

Exploring Generative Models (Variational Autoencoders) for Audio Generation



By:
Rabin Nepal & Sravan Kumar Dumpeti

Department of Electrical and Computer Engineering
University of Memphis

Presentation Outline

- Introduction
- Problem Statement and Objectives
- Methodology
- Results
- Discussion and Conclusion
- References

Introduction

- Speech is a unique to human trait
- Allows to express thoughts, ideas, and emotions
- But, what about Human-Computer Interaction?
 - Computers can recognize human speech
 - However, the responses are pre-recorded and not as dynamic
- This project will explore the area of generative AI to synthesize voices for such systems.

Problem Statement and Objective

- **Problem Statement:**

- AI has advanced, enabling the generation of various data forms.
- Most are focused on textual data, which lacks the naturalness of speech.
- Text-based AI models creates barriers and limits access of the technology.

- **Project Objectives:**

- Explore the application of generative models, specifically Variational AutoEncoders (VAE), in audio data.
- Focus on speech generation and spontaneity of human speech.
- Contribute insights to the understanding of variational autoencoders and their role in audio generation task

Methodology

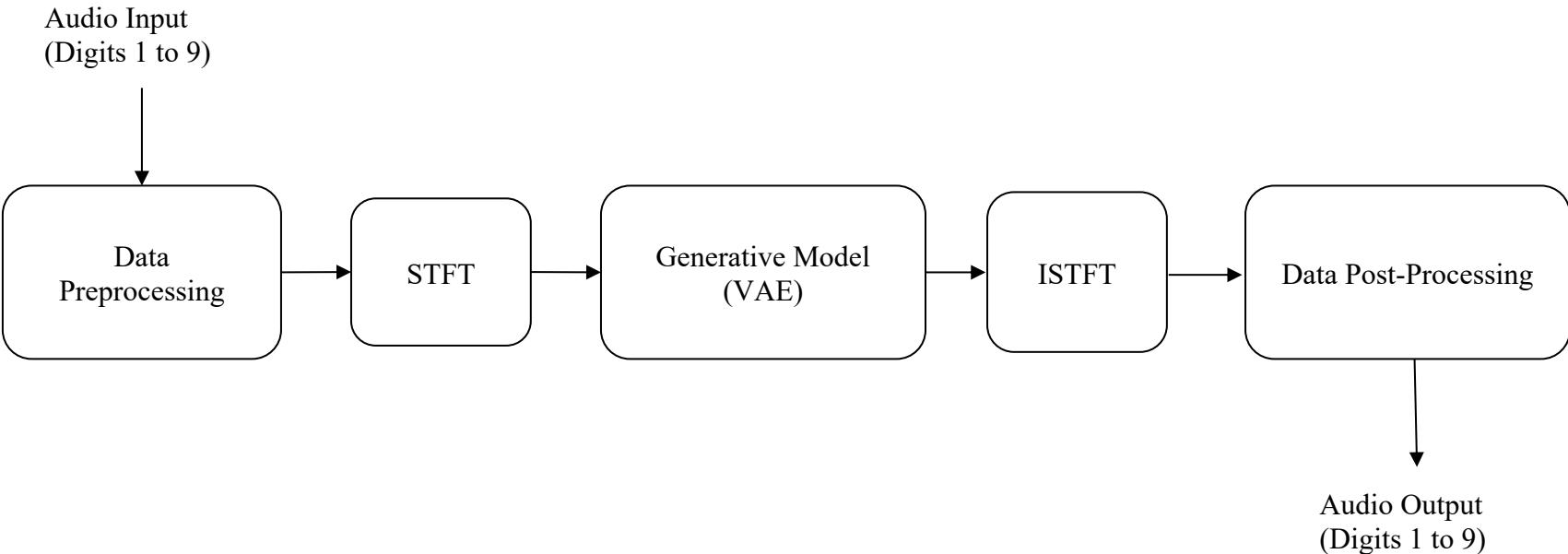


Fig: Block Diagram for the Training Phase of the Proposed Audio Generation System

Methodology

Input Data: Audio samples of spoken digits are used as input to the system.

Data Preprocessing: Raw audio data undergoes preprocessing to eliminates this inconsistency in data samples making them uniform in amplitude, length, etc.

Feature Extraction: STFT is a way of observing the frequency and phase content of a signal over time. It is applied to extract relevant features from the preprocessed audio data.

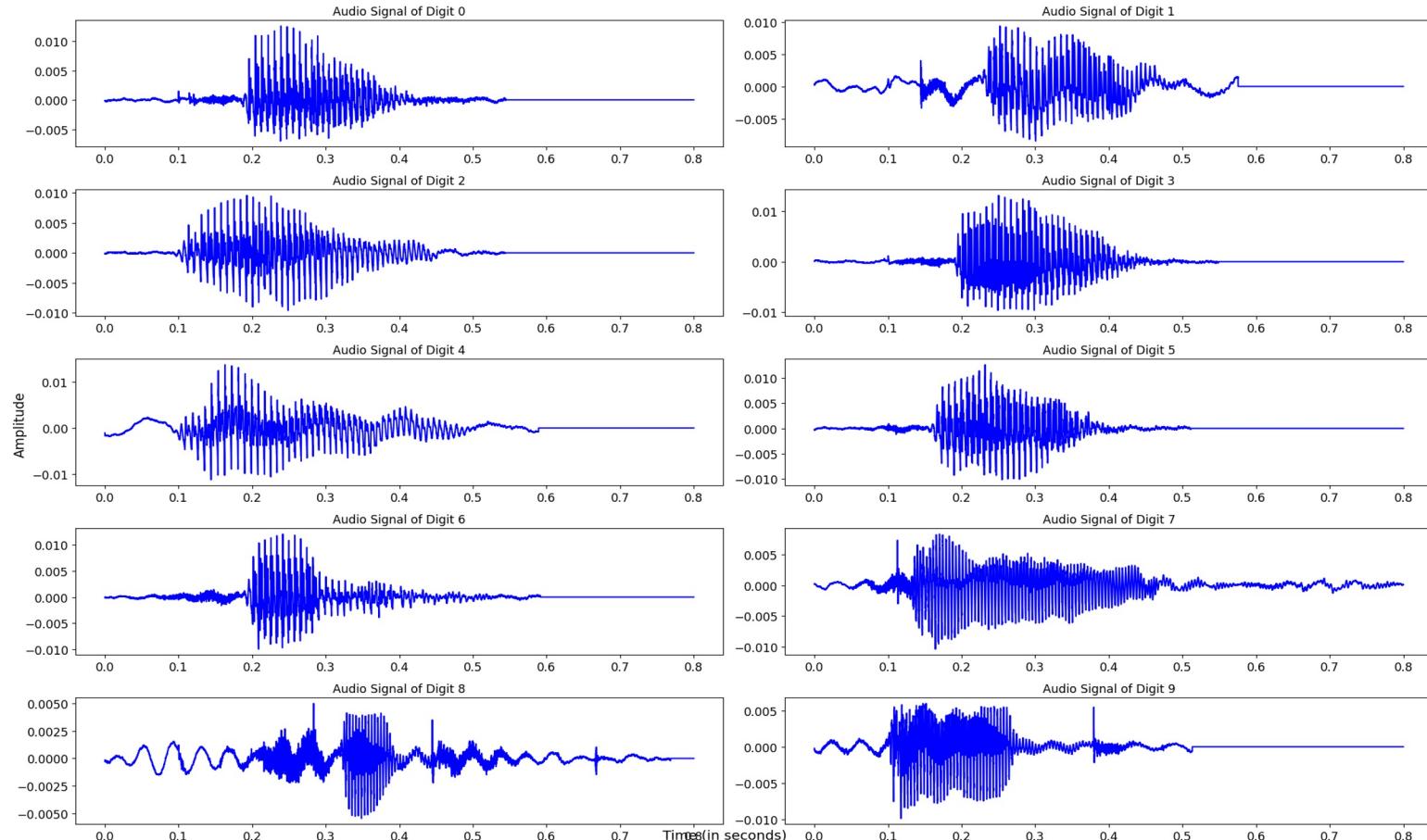
Generative Model: The STFT of audio signals are fed into a Variational Autoencoder (VAE), a generative model, to generate new STFT representations for the audio data.

Inverse Short Time Fourier Transform (ISTFT): The generated STFT data is converted back into an audio signal using Inverse Short Time Fourier Transform(ISTFT), reconstructing the audio waveform.

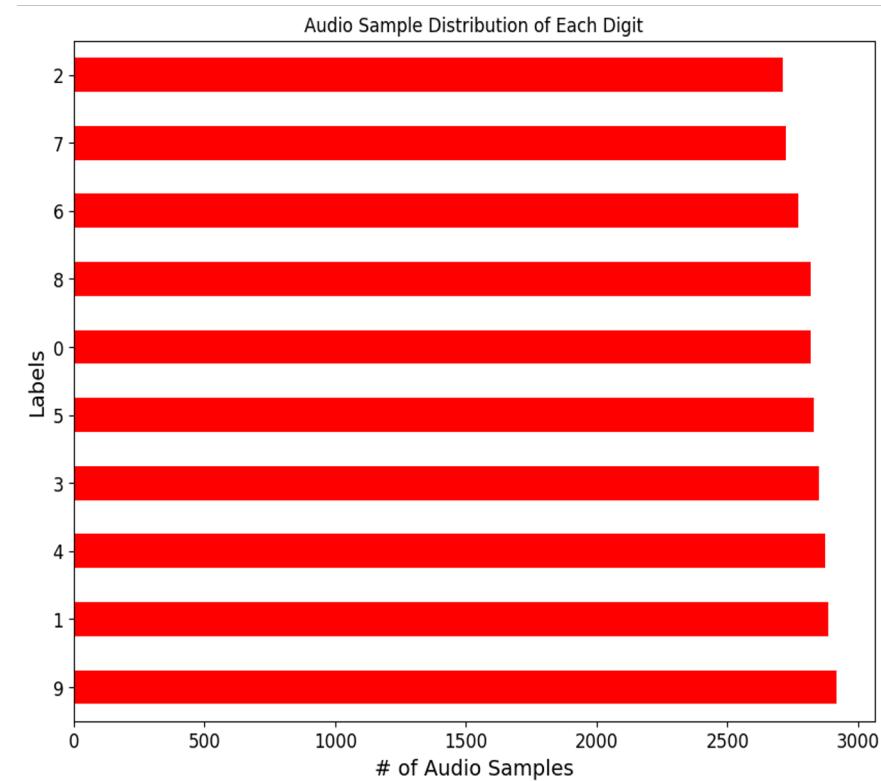
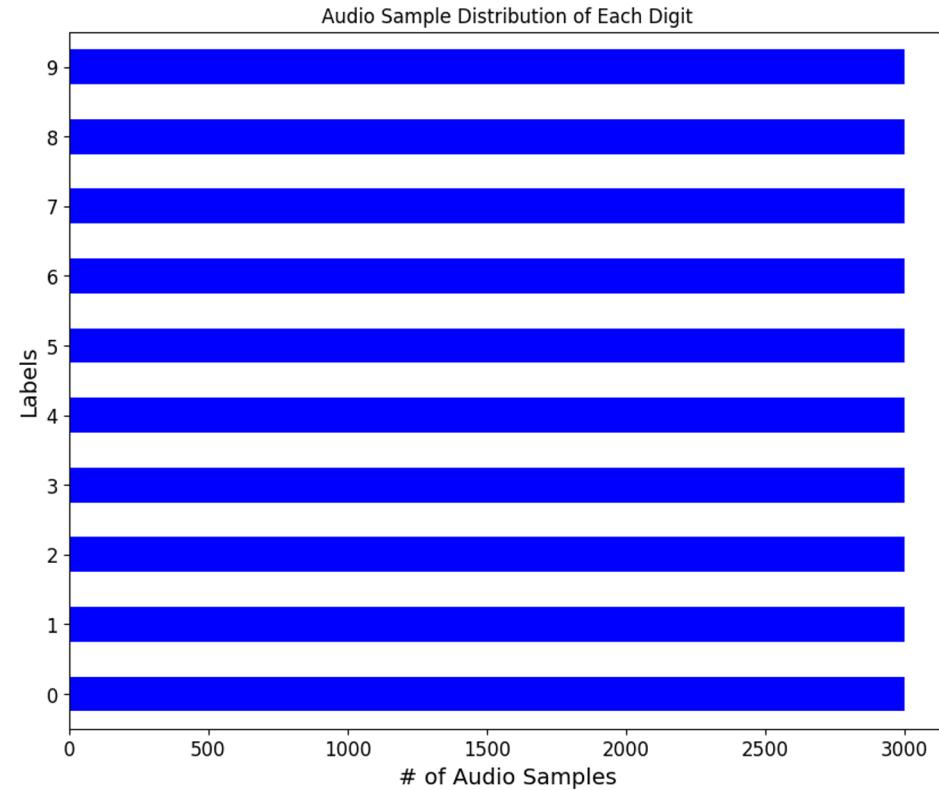
Data Post-Processing: The reconstructed audio signal undergoes post-processing, including amplitude adjustment, frequency conversion, and other necessary enhancements.

Output: The final output is a properly generated audio signal, retaining the natural nuances of human speech.

