University Tunis El Manar
National Engineering School of Tunis

Information and Communication Technologies Department

End-of-Year Project

Created by :

Mohamed MAHDI

Class :

2ATEL 1

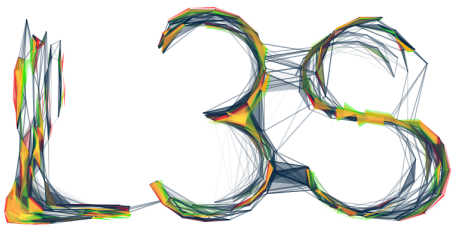
Implicit Neural Representations for High-Fidelity Signal Modeling and Synthesis

Supervised by :

Mme. Sonia Djaziri-Larbi

Conducted at :

Signals & Smart Systems Lab (L3S) ENIT



2024-2025

Abstract

This project investigates the use of Implicit Neural Representations (INRs), specifically Sinusoidal Representations Networks (SIRENs), for representing and processing audio signals. The main objective is to explore the potential of SIRENs in compressing audio data while preserving high-quality reconstruction. In addition, the use of INRs presents opportunities for restoring and enhancing old or degraded audio, improving its continuity and realism, thus making it more similar to natural sound. The project examines the performance of these models in audio compression, enhancement, and synthesis, highlighting the challenges of hardware limitations and the potential benefits of high-performance data centers. It concludes that with further advancements in hardware, INRs could revolutionize audio processing by offering efficient bandwidth usage while maintaining perceptual audio quality.

Keywords : Implicit Neural Representations, SIREN, Audio Compression, Audio Enhancement, Neural Networks, Signal Processing, Audio Restoration

Ce projet examine l'utilisation des Représentations Neuronales Implicites (RNI), en particulier les Réseaux de Représentations Sinusoïdales (SIREN), pour la représentation et le traitement des signaux audio. L'objectif principal est d'explorer le potentiel des SIREN dans la compression des données audio tout en préservant une reconstruction de haute qualité. De plus, l'utilisation des RNI ouvre des perspectives pour restaurer et améliorer les anciens audios ou ceux dégradés, en améliorant leur continuité et leur réalisme, les rendant ainsi plus proches du son naturel. Le projet examine les performances de ces modèles dans la compression audio, l'amélioration et la synthèse, en soulignant les défis liés aux limitations matérielles et les avantages potentiels des centres de données hautes performances. Il conclut qu'avec les progrès futurs du matériel, les RNI pourraient révolutionner le traitement audio en offrant une utilisation efficace de la bande passante tout en maintenant une qualité audio perceptuelle.

Mots-clés : Représentations Neuronales Implicites, SIREN, Compression Audio, Amélioration Audio, Réseaux Neuronaux, Traitement du Signal, Restauration Audio

Acknowledgments

First and foremost, I would like to thank God for giving me the strength, patience, and determination to complete this project. Without this inner support, the journey would have been much harder.

I am deeply grateful to my supervisor, Mrs. **Sonia Djaziri Larbi**, Director of Signals & Smart Systems lab (L3S) of ENIT, for her continuous guidance and collaboration. Her insightful advice helped me think more critically and move forward with clarity and confidence.

I also sincerely thank Mr. **Anas Ghrab**, researcher at L3S, for granting me access to the powerful L3S Lab servers. The outstanding hardware resources played a key role in speeding up my experiments and pushing the project further.

My thanks go as well to ENIT, and especially to the director of the TIC department, Madame **Imen Elloumi**, for ensuring an excellent level of organization throughout this experience, and to the director of ENIT for supporting such an environment of learning and research.

I would also like to acknowledge the global community working on implicit neural representations. Their inspiring work laid a strong foundation for this project and continues to drive innovation in the field.

Finally, I want to take a moment to thank myself for the long hours, the persistence, and the sacrifices made along the way. This achievement would not have been possible without that commitment.

Table of Contents

List of Figures

Abbreviations

AI	Artificial Intelligence
CD	Compact Disc
DAG	Directed Acyclic Graph
DL	Deep Learning
FSMN	Feedforward Sequential Memory Networks
FNN	Feedforward Neural Network
INR	Implicit Neural Representations
MLP	Multilayer Perceptron
MPEG	Moving Picture Experts Group
NN	Neural Network
PCM	Pulse Code Modulation
PESQ	Perceptual Evaluation of Speech Quality
Relu	Rectified Linear Unit
SIREN	Sinusoidal Representation Networks
STOI	Short-Time Objective Intelligibility
VISQOL	Visual Quality of Speech
WAV	Waveform Audio File Format
CDPAM	Content-Dependent Perceptual Audio Model

General Introduction

In today's digital world, working with audio signals has become more important than ever. Whether it's in communication, entertainment, or media technologies, there is a constant need to represent, analyze, and improve audio quality. At the same time, the rapid development of artificial intelligence, especially deep learning, has opened new doors for handling audio in ways that were not possible before. These advances bring fresh solutions for compressing and reconstructing complex signals.

Despite the progress, compressing and reconstructing audio signals remains a real challenge. Traditional methods, based on linear transformations or explicit representations, often fail to fully capture the richness of complex sounds. Recently, a new approach called Implicit Neural Representations (INRs), and especially models like SIRENs that use periodic activation functions, has shown promising results. These models offer a new way to represent continuous signals, including audio, but applying them effectively requires a deep understanding of both the models and the best practices in training and evaluation.

In this project, we aim to apply SIRENs to model, compress, and enhance audio signals. After building a strong understanding of signal basics and deep learning techniques, we developed a complete solution combining SIREN-based interpolation with a refinement stage using the MossFormer2 model. Our goal is to offer a new pipeline that can compress audio efficiently while maintaining, and even improving, the perceived quality.

Throughout this report, we will guide the reader step by step, starting with the essential concepts needed to understand our work. We will first explore the basics of signals and their properties, before moving into the world of deep learning and neural networks. Building on this foundation, we will understand implicit neural representations, uncovering how they can model complex data like audio signals. Finally, we will reveal the details of our proposed solution, the experiments we conducted, and the results we achieved. Each chapter is carefully structured to build curiosity and bring the reader closer to the heart of our project.

CHAPTER 1

Audio Signals Representations and Analysis

Introduction

To understand how implicit neural networks represent audio signals, we first need to grasp the nature of these signals themselves[7]. This chapter lays out the basic mathematical framework for modeling audio signals, preparing the ground for later discussions on implicit neural representations.

Mathematically, audio signals can be expressed using sinusoidal basis functions. A key component of this approach is the complex exponential function :

$$x(t) = Ae^{j2\pi ft}$$

where :

- A is the amplitude of the wave,
- f is the frequency,
- j is the imaginary unit.
- t is the time.

Fourier analysis offers a powerful way to break down signals, revealing that the set of exponentials $e^{j2\pi ft}$, spanning all frequencies in $\mathbb{R}$, forms an orthogonal basis for the space of functions. In simpler terms, this means any audio signal can be represented as a sum of sinusoidal components, each with a distinct frequency.

To keep our focus on practical applications, this chapter skips formal proofs. The rigorous derivations behind classical Fourier theory, while foundational, are beyond our scope here. Instead, we start with the core Fourier transform formulas and their well-known properties, using them as a launching pad. We also introduce the notation needed to connect these ideas to implicit neural representations later on.

1.1 Continuous-Time and Discrete-Time Signals

Signals can be classified into two main types : continuous-time signals and discrete-time signals. These two categories represent different ways in which the independent variable (usually time) is treated in relation to the signal. In this section, we describe both continuous-time and discrete-time signals.

1.1.1 Continuous-Time Audio Signals

Continuous-time audio signals are defined for a continuum of values of the independent variable, typically time. In this case, the independent variable, denoted as t, can take on any real value, meaning that the signal is defined at every point in time. A continuous-time audio signal represents the continuous variations in acoustic pressure, such as sound waves traveling through the air.

For a continuous-time audio signal $x(t)$, the signal is defined at all instances of time as shown in Figure 1.1, and the variations in acoustic pressure are smooth, continuous, and uninterrupted. This allows for the representation of sound with high fidelity, capturing all the nuances and details that are present in the signal at every moment. The continuous nature of the signal enables precise measurements and analysis of sound waves at any given point in time.

Continuous-time audio signals allow for a detailed representation of the signal without losing important information, making it suitable for applications where high resolution and accuracy are required, such as in sound synthesis or high-quality audio processing.

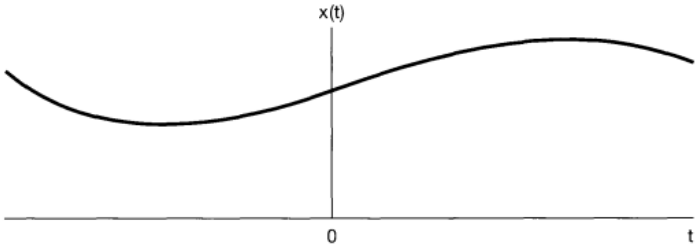


FIGURE 1.1 – Representation of a continuous-time audio signal $x(t)$, illustrating its smooth and uninterrupted nature[7].

1.1.2 Discrete-Time Audio Signals

Discrete-time audio signals, on the other hand, are defined only at discrete points in time as shown in Figure 1.2. These signals are created by sampling a continuous-time audio signal at specific intervals. The independent variable, denoted as n, represents the discrete time indices at which the signal is sampled. In discrete-time audio signals, the acoustic pressure is only known at specific, evenly spaced time points, with the continuous waveform approximated by these samples.

For example, in digital audio systems, a continuous-time audio signal is sampled at a fixed rate to create a discrete-time signal. The resulting signal $x[n]$ is defined only at discrete values of n, and each sample represents the acoustic pressure at a specific point in time. The choice of sampling rate is crucial, as it determines how accurately the discrete-time signal represents the original continuous-time signal. If the sampling rate is too low, aliasing can occur, causing a loss of information and distortion in the reproduced sound.

Discrete-time audio signals are widely used in modern digital audio systems, including audio recording, editing, and playback. Digital processors operate on these discrete-time signals, performing tasks like filtering, compression, and synthesis. In these systems, the continuous sound wave is represented by a sequence of samples, which makes it easier to store, manipulate, and transmit audio data, but at the cost of losing the infinite resolution of the continuous signal.

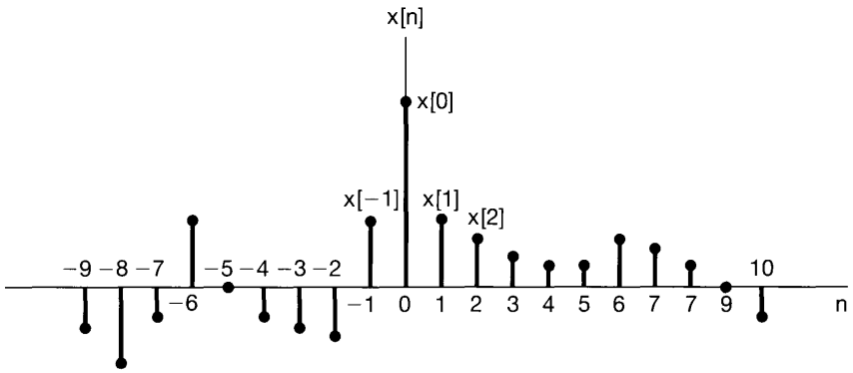


FIGURE 1.2 – Representation of a discrete-time audio signal $x[n]$, obtained by sampling a continuous-time signal at fixed intervals. Each discrete sample corresponds to an acoustic pressure measurement at a specific point in time[7].

1.2 Sampling and Quantization

When working with real-world audio signals, we need a way to store and process them digitally. This is where sampling and quantization come into play they allow us to convert a continuous sound wave into a format that computers can handle.

1.2.1 Sampling

Imagine trying to record a song by writing down the air pressure at every single moment. Since there are infinitely many moments, this would be impossible ! Instead, we take measurements at regular time intervals, a process known as sampling. These snapshots capture the signal's behavior over time and allow us to reconstruct it later. The key is choosing an appropriate sampling rate too low, and we lose important details ; too high, and we waste storage space.

The famous Nyquist-Shannon sampling theorem tells us that to perfectly reconstruct a signal, we must sample it at least twice the highest frequency present :

$$f_s \geq 2f_{\max} \tag{1.1}$$

This is why CDs use a sampling rate of 44.1 kHz to accurately capture sounds up to about 22 kHz, which is the upper limit of human hearing.

1.2.2 Quantization

Once we have sampled our signal, we need to store each measurement as a number. However, computers don't have infinite precision, so we round each value to the closest available level in a process called quantization. This introduces a small error, known as quantization noise, but with enough precision (bits per sample), this error can be minimized.

In uniform quantization, the range of possible values is divided into equal-sized intervals. Any sampled value is then assigned to the nearest interval. If a value is too large or too small to fit within our chosen range, it gets clipped. The real interval $[-\alpha, \alpha]$ is divided into $2^\beta - 1$ bins, each of size :

$$\Delta = \frac{\alpha}{2^\beta} \tag{1.2}$$

The quantized values are obtained using :

$$Q(x_n) = \Delta \left\lfloor \frac{x_n}{\Delta} + 0.5 \right\rfloor \tag{1.3}$$

Together, sampling and quantization, illustrated in Figure 1.3, form the backbone of digital audio. They ensure that rich, analog sounds from the world around us can be faithfully recorded, processed, and played back using digital systems. Understanding these processes is essential, as they directly impact how neural networks interact with audio data.

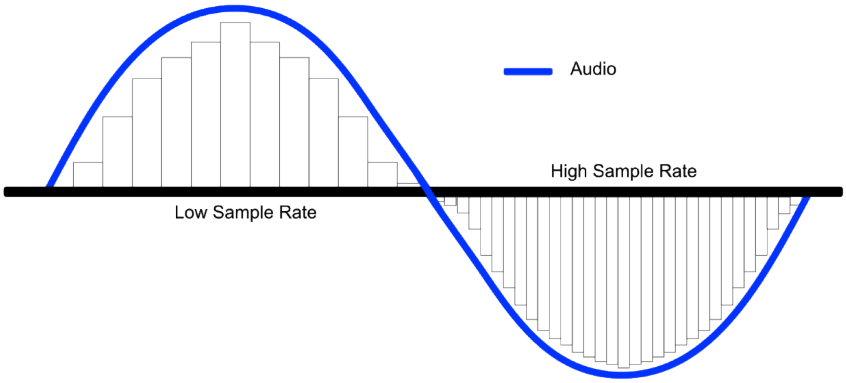


FIGURE 1.3 – Illustration of sampling rates in digital audio processing. The waveform represents an analog audio signal, while the bars indicate sampled values at different rates.

1.3 Exponential and Sinusoidal Audio Signals

When analyzing audio signals, one of the most powerful insights is that any complex sound can be broken down into simpler, periodic components exponentials and sinusoids. These components are not just theoretical, they are practical tools for understanding how sound waves oscillate at different frequencies, amplitudes, and phases. In the continuous-time domain, audio signals can be viewed as a combination of these oscillatory components, where each frequency contributes to the overall texture of the sound.

1.3.1 Continuous-Time Complex Exponential and Sinusoidal Signals

In audio signal processing, a fundamental concept is that complex audio signals can be represented as sums of simpler components, primarily sinusoidal signals. This insight, rooted in Fourier analysis, is based on the idea that any sound, no matter how intricate, can be decomposed into a series of sinusoidal waves, each characterized by its frequency, amplitude, and phase. This approach is not only theoretical but provides practical methods for analyzing and synthesizing sound.

At the core of this decomposition are complex exponential signals, which take the general form $x(t) = Ce^{\alpha t}$, where C and α are complex numbers. When α is real, the exponential signal exhibits either growth or decay. This can be observed in physical phenomena, such as the fading of sound after a note is played, which is mathematically represented by a decaying exponential. Conversely, a growing exponential could describe the amplification of a sound signal.

However, when α is purely imaginary, the signal becomes periodic, oscillating in a sinusoidal manner. A complex exponential of the form $e^{i\omega_0 t}$, where ω_0 represents the angular frequency, can be directly related to sinusoidal signals. Using Euler's formula, we express the complex exponential as :

$$e^{i\omega_0 t} = \cos(\omega_0 t) + i\sin(\omega_0 t) \tag{1.4}$$

which illustrates how complex exponentials can be decomposed into real and imaginary components, both of which are sinusoidal waves. These sinusoidal signals, such as $x(t) = A\cos(\omega_0 t+\phi)$, are inherently periodic, making them an ideal representation for pure tones in audio signals.

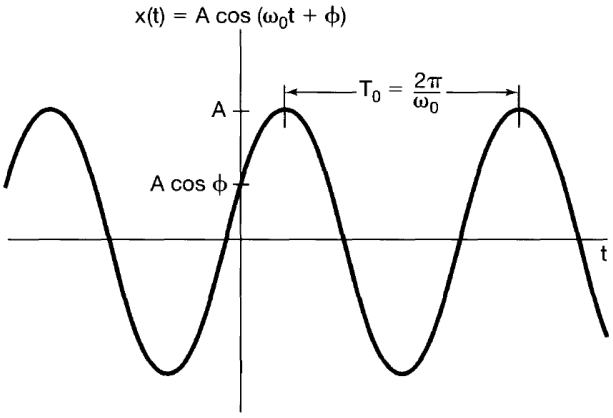


FIGURE 1.4 – Representation of a sinusoidal signal $x(t) = A\cos(\omega_0 t+\phi)$ as a function of time. The amplitude A determines the peak value, while the phase ϕ shifts the waveform along the time axis. The signal is periodic with a fundamental period $T_0 = \frac{2\pi}{\omega_0}$, which is determined by the angular frequency ω_0[7].

The significance of sinusoidal signals lies in their ability to represent virtually any periodic sound. By combining multiple sinusoids at different frequencies, amplitudes, and phases, one can reconstruct complex sounds, from the tone of a musical instrument to the intricacies of

human speech. In fact, a real-world audio signal can be thought of as the sum of many such sinusoids, each contributing a different aspect of the overall sound.
This is mathematically represented by :

$$x(t) = \sum_{k=1}^{N} A_k \cos(\omega_k t + \phi_k) \tag{1.5}$$

where A_k, ω_k, and ϕ_k represent the amplitude, frequency, and phase of the k-th sinusoidal component, respectively. Here, N is the number of sinusoids used to approximate the signal. This sum captures the harmonic structure of the signal, where each sinusoidal component corresponds to a specific frequency (pitch) and contributes to the overall timbre and dynamics of the sound.

1.3.2 Discrete-Time Representation of Audio Signals

In discrete-time signal processing, signals are represented by sequences defined at discrete intervals, which can be considered as samples of their continuous-time counterparts. As discussed in the previous section, complex exponential signals are fundamental in both continuous and discrete time. The discrete-time complex exponential signal takes the form :

$$x[n] = Ce^{j\omega_0 n} \tag{1.6}$$

These discrete-time exponentials can also be expressed as the sum of sinusoidal components using Euler's formula :

$$e^{j\omega_0 n} = \cos(\omega_0 n) + j\sin(\omega_0 n) \tag{1.7}$$

where the real and imaginary parts correspond to cosine and sine waves, respectively. As in continuous time, the resulting sinusoidal signals are periodic but defined only at discrete time instances.

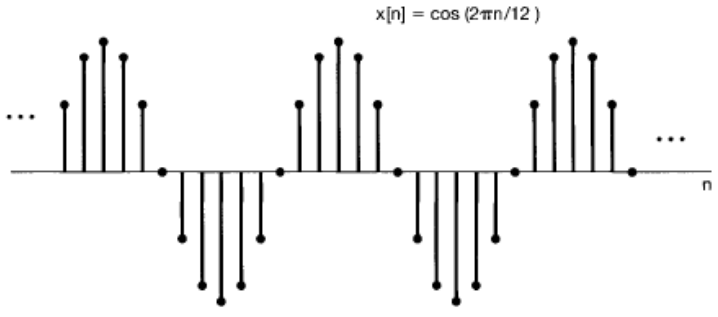


FIGURE 1.5 – Discrete-time sinusoidal signal $x[n] = \cos\left(\frac{2\pi n}{12}\right)$. Unlike continuous-time sinusoids, this signal is defined only at discrete time indices n, forming a sampled version of a continuous cosine wave[7].

Just as a continuous-time signal can be represented as a sum of sinusoids, a real-valued discrete-time signal can also be approximated by the sum of sinusoids at different frequencies, ampli-

tudes, and phases. This is similar to the process described in the previous section for continuous-time signals. For a discrete-time audio signal $x[n]$, we can express it as :

$$x[n] = \sum_{k=1}^{N} A_k \cos(\omega_k n + \phi_k) \tag{1.8}$$

The process of sampling, which is the core concept behind discrete-time signals, involves measuring the continuous signal at regular intervals. When sampled at a sufficiently high rate, the discrete-time version closely approximates the continuous signal, maintaining the essential structure of the original waveform.

1.4 Fourier Series Representation of Audio Signals

Fourier analysis provides a fundamental framework for understanding and representing periodic signals. The central idea is that any periodic signal can be expressed as a sum of sinusoids with different frequencies, amplitudes, and phases. While the original formulation by Joseph Fourier was met with skepticism, it has since become a cornerstone of signal processing, enabling the analysis and synthesis of complex waveforms.

