TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS

A Major Project Report
On
Data-driven Video Codec with Implicit Neural Representations

Submitted By:

Abhinav Chalise	THA077BEI003
Nimesh Gopal Pradhan	THA077BEI026
Nishan Khanal	THA077BEI027
Saugat Neupane	THA077BEI043

Submitted To:
Department of Electronics and Computer Engineering
Thapathali Campus
Kathmandu, Nepal

March 2025

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS

A Major Project Report
On
Data-driven Video Codec with Implicit Neural Representations

Submitted By:

Abhinav Chalise	THA077BEI003
Nimesh Gopal Pradhan	THA077BEI026
Nishan Khanal	THA077BEI027
Saugat Neupane	THA077BEI043

Submitted To:
Department of Electronics and Computer Engineering
Thapathali Campus
Kathmandu, Nepal

In partial fulfillment for the award of the Bachelor of Engineering in Electronics, Communication and Information.

Under the Supervision of
Er. Dinesh Baniya Kshatri

March 2025

DECLARATION

We hereby declare that the report of the project entitled **"Data-driven Video Codec with Implicit Neural Representations"** which is being submitted to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, in the partial fulfillment of the requirements for the award of the Degree of Bachelor of Engineering in **Electronics, Communication and Information**, is a bonafide report of the work carried out by us. The materials contained in this report have not been submitted to any University or Institution for the award of any degree and we are the only author of this complete work and no sources other than the listed here have been used in this work.

Abhinav Chalise (Class Roll No: THA077BEI003) _____

Nimesh Gopal Pradhan (Class Roll No: THA077BEI026) _____

Nishan Khanal (Class Roll No: THA077BEI027) _____

Saugat Neupane (Class Roll No: THA077BEI043) _____

Date: March 2025

CERTIFICATE OF APPROVAL

The undersigned certify that they have read and recommended to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, a major project work titled **"Data-driven Video Codec with Implicit Neural Representations"** submitted by **Abhinav Chalise, Nimesh Gopal Pradhan, Nishan Khanal, Saugat Neupane** in partial fulfillment for the award of Bachelor of Engineering in Electronics, Communication and Information. The Project was carried out under special supervision and within the time frame prescribed by the syllabus.

We found the students to be hardworking, skilled and ready to undertake any related work to their field of study and hence we recommend the award of partial fulfillment of Bachelor of Engineering in Electronics, Communication and Information.

Project Supervisor
Er. Dinesh Baniya Kshatri
Department of Electronics and Computer Engineering, Thapathali Campus

---

External Examiner

Project Co-ordinator
Er. Sudip Rana
Department of Electronics and Computer Engineering, Thapathali Campus

---

Head of the Department
Er. Umesh Kanta Ghimire
Department of Electronics and Computer Engineering, Thapathali Campus

March 2025

ACKNOWLEDGMENT

We would like to acknowledge **IOE, TU** for the inclusion of the major project in the course of Bachelor of Engineering in Electronics, Communication and Information.

We would sincerely like to thank our supervisor **Er. Dinesh Baniya Kshatri** for his invaluable guidance during this project.

We appreciate the **Department of Electronics and Computer Engineering, Thapathali Campus**, for their assistance and recommendations in the selection and hopefully successful completion of this project, as well as for enabling us to advance our professional development through it.

Additionally, we would like to thank all of our teachers, classmates, and other direct and indirect contributors for their insightful advice and recommendations.

Sincerely,

Abhinav Chalise (Class Roll No: THA077BEI003)

Nimesh Gopal Pradhan (Class Roll No: THA077BEI026)

Nishan Khanal (Class Roll No: THA077BEI027)

Saugat Neupane (Class Roll No: THA077BEI043)

ABSTRACT

This project explores a novel approach to audio-video representation and compression using the Sinusoidal Representation Network (SIREN), an Implicit Neural Representation (INR) model leveraging a sine activation function. Trained on five video datasets with audio, the unified model demonstrates competitive video compression results.The architecture consists of separate audio and video initialization layers, followed by fully connected shared hidden layers. Past the hidden layers, the model branches into three outputs: one for video representation and two for audio processing using Siamese layers, designed to enhance noise reduction. For one of the videos in the dataset, which was 6.08 MiB, the quantized student model achieved a PSNR of 28.7153 dB and an SSIM of 0.7529 for video, and a PSNR of 24.1786 dB with an LSD of 10.6928 dB for audio. Prior to quantization and distillation, the teacher model had achieved a PSNR of 21.88 dB, SSIM of 0.67 for video, and a PSNR of 62.50 dB with an LSD of 7.60 dB for audio. By applying 16-bit quantization and knowledge distillation, the unified model size was reduced significantly from 9.05 MiB to 2.48 MiB while maintaining competitive perceptual quality. Additionally, further compression using LZMA with xz reduced the model size to 2.33 MiB, achieving an overall compression ratio of 2.61. These advancements demonstrate the model's potential as a scalable solution for multimedia compression. Despite its success, challenges remain in optimizing representation of dynamic content, training on larger datasets, and fine-tuning for diverse video types and lengths. These findings underscore SIREN's promise as a unified and efficient framework for multimedia compression.

Keywords: Implicit neural representation, Siamese siren, Sine activation, Sinusoidal representation networks

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AVC	Advanced Video Coding
bps	bits per seconds
CNN	Convolutional Neural Network
codec	Coder Decoder
CRC	Cyclic Redundancy Check
CRF	Constant Rate Factor
dB	decibels
DCT	Discrete Cosine Transform
ECM	Enhanced Compression Model
FFT	Fast Fourier Transform
fps	frames per second
GAN	Generative Adversarial Network
GB	gigabytes
GiB	gibibytes
GPU	Graphics Processing Unit
HEVC	High-Efficiency Video Coding
IDE	Integrated Development Environment
IMDCT	Inverse Modified Discrete Cosine Transform
INN	Implicit Neural Networks
INR	Implicit Neural Representation
KB	kilobytes
kbps	kilobits per seconds
KiB	kibibytes
KNN	K-Nearest Neighbours
LPIPS	Learned Perceptual Image Patch Similarity

LSD	Log Spectral Distance
LSTM	Long Short-Term Memory
LZMA	Lempel-Ziv-Markov chain algorithm
MB	megabytes
Mbps	megabits per seconds
MDCT	Modified Discrete Cosine Transform
MiB	mebibytes
MLP	Multilayer Perceptron
MSE	Mean Squared Error
MSQE	Mean Squared Quantization Error
NeRA	Nerual Representations for Audios
NeRV	Neural Representations for Videos
NeRVA	Nerual Representations for Videos and Audios
NVC	Neural Video Coder Decoder
NVR	Neural Video Representation
PCM	Pulse Code Modulation
PSNR	Peak Signal-to-Noise Ratio
RBKD	Response-Based Knowledge Distillation
RGB	Red Green Blue
SIREN	Sinusoidal Representation Networks
SQNR	Signal-to-Quantization Noise Ratio
SSIM	Structural Similarity Index Measure
SVD	Singular Value Decomposition
UML	Unified Modeling Language
ViSQOL	Virtual Speech Quality Objective Listener
VVC	Versatile Video Coding
WebRTC	Web Real-Time Communication

1 INTRODUCTION

The amount of video content generated and consumed daily is growing exponentially [1] which has led to efficient video compression techniques to be increasingly vital. This project aims to explore methods of video compression using deep learning techniques and representing videos through implicit neural representation and compressing these neural networks.

1.1 Background

Video compression is a critical technology in today's digital age. It is used in many areas like streaming services, social media platforms, cloud storage, mobile communication. Traditional compression algorithms such as H.264, H.265 have been the backbone of video compression for decades. These algorithms use techniques like Discrete Cosine Transform (DCT)s and predictive coding to reduce file sizes by removing redundancies and less perceptible details. While effective, these methods face limitations in handling the increasing complexity and higher resolutions of modern media content, often resulting in a trade-off between compression efficiency and quality.

In recent years, deep learning has opened new avenues for video compression. Deep learning models like Convolutional Neural Network (CNN) and Autoencoders have shown remarkable ability to learn intricate patterns and representations within data. These models can outperform traditional algorithms by dynamically adapting to the unique features of different videos, thereby optimizing compression in a more intelligent and flexible manner. Notable advancements include using CNNs for feature extraction and Long Short-Term Memory (LSTM) networks for sequence learning, enabling models to handle spatial and temporal data more effectively. Additionally, hybrid models combining different neural network architectures have demonstrated enhanced performance and efficiency.

Implicit Neural Representation (INR) is an approach to encode data, such as images, 3D shapes, or audio, by using the power of neural networks. Unlike traditional methods that store data in discrete formats like pixels, INRs utilize neural networks to learn and represent continuous functions that can generate this data. In practice, a neural network is trained to take spatial coordinates (e.g., x, y, z for 3D shapes) as input and produce the corresponding data values (such as color or density) as output. This process results in a compact, continuous, and differentiable representation of the data.

Furthermore, the continuous nature of INRs allows for efficient storage and manipulation of data. For instance, instead of storing a large image as a collection of pixels, the

neural network function can generate any part of the image at any desired resolution, potentially saving memory and computational resources.

INRs are also differentiable, meaning they can be easily integrated with other neural network-based systems and optimized using gradient-based methods. This property is beneficial for tasks like optimizing 3D models for better fit with observed data, generating textures for computer graphics, and even compressing video data by learning efficient representations.

Overall, implicit neural representations provide a powerful framework for encoding, manipulating, and generating complex data, offering numerous advantages in terms of flexibility, efficiency, and quality of the resulting representations. We use the advancements of INRs to represnt videos in terms of neural networks and compress it as well as regenerate it as required.

1.2 Motivation

The rapid expansion of digital media consumption has created a need for more efficient compression methods. High-resolution formats, 4k and even 8k videos, and high-fidelity audio are becoming standard, which increases the demand for storage space and bandwidth. This project is motivated by the potential of deep learning and implicit neural representation to improve video compression. By using neural networks' ability to learn and optimize complex patterns, along with the precision of implicit neural representation to represent pixel and amplitude data, we aim to develop a compression model for audio-visual files. This can enhance user experience by providing high-quality media at reduced file sizes, addressing issues such as reducing data transmission costs, and improving download and upload speeds.

1.3 Problem Definition

This project aims to address the growing challenge of efficiently managing increasing volume and complexity of digital video content. As high-resolution formats like 4k videos and high-fidelity audio become more common, the demand for effective storage and transmission solutions is increased. The large file sizes of modern video content results in increased storage costs, longer download times and high bandwidth usage. This project aims to tackle these problems through implicit neural representation to develop a compression model that can compress media files without compromising too much on quality.

1.4 Objectives

The main objectives of our project can be summed up as:

- To develop an implicit neural representation model for videos with audio content
- To implement an efficient compression as well as encoding and decoding framework for the neural representation model

1.5 Project Scope and Applications

The scopes of our project is limited to video compression with some limitations and applications of the project discussed below.

1.5.1 Project Scopes

This project aims to develop advanced algorithms for video compression using innovative techniques such as INR. The primary focus lies in creating efficient methods to represent audio-video data through INR and then compressing these representations while maintaining acceptable audio-video quality. A key objective is to build a functional prototype that demonstrates the feasibility and effectiveness of the proposed compression model.

1.5.2 Project Limitations

While aiming for reduction in file sizes, the compression model may not achieve top-notch video quality preservation compared to uncompressed versions. The effectiveness of the model may vary depending on the complexity of the audio-video content, limiting its applicability to a specific range of video content. The computational resources required for training and deploying the model implementing INR techniques may pose practical constraints, affecting real-time processing capabilities or scalability.

1.5.3 Applications

- **Streaming Services:** Enhancing the efficiency of video compression can significantly reduce bandwidth usage and improve streaming quality, benefitting platforms like Netflix, YouTube, and other online video services.
- **Social Media:** Improved video compression can enable faster upload and download speeds, enhancing user experience on platforms like Instagram, Facebook etc.
- **Cloud Storage:** Efficient video compression can reduce storage costs and im-

prove access times for cloud storage providers like Google Drive, Dropbox

- **Broadcasting:** Television and live broadcast services can benefit from reduced transmission costs.
- **Surveillance Systems:** Compression of high-resolution surveillance footage can save storage space and facilitate faster retrieval and analysis of video data in security systems.

2 LITERATURE REVIEW

Video compression, a critical technology in the digital age, has evolved significantly over the years to meet the increasing demand for efficient data storage and transmission. Initially, human-devised algorithms and methods laid the foundation for video compression techniques. As technology advanced, a myriad of sophisticated algorithms and methodologies emerged, enabling more effective and efficient compression. From traditional methods like DCT and motion estimation to cutting-edge neural network-based approaches, the field of video compression boasts a diverse array of solutions. Numerous systems and tools have been developed to compress and decompress video content autonomously. This literature review summarizes some of the key methodologies that have been developed to address the challenges of video compression, highlighting the evolution from traditional techniques to modern machine learning-based solutions.

The MPEG-1 Audio standard [2], a component of the broader MPEG multimedia format, was developed to enhance the efficiency of digital audio compression. This standard comprises three distinct layers of compression—Layer 1, Layer 2, and Layer 3—with increasing complexity and efficiency. Layer 1 offers simplicity in implementation, while Layer 3, also known as MP3, provides superior performance at lower bit rates through sophisticated techniques such as Huffman coding and psychoacoustic models. These models are pivotal in predicting human auditory responses and minimizing perceptible noise caused by the compression process. By adapting the characteristics of the compression dynamically, the standard aims to maintain audio fidelity even at high compression ratios, thus facilitating efficient digital broadcasting and storage without compromising quality

The H.264 video coding standard [3], developed collaboratively by the Video Coding Expert Group and the Moving Pictures Expert Group, represents a significant advance in video compression technology. Officially known as MPEG-4 Part 10, Advanced Video Coding (AVC), this standard offers enhanced coding efficiency with a simpler syntax compared to its predecessors, making it highly compatible across various network protocols and multiplex architectures. Notably, H.264 incorporates features like intrapicture prediction, multiple reference pictures, variable block sizes, and an in-loop deblocking filter, which collectively contribute to its superior performance in applications ranging from video conferencing to video streaming and broadcasting across different transmission mediums

The High-Efficiency Video Coding (HEVC) standard [4], also known as H.265, has emerged as a successor to the widely used H.264/AVC, setting new benchmarks in video

compression technology. Introduced by the Joint Collaborative Team on Video Coding, HEVC primarily aims to address the increasing demand for higher video resolutions and quality, such as ultra-high definition and 4K, particularly over bandwidth-constrained networks. By incorporating advancements such as improved block partitioning flexibility, enhanced prediction capabilities, and increased parallel processing support, HEVC effectively doubles the compression efficiency compared to its predecessor. This allows HEVC to deliver similar video quality at approximately half the bitrate of H.264/AVC, which significantly optimizes data usage and transmission costs, particularly important for streaming high-definition video in mobile environments and over the internet.

Li et al. [5] introduce a conditional coding framework that utilizes deep learning to encode videos by using contextual information rather than just the residues. Deep contextual video compression framework advances this concept by using feature domain context as a condition, enabling the encoder and decoder to utilize high-dimensional information for reconstructing high-frequency content and achieving superior video quality. This approach addresses the limitations of predictive coding and also offers extensibility for tailored conditions, significantly outperforming previous state-of-the-art methods with a 26.0% bitrate saving for 1080P standard test videos compared to the x265 Coder Decoder (codec) using the very slow preset.

Further deepening our understanding, the exploration of latent spaces by Liu et al. [6] reveals that employing deep autoencoders trained with Generative Adversarial Network (GAN)s can effectively compress video data. This novel framework adopts a GAN-based autoencoder architecture, accompanied by trainable quantization and entropy coding, to compress video sequences within a latent vector space. This method begins by learning efficient latent space representations of video frames via a deep autoencoder trained with GANs, followed by inter-frame prediction using a convolutional LSTM network. This approach only stores the differences between predicted and actual representations in a compact latent space, reducing residual entropy. Additionally, the framework demonstrates its versatility by applying the same ConvLSTM predictor to both video compression and anomaly detection tasks. Performance evaluations show that this method achieves superior or competitive results compared to other state-of-the-art learning-based codec, offering a wide range of rate-distortion trade-offs and effectively reducing both spatial and temporal redundancies.

Adding to the foundational concepts, the paper by Chen et al. [7] provides the underlying methodology that has inspired subsequent developments in neural-based video compression. In this paper, the authors propose Neural Representations for Videos (NeRV), a novel neural representation for videos, which encodes videos into neural

networks, treating videos as neural networks and converting video encoding to model fitting and decoding as a simple feed forward operation. Unlike conventional representations that treat videos as frame sequences, NeRV represents videos as neural networks, taking frame index as input and outputting the corresponding Red Green Blue (RGB) image. This representation improves encoding speed by 25x to 70x, decoding speed by 38x to 132x, while achieving better video quality compared to pixel-wise implicit representations. NeRV allows for the conversion of video compression problems to model compression problems, enabling the use of standard model compression tools and achieving comparable performance with conventional video compression methods such as H.264 and HEVC. NeRV shows promising results in other tasks such as video denoising, outperforming traditional hand-crafted denoising algorithms and ConvNets-based methods.

Panneerselvam et al. [8] propose a novel video compression method using deep learning frameworks, specifically combining CNN and GAN, with an efficient workflow for frame-level compression. The method involves dividing video frames into different groups, using CNN to eliminate duplicate frames, and replacing them with single images by detecting minor changes through GAN, which are then recorded using LSTM. Instead of the entire image, only the small changes detected by GAN are substituted, significantly enhancing compression efficiency. This is complemented by pixel-wise comparisons using K-Nearest Neighbours (KNN), clustering via K-means, and applying Singular Value Decomposition (SVD) on each frame across the RGB color channels to reduce the utility matrix's dimensions by extracting latent factors. The processed video frames are then encoded into video format, and experiments demonstrated substantial size reductions with minimal quality loss, achieving around a 10% deviation in quality and more than a 50% reduction in size compared to the original video. The method, which also includes segmenting videos into key frames and leveraging deep learning networks for object detection, showed superior compression ratios and rate-distortion cost compared to existing Versatile Video Coding (VVC) methods

Another significant stride is presented by Gomes et al. [9] where the compression occurs not in pixel space but within the neural network parameter space. According to the paper, encoding videos as neural networks (known as Neural Video Representation (NVR)) offers innovative video processing capabilities, though traditional methods still outperform NVRs in video compression due to inefficiencies in compact spatio-temporal representation and disjoint rate-distortion optimization. To address these issues, they propose a novel convolutional architecture that effectively represents spatio-temporal information and employs a joint optimization strategy for rate and distortion. This end-to-end approach eliminates the need for post-training operations like quanti-

zation or pruning. Evaluations on the Ultra Video Group dataset show that their method achieves state-of-the-art results, surpassing both traditional HEVC benchmarks and established neural video compression methods.

Li et al.[10] propose enhancing context diversity in both temporal and spatial dimensions. Temporally, the model is guided to learn hierarchical quality patterns across frames, enriching long-term, high-quality temporal contexts. Additionally, group-based offset diversity is introduced to improve temporal context mining within an optical flow-based coding framework. Spatially, a quadtree-based partitioning method increases spatial context diversity for more effective entropy coding.To tap the potential of optical flow-based coding framework, a group-based offset diversity where the cross-group interaction for better context mining is proposed. Experiments demonstrate that this codec achieves a 23.5% bitrate reduction over previous state-of-the-art Neural Video Coder Decoder (NVC)s and surpasses the developing next-generation traditional codec Enhanced Compression Model (ECM) in both RGB and YUV420 colorspaces in terms of Peak Signal-to-Noise Ratio (PSNR).

While INRs have been successfully applied in image and 3D shape compression, their potential for audio compression remains largely unexplored. Lanzendörfer et al. [11] introduce Siamese Sinusoidal Representation Networks (SIREN), an INR model built on the SIREN architecture, which aims to achieve superior audio reconstruction fidelity with fewer network parameters compared to previous architectures. Siamese SIREN utilizes two twin extensions to the standard SIREN model to approximate the original audio signal, using the difference in noise between the two extensions for noise removal in the reconstruction process. The study evaluates the trade-off between compression speed, quality, and ratio, finding that longer training times can reduce noise presence but still leave it audibly detectable. To address this, the authors propose an approach for estimating and removing noise, achieving promising results in audio compression. The proposed Siamese SIREN network reduces parameter count by merging a subset of layers, allowing for effective noise removal without doubling the parameter count.

Zhang et al. [12] further explore the efficiency of neural networks in video compression by utilizing the NeRV technique, which represents each video frame as weights of a neural network. In this paper, the authors propose a universal boosting framework for implicit video representations, aiming to enhance representation capabilities and accelerate convergence speed. Their framework includes a conditional decoder with a temporal-aware affine transform module, a sinusoidal NeRV-like block for generating diverse intermediate features, and a high-frequency information-preserving reconstruction loss. They introduce a consistent entropy minimization scheme and develop video

codec based on the boosted representations. Experimental results on the Ultra Video Group dataset demonstrate significant improvements over baseline methods in tasks such as video regression, compression, inpainting, and interpolation. The authors validate the contribution of each component through extensive ablation studies, setting a new benchmark in the field of implicit video representation.

Sitzmann et al. [13] introduce Sinusoidal Representation Networks (SIREN) which utilize sinusoidal (sine) activation functions within the framework of INR to model discrete audio and video signals, leveraging the continuous and differentiable nature of sine functions for high-fidelity representations. For audio, SIRENs map temporal coordinates to amplitude values of waveforms, capturing cyclical patterns in audio data and allowing for continuous modeling over time, which is beneficial for generating synthetic speech or music. In video, SIRENs treat signals as functions of spatial and temporal coordinates, using sine functions to manage complexities such as rapid pixel intensity changes and motion, thus enabling continuous and differentiable representation that captures dynamics like motion blur and lighting variations. This continuous representation enhances the processing of derivatives, aiding in video tasks like motion estimation and compression, where detailed frame-by-frame analysis is essential.

While implicit neural representations INR have garnered significant attention for their capability to model complex signals like videos and audio through neural networks. Recent advancements have predominantly focused on video INRs, such as SIREN and NeRV, which use spatial and temporal coordinates or frame indices to efficiently generate video frames. However, audio representation via INR remains underexplored, with most existing methods relying solely on Multilayer Perceptron (MLP)s, leading to limitations in parameter efficiency and model expressibility. In response, Nerual Representations for Audios (NeRA) and Nerual Representations for Videos and Audios (NeRVA) frameworks [14] introduce novel approaches for timestamp-based audio and joint video-audio representations. NeRA employs a hybrid architecture combining MLPs and convolutional blocks, optimized for audio-specific characteristics. NeRVA extends this concept by enabling unified representation of multimedia content, utilizing a multi-branch architecture for audio and video modalities.

Quantization in deep learning, as detailed in the PyTorch documentation [15], is a technique that reduces the precision of numerical representations, such as using 8-bit integers instead of 32-bit floating-point numbers for weights and activations. This approach significantly improves model efficiency, reducing computation and storage requirements, particularly for resource-constrained environments like edge devices. PyTorch supports various quantization techniques, including static quantization, which precom-

putes scaling factors, dynamic quantization, applied at runtime for weight optimization, and quantization-aware training (QAT), which adapts models to lower precision during training. These methods help minimize the trade-off between reduced precision and accuracy loss. With user-friendly APIs, PyTorch simplifies quantization workflows for post-training or integration during model development. By enabling hardware compatibility and improved inference performance, quantization serves as a vital tool for deploying deep learning models efficiently in real-world scenarios.

The KD-INR framework [16] addresses the challenge of compressing large-scale time-varying data by combining knowledge distillation and implicit neural representations. Unlike traditional methods that require the entire dataset during training, KD-INR enables sequential learning of each time step through a two-stage process: spatial compression and model aggregation. Spatial compression uses bottleneck layers and features of interest preservation-based sampling to reduce data size while retaining crucial information, and model aggregation employs an offline knowledge distillation algorithm to merge multiple models into a single compact one. Evaluated against state-of-the-art methods like CoordNet, Neur Comp, SIREN, SZ3, ZFP, and TTHRESH, KD-INR demonstrates superior performance in terms of PSNR, LPIPS, and visual quality. Its ability to independently compress each time step makes it ideal for real-time applications such as in-place simulations, allowing for efficient compression and high-quality decompression during post-hoc analysis.

While significant progress has been made in the field of video and audio compression using implicit neural representations (INRs), existing methodologies have primarily treated video and audio as separate entities. Techniques like SIREN and Siamese SIREN have demonstrated remarkable potential for high-fidelity representations within their respective domains. However, a unified approach that simultaneously represents and compresses both video and audio signals within a single INR framework remains largely unexplored. Recent research, such as NeRVA, has successfully represented both audio and video within the same model framework. However, these methods adopt a frame-wise approach, combining MLPs and convolutional blocks for representation. In contrast, representing both video and audio on a pixel-wise basis using only MLPs has not yet been attempted. This project aims to address this gap by developing a comprehensive INR model capable of jointly representing and compressing multimedia content at a pixel-wise level using MLPs alone. Advanced compression techniques such as quantization, efficient encoding strategies, and knowledge distillation will be integrated. Knowledge distillation will enable the transfer of knowledge from a larger, more complex model to a smaller, optimized model, ensuring efficient compression without sacrificing representation quality. These enhancements are expected to improve

compression efficiency, reduce computational and storage requirements, and maintain high fidelity. This unified approach offers a transformative solution for processing and analyzing multimedia content in a resource-efficient manner.

3 FEASIBILITY ANALYSIS

The feasibility analysis for the project includes technical feasibility, economic feasibility, legal feasibility, and time constraints feasibility as discussed below.

3.1 Technical Feasibility

The required technologies for video compression, such as H.264, H.265/HEVC, VP9, AV1, and various encoding and decoding algorithms, are widely available. The project utilizes a neural network-based approach for video compression, thus the availability of computing powers through NVIDIA GTX 1650, NVIDIA RTX 4090 and platforms like Google Colab allows for sufficient training and inference testing.

3.2 Economical Feasibility

For the dataset, we have utilized a free dataset. For the training computing resources, we have access to an RTX 4090 GPU from our supervisor, ensuring that our project is economically feasible. This setup significantly reduces our operational costs and enhances the viability of our project from a financial perspective.

3.3 Legal Feasibility

Legal feasibility considerations include the use of dataset Vimeo-90K for training neural network models. While these datasets are widely used in research, it is important to note that specific datasets may have copyright restrictions or usage terms that require compliance. Proper attribution and adherence to licensing terms are essential to ensure legal compliance and avoid copyright infringement issues.

3.4 Schedule Feasibility

Effective project management practices, including task prioritization, milestone tracking, and regular progress evaluations, are implemented to ensure timely completion of project objectives. The utilization of project management tools and techniques, along with efficient resource allocation, facilitates meeting project deadlines and achieving desired outcomes within the specified time frame. The overview of our project schedule can be seen on Figure A-2.

4 THEORETICAL BACKGROUND

4.1 Introduction to Implicit Neural Networks (INNs)

Implicit Neural Networks (INN) represent a class of deep learning models designed to learn an implicit function that maps input coordinates to output features. INNs treat the weights of the network as data itself i.e. the data is represented within the weights of neural network. Unlike traditional neural networks that explicitly output values for each input in a dataset, INNs define a continuous multidimensional function for the entire data space. This function can be evaluated at any point in its domain, making INN especially suited for tasks involving high-dimensional data and requiring fine-grained control over outputs, such as in generating or reconstructing images, videos, and sounds. The basic concept of INN can be visualized in the Figure 4-1

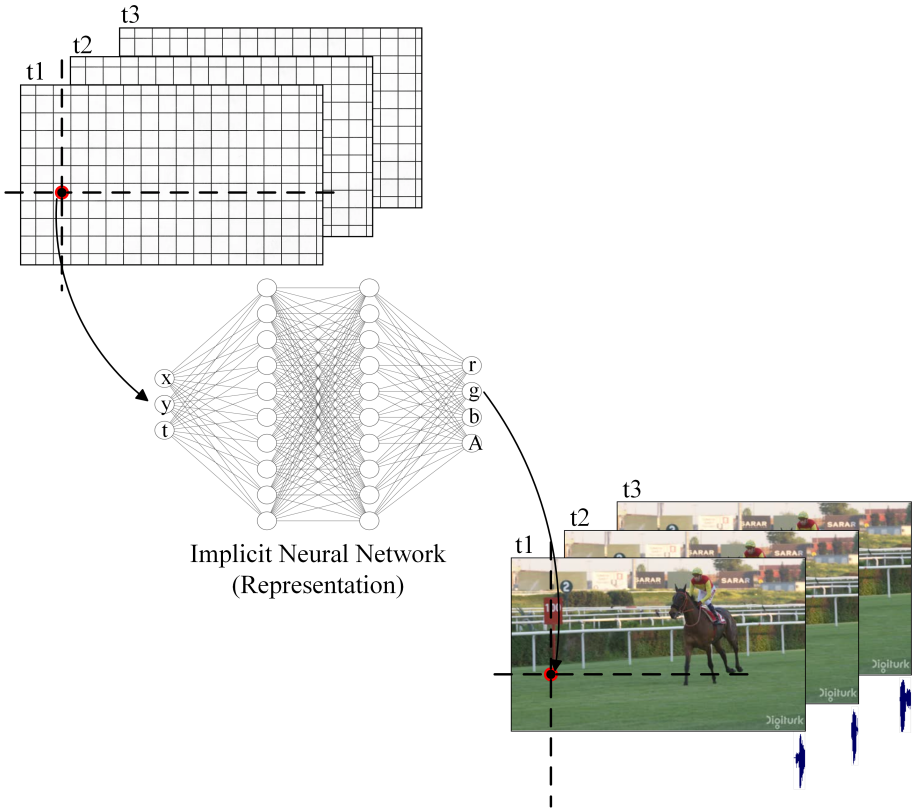


Figure 4-1: Representing Video in Neural Network

Figure 4-1 illustrates an Implicit Neural Representation (INR) where pixel coordinates (x, y) and time index t of video frames are fed into a neural network to output corresponding RGB values for visual data and amplitude for audio data. Sequential video frames $(t1, t2, t3)$ are divided into a grid of pixels. Each pixel's coordinates and time index are inputs to the network, which processes these inputs through several hidden layers. The network outputs the RGB values (r, g, b) for each pixel's color and the am-

plitude A for the associated audio. The outputs are then used to reconstruct the video frames and audio, demonstrating how the neural network encodes and decodes complex visual and audio information efficiently.

4.2 Fundamentals of INNs

INN parameterize an unknown function $f : \mathbb{R}^d \rightarrow \mathbb{R}^m$ where d and m represent the dimensions of the input and output spaces, respectively. This parameterization often involves a coordinate-based MLP where the input coordinates are fed directly into the network, and the output is the function value at those coordinates. The network is trained using a loss function that minimizes the difference between the predicted and true function values over a set of sampled points.

4.3 Differences in Implicit Neural Network over Traditional Neural Network

The Figure 4-2 provides a visual comparison between traditional neural networks and implicit neural networks like SIREN (Sinusoidal Representation Network). Traditional neural networks utilize a single set of shared weights and biases for all inputs, aiming to generalize across a dataset. This generalization allows the network to accurately classify or predict outcomes for new, unseen data based on its training. For instance, in the Figure 4-2, a traditional neural network trained on images of birds, reptiles, and mammals uses the same network architecture to classify any new image into these categories.

In contrast, implicit neural networks represent each piece of media with its own unique neural network, characterized by distinct weights and biases. This approach fundamentally changes the way these networks operate. Overfitting, typically seen as a drawback in traditional neural networks, becomes a beneficial feature in implicit neural networks. Overfitting allows the network to memorize and accurately reproduce the intricate details and nuances of the specific piece of media it represents. For example, in the image, each silent video and audio clip has its own dedicated neural network that predicts frame-wise RGB values per pixel or time-wise amplitude values, respectively.