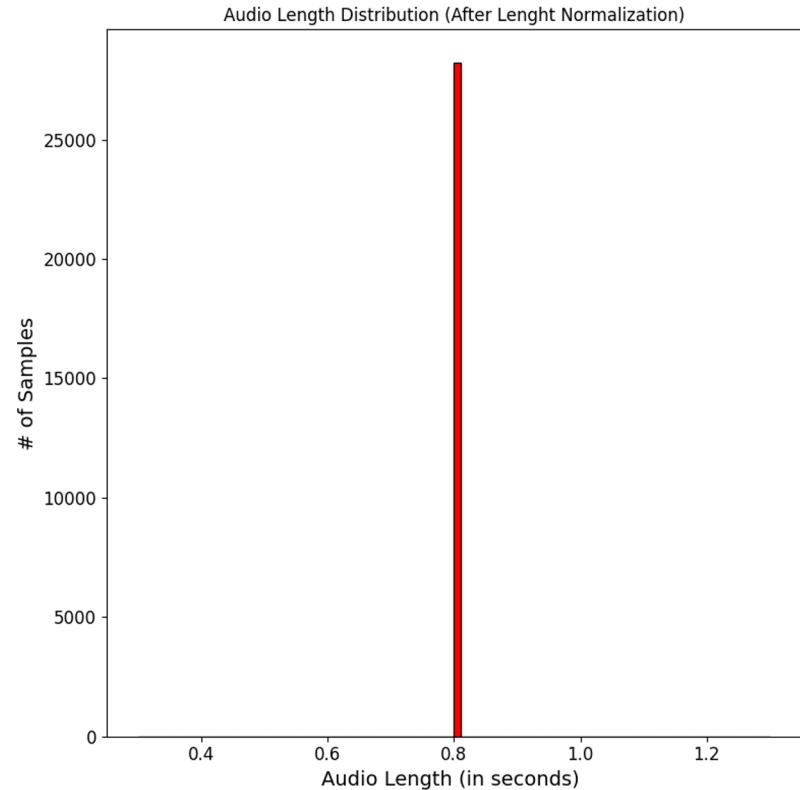
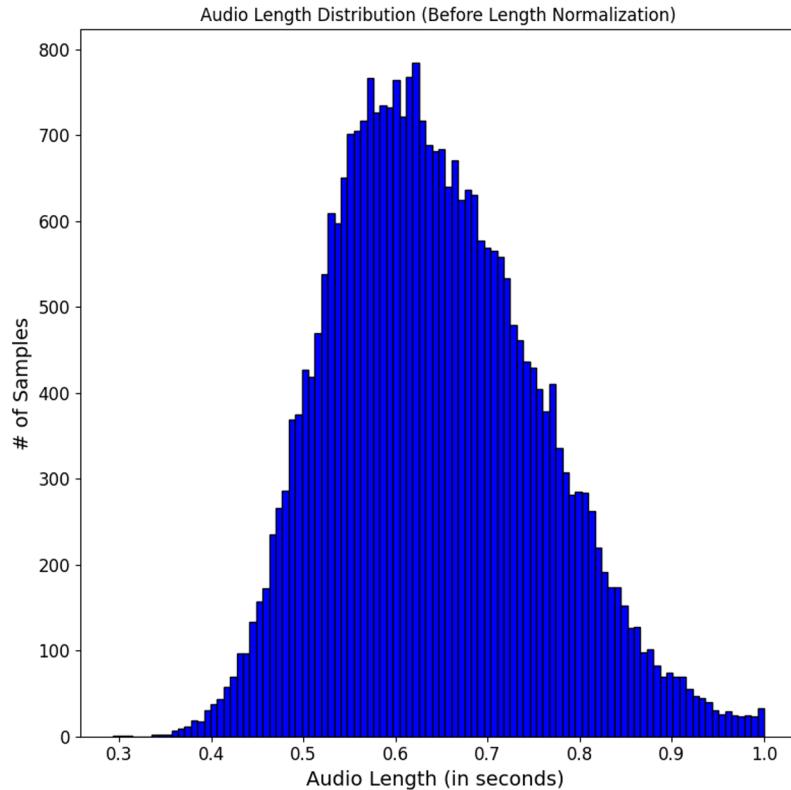
Input Signal



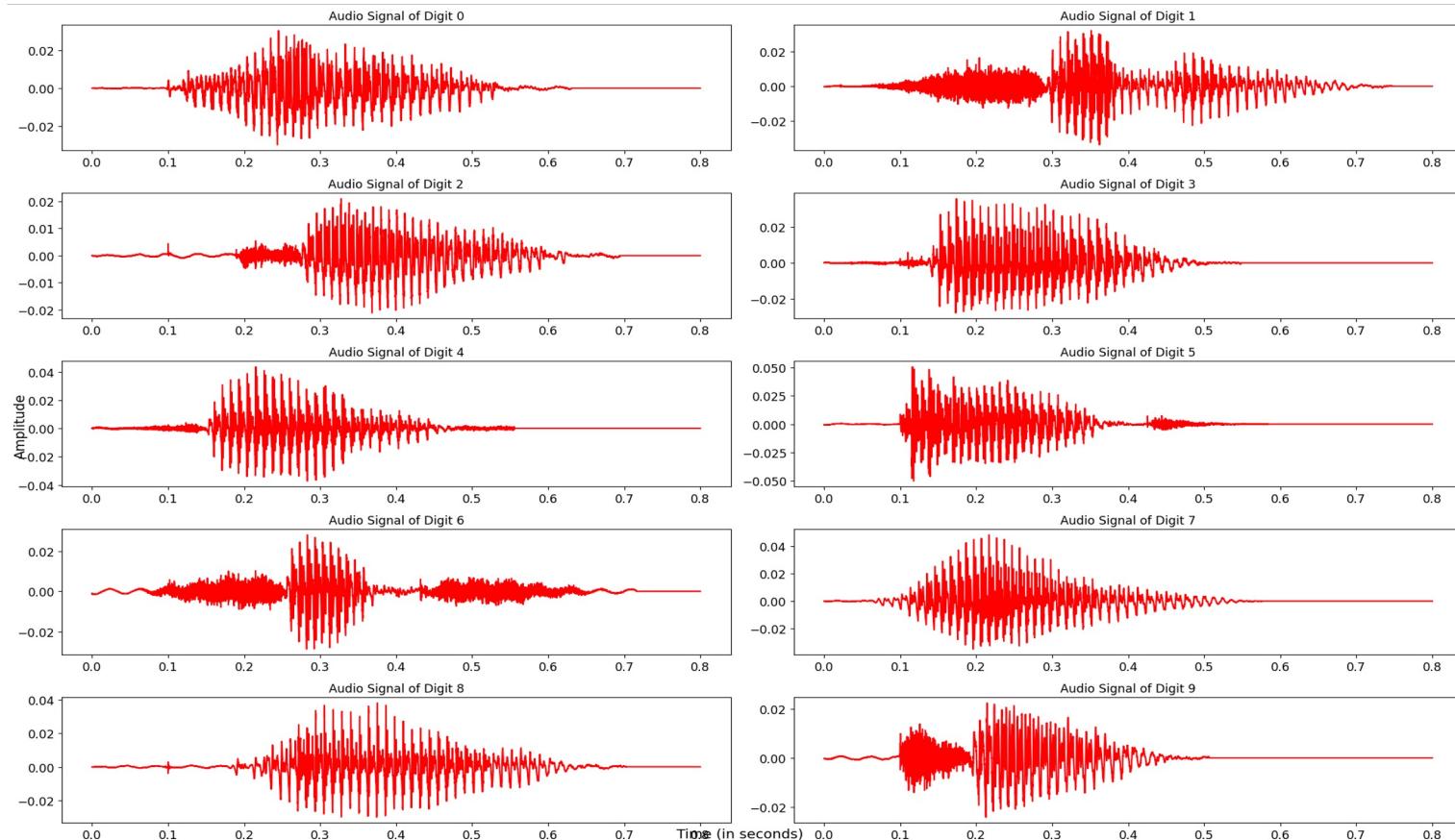
Data Preprocessing [Filtering Unusable Samples]



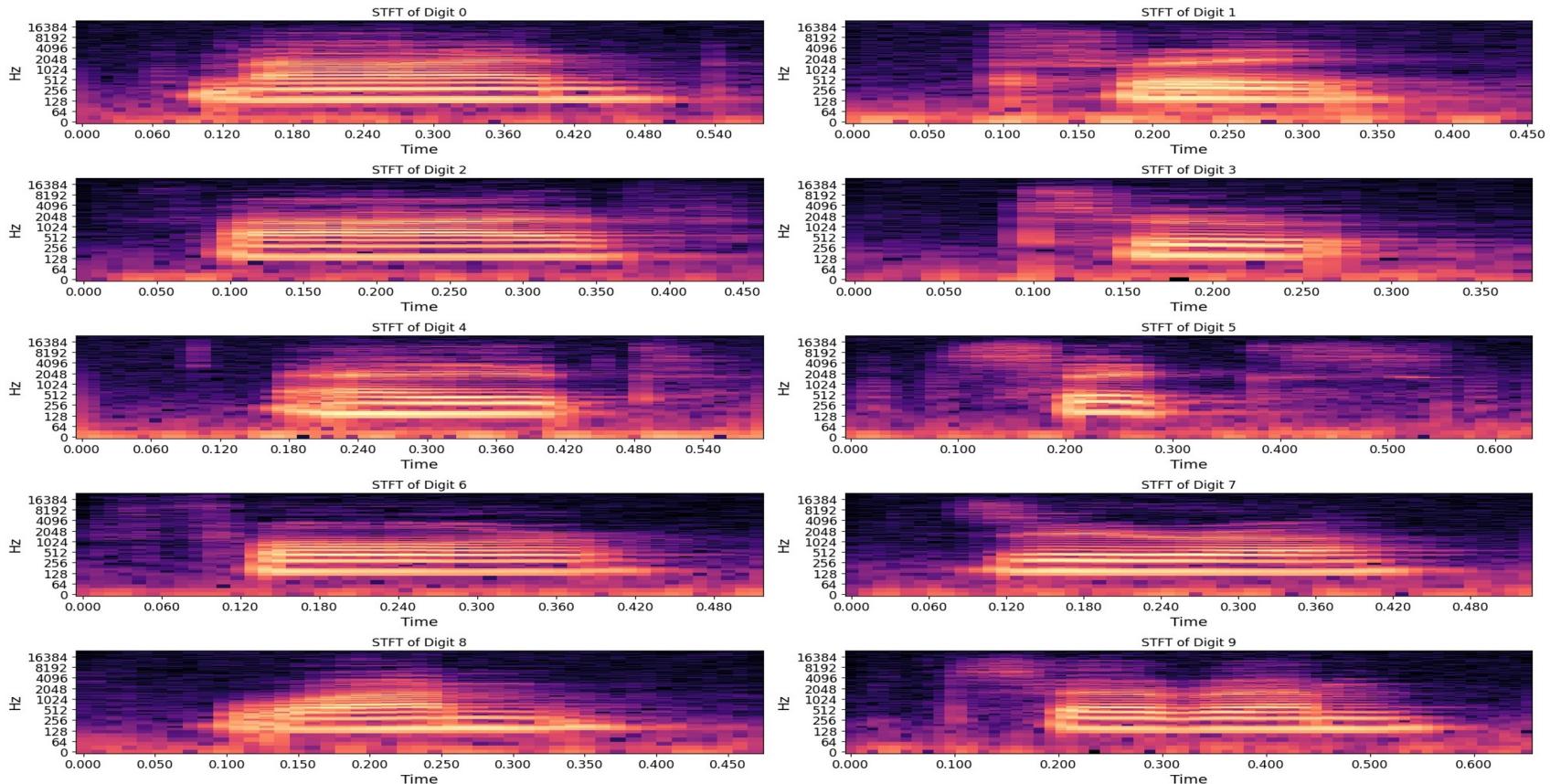
Data Preprocessing [Length Normalization]