For a periodic signal $x(t)$ with period T, the Fourier series representation is given by :

$$x(t) = a_0 + \sum_{k=1}^{\infty} [a_k \cos(2\pi k f_0 t) + b_k \sin(2\pi k f_0 t)] \tag{1.9}$$

where $f_0 = \frac{1}{T}$ is the fundamental frequency, and a_k, b_k are the Fourier coefficients, which determine the contribution of each sinusoidal component. An alternative representation using complex exponentials is :

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{i2\pi k f_0 t} \tag{1.10}$$

where the coefficients c_k encapsulate both amplitude and phase information.

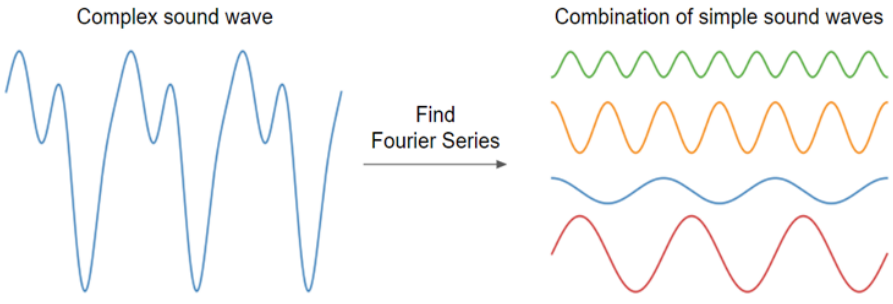


FIGURE 1.6 – Illustration of Fourier series decomposition. A complex sound wave (left) can be represented as a sum of simpler sinusoidal components (right), each corresponding to a different frequency.

In the context of audio signals, Fourier analysis provides a mathematical foundation for understanding how complex sounds are composed of simpler sinusoidal components. Musical tones,

speech signals, and other periodic sounds can be decomposed into a sum of harmonics, where each component represents a specific frequency. This decomposition is essential in various applications, including sound synthesis, filtering, and compression, making Fourier series a powerful tool in audio signal processing.

1.5 How Computers Measure Sound Quality

When we convert real-world sounds into digital audio (through sampling and quantization), some quality is always lost. This creates problems like background noise, distortion, or strange digital artifacts. You might notice music sounding flat, voices becoming harder to understand, or metallic echoes appearing.

Since getting hundreds of people to listen and rate audio quality would take too much time and money, sound engineers use computer measurements instead. These *objective metrics* act like artificial listeners, they analyze the audio mathematically to predict how humans would judge the quality. In this work we use the following metrics :

- **PESQ** (Perceptual Evaluation of Speech Quality) is the industry standard for telecommunications, using psychoacoustic models to evaluate speech clarity and naturalness in phone systems. It analyzes both temporal and spectral distortions to predict human perception.
- **ViSQOL**[1] (Virtual Speech Quality Objective Listener) employs a neurogram-based approach for modern audio applications. Its spectro-temporal analysis makes it particularly effective for evaluating compressed audio in streaming platforms and VoIP services.
- **CDPAM**[4] (Contrastive Decomposition-based Perceptual Audio Metric) leverages deep neural networks trained with contrastive learning to distinguish perceptually relevant distortions from irrelevant ones, aligning closely with human judgment.
- **STOI**[8] (Short-Time Objective Intelligibility) calculates the correlation between short-time spectral envelopes of reference and degraded speech, providing a 0-1 score that predicts speech comprehension in noisy environments like crowded spaces.

These tools help engineers improve audio quality automatically, fixing compression issues or removing noise, all without needing human testers.

Conclusion

This chapter introduced the mathematical foundations of audio signals. We began by modeling audio signals as sums of sinusoidal components, using Fourier analysis to decompose complex sounds into simpler frequencies. We then explored the differences between continuous-time and discrete-time representations, highlighting how sampling and quantization bridge the analog and digital domains. Finally, we examined objective metrics which evaluate audio quality by mimicking human perception. Together, these concepts provide the groundwork for analyzing and optimizing audio signals in neural networks.

CHAPTER 2

Introduction to Deep Learning

Introduction

Traditional *machine learning* techniques are capable of solving several problems. However, they often struggle with complex AI tasks like speech recognition, which are necessary to achieve the goals of this work. The specificity of this work requires algorithms that can find highly complex patterns in data.

Deep learning (DL) was mainly designed to overcome these limitations. It is well suited to finding complex patterns in data and handling the challenges of high-dimensional spaces. Modern *DL* emerges a powerful approach to supervised learning. By inreasing the number of layers and adding more units in each layer, deep networks become efficient in modeling complex functions.

In this chapter we introduce parametric function approximation, which forms the foundation of modern *DL*. We define feedforward deep networks, the basic model for representing messy non-linear functions.

2.1 Deep Feedforward Networks

Deep feedforward networks[2], also known as *feedforward neural networks* or *multilayer perceptrons* (MLPs), are the fundamental models in deep learning. Their primary goal is to approximate a function f^*. For example, in a classification task, this function maps the input annotated x to an output category y, expressed as :

$$y = f^*(x). \tag{2.1}$$

A feedforward network learns a similar mapping, defined as :

$$y = f(x; \theta), \tag{2.2}$$

where θ represents the parameters of the model. The training process optimizes θ to make $f(x;\theta)$ the closest possible to our target function f^*.

These models are called feedforward because information flows in the one and only direction from the input x, through intermediate computations that define f, until the output y. Unlike other advanced designs that are out of the scope of this work, *MLP*s have no feedback connections where the model's outputs are fed back into itself.

We call Feedforward neural networks (FNNs), "networks" because they are constructed by combining multiple building blocks (functions) into a single model. These functions form a directed acyclic graph (DAG), representing their connectivity.

For example, let's consider these three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ that are linked in a chain to form :

$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right). \tag{2.3}$$

This chain structure is very present in *NN*s. Each function corresponds to one layer : $f^{(1)}$ is the first layer, $f^{(2)}$ is the second layer, and so on. The total count of layers is called *the depth* of the network, this is where the term **deep** derives from. In Neurals nets, we call the last layer, that provides the final output, the *output layer*.

In neural network training our goal is to make $f(x)$ approximate $f^*(x)$. The training dataset provides noisy and approximate samples of $f^*(x)$, where each input x comes with a label $y \approx f^*(x)$. The data samples directly specify the behavior of the output layer : it has to generate values the closest possible to y for each x. However, the roles of the other layers are not explicitly defined by the *NN* designer. Instead, the learning algorithm determines how these layers should work together to produce the desired output. Since the training dataset doesn't reveal the output for these intermediate layers, we call them *hidden layers*.

Finally, *FNN*s are called "neural" because they are inspired by neuroscience. Each hidden layer in the network is usually a vector, and the size of this vector determines the *width* of the model. Each element of the vector can be thought of as similar to a neuron. Even the design of the functions $f^{(i)}(x)$, which compute these representations, is mostly based on observations of biological neurons.

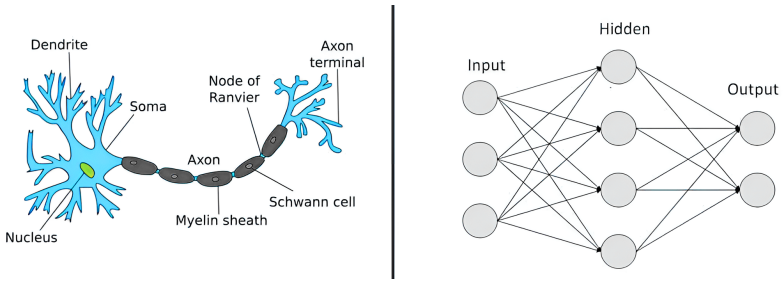


FIGURE 2.1 – Biological Neural Network vs Artificial Neural Network

2.1.1 Gradient-Based Learning

In the heart of deep learning there is gradient based optimization algorithms. These algorithms consist of adjusting the network's parameters iteratively to minimizes the *loss function*, which basically measures the difference between the prediction and the ground truth. Converging to a minimum, means by definition, reducing prediction errors.

Gradient Descent Algorithm

As shown in the figure 2.2 gradient descent algorithms minimize the given function by updating its parameters iteratively in the direction of the negative gradient. This technique is particularly effective for convex optimization problems where convergence to the global minimum is guaranteed, however, we still apply them on non-convex functions like in our case in neural networks training, where it helps us find locally optimal solutions.

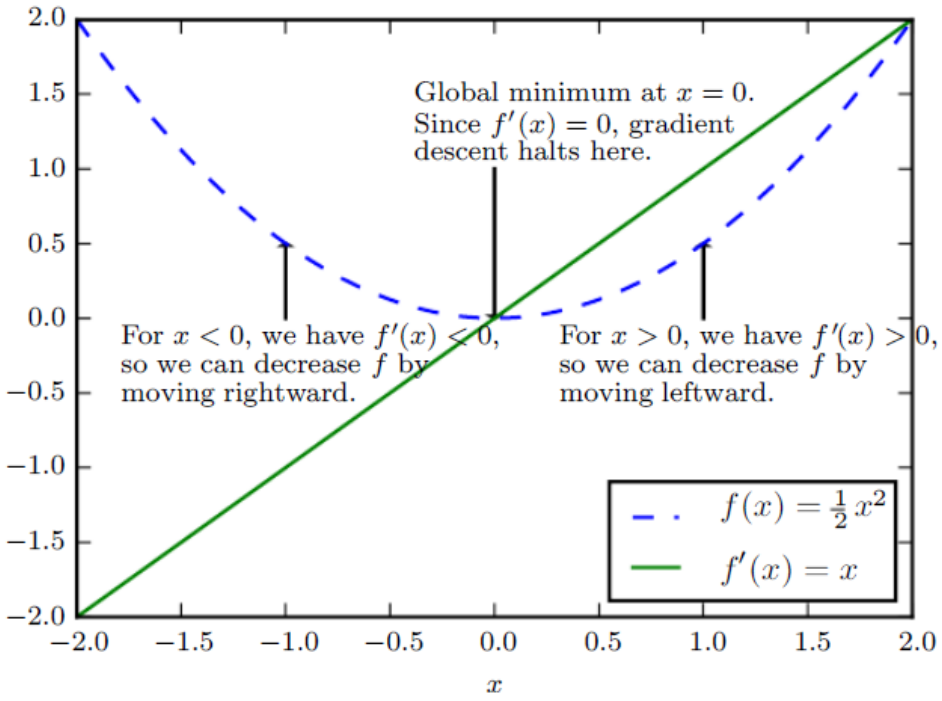


FIGURE 2.2 – An illustration of the way gradient descent algorithms use the derivatives of a function to follow it downhill to a minimum[2].

More formally, given a differentiable function $J(\theta)$ that depends on the model parameters θ, gradient descent updates the parameters iteratively as follows :

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J(\theta^{(t)}) \tag{2.4}$$

where η is the **learning rate**, a hyperparameter that controls the step size of the update.

The following algorithm outlines the general procedure for gradient descent :

Algorithm 1 Gradient Descent Algorithm

1: **Input :** Initial parameters θ randomly, learning rate η
2: **while** not converged **do**
3: Compute gradient : $g \leftarrow \nabla J(\theta)$
4: Update parameters : $\theta \leftarrow \theta - \eta g$
5: **if** stopping criterion is met **then**
6: **break**
7: **end if**
8: **end while**
9: **Output :** Optimized parameters θ

Learning Rate and Convergence

The initial choice of the learning rate η is important for the convergence of gradient descent :

- **Too large :** The algorithm may overshoot the minimum and fail to converge.
- **Too small :** The algorithm converges very slowly, requiring a large number of iterations.
- **Adaptive Methods :** Techniques such as Adam and RMSprop dynamically adjust the learning rate for improved convergence.

To monitor convergence, one can track the cost function $J(\theta)$ over iterations. The algorithm terminates when successive updates produce negligible changes in $J(\theta)$ or a predefined threshold is met.

2.1.2 Hidden Units

This far, our discussion was centered on general design principles for neural networks. We now shift our focus to a fundamental component that is unique to *FNN*s which is the choice of hidden units and their corresponding activation functions.

The design of hidden units affects how well a neural network can represent data. In fact, at the core of each hidden unit, there is an *activation function*, that controls how the unit would processes inputs and sends those information forward. Formally, an activation function applies a non-linear transformation to the input z for each element separately. This input z is obtained by applying a linear transformation to the original input $\mathbf{x}$:

$$z = \mathbf{W}^\top \mathbf{x} + \mathbf{b}, \tag{2.5}$$

where $\mathbf{W}$ represents the weight matrix and $\mathbf{b}$ the bias vector associated with the hidden unit. The activation function $g(z)$ then maps this linear combination to an output :

$$h = g(z). \tag{2.6}$$

One of biggest challenges to neural networks architects is the choice of the optimal activation function as it ensures the non-linearity of the network. An NN without activation functions

would equivalently behave as a single neuron, no matter how deep is it (see Appendix A.1).

For decades, researchers have designed various activation functions, each with its unique properties that make it particularly efficient in a family of tasks. One of the most commonly preferred activation functions is the rectified linear unit (ReLU), due to its simplicity. ReLU is defined as :

$$\text{ReLU}(z) = \max(0, z), \tag{2.7}$$

This function is not only easy to compute but also helps reduce the *vanishing gradient problem* (See Annex A.2).

Other than ReLU, there are various activations each has its own strengths and adapted for a different situation :

- **Sigmoid :** Defined as $\sigma(z) = \frac{1}{1+e^{-z}}$, it maps inputs to the range $(0, 1)$, making it useful for probability-based tasks. However, it can cause vanishing gradients when inputs are very large or very small.
- **Hyperbolic Tangent (**tanh**) :** Defined as $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, it is similar to the sigmoid but centered around zero, which can help speed up training.
- **Leaky ReLU :** A variation of ReLU that allows small negative values, helping to prevent neurons from getting "stuck" at zero output.
- **Swish :** Defined as $\text{Swish}(z) = z \cdot \sigma(z)$, this function has been found to improve performance in some deep networks.

Later in this work, we introduce a new activation function, and show its efficiency in representing audio signals with high fidelity.

2.1.3 Forward Propagation

As the name implies, the input data is propagated through the network in the *forward direction*. Each hidden layer receives the input activations, processes them using the activation function, and passes them to the next layer. In *FNN*s the input data flows in the forward direction only. The data signals should not propagate backward during inference, otherwise, it would create a *feedback infinte loop*, preventing the network from producing the output.

Mathematically, this transformation at a given layer $l \geq 1$ can be described as :

$$\mathbf{h}^{(l)} = g^{(l)} \left(\mathbf{W}^{(l)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \right), \tag{2.8}$$

where :

- $\mathbf{h}^{(l)}$ represents the activations of layer l,
- $g^{(l)}$ is the activation function,
- $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector for layer l, respectively,
- $\mathbf{h}^{(l-1)}$ is the activation from the previous layer.

This process continues layer by layer, culminating in an output prediction :

$$\mathbf{y} = g^{(L)} \left(\mathbf{W}^{(L)} \mathbf{h}^{(L-1)} + \mathbf{b}^{(L)} \right), \tag{2.9}$$

where L is the total number of layers. This final output is then compared to the ground truth using a cost function, the computed loss is then used to adapt the weights of the neural network reducing the distance between the predicted value and the ground truth.

2.1.4 Backpropagation and Differentiation Algorithms

As we have seen in the previous section, when an *FNN* processes an input $\mathbf{x}$ and produces an output $\hat{\mathbf{y}}$, information flows forward through the network. These transformations allows the input $\mathbf{x}$ to be propagated through a series of hidden layers until it reaches the final output. During training, forward propagation results in the calculation of a scalar cost function $J(\boldsymbol{\theta})$, which indicates how well the network is performing. However, for the network to actually learn, information must flow in the opposite direction, we define this process as *backpropagation*.

The *backpropagation algorithm* (often abbreviated as *backprop*) is the fundamental idea behind efficiently computing gradients for neural nets. Contrary to common misconceptions, backpropagation is not the entire learning algorithm but rather a method to efficiently compute the gradient of a function. Learning itself is carried out using optimization algorithms such as gradient descent variations.

At its core, backpropagation applies the chain rule of differentiation in a structured manner, efficiently computing the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ by propagating error signals backward through the network. This enables us to determine how each parameter contributes to the overall cost and adjust it accordingly to minimize J.

Let us consider a neural network with L layers. Each layer l consists of an activation vector $\mathbf{h}^{(l)}$, computed as in equation 2.8. To compute the gradient of the cost function with respect to the parameters, backpropagation proceeds in two steps :

1. **Forward Pass :** Compute the activations of each layer up to the final output $\hat{\mathbf{y}}$, then evaluate the cost function $J(\boldsymbol{\theta})$.
2. **Backward Pass :** Compute the gradients using the chain rule, propagating error signals backward from the output layer to the earlier layers.

We define the *error signal* at layer l as the partial derivative of the cost function with respect to the pre-activation values $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$:

$$\boldsymbol{\delta}^{(l)} = \frac{\partial J}{\partial \mathbf{z}^{(l)}} = \frac{\partial J}{\partial \mathbf{h}^{(l)}} \odot g'^{(l)}(\mathbf{z}^{(l)}), \tag{2.10}$$

where $g'^{(l)}(\cdot)$ denotes the derivative of the activation function, and $\odot$ represents element-wise multiplication (Hadamard product). The error at layer l is computed recursively using the relation :

$$\boldsymbol{\delta}^{(l)} = \left(\mathbf{W}^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)} \odot g'^{(l)}(\mathbf{z}^{(l)}). \tag{2.11}$$

This equation follows from applying the chain rule to differentiate the cost function with respect to earlier layers. The term $\left(\mathbf{W}^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)}$ propagates the error signal from layer $l+1$ to layer l.

Using these error signals, we now compute the gradients required for optimization :

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \big(\mathbf{h}^{(l-1)}\big)^T, \quad \frac{\partial J}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}. \tag{2.12}$$

The weight gradients are obtained via :

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \frac{\partial J}{\partial \mathbf{h}^{(l)}} \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \big(\mathbf{h}^{(l-1)}\big)^T. \tag{2.13}$$

Similarly, since the bias terms $\mathbf{b}^{(l)}$ are added element-wise, their gradients reduce to :

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}. \tag{2.14}$$

Although backpropagation is widely used, it is not the only differentiation algorithm available. In fact, backpropagation is a specific case of *automatic differentiation*, which also includes techniques like forward-mode differentiation and higher-order gradient computations. Backpropagation is highly efficient for deep neural networks due to its recursive reuse of intermediate values, reducing redundant computations. Moreover, it generalizes beyond neural networks and can be applied to any differentiable function.

By understanding backpropagation, we gain deeper insight into how neural networks learn, empowering us to design more effective architectures and optimization strategies. So the next time you see a neural network improving its accuracy over time, remember : *behind every great model is a great gradient.*

2.2 Architecture Design

Designing an effective feedforward network for audio requires several critical decisions. The fundamental choices include selecting an *optimizer*, *loss function*, and *output layer* configuration. Additionally, we must decide suitable *activation functions* for nonlinear processing and determine the network architecture through the *number of layers* and *units per layer* optimized for audio waveforms. Crucially, this involves *hyperparameter finetuning* : the process of selecting optimal values for key parameters like the number of hidden layers, hidden units per layer, and learning rate. These architectural decisions directly control the network's capacity to model both temporal evolution and spectral content in audio signals.

The key architectural considerations involve selecting :

- *Depth :* The number of layers in the network. Deeper networks can capture more complex hierarchical features and often require fewer parameters compared to shallow networks of equivalent representational power. However, they are also more challenging to optimize.
- *Width :* The number of units in each layer. Wider layers provide more capacity for learning intricate patterns but may lead to unnecessary redundancy and increased computational cost.

Selecting the optimal architecture for a specific task remains an empirical process, typically guided by monitoring performance on a validation set. Trial-and-error experimentation is crucial,

as there is no one-size-fits-all solution. While deeper networks promise greater expressiveness, they also introduce challenges such as vanishing gradients and longer training times. Finding the right balance between depth, width, and regularization is a central challenge in deep learning, making architecture design both an art and a science.

2.3 Illustrated Interpretation of Deep Learning Principles

The term *Deep Learning* refers to training neural networks, sometimes very large neural networks, to solve complex problems. But what exactly is a neural network ?

To build an intuition, let us consider a simple yet practical example : predicting student exam scores based on the number of hours studied. Imagine a dataset where we have records of different students, each with the number of hours they studied for an exam and the corresponding scores they achieved. Our goal is to find a function that best represents the relationship between study time and exam performance, allowing us to predict a student's score given their study time.

A straightforward approach to this problem is linear regression, where we assume a linear relationship between study hours and exam scores. If we plot the data points on a graph with study hours on the x-axis and exam scores on the y-axis, we can fit a straight line that best represents the trend in the data. The equation of this line takes the form :

$$y = wx + b \tag{2.15}$$

where x represents the number of study hours, y is the predicted exam score, w is the weight (or slope), and b is the bias (or intercept).