The advantages of this approach are:

- **Detail Preservation:** The overfitting ensures that the network captures and reproduces fine-grained details of the media.
- **Per-Media Specialization:** Each piece of media is encoded into its unique network, allowing for highly specialized and optimized representations.

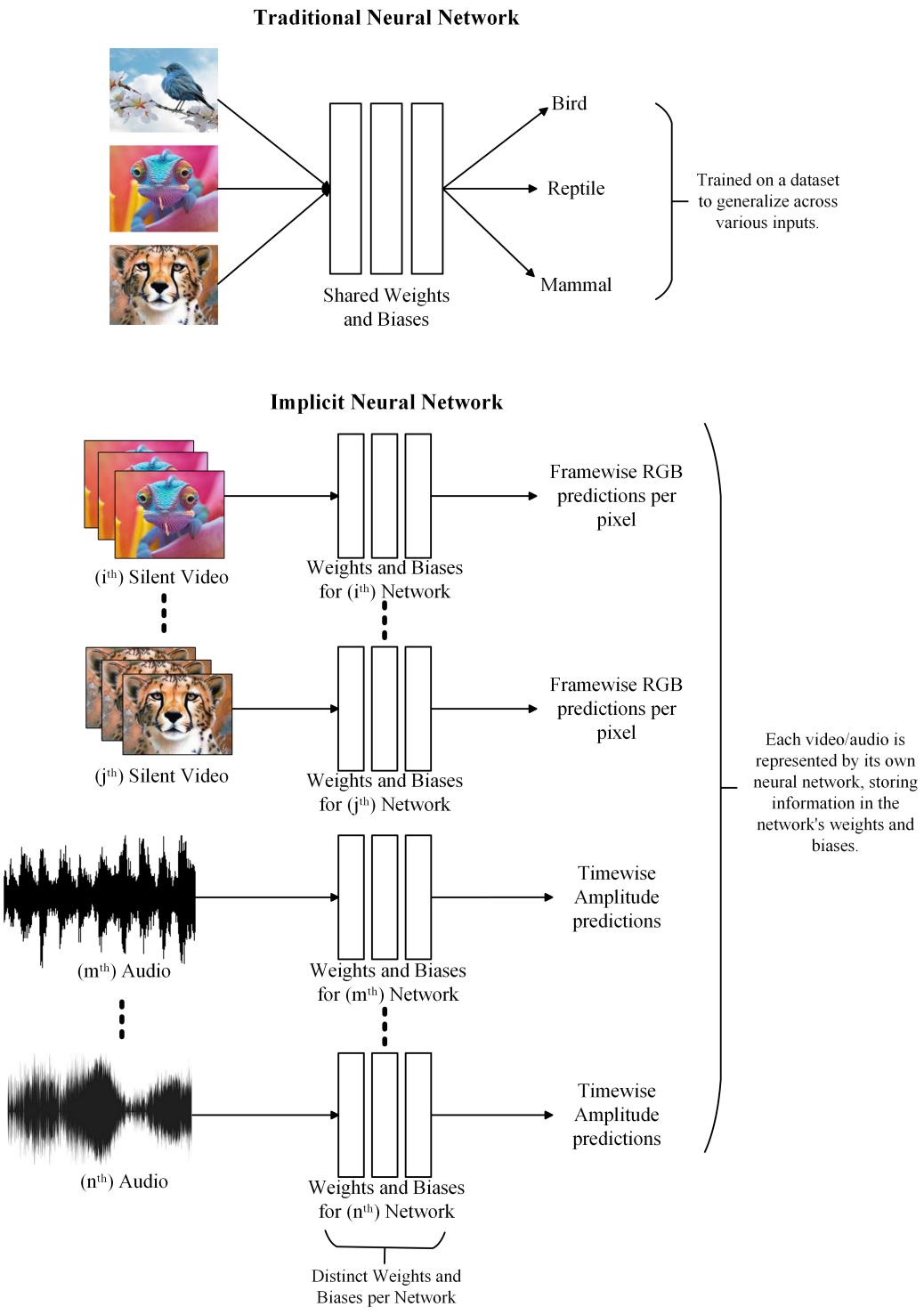


Figure 4-2: Implicit Neural Network vs Traditional Neural Network

Thus implicit neural networks like SIREN create unique networks for each media piece, leveraging overfitting to capture detailed representations. This allows implicit neural networks to store intricate details within their parameters, making overfitting a beneficial feature.

4.4 Significance of Sinusoidal Activations over Traditional Functions

The choice of activation functions in neural networks significantly influences their ability to model and process different types of data. The sine function is particularly suited for handling continuous inputs such as audio and video due to its inherent properties. Its smooth and periodic nature allows for efficient representation of continuous variations typical in audio waves or video frames. The smoothness of both the sine function and its derivative, a cosine wave, ensures that the gradients are well-defined across the entire function domain as shown in Figure 4-3. This characteristic is crucial during the neural network training process, which relies heavily on gradient-based optimization methods. The continuity and differentiability of the sine wave facilitate stable and effective weight updates, leading to reliable convergence in learning tasks.

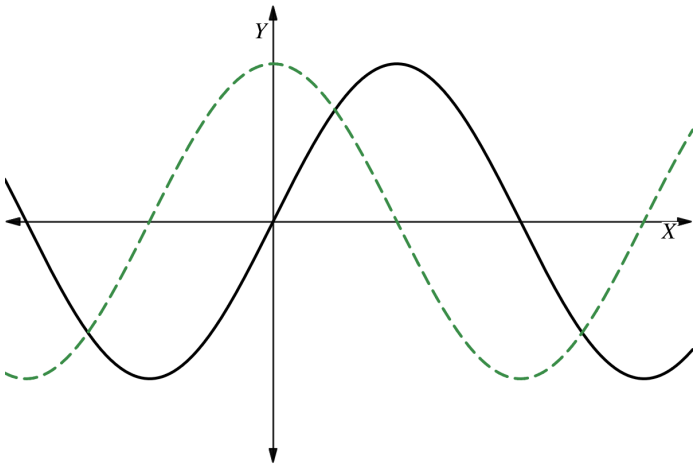


Figure 4-3: Sine Wave and its Derivative

In contrast, other waveforms like the triangular wave, although periodic, exhibit sharp transitions and corners at their peaks. The derivative of a triangular wave includes abrupt changes and discontinuities as we can see in Figure 4-4, which can introduce complications in the learning process, particularly at points where the gradient becomes undefined. These discontinuities can potentially lead to instability in gradient calculations during backpropagation, adversely affecting the training efficiency and model performance. For neural networks involved in processing continuous data, such as in speech recognition or video processing, the sine function's ability to smoothly interpolate between values can capture complex, nuanced patterns more effectively. This smooth interpolation helps in better generalization and reduces the risk of overfitting to noisy variations, making the sine function a preferred choice for activation in networks dealing with continuous input signals.

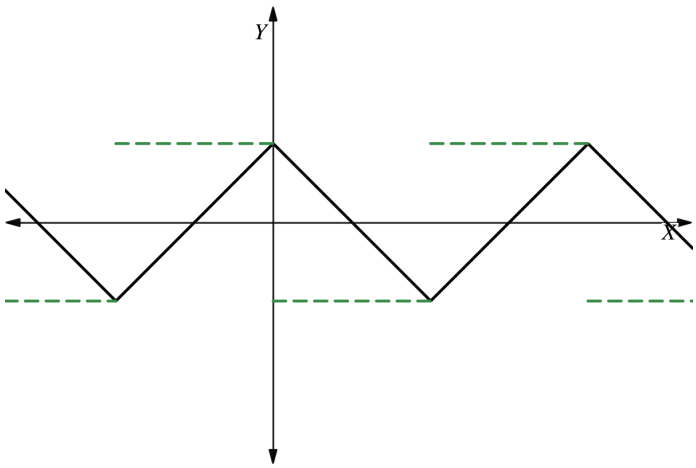


Figure 4-4: Triangular Wave and its Derivative

i. **Complex Signal Representation:** Sinusoidal activation functions enable neural networks to capture fine details and high-frequency components of signals more effectively. Traditional activation functions often struggle with this, leading to smoother, less detailed representations.

ii. **Smooth and Periodic Activation:** The smooth, continuous, and periodic nature of sinusoidal functions helps avoid issues like vanishing or exploding gradients, which are common with traditional activation functions, especially in deeper networks. This leads to more stable and efficient training dynamics.

iii. **Principled Initialization:** SIRENs benefit from principled initialization scheme tailored for sinusoidal activations, ensuring that the network starts with a configuration that is conducive to learning complex patterns without the instability seen with other activation functions.

4.5 Transition to Sinusoidal Representation Networks (SIRENs)

Building on the concept of INN, SIREN introduce a critical innovation by employing sinusoidal activation functions in all layers of the network. The general form of a SIREN layer is expressed as:

$$x_{i+1} = \sin(W_i x_i + b_i) \tag{4-1}$$

where x_{i+1} is the output of the i-th layer, serving as the input to the next layer, sin is the sinusoidal activation function applied to each layer, W_i is the weight matrix of the i-th layer, x_i is the input to the i-th layer, and b_i is the bias vector of the i-th layer.

4.6 Initialization Scheme for Sinusoidal Representation Networks (SIRENs)

Sinusoidal Representation Networks uses the sine function as an activation, which introduces sensitivity to input distributions due to its periodic nature. An effective initialization scheme is crucial to prevent the degradation of the network's performance with increasing depth. This report proposes a principled approach to initialization that preserves the distribution of activations across layers, ensuring that the output at initialization does not vary with the number of layers.

4.6.1 Steps Involved in Initialization

i. Initial Input and Single Neuron Output

- Uniformly Distributed Input: The input x is drawn from a uniform distribution $U(-1,1)$, representing normalized coordinates used in applications such as image processing.

- Output of a Single Sine Neuron: The output of a neuron using the sine activation function is given by:

$$y = \sin(ax + b) \tag{4-2}$$

  where a and b are the frequency and phase parameters, respectively. For $a > \frac{\pi}{2}$, ensuring at least half a period of the sine function, the output distribution y is arcsine distributed over $[-1,1]$.

ii. Layer-wise Propagation

- Weights and Biases: Weights w are initialized uniformly within $U\left(-\frac{c}{\sqrt{n}}, \frac{c}{\sqrt{n}}\right)$ where, c is a scaling factor and n is the fan-in.

- Output of Deeper Layers: In subsequent layers, each input is arcsine distributed due to the previous layer's sine activation. The weighted sum $w^T x$ approaches a normal distribution as the number of inputs n increases. Passing this sum through another sine function keeps the output arcsine distributed, preserving the distribution across layers.

iii. Special Handling in the First Layer

For the first layer, weights are adjusted by a factor ω_0, set to 30. This adjustment ensures that the sine function: $\sin(\omega_0 \cdot Wx + b)$ spans multiple periods over the interval [-1, 1]. This extensive coverage is beneficial for handling the complex patterns and frequencies in video and audio data, enabling the network to capture a broad range of features initially.

4.7 Siamese Siren

The Siamese Siren employs two neural networks with identical architectures but different weight initializations, referred to as the left and right Sirens. When an audio signal f is passed through these networks, they produce two slightly different outputs: $f1$ from the left Siren and $f2$ from the right Siren. These outputs, influenced by the slight variations in the networks, are then used to estimate and reduce noise.

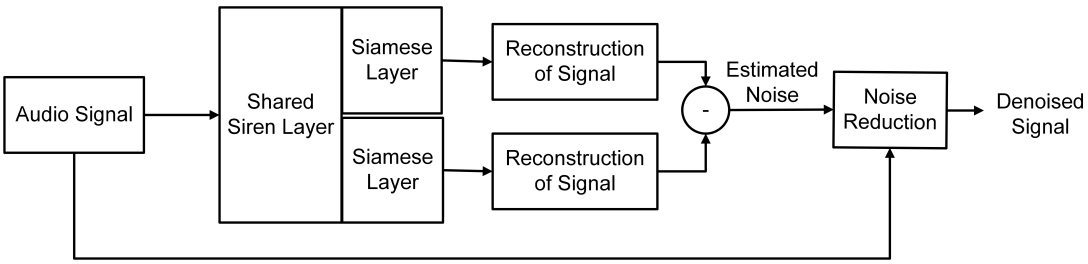


Figure 4-5: Siamese Siren Block Diagram

4.7.1 Noise Estimation in Siamese Siren

Siamese Siren estimates noise by calculating the difference between the outputs of the two networks:

$$\text{Noise} = f1 - f2 \tag{4-3}$$

where $f1$ is the inferenced audio from one network and $f2$ is the inferenced audio from the other network.

4.7.2 Noise Reduction Process

To reduce noise, the Siamese Siren leverages a noise reduction algorithm. This algorithm, often implemented using specialized noise reduction libraries, takes two inputs:

i. A noise estimation derived from the noisy audio.

ii. The noisy signal itself.

4.8 Knowledge Distillation for Model Compression

Knowledge distillation is a technique used in machine learning to transfer knowledge from a larger, more complex model (teacher) to a smaller, simpler model (student). This process aims to improve the efficiency and speed of the smaller model while maintaining or even enhancing its performance. One variant of knowledge distillation is Response-Based Knowledge Distillation (RBKD), which focuses on distilling knowledge related to the model's response or output behavior.

The fundamental idea behind RBKD is to train a student model to mimic not just the final predictions of the teacher model but also the intermediate responses or activations. This approach aims to capture the nuanced decision-making processes of the teacher model, enabling the student to generalize better and achieve comparable performance with fewer parameters.

The main idea behind RBKD revolves around the concept of feature representations and decision boundaries. By learning from the teacher model's response patterns, the student model can develop a deeper understanding of the underlying data distribution and make more informed predictions, especially in complex and high-dimensional input spaces.

Soft Target Loss: The soft target loss remains the primary component in the distillation process. It compares the probabilities predicted by the student model to those of the teacher model:

$$L_{\text{soft}} = \frac{1}{N} \sum_{i=1}^{N} (S(x) - T(x))^2 \tag{4-4}$$

where L_{soft} is the soft target loss, $S(x)$ is the student model's output, and $T(x)$ is the teacher model's output for input x.

Hard Target Loss: The hard target loss is an another component in response-based knowledge distillation. It compares the student model's predictions to the ground truth labels, ensuring that the student learns to represent the actual data accurately. The hard target loss is defined as:

$$L_{\text{hard}} = \frac{1}{N} \sum_{i=1}^{N} (S(x) - y)^2 \tag{4-5}$$

where L_{hard} is the soft target loss, $S(x)$ is the student model's output and y is the ground truth label.

The overall loss function simplifies to just the soft target loss without the response matching component:

$$L_{\text{total}} = \alpha L_{\text{soft}} + (1 - \alpha) L_{\text{hard}} \tag{4-6}$$

where L_{total} is the total loss function, α is a hyperparameter controlling the importance of the soft target loss and hard target loss.

In Figure 4-6, the teacher model is a SIREN (Sinusoidal Representation Networks)

network pretrained on audio-visual data, while the student model is a shallower version designed to learn from the teacher's responses.

The inputs to both models are in the form of (x, y, t), where (x, y) represent spatial coordinates in the video, and t represents time. The teacher model outputs four values: r, g, b (RGB values corresponding to (x, y, t)), and Amplitude (corresponding audio amplitude for the given (x, y, t)). Notably, the (x, y) coordinates are ignored in the context of audio amplitude.

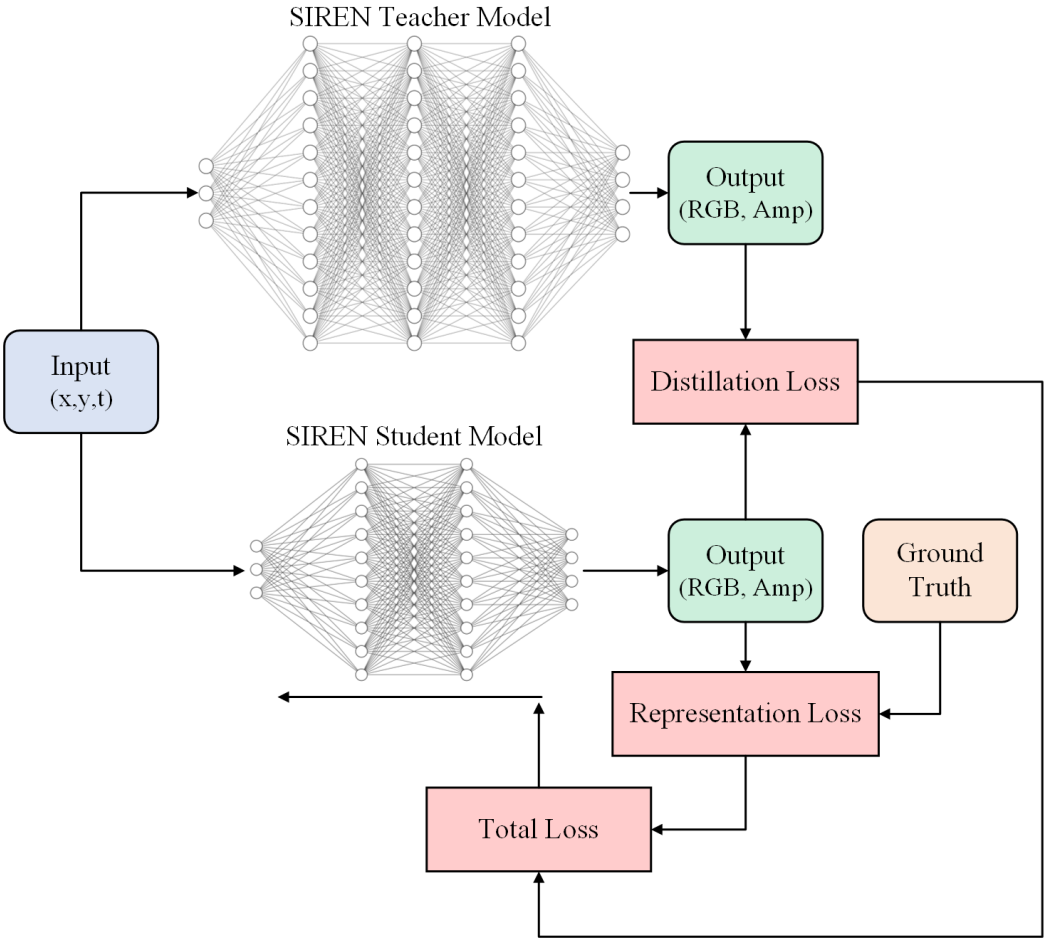


Figure 4-6: Block Diagram of Knowledge Distillation

During response-based knowledge distillation, the student model's predictions are compared against the teacher model's predictions. This comparison helps calculate the distillation loss, which captures the discrepancy between the student's output and the teacher's output. Simultaneously, the ground truth provides the representation loss, measuring how well the student model represents the actual data.

These two types of losses, distillation-loss and representation-loss, are weighted and

combined to calculate the total loss. This total loss is then used for feedback and back-propagation during the training process, allowing the student model to gradually learn and mimic the behavior of the more complex teacher model.

By using response-based knowledge distillation, the student model can benefit from the rich representations learned by the teacher model, even though it has a simpler architecture. This approach is particularly useful for tasks where computational resources or model complexity are constrained, yet high performance is desired.

4.9 Quantization

Quantization involves reducing the precision of the model's parameters, such as weights and biases, from floating-point to lower-bit representations, which decrease the model size.

$$w' = \text{round}\left(\frac{w}{s} + z\right) \tag{4-7}$$

where, w is the original weight, s is a scaling factor, z is zero-point, and w' is the quantized weight.

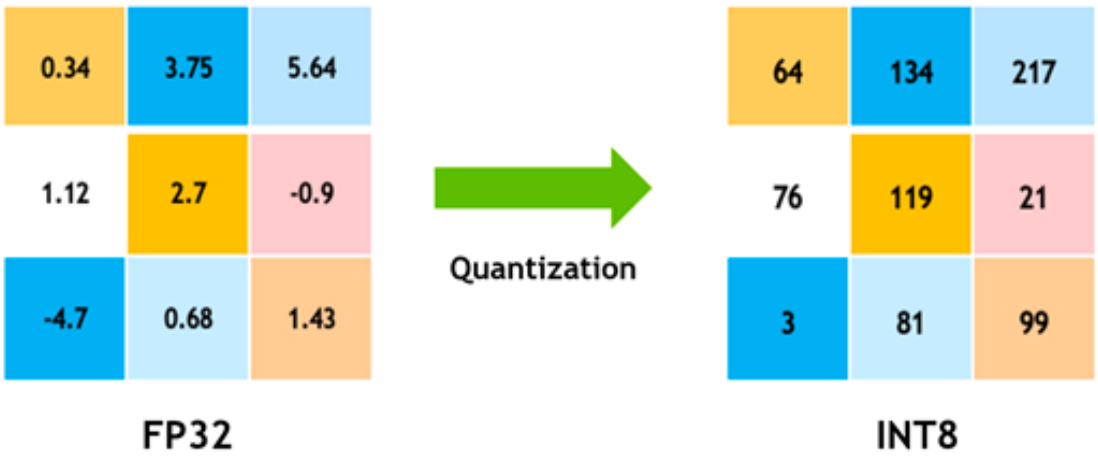


Figure 4-7: Quantization

In Figure 4-7, a sample 3x3 matrix of weights is taken. The weights are quantized to int 8, reducing the precision from 32-bit floating-point. This process reduces the model size and computational requirements, making it more efficient on resource-constrained devices.

Since we will be using symmetric quantization for weights and biases of our model, we will discuss the symmetric quanzation below, however asymmetric quantization can also be done as shown in subsection Appendix F:.

4.9.1 Symmetric Quantization

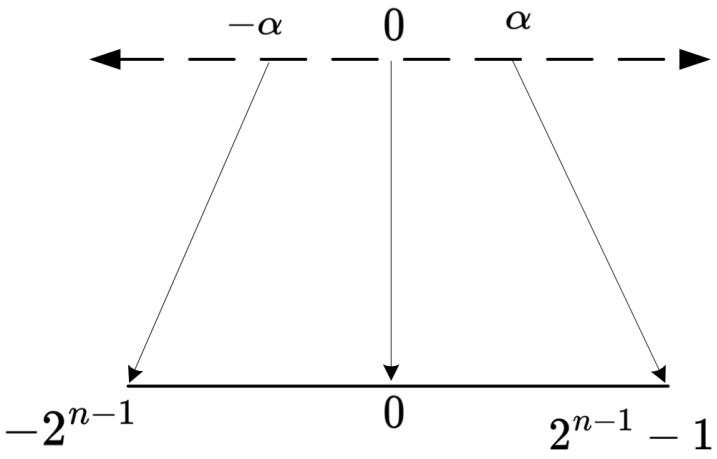


Figure 4-8: Symmetric Quantization

In symmetric quantization, the range of the values represented is symmetric around zero. This means the quantized values are distributed equally between positive and negative numbers.

Characteristics:

i. Zero point is fixed at zero.

ii. Only scale factor is used to calculate quantized value.

4.10 Encoding

Encoding in the context of data compression refers to the process of converting raw data into a more compact representation, using techniques that minimize the amount of storage or transmission required. The goal of encoding is to reduce redundancy and increase efficiency, which can lead to significant reductions in the size of the data, making it easier to store and transfer.

Two important methods for encoding data are Arithmetic Encoding and Range Encoding, which are widely used in various compression algorithms.

4.10.1 Arithmetic Encoding

Arithmetic Encoding is a form of entropy encoding that represents a message as a single number between 0 and 1. The key idea behind arithmetic encoding is to iteratively refine an interval to represent the entire message. Instead of encoding each symbol separately,

arithmetic encoding encodes the entire message as a fraction in a range [0, 1].

- The process begins by initializing an interval [low, high], which initially spans the range [0, 1].
- For each symbol in the message, the interval is divided into subintervals based on the probabilities of the symbols. The interval is then updated based on the symbol's probability.
- The final encoded message is represented by a value within the resulting subinterval that corresponds to the entire message.

This method efficiently encodes data with different symbol frequencies and is highly effective in cases where symbols have different probabilities.

4.10.2 Range Encoding

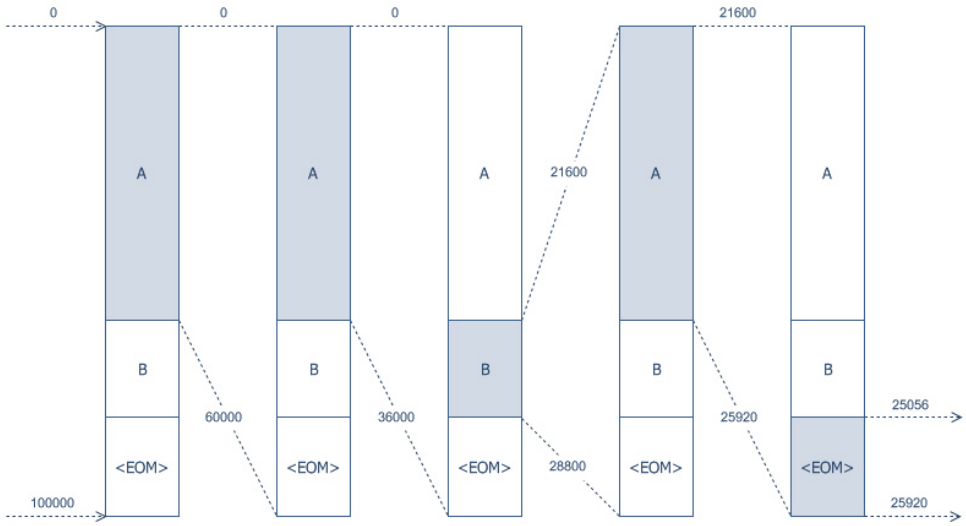


Figure 4-9: Range Encoding

Range Encoding as shown in Figure 4-9 is similar to Arithmetic Encoding in that it also represents the entire message as a number within a range. However, Range Encoding uses a slightly different approach to dividing the range and updating it.

- Like arithmetic encoding, Range Encoding divides the range into subranges based on the probabilities of the symbols.
- However, Range Encoding works with fixed-size integers for each step, which makes it easier to implement and optimize.

- As each symbol is processed, the range is updated, and a final value within the range represents the compressed message.

While Range Encoding is simpler and faster than Arithmetic Encoding in practice, it can be less efficient when dealing with very large data sets or extremely small probabilities. Despite this, it remains popular in modern compression algorithms.

4.10.3 LZMA2 Algorithm

The Lempel-Ziv-Markov chain algorithm (LZMA)2 algorithm is an enhanced successor to the original LZMA (Lempel-Ziv-Markov chain Algorithm). While retaining the core principles of LZMA, LZMA2 addresses several limitations and introduces critical improvements, particularly for modern hardware and large datasets. It is the algorithm used in the xz compression format and is optimized for both compression ratio and performance.

LZMA2 combines dictionary-based compression with range encoding, similar to LZMA, but adds the following key features:

- **Block-Based Processing:** LZMA2 divides input data into independently compressible blocks. This enables parallel processing, allowing multi-core systems to compress or decompress multiple blocks simultaneously, significantly improving speed.
- **Adaptive Dictionary Management:** Unlike LZMA, which uses a single continuous dictionary, LZMA2 resets or updates the dictionary between blocks. This prevents the dictionary from becoming overly large or inefficient when processing heterogeneous data.
- **Handling of Incompressible Data:** LZMA2 detects incompressible regions and stores them uncompressed, avoiding unnecessary computational overhead. This contrasts with LZMA, which would still attempt to compress such data.
- **Streamable Processing:** LZMA2 supports seamless compression of continuous data streams without requiring full data availability upfront, making it suitable for real-time applications.

The algorithm operates as follows:

i. **Initialization:**
   - Configure block size and thread count for parallel processing.

- Initialize a range encoder and allocate dictionaries for each block.

ii. **Block Processing:**
- Split the input data into blocks (if parallelization is enabled).
- For each block, apply dictionary compression:
  - Replace repeated sequences with references to the block-specific dictionary.
  - Add new symbols to the dictionary dynamically.

iii. **Range Encoding:**
- Encode the compressed blocks using range encoding, adjusting probabilities based on symbol frequency.
- For incompressible blocks, encode raw data directly.

iv. **Output Finalization:**
- Combine encoded blocks into a single compressed stream.
- Add metadata (e.g., block boundaries, dictionary reset flags).

Compared to LZMA, LZMA2 provides:

- **Scalability:** Efficient utilization of multi-core CPUs via parallel block compression.
- **Flexibility:** Adaptive handling of mixed compressible/incompressible data.
- **Resource Efficiency:** Reduced memory overhead through dictionary resets.

4.10.4 xz

The xz compression format is the primary implementation of LZMA2. It leverages the algorithm's block-based design and parallelization capabilities to achieve high compression ratios while maintaining competitive speeds on modern hardware. xz is widely adopted in Linux distributions for software distribution and archival purposes.

Key features of xz include:

- **Multi-Threaded Compression:** By default, xz uses LZMA2's block partitioning to distribute workloads across CPU cores.
- **Configurable Block Sizes:** Users can optimize for speed or ratio by adjusting

block sizes.

- **Backward Compatibility:** xz supports legacy LZMA for decompression, ensuring interoperability with older systems.

xz's integration of LZMA2 makes it a versatile tool for scenarios demanding both high compression efficiency and performance, such as large-scale data backups or software package distribution.

4.11 MPEG Audio Layer 3

MPEG Audio Layer 3, commonly known as MP3, is a widely used audio compression standard that reduces file size while preserving perceptual audio quality. The encoding process involves several stages that progressively transform and compress the digital audio signal into a highly efficient format.

4.11.1 MPEG Audio Layer 3 Encoder

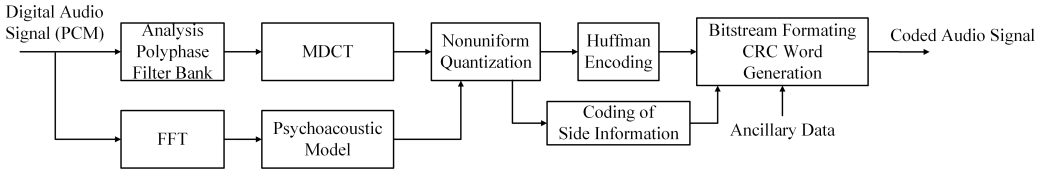


Figure 4-10: MPEG Audio Layer 3 Encoder

i. **Digital Audio Signal (PCM):** Pulse Code Modulation (PCM) represents the raw, uncompressed digital audio signal. It consists of sampled values of the original sound waveform, typically represented as 16-bit signed integers per sample. The input to the MP3 encoder is PCM audio, commonly sampled at rates like 44.1 kHz (CD quality) or 48 kHz.

ii. **Analysis Polyphase Filter Bank:** This stage splits the audio signal into subbands or frequency bands using a polyphase filter bank. Dividing the signal into smaller frequency channels simplifies further processing and enables selective compression.

iii. **FFT (Fast Fourier Transform):** The Fast Fourier Transform (FFT) converts the time-domain audio signal into the frequency domain. This transformation identifies the energy in different frequency components (sine waves) of the audio, providing insight into the spectral content.

iv. **Psychoacoustic Model:** The psychoacoustic model applies principles of human hearing to discard inaudible data.

- Frequencies above 20 kHz or below the threshold of hearing are removed.
- Masking effects eliminate inaudible sounds masked by louder frequencies.
- Loudness analysis determines the importance of each frequency band, prioritizing perceptually significant data.

v. **MDCT (Modified Discrete Cosine Transform):** Modified Discrete Cosine Transform (MDCT) further processes the frequency data by converting overlapping audio blocks into the frequency domain. It reduces blocking artifacts and provides better time-frequency resolution. This step ensures accurate reconstruction during decoding.

vi. **Nonuniform Quantization:** The MDCT coefficients are quantized to compress the audio further:
- Frequencies less perceptible to the human ear are quantized with less precision.
- This step introduces bit-rate control, selectively reducing precision to minimize file size while maintaining quality.

vii. **Huffman Encoding:** A lossless compression technique is applied to the quantized data. Huffman encoding assigns shorter codes to frequently occurring values, optimizing the overall file size.

viii. **Bitstream Formatting and CRC Word Generation:** The encoded audio data is organized into a structured bitstream. Cyclic Redundancy Check (CRC) data is added to verify data integrity during transmission.

ix. **Coding of Side Information:** Side information includes metadata required for decoding:
- Sampling rate
- Bitrate
- Frame length
- Quantization details

This ensures accurate reconstruction of the audio.

x. **Ancillary Data:** Optional ancillary data, such as metadata (ID3 tags), album art, or custom data, can be embedded in the MP3 file. Although not required for decoding, it adds extended functionality.