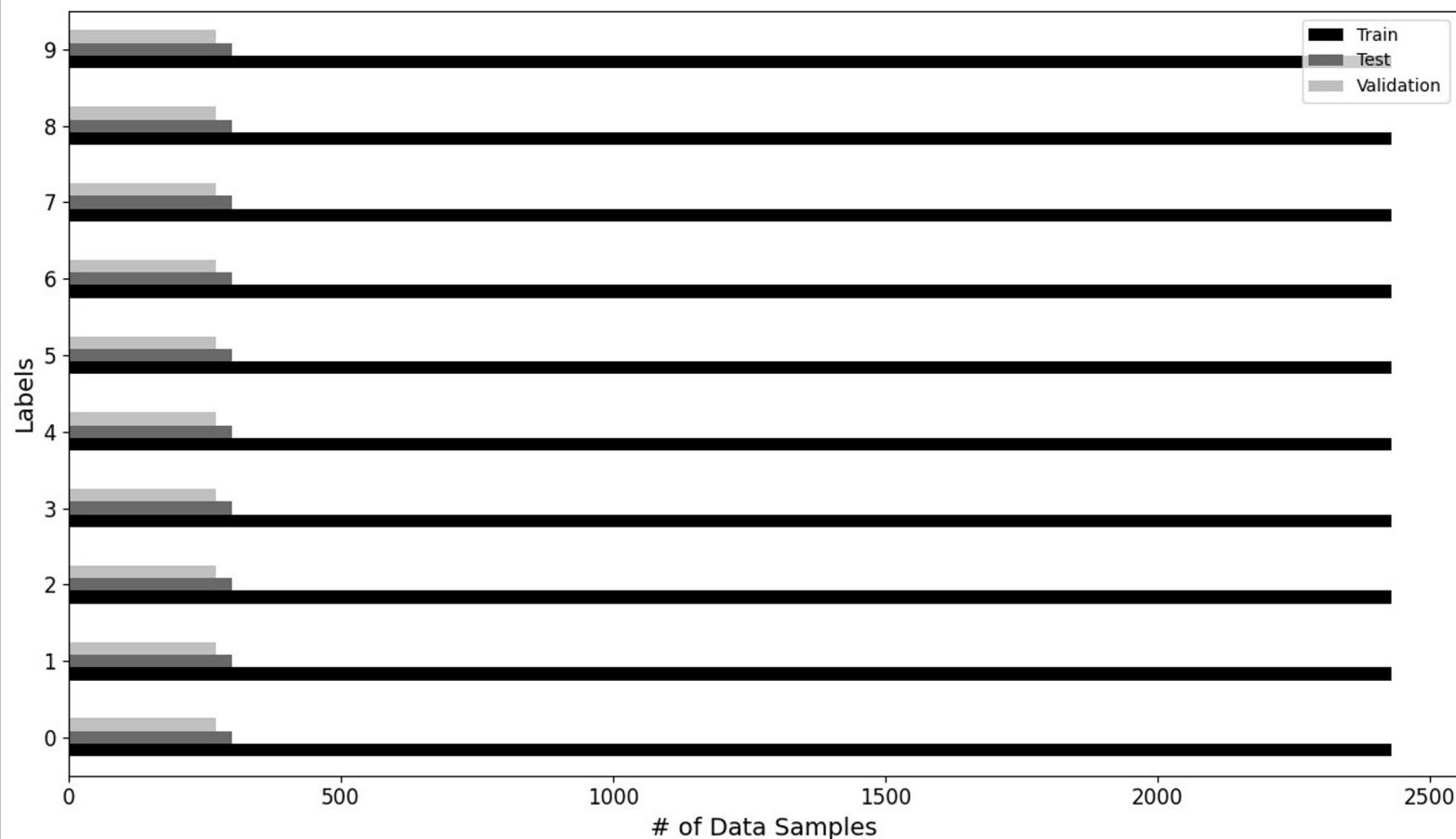
Data Preprocessing [Length Normalization]



Feature Extraction (STFT)

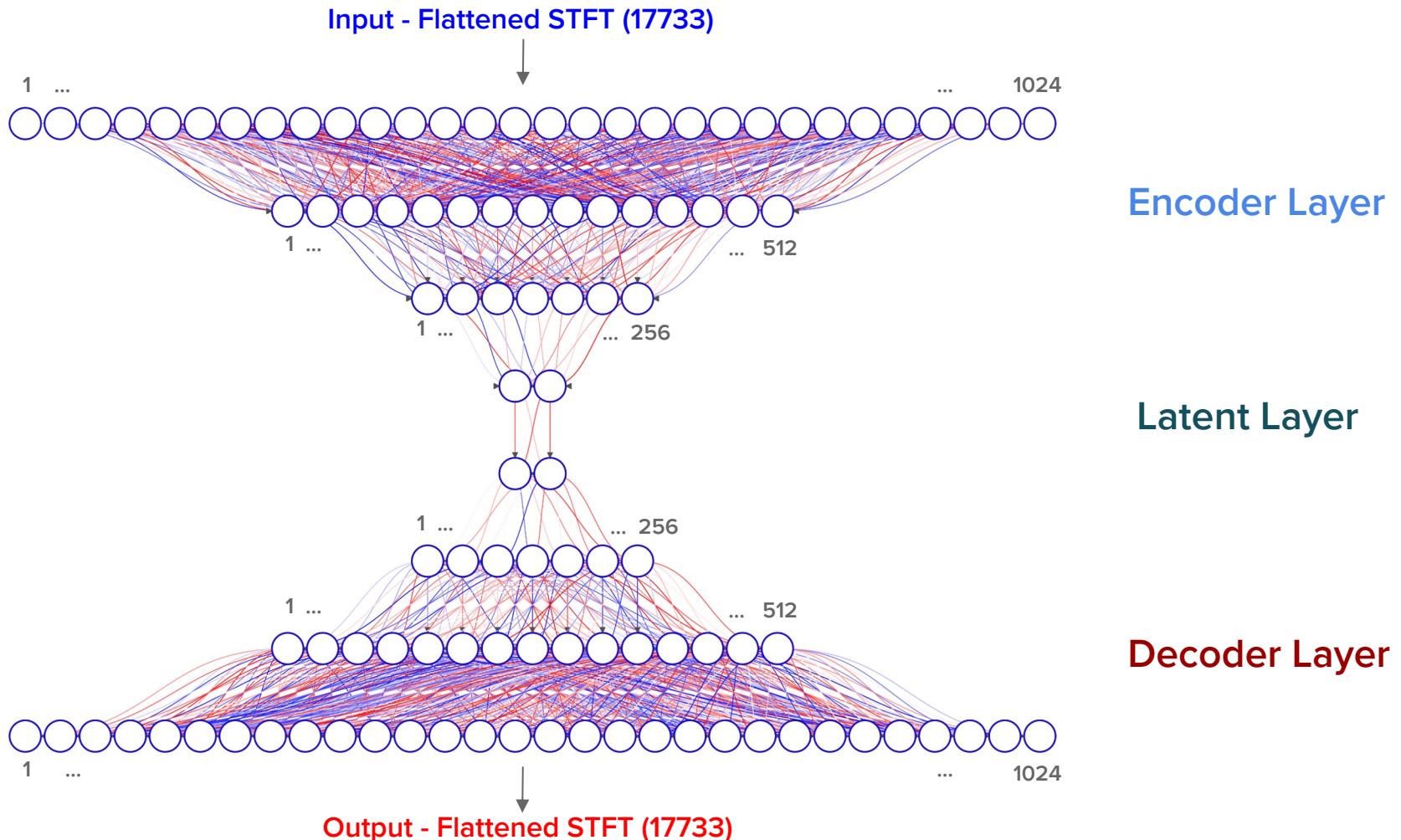


Dataset Split

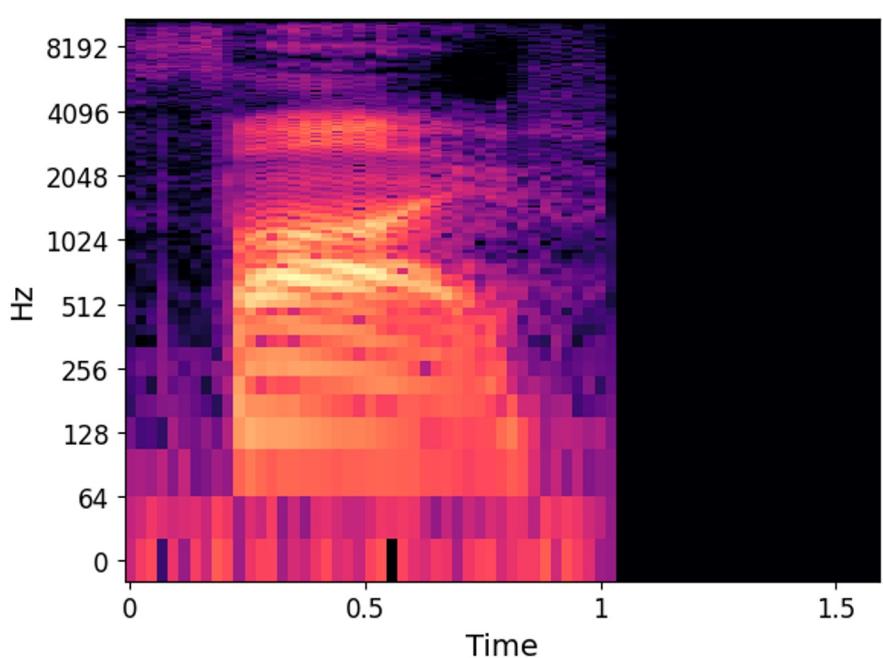


MODEL ARCHITECTURE

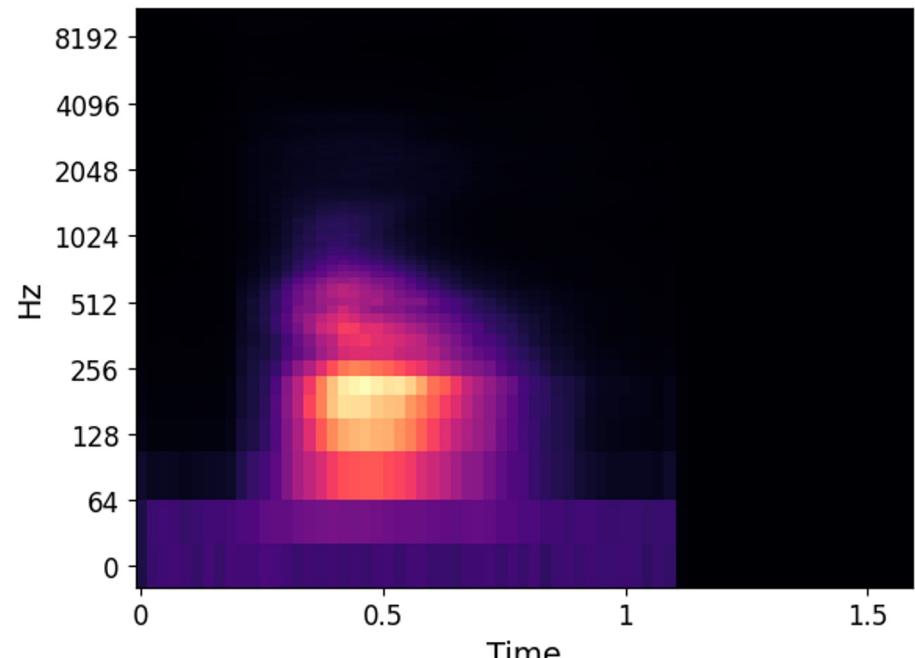
- **Unsupervised Learning:** VAE is an unsupervised learning method designed to comprehend the data distribution and generate new data points.
- **Architecture:** The VAE comprises two vital components - an encoder and a decoder. These components collaborate to minimize the reconstruction error between the encoded-decoded data and the initial dataset.
- **Encoder:** The encoder maps the dataset into a latent representation, creating a joint probability distribution of the samples. This representation serves as the foundation for generating new, meaningful data.
- **Decoder:** A random sample from this latent space is fed into the decoder, reconstructing the data. The reconstruction error, calculated between the input and reconstructed data, guides the training process.
- **Objective:** VAE is trained to minimize this reconstruction error, ensuring that the generated data closely resembles the original dataset, capturing its essential features.
- **Loss = Reconstruction_loss (error in reconstruction) + KL_Divergence (latent space to conform to a specific desired distribution)**



Results [Comparison of Generated STFT]

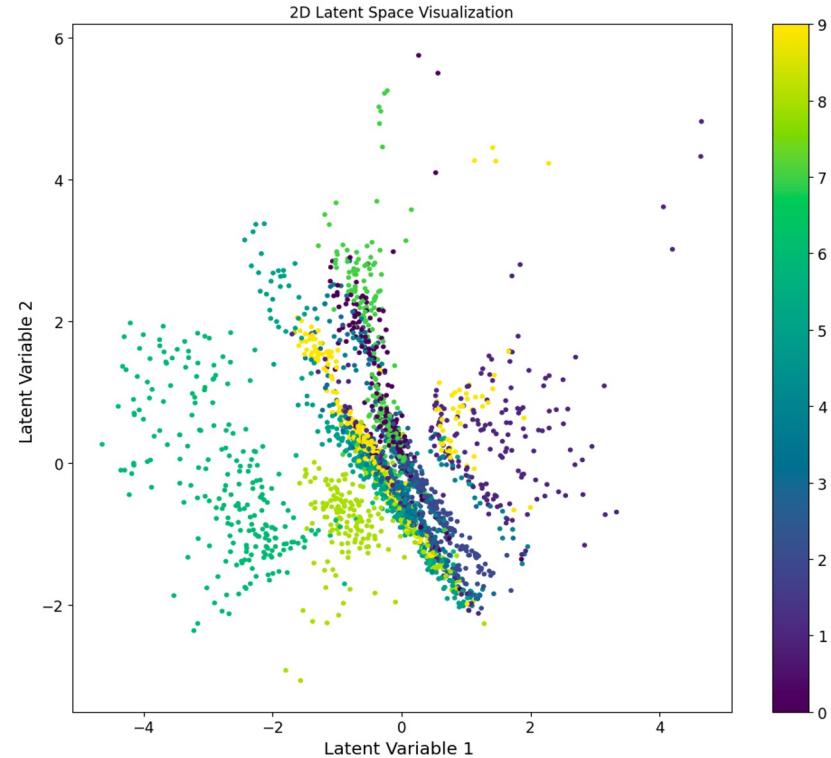
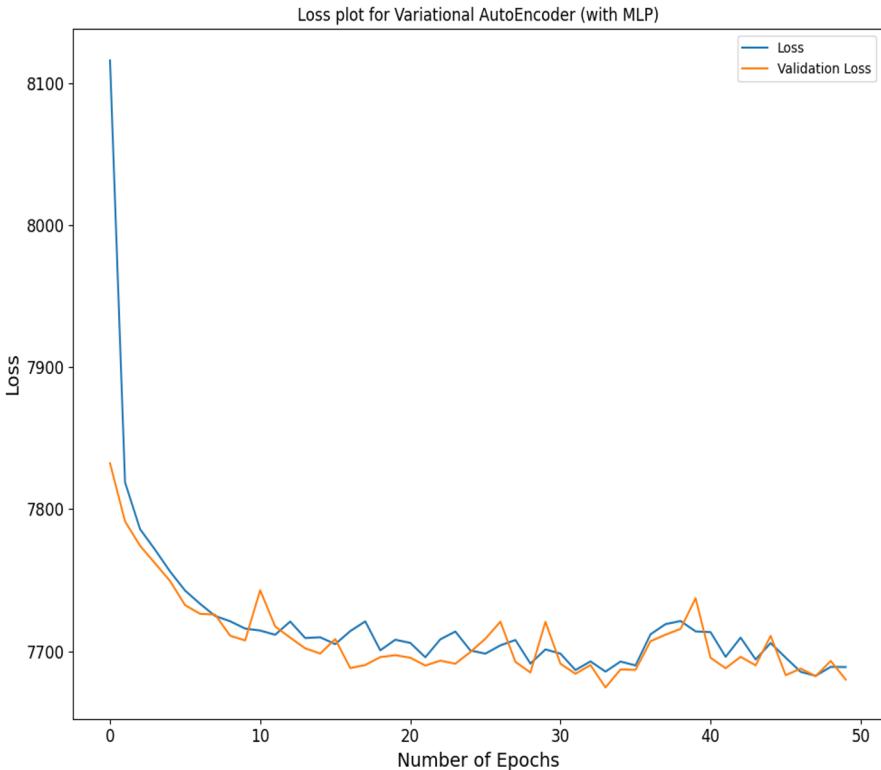