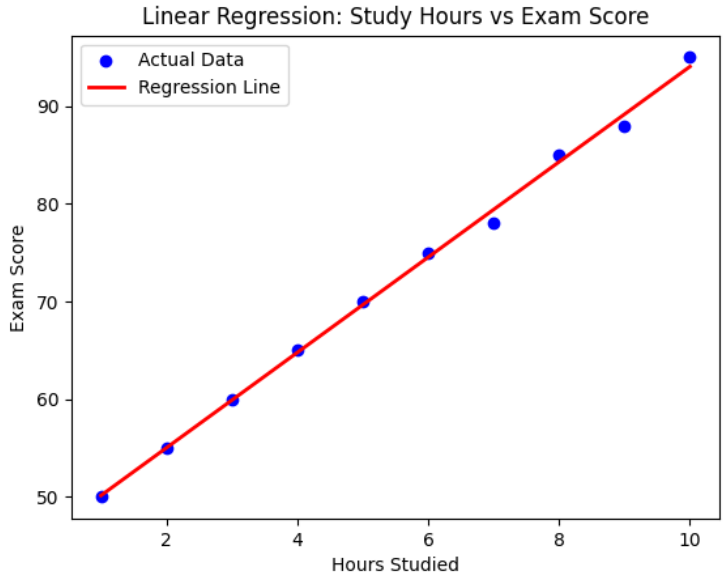


FIGURE 2.3 – Example of Linear Regression on the Plot of Score vs. Study Hours

In fact, this is the simplest form of a neural network just a single neuron with one input. It's the building block upon which more complex models are built. The linear regression we're looking at can be thought of as a neural network with just one neuron, showing how even the most basic models share the same underlying principles as more advanced deep learning networks.

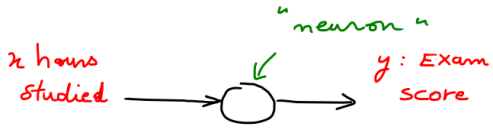


FIGURE 2.4 – Single Neuron Model Representing Linear Regression

The single neuron shown here models the concept of linear regression. It takes the number of study hours as input and outputs the predicted exam score. By adjusting its weight and bias, the neuron learns the relationship between the input and output, representing the simplest form of a neural network.

We have as the input to the neural network the number of hours studied, which we call x. It goes into this node, this little circle (Figure 2.4), and then it outputs the exam score, which we call y. So this little circle, which is a single neuron in a neural network, implements the function that models the relationship between study hours and exam scores. A larger neural network is then formed by taking many of these single neurons and stacking them together. If you think of this neuron as a single Lego brick, you can build a bigger neural network by stacking many of these Lego bricks together, each contributing to a more complex model.

Let's say that instead of predicting the exam score just from the number of hours studied, you now have other features. You know other things about the student, such as their family size. You might think that one of the things that really affects the exam score is how well the student slept before the exam, right ? And it's really based on the number of hours studied and the sleep hours that determines how well they perform. And then maybe you know the student's previous exam scores, which can also help predict the outcome.

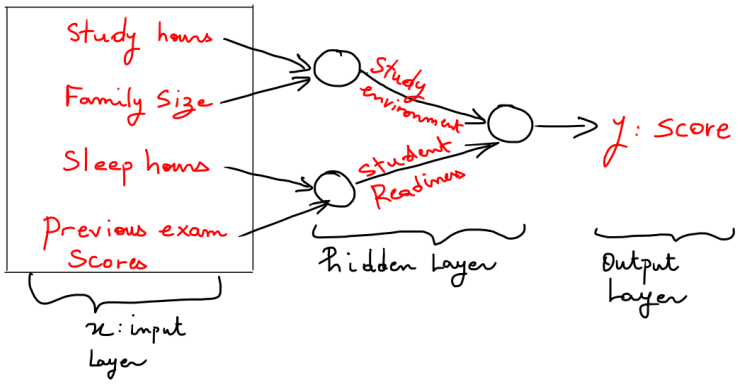


FIGURE 2.5 – Neural Network with Hidden Layer : Combining Study Hours, Family Size, Sleep, and Previous Scores to Predict Exam Results.

Each of these features is processed by the neural network, and through the hidden layer, the network combines them in ways that help predict the final exam score. Based on the number of study hours, family size, sleep hours, and previous performance, the network can generate intermediate features that enhance the prediction.

Finally, you might think that the way students perform on exams is influenced by all these factors : study hours, family size, sleep, and previous performance. By combining these features, we are better able to predict the final exam score.

And so by stacking together a few of the simple neurons, we now have a slightly larger neural network with one hidden layer. This hidden layer combines the features and creates intermediate features that help us predict the output. When you implement this neural network, you provide just the input x (study hours, family size, sleep hours, and previous exam scores) and the output y (exam score) for a number of examples in your training set, and all the hidden layers will figure out how to combine these features and adjust their weights to minimize the error.

2.3.1 Neural Network Representation

To truly grasp how neural networks function, it helps to start with a clear picture of their structure and the language used to describe them. Think of a neural network as a pipeline of layers, each one transforming the data it receives into something more refined and meaningful. In this section, we'll zoom in on a classic example : the **two-layer neural network**. This architecture includes an input layer, a single hidden layer, and an output layer. By breaking it down piece by piece, we'll see how data moves through the network and how each layer plays its part in shaping the final result. Let's start by looking at a visual representation of this setup.

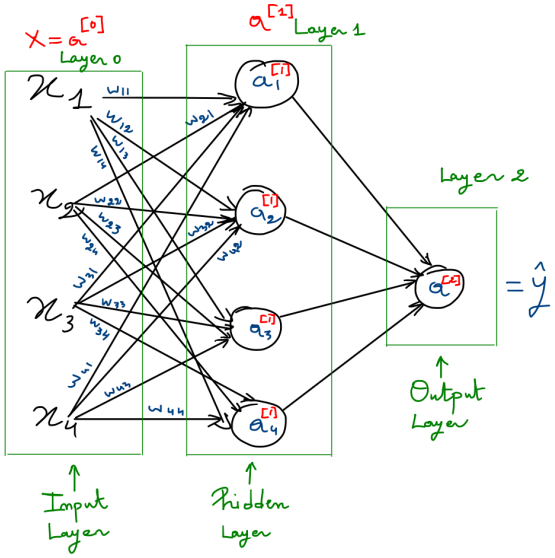


FIGURE 2.6 – A two-layer neural network with an input layer (x_1, x_2, x_3), a hidden layer $(a_1^{[1]}, a_2^{[1]}, a_3^{[1]}, a_4^{[1]})$, and an output layer $(a^{[2]})$.

At the very beginning, we have the **input layer** (layer 0), which consists of the input features x_1, x_2, x_3. These are the raw ingredients fed into the network. From there, the data flows into

the **hidden layer** (layer 1), which contains four nodes. Each of these nodes produces an activation value, labeled $a_1^{[1]}, a_2^{[1]}, a_3^{[1]}$, and $a_4^{[1]}$. The term **activation** (denoted by a) simply refers to the output of a layer, which is then passed on to the next layer. In this case, $a^{[1]}$ represents the activations of the hidden layer, calculated by taking a weighted sum of the inputs and applying a non-linear function. Finally, the **output layer** (layer 2) produces the predicted value $\hat{y}$, which is the same as $a^{[2]}$.

One might question why this architecture is referred to as a **two-layer neural network** when there are clearly three distinct sets of nodes. The explanation lies in a widely adopted convention within the field : the input layer is not formally counted as a standalone layer. Instead, only the hidden and output layers are enumerated. To systematically identify each layer, we employ the notation $a^{[l]}$, where $l = 0$ corresponds to the input layer, $l = 1$ to the hidden layer, and $l = 2$ to the output layer. This notation proves particularly advantageous as neural networks increase in depth and complexity.

At the core of every neuron lies a two-step computational process that transforms input data into an output signal. First, the neuron computes a weighted sum of its inputs, denoted as z. Mathematically, this is expressed as :

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b, \tag{2.16}$$

where x_1, x_2, x_3 represent the input features, w_1, w_2, w_3 are the corresponding weights assigned to each input, and b is the bias term associated with the neuron. The bias allows the neuron to adjust its output independently of the inputs, providing additional flexibility in modeling complex patterns. Once z is calculated, the neuron applies an **activation function** g to z, producing the final output of the neuron :

$$\text{output} = g(z). \tag{2.17}$$

This two-step process forms the foundation of how individual neurons process and transmit information within a neural network.

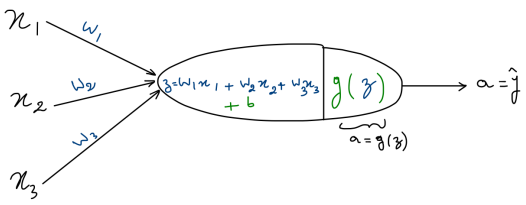


FIGURE 2.7 – Computation within a single neuron.

Now that we have explored the computation within a single neuron, let us extend this understanding to the first neural network introduced earlier. We will begin by calculating the weighted sums $z_1^{[1]}, z_2^{[1]}, \ldots, z_4^{[1]}$ for each of the four neurons in the hidden layer. For the first neuron in the hidden layer, the weighted sum $z_1^{[1]}$ is computed as :

$$z_1^{[1]} = x_1 w_{11}^{[1]} + x_2 w_{21}^{[1]} + x_3 w_{31}^{[1]} + b_1^{[1]}, \tag{2.18}$$

where x_1, x_2, x_3 are the input features, $w_{11}^{[1]}, w_{21}^{[1]}, w_{31}^{[1]}$ are the weights connecting the input layer to the first neuron in the hidden layer, and $b_1^{[1]}$ is the bias term associated with this neuron. Similarly, the weighted sums for the remaining neurons in the hidden layer are calculated as :

$$\begin{aligned} z_2^{[1]} &= x_1 w_{12}^{[1]} + x_2 w_{22}^{[1]} + x_3 w_{32}^{[1]} + b_2^{[1]}, \\ z_3^{[1]} &= x_1 w_{13}^{[1]} + x_2 w_{23}^{[1]} + x_3 w_{33}^{[1]} + b_3^{[1]}, \\ z_4^{[1]} &= x_1 w_{14}^{[1]} + x_2 w_{24}^{[1]} + x_3 w_{34}^{[1]} + b_4^{[1]}. \end{aligned} \tag{2.19}$$

To streamline these calculations, we can represent the input features as a vector $\mathbf{X} = [x_1, x_2, x_3]^T$ and rewrite the weighted sums in a vectorized form. For each neuron j in the hidden layer, the weighted sum $z_j^{[1]}$ can be expressed as :

$$z_j^{[1]} = \mathbf{w}_j^{[1]T}\mathbf{X} + b_j^{[1]}, \tag{2.20}$$

where $\mathbf{w}_j^{[1]} = [w_{1j}^{[1]}, w_{2j}^{[1]}, w_{3j}^{[1]}]^T$ is the weight vector for the j-th neuron and $b_j^{[1]}$ is its corresponding bias term.

Next, we apply the activation function g to each weighted sum to compute the activations of the hidden layer :

$$\begin{aligned} a_1^{[1]} &= g(z_1^{[1]}), \\ a_2^{[1]} &= g(z_2^{[1]}), \\ a_3^{[1]} &= g(z_3^{[1]}), \\ a_4^{[1]} &= g(z_4^{[1]}). \end{aligned} \tag{2.21}$$

To further simplify the notation, we can define the weight matrix $\mathbf{W}^{[1]}$ for the hidden layer as :

$$\mathbf{W}^{[1]} = \begin{bmatrix} \mathbf{w}_1^{[1]T} \\ \mathbf{w}_2^{[1]T} \\ \mathbf{w}_3^{[1]T} \\ \mathbf{w}_4^{[1]T} \end{bmatrix} = \begin{bmatrix} w_{11}^{[1]} & w_{21}^{[1]} & w_{31}^{[1]} \\ w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]} \\ w_{13}^{[1]} & w_{23}^{[1]} & w_{33}^{[1]} \\ w_{14}^{[1]} & w_{24}^{[1]} & w_{34}^{[1]} \end{bmatrix}, \tag{2.22}$$

and the bias vector $\mathbf{b}^{[1]}$ as :

$$\mathbf{b}^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}. \tag{2.23}$$

Using these definitions, the weighted sums for the entire hidden layer can be computed in a single vectorized operation :

$$\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]}. \tag{2.24}$$

Here, $\mathbf{Z}^{[1]} = [z_1^{[1]}, z_2^{[1]}, z_3^{[1]}, z_4^{[1]}]^T$ is the vector of weighted sums, and $\mathbf{X}$ is the input feature vector. Finally, the activations of the hidden layer are obtained by applying the activation function g element-wise to $\mathbf{Z}^{[1]}$:

$$\mathbf{A}^{[1]} = g(\mathbf{Z}^{[1]}), \tag{2.25}$$

where $\mathbf{A}^{[1]} = [a_1^{[1]}, a_2^{[1]}, a_3^{[1]}, a_4^{[1]}]^T$. This vectorized approach not only simplifies the notation but also enables efficient computation using modern numerical libraries.

2.3.2 Forward Propagation Revisited

Now that we have established the computations within a single layer, let us revisit the concept of **forward propagation** in the context of the entire neural network. Forward propagation refers to the process of passing input data through the network, layer by layer, to generate the final output. Each layer l in the network is defined by its weights $\mathbf{W}^{[l]}$, biases $\mathbf{b}^{[l]}$, and an activation function g. The computation for layer l can be expressed as :

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}, \tag{2.26}$$

where :

- $\mathbf{Z}^{[l]}$ is the matrix of weighted sums for layer l,
- $\mathbf{W}^{[l]}$ is the weight matrix for layer l,
- $\mathbf{A}^{[l-1]}$ is the matrix of activations from the previous layer (with $\mathbf{A}^{[0]} = \mathbf{X}$, the input features),
- $\mathbf{b}^{[l]}$ is the bias vector for layer l.

The activations for layer l are then computed by applying the activation function g element-wise to $\mathbf{Z}^{[l]}$:

$$\mathbf{A}^{[l]} = g(\mathbf{Z}^{[l]}). \tag{2.27}$$

Algorithm 2 Forward Propagation Algorithm

1: **Input :** Input features $\mathbf{X}$, weight matrices $\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \ldots, \mathbf{W}^{[L]}$, bias vectors $\mathbf{b}^{[1]}, \mathbf{b}^{[2]}, \ldots, \mathbf{b}^{[L]}$, activation function g
2: Initialize $\mathbf{A}^{[0]} \leftarrow \mathbf{X}$ ▷ Set input layer activations to the input features
3: **for** $l = 1$ to L **do** ▷ Loop through each layer
4: Compute weighted sums : $\mathbf{Z}^{[l]} \leftarrow \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$
5: Apply activation function : $\mathbf{A}^{[l]} \leftarrow g(\mathbf{Z}^{[l]})$
6: **end for**
7: **Output :** Final output $\mathbf{A}^{[L]}$ ▷ Predictions of the network

This layer-wise computation is at the heart of forward propagation. By organizing the calculations in terms of matrix operations, we not only simplify the notation but also reduce calculation time through parallel computing.

Matrix operations allow us to process multiple inputs simultaneously and reduce the time complexity of implementation, although a detailed analysis of complexity is beyond the scope of this chapter.

In essence, forward propagation transforms the input features $\mathbf{X}$ into predictions $\mathbf{A}^{[L]}$ through a series of linear transformations and non-linear activations, enabling the network to model complex relationships in the data.

2.4 Computation Graph and Backward Propagation

We begin by defining the function $J(a, b, c) = 2(a + bc)$. To compute the derivatives $\frac{dJ}{da}$, $\frac{dJ}{db}$, and $\frac{dJ}{dc}$, we will use the chain rule. First, let's break down the function into intermediate steps :

$$u = bc,$$
$$v = a + u,$$
$$J = 2v.$$

This decomposition allows us to construct a computation graph, which will help us visualize the flow of computations and apply the chain rule systematically.

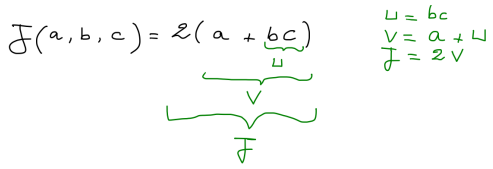


FIGURE 2.8 – Computation graph for $J(a, b, c) = 2(a + bc)$. The graph shows the intermediate variables $u = bc$, $v = a + u$, and $J = 2v$.

Using the chain rule, we can compute the derivatives of J with respect to a, b, and c as follows :

1. **Compute $\frac{dJ}{dv}$:**

$$\frac{dJ}{dv} = \frac{d}{dv}(2v) = 2.$$

2. **Compute $\frac{dv}{da}$ and $\frac{dv}{du}$:**

$$\frac{dv}{da} = \frac{d}{da}(a + u) = 1,$$

$$\frac{dv}{du} = \frac{d}{du}(a + u) = 1.$$

3. **Compute $\frac{du}{db}$ and $\frac{du}{dc}$:**

$$\frac{du}{db} = \frac{d}{db}(bc) = c,$$

$$\frac{du}{dc} = \frac{d}{dc}(bc) = b.$$

4. **Apply the chain rule to compute $\frac{dJ}{da}$, $\frac{dJ}{db}$, and $\frac{dJ}{dc}$:**

$$\frac{dJ}{da} = \frac{dJ}{dv} \cdot \frac{dv}{da} = 2 \cdot 1 = 2,$$

$$\frac{dJ}{db} = \frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{db} = 2 \cdot 1 \cdot c = 2c,$$

$$\frac{dJ}{dc} = \frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{dc} = 2 \cdot 1 \cdot b = 2b.$$

The derivatives of J with respect to a, b, and c are :

$$\frac{dJ}{da} = 2, \quad \frac{dJ}{db} = 2c, \quad \frac{dJ}{dc} = 2b. \tag{2.28}$$

These derivatives represent the sensitivity of the output J to changes in the inputs a, b, and c, respectively. This process of computing derivatives by moving backward through the computation graph is known as **backward propagation** or **backpropagation**.

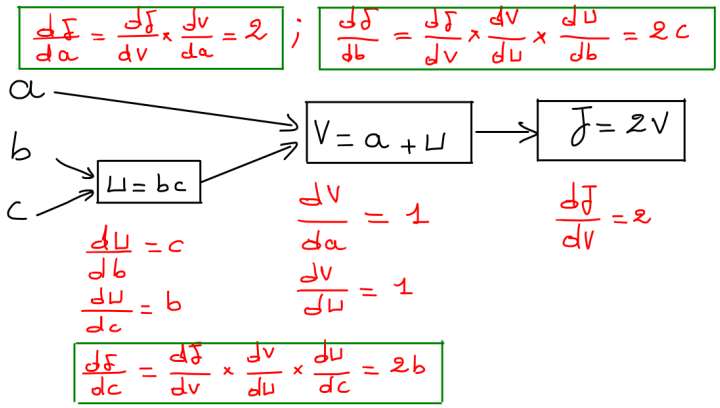


FIGURE 2.9 – Backward propagation through the computation graph. The arrows indicate the flow of gradients, starting from $\frac{dJ}{dv}$ and moving backward to compute $\frac{dJ}{da}$, $\frac{dJ}{db}$, and $\frac{dJ}{dc}$.

The analogy of the computation graph and chain rule provides a clear framework for understanding how backpropagation is implemented in a neural network. In a neural network, the goal is to minimize the loss function $J = L(\hat{y}, y)$, which measures the difference between the predicted output $\hat{y}$ and the true output y. Backpropagation works by computing the gradient of the loss function with respect to each parameter in the network, starting from the output layer and moving backward through the hidden layers. Specifically, we first compute the derivative of the loss with respect to the output of the final layer, $\frac{dL}{d\hat{y}}$. This gradient is then propagated backward through the network using the chain rule. For each layer l, we compute the derivative of the loss with respect to the pre-activation values $\mathbf{Z}^{[l]}$, denoted as $\frac{dL}{d\mathbf{Z}^{[l]}}$, by considering the activation function g. The relationship between the activations $\mathbf{A}^{[l]}$ and the pre-activations $\mathbf{Z}^{[l]}$ is given by :

$$\mathbf{A}^{[l]} = g(\mathbf{Z}^{[l]}). \tag{2.29}$$

Using the chain rule, we compute :

$$\frac{dL}{d\mathbf{Z}^{[l]}} = \frac{dL}{d\mathbf{A}^{[l]}} \cdot \frac{d\mathbf{A}^{[l]}}{d\mathbf{Z}^{[l]}} = \frac{dL}{d\mathbf{A}^{[l]}} \cdot g'(\mathbf{Z}^{[l]}), \tag{2.30}$$

where g' is the derivative of the activation function. This gradient is then used to compute the derivatives with respect to the weights $\mathbf{W}^{[l]}$ and biases $\mathbf{b}^{[l]}$. For example, in layer l, we compute :

$$\frac{dL}{d\mathbf{W}^{[l]}} = \frac{dL}{d\mathbf{Z}^{[l]}} \cdot \frac{d\mathbf{Z}^{[l]}}{d\mathbf{W}^{[l]}}, \quad \frac{dL}{d\mathbf{b}^{[l]}} = \frac{dL}{d\mathbf{Z}^{[l]}} \cdot \frac{d\mathbf{Z}^{[l]}}{d\mathbf{b}^{[l]}}. \tag{2.31}$$

This process is repeated for each layer, moving backward through the network, until we have computed the gradients for all parameters. These gradients, which represent the sensitivity of the loss function to changes in the weights and biases, are then used to update the parameters using gradient descent.

Differentiability Requirement

A crucial requirement for backpropagation is that the loss function must be *differentiable* - meaning we can compute its derivative at every point in its domain. Differentiability ensures we can calculate how small changes to each parameter affect the final loss. For a function to be differentiable, it must be smooth (no sharp corners or discontinuities) at every point where we want to compute gradients.