4.11.2 MPEG Audio Layer 3 Decoder

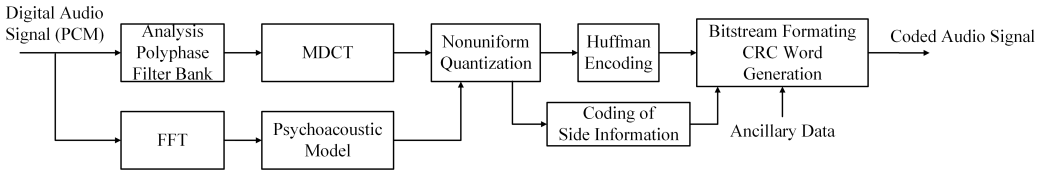


Figure 4-11: MPEG Audio Layer 3 Decoder

i. **Bitstream:** The bitstream is the compressed MP3 file produced during encoding. It contains the encoded audio data along with side information necessary for decoding.

ii. **Synchronization and Error Checking:** This block resolves synchronization issues in the bitstream and verifies its integrity using techniques like CRC (Cyclic Redundancy Check).

iii. **Huffman Decoding:** This step reverses the Huffman encoding applied during compression, recovering the quantized audio data.

iv. **Scalefactor Decoding:** The scalefactor values embedded in the bitstream are decoded. These values are crucial for restoring the magnitude of the frequency-domain coefficients.

v. **Descaling:** The quantized values are rescaled to their proper range to ensure accurate reconstruction of the audio data.

vi. **Reordering:** The data reordered during encoding for compression efficiency is restored to its original order.

vii. **Joint Stereo Decoding:** If joint stereo techniques were used in encoding, the stereo channels are reconstructed. This step restores the correlation between the left and right channels.

viii. **Alias Reduction:** Alias reduction eliminates aliasing artifacts caused during the encoding process, ensuring clean audio output.

ix. **IMDCT (Inverse Modified Discrete Cosine Transform):** The Inverse Modified Discrete Cosine Transform (IMDCT) step reverses the MDCT transformation applied during encoding, converting frequency-domain data back into the time domain.

x. **Frequency Inversion:** Any frequency inversion applied during encoding is reversed in this step.

xi. Synthesis Polyphase Filter Bank: The Synthesis Polyphase Filter Bank reconstructs the original time-domain signal from the time-frequency data. It operates as the inverse of the Analysis Polyphase Filter Bank used during encoding.

xii. PCM Output: The final output is in Pulse Code Modulation PCM format, an uncompressed audio representation ready for playback. Separate left and right channels are produced for stereo audio.

4.12 Advanced Video Codec (AVC)

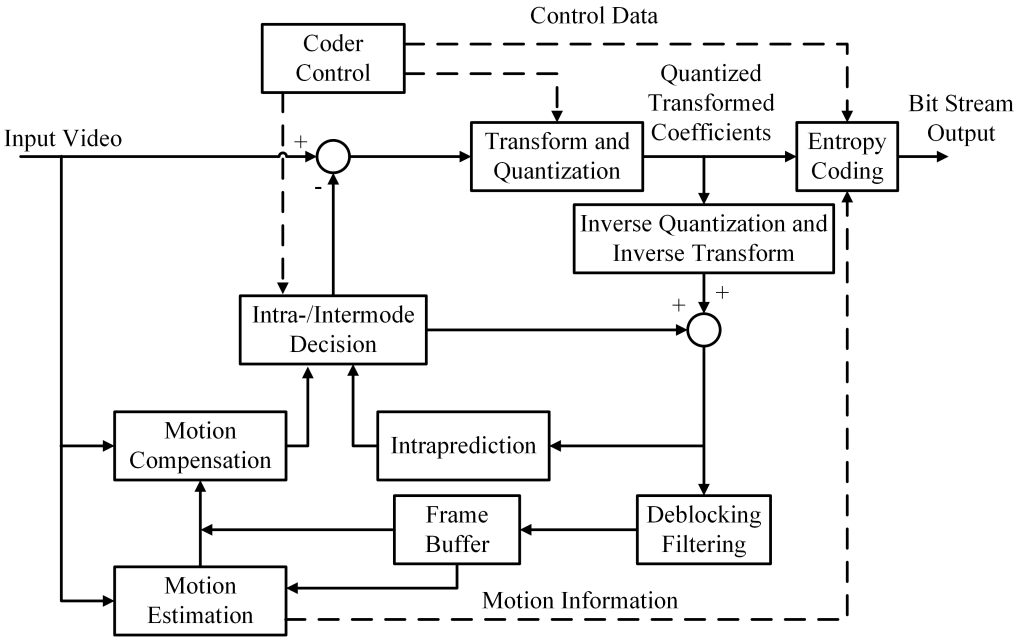


Figure 4-12: Block Diagram of Advanced Video Codec

Figure 4-12 outlines the encoding process for the Advanced Video Codec. The process involves a series of steps, starting with the input video and proceeding through various prediction, transformation, and compression techniques to produce a highly efficient encoded bitstream.

i. Input Video: The process begins with an input video consisting of a sequence of frames, each composed of three components: luminance (Y) and chrominance (Cb and Cr). The luminance provides brightness information, while chrominance adds color details. Frames are divided into macroblocks of 16×16 pixels for Y and 8×8 pixels for Cb and Cr.

ii. Intra-/Intermode Decision: This step determines how each macroblock will be encoded.

- **Intra-mode:** Macroblocks are predicted based on previously encoded blocks

within the same frame. This mode is effective for spatial redundancy.

- **Inter-mode:** Macroblocks are predicted using reference frames from the past or future. This mode is effective for temporal redundancy.

iii. **Motion Estimation:** When inter-mode is selected, motion estimation determines object movement relative to reference frames.
   - Macroblocks can be divided into smaller partitions (e.g., 16×16, 8×8, 4×4).
   - Motion vectors with sub-pixel precision (e.g., quarter-pixel for luma) are calculated.
   - A search algorithm minimizes differences between current and reference macroblocks.

iv. **Motion Compensation:** Using motion vectors, motion compensation reconstructs predicted macroblocks.
   - Fractional-pixel values are interpolated using filters like six-tap FIR filters.
   - Multiple reference frames enhance prediction accuracy.

v. **Intraprediction:** When intra-mode is selected, predictions rely on spatial redundancy:
   - Macroblocks are predicted using neighboring reconstructed macroblocks.
   - Prediction modes include:
     - 4×4 blocks: Nine directional modes.
     - 8×8 blocks: Nine modes.
     - 16×16 macroblocks: Four modes.

vi. **Residual Calculation:** The residual is calculated by subtracting the predicted macroblock from the actual macroblock, representing the difference or error for compression.

vii. **Transform and Quantization:** *Transformation:* Converts residuals into the frequency domain using an integer transform derived from the DCT, reducing spatial redundancy.
   Quantization: Reduces the precision of transformed coefficients:
   - Low-frequency components are preserved for quality.

- High-frequency components are aggressively compressed.

viii. **Entropy Coding:** Encodes quantized coefficients, motion vectors, and control data into a compressed bitstream.
   - **CAVLC:** Context-Adaptive Variable Length Coding.
   - **CABAC:** Context-Adaptive Binary Arithmetic Coding.

ix. **Deblocking Filtering:** Mitigates blocking artifacts introduced by quantization.
   - Adaptive filtering smooths block boundaries.
   - Strength is adjusted based on slice, block edge, and pixel levels.

x. **Inverse Transform and Inverse Quantization:** Quantized coefficients are inverse-quantized and transformed back to the spatial domain. Reconstructed residuals are added to predictions to recreate macroblocks.

xi. **Frame Buffer:** Stores reference frames.
   - Short-term frames for recent predictions.
   - Long-term frames for scalability and large temporal gaps.

xii. **Output Bit Stream:** Produces a final encoded bitstream containing motion vectors, quantized residual coefficients, prediction modes, and control information.

4.13 High Efficiency Video Coding (HEVC)

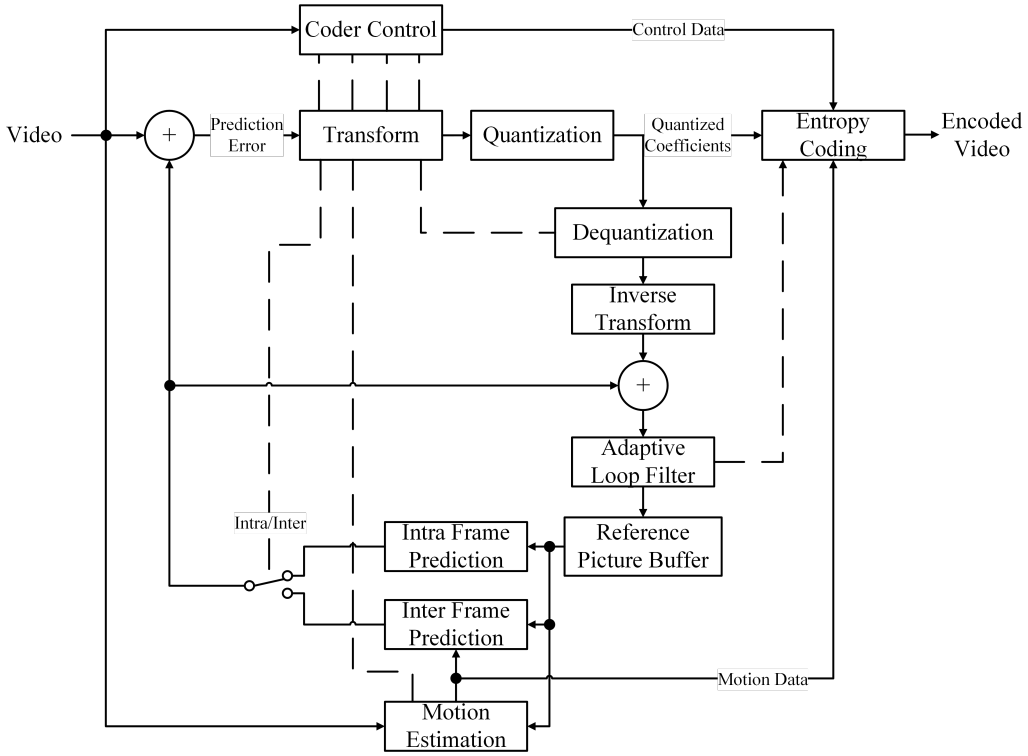


Figure 4-13: Block Diagram of High Efficiency Video Coding

The block diagram of HEVC shares a similar high-level structure with AVC, reflecting its hybrid coding framework based on intra-frame and inter-frame prediction, transform, quantization, and entropy coding. However, HEVC introduces several key advancements to improve compression efficiency, particularly for high-resolution content. Additionally, some blocks in HEVC feature entirely new functionalities.

i. **Input Video:** HEVC processes input video frames divided into luminance (Y) and chrominance (Cb, Cr) components, similar to H.264. However, HEVC replaces macroblocks with flexible Coding Tree Units (CTUs), which can be as large as 64×64 and recursively partitioned into smaller Coding Units (CUs), down to 8×8. This flexibility reduces overhead for high-resolution content.

ii. **Intra-/Intermode Decision:** HEVC decides between intra-prediction (within a frame) and inter-prediction (between frames) based on the redundancy present. It offers 35 intra-prediction modes for luma and 6 for chroma, significantly improving texture handling compared to H.264's limited modes. Inter-prediction is enhanced with asymmetric motion partitions (AMPs) and motion merge mode.

iii. **Motion Estimation and Compensation:** HEVC uses refined sub-pixel precision

for motion estimation, improving upon H.264's quarter-pixel accuracy. Sophisticated search strategies optimize motion vectors and reduce complexity. Motion merge mode reuses parameters to compress the bitstream further, while multi-reference frame compensation improves prediction accuracy.

iv. **Transform and Quantization:** HEVC supports larger transform block sizes (up to 32×32) compared to H.264's maximum of 8×8, enhancing compression for smooth areas. It introduces the Discrete Sine Transform (DST) for intra-predicted blocks and uses improved quantization techniques, simplifying the decoding process.

v. **Entropy Coding:** HEVC exclusively employs Context-Adaptive Binary Arithmetic Coding (CABAC), optimizing it for larger block sizes and parallel processing, unlike H.264, which also used the less efficient Context-Adaptive Variable Length Coding (CAVLC).

vi. **Deblocking Filter:** HEVC improves deblocking decisions by considering multiple block boundary types (e.g., CUs, PUs, TUs). This provides better artifact suppression, especially for high-resolution content.

vii. **Adaptive Loop Filtering (ALF):** HEVC applies adaptive filters optimized for each frame's content, reducing distortions introduced during encoding. This ensures better visual quality by minimizing blocking and blurring artifacts.

viii. **Reference Picture Buffer:** HEVC stores reference frames in an optimized buffer, supporting larger coding tree units and parallel processing methods like wavefront parallel processing.

ix. **Coder Control:** HEVC integrates advanced coder control to manage partitioning, mode selection, and bit allocation. It supports parallel encoding strategies to maximize multi-core processor efficiency.

5 MATHEMATICAL FORMULATION

5.1 Mathematical Modeling in SIRENs

SIREN excel in tasks requiring the modeling of periodic and smooth phenomena, which are common in audio and video signals. The continuous nature of the function modeled by SIREN makes them ideal for compressing these signals efficiently. The typical loss function used for training SIREN in the context of compression is the Mean Squared Error (MSE), which ensures the fidelity of the reconstructed signal:

$$L = \frac{1}{N}\sum_{n=1}^{N}(\Phi(\mathbf{x}_n) - y_n)^2 \tag{5-1}$$

Additionally, to capture temporal dynamics in video and audio, derivatives of the function are often included in the training objective:

$$L = \frac{1}{N}\sum_{n=1}^{N}\left((\Phi(\mathbf{x}_n) - y_n)^2 + \lambda(\nabla\Phi(\mathbf{x}_n) - \nabla y_n)^2\right) \tag{5-2}$$

where, N is the total number of sampled points, Φ is the SIREN model, $\mathbf{x}_n$ are the input coordinates for the n-th data point, y_n are the actual output values at the n-th data point, λ is a regularization parameter, $\nabla\Phi(\mathbf{x}_n)$ is the derivative of the SIREN output with respect to the input at $\mathbf{x}_n$, and ∇y_n is the derivative of the true output values with respect to the input at $\mathbf{x}_n$.

5.2 Forward and Backward Propagation in SIREN Network

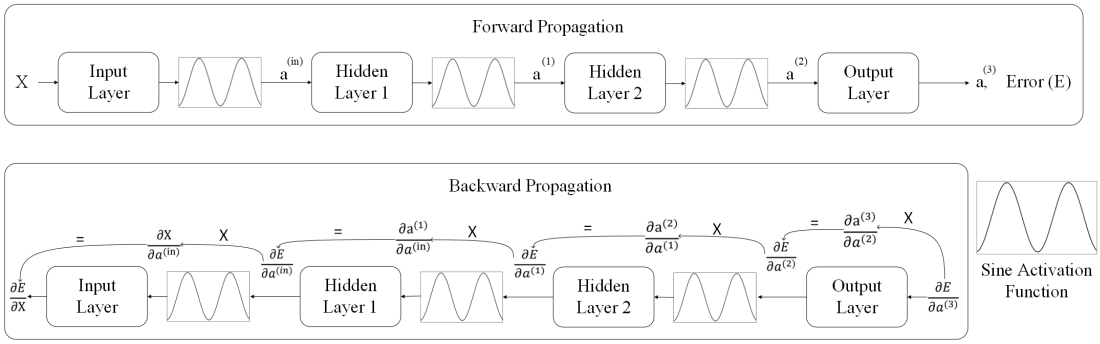


Figure 5-1: Forward and Backward Propagation in SIREN Network

The figure provides an illustration of the forward and backward propagation processes in a SIREN neural network. In the forward propagation section, the input **X** is passed through the input layer, then sine activation is applied, and then sequentially through hidden layers 1 (with sine activation) and hidden layer 2 (with sine activation), and

finally through the output layer. The forward propagation process can be described by the equations:

$$\mathbf{Z}^{(l)} = \mathbf{W}^{(l)T}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{5-3}$$

$$\mathbf{a}^{(l)} = \sin(\mathbf{Z}^{(l)}) \tag{5-4}$$

where,

- $\mathbf{Z}^{(l)}$ is the linear combination of inputs at layer l.
- $\mathbf{W}^{(l)}$ represents the weights of the l-th layer.
- $\mathbf{a}^{(l-1)}$ is the activation from the previous layer.
- $\mathbf{b}^{(l)}$ represents the biases of the l-th layer.
- $\mathbf{a}^{(l)}$ is the activation of the l-th layer after applying the sine function.

The error $\mathbf{E}$ is calculated based on the difference between the predicted output $\mathbf{a}^{(3)}$ and the actual output $\mathbf{y}_i$. Specifically, we use the Mean Squared Error (MSE) loss function:

$$\mathbf{E} = \frac{1}{2}\sum_{i=1}^{N}(\mathbf{a}_i^{(3)} - \mathbf{y}_i)^2 \tag{5-5}$$

where,

- $\mathbf{a}_i^{(3)}$ is the predicted output.
- $\mathbf{y}_i$ is the actual output.

The backward propagation section shows the gradient calculation needed to update the network's weights. It starts with the error gradient $\frac{\partial \mathbf{E}}{\partial \mathbf{a}^{(3)}}$ at the output layer and propagates backwards through hidden layers 2 and 1, and finally to the input layer. This involves computing the gradients $\frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{a}^{(2)}}$, $\frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{a}^{(1)}}$, and $\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{a}_{(in)}}$.

The weights and biases are updated as per the following equations:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta\frac{\partial \mathbf{E}}{\partial \mathbf{W}^{(l)}} \tag{5-6}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta\frac{\partial \mathbf{E}}{\partial \mathbf{b}^{(l)}} \tag{5-7}$$

where,

- $\mathbf{W}^{(l)}$ represents the weights of the l-th layer.
- $\mathbf{b}^{(l)}$ represents the biases of the l-th layer.
- η is the learning rate.
- $\frac{\partial \mathbf{E}}{\partial \mathbf{W}^{(l)}}$ is the gradient of the error with respect to the weights of the l-th layer.
- $\frac{\partial \mathbf{E}}{\partial \mathbf{b}^{(l)}}$ is the gradient of the error with respect to the biases of the l-th layer.

The detailed forward and backward propagation equations and the gradients of error with respect to weight and bias matrices are provided in subsection Appendix B:.

5.3 Quantization Modeling

Quantization maps a floating-point weight w to a discrete integer w':

Then the dequantized value can be calculated as:

$$\text{dequantized value} = \text{quantized value} \cdot \text{scale} \tag{5-8}$$

$$w' = \text{round}\left(\frac{w}{s} + z\right) \tag{5-9}$$

where:

- $w \in \mathbb{R}$ is the original floating-point weight.
- $w' \in \mathbb{Z}$ is the quantized integer weight.
- $s \in \mathbb{R}^+$ is the scaling factor (step size).
- $z \in \mathbb{R}$ is the zero-point, which may be a floating-point or integer value.

5.3.1 Dequantization

The dequantized value $\hat{w}$ approximates the original floating-point value:

$$\hat{w} = s \cdot \left(w' - z\right) \tag{5-10}$$

5.3.2 Calculation of Scale and Zero-Point

The scale s and zero-point z are determined based on the range of the floating-point weights $\left[w_{\min}, w_{\max}\right]$ and the quantized range $\left[q_{\min}, q_{\max}\right]$:

$$s_{\text{asym}} = \frac{w_{\max} - w_{\min}}{2^n} \tag{5-11}$$

$$s_{\text{sym}} = \frac{\max\left(|w_{\min}|, |w_{\max}|\right)}{2^{n-1} - 1} \tag{5-12}$$

$$z_{\text{asym}} = q_{\min} - \frac{w_{\min}}{s_{\text{asym}}} \tag{5-13}$$

$$z_{\text{sym}} = 0 \tag{5-14}$$

5.3.3 Quantization Error

The error introduced by quantization is:

$$\epsilon = w - \hat{w} \tag{5-15}$$

This is used in the calculation of Signal-to-Quantization Noise Ratio (SQNR) as explained in subsubsection 6.4.2.

5.4 Mathematical Formulation of the Arcsine Distribution

We use the arcsine distribution in our analysis. The key mathematical properties of this distribution are outlined below, with detailed derivations provided in the appendix.

5.4.1 Probability Density Function (PDF)

The PDF of the arcsine distribution on the interval $(-1, 1)$ is given by:

$$f(x) = \frac{1}{\pi\sqrt{1 - x^2}} \tag{5-16}$$

5.4.2 Cumulative Distribution Function (CDF)

The cumulative distribution function (CDF), $F(x)$, is defined as the integral of the probability density function (PDF) from the lower bound of the interval to x:

$$F(x) = \int_{-1}^{x} f(t)\, dt \tag{5-17}$$

5.4.3 Mean

The mean μ of the distribution is calculated as follows:

$$\mu = \int_{-1}^{1} x f(x)\, dx = \int_{-1}^{1} x \cdot \frac{1}{\pi\sqrt{1-x^2}}\, dx \tag{5-18}$$

5.4.4 Variance

The variance σ^2 of the distribution is derived from the expected square and the square of the expected value:

$$\sigma^2 = \mathbf{E}[X^2] - (\mathbf{E}[X])^2 \tag{5-19}$$

These equations are fundamental to our project's analysis section and form the basis of our statistical analysis.

5.5 Mathematical Framework in LZMA2 Compression

LZMA2 compression builds upon the classical Lempel-Ziv (LZ) approach by integrating advanced entropy coding techniques. The major mathematical framework behind LZMA2 is the combination of dictionary-based redundancy elimination and probabilistic modeling via range encoding.

5.5.1 Dictionary-Based Compression

At its core, LZMA2 uses a sliding-window dictionary to identify repeated sequences in the input data. Let

$$S = \{s_1, s_2, \ldots, s_N\}$$

denote the input data sequence. The algorithm searches for the longest match $s_i \ldots s_{i+k}$ that has appeared previously within a dictionary window. When a match is found, it is represented as a pair:

$$\text{match} = \{\text{distance}, \text{length}\} \tag{5-20}$$

which replaces the explicit data sequence, thereby reducing redundancy.

5.5.2 Probabilistic Modeling and Range Encoding

After parsing the data into literals (unmatched bytes) and match references, LZMA2 applies range encoding—a variant of arithmetic coding—to compress the sequence based on probability models. Suppose the probability of observing a symbol x is $p(x)$. The

range encoder maintains a current interval $[L, H)$ and refines it for each symbol:

$$\begin{aligned} \text{Range}_{\text{new}} &= \text{Range}_{\text{old}} \times p(x), \\ L_{\text{new}} &= L_{\text{old}} + \text{Range}_{\text{old}} \times C(x), \end{aligned} \tag{5-21}$$

where $C(x)$ is the cumulative probability of all symbols preceding x. This mechanism effectively encodes the entire input as a single number within the final interval.

The probability model is updated adaptively using context modeling. Given a context, the probability $P(x \mid \text{context})$ is estimated by:

$$P(x \mid \text{context}) = \frac{\text{count}(x, \text{context}) + \alpha}{\sum_{x'} (\text{count}(x', \text{context}) + \alpha)} \tag{5-22}$$

where $\text{count}(x, \text{context})$ is the frequency of symbol x within the context and α is a smoothing parameter that prevents zero probabilities.

5.5.3 Integration of Dictionary Matching and Entropy Coding

The synergy between dictionary-based matching and range encoding is what grants LZMA2 its high compression efficiency. The encoder decides between encoding a literal or a match by estimating the compression gain:

$$\text{Compression Gain} = \log_2 \left(\frac{1}{p(\text{match or literal})} \right) \tag{5-23}$$

A higher gain indicates that representing the data via a match or literal will yield a more compact encoded output. This decision process, driven by the underlying probability distributions, allows LZMA2 to balance between exploiting local redundancy (via dictionary matches) and minimizing the bit-cost (via entropy coding).

6 SYSTEM ARCHITECTURE AND METHODOLOGY

This section provides an overview of the system architecture and the methodology used in video codec with a implicit neural network-based approach.

6.1 Block Diagram

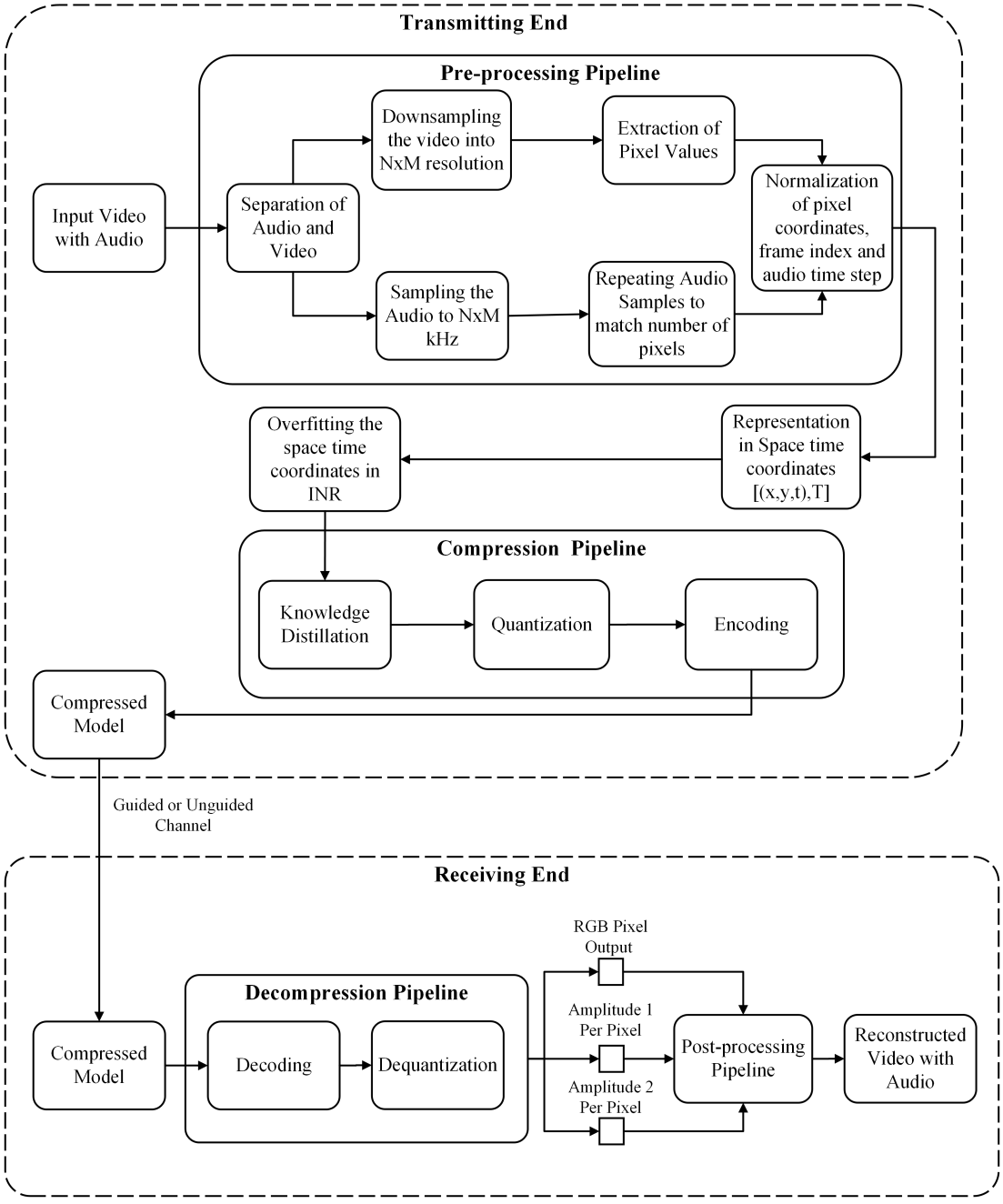


Figure 6-1: System Block Diagram

The block diagram outlines a comprehensive workflow for transmitting and receiving a video with audio through a compression and decompression pipeline. The process be-

gins with the input video that includes both video and audio streams. In the transmitting end, the pre-processing pipeline separates the audio and video streams, downsampling the video to an NxM resolution and sampling the audio to match this resolution in terms of frequency. The extracted pixel values from the video are aligned with repeated audio samples to match the number of pixels, followed by normalization of pixel coordinates (x, y), the frame index (t), and the audio time step (T). The space-time coordinates (x, y, t) are then overfitted using Implicit Neural Representations (INR), preparing the data for compression. The compression pipeline applies knowledge distillation to simplify the model, followed by quantization to reduce data precision and encoding for efficient transmission. The compressed model is then transmitted through a guided or unguided channel. On the receiving end, the compressed model undergoes a decompression pipeline that includes decoding and dequantization to restore the original scale of the data. The post-processing pipeline processes the RGB pixel output along with derived amplitude values per pixel, which are used to reconstruct the video and audio. Finally, the output is the reconstructed video with audio, ensuring the transmitted data is efficiently compressed and accurately restored at the receiving end.

6.1.1 Input Video with Audio

The process begins with an input video that includes both visual and auditory information, serving as the source material for the subsequent encoding and compression processes.

6.1.2 Pre-processing Pipeline

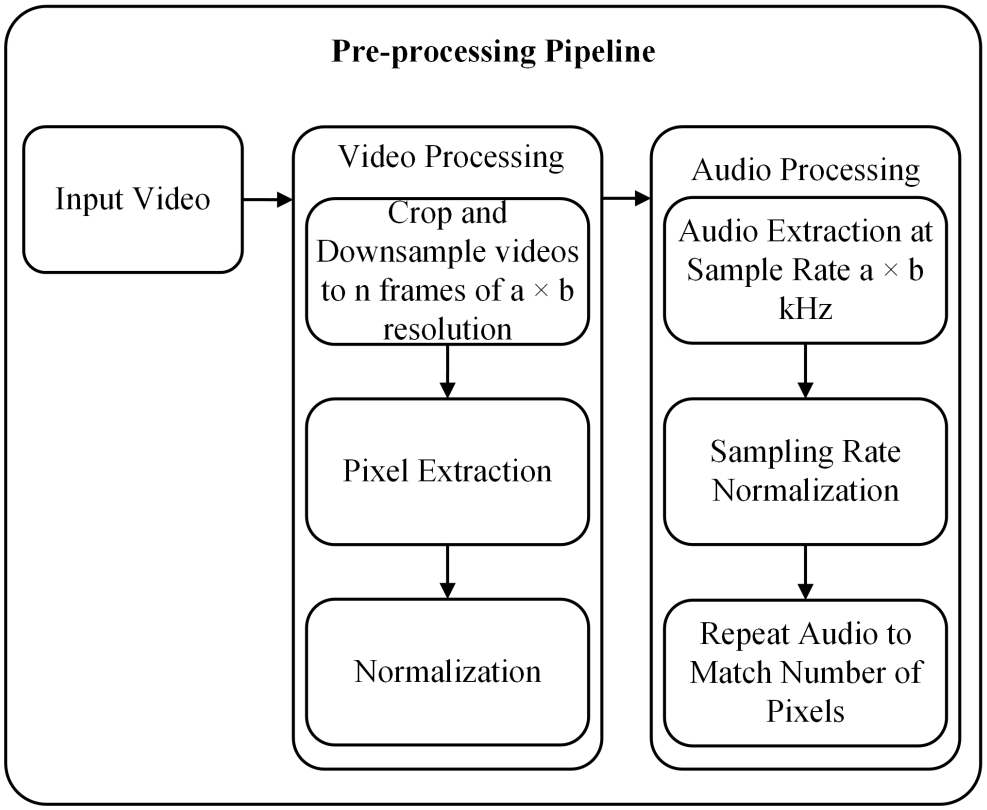


Figure 6-2: Pre-processing Pipeline

Overview of the Pre-processing Pipeline for Video with Audio The figure illustrates a pre-processing pipeline designed for handling video files with accompanying audio tracks. The pipeline is organized into two main branches: one for processing video data and the other for audio data. Each branch standardizes and optimizes the respective data types, ensuring compatibility for integration and further analysis or processing.