Input STFT for Digit 5

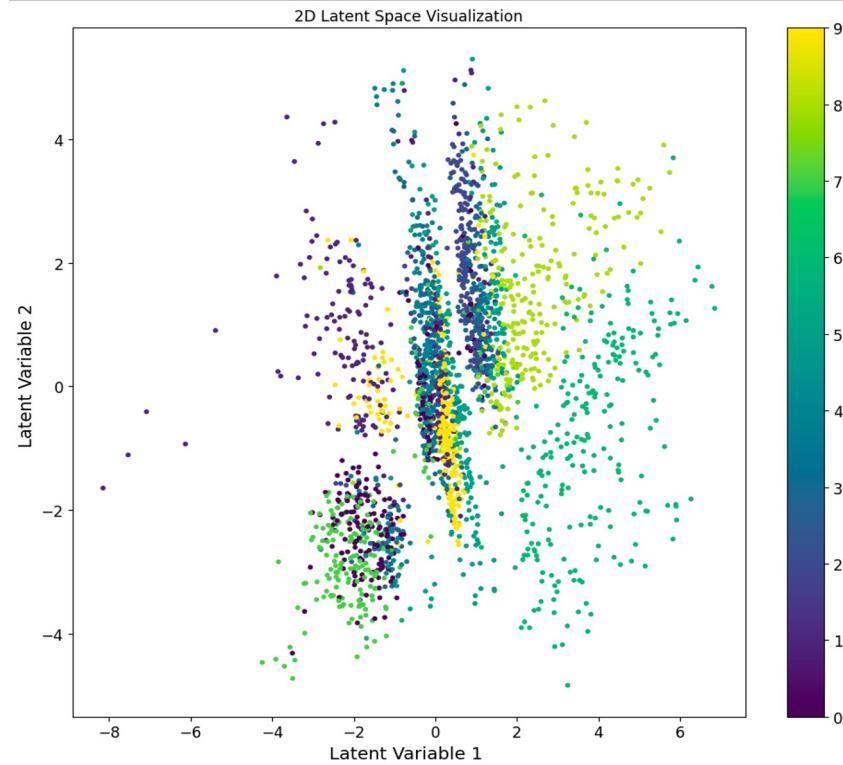
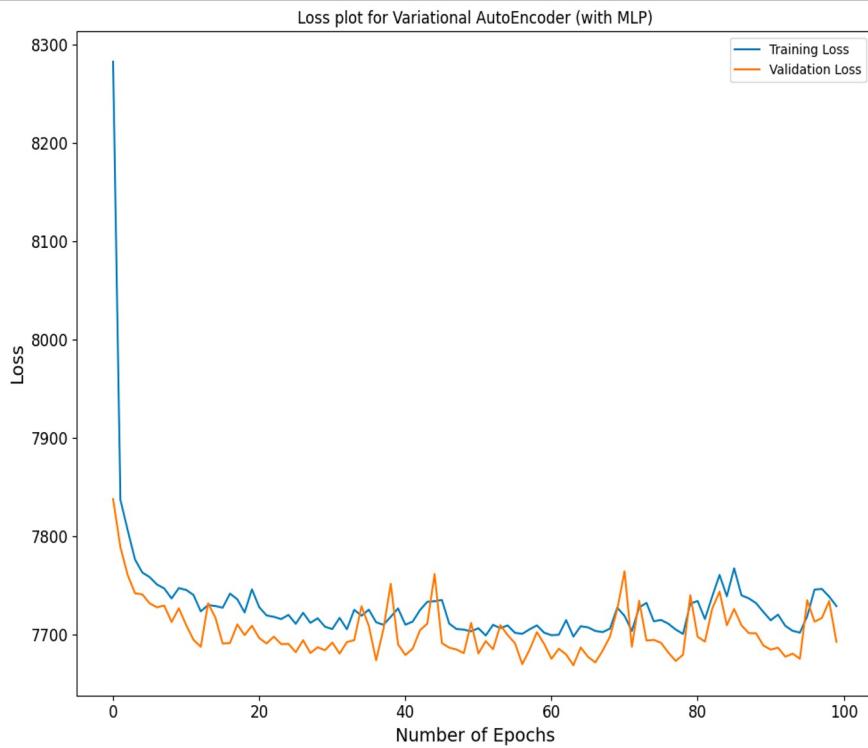


Generated STFT for Digit 5

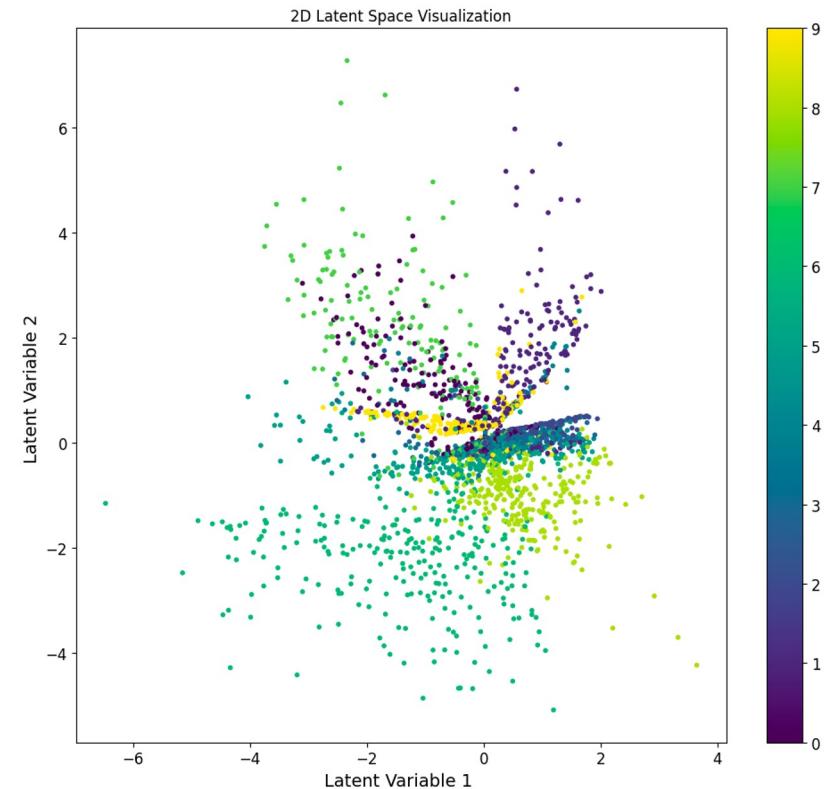
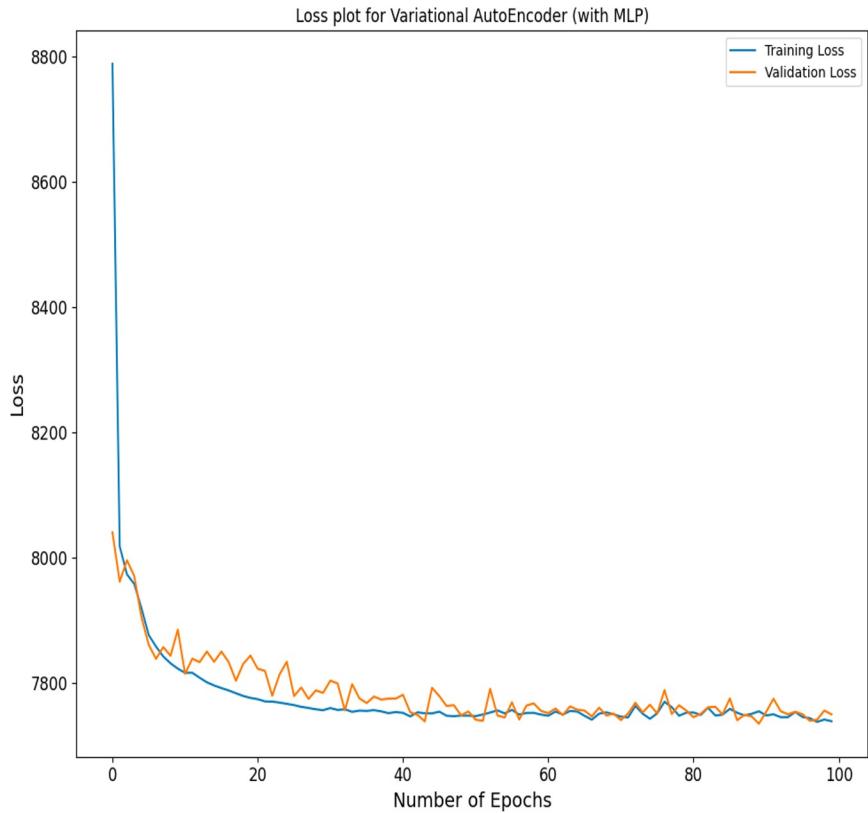
Results [MLP-VAE Performance]



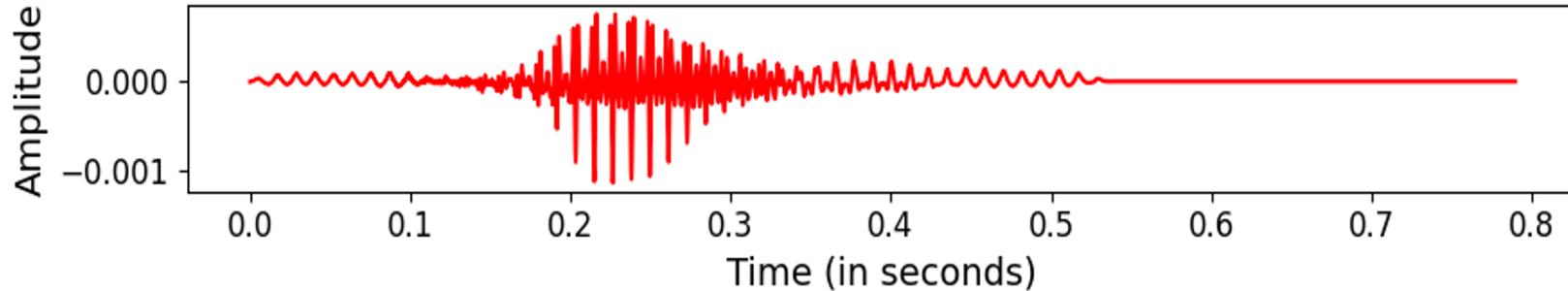
Results [MLP-VAE 100 Epochs]



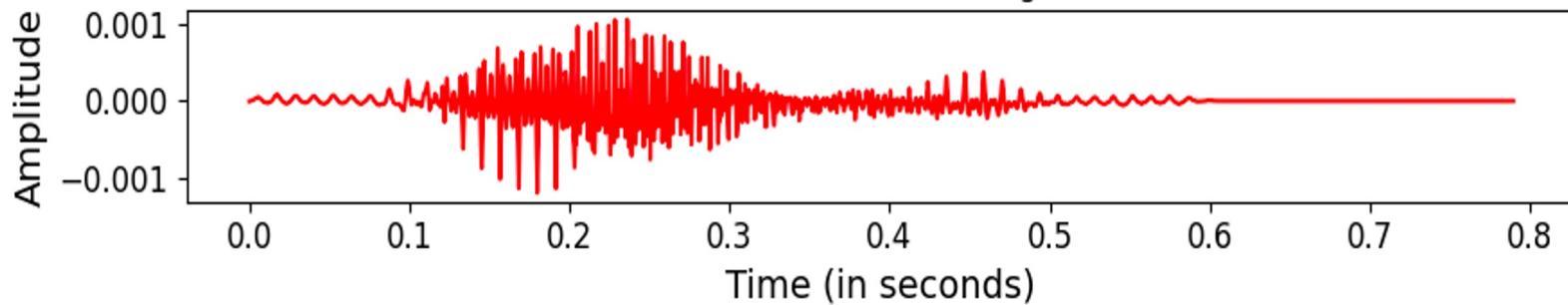
Results [MLP-VAE Performance After Regularization]