Algorithm 3 Gradient Descent with Backpropagation

Input : Input features $\mathbf{X}$, true labels $\mathbf{Y}$, weight matrices $\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \ldots, \mathbf{W}^{[L]}$, bias vectors $\mathbf{b}^{[1]}, \mathbf{b}^{[2]}, \ldots, \mathbf{b}^{[L]}$, learning rate η, and activation function g
while not converged **do**
    Forward Propagation :
    Initialize $\mathbf{A}^{[0]} \leftarrow \mathbf{X}$ ▷ Set input layer activations to the input features
    for $l = 1$ to L **do** ▷ Loop through each layer
        Compute weighted sums : $\mathbf{Z}^{[l]} \leftarrow \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$
        Apply activation function : $\mathbf{A}^{[l]} \leftarrow g(\mathbf{Z}^{[l]})$
    end for
    Compute the loss : $J \leftarrow L(\mathbf{A}^{[L]}, \mathbf{Y})$ ▷ Loss function
    Backpropagation :
    Compute $\frac{dJ}{d\mathbf{A}^{[L]}}$ ▷ Gradient of the loss w.r.t. the output
    for $l = L$ to 1 **do** ▷ Loop backward through each layer
        Compute $\frac{dJ}{d\mathbf{Z}^{[l]}} \leftarrow \frac{dJ}{d\mathbf{A}^{[l]}} \cdot g'(\mathbf{Z}^{[l]})$ ▷ Gradient w.r.t. pre-activations
        Compute $\frac{dJ}{d\mathbf{W}^{[l]}} \leftarrow \frac{dJ}{d\mathbf{Z}^{[l]}} \cdot \mathbf{A}^{[l-1]T}$ ▷ Gradient w.r.t. weights
        Compute $\frac{dJ}{d\mathbf{b}^{[l]}} \leftarrow \sum \frac{dJ}{d\mathbf{Z}^{[l]}}$ ▷ Gradient w.r.t. biases (sum over examples)
        Compute $\frac{dJ}{d\mathbf{A}^{[l-1]}} \leftarrow \mathbf{W}^{[l]T} \cdot \frac{dJ}{d\mathbf{Z}^{[l]}}$ ▷ Gradient w.r.t. previous layer activations
    end for
    Update Parameters :
    for $l = 1$ to L **do** ▷ Loop through each layer
        Update weights : $\mathbf{W}^{[l]} \leftarrow \mathbf{W}^{[l]} - \eta \cdot \frac{dJ}{d\mathbf{W}^{[l]}}$
        Update biases : $\mathbf{b}^{[l]} \leftarrow \mathbf{b}^{[l]} - \eta \cdot \frac{dJ}{d\mathbf{b}^{[l]}}$
    end for
end while
Output : Updated weights and biases

In the case of Mean Squared Error (MSE) loss, which is commonly used for regression tasks like audio signal prediction, the differentiability is guaranteed. The MSE loss for a single example is calculated as :

$$L = \frac{1}{2}(y - \hat{y})^2 \tag{2.32}$$

where y is the true value and $\hat{y}$ is the prediction. The gradient with respect to the prediction is simply :

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y \tag{2.33}$$

This straightforward derivative allows the error signal to flow backward efficiently through the network. The smooth quadratic form of MSE ensures stable gradient computation, which is why it's particularly well-suited for gradient-based optimization methods like backpropagation.

Forward propagation computes the network's output and the loss, while backpropagation efficiently computes the gradients of the loss with respect to the parameters using the chain rule. Gradient descent then updates the parameters to minimize the loss. However, the convergence of neural networks is not always guaranteed. Depending on factors such as initialization, learning rate, and architecture, a network may converge to a good solution, get stuck in a poor local minimum, or fail to converge altogether. These challenges and their solutions will be discussed in detail in the chapter on implicit neural representations.

Conclusion

This chapter has presented the fundamental building blocks of deep learning, with a focus on feedforward neural networks as powerful function approximators. We examined their architecture through the lens of computational graphs, demonstrating how information flows forward during prediction and gradients propagate backward during training. The mathematical framework developed here provides the necessary foundation for understanding more advanced architectures that we will detail in the next chapter. These concepts will prove essential as we explore specialized neural networks for audio signal processing in subsequent chapters.

CHAPTER 3

Implicit Neural Representations with Periodic Activations

Introduction

Implicitly defined, continuous, and differentiable signal representations parameterized by neural networks is gaining an increased interest in the current years because of its significant advantages over conventional representations for audio. Unlike traditional methods that discretize audio signals into fixed samples, such as waveforms or spectrograms, implicit neural representations (INRs) encode audio as a continuous function $f(t)$, where t represents time and $f(t)$ outputs the corresponding audio signal value. This approach allows for compact, high-fidelity representations of audio signals, enabling applications such as audio compression, synthesis, and super-resolution. A key innovation in this domain is the use of **periodic activation functions**, which are particularly well-suited for representing complex audio signals and their derivatives. Networks employing these activations, known as **Sinusoidal Representation Networks (SIRENs)**[6], excel at capturing the fine details and high-frequency components inherent in audio data. SIRENs can model audio signals with any predetermined precision exploiting the infinitely differentiable nature of sine waves, which makes sines an ideal choice for tasks requiring accurate reconstruction of audio.

SIRENs are uniquely suited for audio implicit neural representations due to their ability to represent high-frequency signals and their derivatives effectively. Audio signals are known to be rich in frequency content, thus traditional activation functions such as ReLU often struggle to capture these fine details. In contrast, the periodic nature of sine activations allows SIRENs to naturally model the oscillatory behavior of sound waves, making them a perfect fit for audio applications.

In this work we show that SIRENs are very useful in several audio tasks, including audio compression, where representing audio signals as continuous functions allows for highly efficient storage and transmission ; audio synthesis, enabling the generation of high-quality audio signals with fine control over frequency and dynamics ; and audio super-resolution, reconstructing

high-resolution audio from low-resolution inputs. Additionally, SIRENs' ability to model audio derivatives opens up possibilities for physics-based audio simulation, where not only the signal but also its higher-order dynamics can be accurately represented. The combination of SIRENs with hypernetworks further extends their potential, enabling the learning of priors over the space of audio functions. This could lead to generative models for audio, where SIRENs are used to sample realistic and diverse audio signals.

3.1 Intuition Behind Sinusoidal Activation Functions in SIRENs

To understand why sinusoidal activation functions are so effective in SIREN architectures, let's consider the behavior of two consecutive SIREN neurons. Suppose we have two neurons with sinusoidal activations, and for simplicity, we set the bias term $b = 0$. The output of the first neuron can be written as :

$$f_1(t) = \sin(w_1 t) \tag{3.1}$$

where w_1 is the weight of the first neuron. This neuron produces a simple sine wave with frequency w_1.
Now, let's pass this output as input to a second SIREN neuron. The output of the second neuron is :

$$f_2(t) = \sin(w_2 f_1(t)) = \sin(w_2 \sin(w_1 t)). \tag{3.2}$$

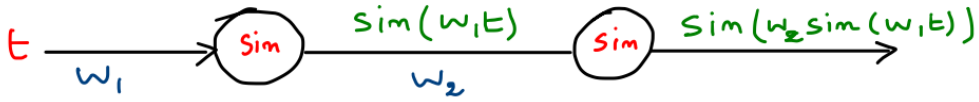


FIGURE 3.1 – Two consecutive SIREN neurons with sinusoidal activation functions.

Here, w_2 is the weight of the second neuron. This nested sine function, $\sin(w_2 \sin(w_1 t))$, is already capable of modeling more complex waveforms. For example, if we set $w_1 = 1$ and $w_2 = a$, the output becomes :

$$f_2(t) = \sin(a \sin(t)) \tag{3.3}$$

which is a well-known waveform in signal processing and mathematics. This function can be expanded into a series of sine waves with odd harmonics, as shown by the identity[5]

$$\sin(a \sin(t)) = 2 \sum_{n=0}^{\infty} J_{2n+1}(a) \sin((2n+1)t), \tag{3.4}$$

where $J_{2n+1}(a)$ are Bessel functions of the first kind. This means that just two SIREN neurons are already capable of modeling a rich set of frequencies, including higher-order harmonics.

The key takeaway is that sinusoidal activations allow SIRENs to naturally represent complex waveforms by composing sine functions. When we stack multiple SIREN neurons, this composition becomes even more powerful, enabling the network to model highly complex signals. This property makes SIRENs particularly well-suited for tasks like audio representation, where signals are often composed of multiple frequencies and harmonics.

In a large SIREN network, the ability to model such complex waveforms is further amplified. By carefully choosing the weights and stacking multiple layers, SIRENs can approximate virtually any continuous signal with high precision. This is why sinusoidal activation functions are so functional in SIREN architectures : they provide a natural and efficient way to represent complex, high-frequency signals, making them ideal for tasks like audio processing.

3.2 Mathematical Formalism of Implicit Neural Representations

Implicit neural representations (INRs) are a powerful framework for modeling continuous, differentiable functions that satisfy specific constraints. At their core, INRs are concerned with learning a function Φ that maps spatial or spatio-temporal coordinates $x \in \mathbb{R}^m$ to some quantity of interest, while satisfying a set of implicit equations of the form :

$$F\left(x, \Phi, \nabla_x \Phi, \nabla_x^2 \Phi, \dots\right) = 0, \quad \Phi : x \mapsto \Phi(x). \tag{3.5}$$

Here, F represents a constraint that may involve Φ and its derivatives, such as gradients ($\nabla_x \Phi$) or higher-order terms ($\nabla_x^2 \Phi$). The goal is to parameterize Φ using a neural network that not only approximates the desired mapping but also satisfies the constraints defined by F. This implicit formulation allows us to model a wide range of problems, from representing discrete audio signals as continuous functions as shown in the Figure 3.2.

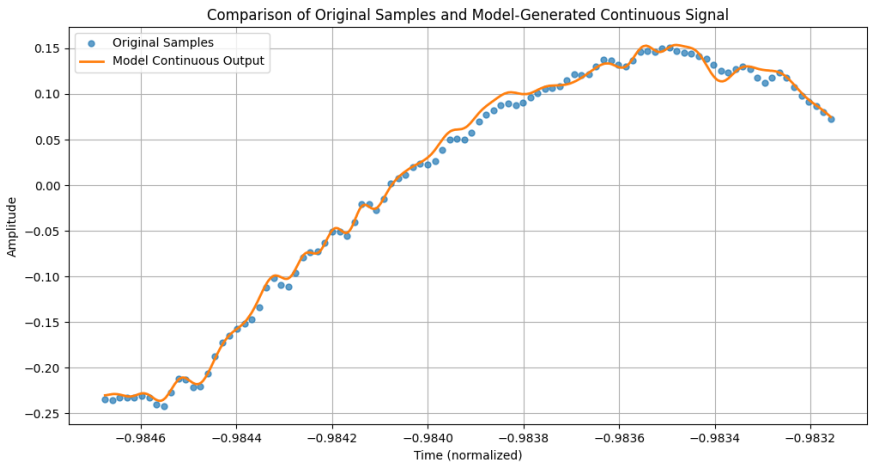


FIGURE 3.2 – Modeling discrete samples with a continous signal using SIREN.

In this experiment (figure 3.1), we compare the original discrete audio samples (blue dots) with the continuous output generated by our SIREN model (orange line). To ensure stable training and prevent activation explosion caused by the high-frequency nature of sine functions, we applied time normalization, which results in the negative time values seen on the x-axis. This normalization helps the model capture fine-grained signal details while maintaining numerical stability.

We show in the next chapter that this ability to represent signals in a continuous and differentiable way is a very powerful engineering tool. While discrete grid-based representations, are limited by resolution and memory, INRs define Φ over a continuous domain. This means they can precisely model super resolutions, relying on the continuity of the neural network.

3.2.1 The Feasibility Problem

To formalize the learning process, we cast the problem as a **feasibility problem**. Specifically, we seek a function Φ that satisfies a set of M constraints $\{C_m\}_{m=1}^{M}$, each of which relates Φ and/or its derivatives to some known quantities $a(x)$. Mathematically, this is expressed as :

$$\text{find } \Phi(x) \quad \text{subject to} \quad C_m\left(a(x), \Phi(x), \nabla\Phi(x), \dots\right) = 0, \quad \forall x \in \Omega_m, \quad m = 1, \dots, M.$$

Here, Ω_m represents the domain over which the m-th constraint is enforced. To solve this problem, we define a loss function that penalizes deviations from the constraints :

$$L = \int_{\Omega} \sum_{m=1}^{M} \mathbf{1}_{\Omega_m}(x) \left\| C_m(a(x), \Phi(x), \nabla\Phi(x), \dots) \right\| dx, \tag{3.6}$$

where $\mathbf{1}_{\Omega_m}(x)$ is an indicator function that is 1 when $x \in \Omega_m$ and 0 otherwise. In practice, we approximate this loss by sampling points x_i from the domain Ω, yielding a dataset $D = \{(x_i, a(x_i))\}_i$. The loss then becomes :

$$\tilde{L} = \sum_{i \in D} \sum_{m=1}^{M} \left\| C_m(a(x_i), \Phi(x_i), \nabla\Phi(x_i), \dots) \right\|. \tag{3.7}$$

This approximation allows us to train the neural network efficiently using gradient descent.

3.2.2 Parameterizing Φ with Neural Networks

To solve the feasibility problem, we parameterize Φ as a fully connected neural network with parameters θ. The network takes coordinates x as input and outputs the corresponding value $\Phi(x)$. During training, we optimize θ to minimize the loss $\tilde{L}$, effectively finding a function Φ_θ that satisfies the constraints as closely as possible.

This approach uses the universal approximation capabilities of neural networks while maintaining the flexibility to model complex, high-dimensional functions.

3.3 Representing Audio Signals with SIRENs

Audio waveforms are highly periodic, with structure at multiple time scales, making them an ideal candidate for SIRENs' continuous and differentiable representations. Unlike traditional neural networks, which struggle to accurately model raw waveforms, SIRENs can efficiently represent audio signals with minimal distortion, even when trained on relatively short clips of music or speech.

To demonstrate this, we fit a SIREN to raw audio waveforms using a loss function of the form :

$$L = \int_{\Omega} \|X_a(\Phi(x)) - a(x)\|^2 \, dx, \tag{3.8}$$

where X_a samples the SIREN at the waveform measurement locations, and $a(x)$ represents the ground truth audio signal. This formulation ensures that the SIREN learns to reproduce the original waveform as accurately as possible. Remarkably, SIRENs converge quickly to a high-quality representation, even for challenging audio data like music and speech.

3.4 Capturing a Tonal Audio Signal with SIRENs

To evaluate SIRENs' ability to represent high-frequency signals, we conducted an experiment using a sinusoidal waveform. The signal was defined as a pure sine wave with a frequency of 440 Hz, corresponding to the standard A4 musical note :

$$s(t) = C \sin(2\pi \cdot 440 \cdot t) \tag{3.9}$$

This signal provides a clear benchmark for assessing how well SIRENs can reconstruct continuous waveforms from limited samples.
The experimental setup was as follows :

Parameter	**Description**
Signal Definition	A pure sine wave with a frequency of 440 Hz (A4 note), represented by $s(t) = C \sin(2\pi \cdot 440 \cdot t)$.
Data Sampling	100 discrete samples were taken from the sine wave at a sampling rate of 44.1 kHz, simulating standard audio recording conditions. These samples served as training data.
Network Configuration	A SIREN model with a single neuron and sine activation function was used, as this structure is well-suited for oscillatory signals.
Training Objective	The model was trained to minimize the error between its predictions and the target sine wave values at the sampled points.

TABLE 3.1 – Experimental setup for modeling a 440 Hz sine wave (A4 note) with a SIREN.

After training, the model accurately estimated the signal's frequency at 439.56 Hz, closely matching the target 440 Hz. The reconstructed waveform was smooth and continuous, demonstrating that SIRENs can effectively capture high-frequency patterns even with sparse

input data. These results highlight the suitability of sinusoidal activations for learning periodic signals, supporting their use in audio processing tasks.

Key Insights

— **Efficiency** : A single neuron was enough to learn a sinusoidal signal.
— **Precision** : The reconstructed waveform closely matched the original A4 note.
— **Generalization** : Even with only 100 points, the SIREN inferred the full structure of the signal.

This experiment shows why sinusoidal networks are a game-changer for high-fidelity audio modeling. They don't just memorize data points, they truly *understand* the patterns in the signal.

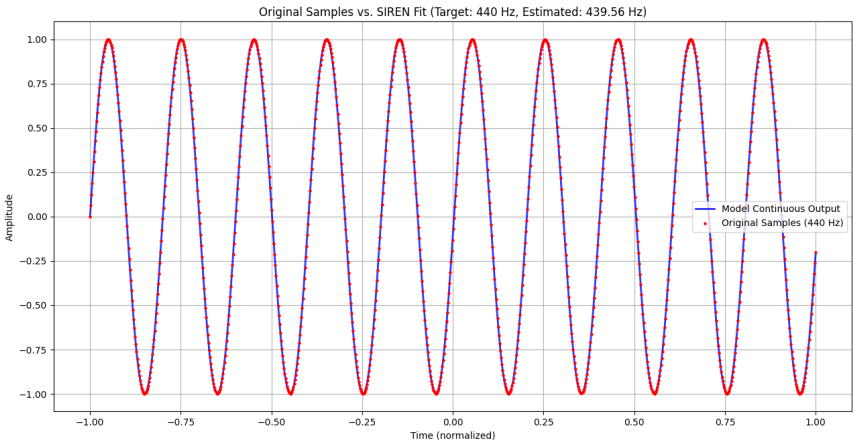


FIGURE 3.3 – Original samples of the A4 note (blue dots) compared to the SIREN's continuous output (orange curve). The SIREN, using only one neuron, accurately models the 440 Hz sine wave, showing how well it can represent high-frequency, periodic signals.

The figure 3.3 above shows the results of this experiment. The red dots are the original A4 samples, and the bleu curve is the SIREN's output. As you can see, the SIREN's curve fits the original signal almost perfectly, capturing both the amplitude and frequency of the sine wave. What's even more impressive is that the SIREN achieved this with just 100 samples and only one neuron.

The storage efficiency of this approach becomes clear when comparing formats. A standard PCM .wav file stores the 440 Hz sine wave as 100 discrete samples (400 bytes at 32-bit precision), while our SIREN implementation achieves equivalent quality through just two learned parameters (8 bytes), a 50 :1 compression ratio. Unlike simple frequency/phase storage, the SIREN's continuous representation $f(t) = \sin(w \cdot t + b)$ inherently compensates for sampling artifacts and phase shifts while enabling exact reconstruction at arbitrary resolutions.

This demonstrates SIRENs' unique advantage : they provide not just compression but a differentiable, continuous signal representation. While this single-neuron case illustrates the basic

principle, the architecture naturally extends to complex signals through additional layers and parallel frequency components. These properties make SIRENs particularly valuable for applications requiring both storage efficiency and mathematical manipulability of audio signals.

Tracking the Training Process

Now, let's take a closer look at how the SIREN learns to model the A4 note over time. To do this, we tracked the training process at three key stages : **0 steps**, **20 steps**, and **200 steps**. At each stage, we compared the ground truth (the original A4 waveform) with the SIREN's output to see how well it was doing. The goal was to observe the mismatch between the ground truth and the model decreased as training progressed.

At **0 steps**, the SIREN hadn't started learning yet, so its output was completely off. The phase shift was significant, and the model's curve looked nothing like the ground truth.
By **20 steps**, the SIREN had already started to figure things out. The phase shift began to shrink, and the model's output started to align with the ground truth. You could see the sine wave taking shape, though it wasn't perfect yet. The weights, biases, and average loss at this stage (which are clearly shown in the figure) reflected this improvement.
Finally, at **200 steps**, the SIREN had fully converged. The phase shift was almost gone, and the model's output matched the ground truth almost perfectly. The weights, biases, and average loss at this stage (clearly visible in the figures 3.4, 3.5 and 3.6) demonstrated how well the SIREN had learned to represent the A4 note.

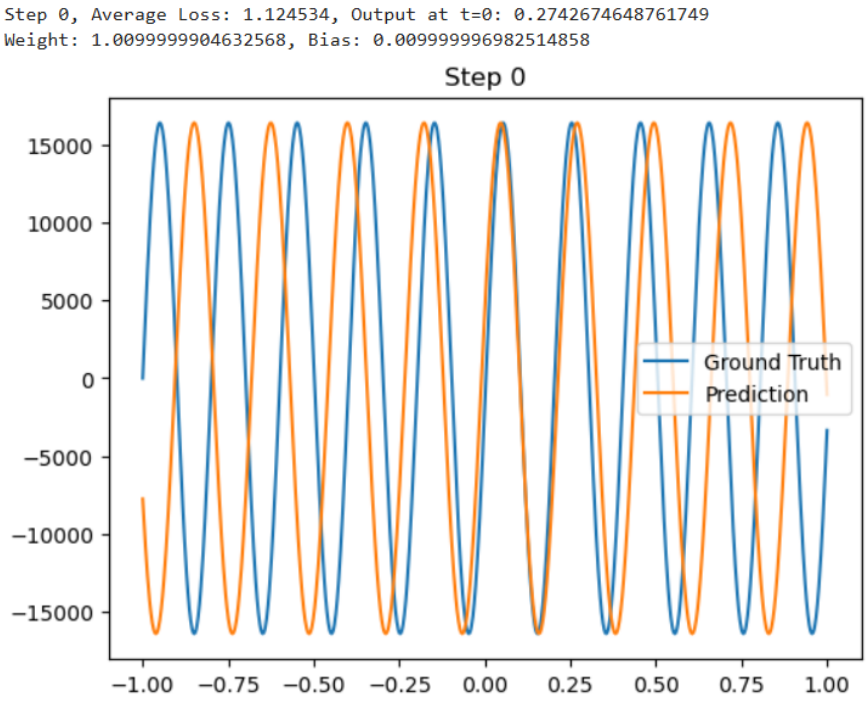


FIGURE 3.4 – Comparison between the ground truth (blue) and the SIREN's output (orange) at 0 training steps. The phase shift is significant, and the model's output is far from the ground truth. The weights, biases, and average loss at this stage are clearly shown.