Video Processing Branch

- **Crop and Downsample Videos to n Frames of $a \times b$ Resolution:** Video frames are cropped and resized to a fixed resolution of $a \times b$. This step reduces the complexity of the video data, making it computationally efficient while preserving essential visual information. Additionally, the video is downsampled to n frames, ensuring a uniform frame rate and minimizing redundancy.
- **Pixel Extraction:** After downsampling, individual pixel values are extracted from each video frame. These pixel values represent the visual content of the video and encapsulate the color and intensity information for each frame.

- **Normalization:** The extracted pixel values are normalized to a standard range, typically between 0 and 1. Normalization ensures consistency in data representation, minimizes numerical instability, and facilitates improved performance in subsequent processing stages.

Audio Processing Branch

- **Audio Extraction at Sample Rate $a \times b$ kHz:** The audio data is extracted from the input video and resampled to match the video resolution, specifically at a sample rate proportional to the resolution ($a \times b$ kHz). This alignment ensures that the audio data corresponds directly to the spatial dimensions of the video frames.

- **Sampling Rate Normalization:** The resampled audio undergoes normalization to standardize its amplitude range. This ensures uniformity in the audio data and aligns it with the normalized video pixel data.

- **Repeat Audio to Match the Number of Pixels:** To synchronize the audio with the video data, audio samples are repeated so that their quantity matches the number of pixels in each video frame. This step ensures compatibility between the audio and video data for seamless integration and processing.

After pre-processing, the video and audio data are standardized and aligned for integration. The video frames with normalized pixel data and the corresponding audio samples are represented in a unified format. This unified representation ensures that the input data is prepared and ready for further processing, including model fitting and analysis.

6.1.3 Training Model (Fully Connected Neural Network)

A specially designed neural network consisting of multiple fully connected layers is trained to accept space-time coordinates as input and output the corresponding RGB values for pixels and the amplitude for the audio, effectively mapping these inputs to their visual and auditory outputs.

6.1.4 Compression Pipeline

i. **Knowledge Distillation:** A smaller model is trained to mimic the behavior of a larger overfitted model, effectively transferring knowledge while reducing the complexity of the model.

ii. **16-bit Integer Quantization:** The network parameters are reduced in precision

by quantizing them to 16-bit integers, which decreases the model size while retaining essential information and minimizing performance loss.

iii. **Model Encoding:** The quantized model weights are encoded using LZMA. This process compresses the quantized weights into a compact and efficient representation for transmission or storage.

6.1.5 Transmission and Decompression

After the model has been processed through the compression pipeline, it is transmitted over a guided or unguided channel to the receiving end. This transmission ensures that the compressed and encoded model reaches its destination efficiently, preserving the integrity of the compressed data.

Once received, the model undergoes a decompression process, starting with **decoding**. In this step, the encoded data is converted back into its quantized format. The decoding process reverses the arithmetic or range encoding applied during the compression pipeline in LZMA, ensuring that the quantized weights and parameters are accurately restored.

Following decoding, the model is passed through **dequantization**. During this step, the quantized data, represented as 16-bit integers, is scaled back to its original precision range. This scaling is achieved using the quantization parameters (such as scale and zero-point) that were preserved during the compression process. Dequantization restores the numerical accuracy of the model while maintaining the reduced size achieved through quantization.

The decompressed model, now in its restored form, is ready for inference. At this stage, it provides RGB values for each pixel and amplitude values for audio corresponding to the space-time coordinates $[(x,y,t),T]$. These outputs serve as the foundation for the subsequent steps in the post-processing pipeline.'

6.1.6 Post-processing Pipeline

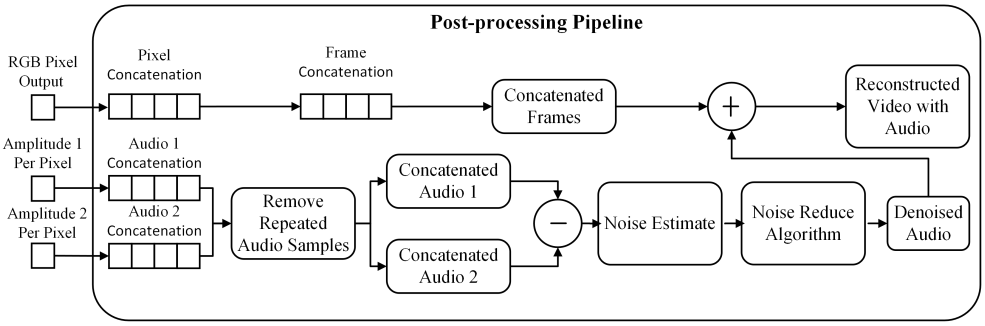


Figure 6-3: Post-processing Pipeline

The post-processing pipeline illustrated in Figure 6-3 outlines the steps taken after the transmitted model has been decompressed and dequantized. The decompressed model is inferenced to retrieve the respective RGB values and audio amplitude values for the space-time coordinates $[(x,y,t),T]$. These outputs are critical for reconstructing the video and audio components.

The RGB values for each pixel are extracted and concatenated into a tensor that represents the color information for the video frames. Similarly, the model outputs two amplitude values for the audio—Amplitude 1 and Amplitude 2—corresponding to each pixel. These amplitude values are concatenated to form a unified audio signal. However, the concatenated audio includes repeated samples, which are removed in the next step to match the original audio sample rate and timing. This ensures the audio remains synchronized with the video frames.

To enhance the audio quality, a noise reduction process is performed. The two audio outputs (Concatenated Audio 1 and Concatenated Audio 2) are subtracted to estimate the noise present in the audio. The estimated noise, along with one of the audio outputs, is then passed through a noise reduction algorithm. This algorithm processes the signals to produce a denoised audio output that is free from distortions and artifacts.

Meanwhile, the concatenated RGB values are used to generate individual video frames. These frames are then merged to reconstruct the full video. Finally, the denoised audio is combined with the reconstructed video to create the complete decompressed video with synchronized audio. This pipeline ensures a high-quality output with minimal noise and distortion, preserving the fidelity of the original input.

6.1.7 Noise Reduction Process

During the training and inference phases, the generated audio signals contained residual noise, which was due to imperfections in the model's processing. To address this, we applied a noise reduction algorithm specifically designed to suppress this residual noise while preserving the integrity of the audio content. The noise estimate was obtained by leveraging the two audio output branches, which generated two inferred outputs for the same signal; the difference between these outputs was used to estimate the noise. Using this noise estimate, a dedicated noise clip was extracted to compute noise characteristics such as mean and standard deviation across frequency bands. A dynamic noise threshold was then determined and applied to the audio spectrogram, followed by Gaussian smoothing to refine the noise suppression mask. This process effectively removed the residual noise and ensured the reconstructed audio maintained high fidelity. The following section details the implementation of this noise reduction algorithm.

Algorithm 1 Noise Reduction Algorithm

Require: audio_clip, noise_clip, parameters (smoothing parameters, FFT size, and threshold control)

Compute Spectrogram of Noise:
noise_stft ← `STFT(noise_clip)`
noise_stft_db ← `ConvertToDB(noise_stft)`
mean_freq_noise ← `Mean(noise_stft_db, axis=freq)`
std_freq_noise ← `Std(noise_stft_db, axis=freq)`
noise_threshold ← mean_freq_noise + $k\times$ std_freq_noise
Compute Spectrogram of Audio:
sig_stft ← `STFT(audio_clip)`
sig_stft_db ← `ConvertToDB(sig_stft)`
Generate Mask:
sig_mask ← (sig_stft_db > noise_threshold)
smoothed_mask ← `ApplyGaussianFilter(sig_mask)`
Apply Mask:
masked_stft ← sig_stft × smoothed_mask
Reconstruct Signal:
recovered_signal ← `ISTFT(masked_stft)`
return recovered_signal

6.2 Quantization

Quantization in machine learning is the process of reducing the number of bits that represent the model's weights and activations. It typically involves converting 32-bit

floating-point numbers to lower precision formats like 16-bit, 8-bit, or even binary (1-bit) representations. This reduces the model size and accelerates inference on hardware with limited computational resources.

Algorithm 2 outline the process of model quantization, which aims to reduce the precision of neural network weights to improve efficiency. The process begins with a trained neural network model. The first step is to choose a quantization scheme, which can be either uniform quantization, where weights are mapped to fixed, uniformly distributed levels, or non-uniform quantization, where techniques like k-means clustering are used to determine the mapping. Following this, the number of quantization levels Q is determined (e.g., 8-bit, 16-bit). Each weight W_{ij} in the model is then mapped to the nearest quantization level within Q. The original weights W are replaced with these quantized weights, resulting in a quantized model. Optionally, the quantized model can be fine-tuned on the training dataset to recover any lost accuracy. The final step is to return the quantized model $Q(W)$. This approach effectively reduces the model's memory footprint and potentially improves its computational efficiency while maintaining its performance.

Algorithm 2 Model Quantization

Require: Neural network model with weights W, training dataset, quantization levels Q (e.g., 8-bit, 16-bit)

1: Train the model on the training dataset
2: Find zero-point and scale factor for each layer
3: **for** each weight W_{ij} in W **do**
4: Map W_{ij} to the nearest quantization level in Q
5: **end for**
6: Replace original weights W with quantized weights $Q(W)$
7: Also store the scale factor and zero-point for each layer
8: **return** $Q(W)$

6.3 Loss Function

To effectively generate INR for audio-video data, it is essential to define appropriate loss functions that guide the learning process. Below, the specific loss functions for audio and video are introduced, followed by the formulation of a combined loss function that integrates both modalities.

6.3.1 Audio Loss Function

The loss function for audio data is defined as:

$$\mathcal{L}_{\text{audio}} = 0.5\left(\int_{\Omega_{\text{audio}}} \|\Phi_{\text{audio},1}(t) - f_{\text{audio}}(t)\|\, dt\right) + 0.5\left(\int_{\Omega_{\text{audio}}} \|\Phi_{\text{audio},2}(t) - f_{\text{audio}}(t)\|\, dt\right) \tag{6-1}$$

where,

- $t \in \mathbb{R}$ is the time point,
- $\Phi_{\text{audio},1}(t)$ is the first audio output of the model at time t,
- $\Phi_{\text{audio},2}(t)$ is the second audio output of the model at time t,
- $f_{\text{audio}}(t)$ is the ground truth audio amplitude at time t,
- Ω_{audio} is the time interval over which the audio is defined.

6.3.2 Video Loss Function

The loss function for video data is defined as:

$$\mathcal{L}_{\text{video}} = \int_{\Omega_{\text{video}}} \|\Phi_{\text{video}}(x,t) - f_{\text{video}}(x,t)\|\, dx\, dt \tag{6-2}$$

where,

- $(x,t) \in \mathbb{R}^3$ represents the space-time coordinates,
- $\Phi_{\text{video}}(x,t)$ is the model's output for the video at space-time coordinate (x,t),
- $f_{\text{video}}(x,t)$ is the ground truth RGB value at space-time coordinate (x,t),
- Ω_{video} is the space-time region over which the video is defined.

6.3.3 Combined Teacher Loss Function

The combined loss function for both audio and video data is formulated as:

$$\mathcal{L}_{\text{combined}} = \lambda_{\text{audio}}\mathcal{L}_{\text{audio}} + \lambda_{\text{video}}\mathcal{L}_{\text{video}} \tag{6-3}$$

where,

- λ_{audio} is the weight that balances the contribution of the audio loss,
- λ_{video} is the weight that balances the contribution of video loss.

This combined loss function ensures that both the audio and video outputs are supervised and optimized simultaneously.

6.3.4 Knowledge Distillation Loss Function

The Knowledge Distillation (KD) loss function is formulated to combine both the hard and soft loss components for audio and video data. The overall loss is defined as:

$$\mathcal{L}_{\text{KD}} = \alpha \mathcal{L}_{\text{hard}} + (1 - \alpha) \mathcal{L}_{\text{soft}} \tag{6-4}$$

where,

- α is a hyperparameter that balances the contribution of the hard and soft loss components,
- $\mathcal{L}_{\text{hard}}$ is the total hard loss (based on the student's output and the ground truth),
- $\mathcal{L}_{\text{soft}}$ is the total soft loss (based on the student's output and the teacher's output).

The hard and soft losses are defined as:

$$\mathcal{L}_{\text{hard}} = 0.5 \times \mathcal{L}_{\text{video, hard}} + 0.5 \times \left(0.5 \times \mathcal{L}_{\text{audio, hard}} + 0.5 \times \mathcal{L}_{\text{audio(siam), hard}}\right) \tag{6-5}$$

$$\mathcal{L}_{\text{soft}} = 0.5 \times \mathcal{L}_{\text{video, soft}} + 0.5 \times \left(0.5 \times \mathcal{L}_{\text{audio, soft}} + 0.5 \times \mathcal{L}_{\text{audio(siam), soft}}\right) \tag{6-6}$$

6.4 Evaluation Metrics for Video

To rigorously evaluate the effectiveness of our proposed INR based video compression technique, we will utilize a suite of metrics that assess various dimensions of video quality, including visual fidelity, perceptual quality, and predictive accuracy. Here's an overview of each metric, their numerical value ranges, and whether it's preferable for their values to increase or decrease:

6.4.1 Peak Signal-to-Noise Ratio

- **Overview**: PSNR is commonly employed to measure the quality of reconstruction in lossy compression codecs. It compares the similarity between the original and compressed video by measuring the ratio between the maximum possible power of a signal and the power of distorting noise.

- **Value Range**: Typically expressed in decibels (dB), the higher the PSNR, the better the quality of the compressed video. A higher PSNR value indicates that the reconstruction is of higher quality.

- **Preference ↑**: Values typically range from 20 to 50 dB, where higher values (around 40 dB or more) represent excellent quality.

- **Formula**

$$\text{PSNR} = 20 \cdot \log_{10}\left(\frac{\text{MAX}_I}{\sqrt{\text{MSE}}}\right) \tag{6-7}$$

where, MAX_I is the maximum possible pixel value of the image (e.g., 255 for 8-bit images), and MSE is the mean squared error between the original and compressed image.

6.4.2 Signal-to-Quantization Noise Ratio (SQNR)

- **Overview**: SQNR is a widely used metric to assess the quality of quantized signals in digital signal processing and machine learning models. It measures the ratio of the power of the original signal to the power of the noise introduced during quantization.

- **Value Range**: Typically expressed in dB, higher SQNR values indicate better fidelity of the quantized signal to the original. A perfect quantization would result in an infinite SQNR.

- **Preference ↑**: Higher values are preferred, as they represent lower quantization noise. For practical systems, values depend on the bit depth and dynamic range of the quantizer.

- **Formula**

$$\text{SQNR} = 10 \cdot \log_{10}\left(\frac{P_s}{P_n}\right) \tag{6-8}$$

where, P_s is the signal power, defined as:

$$P_s = \frac{1}{N}\sum_{i=1}^{N} x_i^2 \tag{6-9}$$

and P_n is the quantization noise power, defined as:

$$P_n = \frac{1}{N}\sum_{i=1}^{N} (x_i - \hat{x}_i)^2 \tag{6-10}$$

Here, x_i is the original signal, $\hat{x}_i$ is the quantized signal, and N is the number of

samples.

6.4.3 Learned Perceptual Image Patch Similarity

- **Overview**: Learned Perceptual Image Patch Similarity (LPIPS) metric evaluates the perceptual difference between two images or videos. Unlike traditional metrics that focus on pixel-level accuracy, LPIPS uses deep learning to assess how perceptually similar two images are to the human eye.
- **Value Range**: This metric produces a similarity score that ranges from 0 to 1, where lower scores indicate greater perceptual similarity.
- **Preference** ↓: A lower LPIPS score means that the compressed video is perceptually closer to the original, indicating better compression performance.

6.4.4 Structural Similarity Index Measure

- **Overview**: Structural Similarity Index Measure (SSIM) is used to measure the similarity between two images or videos. It considers attributes that are important to the human visual system such as luminance, contrast, and structure.
- **Value Range**: SSIM values range from -1 to 1. A score of 1 indicates perfect similarity, meaning there is no loss of structural information.
- **Preference** ↑: Higher SSIM values signify less visual distortion and better compression quality.
- **Formula**

$$\mathrm{SSIM}(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{6-11}$$

  where, μ_x and μ_y are the average intensities of images x and y, σ_x and σ_y are the variance, σ_{xy} is the covariance between x and y, and c_1 and c_2 are variables to stabilize division.

6.4.5 File Size

- **Overview**: File size measures the total amount of storage space required by a compressed video file. It is a direct indicator of the efficiency of a compression technique.
- **Value Range**: File size is typically measured in bytes, kilobytes (KB), megabytes (MB), or gigabytes (GB). The value range can vary widely depending on the

length and resolution of the video.

- **Preference ↓**: Lower file sizes are preferable as they indicate more efficient compression.

6.4.6 Bandwidth Usage

- **Overview**: Bandwidth usage measures the amount of data transmitted over a network to stream or download a video. It evaluates the feasibility of video transmission in various network conditions.
- **Value Range**: Bandwidth usage is typically measured in bits per seconds (bps), kilobits per seconds (kbps), or megabits per seconds (Mbps). The value range can vary depending on the video quality and compression efficiency.
- **Preference ↓**: Lower bandwidth usage is preferable as it indicates more efficient data transmission.

6.4.7 Encoding/Decoding Time

- **Overview**: Encoding/decoding time measures the time required to compress (encode) or decompress (decode) a video file. It is a critical factor in real-time applications and systems with limited processing capabilities.
- **Value Range**: Encoding/decoding time is typically measured in seconds or milliseconds. The exact value can vary widely depending on the complexity of the video content.
- **Preference ↓**: Lower encoding/decoding times are preferred, as they indicate more efficient compression.

6.5 Evaluation Metrics for Audio

To thoroughly assess the effectiveness of our proposed audio processing techniques, we will utilize a comprehensive suite of metrics that evaluate various aspects of audio quality, including clarity, fidelity, intelligibility, and perceptual similarity. Here's an overview of each metric, their numerical value ranges, and the preferred direction for their values:

6.5.1 Log Spectral Distance (LSD)

- **Overview:** Log Spectral Distance (LSD) measures the difference between the log magnitude spectra of the reference and processed audio signals, making it sensitive to spectral distortions.
- **Value Range:** LSD is typically measured in decibels (dB) and can range from 0 (no spectral distortion) to higher positive values.
- **Preference ↓:** Lower values are preferable, indicating lower spectral distortion.
- **Formula:**
  LSD is computed as:

$$\text{LSD} = \frac{1}{N}\sum_{n=1}^{N}\sqrt{\frac{1}{K}\sum_{k=1}^{K}\left[\log|X(k,n)| - \log|Y(k,n)|\right]^2} \tag{6-12}$$

  where, $X(k,n)$ and $Y(k,n)$ are the magnitude spectra of the original and processed signals for the n-th frame and k-th frequency bin, respectively, N is the number of frames, and K is the number of frequency bins.

6.5.2 Peak Signal-to-Noise Ratio (PSNR)

- **Overview:** PSNR is a widely used metric to measure the quality of reconstruction of lossy compression codecs. It represents the ratio between the maximum possible power of a signal and the power of corrupting noise.
- **Value Range:** PSNR is expressed in decibels (dB) and typically ranges from 20 to 50 dB, with higher values indicating better quality.
- **Preference ↑:** Higher values are preferable, indicating less noise and better signal reconstruction quality.
- **Formula:**
  PSNR is calculated as:

$$\text{PSNR} = 20 \cdot \log_{10}\left(\frac{\text{MAX}_I}{\sqrt{\text{MSE}}}\right) \tag{6-13}$$

  where, MAX is the maximum possible pixel value (in audio, the signal's peak value), and MSE is the mean squared error between the original and processed audio signals.

6.5.3 Virtual Speech Quality Objective Listener (VISQOL)

- **Overview:** Virtual Speech Quality Objective Listener (ViSQOL) is an objective, full-reference metric for perceived audio quality. It uses a spectro-temporal measure of similarity between a reference and a test speech signal to produce a MOS-LQO (Mean Opinion Score - Listening Quality Objective) score. MOS-LQO scores range from 1 (the worst) to 5 (the best).
- **Value Range:** ViSQOL scores typically range from **1 to 5**, where **1** indicates poor perceived quality and **5** represents near-perfect audio quality.
- **Preference** ↑: Higher ViSQOL values are preferable, as they indicate better perceptual audio quality and closer resemblance to the original signal.
- **Formula:**
  ViSQOL is computed using a *similarity index* between the original and processed audio signals. The process involves:
  i. Computing frame-level perceptual similarities using frequency representations of the signals.
  ii. Aggregating these similarities into a single perceptual quality score.

6.5.4 File Size

- **Overview**: File size measures the total amount of storage space required by a compressed video file. It is a direct indicator of the efficiency of a compression technique.
- **Value Range**: File size is typically measured in bytes, KB, MB, or GB. The value range can vary widely depending on the length and resolution of the video.
- **Preference** ↓: Lower file sizes are preferable as they indicate more efficient compression.

7 DATASET EXPLORATION

Table 7-1 provides a summary of various audio-video files, detailing their serial numbers (SN), file names, resolution, frames per second (fps), audio sample rate,file size in mebibytes (MiB) and video duration in seconds. All the audio-video files are stored in AVI/wav format because AVI is an uncompressed format that preserves the original, raw audio-video quality without any loss of data. Unlike compressed video formats, which reduce file size by discarding some visual information, AVI/wav files maintain the full integrity of audio-video streams.

Table 7-1: Video Files Characteristics

SN	Video File	Resolution	FPS	Sample Rate (Hz)	Size (MiB)	Duration (s)
1	Video 1	448 x 256	1	114688	1.92	5
2	Video 2		1	96000	5.33	10
3	Video 3		10	114688	6.08	3
4	Video 4		2	114688	13.6	25
5	Video 5		24	114688	12.4	3

Videos 1 and 2 contain temporally uncorrelated frames, selected to assess the model's performance when successive frames undergo significant changes. In Video 1, the audio features a melody played on the violin, with frequency components ranging from 1.60 Hz to 4437.80 Hz, while Video 2 includes the voice of a Nepali male speaker, whose audio spans a frequency range of 0.3 Hz to 5739.9 Hz.

On the other hand, Videos 3, 4, and 5 consist of temporally correlated frames and were chosen to evaluate how the model handles sequentially related data. Video 3 showcases a man dancing, accompanied by a song with audio frequencies spanning 0.1 Hz to 15537.67 Hz. Similarly, Video 4 involves people dancing with funky beats, and its audio frequency components range from 0.4 Hz to 10340.8 Hz. Video 5 presents a person delivering a speech in English, with its audio frequency range being 1.2 Hz to 10512.67 Hz.

The audio sample rates are set higher to accommodate the model's need for pixel coordinates, frame indices, and audio time steps as inputs. Given that the number of pixel coordinates is typically larger than the number of audio time steps, repeating the audio time steps to match the resolution would introduce significant repetition. To minimize this repetition and improve model performance, a higher sample rate is used.

8 IMPLEMENTATION DETAILS

The project is developed using Python, leveraging PyTorch as the primary deep learning framework. Interactive Python sessions are utilized for testing and rapid prototyping. Jupyter Notebooks are employed for exploratory data analysis and iterative experimentation, allowing for an interactive and visual approach to model development. TensorBoard is used to visualize training metrics and monitor the performance of the SIREN network during training. Visual Studio Code serves as the main Integrated Development Environment (IDE) for code writing and debugging.

8.1 Instrumentation

The software and hardware used for this project are listed below:

Table 8-1: Instrumentation Table

Requirement	**Solution**	**Reason**
High-Performance Computing	NVIDIA RTX 4090	Overfitting the video to the model
Deep Learning Frameworks	PyTorch	Developing and implementing neural network models
Development Environment	Visual Studio Code with Notebook extensions	Running and Debugging Code

8.1.1 Hardware Specifications

The project is trained on a system with the following specifications:

- **CPU:** The system is equipped with a 13th Gen Intel® Core™ i9-13980HX processor, featuring 32 cores with a base clock speed of 2.42 GHz. This provides substantial computational power for training deep learning models.
- **GPU:** The primary Graphics Processing Unit (GPU) is an NVIDIA GeForce RTX 4090 Laptop GPU with 16 gibibytes (GiB) of VRAM, which accelerates the training and inference processes of the SIREN network. This powerful GPU is crucial for handling the complex computations required for video compression tasks.
- **Memory:** The system has a total of 24 GiB of RAM, ensuring ample memory for handling large datasets and complex models during training and evaluation.

8.2 Neural Network Architecture (Audio + Video)

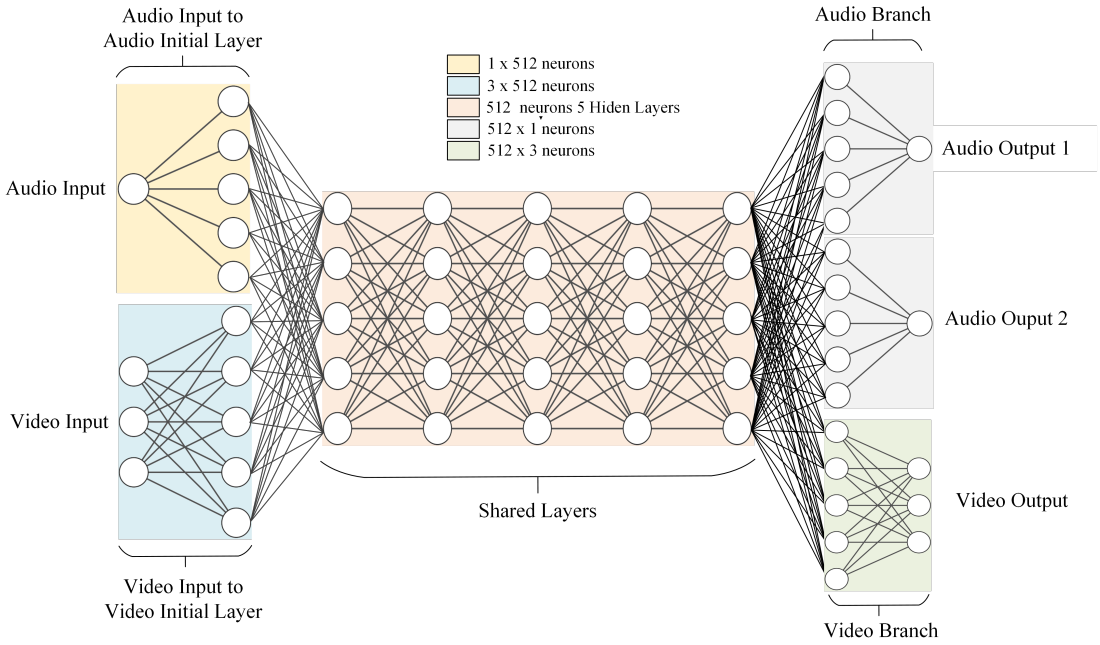


Figure 8-1: SIREN Architecture for Video with Audio

The SIREN architecture for video with audio consists of three major components: the initial layers, the shared layers, and the video/audio branches. The input to the network includes the spatial-temporal coordinates x, y, t, the frame index t, and the audio timestep T, while the output consists of the RGB pixel values and the audio amplitude A.

The input T is passed to the audio initial branch and the spatial-temporal input x, y, t is fed into the video initial branch. The outputs from these initial layers are then combined and sent to the shared layers, which consist of five layers for the teacher model and three layers for student.

The output from the shared layers is branched into three separate layers: two branches for audio and one for video. The two audio branches are identical, each comprising one layer, which is a linear layer. These two branches each produce slightly different amplitudes A for the same audio timestep T, and these outputs are used to estimate and remove noise from the audio data. The video branch consists of one layer as well, being a linear layer, whose output is the RGB values.

The distinct initial layers for audio and video are necessary because it is not possible to have a good initialization for both audio and video if they were in the same branch. The audio branch works well with weights initialized within the range $[-25, 25]$, and the

video branch performs well with weights initialized in the range $[-2/3, 2/3]$. Therefore, separate initial layers are required to optimize the reconstruction quality for both modalities.

The two identical audio branches are crucial because they provide the necessary outputs for noise estimation and removal from the audio data. All hidden layers in the shared and initial branches consist of 512 neurons for teacher model while all layers except hidden layers consists of 300 neurons with hidden layers consisting of 512 neurons for student model, with sine activations, except for the last layers in the two audio output branches and the video output branch, which uses linear activation.

8.2.1 Trainable Parameters

The breakdown of the trainable parameters (weights and biases) are as follows:

Table 8-2: Trainable Parameters in the SIREN Models

Layer	**Parameter**	**Dimensions**		**Total**	
		Teacher	**Student**	**Teacher**	**Student**
Audio Initial Branch	Weight matrix	1×512	1×300	512	300
	Bias vector	512	300	512	300
Video Initial Branch	Weight matrix	3×512	3×300	1,536	900
	Bias vector	512	300	512	300
Shared Layers	Weight matrix	1024×512	600×512	524,288	307,200
	Bias vector	512	512	512	512
	Weight matrix	$4 \times (512 \times 512)$	$2 \times (512 \times 512)$	1,048,576	524,288
	Bias vector	4×512	2×512	2,048	1,024
Video Branch	Weight matrix	512×512	512×300	262,144	153,600
	Bias vector	512	300	512	300
	Weight matrix	512×3	300×3	1,536	900
	Bias vector	3	3	3	3
Audio Branch 1	Weight matrix	512×512	512×300	262,144	153,600
	Bias vector	512	300	512	300
	Weight matrix	512×1	300×1	512	300
	Bias vector	1	1	1	1
Audio Branch 2	Weight matrix	512×512	512×300	262,144	153,600
	Bias vector	512	300	512	300
	Weight matrix	512×1	300×1	512	300
	Bias vector	1	1	1	1
Total				**2,369,029**	**1,298,029**

The total number of trainable parameters in the neural network is 2,369,029 for teacher

model and 1,298,029 for student model.

8.2.2 Hyperparameters for the Model

Table 8-3: Hyperparameters for Training the Teacher and Student Models

Hyperparameter (θ)	**Teacher Model**	**Student Model**
Number of Epochs	7000	6000
Omega (ω)	30	
First Layer Weight (Audio) (β_a)	$(-25, 25)$	
First Layer Weight (Video) (β_v)	$\left(-\frac{2}{3}, \frac{2}{3}\right)$	
Learning Rate (η)	1×10^{-5}	1×10^{-4} for the first 1000 epochs, 1×10^{-5} for the remaining 5000 epochs
Optimizer (Opt)	Adam	

The model was trained for 7000 epochs for the teacher model and 6000 epochs for the student model, with the best-performing version saved based on the lowest loss achieved during training. The value of ω was set to 30 for both models, following the default configuration of the SIRENmodel, as it effectively balances frequency representation.

For the first layer weight initialization, the audio branch used a range of $(-25, 25)$ (β_a) for both the teacher and student models. This range was specifically chosen to accurately represent high-frequency audio features while avoiding unnecessary noise. Larger ranges were tested but resulted in excessive noise, making $(-25, 25)$ the ideal choice.

Similarly, the video branch utilized a narrower initialization range of $\left(-\frac{2}{3}, \frac{2}{3}\right)$ (β_v) for both models, reflecting the lower frequency nature of video data. Wider ranges led to grainy images and inaccurate color predictions, confirming this range as optimal.