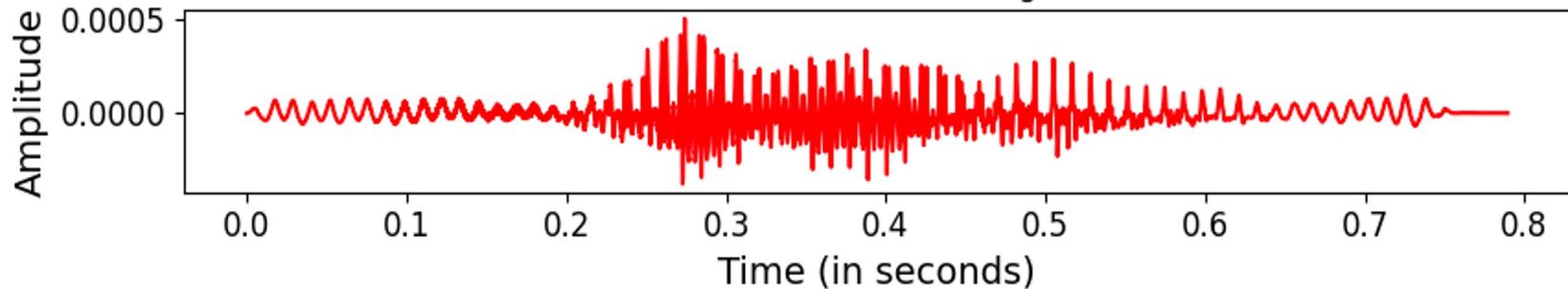
Generated Audio for Digit 3



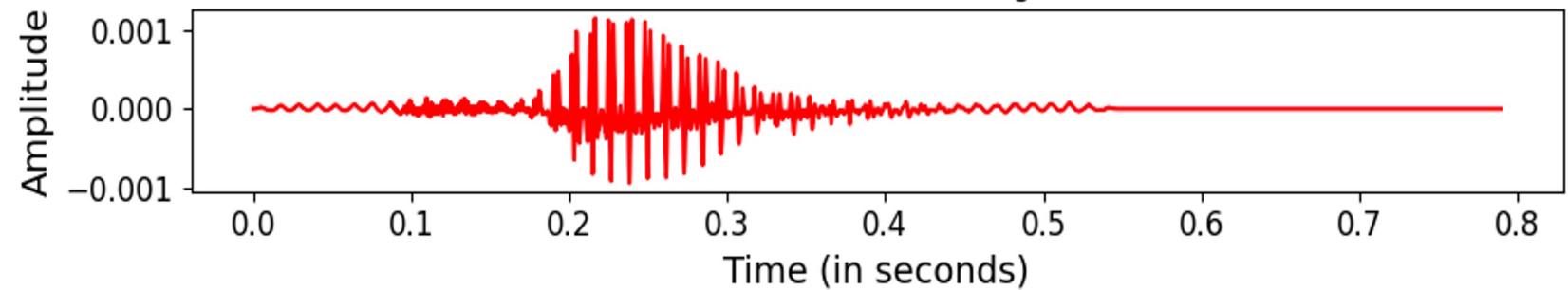
Generated Audio for Digit 1



Generated Audio for Digit 0

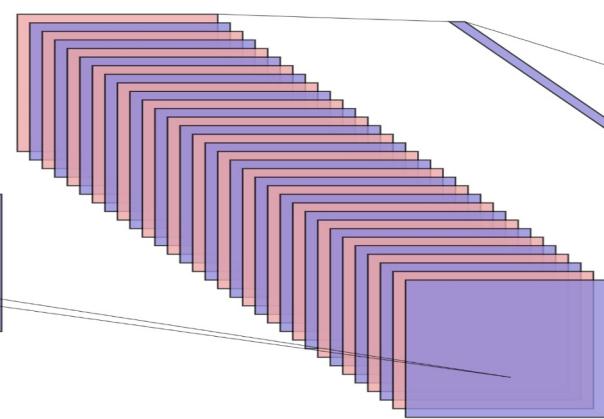
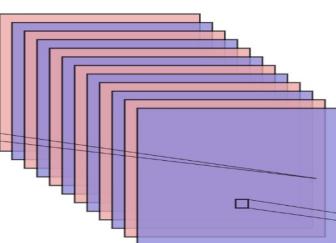


Generated Audio for Digit 2



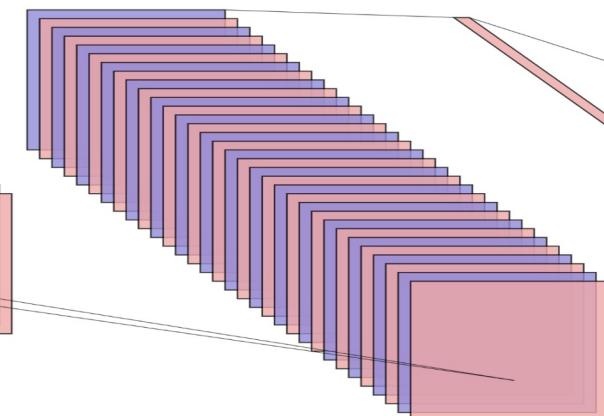
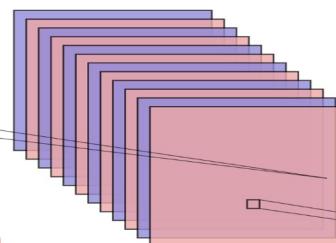
Encoder Layer

Input STFT (257*69*1)



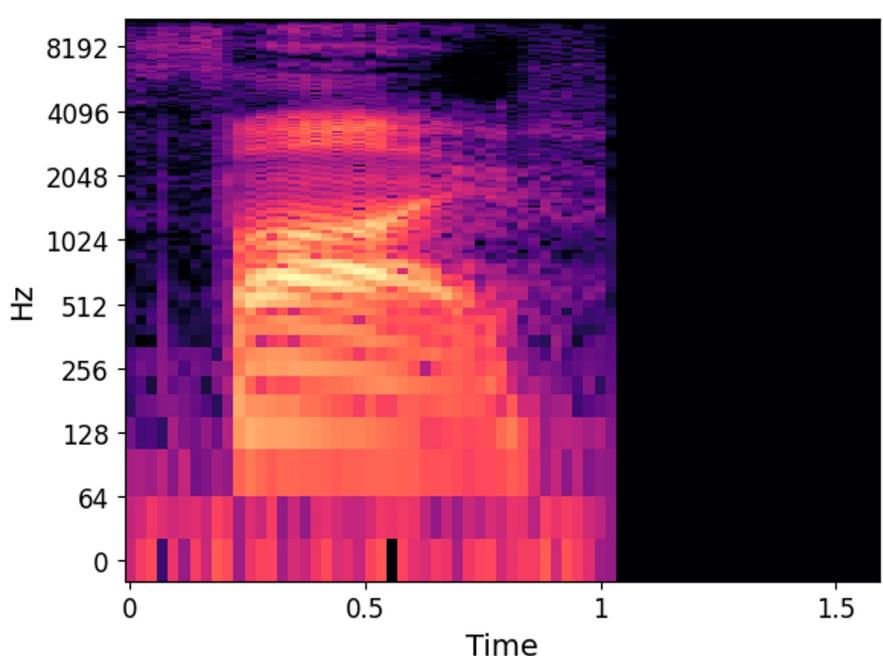
Latent Layer

Output STFT (257*69*1)

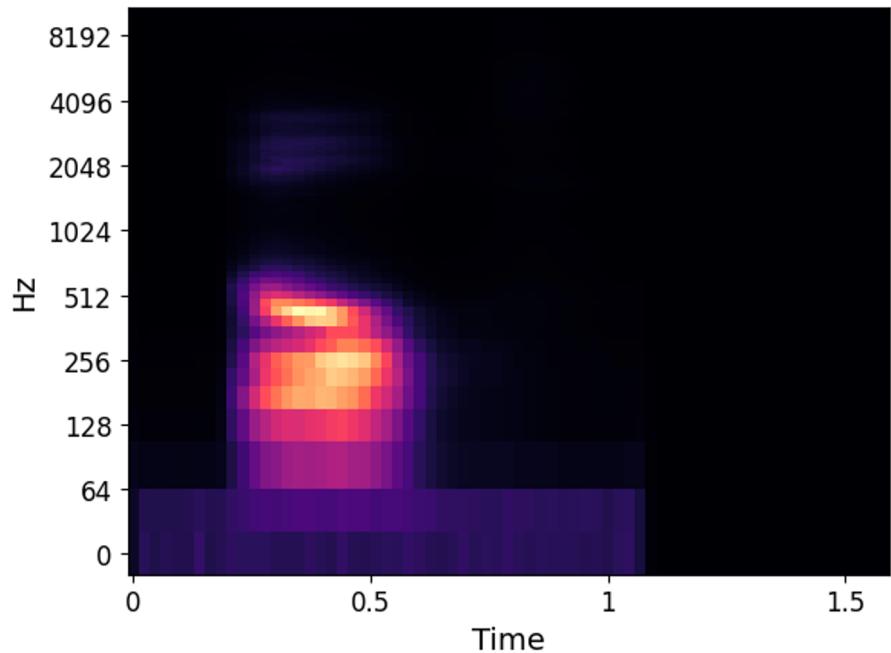


Decoder Layer

Results [Comparison of Generated STFT]

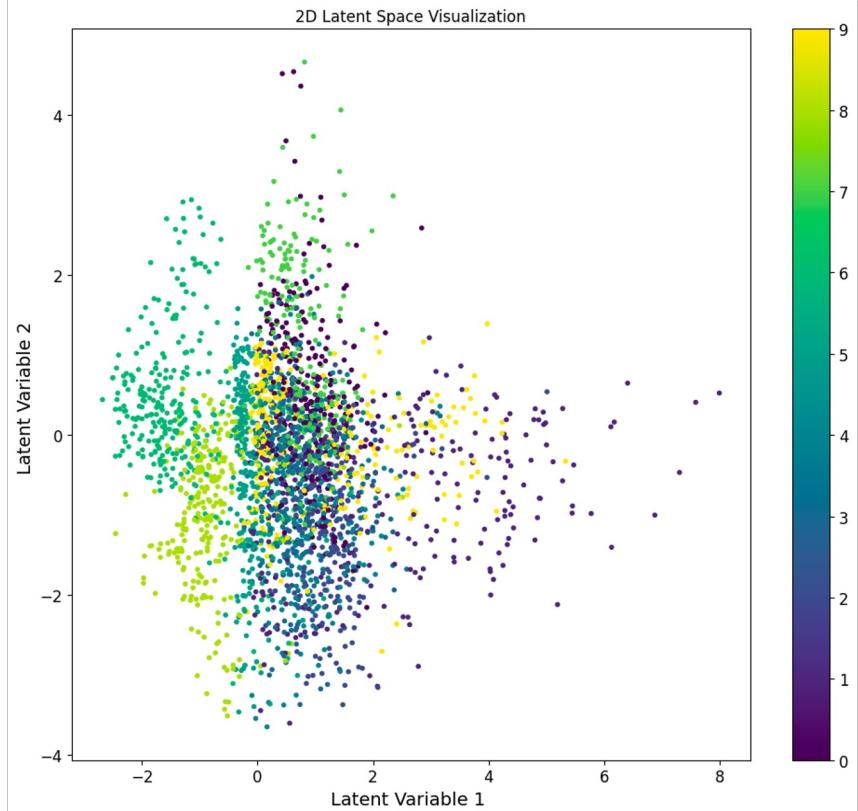
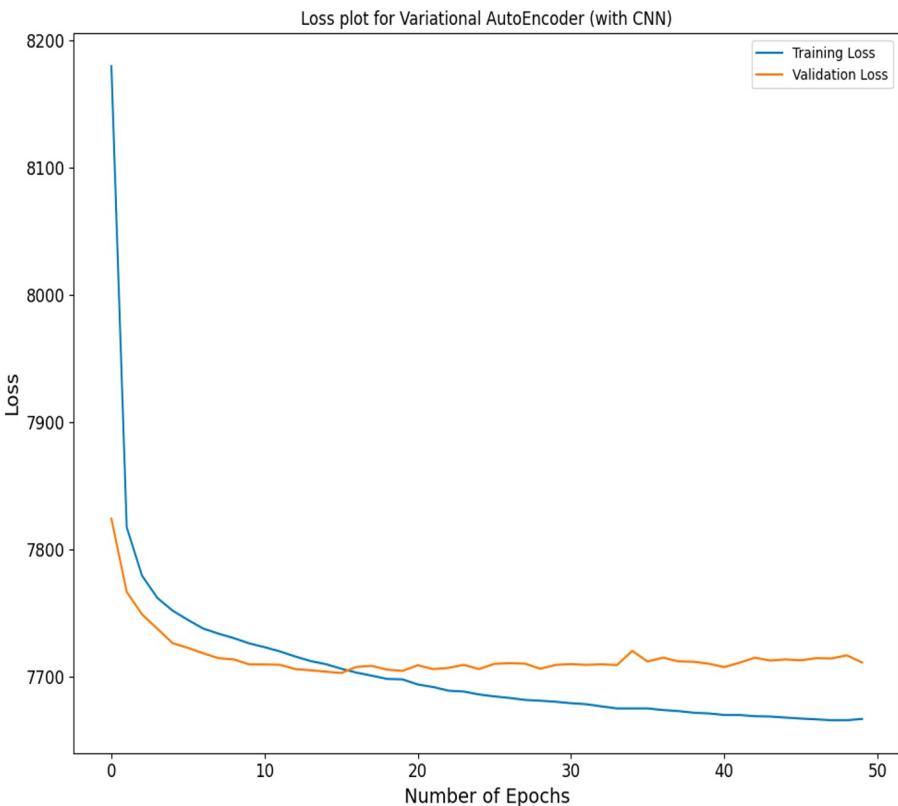


