional autoencoder architecture with
    mirrored encoder and decoder components.
    """

    def __init__(self,
                 input_shape,
                 conv_filters,
                 conv_kernels,
                 conv_strides,
                 latent_space_dim):
        self.input_shape = input_shape # [28, 28, 1]
        self.conv_filters = conv_filters # [2, 4, 8]
        self.conv_kernels = conv_kernels # [3, 5, 3]
        self.conv_strides = conv_strides # [1, 2, 2]
        self.latent_space_dim = latent_space_dim # 2

        self.encoder = None
        self.decoder = None
        self.model = None

        self._num_conv_layers = len(conv_filters)
```

```
        self._shape_before_bottleneck = None
        self._model_input = None

        self._build()

    def summary(self):
        self.encoder.summary()
        self.decoder.summary()
        self.model.summary()

    def compile(self, learning_rate=0.0001):
        optimizer = Adam(learning_rate=learning_rate)
        mse_loss = MeanSquaredError()
        self.model.compile(optimizer=optimizer, loss=mse_loss)

    def train(self, x_train, batch_size, num_epochs):
        self.model.fit(x_train,
                       x_train,
                       batch_size=batch_size,
                       epochs=num_epochs,
                       shuffle=True)

    def save(self, save_folder="."):
        self._create_folder_if_it_doesnt_exist(save_folder)
        self._save_parameters(save_folder)
        self._save_weights(save_folder)

    def load_weights(self, weights_path):
        self.model.load_weights(weights_path)

    def reconstruct(self, images):
            latent_representations = self.encoder.predict(images)
```

```
        reconstructed_images = self.decoder.predict(latent_representations)
        return reconstructed_images, latent_representations

@classmethod
def load(cls, save_folder="."):
    parameters_path = os.path.join(save_folder, "parameters.pkl")
    with open(parameters_path, "rb") as f:
        parameters = pickle.load(f)
    autoencoder = Autoencoder(*parameters)
    weights_path = os.path.join(save_folder, "weights.h5")
    autoencoder.load_weights(weights_path)
    return autoencoder

def _create_folder_if_it_doesnt_exist(self, folder):
    if not os.path.exists(folder):
        os.makedirs(folder)

def _save_parameters(self, save_folder):
    parameters = [
        self.input_shape,
        self.conv_filters,
        self.conv_kernels,
        self.conv_strides,
        self.latent_space_dim
    ]
    save_path = os.path.join(save_folder, "parameters.pkl")
    with open(save_path, "wb") as f:
        pickle.dump(parameters, f)

def _save_weights(self, save_folder):
    save_path = os.path.join(save_folder, "weights.h5")
    self.model.save_weights(save_path)
```

```
def _build(self):
    self._build_encoder()
    self._build_decoder()
    self._build_autoencoder()

def _build_autoencoder(self):
    model_input = self._model_input
    model_output = self.decoder(self.encoder(model_input))
    self.model = Model(model_input, model_output, name="autoencoder")

def _build_decoder(self):
    decoder_input = self._add_decoder_input()
    dense_layer = self._add_dense_layer(decoder_input)
    reshape_layer = self._add_reshape_layer(dense_layer)
    conv_transpose_layers = self._add_conv_transpose_layers(reshape_layer)
    decoder_output = self._add_decoder_output(conv_transpose_layers)
    self.decoder = Model(decoder_input, decoder_output, name="decoder")

def _add_decoder_input(self):
    return Input(shape=self.latent_space_dim, name="decoder_input")

def _add_dense_layer(self, decoder_input):
    num_neurons = np.prod(self._shape_before_bottleneck) # [1, 2, 4] -> 8
    dense_layer = Dense(num_neurons, name="decoder_dense")(decoder_input)
    return dense_layer

def _add_reshape_layer(self, dense_layer):
    return Reshape(self._shape_before_bottleneck)(dense_layer)

def _add_conv_transpose_layers(self, x):
    """Add conv transpose blocks."""
```

```
    # loop through all the conv layers in reverse order and stop at the
    # first layer
    for layer_index in reversed(range(1, self._num_conv_layers)):
        x = self._add_conv_transpose_layer(layer_index, x)
    return x

def _add_conv_transpose_layer(self, layer_index, x):
    layer_num = self._num_conv_layers - layer_index
    conv_transpose_layer = Conv2DTranspose(
        filters=self.conv_filters[layer_index],
        kernel_size=self.conv_kernels[layer_index],
        strides=self.conv_strides[layer_index],
        padding="same",
        name=f"decoder_conv_transpose_layer_{layer_num}"
    )
    x = conv_transpose_layer(x)
    x = ReLU(name=f"decoder_relu_{layer_num}")(x)
    x = BatchNormalization(name=f"decoder_bn_{layer_num}")(x)
    return x

def _add_decoder_output(self, x):
    conv_transpose_layer = Conv2DTranspose(
        filters=1,
        kernel_size=self.conv_kernels[0],
        strides=self.conv_strides[0],
        padding="same",
        name=f"decoder_conv_transpose_layer_{self._num_conv_layers}"
    )
    x = conv_transpose_layer(x)
    output_layer = Activation("sigmoid", name="sigmoid_layer")(x)
    return output_layer
```