The learning rate was set to 1×10^{-5} (η) for the teacher model and 1×10^{-4} for the first 1000 epochs, then 1×10^{-5} for the remaining 5000 epochs for the student model, ensuring stable and gradual optimization without large fluctuations in the loss function. The Adam optimizer (Opt) was employed for both models, due to its suitability for SIREN-based models, leveraging its adaptive learning capabilities to efficiently handle sinusoidal feature extraction and optimize the model effectively.

8.3 Data Preparation and Training Implementation

8.3.1 Pseudocode for Data Preparation

Algorithm 3 ensures that the total number of video pixels and corresponding audio samples are synchronized for input compatibility with the model, which requires inputs in the format $[(x, y, t), T]$.

If the number of video samples (`video_size`) is less than the number of audio samples (`audio_size`), the algorithm repeats the video coordinates and data until they match the audio sample count. The repetition is performed by calculating a repeat factor as the integer division of `audio_size` by `video_size`, and appending any remaining samples as needed.

Similarly, if the audio sample count is smaller, the audio coordinates and data are repeated to align with the video sample count. The process ensures that both video and audio inputs are synchronized and ready for processing by the model.

The synchronized outputs are returned as `video_coords`, `video_data`, `audio_coords`, and `audio_data`.

Algorithm 3 Synchronize Video and Audio Samples

Require: video_coords, video_data, audio_coords, audio_data
video_size ← Number of video samples
audio_size ← Number of audio samples
if video_size $<$ audio_size **then**
    repeat_factor ← $\lfloor$ audio_size / video_size $\rfloor$
    remainder ← audio_size mod video_size
    Repeat video coordinates and data by repeat_factor
    if remainder $>$ 0 **then**
        Append first remainder video samples
    end if
else if audio_size $<$ video_size **then**
    repeat_factor ← $\lfloor$ video_size / audio_size $\rfloor$
    remainder ← video_size mod audio_size
    Repeat audio coordinates and data by repeat_factor
    if remainder $>$ 0 **then**
        Append first remainder audio samples
    end if
end if
return Synchronized video_coords, video_data, audio_coords, audio_data

Algorithm 4 prepares ground truth data and input coordinates for the model. It loads and normalizes video and audio data from the provided paths, generating spatial (x, y) and temporal (frame index, time) coordinates. Video and audio are synchronized using

Algorithm 3, ensuring alignment between frames and audio samples. The combined coordinates and normalized data are returned as the final input for the model.

Algorithm 4 Load and Prepare Video with Audio Data

Require: path_to_video, path_to_audio, sidelength
Load Video Data:
if path_to_video contains '.npy' **then**
    video_data ← Load data using NumPy
else if path_to_video contains '.mp4' **or** path_to_video contains '.avi' **then**
    video_data ← Read frames using skvideo.io.vread
    Normalize video_data ← video_data / 255.0
end if
video_shape ← (frames, height, width)
channels ← RGB channels of video_data
Load or Extract Audio Data:
if path_to_audio contains '.mp4' **or** path_to_audio contains '.avi' **then**
    Extract audio using FFmpeg and save as a temporary .wav file
    audio_rate, audio_data ← Load .wav file using wavfile.read
    Remove the temporary .wav file
else
    audio_rate, audio_data ← Load directly using wavfile.read
end if
Normalize audio_data ← audio_data / max(abs(audio_data))
Compute audio_samples_per_frame ← len(audio_data) / Number of video frames
Generate Video Coordinates:
mgrid_video ← Generate (x, y) grid for each frame using get_mgrid
Normalize frame indices frameindex ← linspace(0, 1, Number of frames)
Repeat (x, y) coordinates for all frames and append frameindex
video_coords ← Flattened (x, y, frameindex)
Flatten video_data ← Normalize and reshape to (N, channels)
Generate Audio Coordinates:
Normalize time audio_coords ← linspace(0, 1, len(audio_data))
Flatten audio_data ← Reshape to (N, 1)
Synchronize Video and Audio:
(video_coords, video_data, audio_coords, audio_data) ← **Algorithm 3**
Combine Video and Audio Data:
combined_coords ← Concatenate video_coords and audio_coords
combined_data ← Concatenate video_data and audio_data
return (combined_coords, combined_data)

8.3.2 Pseudocode for Training

The following algorithms describe training the teacher and student models using knowledge distillation, where the teacher guides the student to achieve efficient performance by transferring knowledge.

Algorithm 5 Training the Teacher Model

Require: Dataset `D`, model `M`, optimizer `O`, hyperparameters: epochs `N`, learning rate `lr`, batch size `B`, best loss `best_loss`

Initialize dataset `D`, DataLoader `D_loader`, and model `M`.
Initialize optimizer `O` with learning rate `lr`.
Set the number of epochs `epochs = N` and initialize `best_loss = +∞`.
for epoch `i = 1 to N` **do**
    Sample a batch `(inputs, targets)` from `D_loader`.
    Move data to device: `inputs, targets`.
    Step 1: Model Inference
    Compute model outputs:

$$\hat{y} = M(x)$$

    Step 2: Compute Loss
    Compute the loss function:

$$L = \lambda_1 \cdot \text{MSE}(\hat{y}_{\text{video}}, y_{\text{video}}) + \lambda_2 \cdot \text{MSE}(\hat{y}_{\text{audio}}, y_{\text{audio}})$$

    Step 3: Backpropagation and Optimization
    Zero gradients and perform backpropagation:

$$\nabla_\theta L$$

    Update model parameters:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L$$

    Step 4: Save Best Model
    if `L < best_loss` **then**
        Save the model weights and update best_loss.
    end if
    Step 5: Log Training Progress
    Log or print current loss and training progress.
end for

Algorithm 5 describes the training process for the teacher model. It initializes the dataset, model, optimizer, and training parameters such as learning rate. During training, batches of input data and their corresponding targets are sampled and used for inference. The model computes outputs, calculates the loss based on MSE for both video and audio components, and updates its weights using backpropagation.

Algorithm 6 outlines training the student model using knowledge distillation. It leverages a pretrained teacher model to guide the learning of the student model. The teacher's

Algorithm 6 Training the Student Model Using a Teacher Model

Require: Pretrained teacher model `T`, student model `S`, dataset `D`, optimizer, hyperparameters: epochs `N`, learning rate η, weight α

Load the dataset `D` and prepare a data loader.
Load teacher model `T` with pretrained weights.
Initialize student model `S` and optimizer with learning rate η.
for each epoch $i = 1, \ldots, N$ **do**
    Sample a batch `(x, y)` from the data loader, where `x` is the input and `y` is the ground truth.
    Move data `(x, y)` to the computation device (e.g., GPU).
    Step 1: Teacher Model Inference
    Compute teacher outputs:

$$\hat{y}_T = T(x)$$

    Step 2: Student Model Inference
    Compute student outputs:

$$\hat{y}_S = S(x)$$

    Step 3: Compute Losses
    Split ground truth `y` into components (e.g., `y = (y_video, y_audio)`).
    Compute the hard loss (comparison with ground truth):

$$L_{\text{hard}} = \beta_1 \cdot \text{MSE}(\hat{y}_S^{\text{video}}, y_{\text{video}}) + \beta_2 \cdot \text{MSE}(\hat{y}_S^{\text{audio}}, y_{\text{audio}})$$

    Compute the soft loss (comparison with teacher outputs):

$$L_{\text{soft}} = \gamma_1 \cdot \text{MSE}(\hat{y}_S^{\text{video}}, \hat{y}_T^{\text{video}}) + \gamma_2 \cdot \text{MSE}(\hat{y}_S^{\text{audio}}, \hat{y}_T^{\text{audio}})$$

    Combine hard and soft losses:

$$L_{\text{total}} = \alpha \cdot L_{\text{hard}} + (1 - \alpha) \cdot L_{\text{soft}}$$

    Step 4: Update Student Model
    Perform backpropagation and update `S`:

$$\theta_S \leftarrow \theta_S - \eta \cdot \nabla_{\theta_S} L_{\text{total}}$$

    Step 5: Save Best Model (if applicable)
    if $L_{\text{total}} < L_{\text{best}}$ **then**
        Save `S` and update $L_{\texttt{best}}$.
    end if
    Log training progress (e.g., loss values) to a file.
end for

outputs serve as "soft targets", in addition to the "hard targets" from the ground truth. Losses are computed for both soft and hard targets, and a weighted loss drives the student model's optimization enables the student model to mimic the teacher while retaining its own efficiency.

8.4 LZMA2 Compression of Model

Algorithm 7 LZMA2 Compression Algorithm

Require: input_data, dictionary_size, compression_level
Initialize dictionary with size dictionary_size
Configure compression parameters based on compression_level
Initialize range encoder and output buffer
pos ← 0
while pos < len(input_data) **do**
  Determine block_size based on remaining data and performance criteria
  Evaluate compression gain for the current block
  if compression gain is insufficient **then**
    Set Block Type: Uncompressed
    Write control byte with uncompressed flag and block_size to output buffer
    Copy raw data block from input_data[pos : pos+block_size] to output buffer
  else
    Set Block Type: Compressed
    Write control byte with compressed flag and block_size to output buffer
    for each position `i` in input_data[pos : pos+block_size] **do**
      Search dictionary for the longest match for input_data[i:]
      if match is found **then**
        Encode match length and distance using range encoder
        Update dictionary with matched sequence
        Advance `i` by match length
      else
        Encode literal byte input_data[i] using range encoder
        Update dictionary with literal byte
        Advance `i` by 1
      end if
    end for
    Append encoded block from range encoder to output buffer
  end if
  Update pos ← pos + block_size
end while
Write end-of-stream marker to output buffer
return compressed_data from output buffer

Algorithm 7 outlines the abstract LZMA2 compression process. The algorithm begins by initializing a dictionary and configuring the compression parameters based on the desired compression level. It then iterates over the input data in blocks, dynamically deciding whether to compress a block or store it uncompressed based on the potential compression gain. For compressed blocks, the algorithm performs match searching within the dictionary, encoding matches and literal bytes via a range encoder. Finally, it writes an end-of-stream marker and returns the assembled compressed data.

8.5 Website Development

The website serves as the primary interface for users to interact with the system, allowing them to upload videos, train INR models, perform inference, and transfer videos or trained models to other users within the same network. The development of the website involved the following key techniques:

- Client-Server Architecture: The frontend communicates with the backend using HTTP requests for model training and inference while using WebRTC and WebSockets for peer-to-peer file transfers.
- Real-Time Communication: WebRTC is used for direct peer-to-peer transfer of trained INR models and videos between users in the same network, reducing the dependency on a central server.
- Interactive and Responsive UI: The frontend is built using modern web development technologies, ensuring a seamless user experience for managing video uploads, model training, and inference.
- Efficient Model Processing: The backend, developed using FastAPI, handles INR model training and inference efficiently using Python subprocess execution to run deep learning scripts.

8.5.1 Frontend Technologies

The frontend of the website is designed to ensure usability, responsiveness, and seamless interaction between users. The key technologies used are:

- HTML, CSS, and JavaScript: These form the core structure and styling of the website.
- Tailwind CSS: Used for rapid UI development and maintaining a clean and responsive design.
- WebSockets: Facilitates real-time communication with the backend for managing WebRTC connections and signaling.
- WebRTC API: Enables real-time, peer-to-peer file transfers without requiring an intermediary server.
- UUID-based Client Identification: Ensures unique client identification for WebRTC connections.

8.5.2 Backend Technologies

The backend of the system is implemented using FastAPI, a high-performance web framework for building APIs in Python. The backend handles the following responsibilities:

- Model Training and Inference: The backend processes videos by running INR model training scripts and returning trained models.
- File Storage Management: Uploaded videos and trained INR models are structured in organized directories for efficient retrieval.
- WebSocket Communication for Signaling: The backend manages WebRTC connection establishment by relaying signaling messages between users.
- Real-Time Client Tracking: It maintains an active list of connected users and dynamically updates the user interface to reflect available clients for file transfers.

8.5.3 WebRTC for Real-Time Communication

WebRTC (Web Real-Time Communication) is a core component of the system that enables direct peer-to-peer communication. It allows users within the same network to transfer trained INR models and videos efficiently without routing data through a central server.

WebRTC Architecture The system follows a structured WebRTC connection flow:

i. Client Discovery and Connection Establishment: Users are identified via a WebSocket connection, which is used to establish initial contact.

ii. Signaling via WebSockets: WebRTC requires an external signaling mechanism to exchange metadata (such as session descriptions and ICE candidates). WebSockets are used for this purpose.

iii. Peer-to-Peer Data Transmission: Once a connection is established, file transfers occur directly between users over WebRTC's Data Channel.

iv. Low-Latency Data Transfer: The use of WebRTC minimizes latency and maximizes transfer speed by avoiding cloud-based storage or intermediary servers.

ICE Server Configuration To facilitate NAT traversal and allow direct communication between peers, WebRTC requires an ICE (Interactive Connectivity Establishment) server. The system employs a public STUN (Session Traversal Utilities for NAT) server

to ensure connection establishment across different network configurations.

WebRTC and WebSockets Integration WebRTC connections rely on WebSockets for signaling before peer-to-peer communication can be established. The integration follows these steps:

- A user requesting a file transfer sends an offer via WebSocket.
- The receiving user replies with an answer, establishing the connection.
- ICE candidates (network details) are exchanged to finalize the connection.
- Once established, the file transfer occurs directly between peers over the WebRTC Data Channel.

8.5.4 Backend Implementation Using FastAPI

The backend is designed to handle INR model training, inference requests, and WebRTC signaling efficiently. It includes the following components:

WebSocket Management The backend manages WebSocket connections to enable real-time client communication and WebRTC session initiation. It maintains an updated list of connected users and their available models for transfer.

Model Training and Inference Pipeline When a user uploads a video for training, the backend performs the following steps:

i. Storage: The uploaded video is stored in a designated directory.

ii. Processing: A Python subprocess executes the INR model training script on the uploaded video.

iii. Model Generation: The trained model is saved and made available for inference or peer-to-peer transfer.

iv. Inference Support: Users can submit trained models for inference, generating reconstructed video representations.

File Management and API Endpoints The backend provides API endpoints to manage file uploads, inference execution, and retrieving processed results. It ensures efficient file handling, secure storage, and dynamic retrieval of trained INR models.

8.6 UML Diagrams

To better visualize and understand the workflow and interactions within the application, the following Unified Modeling Language (UML) diagrams illustrate the key processes involved in model training, inference, and file transfer.

8.6.1 Sequence Diagram

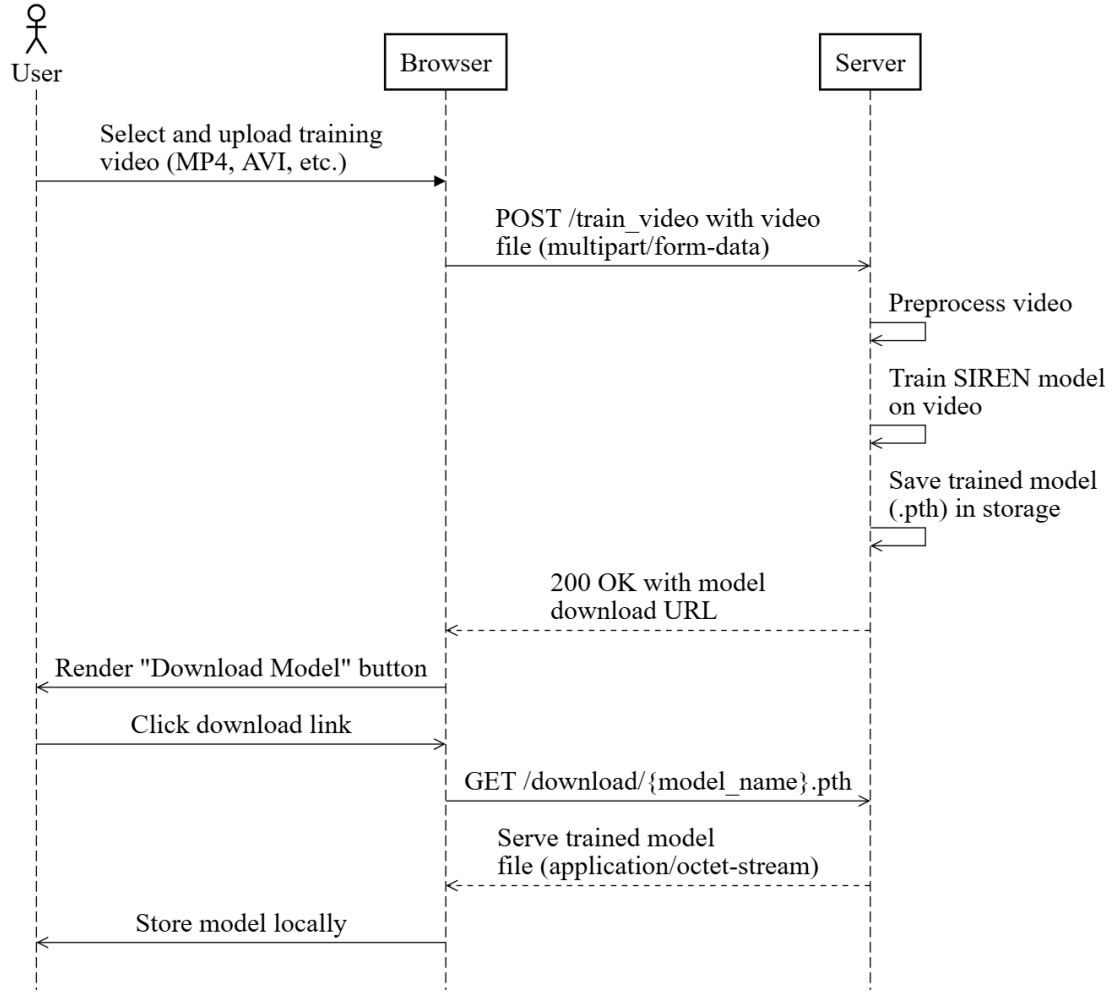


Figure 8-2: Sequence Diagram for Training the Model

Figure 8-2 illustrates the sequence of steps involved in the training of the model. It begins when the user selects and uploads a training video through the browser. The browser sends the video file to the server, which then preprocesses the video and trains the SIREN model on it. Once the model is trained, it is saved and a download URL is provided back to the user through the browser. The user can then download the trained model.

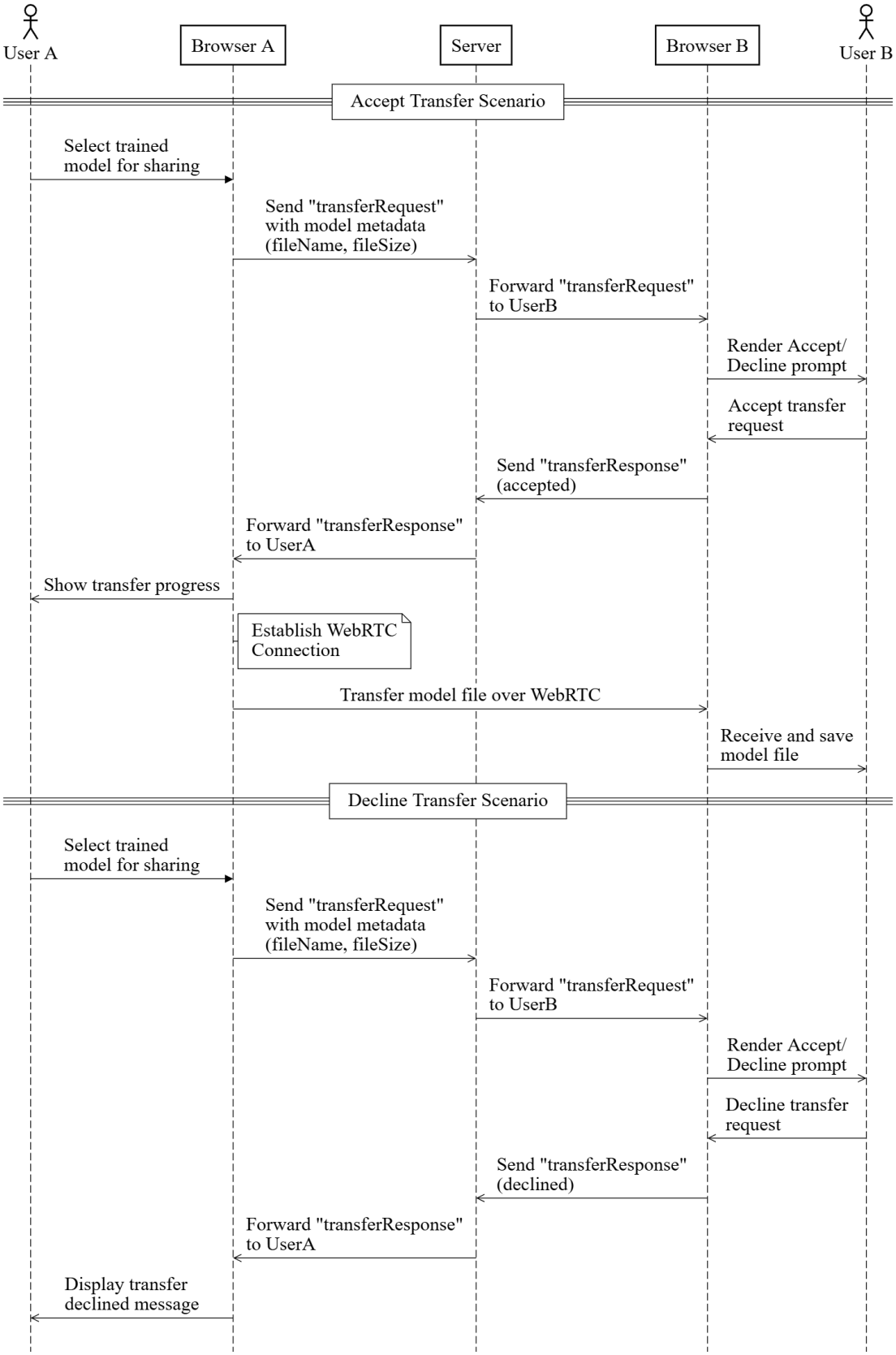


Figure 8-3: Sequence Diagram for the File Transfer

Figure 8-3 shows the process of transferring the trained model from one user to another. The first user selects a model for sharing, and the browser sends a transfer request to the server, including the model's metadata. The server forwards the request to the second user's browser, which prompts them to accept or decline the transfer. If accepted, the model file is transferred over a Web Real-Time Communication (WebRTC) connection; if declined, the first user is notified. This process includes scenarios for both accepting and declining the transfer.

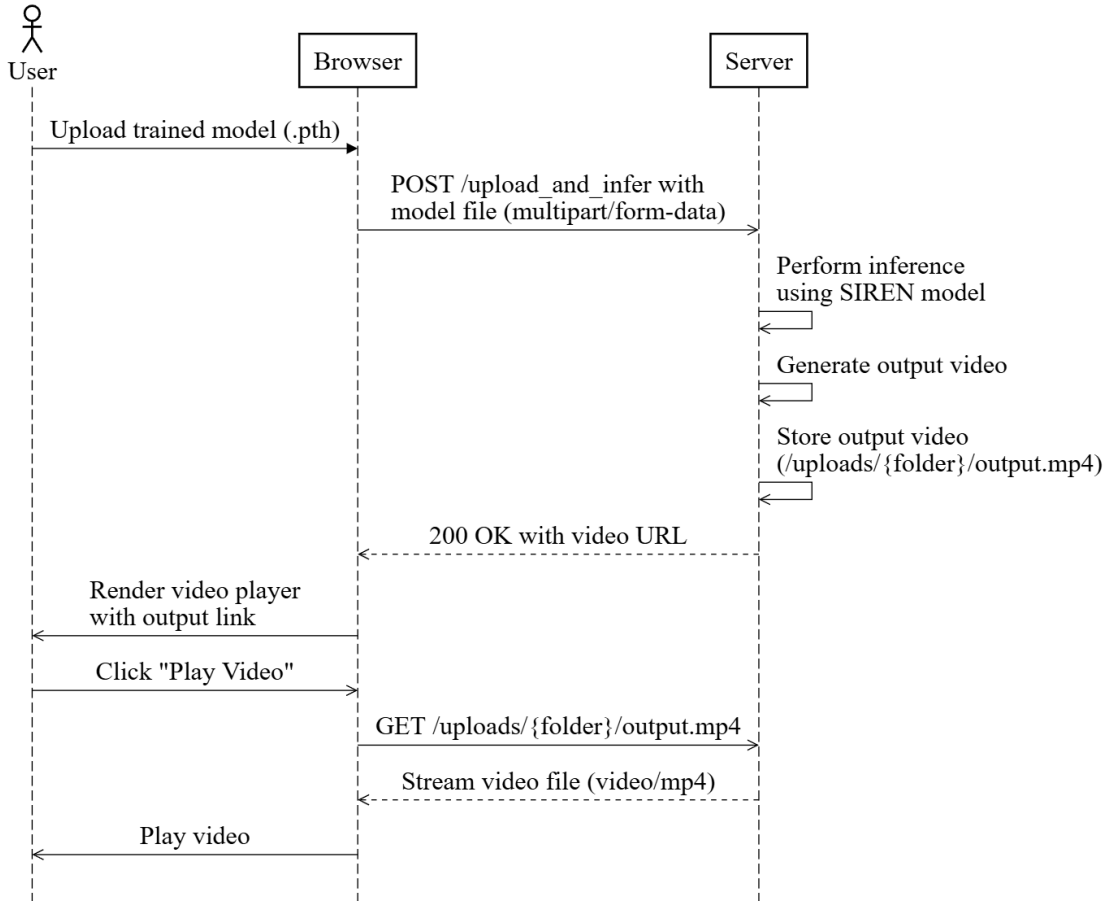


Figure 8-4: Sequence Diagram for Inferencing the Model

Figure 8-4 shows the process where the user uploads a trained model file to the server for inference. The server inferences the trained SIREN model, generates the output video, and stores it. The server then sends the URL of the output video back to the browser, where the user can view the video player and click "Play Video" to stream the video from the server.

8.6.2 Usecase Diagram

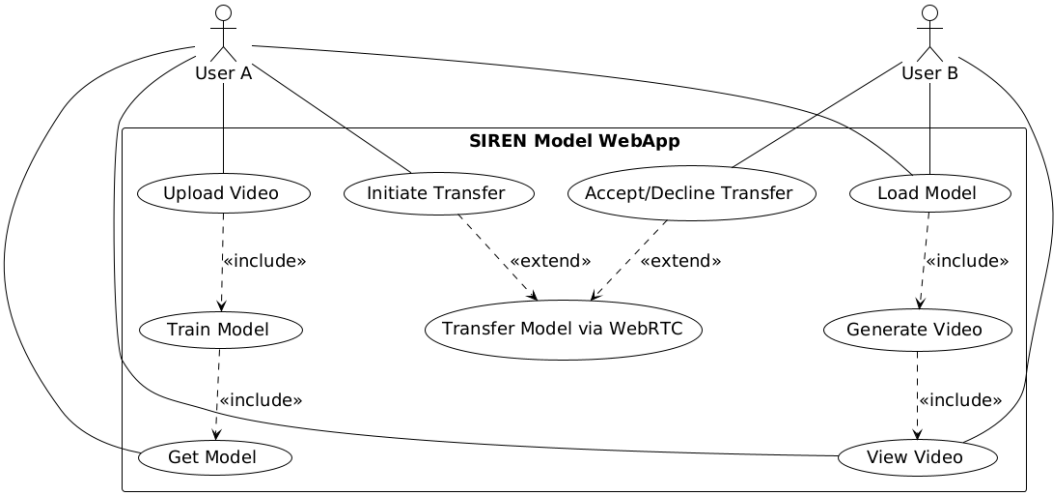


Figure 8-5: Usecase Diagram

Figure 8-5 illustrates the use case diagram for the SIREN model application. It defines the interactions between two actors, User A and User B, with various use cases related to model training, inference, and file transfer.

The diagram is divided into three primary areas:

- **Model Training Related Use Cases:** This includes uploading the training video, training the SIREN model, and downloading the trained model.
- **Model Inference Related Use Cases:** This section involves uploading the trained model, generating the output video, and viewing the output video.
- **File Transfer Related Use Cases:** This area includes initiating the model transfer, accepting or declining the transfer, and transferring the model via WebRTC.

9 RESULTS AND ANALYSIS

This section covers the results and analysis of the models, including loss curves, audio analysis, comparisons with traditional codecs, and the effects of quantization.