Input STFT for Digit 5

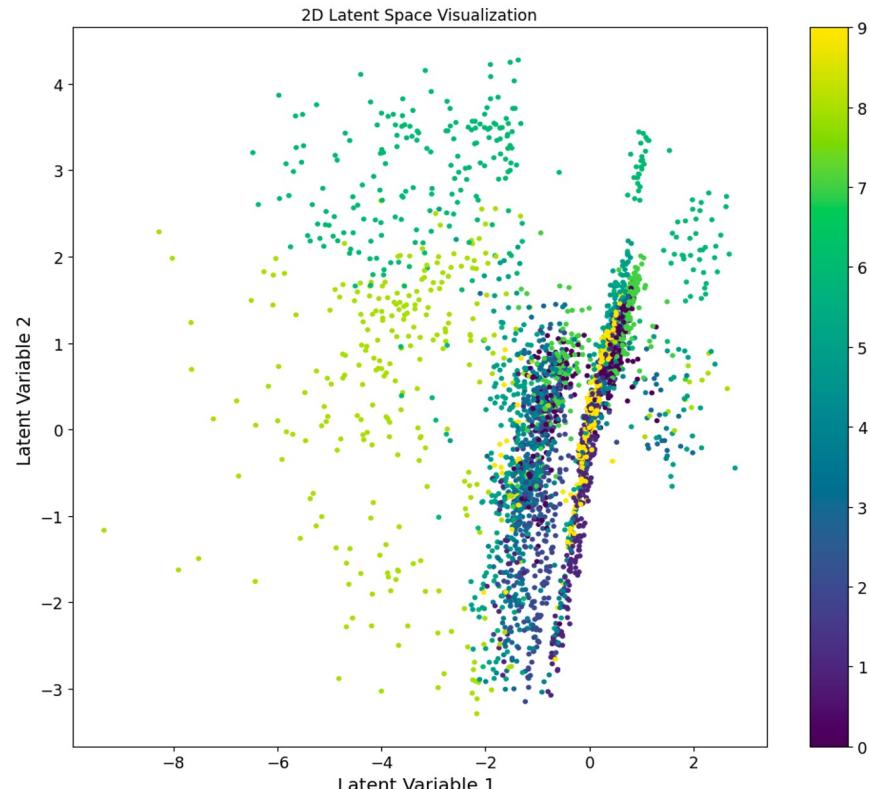
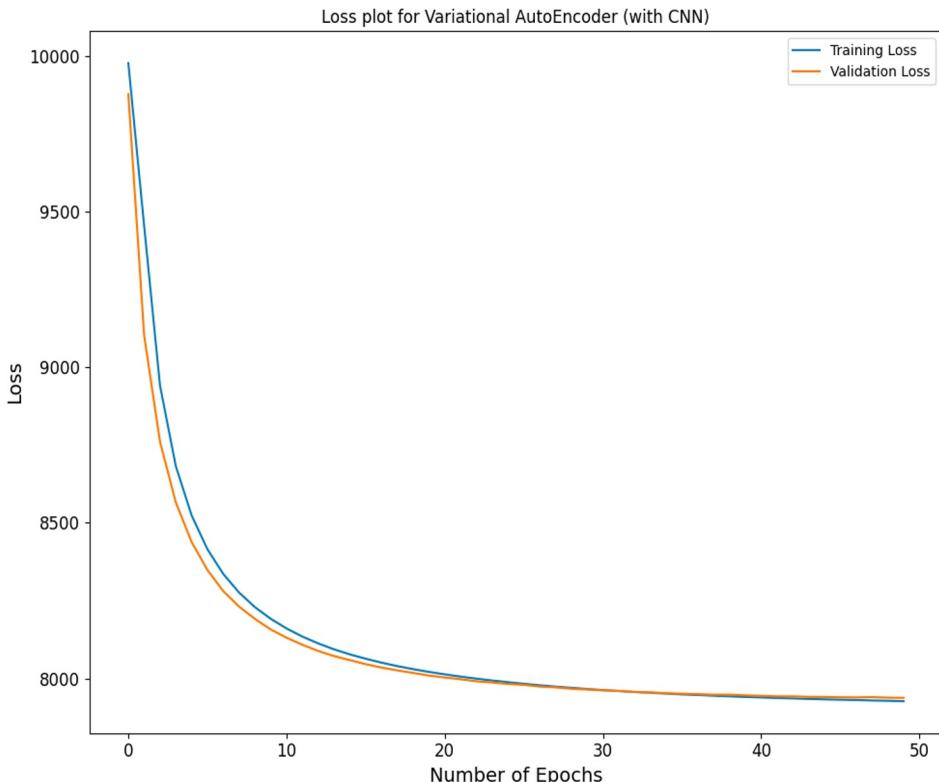


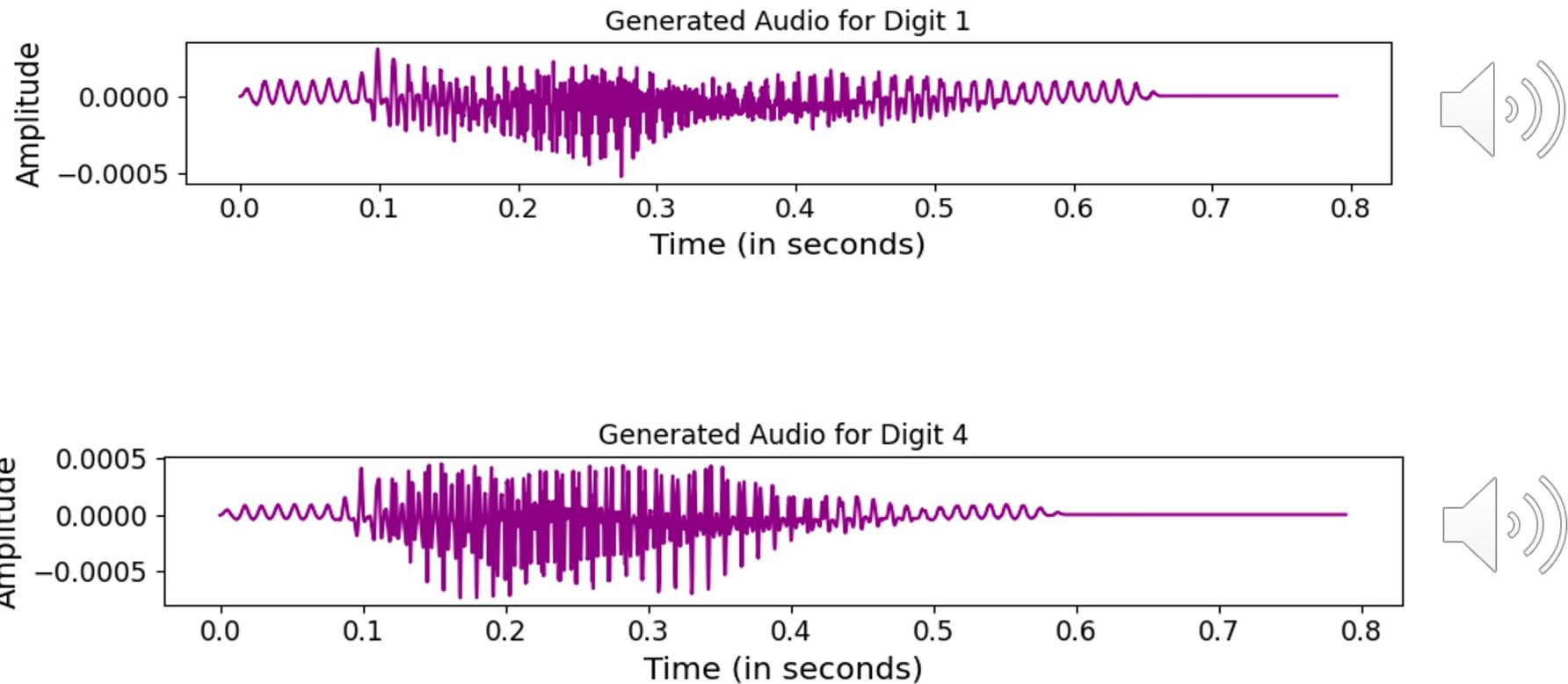
Generated STFT for Digit 5

Results [VAE - CNN Model Performance]

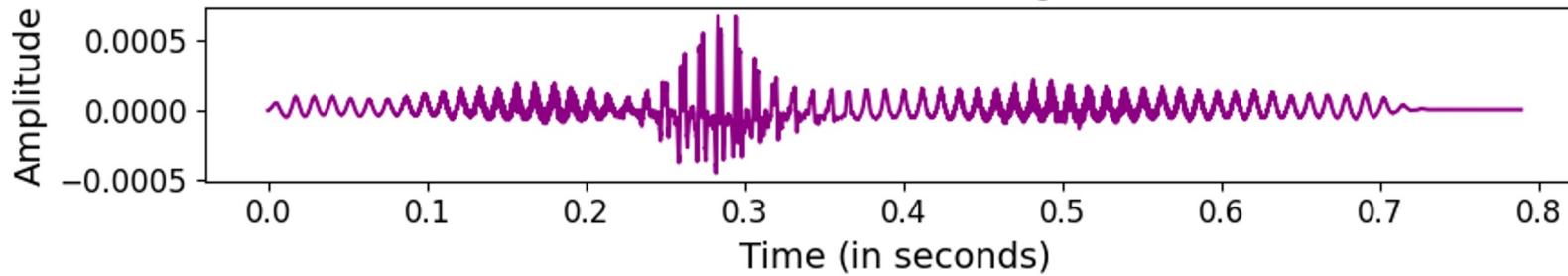


Results [VAE - CNN Model Performance with Regularization]

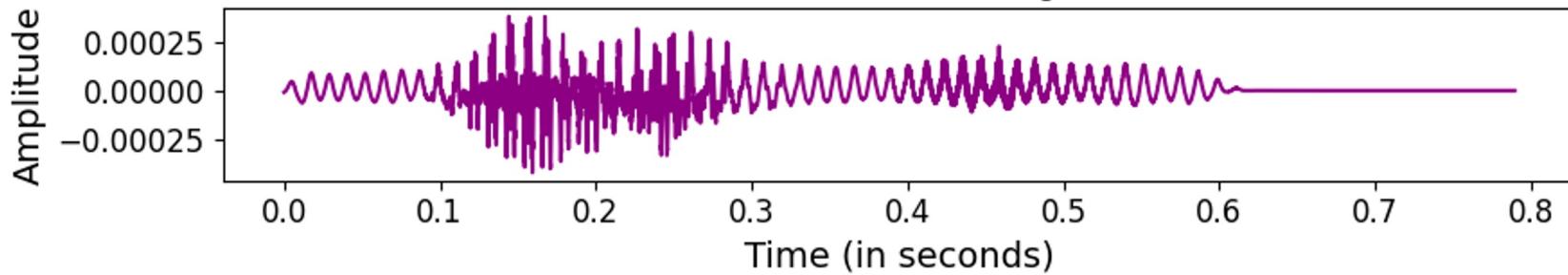




Generated Audio for Digit 6



Generated Audio for Digit 8



Results [MLP-VAE vs CNN-VAE]

Metrics \ Model	MLP-VAE Model 1	MLP-VAE Model 2	MLP-VAE Regularized Model	CNN-VAE Model 1	CNN-VAE Regularized Model
Epochs	50	100	100	50	50
Training Time (on Quadro k2200)	2.1 hours	5.3 hours	6.1 hours	18.7 hours	20.41 hours
Training Loss	7673.74	7728.95	7738.79	7927.34	7987.90
Validation Loss	7667.33	7697.56	7749.78	7937.97	7989.33
Test Loss	7692.69	7707.67	7773.57	7962.82	7993.68

Discussion and Conclusion

● Discussion

- Performance can be further improved by adding more training data, fine tuning model's parameters (size of latent space, regularization parameters, etc),
- This project can be extended to dataset that contains sentences instead of discrete digits.

● Conclusion

- A total five generative models were trained on the Audio-MNIST dataset.
- Regularized VAE with MLP (100 epochs) performed the best for generation task as it had the most distinct latent representation for the spoken digits.
- This project sheds light on the potential of VAEs for speech synthesis and their ability to achieve enhanced fidelity and distinctiveness in generating spoken digits.