```
def _build_encoder(self):
    encoder_input = self._add_encoder_input()
    conv_layers = self._add_conv_layers(encoder_input)
    bottleneck = self._add_bottleneck(conv_layers)
    self._model_input = encoder_input
    self.encoder = Model(encoder_input, bottleneck, name="encoder")

def _add_encoder_input(self):
    return Input(shape=self.input_shape, name="encoder_input")

def _add_conv_layers(self, encoder_input):
    """Create all convolutional blocks in encoder."""
    x = encoder_input
    for layer_index in range(self._num_conv_layers):
        x = self._add_conv_layer(layer_index, x)
    return x

def _add_conv_layer(self, layer_index, x):
    """Add a convolutional block to a graph of layers, consisting of
    conv 2d + ReLU + batch normalization.
    """
    layer_number = layer_index + 1
    conv_layer = Conv2D(
        filters=self.conv_filters[layer_index],
        kernel_size=self.conv_kernels[layer_index],
        strides=self.conv_strides[layer_index],
        padding="same",
        name=f"encoder_conv_layer_{layer_number}"
    )
    x = conv_layer(x)
    x = ReLU(name=f"encoder_relu_{layer_number}")(x)
    x = BatchNormalization(name=f"encoder_bn_{layer_number}")(x)
```

```
        return x

    def _add_bottleneck(self, x):
        """Flatten data and add bottleneck (Dense layer)."""
        self._shape_before_bottleneck = K.int_shape(x)[1:]
        x = Flatten()(x)
        x = Dense(self.latent_space_dim, name="encoder_output")(x)
        return x


if __name__ == "__main__":
    autoencoder = Autoencoder(
        input_shape=(28, 28, 1),
        conv_filters=(32, 64, 64, 64),
        conv_kernels=(3, 3, 3, 3),
        conv_strides=(1, 2, 2, 1),
        latent_space_dim=2
    )
    autoencoder.summary()
```

Autoencoder training.

```
import os
from tensorflow.keras.datasets import mnist
from autoencoder import Autoencoder
from PIL import Image
import numpy as np

LEARNING_RATE = 0.0005
BATCH_SIZE = 10
EPOCHS = 2500



def load_custom_data(data_dir):
    # Initialize empty lists to store images and labels (if applicable)
    data = []
    labels = []  # If your dataset includes labels, otherwise remove this line

    # List all files in the data directory
    file_list = os.listdir(data_dir)

    for filename in file_list:
        if filename.endswith(".jpg") or filename.endswith(".png"):
            # Load and preprocess images
            img = Image.open(os.path.join(data_dir, filename))
            img = img.resize((28, 28))  # Adjust dimensions to match your model
            img = np.array(img)  # Convert to numpy array
            img = img.astype("float32") / 255.0  # Normalize pixel values to [0, 1]

            data.append(img)

            # If your dataset includes labels, you can load them here and append to 'labels' list
```

```
    # Convert the data list to a numpy array
    x_train = np.array(data)

    return x_train


# Example usage:
data_dir = "C:/Users/Rohit/Desktop/28"
x_train = load_custom_data(data_dir)


def train(x_train, learning_rate, batch_size, epochs):
    autoencoder = Autoencoder(
        input_shape=(28, 28, 1),
        conv_filters=(32, 64, 64, 64),
        conv_kernels=(3, 3, 3, 3),
        conv_strides=(1, 2, 2, 1),
        latent_space_dim=2
    )
    autoencoder.summary()
    autoencoder.compile(learning_rate)
    autoencoder.train(x_train, batch_size, epochs)
    return autoencoder


if __name__ == "__main__":
    x_train = load_custom_data(data_dir)
    autoencoder = train(x_train[:800], LEARNING_RATE, BATCH_SIZE, EPOCHS)
    autoencoder.save("model")
    autoencoder2 = Autoencoder.load("model")
    autoencoder2.summary()
```

Reconstruction:

```
import numpy as np
import matplotlib.pyplot as plt
import os  # Import the os module for file operations

from autoencoder import Autoencoder  # Import your Autoencoder class
from train import load_custom_data  # Import your data loading function

def select_and_save_images(x_train, indices, output_dir):
    # Create the output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    # Select the images from x_train using the provided indices
    selected_images = x_train[indices]