9.1 Teacher Model Loss Curves

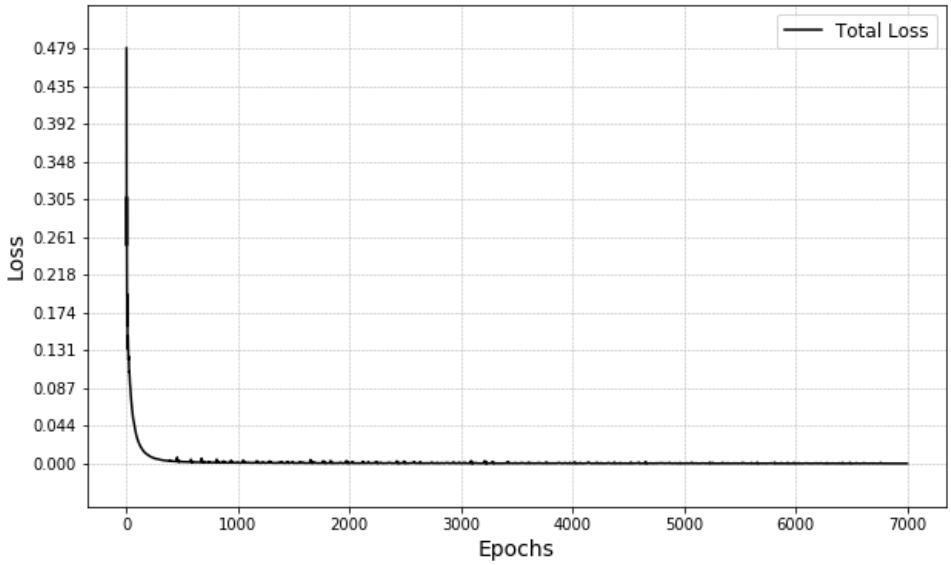


Figure 9-1: Teacher Model Training Loss Curve for Video 1

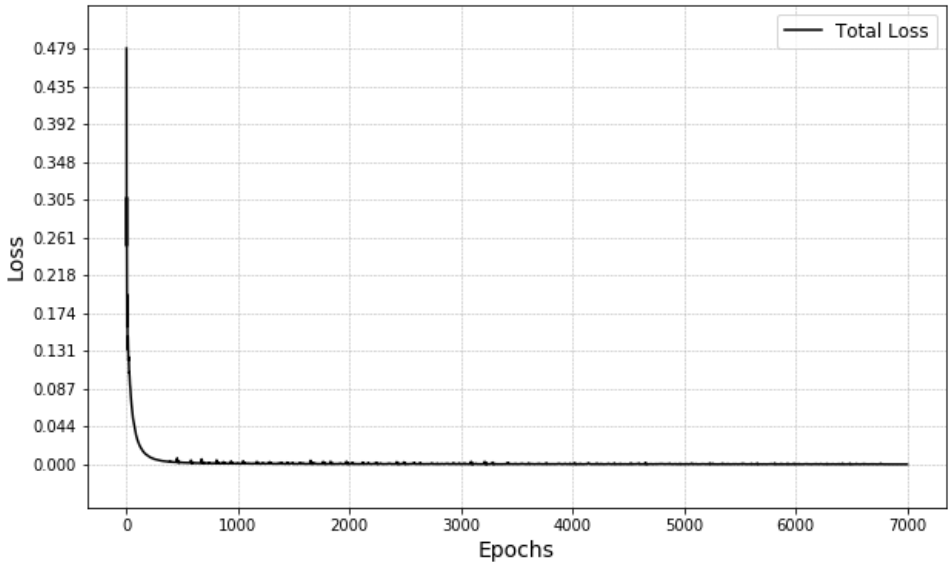


Figure 9-2: Teacher Model Training Loss Curve for Video 2

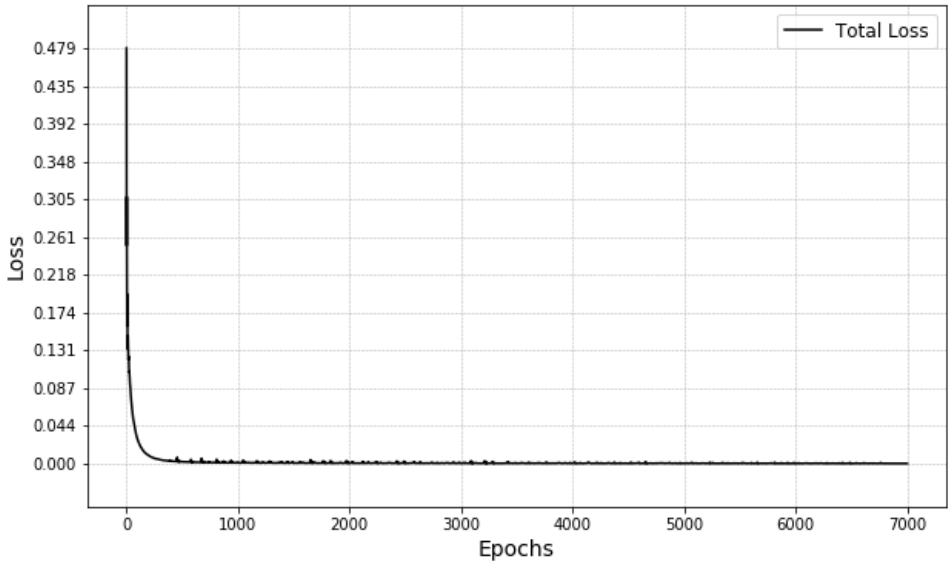


Figure 9-3: Teacher Model Training Loss Curve for Video 3

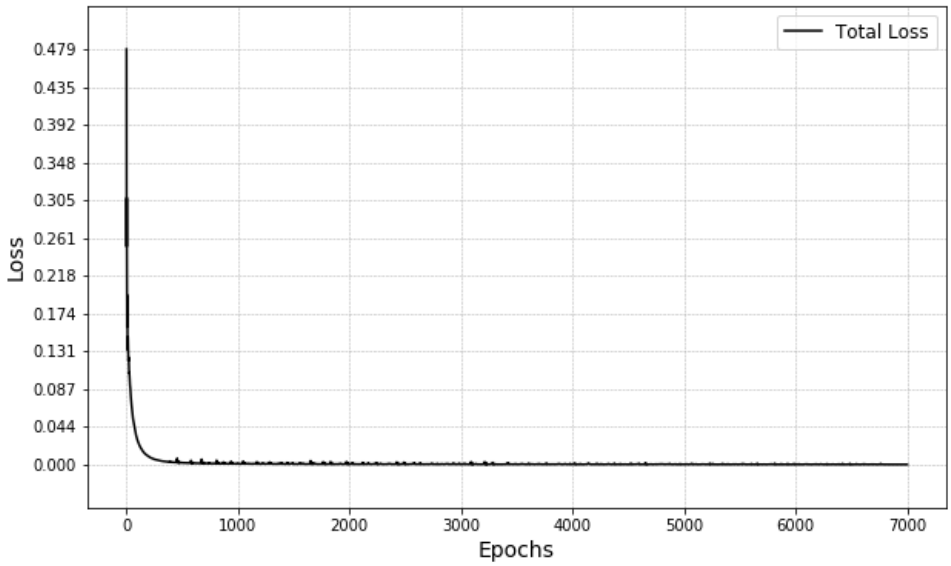


Figure 9-4: Teacher Model Training Loss Curve for Video 4

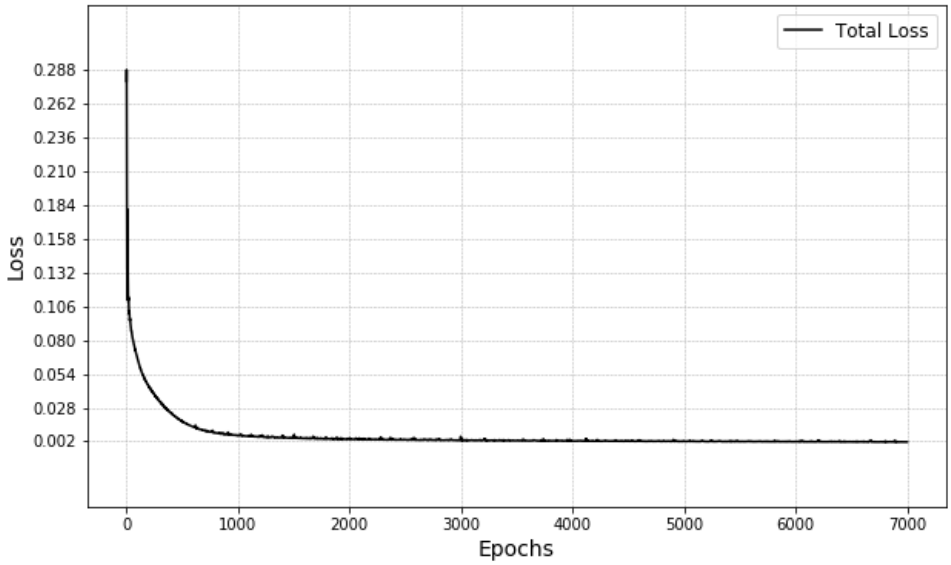


Figure 9-5: Teacher Model Training Loss Curve for Video 5

The loss values for the model start at a low level and show a very steep decline, approaching nearly zero around epoch 500. After this point, the rate of decline in the loss slows down almost being linear. Although the loss value approaches zero, training continues beyond this point. This is because even though there were only minute improvements in the loss function the performance metrics such as PSNR, SSIM, LPIPS, LSD were improving. The continued training ensured that these evaluation metrics were optimized, further enhancing the quality of the video representation model.

9.2 Student Model Loss Curves

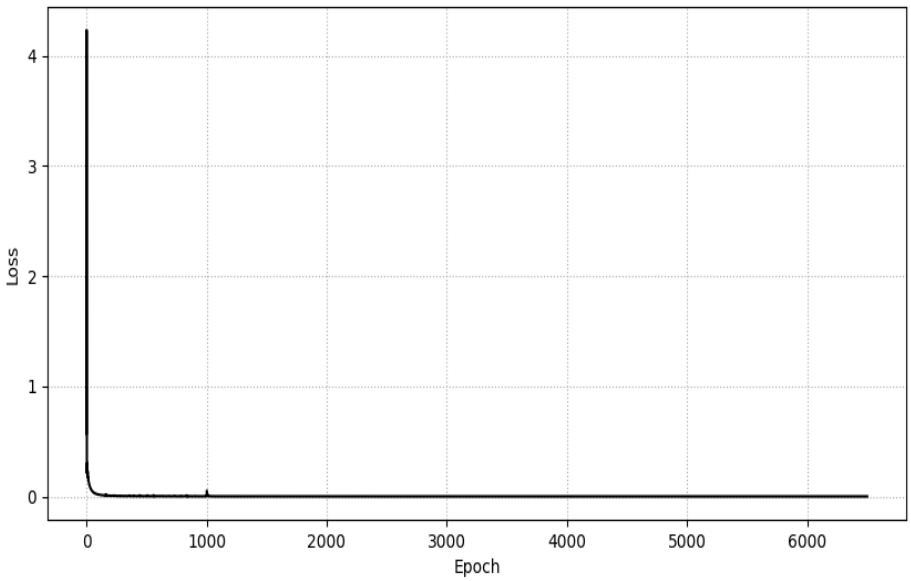


Figure 9-6: Student Model Training Loss Curve of Video 1

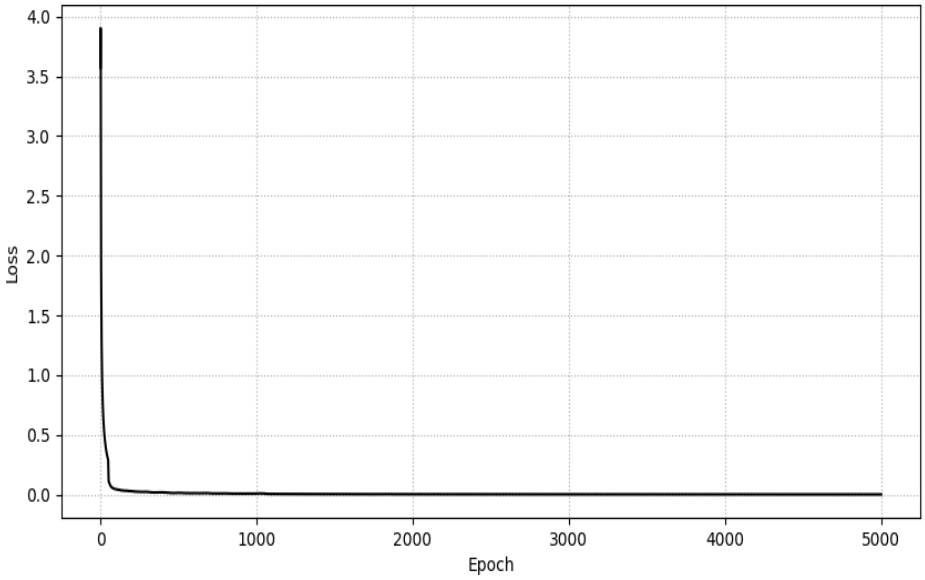


Figure 9-7: Student Model Training Loss Curve of Video 3

Initially, the loss begins at a relatively high value, with a sharp and rapid decline within the first few hundred epochs, indicating significant progress in optimizing the model. Within epoch 200, the loss value stabilizes and approaches nearly zero, with further reductions becoming progressively smaller and almost linear in trend. Although the

improvements in the loss function become minimal after this point, training continues in the same manner as for the teacher models.

9.3 FFT and Spectrogram Analysis of Audio Content Across Videos

This section provides a visual analysis of the fidelity of the audio predictions by comparing the ground truth and predicted signals in both the time and frequency domains.

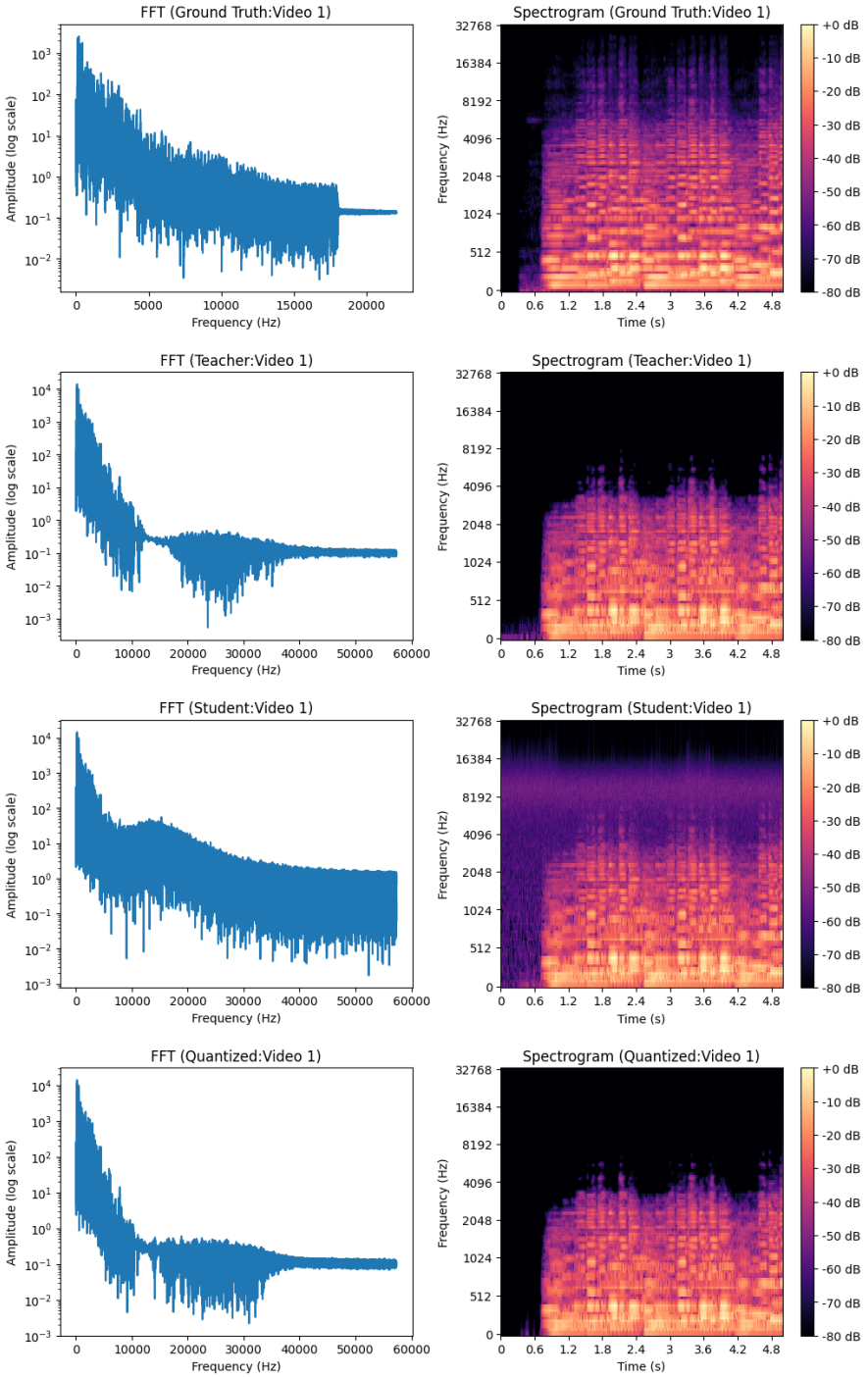


Figure 9-8: FFT and Spectrogram Analysis of Audio Content in Video 1

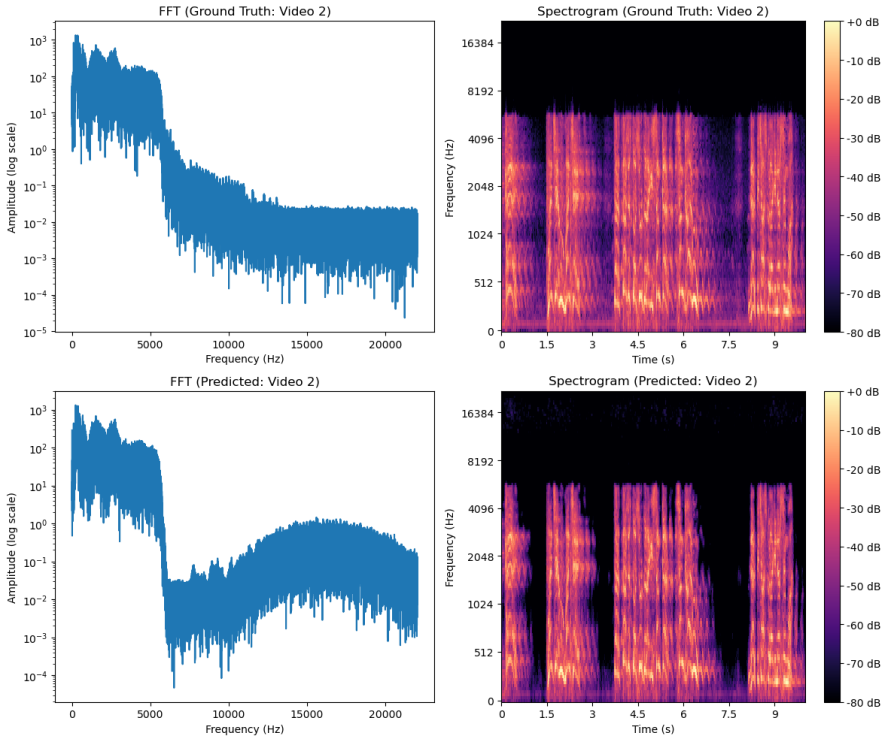


Figure 9-9: FFT and Spectrogram Analysis of Audio Content in Video 2

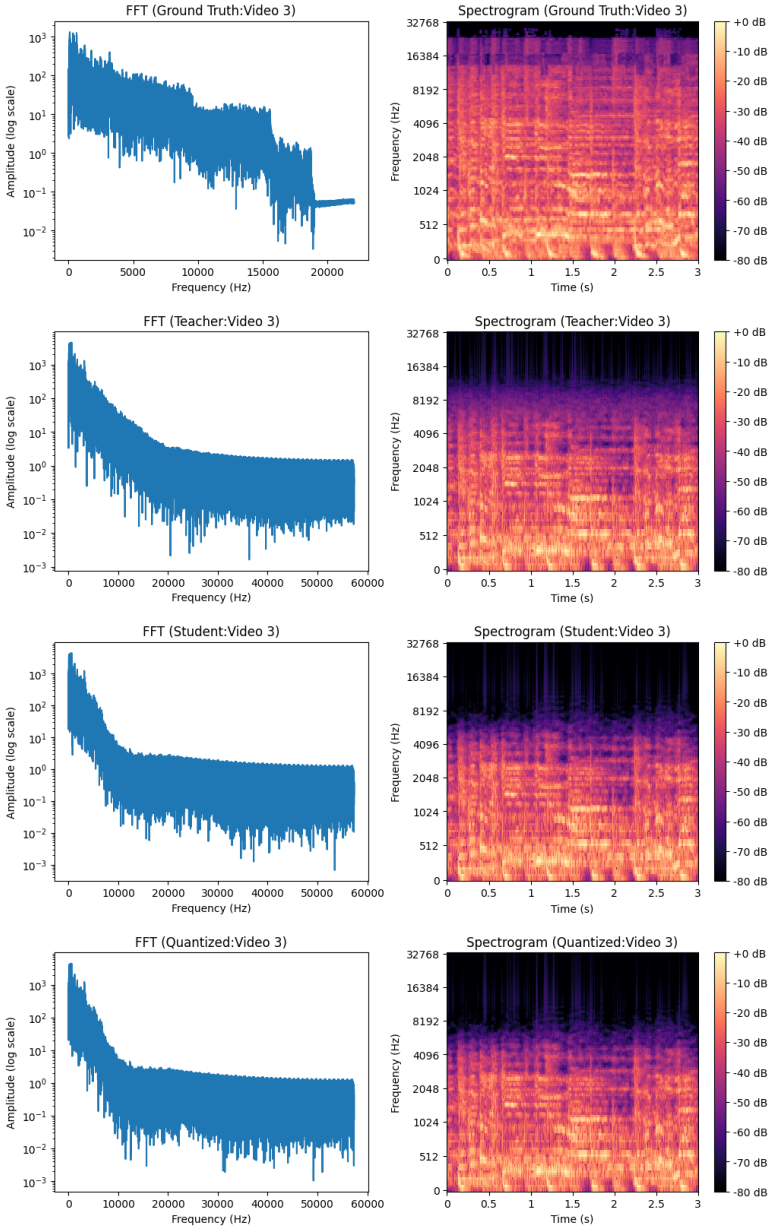


Figure 9-10: FFT and Spectrogram Analysis of Audio Content in Video 3

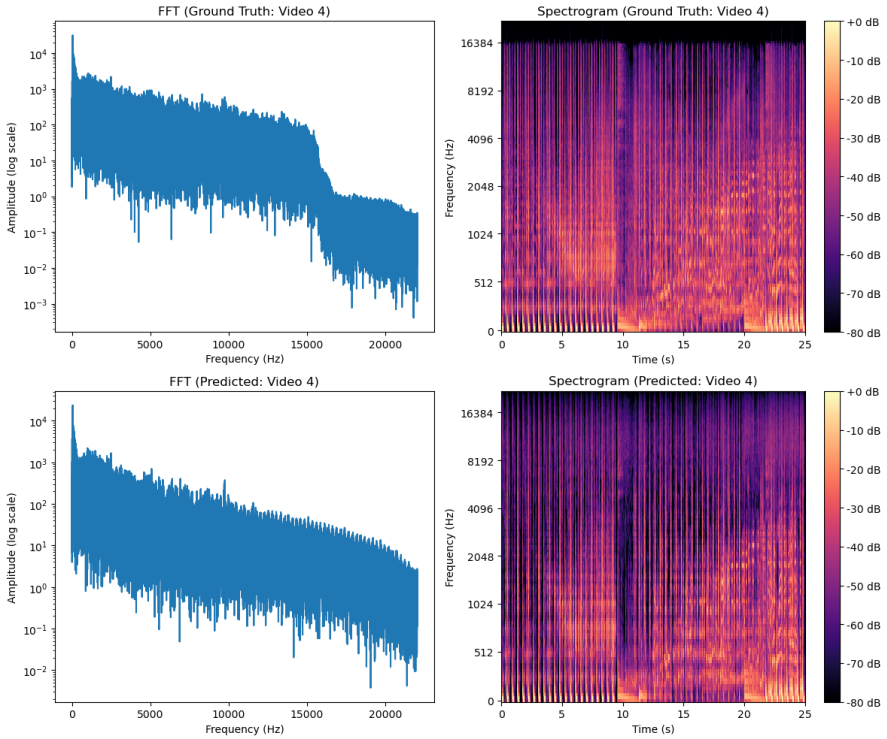


Figure 9-11: FFT and Spectrogram Analysis of Audio Content in Video 4

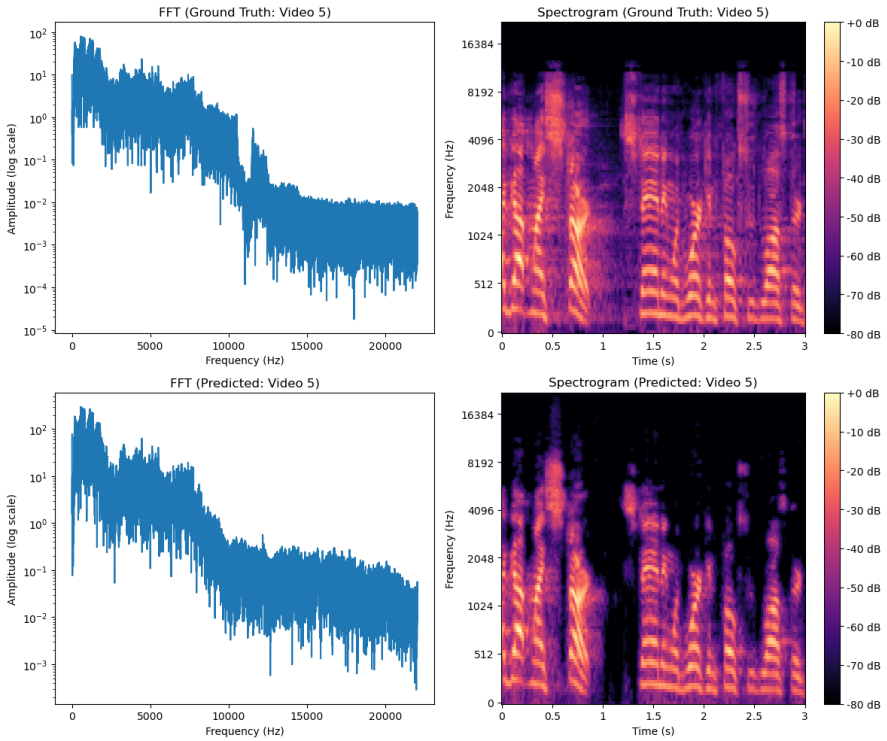


Figure 9-12: FFT and Spectrogram Analysis of Audio Content in Video 5

The FFT and spectrogram plots of the audio content in the inferenced videos closely resemble the ground truth plots across all videos. The model successfully recreates

the audio signals with various frequency components, although slight deviations are observed in the FFT plots. However, the spectrograms remain almost identical across all videos, demonstrating the model's ability to capture the temporal structure of the audio.

In the spectrogram of Video 2 and Video 5, a slight loss of signal is noticable, which can be attributed to the application of the noise reduction algorithm. This algorithm appears to have mistakenly identified some parts of the signal as noise and removed them, leading to this minor discrepancy.

9.4 Comparision with Traditional Codecs

9.4.1 Performance Metrics of Teacher Model

Table 9-1: Video Metrics of Teacher Model

Video	PSNR (dB)	LPIPS	SSIM	Original Size (KiB)	Model Size (KiB)	Compression Ratio
Video 1	42.69	0.05	0.98	1972	9260	0.21
Video 2	35.08	0.17	0.92	5462		0.58
Video 3	21.88	0.39	0.67	6235		0.67
Video 4	26.89	0.41	0.72	14018		1.50
Video 5	32.88	0.21	0.92	12781		1.37

Table 9-2: Audio Metrics of Teacher Model

Video	PSNR (dB)	LSD (dB)	ViSQOL
Video 1	57.57	4.50	3.54
Video 2	57.09	4.28	3.35
Video 3	62.50	7.60	4.51
Video 4	64.33	8.15	3.58
Video 5	46.60	6.89	2.66

Table 9-1 presents the video metrics of the teacher model, showing the PSNR, LPIPS, SSIM, original video size, model size, and compression ratio for five different videos. The PSNR values range from 21.88 dB to 42.69 dB, indicating a notable variation in video quality. Video 1 achieves the highest PSNR at 42.69 dB, demonstrating superior quality compared to the other videos. In contrast, Video 3 has the lowest PSNR of 21.88 dB, reflecting a lower-quality reconstruction. The SSIM values remain high, suggesting

that the model preserves structural details well across all videos, with values ranging from 0.67 to 0.98. The compression ratios vary from 0.21 for Video 1 to 1.50 for Video 4, showing how compression affects the model's storage efficiency and output size.

Table 9-2 shows the audio metrics for the teacher model, including PSNR, LSD, and ViSQOL scores for each video. The PSNR values indicate high fidelity, with Video 3 achieving the highest PSNR at 62.50 dB, while Video 5 has the lowest at 46.60 dB. The LSD values indicate the level of distortion, with Video 1 having the lowest value of 4.50 dB, signifying better quality. The ViSQOL scores, which evaluate perceptual quality, show consistency across most videos, with Video 3 achieving the highest score of 4.51. Overall, the audio metrics indicate that the teacher model performs well, but there is a slight decline in performance for Video 5, particularly in terms of PSNR and ViSQOL.

9.4.2 Performance Metrics of Student Model with Quantization

Quantization impacts the quality of reconstructed images and audio, as measured by PSNR, SSIM and LSD. It also reduces file size. The following table shows the performance metrics for student model before and after quantization to 16-bit.

Table 9-3: Metrics Before and After Quantization of Student Model for Video 1

Metrics	**Value**	
	Student Model	**16-bit Quantization**
PSNR(Frames)(dB)	35.5525	35.3520
SSIM	0.9194	0.9189
LPIPS	0.0794	0.0848
PSNR(Audio)(dB)	18.7929	18.8487
LSD(dB)	4.5930	4.8484
SQNR(dB)	-	39.214 dB
File Size(MiB)	4.96	2.48
Compression Ratio	0.38	0.77

The results from the Table 9-3 demonstrate the effect of quantization on both video and audio quality for Video 1. A significant reduction in file size is observed, dropping from 4.96 MiB to 2.48 MiB, a 50% reduction. Despite this compression, the quality metrics indicate only minor degradation. The PSNR for the video frames decreased slightly from 35.5525 dB to 35.3520 dB, showing that the overall visual quality remains almost unchanged post-quantization. Similarly, the SSIM metric, which measures structural similarity, experienced a marginal decrease from 0.9194 to 0.9189, indicating that the structural integrity of the video is nearly preserved.

The SQNR value at 39.214 dB provides a robust measure of the quantization noise introduced during the compression process. This relatively high value suggests that the noise is effectively controlled, ensuring that the perceived quality remains satisfactory. The audio metrics show a slight improvement, with the PSNR increasing from 18.7929 dB to 18.8487 dB, suggesting that the audio quality benefits slightly from the quantization process. However, the LSD value increased from 4.5930 dB to 4.8484 dB, indicating a slight degradation in the audio frequency spectrum. Nevertheless, this change is minimal and does not significantly affect the overall audio experience. The overall file size reduction highlights the efficiency of quantization in compressing both video and audio, with manageable trade-offs in quality.

Table 9-4: Metrics Before and After Quantization of Student Model for Video 3

Metrics	**Value**	
	Student Model	**16-bit Quantization**
PSNR(Frames)(dB)	28.8588	28.7153
SSIM	0.7579	0.7529
LPIPS	0.4039	0.4126
PSNR(Audio)(dB)	24.1992	24.1786
LSD(dB)	10.7038	10.6928
File Size(MiB)	4.96	2.48
Compression Ratio	1.22	2.45

The results in Table 9-4 show the impact of quantization on Video 2. As with Video 1, the file size is reduced by 50%, from 4.96 MiB to 2.48 MiB, demonstrating the effectiveness of quantization in compressing the video. The PSNR for video frames decreased slightly from 28.8588 dB to 28.7153 dB, indicating minimal visual quality degradation, and the SSIM dropped marginally from 0.7579 to 0.7529, suggesting that the structural integrity is similarly well-preserved.

For the audio, the PSNR showed a negligible change, from 24.1992 dB to 24.1786 dB, and the LSD remained almost the same, moving from 10.7038 dB to 10.6928 dB, further confirming that the audio quality remains stable after quantization. Overall, the results are consistent with those observed in Video 1, indicating a good balance between file size reduction and quality retention.

9.4.3 Performance Metrics of Traditional Codecs

Table 9-5: Metrics for Video Content of Video 1

Video 1 (1972 KiB)							
Codec	**CRF**	**Bitrate (kbps)**	**PSNR (dB)**	**SSIM**	**LPIPS**	**File Size (KiB)**	**Compression Ratio**
H.264/MP3	1	320	61.60	0.99	0.004	429.64	4.59
	23	192	47.24	0.99	0.005	178.59	11.04
	51	64	32.00	0.83	0.23	44.28	44.52
H.265/MP3	1	320	59.53	0.99	0.004	378.32	5.21
	28	192	45.32	0.98	0.02	154.51	12.76
	51	64	33.37	0.85	0.21	45.70	43.14

Table 9-6: Metrics for Audio content of Video 1

Codec	**Bitrate (kbps)**	**PSNR (dB)**	**LSD**	**VISQOL**
MP3	64	35.34	1.97	4.66
	192	43.28	0.42	4.73
	320	64.78	0.10	4.73

Table 9-7: Metrics for Video Content of Video 2

Video 2 (5462 KiB)							
Codec	**CRF**	**Bitrate (kbps)**	**PSNR (dB)**	**SSIM**	**LPIPS**	**File Size (KiB)**	**Compression Ratio**
H.264/MP3	1	320	59.70	0.99	0.003	632	2.67
	23	192	44.94	0.98	0.005	190	8.87
	51	64	27.75	0.78	0.22	12	140.50
H.265/MP3	1	320	52.13	0.99	0.004	434	3.88
	28	192	41.57	0.97	0.01	128	13.17
	51	64	27.83	0.76	0.29	14	120.40

Table 9-8: Metrics for Audio content of Video 2

Codec	**Bitrate (kbps)**	**PSNR (dB)**	**LSD**	**VISQOL**
MP3	64	38.99	0.73	4.72
	192	45.82	0.04	4.73
	320	86.68	0.01	4.73

Table 9-9: Metrics for Video Content of Video 3

Video 3 (6235 KiB)							
Codec	**CRF**	**Bitrate (kbps)**	**PSNR (dB)**	**SSIM**	**LPIPS**	**File Size (KiB)**	**Compression Ratio**
H.264/MP3	1	320	58.39	0.98	0.16	1218.82	5.12
	23	192	42.22	0.96	0.16	161.85	38.52
	51	64	22.17	0.72	0.34	31.65	196.98
H.265/MP3	1	320	56.07	0.99	0.16	861.80	7.23
	28	192	40.20	0.96	0.18	113.02	55.16
	51	64	24.16	0.73	0.35	31.42	198.39

Table 9-10: Metrics for Audio content of Video 3

Codec	**Bitrate (kbps)**	**PSNR (dB)**	**LSD**	**VISQOL**
MP3	64	27.99	4.21	4.64
	192	42.23	0.75	4.73
	320	66.45	0.18	4.73

Table 9-11: Metrics for Video Content of Video 4

Video 4 (14018 KiB)							
Codec	**CRF**	**Bitrate (kbps)**	**PSNR (dB)**	**SSIM**	**LPIPS**	**File Size (KiB)**	**Compression Ratio**
H.264/MP3	1	320	59.10	0.99	0.00	4225.52	3.32
	23	192	41.20	0.98	0.00	1146.30	12.23
	51	64	24.39	0.61	0.32	215.05	65.18
H.265/MP3	1	320	57.61	0.99	0.00	4061.62	3.45
	28	192	37.99	0.95	0.02	930.10	15.07
	51	64	24.96	0.64	0.33	213.73	65.59

Table 9-12: Metrics for Audio content of Video 4

Codec	**Bitrate (kbps)**	**PSNR (dB)**	**LSD**	**VISQOL**
MP3	64	28.25	2.73	4.67
	192	41.05	0.42	4.73
	320	64.38	0.11	4.73

Table 9-13: Metrics for Video Content of Video 5

Video 5 (12781 KiB)							
Codec	**CRF**	**Bitrate (kbps)**	**PSNR (dB)**	**SSIM**	**LPIPS**	**File Size (KiB)**	**Compression Ratio**
H.264/MP3	1	320	57.77	0.99	0.00	869.99	14.69
	23	192	41.28	0.98	0.02	142.82	89.49
	51	64	25.10	0.81	0.39	30.53	418.59
H.265/MP3	1	320	57.04	0.99	0.00	875.66	14.60
	28	192	39.02	0.97	0.06	107.50	118.89
	51	64	26.26	0.84	0.32	32.04	398.91

Table 9-14: Metrics for Audio content of Video 5

Codec	**Bitrate (kbps)**	**PSNR (dB)**	**LSD**	**VISQOL**
MP3	64	38.95	1.59	4.66
	192	50.24	0.22	4.72
	320	82.64	0.07	4.73

The traditional codecs AVC, HEVC, and MP3 demonstrate highly efficient performance in terms of compression, achieving significant reductions in file size while maintaining excellent metric values. The compression ratios, particularly at lower Constant Rate Factor (CRF) values, indicate that these codecs can efficiently compress video and audio data with minimal loss of quality. For video, this efficiency is reflected in the high PSNR, SSIM, and low LPIPS scores, while MP3 achieves similarly strong metric values for audio compression.