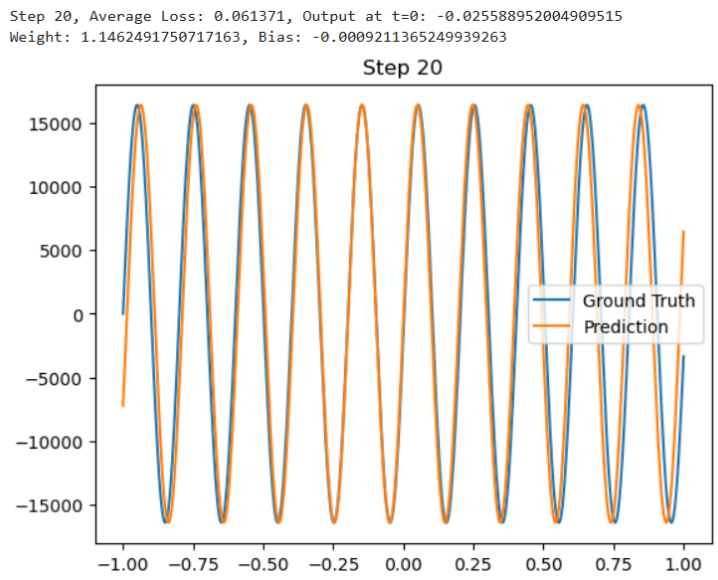


FIGURE 3.5 – Comparison between the ground truth (blue) and the SIREN's output (orange) at 20 training steps. The phase shift has decreased, and the model's output is starting to align with the ground truth. The weights, biases, and average loss at this stage are clearly shown.

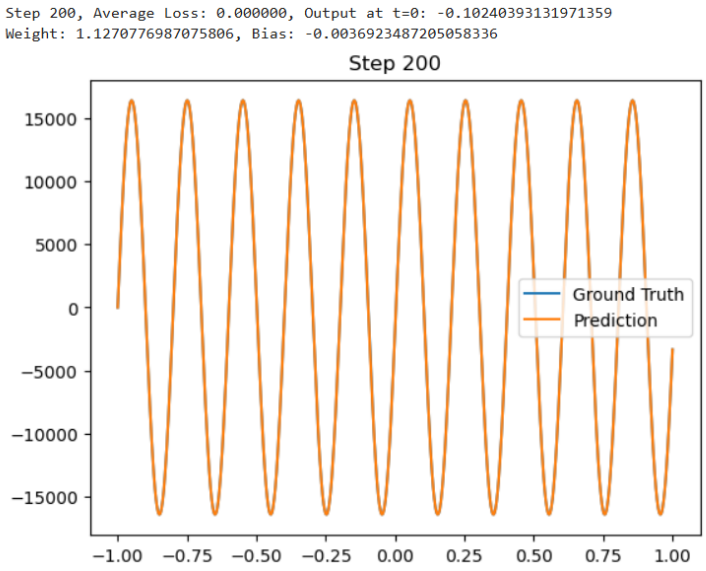


FIGURE 3.6 – Comparison between the ground truth (blue) and the SIREN's output (orange) at 200 training steps. The phase shift is almost gone, and the model's output matches the ground truth almost perfectly. The weights, biases, and average loss at this stage are clearly shown.

Conclusion

This chapter has demonstrated the remarkable capabilities of SIRENs for representing audio signals through continuous, differentiable functions. SIRENs solve the limitations of traditional discrete representations, enabling high-fidelity reconstruction of complex waveforms from sparse samples. Our experiments with tonal signals revealed that even a single SIREN neuron can accurately capture the frequency and phase characteristics of a sine wave.

CHAPTER 4

Audio Signal Processing with SIRENs

This chapter brings together everything we've explored so far about the power of SIRENs in audio signal processing. After understanding the theory behind audio signals, the mathematics involved, and the concepts of deep learning and implicit neural representations, it's time to turn theory into practice. In this section, we'll see how SIRENs can be applied to real-world challenges, from enhancing audio quality to compressing signals, and much more.

Through a series of carefully crafted experiments, we'll put SIRENs to the test. These experiments are not just technical exercises, they're meant to demonstrate how SIRENs can truly change the way we process and interact with audio data. Each experiment will add another piece to the puzzle, bringing the concepts we've covered to life. By the end of this chapter, you'll have a clear understanding of how techniques like Fourier series and gradient-based optimization come together to make SIRENs so effective.

To ensure the reliability of our findings, each experiment was conducted multiple times under the same conditions. This repetition helps confirm that the results are consistent and not due to random variations. However, for clarity and conciseness, we present only one representative trial in this report. This approach allows us to focus on the key insights while maintaining scientific rigor.

It's important to highlight that this chapter is **highly technical and experimental**. We'll be doing simulations, carefully analyzing the results, and testing the limits of what SIRENs can achieve. While the material can be challenging, our goal is to ensure that every step is explained in a way that's both clear and insightful. You'll not only understand the "how" behind the methods, but also the "why" that guides our approach.

4.1 The Architecture of SIREN

In this section, we will break down the key components of the SIREN architecture, explain its initialization scheme, and discuss how it achieves promosing performance in representing audio signals.

4.1.1 Core Components of SIREN

The SIREN architecture consists of several layers of sine-activated neurons, each designed to preserve the distribution of activations and gradients throughout the network. The main components are :

1. **SineLayer** : The fundamental building block of SIREN is the `SineLayer`, which applies a sine function to a linear transformation of its inputs. Mathematically, for an input x, the output of a `SineLayer` is given by :

$$y = \sin(\omega_0 \cdot (Wx + b)), \tag{4.1}$$

   where W is the weight matrix, b is the bias term, and ω_0 is a frequency scaling factor. The choice of ω_0 is critical : in the first layer, it determines the range of frequencies the network can model, while in subsequent layers, it ensures stable gradient flow.
2. **Network Structure** : A SIREN is typically composed of multiple `SineLayer` instances stacked together. The network takes spatial or temporal coordinates as input (e.g., time points for audio signals) and outputs the corresponding signal values. The final layer can either be another `SineLayer` or a linear layer, depending on the application.
3. **Input and Output Handling** : The input to the network is a set of coordinates (e.g., time points for audio), and the output is the corresponding signal value. During training, the network learns to map these coordinates to the desired signal by minimizing a loss function, such as the mean squared error between the predicted and actual signal values.

4.1.2 Initialization Scheme

A key challenge in training SIRENs is ensuring that the network can model high-frequency signals without destabilizing the training process. To address this, we use a principled initialization scheme that preserves the distribution of activations and gradients across layers. The initialization scheme is based on the following insights :

1. **Distribution of Activations** : For a single sine neuron with uniformly distributed inputs $x \sim U(-1, 1)$, the output $y = \sin(ax + b)$ follows an arcsine distribution when $a > \frac{\pi}{2}$. This property ensures that the activations remain well-behaved throughout the network.
2. **Weight Initialization** : To maintain stable gradients, the weights of each `SineLayer` are initialized uniformly within a specific range. For layers beyond the first, the weights

are scaled by $\frac{1}{\omega_0}$ to prevent the activations from growing too large. This ensures that the network can model high-frequency signals without causing **gradient explosions**.

Gradient explosion occurs when the gradients of the loss function with respect to the model's parameters grow exponentially during backpropagation, leading to unstable training. In the context of SIRENs, this can happen if the inputs to the sine function become too large, causing the gradients (which depend on the cosine of these inputs) to grow uncontrollably. For example, if the input to a sine neuron is $z = \omega_0 \cdot (Wx + b)$, its gradient with respect to the weights W is proportional to $\cos(z)$. If z is too large, $\cos(z)$ can oscillate wildly, leading to large and unstable gradients. By scaling the weights with $\frac{1}{\omega_0}$, we ensure that z remains within a reasonable range, preventing the gradients from exploding and allowing the network to train stably even when modeling high-frequency signals.

3. **Frequency Scaling** : The first layer is initialized with a larger frequency scaling factor ω_0 (typically set to 30) to allow the network to model a wide range of frequencies. This is particularly important for audio signals, which often contain high-frequency components.

The combination of sine activations and the principled initialization scheme makes SIRENs uniquely suited for modeling audio signals. Unlike traditional neural networks, which struggle with high-frequency data, SIRENs can accurately represent complex waveforms while maintaining stable training dynamics.

4.2 Hyperparameter Tuning Methodology

In contrast to traditional neural networks where generalization is essential, training implicit neural representations requires careful optimization of hyperparameters to achieve perfect signal fitting. For SIREN networks modeling individual signals, our objective shifts to maximizing representational capacity while maintaining training efficiency. This section details our experimental approach and presents initial findings from systematic architecture exploration.

In neural network design, we distinguish between two fundamental classes of variables :

- **Parameters** : The internal weights learned during training through backpropagation, which constitute the model's knowledge base (e.g., connection weights between neurons).
- **Hyperparameters** : The architectural and optimization settings predetermined before training commences, including :
  - Structural choices (hidden layer dimensions, network depth)
  - Learning process configurations (learning rate)

For SIREN networks specifically, hyperparameter selection carries additional significance due to the unique properties of sinusoidal activation functions. The periodic nature of these activations introduces complex gradient dynamics that make architectural choices particularly sensitive.

4.2.1 Testing Different Sizes and Depths

Our experimental protocol systematically varied two key architectural hyperparameters :

— **Hidden size** : The dimensionality of each layer's learned representation, tested across a logarithmic scale from 2 to 1024 units.

— **Network depth** : The number of hidden layers, evaluated from 1 to 6 layers.

All configurations were evaluated on a consistent *4-second* audio reconstruction task, with Table 4.2 presenting key results from this exploration. In these settings, the loss metric represents the *mean squared error* between the network's output and the target waveform. The specifications of the audio file used for these experiments are summarized in Table 4.1.

TABLE 4.1 – Specifications of the 4-second audio file used for hyperparameter fine-tuning.

Property	**Value**
Duration	4.00 s
Size	344 KB (352,592 bytes)
Bitrate	705 kb/s
Format	PCM (Floating-Point 32-bit)
Sample Rate	22050 Hz
Channels	Mono

TABLE 4.2 – Representative Hyperparameter Configurations and Performance

Hidden Size	**Layers**	**Parameters**	**Loss (MSE)**
2	6	43	0.0254
16	3	865	0.0161
32	6	6,433	0.0015
128	4	66,433	3.15×10^{-5}
256	6	395,521	3.74×10^{-7}
512	5	1,314,817	6.77×10^{-9}
1024	3	3,151,873	3.07×10^{-9}

Analysis of Hyperparameter Results

Our experiments reveal clear patterns in how network size affects reconstruction quality. As shown in Figure 4.1, increasing the hidden units produces dramatically better results, but with rapidly diminishing returns :

— **Small networks** (up to 16 units) struggle to learn the signal properly, achieving only moderate accuracy (MSE $\sim$0.02)

— **Medium networks** (32-128 units) show the most dramatic improvements, reducing errors by several orders of magnitude

— **Large networks** (512+ units) achieve near-perfect reconstruction (MSE $< 10^{-8}$), but require significantly more computation

The complete results (all 60 configurations available in our repository) suggest that most practical applications will benefit from networks in the 32-256 unit range. These provide excellent

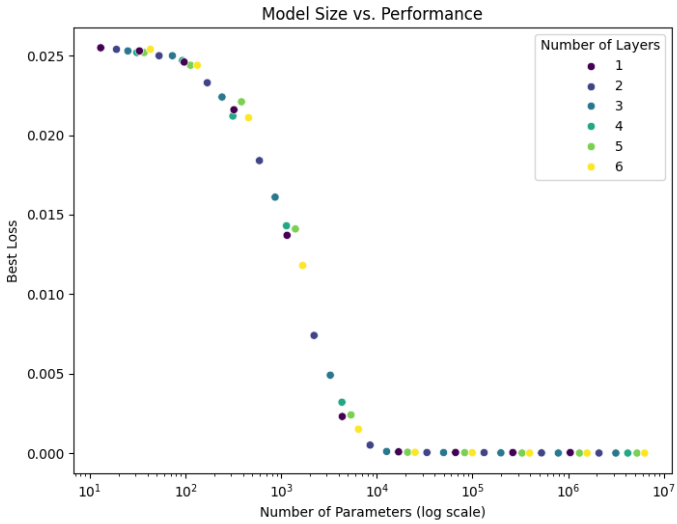


FIGURE 4.1 – Reconstruction error versus model size, showing three distinct regimes of performance

accuracy without the excessive computational costs of the largest models.

Interestingly, we observe that :

— Adding layers helps more for wider networks than narrow ones
— Training time scales predictably with parameter count
— Very small networks (2-8 units) fail to learn regardless of depth

These findings have important implications for practical applications of SIREN networks, which we explore in Section 4.4.2.

Model Size vs. Performance Trade-off

In this subsection we analyze the relationship between model complexity and reconstruction accuracy. Figure 4.2 presents these findings through a comprehensive scatter plot :

— **X-axis** : Total number of parameters (shown in logarithmic scale) - representing the model's overall size
— **Y-axis** : Best achieved loss value - indicating reconstruction performance
— **Color** : Number of layers in the network (from 1 to 6)
— **Point size** : Hidden layer dimensionality (ranging from 2 to 1024 units)

The logarithmic scale on the x-axis ensures we can clearly see patterns across all model sizes, from our smallest (13 parameters) to largest (6.3 million parameters) configurations. The color gradient and point sizing help distinguish how different combinations of depth and width contribute to the overall results.

As clearly shown in Figure 4.2 as the number of parameters increases, the best loss decreases exponentially indicating improved performance. Initially, for small models (e.g., *13 parameters, 1 layer*), the best loss remains relatively high at approximately 0.0255. This suggests that these models lack the capacity to capture meaningful patterns in the audio signal. However, as we

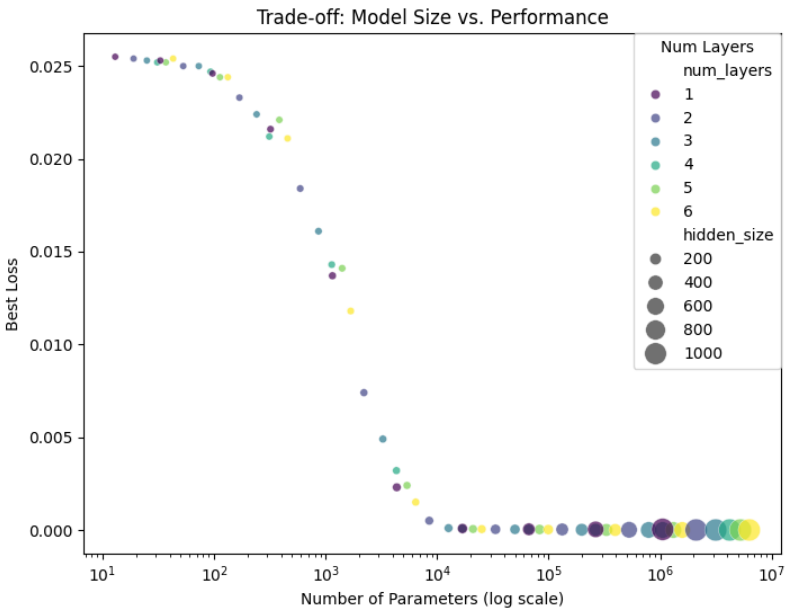


FIGURE 4.2 – Relationship between model complexity and reconstruction performance. Point size represents hidden layer dimension, color indicates number of layers. Note logarithmic scale on x-axis.

increase the number of parameters, the loss starts decreasing noticeably. For instance, a model with *8,513 parameters and 2 layers* achieves a best loss of approximately 0.0005, a dramatic improvement.

Hyperparameter Analysis Using 3D Visualization

In this subsection, we analyse our dataset using the 3D scatter plot. Figure 4.3 shows the the relationship between hidden size, number of layers, and the achieved loss. The color scale represents the number of parameters, ranging from blue (fewer parameters) to yellow (a large number of parameters).
The 3D plot reveals a sharp decline in loss as both hidden size and the number of layers increase. Smaller, shallower networks struggle with higher loss, whereas deeper and wider networks achieve significantly better accuracy.

Unlike traditional models, overfitting is not a drawback but rather a necessary feature, as the network is designed to memorize input data efficiently. However, while increasing the number of parameters improves approximation, it also raises computational costs. Interestingly, some smaller networks perform nearly as well as much deeper ones, indicating that an optimal balance exists.

The blue markers near the bottom of the plot highlight the most interesting neural networks. These models have fewer parameters but still achieve low loss, meaning they perform well while staying efficient. This is important for Implicit Neural Representations (INR) because it shows that we don't always need large networks to get good results.

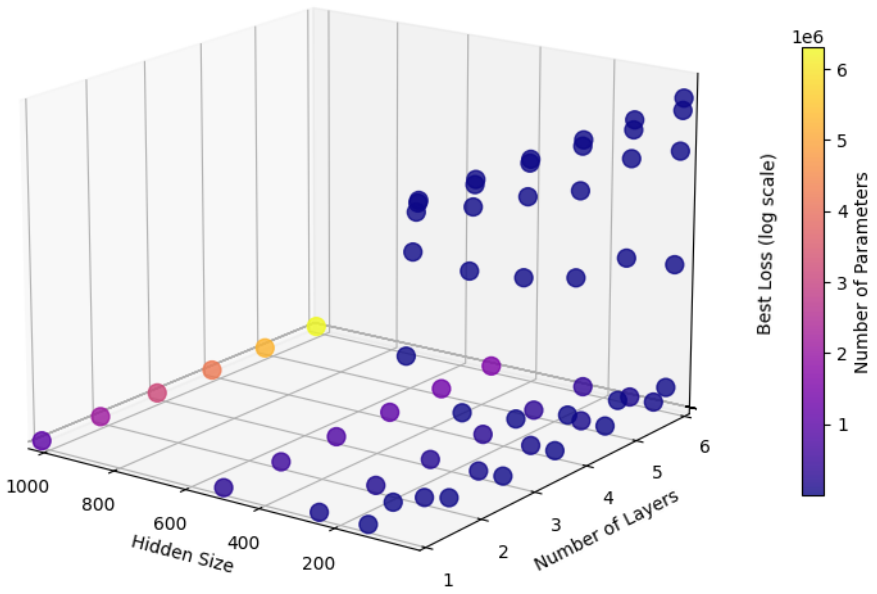


FIGURE 4.3 – 3D visualization of hyperparameter influence on model performance.

While this 3D plot helped visualize how the loss changed with different model configurations, it wasn't the best way to spot detailed patterns. The perspective made it tricky to see smaller variations clearly. To get a better understanding of the data, we used a heatmap instead.

Hyperparameter Analysis Using Heatmap

This heatmap Figure 4.4 shown below makes it much easier to see how loss decreases as the number of parameters increases. Looking at the data, we can obviously see the general trend : larger models with more hidden units and layers tend to have lower loss. However, this improvement isn't always straightforward. For example, increasing the hidden size from 2 to 64 leads to a big drop in loss, but after a certain point, adding more parameters doesn't help as much.

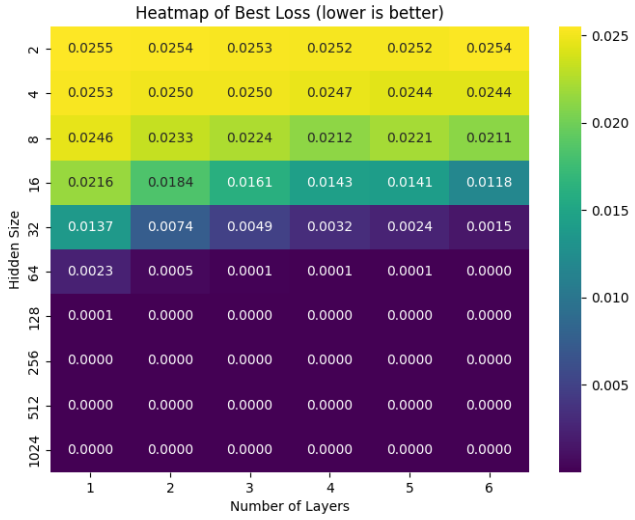


FIGURE 4.4 – Heatmap showing loss as a function of model parameters.

One key takeaway is that moving from small models (like hidden size 2, num layers 1) to medium-sized ones (like hidden size 32, num layers 4) brings significant improvements. For example, a model with hidden size 16 and 6 layers (1,681 parameters) has a loss of 0.0118, while increasing to hidden size 32 and 6 layers (6,433 parameters) drops the loss to 0.0015. However, jumping to hidden size 512 and 6 layers gives only a tiny improvement, even though the number of parameters increases drastically.

We want a model that balances low loss with a reasonable number of parameters. While it's tempting to go for the absolute lowest loss, models with millions of parameters are expensive to train. Based on the heatmap, models with a hidden size of 32 to 64 and around 3 to 6 layers seem to offer the best trade-off. They achieve very low loss (around 0.0001 to 0.0023) without becoming too large.