    # Generate the corresponding reconstructed images
    reconstructed_images, _ = autoencoder.reconstruct(selected_images)

    # Iterate through selected images and save them
    for i, (original_image, reconstructed_image) in enumerate(zip(selected_images,
reconstructed_images)):
        # Save the original image
        original_path = os.path.join(output_dir, f"original_{i}.png")
        plt.imsave(original_path, original_image.squeeze(), cmap="gray_r")

        # Save the reconstructed image
        reconstructed_path = os.path.join(output_dir, f"reconstructed_{i}.png")
        plt.imsave(reconstructed_path, reconstructed_image.squeeze(), cmap="gray_r")

if __name__ == "__main__":
    autoencoder = Autoencoder.load("model")  # Load your Autoencoder model
```

```
    data_dir = "C:/Users/Rohit/Desktop/28"
    x_train = load_custom_data(data_dir)  # Load your custom data

    # Define the indices you want to select from x_train
    selected_indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
# Replace with the indices you want

    # Define the output directory where images will be saved
    output_directory = "C:/Users/Rohit/Desktop/output_images"

    # Call the function to select and save images
    select_and_save_images(x_train, selected_indices, output_directory)
```

Inverse spectrogram:

```
import numpy as np

import librosa

import librosa.display

import matplotlib.pyplot as plt

from PIL import Image

from IPython.display import Audio


# Load the spectrogram from a PNG image (replace 'your_spectrogram.png' with your file path)

spectrogram_image = Image.open('C:/Users/Rohit/Desktop/final_reconstructed/output_image4.png')

spectrogram_array = np.array(spectrogram_image)
```

```
# Parameters
hop_length = 450  # Adjust as needed, typically less than or equal to spectrogram width
win_length = 1088  # Set to the width of your spectrogram (28x28)
sr = 44000  # Sample rate

# Perform iSTFT with Hanning window
reconstructed_signal = librosa.istft(
    spectrogram_array, hop_length=hop_length, win_length=win_length, window="hann"
)

# Plot the reconstructed sound wave
plt.figure(figsize=(10, 6))
librosa.display.waveshow(reconstructed_signal, sr=sr)
plt.title("Reconstructed Sound Wave with Hanning Window")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.tight_layout()
plt.show()

# Play the reconstructed audio
Audio(data=reconstructed_signal, rate=sr)
```

References

Amruta Savagave, P. A. (2014). Study of Image Interpolation. *International Journal of Innovative Science, Engineering & Technology*, 529-534.

Apple. (2023, May 16). https://www.apple.com/newsroom/2023/05/apple-previews-live-speech-personal-voice-and-more-new-accessibility-features/

audiolabs-erlangen. (2015). https://www.audiolabs-erlangen.de/resources/MIR/FMP/C2/C2_STFT-Basic.html#Formal-Definition-of-the-Discrete-STFT

audiolabs-erlangen. (2015). *audiolabs-erlangen*. https://www.audiolabs-erlangen.de/resources/MIR/FMP/C2/C2_STFT-Inverse.html

Foster, D. (2019). *Generative Deep Learning.* O'Reilly Media.

Geekflare. (2023, July). Retrieved from https://geekflare.com/ai-voice-cloning-tools/

Hinton, G. &. (1986). Hinton, G.E., & Sejnowski, T.J. (1986). Learning and relearning in Boltzmann machines.

IBM. *IBM*. https://www.ibm.com/topics/convolutional-neural-networks

Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization.

Kulkarni, S. R. (2002). *princeton.edu.* https://www.princeton.edu/~cuff/ele201/kulkarni_text/frequency.pdf

Lyon, R. G. (2004). Understanding Digital signal Processing. Bernard Goodwin.

MacKay, D. (2003). Information Theory, Inference, and Learning Algorithms. Cambridge University Press.

MNIST. (2019, Jan 08). https://www.kaggle.com/datasets/hojjatk/mnist-dataset

Moore, D. S., McCabe, G. P., & Craig, B. A. (2019). *Introduction to the Practice of Statistics.* W. H. Freeman.

NASA. *NASA.* https://www.nasa.gov/wav/62284main_onesmall2.wav

Nielsen, M. A. (2015). *Neural Networks and Deep Learning* . Determination Press.

Pixabay/effects. (n.d.). https://pixabay.com/sound-effects/distant-ambulance-siren-6108/

R., Z. J. (2020, September 4). *Kaggle*. https://www.kaggle.com/datasets/joserzapata/free-spoken-digit-dataset-fsdd

SandipanDey. (2018, January 24). *datasciencecentral.com*. https://www.datasciencecentral.com/recursive-graphics-bi-tri-linear-interpolation-anti-aliasing-and/

Sebastian, R. (2014, July 11). *sebastianraschka.com*. https://sebastianraschka.com/Articles/2014_about_feature_scaling.html