Project Source code available at: <https://github.com/Bot-Ro-Bot/Exploring-Gen-AI-for-Audio-Synthesis>

THANK YOU !

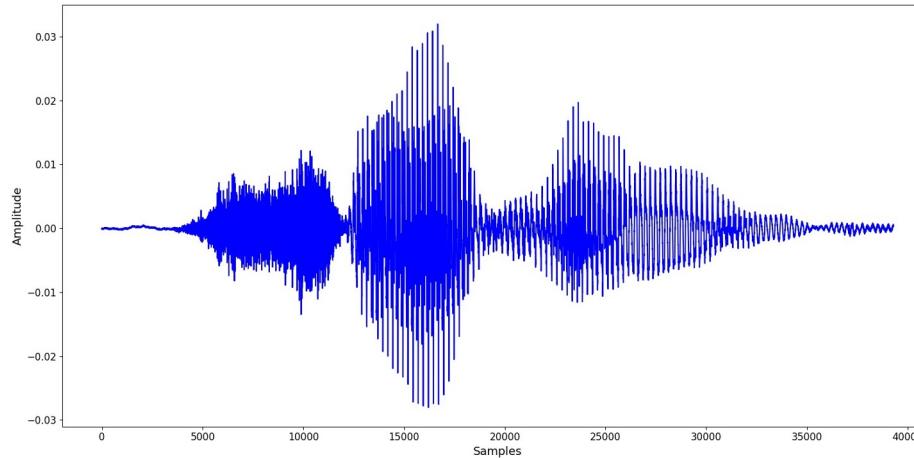
REFERENCES

- [1] L. Weng, “From Autoencoder to Beta-VAE,” Lil’Log, Aug. 12, 2018. <https://lilianweng.github.io/posts/2018-08-12-vae/> (accessed Sep. 30, 2023).
- [2] S. Becker, M. Ackermann, S. Lapuschkin, K.-R. Müller, and W. Samek, “Interpreting and Explaining Deep Neural Networks for Classification of Audio Signals,” arXiv.org, Jul. 09, 2018. <https://arxiv.org/abs/1807.03418>
- [3] <https://agustinus.kristia.de/techblog/2016/12/17/conditional-vae/>

Appendix

AudioMNIST Dataset

- AudioMNIST Dataset has an audio recording of all the digits (from 0 to 9), from a total of 60 speakers, totaling 30,000 samples. [2]
- 9.5 hours long and is recorded from speakers with a broad range of accents, including both native and non-native speakers.
- The recorded sample is 16-bit single-channel audio, sampled at 48KHz.



VAE SIMPLE MODEL ARCHITECTURE

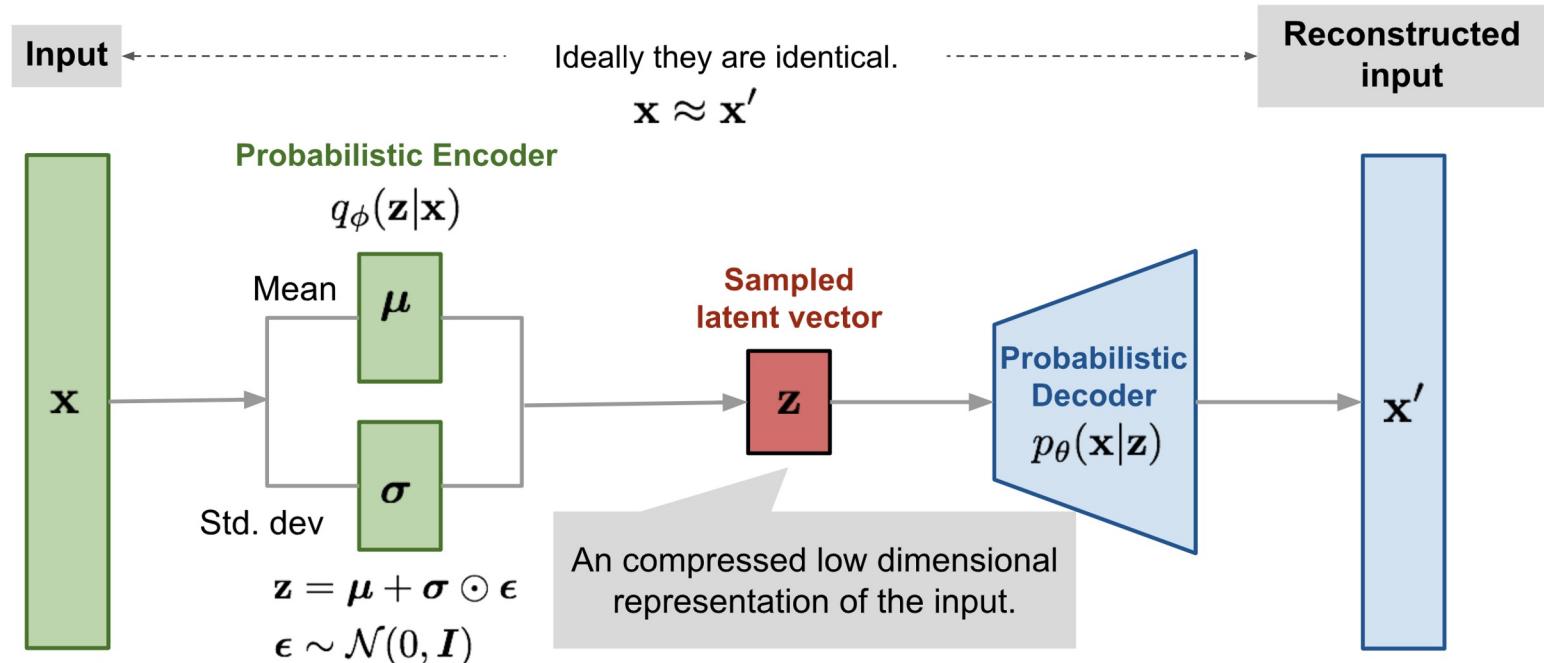


Fig 3: Architecture of Variational AutoEncoders [1]

Code Snippets [Extracting Data and Length Normalization]

Normalize Data Lengths

```
sample_len = [len(x) for x in audios]
print("1st Percentile: ",np.percentile(sample_len,1))
print("3rd Percentile: ",np.percentile(sample_len,3))
print("5th Percentile: ",np.percentile(sample_len,5))
print("10th Percentile: ",np.percentile(sample_len,10))
print("95th Percentile: ",np.percentile(sample_len,95))
print("97th Percentile: ",np.percentile(sample_len,97))
```

```
1st Percentile:  9391.94
3rd Percentile: 10150.0
5th Percentile: 10535.0
10th Percentile: 11168.0
95th Percentile: 18502.0
97th Percentile: 19185.03
```

```
def fetch_data(**kwargs):
    audio = []
    labels = []
    for file in tqdm.tqdm(files,desc="Reading Data ... "):
        _audio, SR = librosa.load(file,sr=22050,dtype=np.float32)

        if(kwargs):
            length = kwargs["length"]
            min_allowed_len = kwargs["min"]
            max_allowed_len = kwargs["max"]
            if((len(_audio) > min_allowed_len) and (len(_audio) < max_allowed_len)):
                if(len(_audio)>length):
                    trim_front = math.floor((len(_audio) - length) / 2 )
                    trim_back = math.ceil((len(_audio) - length) / 2 )
                    _audio = _audio[trim_front:-trim_back]
                elif(len(_audio)<length):
                    _audio = np.pad(_audio,(0,length-len(_audio)),constant_values=(0.0,0.0))
                else:
                    continue
            audio.append(_audio)
            labels.append(os.path.split(file)[-1][0])
    return audio,SR,labels
```

Code Snippets [MLP Encoder & Decoder]

```
# see encoder summary  
encoder.summary()
```

✓ 0.0s

Model: "encoder"

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 17733)]	0	[]
dense_7 (Dense)	(None, 1024)	1815961	['input_2[0][0]']
		6	
dense_8 (Dense)	(None, 512)	524800	['dense_7[0][0]']
dense_9 (Dense)	(None, 256)	131328	['dense_8[0][0]']
z_mean (Dense)	(None, 2)	514	['dense_9[0][0]']
z_log_var (Dense)	(None, 2)	514	['dense_9[0][0]']
z (Lambda)	(None, 2)	0	['z_mean[0][0]', 'z_log_var[0][0]']

Total params: 18816772 (71.78 MB)

Trainable params: 18816772 (71.78 MB)

Non-trainable params: 0 (0.00 Byte)

```
# see decoder summary  
decoder.summary()
```

✓ 0.0s

Model: "decoder"

Layer (type)	Output Shape	Param #
decoder_input (InputLayer)	[(None, 2)]	0
dense_10 (Dense)	(None, 256)	768
dense_11 (Dense)	(None, 512)	131584
dense_12 (Dense)	(None, 1024)	525312
dense_13 (Dense)	(None, 17733)	18176325

Total params: 18833989 (71.85 MB)

Trainable params: 18833989 (71.85 MB)

Non-trainable params: 0 (0.00 Byte)

Code Snippets [CNN Encoder & Decoder]

```
encoder = Model(inputs, z)
encoder.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_1 (InputLayer)	[None, 257, 69, 1]	0	[]
conv2d (Conv2D)	(None, 257, 69, 32)	320	['input_1[0][0]']
conv2d_1 (Conv2D)	(None, 257, 69, 64)	18496	['conv2d[0][0]']
flatten (Flatten)	(None, 1134912)	0	['conv2d_1[0][0]']
dropout (Dropout)	(None, 1134912)	0	['flatten[0][0]']
dense (Dense)	(None, 256)	2905377 28	['dropout[0][0]']
dropout_1 (Dropout)	(None, 256)	0	['dense[0][0]']
dense_1 (Dense)	(None, 2)	514	['dropout_1[0][0]']
dense_2 (Dense)	(None, 2)	514	['dropout_1[0][0]']
lambda (Lambda)	(None, 2)	0	['dense_1[0][0]', 'dense_2[0][0]']
<hr/>			

Total params: 290557572 (1.08 GB)

Trainable params: 290557572 (1.08 GB)

Non-trainable params: 0 (0.00 Byte)

```
decoder.summary()
```

Model: "model_1"

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[None, 2]	0
dense_3 (Dense)	(None, 256)	768
dense_4 (Dense)	(None, 1134912)	291672384
reshape (Reshape)	(None, 257, 69, 64)	0
conv2d_transpose (Conv2DTranspose)	(None, 257, 69, 32)	18464
conv2d_transpose_1 (Conv2DTranspose)	(None, 257, 69, 1)	289
<hr/>		

Total params: 291691905 (1.09 GB)

Trainable params: 291691905 (1.09 GB)

Non-trainable params: 0 (0.00 Byte)

Code Snippets [MLP VAE Architecture]

```
# Encoder network
inputs = Input(shape=(X_train.shape[1],))

x = Dense(1024, activation='relu')(inputs)
x = Dense(512, activation='relu')(x)
x = Dense(256, activation='relu')(x)

z_mean = Dense(latent_dim, name='z_mean')(x)
z_log_var = Dense(latent_dim, name='z_log_var')(x)

def sampling(args):
    z_mean, z_log_var = args
    batch_size = K.shape(z_mean)[0]
    epsilon = K.random_normal(shape=(batch_size, latent_dim), mean=0., stddev=1.)
    return z_mean + K.exp(0.5 * z_log_var) * epsilon

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

# Decoder network
decoder_inputs = Input(shape=(latent_dim,), name='decoder_input')
x = Dense(256, activation='relu')(decoder_inputs)
x = Dense(512, activation='relu')(x)
x = Dense(1024, activation='relu')(x)
outputs = Dense(17733, activation='sigmoid')(x)

# Define the encoder and decoder models
encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
decoder = Model(decoder_inputs, outputs, name='decoder')

✓ 0.0s
```

```
# Define the loss function for VAE
def vae_loss(inputs, x_decoded_mean):
    recon_loss = original_dim * binary_crossentropy(inputs, x_decoded_mean)
    kl_loss = -0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
    return K.mean(recon_loss + kl_loss)
```

✓ 0.0s

```
# VAE model
outputs = decoder(encoder(inputs)[2])
dense_vae = Model(inputs, outputs, name='vae')
dense_vae.compile(optimizer='adam', loss=vae_loss)
dense_vae.summary()
```

✓ 0.0s

Model: "vae"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 17733)]	0
encoder (Functional)	[(None, 2), (None, 2), (None, 2)]	18816772
decoder (Functional)	(None, 17733)	18833989
<hr/>		
Total params: 37650761 (143.63 MB)		
Trainable params: 37650761 (143.63 MB)		
Non-trainable params: 0 (0.00 Byte)		

Code Snippets [CNN VAE Architecture]

```
# Encoder network
inputs = Input(shape=(X_train.shape[1],))

x = Dense(1024, activation='relu')(inputs)
x = Dense(512, activation='relu')(x)
x = Dense(256, activation='relu')(x)

z_mean = Dense(latent_dim, name='z_mean')(x)
z_log_var = Dense(latent_dim, name='z_log_var')(x)

def sampling(args):
    z_mean, z_log_var = args
    batch_size = K.shape(z_mean)[0]
    epsilon = K.random_normal(shape=(batch_size, latent_dim), mean=0., stddev=1.)
    return z_mean + K.exp(0.5 * z_log_var) * epsilon