After testing many different network designs (Table 4.2), we found the 256-unit, 4-layer version works best because it gives nearly the same good results (loss : 9.33×10^{-6}) as much larger networks while using far fewer computer resources. This balanced choice performs well consistently, uses a practical 263,937 parameters (just 4.2% of the 6.3 million parameters in our largest 1024-unit, 6-layer network), and trains reliably every time. We'll use this network size for all following tests to properly study how audio length affects performance.

4.2.2 Testing Different Audio Lengths

Having established our potentially optimal network architecture (256 units, 4 layers), we now investigate how different audio file sizes affect the model's performance. Using the same network configuration across all tests, we systematically evaluate reconstruction quality for WAV files ranging from 1.23MB to 0.06MB in size. This experiment will reveal how the network's accuracy scales with varying input sizes while maintaining consistent parameters (263,937) and sample rate (44.1kHz). The results demonstrate the relationship between audio file size and reconstruction error.

Table 4.3 – Fixed Experimental Parameters for File Size Analysis

Parameter	**Value**
Hidden Layer Size	256 units
Number of Layers	4
Total Parameters	263,937
Sample Rate	44.1 kHz
Audio Format	32-bit PCM WAV
Audio Duration Range	0.66s to 14.62s

Our experiments reveal a clear power-law relationship between audio file size and reconstruction accuracy. The model achieves dramatically better results with smaller files, with the 0.06MB file attaining a near-perfect loss value of 1.29×10^{-8} - a 7700-fold improvement over the 1.23MB file's 9.95×10^{-5} error.

The data reveals three distinct quality regimes. For files above 0.74MB, the loss exceeds 1×10^{-5}, producing perceptible artifacts. Mid-range files (0.28-0.74MB) achieve good quality with losses between 1×10^{-6} and 1×10^{-5}. Most impressively, files below 0.28MB reach excellent transparency with losses under 5×10^{-7}. This behavior suggests the 256×4 architecture has an optimal operating range for files under 0.5MB (approximately 5 seconds at 44.1kHz).

TABLE 4.4 – Reconstruction Error vs. Audio File Size

Duration (s)	**Size (MB)**	**Loss**	**Quality**
14.62	1.23	9.95×10^{-5}	Acceptable
9.95	0.84	4.30×10^{-5}	Good
4.42	0.37	1.26×10^{-6}	Very Good
1.99	0.17	2.83×10^{-7}	Excellent
0.66	0.12	1.15×10^{-7}	Near-Perfect

Visualized in Figure 4.5, this power-law relationship demonstrates the network's remarkable capability with short audio segments. However, the sharp error reduction below 0.5MB indicates a memory-bandwidth constraint for longer sequences. While the current architecture delivers excellent results for brief audio chunks, extended durations would require either chunked processing or increased model capacity to maintain equivalent quality. The clear correlation between file size and reconstruction error provides valuable guidance for practical applications of this architecture.

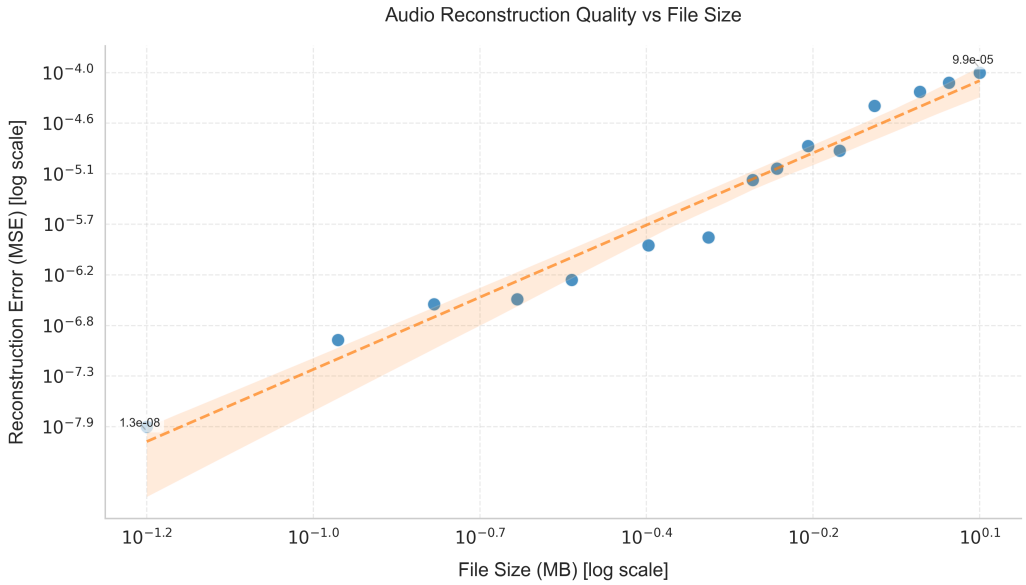


FIGURE 4.5 – Log-log relationship between file size and reconstruction error. The shaded region (0.1-0.5MB) indicates the optimal operating range for the 256×4 architecture.

4.2.3 What We Learned About SIREN Performance

Our experiments reveal two fundamental insights about SIREN performance. First, the network size needed depends on the sound complexity : simple audio signals can be accurately

represented by compact networks (32-128 units), while complex waveforms require larger architectures (256-1024 units) to capture all details. Second, we observe that increasing network size consistently improves reconstruction accuracy, though with diminishing returns as models become very large. Each expansion in neurons or layers brings smaller improvements once the network reaches a certain size.

These findings provide practical guidance for SIREN implementation. For most applications, we recommend starting with moderate-sized networks (32-128 units) and scaling up only when necessary for complex signals or higher precision requirements. The optimal architecture balances three factors : the desired reconstruction accuracy, the inherent complexity of the target sound, and available computational resources. As shown in our earlier figures, this approach maximizes the practical benefits of SIRENs while avoiding unnecessary computational overhead.

4.3 Perceptual Evaluation of Reconstructed Audio

While objective metrics measure technical accuracy, they don't always reflect human perception of sound quality. To address this limitation, we employ four complementary perceptual metrics : PESQ for speech quality assessment, ViSQOL as a virtual listening tool, CDPAM for deep learning-based evaluation, and STOI for speech clarity measurement. This multi-metric approach ensures comprehensive evaluation, as each tool assesses different aspects of audio quality.

Our analysis serves three key purposes : First, we verify whether numerical measurements correlate with subjective listening quality. Second, we determine the required reconstruction precision for perceptually indistinguishable results. Third, we establish practical guidelines for selecting network sizes that optimally balance computational efficiency with perceptual quality.

Among these metrics, we prioritize CDPAM for our primary interpretation, as it has demonstrated superior alignment with human auditory perception in recent studies. The remaining metrics provide valuable supplementary perspectives, creating a robust evaluation framework.

The following analysis correlates each network configuration's performance (from Section 4.2.1) with these perceptual metrics, offering comprehensive insights into SIRENs' real-world audio reconstruction capabilities. For our compression objectives, we specifically examine how perceptual quality varies across configurations, seeking the minimal network architecture that maintains perceptually good results. This is achieved by analyzing relationships between quality metrics and the mean squared error between original and reconstructed signals.

4.3.1 PESQ vs. Loss Analysis

Figure 4.6 demonstrates an inverse relationship between the loss value and PESQ score, as the loss decreases, the perceptual audio quality improves significantly. Our experimental data reveals this critical pattern :

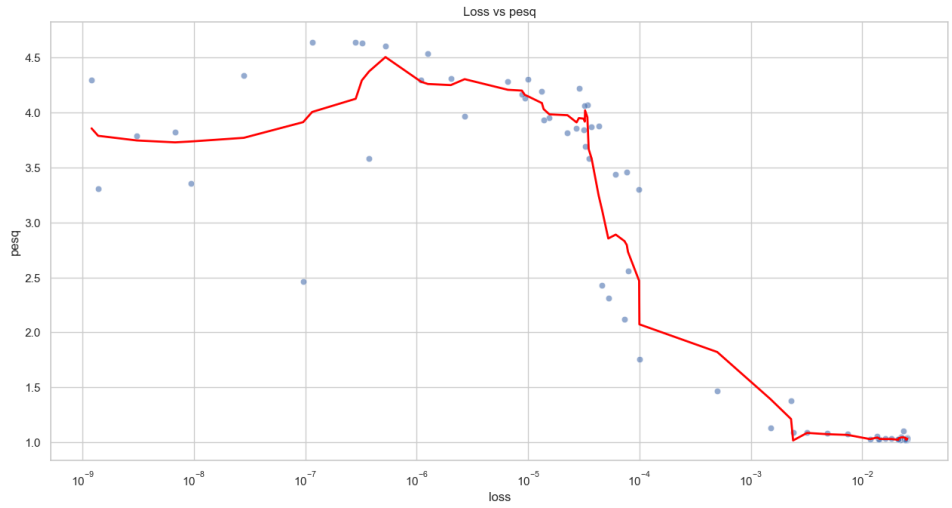


FIGURE 4.6 – Relationship between loss values (x-axis) and PESQ scores (y-axis). The red line marks the PESQ values.

— At high loss values (∼0.025), PESQ scores drop to approximately 1.0-1.04, indicating unintelligible speech quality
— When loss decreases to 7.35×10^{-5}, PESQ reaches 2.12 - crossing the minimum threshold for intelligible speech
— Further reduction to 3.22×10^{-7} yields excellent quality (PESQ = 4.63)

Human listening tests confirm that PESQ scores ≥ 2.0 correspond to acceptable speech intelligibility, while scores ≥ 3.5 represent near-transparent quality. The sharp improvement curve shows that small reductions in loss below 1×10^{-4} lead to disproportionately large gains in perceptual quality.

Three key observations emerge from this analysis :

1. The PESQ=2.0 threshold occurs at loss $\approx 7.35 \times 10^{-5}$ (sample 37 in our dataset)
2. Quality improves linearly with log-scaled loss reduction
3. Beyond PESQ=4.0, additional loss reduction provides diminishing returns

This relationship guides our network optimization, we prioritize configurations achieving losses below 1×10^{-5}, as they consistently deliver PESQ scores above the 2.0 intelligibility threshold while balancing computational costs.

4.3.2 STOI vs. Loss Analysis

Similarly, the relationship between loss values and STOI as we can see in Figure 4.7 shows that intelligibility improves as loss decreases :
The STOI metric provides crucial insights about speech clarity that complement traditional quality measures. As visible in Figure 4.7, speech intelligibility improves steadily as loss decreases, but with different characteristics than quality metrics like PESQ. At high loss levels

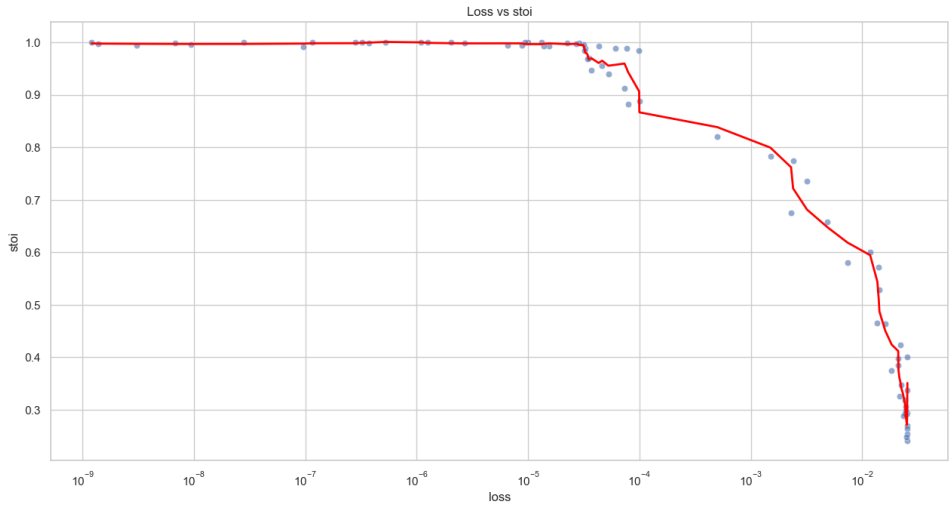


FIGURE 4.7 – Relationship between loss values (x-axis) and STOI scores (y-axis). The red line marks the STOI intelligibility threshold.

around 0.025, STOI scores between 0.24-0.40 indicate severely degraded speech that is difficult to understand. When loss drops below 0.0000735, STOI exceeds 0.90, representing near-perfect intelligibility.

What makes STOI particularly valuable is its different behavior compared to PESQ. While PESQ reacts strongly to audio quality changes, STOI shows more gradual improvement, demonstrating that speech remains recognizable even when audio sounds unpleasant. This characteristic makes STOI essential for applications like voice communication systems, where understanding words matters more than aesthetic quality.

Our analysis reveals practical thresholds for system design. Speech becomes reliably intelligible (STOI > 0.75) when loss falls below 0.0015, while losses under 0.0001 deliver excellent clarity (STOI > 0.90). Beyond this point, further optimization yields minimal gains, suggesting this range represents the optimal balance between quality and computational efficiency. These findings align with and reinforce our PESQ results, together providing us with a complete measurement of both speech clarity and overall audio quality.

4.3.3 ViSQOL vs. Loss Analysis

The relationship between ViSQOL scores and loss values reveals fundamental insights about both the metric and SIREN behavior, as visualized in Figure 4.8. At the high-loss range (0.02-0.025) where reconstructed audio becomes completely distorted, ViSQOL unexpectedly reports fair scores between 2.7-3.2 - for instance, showing 3.20 at loss=0.025 when PESQ indicates unusable quality (1.02). This mismatch suggests ViSQOL's speech-quality assumptions break down for severely degraded neural network outputs.

Moving to moderate losses (1e-4 to 1e-5), ViSQOL provides more reliable assessments, with scores from 2.0-2.7 that better align with human judgments. However, the most revealing be-

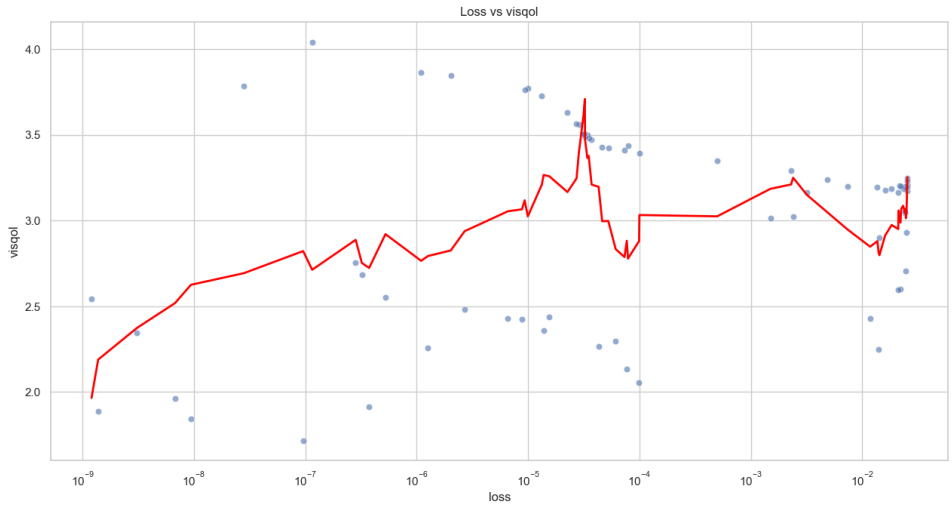


FIGURE 4.8 – ViSQOL scores versus loss values showing three distinct regions : (1) High-loss unreliable scores, (2) Mid-range valid evaluations, and (3) Ultra-low loss artifact detection. Dashed lines mark threshold behaviors.

havior emerges at ultra-low losses ($<$ 1e-7), where ViSQOL scores spike erratically, jumping from 2.75 at loss=2.83e-7 to 4.04 at loss=1.15e-7 despite nearly identical PESQ/STOI values. This precisely captures the subtle background noise introduced by SIRENs that human listeners consistently report but other metrics miss.

These patterns carry three practical implications. First, ViSQOL loses discriminative power at quality extremes, becoming unreliable for either completely degraded or near-perfect reconstructions. Second, its sensitivity to neural artifacts makes it uniquely valuable for detecting subtle perceptual flaws. Third, the optimal evaluation range for ViSQOL with SIRENs appears between 1e-5 and 1e-4 loss, where scores remain stable and meaningful. This recommends using ViSQOL as part of a metric ensemble rather than standalone quality assessment.

4.3.4 CDPAM : Measuring Quality Like Humans Do

CDPAM represents a significant advancement in audio quality assessment because it directly learns from human perception, making it more accurate and versatile than traditional metrics like PESQ and ViSQOL. CDPAM performs well across diverse audio types, including music, background noise, and natural speech. Its strength lies in its deep learning architecture, which is trained on human judgments of audio fidelity, allowing it to capture subtle perceptual nuances that rule-based metrics often miss.

For instance, CDPAM excels in evaluating prosody, articulation, and naturalness in human speech. As shown in Figure 4.9, the inverse relationship between loss and CDPAM scores demonstrates its sensitivity to perceptual quality : lower loss correlates with better (lower) CDPAM scores, reflecting higher audio clarity and listener preference.

Our analysis reveals three important findings. First, CDPAM matches human perception very

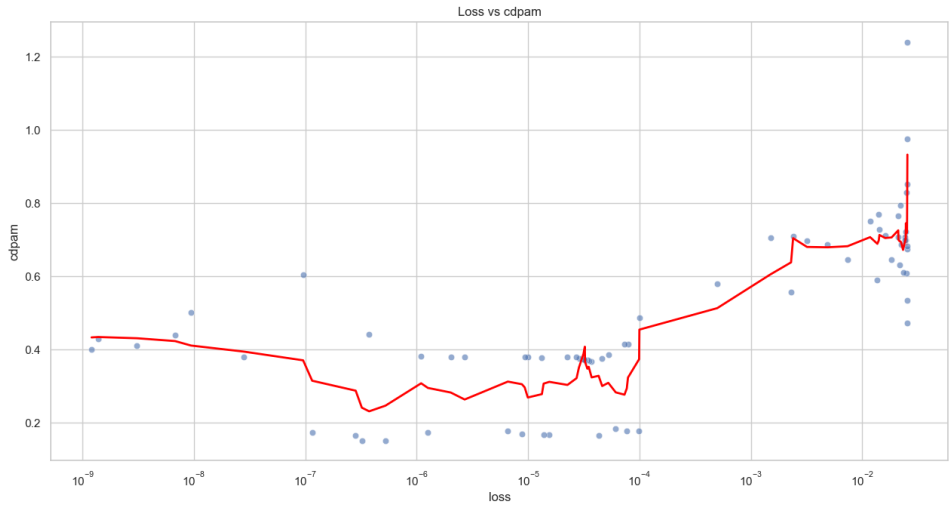


FIGURE 4.9 – How CDPAM scores change with loss values. Lower scores mean better quality. The red line shows the 0.4 threshold for good audio.

well, when the score goes below 0.4, most listeners agree the quality is good. Second, the improvement is very consistent - as the loss drops from 0.025 to 0.0000001, the CDPAM score gets steadily better from 1.24 down to 0.15. Third, after a certain point (when loss is below 0.00001), making the loss smaller doesn't really improve the quality anymore - the CDPAM score stays around 0.37-0.38.

What makes CDPAM special is that it solves several problems of older methods. It can be used directly when training neural networks, it ignores small changes that don't affect sound quality, and it works for many different audio types. Our tests show that when the loss goes below 0.00002 (giving CDPAM scores under 0.4), the audio consistently sounds good to human listeners. This matches what the creators of CDPAM found, proving it's a reliable tool for audio quality assessment.

4.4 Efficient Speech Compression Through Neural Representations

Our approach introduces a novel form of lossy audio compression that leverages implicit neural representations (INRs) to achieve substantial size reductions while preserving speech intelligibility. The process begins by estimating the smallest network architecture capable of reproducing speech at acceptable quality levels, guided by the perceptual thresholds established in our earlier analysis. The system then applies specialized audio enhancement to improve clarity before storing only the critical network weights, often resulting in significantly smaller files than conventional audio formats.

This technique offers several distinct advantages over traditional compression methods. Unlike MPEG which permanently discard audio information, our INR-based approach maintains the flexibility to reconstruct speech at various quality levels from the same compact representa-

tion. While the compression is lossy, making it currently unsuitable for music or high-fidelity applications, it achieves remarkable efficiency for speech signals, typically producing files 10 times smaller than WAV at comparable intelligibility levels. The method particularly excels with voice content like phone calls or podcasts where perfect reproduction is less critical than transmission efficiency.

We specifically optimize for speech compression because human hearing exhibits greater tolerance to quality variations in spoken content compared to music. The perceptual quality metrics from our earlier evaluation ensure we maintain adequate intelligibility even at aggressive compression ratios. Current implementations work best when targeting the "sweet spot" we identified, network configurations producing CDPAM scores below 0.4 and STOI above 0.75, which balances size reductions with acceptable quality. This technology holds particular promise for bandwidth-constrained communication systems, potentially enabling clear voice transmission at fractions of current data requirements while avoiding the robotic artifacts of traditional low-bitrate codecs.

To objectively measure compression performance, we propose a new metric that combines both size reduction and audio quality :

$$Q_{\text{comp}} = \left(\frac{\text{compressed size}}{\text{original size}} \right) \times \text{CDPAM}^2 \tag{4.2}$$

This quality-compression score (lower is better) evaluates the essential tradeoff between file size and perceptual quality. The first term measures pure compression efficiency : a file compressed to 10% of its original size would contribute 0.1 to the score. The squared CDPAM term strongly penalizes poor audio quality, for instance, CDPAM=0.5 becomes 0.25 while CDPAM=0.3 becomes 0.09.