In contrast, the teacher model exhibits a compression ratio below 1 for Videos 1, 2, and 3, which means the resulting file size increases instead of reducing. However, it is noteworthy that the output size of the teacher model remains constant at 9,260 kibibytes (KiB), regardless of the original video size. This indicates that, if computational limitations were not a factor, the teacher model could potentially compress a video of several gigabytes to this fixed size, offering a constant compression output.

When comparing quality metrics, the teacher model slightly lags behind the metric scores of HEVC and AVC for CRF values of 28 and 23 in the case of Videos 1, 2, and 5. For Videos 3 and 4, the metrics achieved by the teacher model are comparable to those of HEVC and AVC at CRF 51.

For the audio content, the teacher model performs slightly below the MP3 codec at a bitrate of 320 kbps for Videos 1, 2, 3, and 4. However, for Video 5, the audio content quality is similar to that of the MP3 codec at a bitrate of 192 kbps.

Regarding the student model, its performance on Video 1 is comparable to the traditional codecs at CRF 51 for video, while the audio metric is slightly worse than the MP3 codec at a bitrate of 64 kbps. Similarly, for Video 3, the student model achieves slightly better video metrics compared to the traditional codecs at the lowest CRF setting of 51, while the audio metrics are comparable to those of the MP3 codec at a bitrate of 64 kbps.

9.5 Encoding Results

Encoding was performed using LZMA with xz, which is a lossless compression method. As a result, other metric values such as quality metrics (PSNR, SSIM, LPIPS, etc.) remain unchanged. The observed reduction in file sizes and improvement in compression ratios highlight the efficiency of the LZMA algorithm in reducing storage requirements without compromising the integrity of the original data.

Furthermore, the slight differences in compression ratios between Videos 1 and 3 indicate that the effectiveness of LZMA can vary based on the complexity and redundancy of the content. For Video 1, which may exhibit lower redundancy, the compression ratio improvement is modest. In contrast, Video 3, with potentially higher redundancy, benefits more from the compression, as reflected in the larger improvement in the compression ratio.

Table 9-15: Metrics for Encoding

Metrics	**Video 1**		**Video 3**	
	Before	**After**	**Before**	**After**
File Size (MiB)	2.48	2.43	2.48	2.33
Compression Ratio	0.77	0.79	2.45	2.61

9.6 Quantization Analysis

To analyze the effect of quantization on student model size and its ability to accurately recreate video frames with minimal noise, quantization was performed on the student model of dataset Video 1. Initially, this student model was 4.96MiB. The original model, saved as a .pth file, was stored in the float32 format by default in PyTorch. The model was quantized to various precision levels from int1 to int32. The effect of quantization on the model's performance was evaluated using PSNR, SSIM and SQNR.

Due to technical constraints, only the file sizes for the int8, int16, float16 and int32 quantized models could be exported and included in the size comparison graph. Despite this, these four levels provide sufficient insight into the impact of quantization on model size. The SSIM, PSNR and SQNR graphs include a broader range of bit-widths, highlighting the relationship between quantization precision and the model's ability to accurately reproduce video frames.

To assess PSNR and SSIM, five original images were used as a reference. Each quantized model was used to generate outputs corresponding to these images. The PSNR and SSIM values were calculated by comparing each original image with its corresponding output from the quantized models. For each quantization level, the average PSNR and SSIM values across the five images were computed to provide a comprehensive evaluation of the quantization's effect.

Similarly, for SQNR, the calculation involves comparing the frames before and after quantization. The original unquantized frames were used as reference signals, and the corresponding frames obtained after quantization were used as the output. For each quantization level, the SQNR was calculated by measuring the ratio of the power of the original signal to the power of the quantization noise. The average SQNR values across the five images were computed to provide a comprehensive evaluation of the quantization's effect on signal quality.

The results, including file size comparisons, glssqnr and the average PSNR and SSIM values for each quantization level, are summarized in the graphs Figure 9-13, Figure 9-16, Figure 9-14 and Figure 9-15.

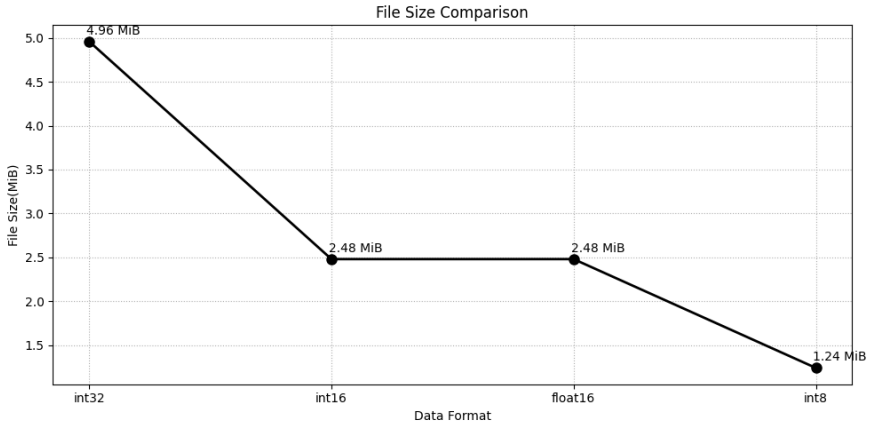


Figure 9-13: File Size Comparision between Quantized Models

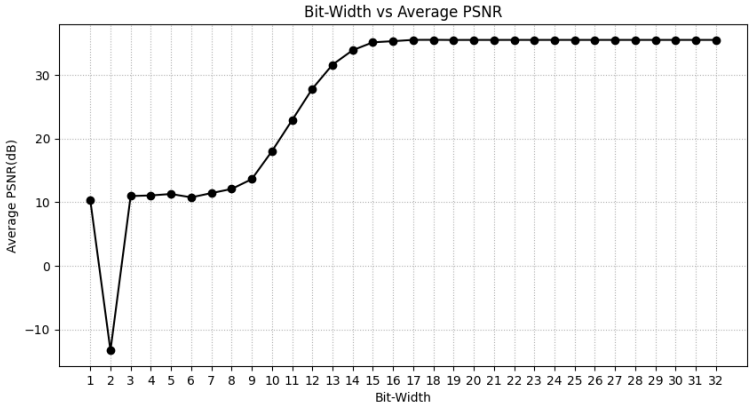


Figure 9-14: Average PSNR for Quantized Models

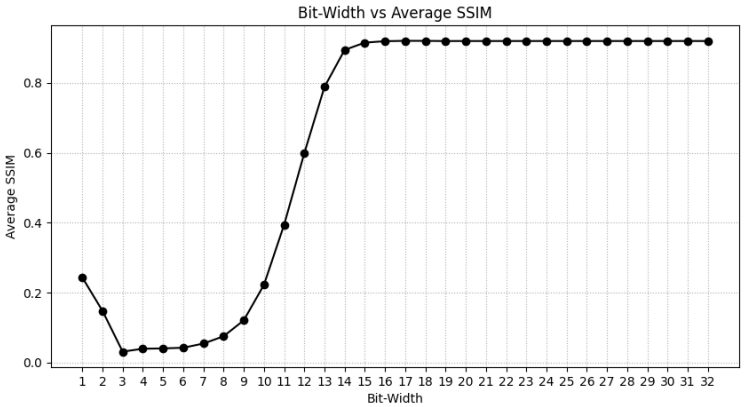


Figure 9-15: Average SSIM for Quantized Models

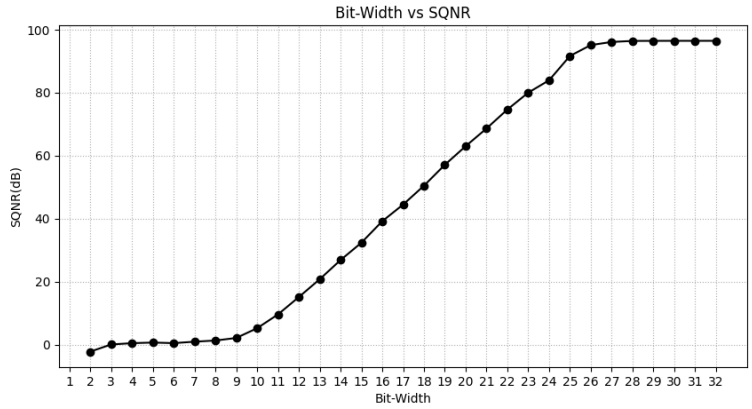


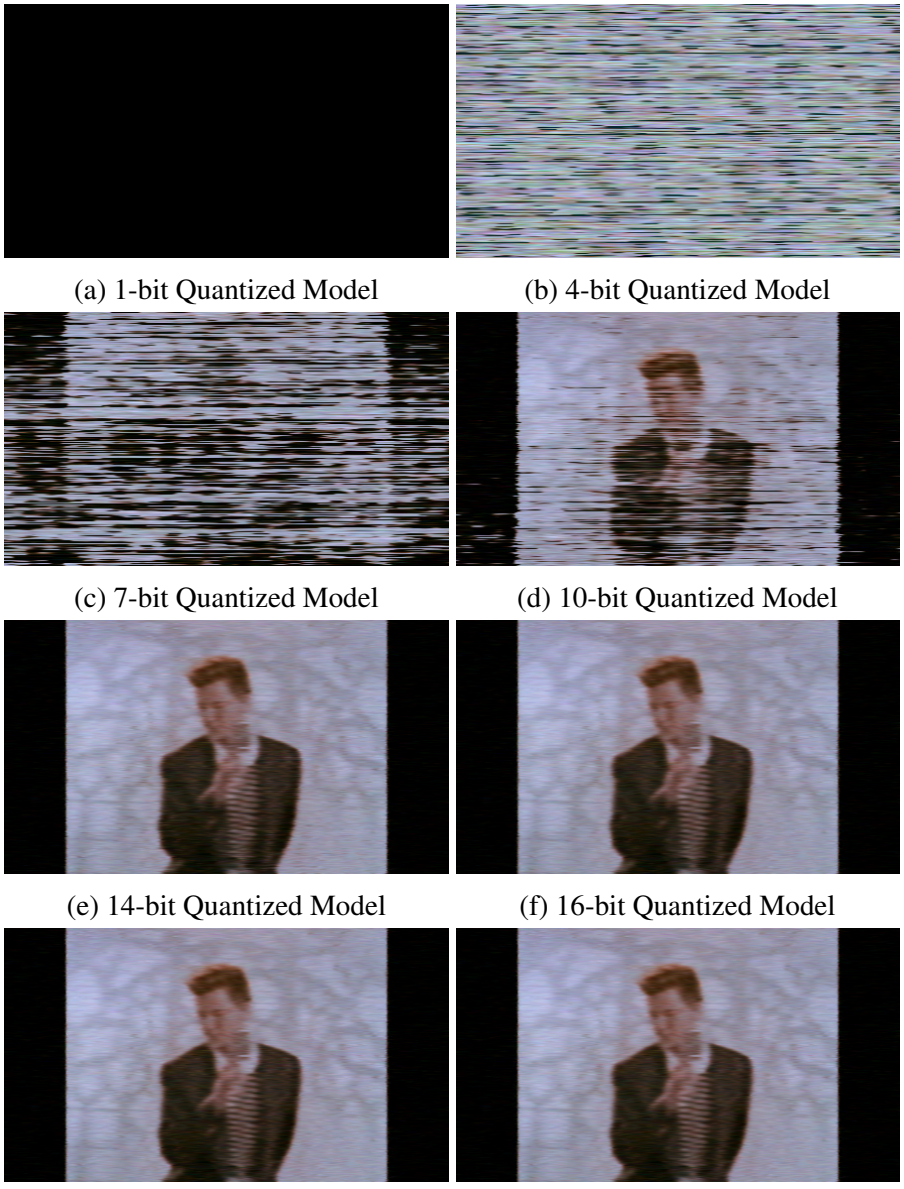
Figure 9-16: SQNR for Quantized Models

The results, as observed from the graphs, show that quantizing the model to 16-bit integer results in reduction in file size. Moreover, both the SSIM and PSNR graphs indicate

that reducing the bit-width initially affects these metrics, but from 16-bit onwards, the values stabilize, and further increase in bit-width has minimal impact on the model's accuracy. The SQNR value for 16-bit quantization is also satisfactory. Given this balance between efficiency and performance, we chose to proceed with the 16-bit integer model for further experimentation and analysis. The results presented from this point onward are based on the 16-bit integer quantized model.

9.6.1 Visualization of Frames of Quantized Models

Here are the 10th frames of the videos generated by the student models for Video 1 after quantization to various bit-widths: 1, 4, 7, 10, 14, 16, 18, and 32.



(a) 1-bit Quantized Model (b) 4-bit Quantized Model

(c) 7-bit Quantized Model (d) 10-bit Quantized Model

(e) 14-bit Quantized Model (f) 16-bit Quantized Model

(g) 18-bit Quantized Model (h) 32-bit Quantized Model

Figure 9-17: Frame 10 predictions at different quantization levels

To provide a clear understanding of the impact of quantization on model output, we present visualizations of frames produced by the model after quantization to various bit-widths: 1, 4, 7, 10, 14, 16, 18, and 32. These frames illustrate how the quality of the output improves as the bit-width increases.

By observing these visualizations, we can clearly see the trade-offs involved in reducing precision. Lower bit-widths (e.g., 1 and 4) result in significant degradation of image quality, making the output highly distorted or unrecognizable. As the bit-width increases to 7, 10, and 14, the images become progressively clearer. Notably, at 16-bit quantization, the output reaches a level where details are sufficiently preserved, offering a balance between efficiency and visual fidelity.

Beyond 16-bit, further increases in bit-width (18 and 32) do not introduce drastic improvements that are perceivable to the naked eye. This observation justifies the choice of 16-bit quantization as an optimal trade-off between model efficiency and output quality.

9.6.2 Visualization of Weight Quantization

The weight quantization process is depicted in Figure 9-18, which consists of 3x3 grid of some sample weights. An example calculation is shown for the first weight in subsection Appendix F:.

i. **Original Weights:** The weights before quantization.

ii. **Quantized Weights:** The weights after quantization into 8-bit integers.

iii. **Integer Representation:** The representation of quantized weights in int8 format.

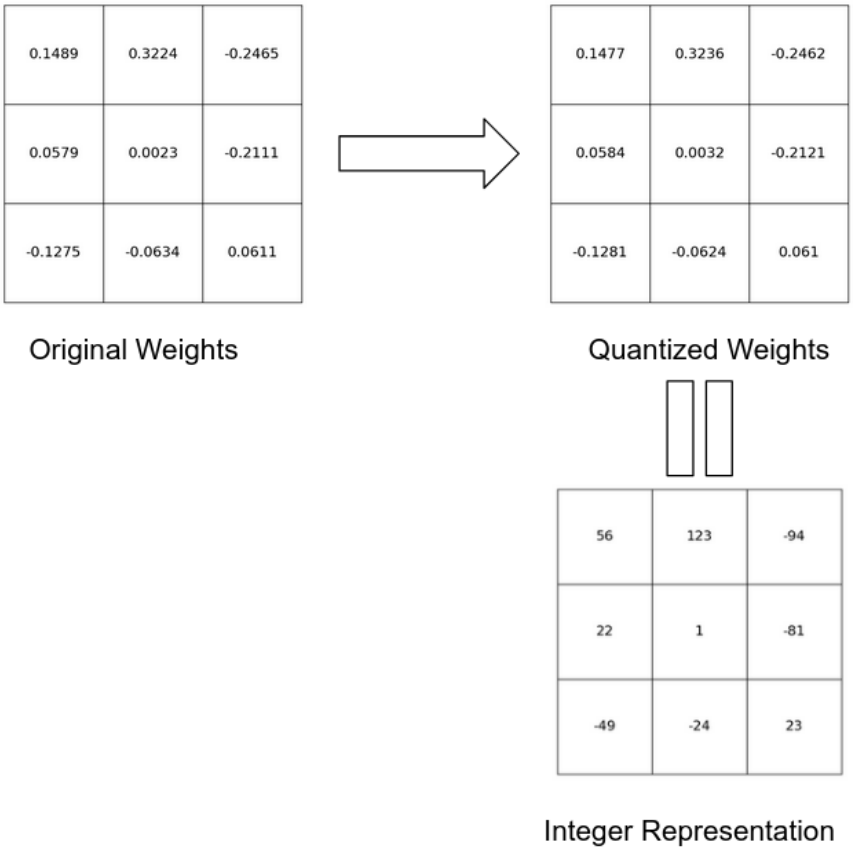


Figure 9-18: Quantization Sample Weights

9.6.3 Histogram of Student Models

Figure 9-19 and Figure 9-20 show the histogram of weights and biases of the Student Model of Video 1 and Video 3.

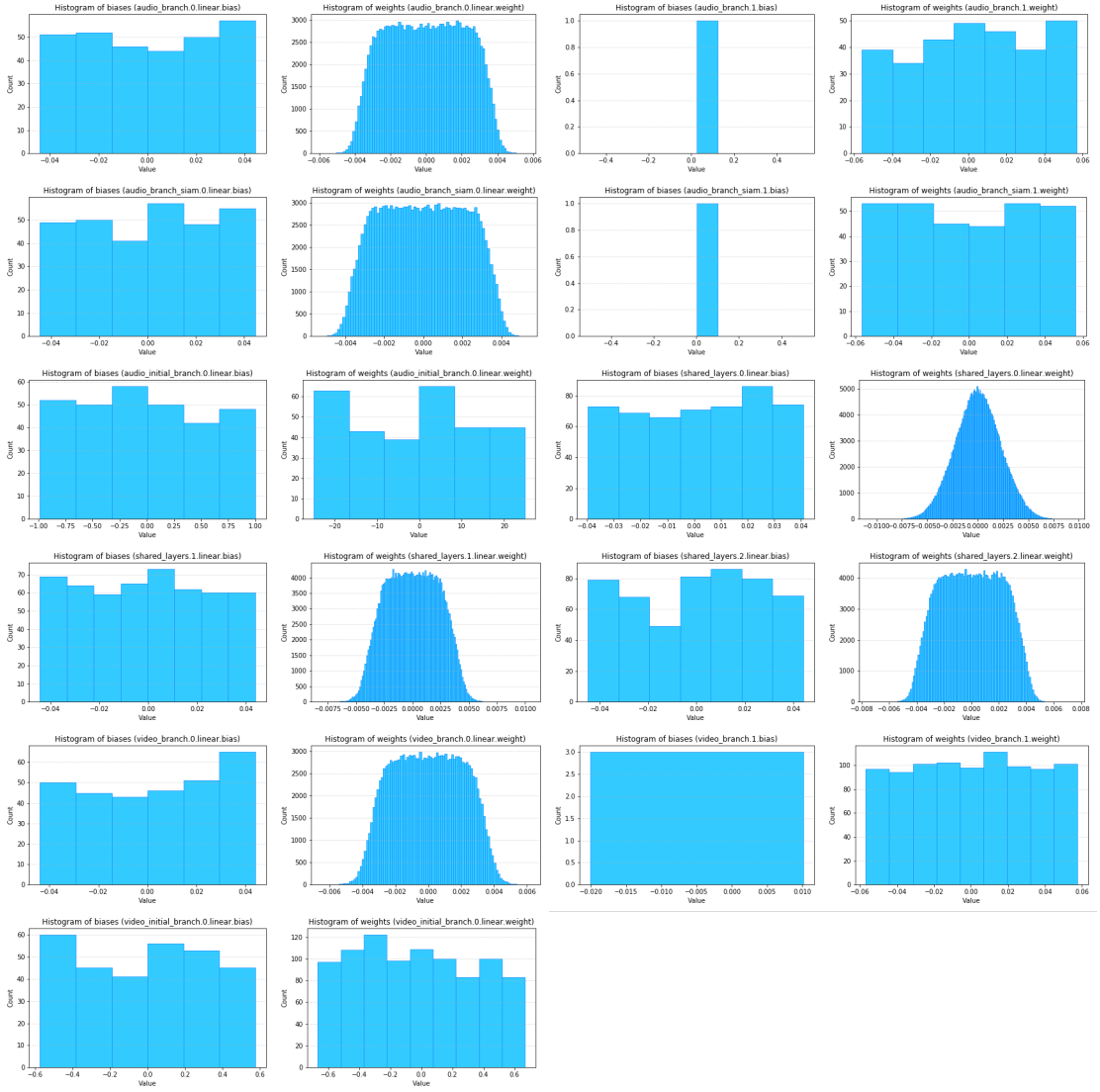


Figure 9-19: Histogram of Student Model:Video 1

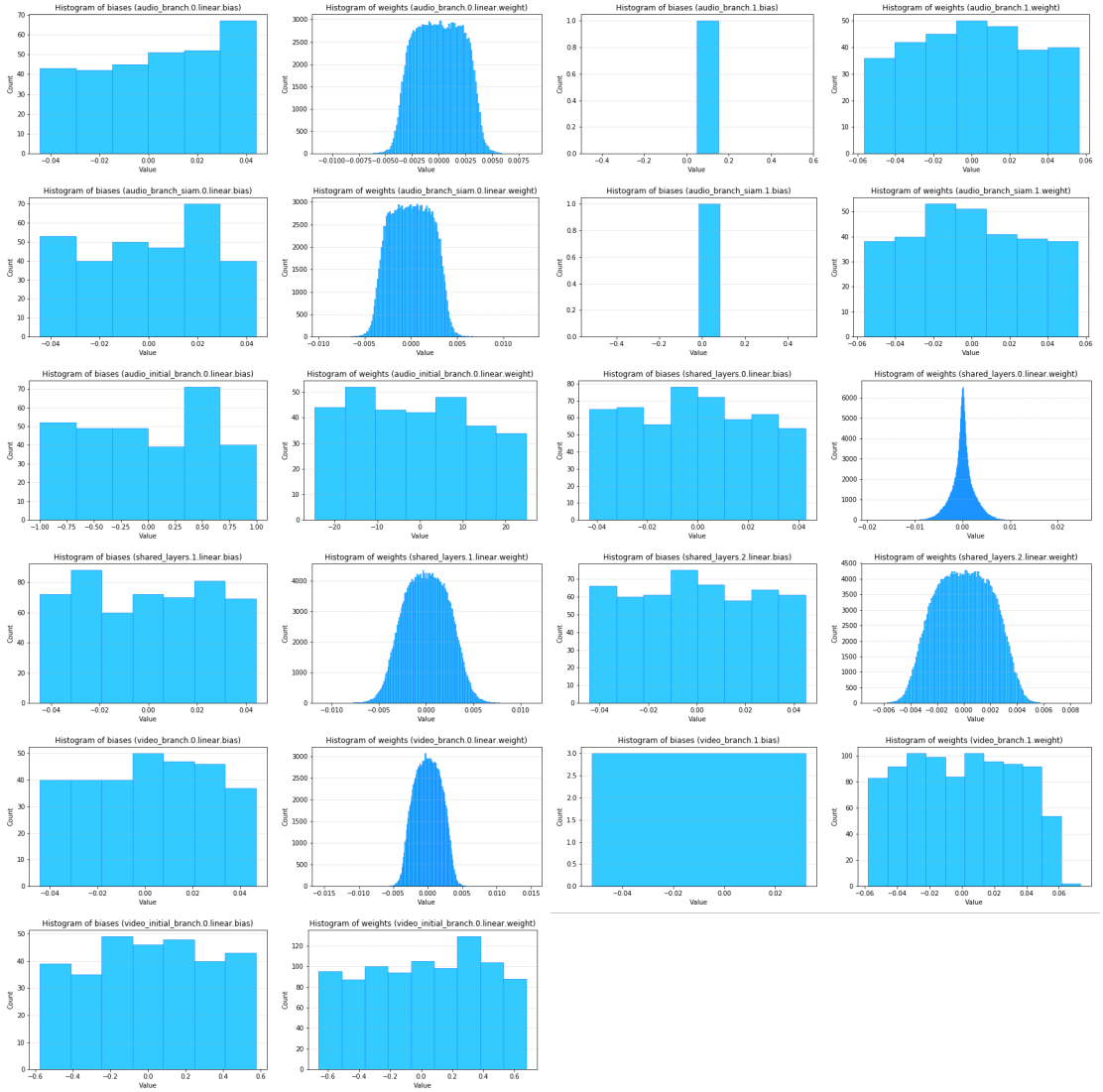


Figure 9-20: Histogram of Student Model:Video 3

These histograms visualize the distributions of weights and biases for two student models across different layers and components, such as audio and video branches, shared layers, and initial layers. The distributions are diverse, with weights generally exhibiting Gaussian-like shapes centered around zero. Meanwhile, biases show more uniformly or slightlys skewed distribution.

9.7 User Interface

The developed system enables efficient video representation using Implicit Neural Representations (INR) and compresses the resulting models using quantization, knowledge distillation, and encoding. The web-based platform facilitates the entire process, from video training to compressed model transfer and inference. The results presented here demonstrate the complete workflow of video representation, compression, and reconstruction.

9.7.1 Video-to-INR Representation and Compression

The process begins with the user uploading a video to the system for INR-based representation. Once uploaded, the backend processes the video by converting it into an INR model. This INR model is then compressed using quantization and knowledge distillation techniques to reduce its size while maintaining essential features.

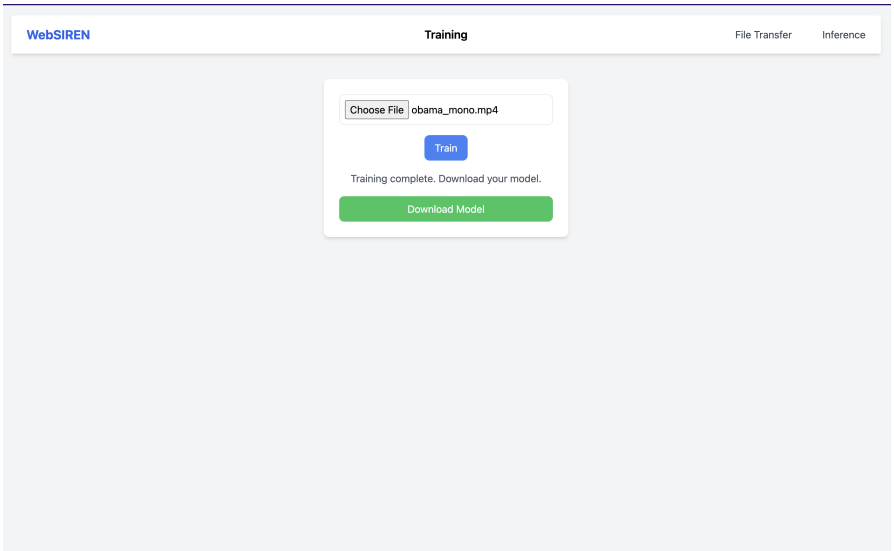


Figure 9-21: Video uploaded for INR representation and compression

9.7.2 Compressed INR Model Transfer and Storage

After compression, the INR model is available for transfer or storage. The system provides a peer-to-peer model transfer mechanism using WebRTC, allowing users to share compressed INR models efficiently within the same network. This minimizes dependency on external storage and ensures fast transmission.

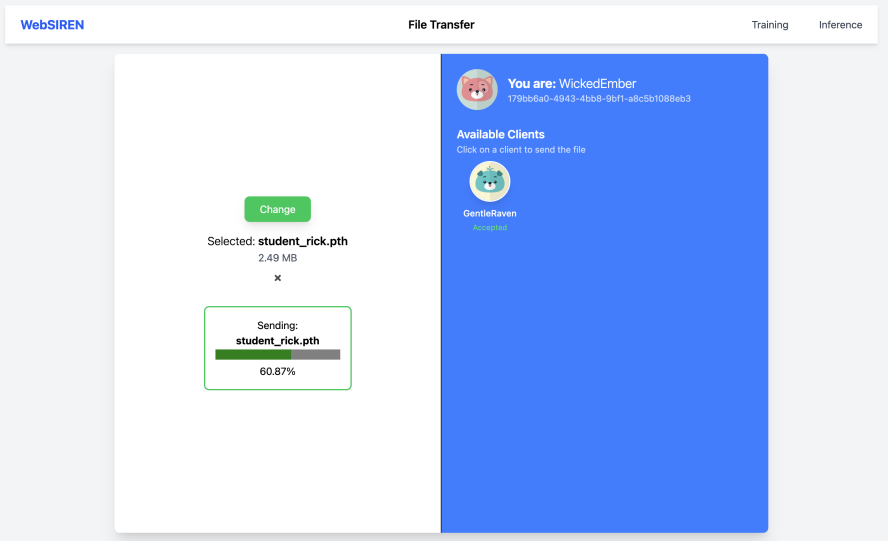


Figure 9-22: Compressed INR model being transferred in the network

9.7.3 Model Inference and Video Reconstruction

Once the compressed INR model is received, it undergoes inference to reconstruct the original video. The system uses a trained model to generate the video representation, aiming to retain visual quality despite the compression. The reconstructed video is then played within the web interface.

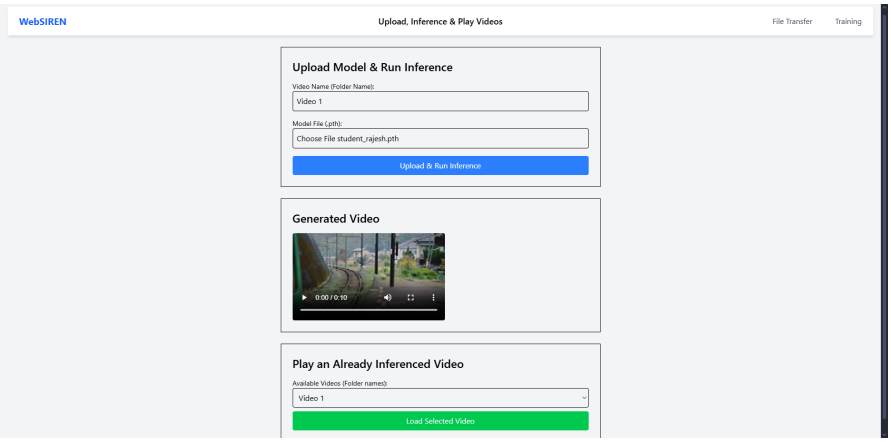


Figure 9-23: Inference of the INR model to reconstruct the video

The final step in the process is the playback of the reconstructed video. Users can view the inferred video directly on the web platform, comparing it with the original input to evaluate the quality preservation after compression and inference.