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

# Decoder network
decoder_inputs = Input(shape=(latent_dim,), name='decoder_input')
x = Dense(256, activation='relu')(decoder_inputs)
x = Dense(512, activation='relu')(x)
x = Dense(1024, activation='relu')(x)
outputs = Dense(17733, activation='sigmoid')(x)

# Define the encoder and decoder models
encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
decoder = Model(decoder_inputs, outputs, name='decoder')

✓ 0.0s
```

```
# Define the loss function for VAE
def vae_loss(inputs, x_decoded_mean):
    recon_loss = original_dim * binary_crossentropy(inputs, x_decoded_mean)
    kl_loss = -0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
    return K.mean(recon_loss + kl_loss)
```

✓ 0.0s

```
# VAE model
outputs = decoder(encoder(inputs)[2])
dense_vae = Model(inputs, outputs, name='vae')
dense_vae.compile(optimizer='adam', loss=vae_loss)
dense_vae.summary()
```

✓ 0.0s

Model: "vae"

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[(None, 17733)]	0
<hr/>		
encoder (Functional)	[(None, 2), (None, 2), (None, 2)]	18816772
<hr/>		
decoder (Functional)	(None, 17733)	18833989
<hr/>		
Total params: 37650761 (143.63 MB)		
Trainable params: 37650761 (143.63 MB)		
Non-trainable params: 0 (0.00 Byte)		

Code Snippets [Amplitude Normalization using MinMax]

```
def MinMaxScaler(features):
    scaled = []
    min_max_values = []
    for f in features:
        min_val = np.min(f)
        max_val = np.max(f)
        f_normalized = (f - min_val) / (max_val - min_val)
        min_max_values.append((min_val,max_val))
        scaled.append(f_normalized)
    return np.array(scaled),np.array(min_max_values)
```

✓ 0.0s

```
def MinMaxUnScaler(features, min_max_values):
    unscaled = []
    for f,min_max in zip(features,min_max_values):
        f = f.reshape(257,69)
        unscaled_feature = (f * (min_max[1] - min_max[0])) + min_max[0]
        unscaled_feature = librosa.db_to_amplitude(unscaled_feature)
        unscaled.append(unscaled_feature)
    return np.array(unscaled)
```

✓ 0.0s

Code Snippets [Feature Extraction (STFT)]

Feature Extraction

```
def extract_stft(audios):
    stft_features = []
    for audio in tqdm.tqdm(audios,desc="Extracting Features ... "):
        stft = librosa.amplitude_to_db(np.abs(librosa.stft(audio,n_fft=512,hop_length=256)), ref=np.max)
        stft = stft.reshape(*stft.shape, 1) # Add channel dimension

        stft_features.append(stft)
    return np.array(stft_features)
```

```
X_train_features = extract_stft(X_train)
X_test_features = extract_stft(X_test)
X_val_features = extract_stft(X_val)
```

```
Extracting Features ... : 100%|██████████| 22840/22840 [00:15<00:00, 1502.06it/s]
Extracting Features ... : 100%|██████████| 2820/2820 [00:02<00:00, 1265.28it/s]
Extracting Features ... : 100%|██████████| 2538/2538 [00:02<00:00, 1167.23it/s]
```

Code Snippets [iSTFT]

Invserset STFT

```
def ISTFT(feature):
    audios = []
    for f in tqdm.tqdm(feature,desc="Appling inverse STFT ..."):
        audio = librosa.istft(f,hop_length=256)
        audios.append(audio)
    return np.array(audios)
```

✓ 0.0s

```
decoded_istft = ISTFT(decoded_stft)
```

✓ 1.4s

Appling inverse STFT ...: 100%|| 2538/2538 [00:01<00:00, 1818.40it/s]

Code Snippets [Train-Test-Val Split]

Train test split

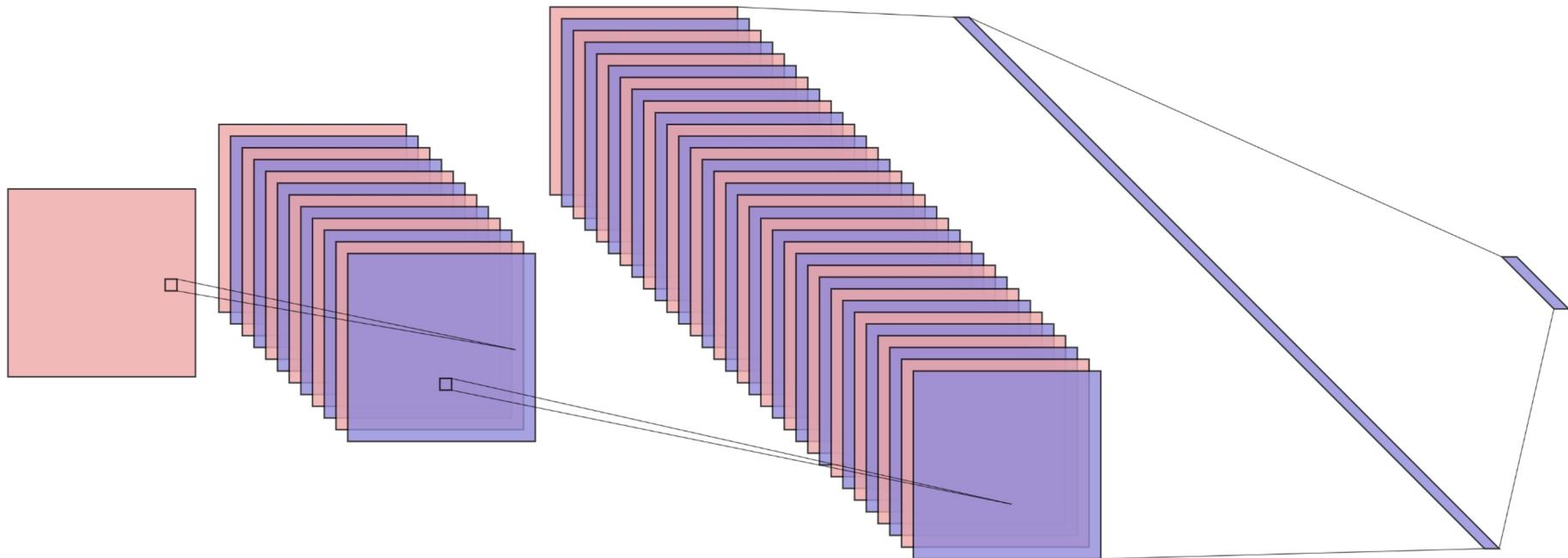
```
# Perform an initial split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(audios, labels, test_size=0.1, random_state=42, stratify=labels)

# Perform an second split into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.1, random_state=42, stratify=y_train)

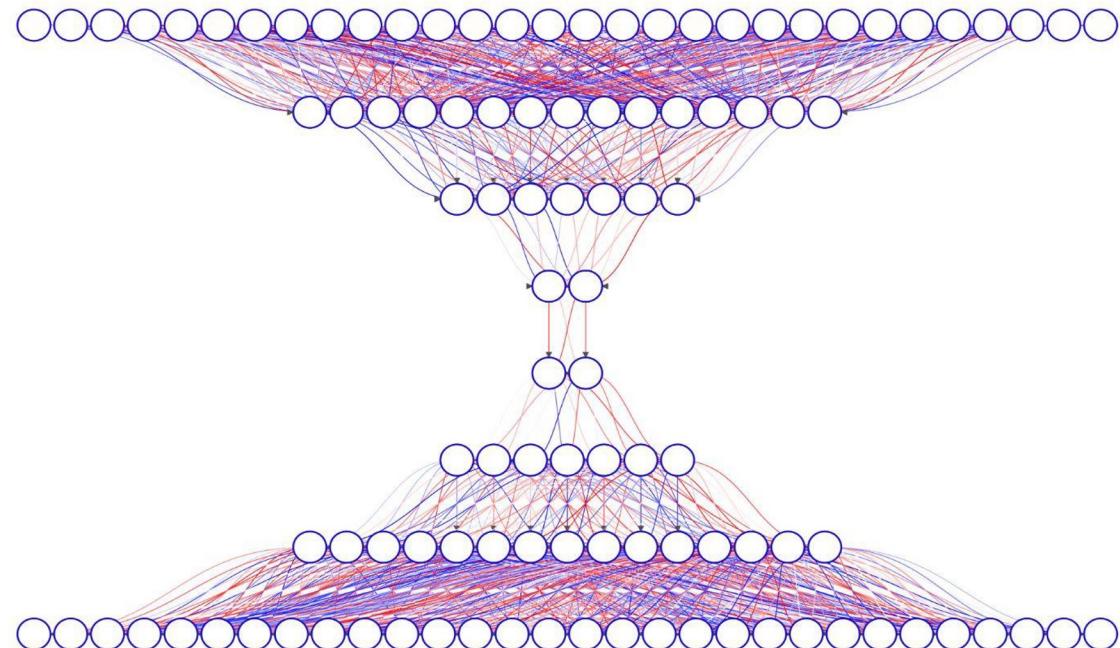
print("Shape of training data: ", X_train.shape)
print("Shape of test data: ", X_test.shape)
print("Shape of validation data: ", X_val.shape)
```

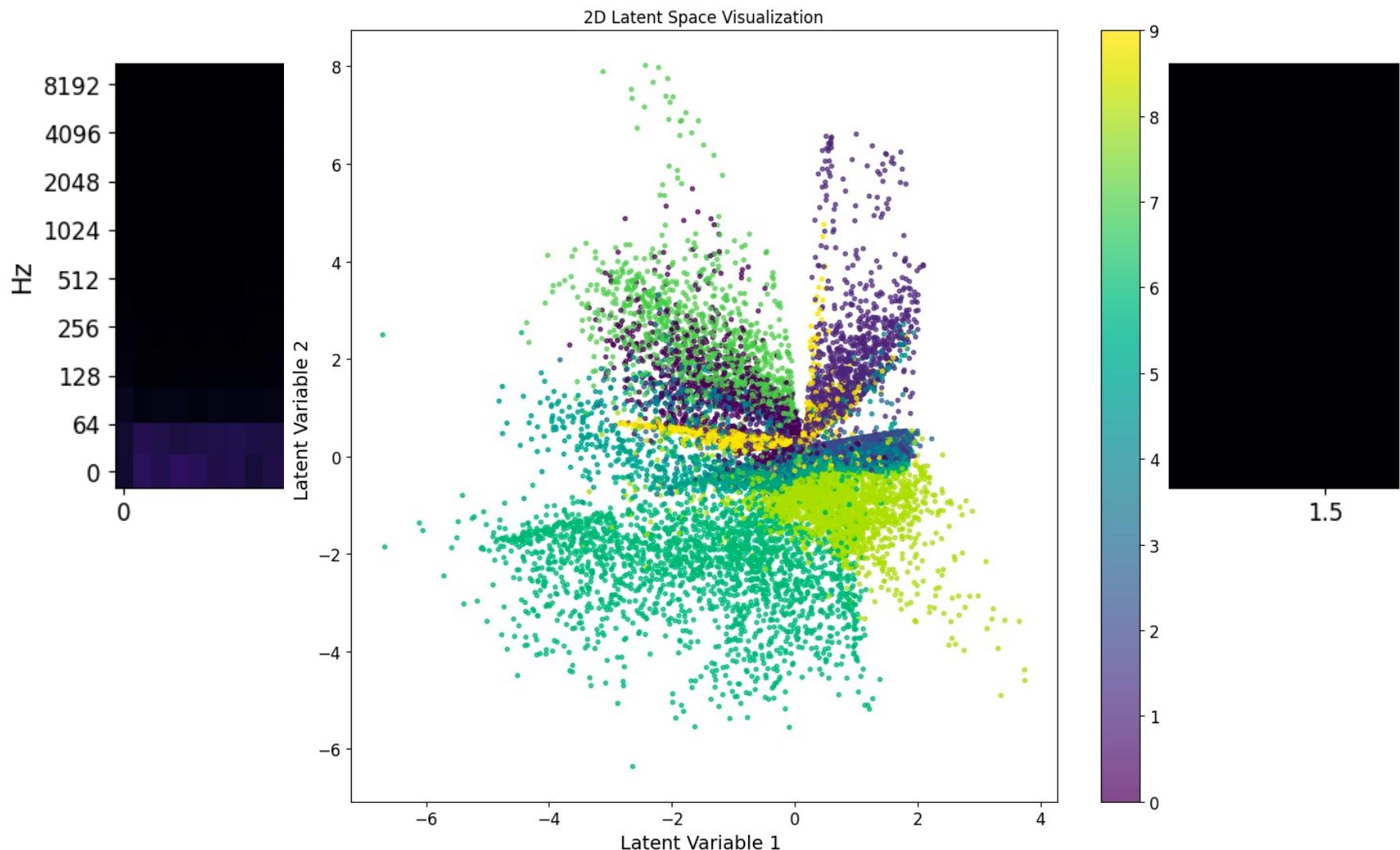
```
Shape of training data: (22840, 17640)
Shape of test data: (2820, 17640)
Shape of validation data: (2538, 17640)
```

CNN Model Drawn in NN-SVG



MLP Model Drawn in NN-SVG





2D Latent Space Visualization