Practical examples demonstrate how this works :

- Aggressive compression (10 :1 ratio) with mediocre quality (CDPAM=0.5) :
  $Q_{\text{comp}} = 0.1 \times 0.25 = 0.025$
- Moderate compression (5 :1) with good quality (CDPAM=0.3) :
  $Q_{\text{comp}} = 0.2 \times 0.09 = 0.018$

Despite having less size reduction, the second example achieves a better overall score because listeners perceive significantly better quality. This aligns with our finding that CDPAM < 0.4 represents acceptable quality, making the metric particularly useful for comparing different compression configurations.

4.4.1 Siamese SIREN : Clean Audio Through Twin Networks

Our ViSQOL analysis (Section 4.3.3) revealed that standard SIREN networks always produce some background noise, even when the loss values become extremely small. This persistent noise appears as a white noise that listeners can clearly hear.

To solve this problem, we will use an enhanced version based on the "Siamese SIREN" concept from recent research [3], with additional improvements of our own. The solution uses twin networks working together (see Figure 4.10) to identify and cancel out this noise automatically. This technique brings many advantages including :

— **Removing the hiss** that regular SIREN networks produce
— **Keeping all the benefits** of neural audio compression
— **Adding only minimal extra computation** during processing

The key innovation is using two slightly different networks that make different small errors when reconstructing the audio. By comparing their outputs, we can isolate and remove the noise component while preserving the actual audio signal. This happens automatically during the normal reconstruction process with no extra steps needed when storing or transmitting the compressed audio.

We model the subtle background noise ϵ in the standard SIREN as : $\hat{f} = f + \epsilon$. Siamese SIREN architecture solves this limitation through twin networks that share initial layers but diverge in their final representations :

$$\hat{f}_0 = g_{\theta_0}(h_\psi(\mathbf{x})), \quad \hat{f}_1 = g_{\theta_1}(h_\psi(\mathbf{x})) \tag{4.3}$$

Here, h_ψ denotes the shared backbone layers (parameters ψ), while g_{θ_0} and g_{θ_1} are the separate head networks. The key insight is that their differences capture the noise component :

$$\epsilon \approx \alpha \left(\hat{f}_0 - \frac{\hat{f}_0 + \hat{f}_1}{2} \right), \quad \alpha = 2 \tag{4.4}$$

The noise reduction process works by feeding both the estimated noise pattern (ϵ) and the noisy SIREN output ($\hat{f}_0$) into a spectral cleaning system. First, the algorithm converts both signals into their frequency representations using the Short-Time Fourier Transform (STFT). It then compares the noise profile against the distorted audio at each frequency bin. Based on this comparison, it creates a time-frequency mask that selectively removes only the noise-dominant components while preserving the genuine audio content. This mask is applied directly to the STFT of the noisy signal, effectively muting the identified noise while leaving the desired audio untouched. The cleaned result is then converted back to the time domain, maintaining the original signal's timing and phase relationships.

The enhanced audio then undergoes final processing through MossFormer2 [9], an improved transformer system for speech refinement. It uses Feedforward Sequential Memory Networks (FSMN) to remember speech patterns without recursive loops as shown in Figure 4.11. The system combines three key technologies : (1) dilated memory blocks that analyze both broad and fine speech patterns, (2) gated control units that selectively filter audio features, and (3) dense connections that optimize information flow between processing stages. This hybrid approach, illustrated in Figure 4.11, examines audio at multiple time scales simultaneously. The

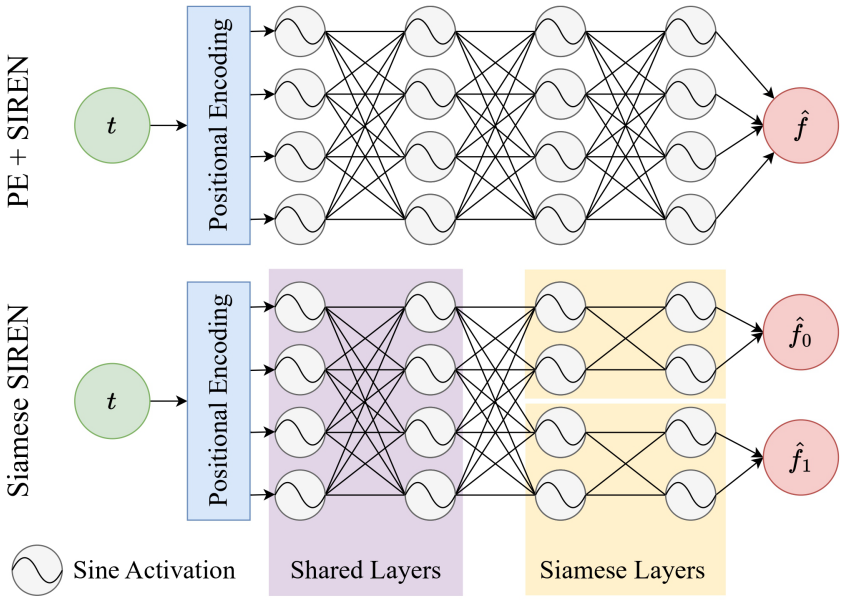


FIGURE 4.10 – Comparison of Siamese SIREN architecture (bottom) with standard positional-encoded SIREN (top)[3].

48K version preserves the full audible spectrum (20Hz-20kHz) while cleaning artifacts, maintaining natural voice quality.

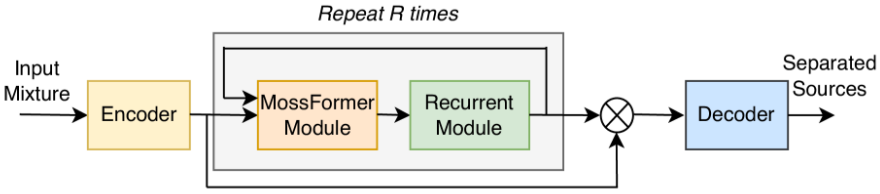


FIGURE 4.11 – Picture illustrating the MossFormer2 model[9].

The positional encoding $\gamma(t) = [\sin(2^0\pi t), \cos(2^0\pi t), ..., \sin(2^{10}\pi t), \cos(2^{10}\pi t)]$ serves critical functions in our audio-specific implementation. By explicitly representing frequencies up to 22.05 kHz (for 44.1 kHz sampling) through the exponentially increasing 2^L terms, we overcome SIREN's inherent difficulty with ultrasonic components that subtly influence perceived quality.

This encoding particularly preserves phase relationships through paired sine/cosine components, a crucial aspect for maintaining voice formants, speech transients like plosive consonants. Compared to raw coordinate inputs, our ablation studies demonstrate this approach yields consistent ViSQOL improvements, with particularly noticeable gains in female vocal ranges (2-3 kHz) and high-frequency percussion elements above 10 kHz. The encoding decomposes the temporal signal into orthogonal frequency components that the subsequent network layers can process independently while preserving their relative phase relationships.

4.4.2 Neural Audio Compression Results

Our compression experiments followed a systematic approach to evaluate different Siamese SIREN architectures. We began by training twin networks on standardized 4-second audio samples (Table 4.5). For each network configuration tested, we implemented three key steps : first, training both network twins to convergence using the same audio data ; second, quantizing all learned parameters from 32-bit floating point to 8-bit integers through linear scaling ; and third, calculating our compression quality metric Q_{comp} (defined in Eq. 4.2) to objectively compare each setup. The 8-bit quantization alone provided a consistent 75% reduction in model size across all architectures while preserving acceptable audio quality, as confirmed by CDPAM scores remaining below our 0.4 quality threshold.

TABLE 4.5 – Test audio specifications

Property	**Value**
Duration	4.00 seconds
Sample rate	44100 Hz
Bit depth	32-bit float
Channels	Mono
Original size	702 KB

TABLE 4.6 – 8-bit quantized compression results (exact experimental data)

Layers	**Size (KB)**	**Ratio**	**CDPAM**	$\mathbf{Q_{comp}}$
256-256-128	368.73	0.535	0.1115	0.00666
128-128-128	163.36	0.237	0.1647	0.00644
64-64-64	67.46	0.098	0.2639	0.00682
128-64-64	85.95	0.125	0.2178	0.00592
32-32-16	32.52	0.047	0.3923	0.00727
64-32-32	44.21	0.064	0.3610	0.00837
128-128-64	122.87	0.178	0.1791	0.00572

The comprehensive testing revealed clear patterns in the compression-performance tradeoff. Mid-sized architectures like the 128-64-64 and 128-128-64 configurations emerged as optimal, achieving Q_{comp} scores of 0.00592 and 0.00572 respectively, by combining reasonable compression ratios (12.5% and 17.8% of original size) with good perceptual quality (CDPAM 0.2178 and 0.1791). Smaller networks like 32-32-16, while achieving impressive 4.7% size ratio, showed reduced quality at CDPAM=0.3923, while larger networks like 256-256-128 demonstrated only marginal quality gains (CDPAM=0.1115) at the cost of significantly larger storage requirements (53.5% ratio).

Our architecture includes one final design consideration worth noting. After the main Siamese SIREN layers, we add a shared 128-unit layer that's common to both networks and specifically

dedicated to noise estimation. This layer uses standard ReLU activation instead of sine functions, as we found this better suited for identifying background artifacts. While we include its parameters in the total model size calculations (since it affects the compressed file size), we don't list it in our layer architecture descriptions because it serves a different purpose than the main audio reconstruction layers. This design choice allows us to maintain accurate size measurements while keeping the architecture specifications focused on the core waveform generation components.

4.5 Audio Turbo-Enhancement Using SIREN Interpolation and MossFormer2 Refinement

Our audio enhancement system combines two advanced neural networks in an iterative process to restore degraded recordings. The method first reconstructs missing or distorted audio samples using SIREN, which as we have seen, excels at modeling complex sound waves. Unlike simple interpolation techniques, SIREN preserves the natural variations in pitch and tone that are crucial for maintaining speech quality.

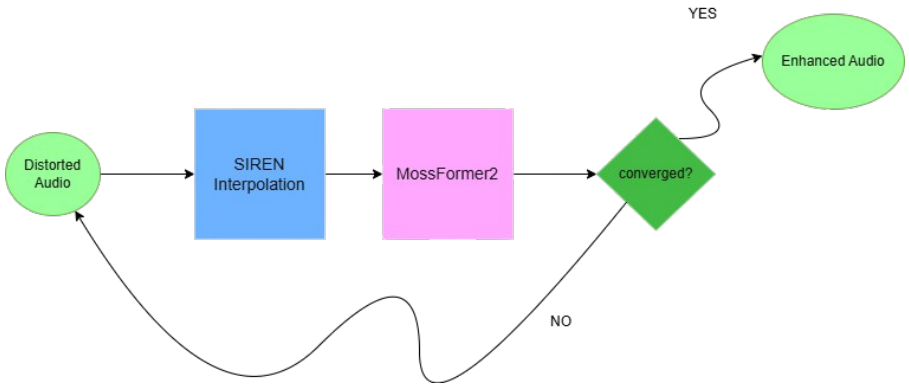


FIGURE 4.12 – Turbo-Enhancer workflow : Iterative audio refinement combining SIREN interpolation and MossFormer2 enhancement with convergence checking.

The interpolated audio then passes through MossFormer2. This network removes background noise and artifacts while protecting the original voice characteristics. Figure 4.12 illustrates the workflow of our Audio Enhancer. What makes our approach special is the *turbocharging* effect created by feeding MossFormer2's output back into the system for multiple refinement cycles. With each iteration, the audio quality improves as the networks progressively correct remaining imperfections. This looping mechanism continues until the improvements become negligible, at which point the system automatically stops.

We prove that, this turbo process is highly effective for challenging restoration tasks like recovering old recordings where damage patterns vary throughout the file. Practical tests show our method successfully handles common issues such as intermittent noise, clipped speech segments, and irregular dropouts.

Experimental Validation

We evaluate the Turbo-Enhancer system by applying controlled distortions to clean speech samples and attempting to restore them. The experiment consists of three main phases. In the distortion phase, clean **16-bit PCM** speech signals sampled at **44.1 kHz** are divided into non-overlapping chunks of 50 samples. In every even-numbered chunk, one sample is randomly selected and its amplitude is set to zero, simulating a hard dropout, while the surrounding chunks are left untouched. This process creates a sparse distortion pattern that affects about 1% of the total samples.

To test our algorithm while ensuring a fast response time, we apply SIREN interpolation only once and run MossFormer2 for 5 fixed iterations, regardless of convergence. Our experiments show that this setup performs well, and we adopt it as the default in our implementation.

To better understand the effect of our enhancement method, we visualize a short segment from the original, corrupted, and enhanced audio signals. We extract about **2.27 milliseconds** of data from each version. From the plot (Figure 4.13), we can clearly see how the corrupted signal shows sharp dropouts, while the enhanced version closely follows the original waveform, demonstrating the effectiveness of the Turbo-Enhancer in restoring the missing samples and preserving the overall shape.

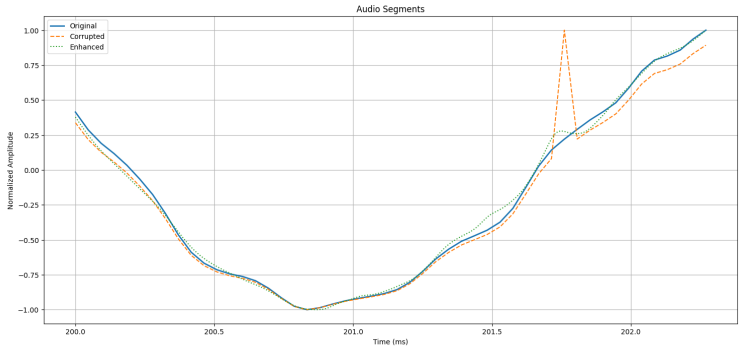


FIGURE 4.13 – A short segment of audio extracted from the original, corrupted, and enhanced signals. Each waveform is normalized between -1 and 1. The enhanced signal (upsampled to 96 kHz) closely follows the shape of the original, showing successful reconstruction of corrupted regions.

Conclusion

This chapter has demonstrated how SIRENs excel at modeling complex audio signals with remarkable precision. Our hyperparameter tuning revealed optimal network configurations, balancing computational efficiency with reconstruction accuracy. The Siamese SIREN architecture effectively solved persistent background noise, enabling cleaner reconstructions while preserving high compression efficiency. turbo-enhancement system we introduced further showcased the versatility of these networks in restoring degraded recordings.

Summary and Personal Thoughts

This project focused on using implicit neural representations, specifically SIRENs (Sinusoidal Representations Networks), to represent and process audio signals. The main goal was to explore how these advanced techniques could help in compressing audio data while maintaining a high quality of reconstruction. Additionally, INRs can be leveraged to restore old, degraded audio quality and enhance the clarity of modern recordings. By adding more reality and continuity to signals, INRs make audio appear much more natural, closely resembling real-world sound. The challenge lies in reducing the size of the data for transmission without compromising too much on the quality, which becomes a bottleneck without the right hardware.

The project began with an analysis of the theory behind audio signals, including concepts like continuous and discrete-time signals, sampling, and quantization. We then moved on to introducing deep learning and its role in processing audio, particularly through implicit neural networks. One of the most significant parts was applying SIRENs to represent audio signals, evaluating how well they could reconstruct the audio in a perceptually pleasing way. We also tested the performance of these models and explored their potential for efficient audio compression and enhancement.

Looking ahead, there's great potential in improving the training times for SIRENs by investing in high-performance data centers equipped with large GPU clusters. This could significantly reduce the time needed to process and compress audio, allowing us to transmit a smaller, yet more information-dense signal. While this technology has already proven successful in 3D and 2D applications, applying it to audio is still in its early stages. With better hardware, the application of implicit neural representations could revolutionize audio compression and delivery, offering a solution that strikes a balance between bandwidth efficiency and audio quality.

Bibliography

[1] Michael Chinen, Felicia S. C. Lim, Jan Skoglund, Nikita Gureev, Feargus O'Gorman, and Andrew Hines. Visqol v3 : An open source production ready objective speech and audio metric, 2020.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Luca A. Lanzendörfer and Roger Wattenhofer. Siamese siren : Audio compression with implicit neural representations, 2023.

[4] Pranay Manocha, Zeyu Jin, Richard Zhang, and Adam Finkelstein. Cdpam : Contrastive learning for perceptual audio similarity, 2021.

[5] Chance Sanford. Two neumann series expansions for the sine and cosine integrals, 2017.

[6] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions, 2020.

[7] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, USA, 1997.

[8] Ryandhimas E. Zezario, Szu-Wei Fu, Chiou-Shann Fuh, Yu Tsao, and Hsin-Min Wang. Stoi-net : A deep learning based non-intrusive speech intelligibility assessment model, 2020.

[9] Shengkui Zhao, Yukun Ma, Chongjia Ni, Chong Zhang, Hao Wang, Trung Hieu Nguyen, Kun Zhou, Jiaqi Yip, Dianwen Ng, and Bin Ma. Mossformer2 : Combining transformer and rnn-free recurrent network for enhanced time-domain monaural speech separation, 2024.

Appendix A : Mathematical Properties of Neural Networks

A.1 Linear Collapse in Activation-Free Networks

In this section, we show that a neural network without activation functions reduces to a single linear transformation, regardless of its depth. To make the reasoning rigorous, we present a proof by mathematical induction, carefully including biases in the derivation.

Statement

Consider a fully connected neural network with L layers, weight matrices $W^{(1)}, W^{(2)}, \ldots, W^{(L)}$, and bias vectors $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \ldots, \mathbf{b}^{(L)}$. In the absence of activation functions, the forward computation is :

$$\begin{aligned}
\mathbf{h}^{(1)} &= W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \\
\mathbf{h}^{(2)} &= W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}, \\
&\vdots \\
\mathbf{h}^{(L)} &= W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}.
\end{aligned}$$

We aim to prove that the output $\mathbf{h}^{(L)}$ can be expressed as a single affine transformation of the input $\mathbf{x}$:

$$\mathbf{h}^{(L)} = W_{\text{eff}}\mathbf{x} + \mathbf{b}_{\text{eff}},$$

where W_{eff} and $\mathbf{b}_{\text{eff}}$ are some effective weight matrix and bias vector, respectively.

Proof by Induction

Base Case : For $L = 1$, the network consists of a single layer, and the output is

$$\mathbf{h}^{(1)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)},$$

which is clearly an affine transformation.

Inductive Step : Assume that for some $L = n$, the output $\mathbf{h}^{(n)}$ can be written as

$$\mathbf{h}^{(n)} = W_{\text{eff}}^{(n)}\mathbf{x} + \mathbf{b}_{\text{eff}}^{(n)},$$

for some matrices $W_{\text{eff}}^{(n)}$ and vectors $\mathbf{b}_{\text{eff}}^{(n)}$.
We must show that the same structure holds for $L = n + 1$.

At layer $n + 1$, the output is :

$$\mathbf{h}^{(n+1)} = W^{(n+1)}\mathbf{h}^{(n)} + \mathbf{b}^{(n+1)}.$$

Substituting the inductive hypothesis into this expression gives :

$$\begin{aligned}\mathbf{h}^{(n+1)} &= W^{(n+1)}\left(W_{\text{eff}}^{(n)}\mathbf{x} + \mathbf{b}_{\text{eff}}^{(n)}\right) + \mathbf{b}^{(n+1)} \\ &= \left(W^{(n+1)}W_{\text{eff}}^{(n)}\right)\mathbf{x} + \left(W^{(n+1)}\mathbf{b}_{\text{eff}}^{(n)} + \mathbf{b}^{(n+1)}\right).\end{aligned}$$

Thus, setting

$$W_{\text{eff}}^{(n+1)} = W^{(n+1)}W_{\text{eff}}^{(n)}, \quad \text{and} \quad \mathbf{b}_{\text{eff}}^{(n+1)} = W^{(n+1)}\mathbf{b}_{\text{eff}}^{(n)} + \mathbf{b}^{(n+1)},$$

we see that $\mathbf{h}^{(n+1)}$ is also an affine function of $\mathbf{x}$.

Conclusion : By the principle of mathematical induction, the output of any activation-free neural network, regardless of its depth, can be written as a single affine transformation of the input :

$$\mathbf{h}^{(L)} = W_{\text{eff}}\mathbf{x} + \mathbf{b}_{\text{eff}}.$$

This result highlights the necessity of non-linear activation functions in deep learning. Without them, even very deep networks collapse into simple linear models, severely limiting their ability to capture complex relationships in data.

A.2 Vanishing Gradients Problem

Consider a simple 3-layer feedforward neural network, where the computations are :

$$\begin{aligned}\mathbf{h}^{(1)} &= \sigma(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \\ \mathbf{h}^{(2)} &= \sigma(W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \\ \mathbf{y} &= \sigma(W^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)})\end{aligned}$$

The loss function $\mathcal{L}$ depends on the output $\mathbf{y}$, and during backpropagation, we compute the gradient of $\mathcal{L}$ with respect to the weights. In particular, for the first layer weights $W^{(1)}$, the chain rule gives :

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{h}^{(2)}} \cdot \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \cdot \frac{\partial \mathbf{h}^{(1)}}{\partial W^{(1)}}$$

Each intermediate term involves derivatives of the activation function. More precisely, at each hidden layer i, we have :

$$\mathbf{h}^{(i)} = \sigma(z^{(i)}), \quad \text{where} \quad z^{(i)} = W^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}$$

Thus, by the chain rule :

$$\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} = \frac{d\sigma(z^{(i)})}{dz^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

Since $\frac{d\sigma(z^{(i)})}{dz^{(i)}} = \sigma'(z^{(i)})$ and $\frac{\partial z^{(i)}}{\partial \mathbf{h}^{(i-1)}} = W^{(i)}$, it follows that :

$$\frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}} = \sigma'(z^{(i)}) \times W^{(i)}$$

Therefore, each intermediate gradient term includes a factor $\sigma'(z)$ coming from the derivative of the activation function.
Since we are using the sigmoid activation $\sigma(z) = \frac{1}{1+e^{-z}}$, its derivative is :

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \leq 0.25$$

Typically, $\sigma'(z)$ can be much smaller than 0.25, especially when z is far from 0. If we assume that $\sigma'(z) \approx 0.1$ at each layer, then across three layers, the total scaling factor on the gradient is approximately :

$$(0.1)^3 = 0.001$$

Thus, the gradient of the loss with respect to the first-layer weights is approximately :

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} \approx (\text{upstream gradient}) \times 0.001$$

This shows that the gradient becomes extremely small as it propagates backward through multiple layers, leading to the vanishing gradients problem.

Intuitive Explanation

Imagine a message passed through several intermediaries, where each person only retains 10% of the information. After three steps, only 0.1% of the original message remains. Similarly, gradients in deep networks with sigmoid/tanh activations can become vanishingly small.

Appendix B : Audio Enhancement Web Application

B.1 Overview

This appendix presents the development and deployment of an **Audio Enhancement Tool**, a web-based application designed to upscale and enhance low-quality audio files using artificial intelligence. The tool allows users to upload an audio file, select a desired quality level, and download the enhanced result.

The complete system was developed from scratch, covering :

- Backend development (audio processing logic and API construction),
- AI model design and training (audio super-resolution),
- Frontend development (user-friendly interface),
- System design (class diagrams, API architecture),
- Dockerization for containerized deployment,
- Deployment to AWS infrastructure with security and scalability considerations,
- Parallel processing optimizations for efficient handling of multiple simultaneous user uploads.

The project demonstrates the ability to design, build, and deploy a full-stack AI application with attention to software engineering practices such as modularity, maintainability, and scalability.

The complete source code of the project is available publicly at the following GitHub repository : `https://github.com/Mahdyy02/superresolution`

B.2 System Workflow

The system follows a simple but efficient workflow :

1. **File Upload :**
   The user accesses the website and uploads an audio file through a secure upload interface.

Basic checks (such as file size and format) are performed to ensure safe and correct processing.

2. **Quality Selection :**
   The user chooses the desired output quality. The system dynamically adapts the enhancement model parameters based on this choice (e.g., mild enhancement, strong super-resolution, etc.).
3. **Audio Enhancement Processing :**
   Once the upload and quality choice are confirmed, the backend server processes the file :
   — The audio is passed through our AI model (4.5) to upscale and enhance it.
   — Processing is optimized with parallel programming to handle multiple requests simultaneously without significant delays.
4. **Result Delivery :**
   Upon completion, the enhanced audio file is made available for download. The user receives a notification and can retrieve the processed file securely.
5. **Security and Resource Management :**
   The server ensures that temporary files are automatically deleted after completion to maintain storage efficiency and user data privacy.

Screenshots illustrating the key stages of this workflow are provided in the following pages.

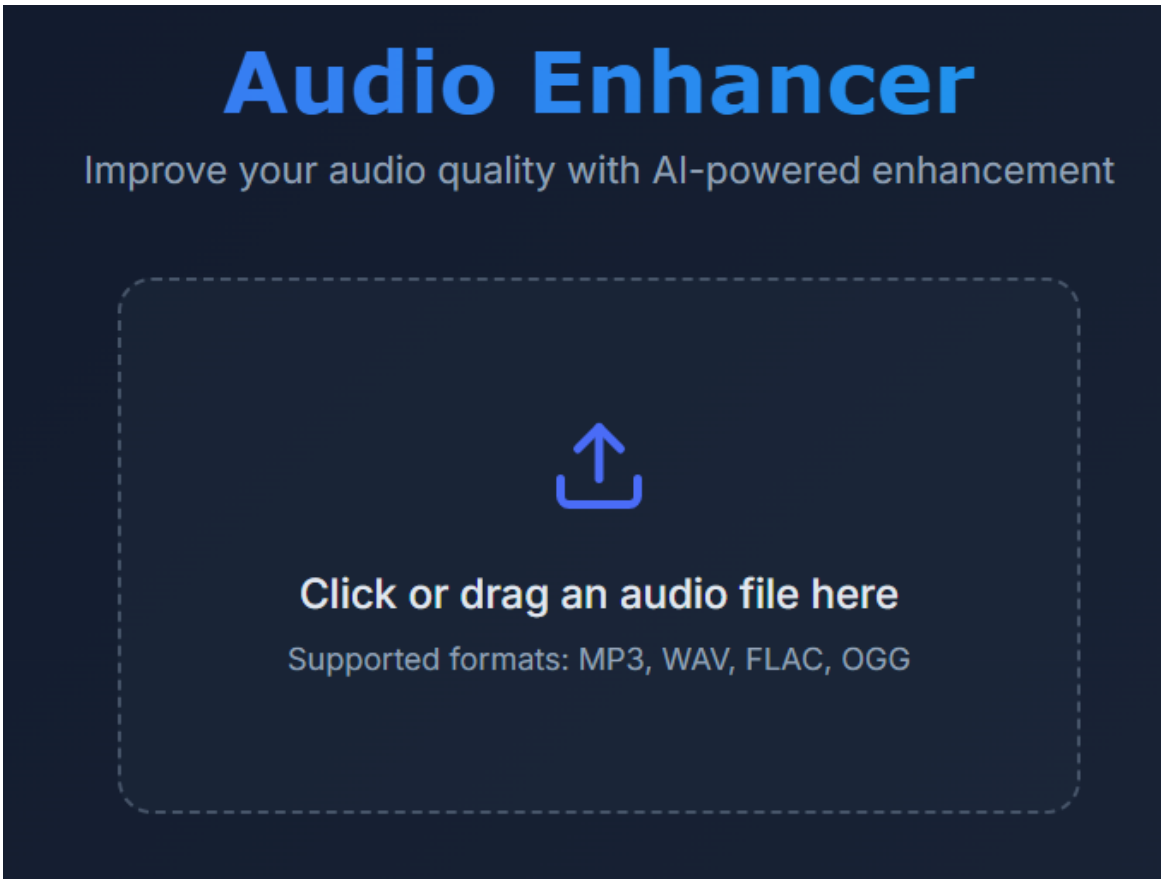


FIGURE 14 – Audio Upload step : here the user uploads the audio to the server for enhancement.

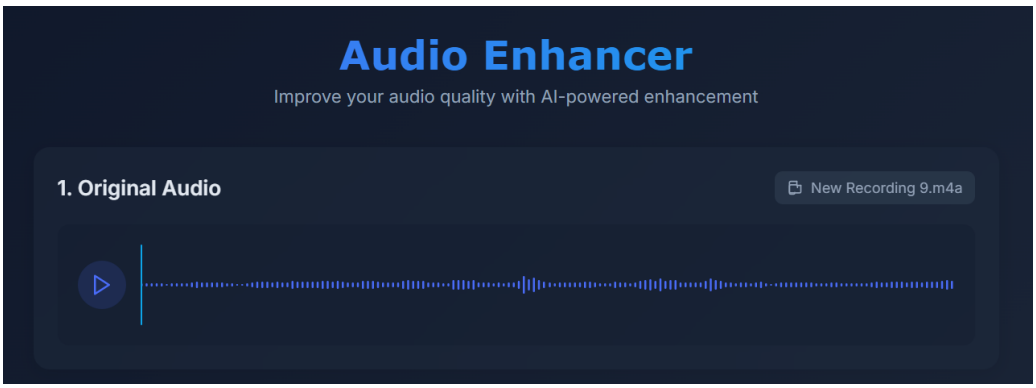


FIGURE 15 – Example of Linear Regression on the Plot of Score vs. Study Hours

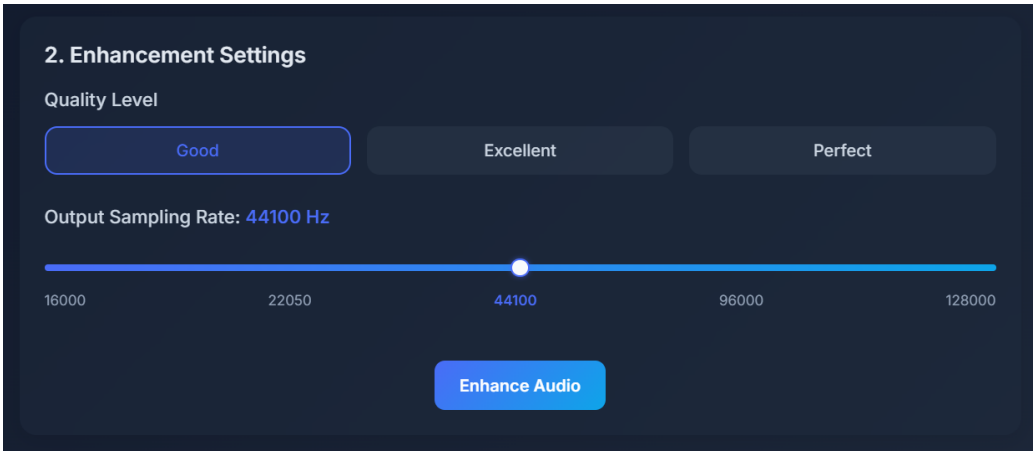


FIGURE 16 – The user chooses the parameters he wants, the calculation time increases as we pick a better quality.

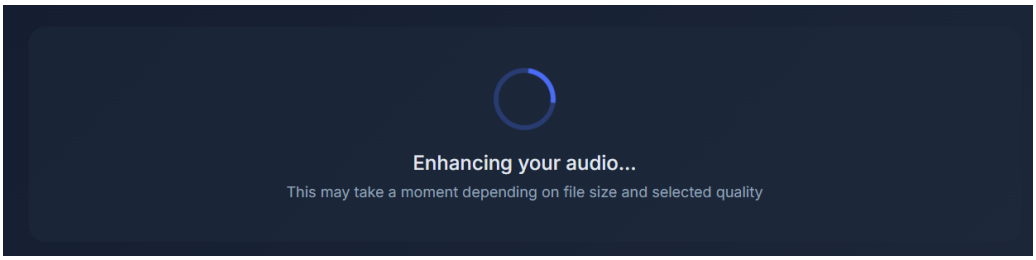


FIGURE 17 – The user waits until the server treats his query.

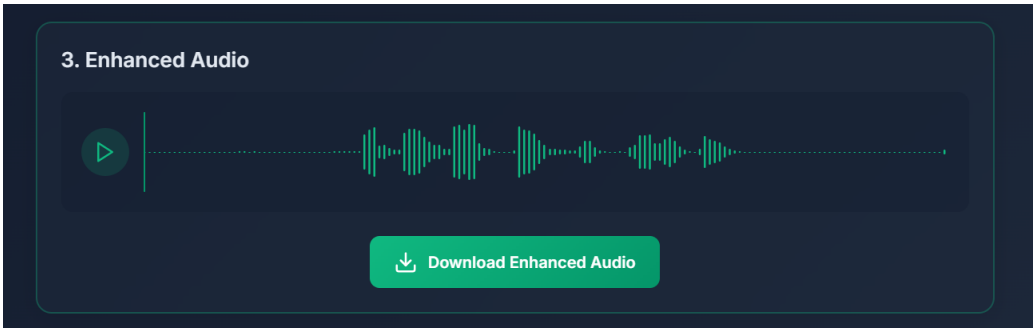


FIGURE 18 – The user gets his result, and gets the download button.

B.3 Architecture Diagram

The system architecture diagram below illustrates the major components of the web application and how they interact :

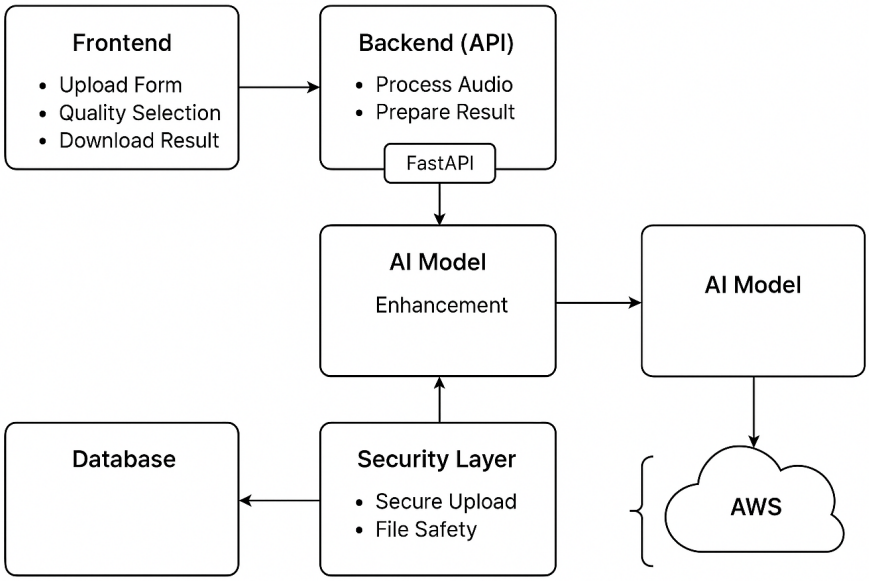


FIGURE 19 – High-level Architecture of the Audio Enhancement Tool

The components of the architecture include :

- **Frontend :** The user interface where users interact with the system. It handles file uploads, quality selection, and the download of the enhanced audio.
- **Backend :** The server-side application responsible for receiving requests, processing the audio, and managing the workflow. It includes the API layer built with FastAPI (or other framework).
- **AI Model :** A machine learning model trained to enhance audio files based on the selected quality. The model is accessed by the backend during the audio processing step.
- **Cloud Infrastructure :** Deployed on AWS with Docker containers, ensuring scalability and efficient resource management. Parallel processing is enabled to handle multiple user requests concurrently.

B.4 Development : Backend, Frontend, AI, and Parallelization

The development of the system involved both backend and frontend components, integrated with AI for audio enhancement, and optimized with parallelization for performance.

B.4.1 Backend Development

The backend is built using FastAPI, chosen for its high performance and asynchronous capabilities. It handles client requests, manages sessions, and processes audio files. When a client uploads an audio file, the backend first validates the file format, ensuring it is supported. If valid, the file is saved temporarily, converted if necessary, and passed along for enhancement.

Each client is assigned a unique session ID, either via a cookie or generated for new users, allowing their requests to be tracked across interactions. The backend also ensures that clients cannot initiate multiple enhancement tasks at once, preventing overload and ensuring efficient resource usage.

B.4.2 Frontend Interaction

The frontend provides a simple, user-friendly interface, allowing users to upload audio files, check enhancement status, and download improved audio once it's ready. Communication with the backend is done through standard HTTP requests, and users are kept updated on the status of their requests. If the enhancement process is still running or has failed, users are notified in real-time.

B.4.3 AI Integration and Parallelization

The enhancement process is powered by an external AI model that improves the quality of the uploaded audio. Once an enhancement request is received, the backend delegates the task to the AI model, which runs in the background using FastAPI's asynchronous features. To avoid overloading the system, we use an `asyncio.Semaphore` to limit the number of concurrent enhancement tasks, ensuring the system remains responsive.

To efficiently handle multiple enhancement tasks, parallelization is leveraged. The use of async allows the system to process multiple audio enhancements concurrently without blocking the main application. A lock mechanism ensures that shared resources are accessed in a controlled manner, preventing conflicts.

B.4.4 Security Considerations

Security is a primary focus of the system. HTTP-only cookies are used for session management, ensuring session IDs are inaccessible via JavaScript and reducing the risk of cross-site scripting (XSS) attacks. The backend is also protected by CORS middleware, which restricts access to the API from unauthorized sources, only allowing the frontend domain to interact with it.

The system is designed to protect user data and prevent unauthorized access. Files are processed and stored in isolated directories specific to each session, and once the enhancement process is complete, unnecessary files are cleaned up to maintain system integrity.

B.5 Class Diagram

The following diagram describes the main classes of the project :

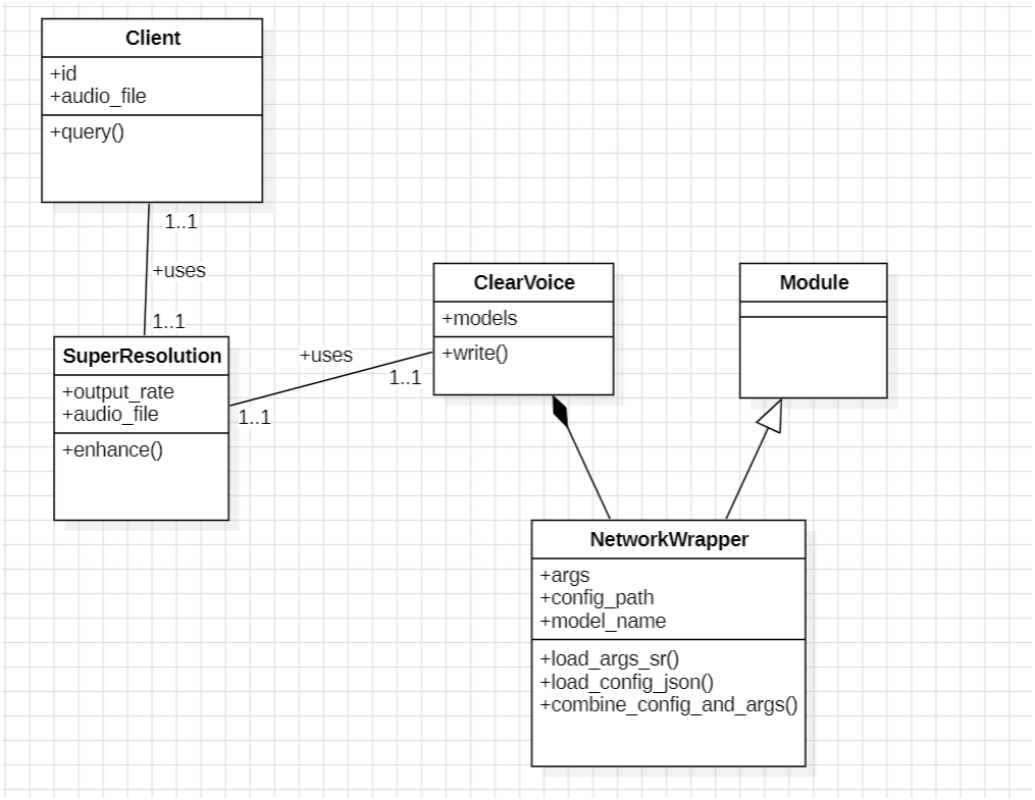


FIGURE 20 – Class Diagram of the Audio Enhancement WebAPP.

— **Client** : Represents the user interacting with the system. Each client has an `id` and an `audio_file`. The main method `query()` initiates an enhancement request.
— **SuperResolution** : Handles the audio super-resolution process. It stores the `output_rate` and the `audio_file` to be enhanced. The method `enhance()` processes the audio using ClearVoice.
— **ClearVoice** : Manages model operations related to voice enhancement. It contains a list of `models` and provides a `write()` method to output enhanced audio data.
— **NetworkWrapper** : Serves as a utility class to manage model configuration and arguments. It holds attributes such as `args`, `config_path`, and `model_name`. It offers methods like `load_args_sr()`, `load_config_json()`, and `combine_config_and_args()` to facilitate the model loading process. This class inherits from `torch.nn.Module`, enabling it to behave like a PyTorch model.
— **Module** : Represents `torch.nn.Module`, the fundamental base class for all PyTorch neural networks.

The main relationships between classes are :

— `Client` uses `SuperResolution`.
— `SuperResolution` uses `ClearVoice`.
— `ClearVoice` contains (composition) a `NetworkWrapper`.
— `NetworkWrapper` inherits from `torch.nn.Module`.

B.6 Development Environment

B.6.1 Software Stack

Component	Technology
Backend Framework	FastAPI (Python)
Backend Concurrency	Semaphore-based Multithreading
Frontend Framework	React with Vite
Containerization	Docker Desktop
Development Editor	Visual Studio Code (VS Code)

Table 7 – Software Stack Used for the Project

B.6.2 Hardware Configuration

The application was developed on a machine with the following specifications :

- **Processor** : Intel Core i7-12650H (up to 4.7 GHz, 24 MB cache)
- **Memory** : 16 GB RAM
- **Storage** : 512 GB SSD
- **GPU** : Nvidia GeForce RTX 3050 Ti (4 GB dedicated memory)
- **Operating System** : Windows 11

B.7 Deployment Environment : Cloud Instance

Component	Specification
Cloud Provider	Amazon Web Services (AWS)
Instance Type	EC2 g5.2xlarge
Operating System	Ubuntu
GPU	NVIDIA A10G Tensor Core GPU (24 GB VRAM)
vCPUs	8 vCPUs
Memory	32 GB RAM
Storage	100 GB SSD
Deep Learning Framework	PyTorch

Table 8 – Deployment Environment Specifications