The results demonstrate the system's ability to efficiently represent, compress, transfer, and reconstruct videos using INR models. The integration of WebRTC for model

transfer and FastAPI for inference enables real-time processing and low-latency video retrieval. The compressed INR models significantly reduce storage and bandwidth requirements while maintaining video quality, making this approach highly effective for scalable video representation and compression.

10 FUTURE ENHANCEMENTS

The project has successfully demonstrated the potential of SIREN as a unified model for audio-video compression. However, several areas remain for future exploration and enhancement. These include:

i. **Dynamic Content Representation:** The current model has shown promising results for static content but struggles with dynamic content. Future work could focus on enhancing the model's ability to represent dynamic content, such as videos with higher frames per second (fps).

ii. **Optimized Quantization:** While the current model achieves competitive results with 16-bit quantization, further optimization of the quantization process could improve compression efficiency without sacrificing quality.

11 APPENDICES

Appendix A: Project Schedule

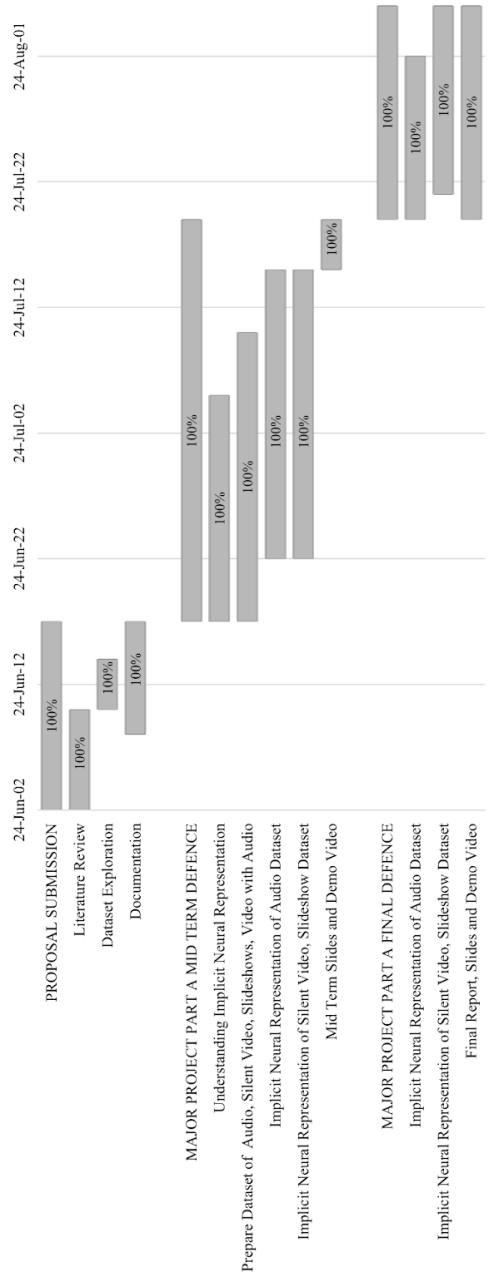


Figure A-1: Gantt Chart for Major Project Part A

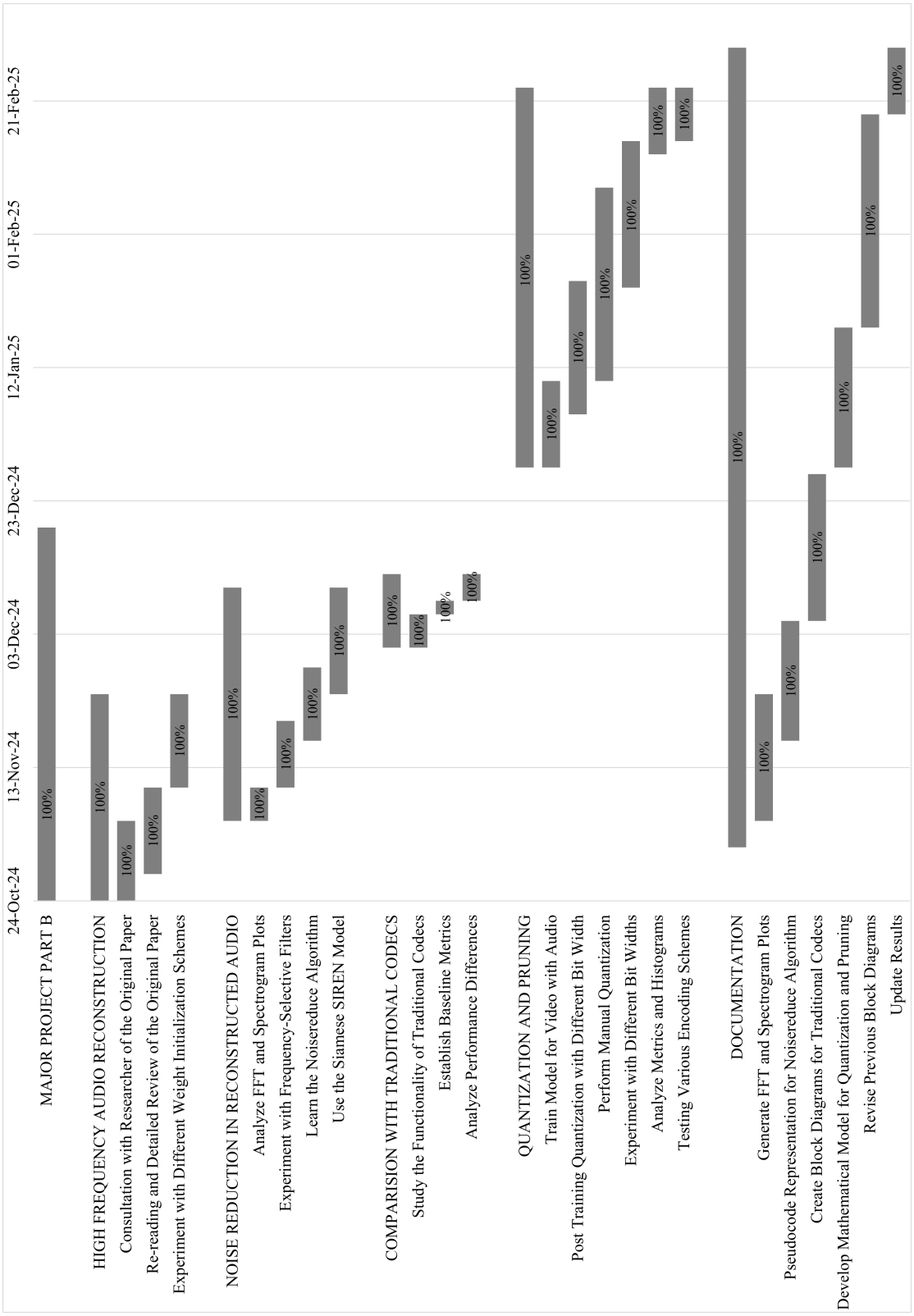


Figure A-2: Gantt Chart for Major Project Part B

Appendix B: Forward and Backward Propagation

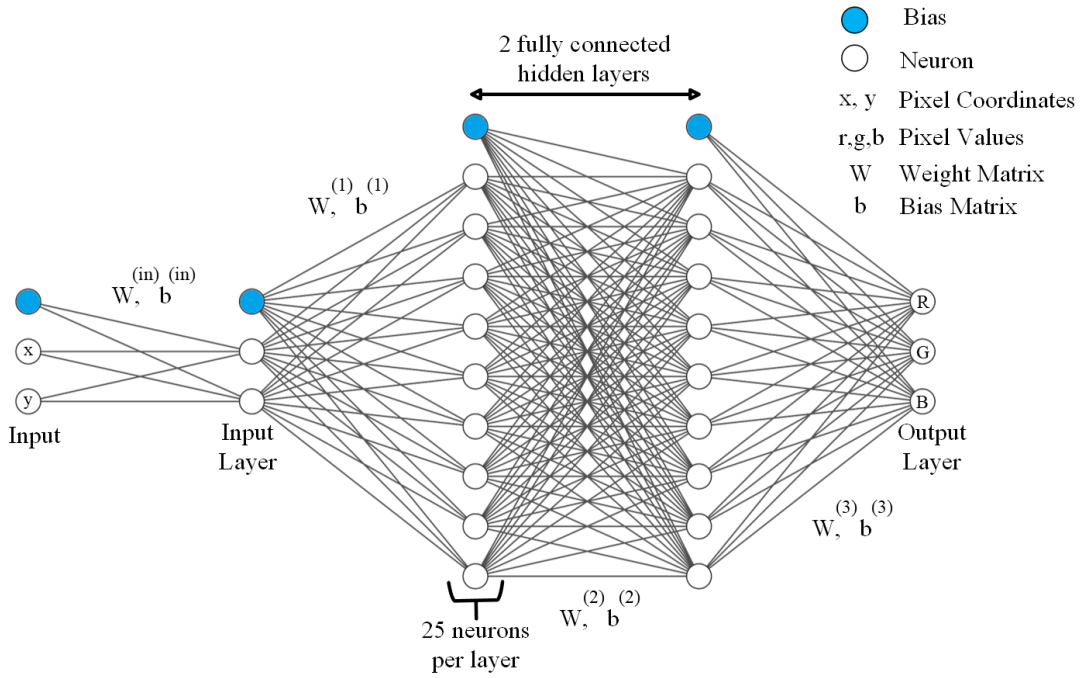


Figure B-1: Neural Network Architecture for 32 by 32 image

The below derivations have been done using Figure B-1 as reference.

B.1 Forward Propagation

Step 1: Input Layer

- Input Vector $\mathbf{x}$: Dimension 2×1
- Weight Matrix $\mathbf{W}^{(in)}$: Dimension 2×2
- Bias Vector $\mathbf{b}^{(in)}$: Dimension 2×1
- Calculation:

$$\mathbf{z}^{(in)} = \mathbf{W}^{(in)}\mathbf{x} + \mathbf{b}^{(in)} \tag{B-1}$$

$$\mathbf{a}^{(in)} = \mathbf{z}^{(in)} \tag{B-2}$$

  Where $\mathbf{z}^{(in)}$ and $\mathbf{a}^{(in)}$ both have dimensions 2×1.

Step 2: Hidden Layer 1

- Input Vector $\mathbf{x}$: Dimension 2×1
- Weight Matrix $\mathbf{W}^{(1)}$: Dimension 25×2

- Bias Vector $\mathbf{b}^{(1)}$: Dimension 25×1
- Calculation:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{B-3}$$

$$\mathbf{a}^{(1)} = \sin(\mathbf{z}^{(1)}) \tag{B-4}$$

Where $\mathbf{z}^{(1)}$ and $\mathbf{a}^{(1)}$ both have dimensions 25×1.

Step 3: Hidden Layer 2

- Weight Matrix $\mathbf{W}^{(2)}$: Dimension 25×25
- Bias Vector $\mathbf{b}^{(2)}$: Dimension 25×1
- Calculation:

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \tag{B-5}$$

$$\mathbf{a}^{(2)} = \sin(\mathbf{z}^{(2)}) \tag{B-6}$$

Both $\mathbf{z}^{(2)}$ and $\mathbf{a}^{(2)}$ are 25×1.

Step 4: Output Layer

- Weight Matrix $\mathbf{W}^{(3)}$: Dimension 3×25
- Bias Vector $\mathbf{b}^{(3)}$: Dimension 3×1
- Calculation:

$$\mathbf{z}^{(3)} = \mathbf{W}^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)} \tag{B-7}$$

$$\mathbf{a}^{(3)} = \sin(\mathbf{z}^{(3)}) \tag{B-8}$$

Where $\mathbf{z}^{(3)}$ and $\mathbf{a}^{(3)}$ both have dimensions 3×1.

B.2 Backpropagation

Step 1: Output Layer

i. **Gradient of Loss with respect to output activations** ($\mathbf{a}^{(3)}$):
Given the Mean Squared Error (MSE) loss function:

$$\text{Loss} = \frac{1}{2}\sum_{i=1}^{3}(y_i - a_i^{(3)})^2 \tag{B-9}$$

Differentiate Loss w.r.t. $\mathbf{a}^{(3)}$:

$$\frac{\partial \text{Loss}}{\partial a_i^{(3)}} = a_i^{(3)} - y_i \tag{B-10}$$

This will result in a vector $\mathbf{a}^{(3)} - \mathbf{y}$.

ii. Gradient with respect to $\mathbf{z}^{(3)}$ (using chain rule and recognizing $\mathbf{a}^{(3)} = \sin(\mathbf{z}^{(3)})$):

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}} = \frac{\partial \text{Loss}}{\partial \mathbf{a}^{(3)}} \cdot \frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{z}^{(3)}}$$

Since $\frac{\partial a_i^{(3)}}{\partial z_i^{(3)}} = \cos(z_i^{(3)})$, the gradient becomes:

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}} = (\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)}) \tag{B-11}$$

iii. Gradient with respect to weights $\mathbf{W}^{(3)}$ and biases $\mathbf{b}^{(3)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(3)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}} \cdot \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{W}^{(3)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(3)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}} \mathbf{a}^{(2)T}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(3)}} = \left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)}) \right) \mathbf{a}^{(2)T} \tag{B-12}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(3)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}} \cdot \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{b}^{(3)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(3)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(3)}} = (\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)}) \tag{B-13}$$

Step 2: Hidden Layer 2

i. Gradient with respect to activations $\mathbf{a}^{(2)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(2)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}} \cdot \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{a}^{(2)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(2)}} = \mathbf{W}^{(3)T} \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(3)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(2)}} = \mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right) \tag{B-14}$$

ii. Gradient with respect to $\mathbf{z}^{(2)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}} = \frac{\partial \text{Loss}}{\partial \mathbf{a}^{(2)}} \cdot \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}} = \frac{\partial \text{Loss}}{\partial \mathbf{a}^{(2)}} \cdot \cos(\mathbf{z}^{(2)})$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}} = \left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)}) \tag{B-15}$$

iii. Gradient with respect to weights $\mathbf{W}^{(2)}$ and biases $\mathbf{b}^{(2)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}} \cdot \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}} \mathbf{a}^{(1)T}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(2)}} = \left(\left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)})\right) \mathbf{a}^{(1)T} \tag{B-16}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(2)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}} \cdot \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{b}^{(2)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(2)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(2)}} = \left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)}) \tag{B-17}$$

Step 3: Hidden Layer 1

i. Gradient with respect to activations $\mathbf{a}^{(1)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(1)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}} \cdot \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(1)}} = \mathbf{W}^{(2)T} \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(2)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(1)}} = \mathbf{W}^{(2)T}\left(\left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)})\right) \tag{B-18}$$

ii. Gradient with respect to $\mathbf{z}^{(1)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \text{Loss}}{\partial \mathbf{a}^{(1)}} \cdot \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \text{Loss}}{\partial \mathbf{a}^{(1)}} \cdot \cos(\mathbf{z}^{(1)})$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}} = \left(\mathbf{W}^{(2)T}\left(\left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)})\right)\right) \cdot \cos(\mathbf{z}^{(1)}) \quad \text{(B-19)}$$

iii. Gradient with respect to weights $\mathbf{W}^{(1)}$ and biases $\mathbf{b}^{(1)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}} \cdot \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}} \mathbf{x}^T$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(1)}} = \left(\left(\mathbf{W}^{(2)T}\left(\left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)})\right)\right) \cdot \cos(\mathbf{z}^{(1)})\right) \mathbf{x}^T \quad \text{(B-20)}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}} \cdot \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{b}^{(1)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(1)}} = \left(\mathbf{W}^{(2)T}\left(\left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)})\right)\right) \cdot \cos(\mathbf{z}^{(1)}) \quad \text{(B-21)}$$

Step 4: Input Layer

i. Gradient with respect to activations $\mathbf{a}^{(in)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(in)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}} \cdot \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{a}^{(in)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(in)}} = \mathbf{W}^{(1)T} \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(1)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{a}^{(in)}} = \mathbf{W}^{(1)T}\left(\left(\mathbf{W}^{(2)T}\left(\left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)})\right)\right) \cdot \cos(\mathbf{z}^{(1)})\right) \quad \text{(B-22)}$$

ii. Gradient with respect to $\mathbf{z}^{(in)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(in)}} = \frac{\partial \text{Loss}}{\partial \mathbf{a}^{(in)}} \cdot \frac{\partial \mathbf{a}^{(in)}}{\partial \mathbf{z}^{(in)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(in)}} = \frac{\partial \text{Loss}}{\partial \mathbf{a}^{(in)}} \cdot \cos(\mathbf{z}^{(in)})$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{z}^{(in)}} = \mathbf{W}^{(1)T}\left(\left(\mathbf{W}^{(2)T}\left(\left(\mathbf{W}^{(3)T}\left((\mathbf{a}^{(3)} - \mathbf{y}) \cdot \cos(\mathbf{z}^{(3)})\right)\right) \cdot \cos(\mathbf{z}^{(2)})\right)\right) \cdot \cos(\mathbf{z}^{(1)})\right) \cdot \cos(\mathbf{z}^{(in)}) \quad \text{(B-23)}$$

iii. Gradient with respect to weights $\mathbf{W}^{(in)}$ and biases $\mathbf{b}^{(in)}$:

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(in)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(in)}} \cdot \frac{\partial \mathbf{z}^{(in)}}{\partial \mathbf{W}^{(in)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{W}^{(in)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(in)}} \mathbf{x}^T \quad \text{(B-24)}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(in)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(in)}} \cdot \frac{\partial \mathbf{z}^{(in)}}{\partial \mathbf{b}^{(in)}}$$

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(in)}} = \frac{\partial \text{Loss}}{\partial \mathbf{z}^{(in)}} \quad \text{(B-25)}$$

Appendix C: Arcsine Distribution on [-1, 1]

Probability Density Function (PDF)

The PDF of the arcsine distribution on the interval $(-1, 1)$ is given by:

$$f(x) = \frac{1}{\pi\sqrt{1-x^2}} \tag{C-1}$$

Cumulative Distribution Function (CDF)

The cumulative distribution function (CDF), $F(x)$, is defined as the integral of the probability density function (PDF) from the lower bound of the interval to x:

$$F(x) = \int_{-1}^{x} f(t)\,dt \tag{C-2}$$

Given the PDF for the arcsine distribution:

$$f(x) = \frac{1}{\pi\sqrt{1-x^2}}$$

we need to integrate this PDF from -1 to x:

$$F(x) = \int_{-1}^{x} \frac{1}{\pi\sqrt{1-t^2}}\,dt$$

To solve this integral, we use the substitution $t = \sin\theta$.
Thus, $dt = \cos\theta\,d\theta$ and $\sqrt{1-t^2} = \sqrt{1-\sin^2\theta} = \cos\theta$.
When $t = -1$, $\theta = -\frac{\pi}{2}$.
When $t = x$, $\theta = \arcsin x$.
Therefore, the integral becomes:

$$F(x) = \int_{-\frac{\pi}{2}}^{\arcsin x} \frac{1}{\pi\cos\theta}\cos\theta\,d\theta = \int_{-\frac{\pi}{2}}^{\arcsin x} \frac{1}{\pi}\,d\theta$$

$$F(x) = \frac{1}{\pi}\int_{-\frac{\pi}{2}}^{\arcsin x} d\theta = \frac{1}{\pi}\left[\theta\right]_{-\frac{\pi}{2}}^{\arcsin x}$$

$$F(x) = \frac{1}{\pi}\left(\arcsin x - \left(-\frac{\pi}{2}\right)\right)$$

$$F(x) = \frac{1}{\pi} \arcsin x + \frac{1}{\pi} \cdot \frac{\pi}{2}$$

$$F(x) = \frac{1}{\pi} \arcsin x + \frac{1}{2}$$

Thus, the CDF of the arcsine distribution on $[-1, 1]$ is:

$$F(x) = \frac{1}{\pi} \arcsin(x) + \frac{1}{2} \tag{C-3}$$

Mean

The mean μ of the distribution is given by:

$$\mu = \int_{-1}^{1} x f(x)\, dx = \int_{-1}^{1} x \cdot \frac{1}{\pi\sqrt{1-x^2}}\, dx \tag{C-4}$$

This integral evaluates to zero because $x \cdot f(x)$ is an odd function integrated over a symmetric interval around zero:

$$\mu = 0 \tag{C-5}$$

Variance

The variance σ^2 is calculated as:

$$\sigma^2 = \mathbf{E}[X^2] - (\mathbf{E}[X])^2 \tag{C-6}$$

Since $\mathbf{E}[X] = 0$, we have:

$$\mathbf{E}[X^2] = \int_{-1}^{1} x^2 f(x)\, dx = \int_{-1}^{1} x^2 \cdot \frac{1}{\pi\sqrt{1-x^2}}\, dx$$

By symmetry and properties of the arcsine distribution, this integral evaluates to:

$$\mathbf{E}[X^2] = \frac{1}{2}$$

Thus, the variance is:

$$\sigma^2 = \frac{1}{2} - 0^2 = \frac{1}{2} \tag{C-7}$$

Appendix D: IEEE 754 Floating Point Representation

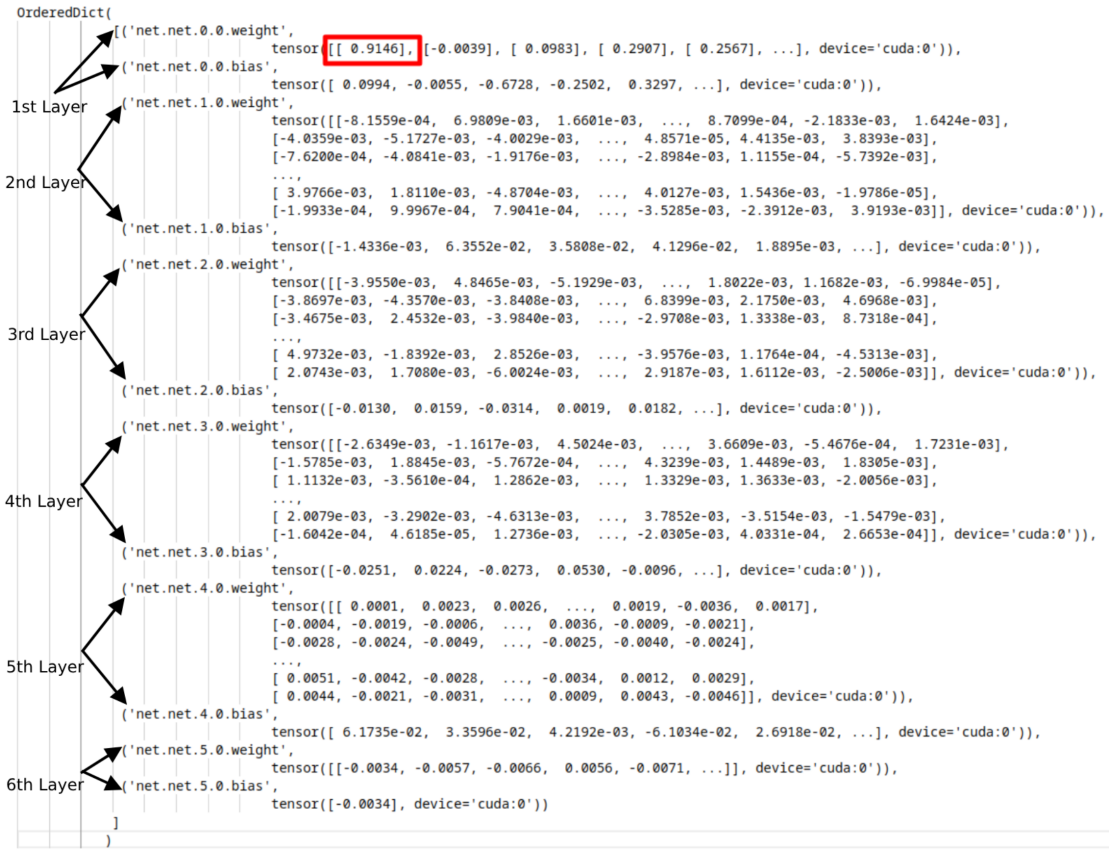


Figure D-1: Final Weights for Audio 1

Table D-1: Derivation and PyTorch Notation for Weights and Biases

Layer		**Derivation Notation**		**PyTorch Notation**	
From	**To**	**Weights**	**Biases**	**Weights**	**Biases**
Input	Input Layer	W_{in}	b_{in}	`net.net.0.0.weight`	`net.net.0.0.bias`
Input Layer	Hidden 1	$W^{(1)}$	$b^{(1)}$	`net.net.1.0.weight`	`net.net.1.0.bias`
Hidden 1	Hidden 2	$W^{(2)}$	$b^{(2)}$	`net.net.2.0.weight`	`net.net.2.0.bias`
Hidden 2	Hidden 3	$W^{(3)}$	$b^{(3)}$	`net.net.3.0.weight`	`net.net.3.0.bias`
Hidden 3	Hidden 4	$W^{(4)}$	$b^{(4)}$	`net.net.4.0.weight`	`net.net.4.0.bias`
Hidden 4	Output	W_{out}	b_{out}	`net.net.5.0.weight`	`net.net.5.0.bias`

To represent the number 0.9146 in IEEE 754 single precision floating-point format:

1. **Convert to Binary:**

$$0.9146_{10} \approx 0.11101010001000110011101 0_2$$

2. **Normalize the Binary Number:**

$$0.11101010001000110011101010_2 = 1.11010100010001100111010 \times 2^{-1}$$

3. **Bias the Exponent:**

$$\text{Exponent} = -1 + 127 = 126$$

In binary:

$$126_{10} = 01111110_2$$

4. **IEEE 754 Representation:**

Sign Bit	Exponent	Mantissa
0	01111110	11010100010001100111010

(D-1)

5. **Convert IEEE 754 Representation to Decimal:**

- **Sign Bit**: 0 (indicating a positive number)
- **Exponent**: 01111110_2 in binary, which is 126 in decimal.
- **Bias**: 127
- **Actual Exponent**: $126 - 127 = -1$
- **Mantissa**: $1.11010100010001100111010_2$ (implicit leading 1 added)

Calculate the Decimal Value:

$$\text{Value} = (-1)^{\text{sign}} \times 1.\text{Mantissa} \times 2^{\text{Actual Exponent}}$$

$$\text{Value} = (-1)^0 \times 1.11010100010001100111010_2 \times 2^{-1}$$

Convert $1.11010100010001100111010_2$ to decimal:

$$\begin{aligned}1.11010100010001100111010_2 &= 1+0.5+0.25+0.0625+0.015625+9.765625\times10^{-4}\\&\quad+6.103515625\times10^{-5}+3.051757813\times10^{-5}+\\&\quad3.814697266\times10^{-6}+1.907348633\times10^{-6}\\&\quad+9.536743164\times10^{-7}+2.384185791\times10^{-7}\\&\quad=1.829200029373168945312\end{aligned}$$

$$\begin{aligned}\text{Value} &= 1.829200029373168945312\times2^{-1}\\&= 1.829200029373168945312\times0.5\\&= 0.914600014686584472656\end{aligned}\tag{D-2}$$

The discrepancy arises because the true value of 0.9146 cannot be exactly represented in binary. The value stored in IEEE 754 format is 0.914600014686584472656, which is very close to the original value 0.9146, with the difference due to rounding in the conversion process.

Appendix E: Custom Floating-Point Format

i. **FP32:** FP32 (32-bit Floating Point) is the standard floating-point format widely used in many applications, offering a good balance of precision and dynamic range. It consists of 1 sign bit, 8 exponent bits, and 23 mantissa bits, offering a high degree of precision.

Sign Bit	Exponent	Mantissa
1bit	8bits	23bits

FP32 is commonly used in applications where high precision is necessary, such as scientific computing, 3D graphics, and high-performance computing.

Calculation of Minimum and Maximum values:

$$\text{Value} = (-1)^{\text{sign}} \times 1.\text{Mantissa} \times 2^{\text{Exponent - 127}}$$

For maximum value:
Sign = 0
Exponent = 11111110 (All 1s is reserved for NaN values)
Mantissa = 11111111111111111111

$$\begin{aligned}\text{Maximum Value} &= (-1)^0 \times 1.11111111111111111111111 \times 2^{11111110} \\ &\approx (1) \times 1.999999880 \times 2^{254-127} \\ &\approx 1.999999880 \times 2^{127} \\ &\approx 3.4028235 \times 10^{38}\end{aligned}$$

For minimum value:
The minimum value that can be represented in FP32 is a subnormal number. To indicate a subnormal number, all 0s are stored in Exponent.
Sign = 1
Exponent = 00000000 (Indicating a subnormal number)
Mantissa = 00000000000000000000001

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 0.00000000000000000000001 \times 2^{-126} \\ &= (-1) \times 2^{-23} \times 2^{-126} \\ &= 2^{-149} \\ &\approx -1.401298 \times 10^{-45}\end{aligned}$$

The minimum normal value can also be calculated as :
Sign = 1

Exponent = 00000001 (Smallest non-zero exponent for normalized numbers)
Mantissa = 00000000000000000000000

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 1 \times 2^{-126} \\ &= (-1) \times (1) \times 2^{-126} \\ &= -2^{-126} \\ &\approx -1.1754944 \times 10^{-38}\end{aligned}$$

Table E-1: Advantages and Disadvantages of FP32

Advantages	**Disadvantages**
High Precision: Suitable for applications requiring high numerical accuracy.	Computationally Expensive: Requires more memory and processing power.
Wide Dynamic Range: Supports a wide range of values.	Slower Operations: More resource-intensive than lower precision formats.

ii. FP16: FP16 (16-bit Floating Point) is a lower-precision floating-point format that sacrifices some precision in favor of reducing memory usage and improving computation speed. It uses 1 sign bit, 5 exponent bits, and 10 mantissa bits.

Sign Bit	Exponent	Mantissa
1bit	5bits	10bits

FP16 is commonly used in machine learning tasks to speed up training while reducing memory consumption.

Calculation of Minimum and Maximum values:

$$\text{Value} = (-1)^{\text{sign}} \times 1.\text{Mantissa} \times 2^{\text{Exponent - 15}}$$

For maximum value:
Sign = 0
Exponent = 11110 (All 1s is reserved for NaN values)

Mantissa = 1111111111

$$\begin{aligned}\text{Maximum Value} &= (-1)^0 \times 1.1111111111 \times 2^{11110} \\ &\approx (1) \times 1.999999880 \times 2^{30-15} \\ &\approx 1.999999880 \times 2^{15} \\ &\approx 65504\end{aligned}$$

For minimum value:
The minimum value that can be represented in FP16 is a subnormal number. To indicate a subnormal number, all 0s are stored in Exponent.
Sign = 1
Exponent = 00000 (Indicating a subnormal number)
Mantissa = 0000000001

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 0.0000000001 \times 2^{-14} \\ &= (-1) \times 2^{-10} \times 2^{-14} \\ &= 2^{-24} \\ &\approx -5.960464477 \times 10^{-8}\end{aligned}$$

The minimum normal value can also be calculated as :
Sign = 1
Exponent = 00001 (Smallest non-zero exponent for normalized numbers)
Mantissa = 0000000000

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 1 \times 2^{-14} \\ &= (-1) \times (1) \times 2^{-15} \\ &= -2^{-15} \\ &\approx -3.051757 \times 10^{-5}\end{aligned}$$

Table E-2: Advantages and Disadvantages of FP16

Advantages	**Disadvantages**
Reduced Memory Usage: Saves memory space, especially for large models.	Lower Precision: Less accurate for some tasks requiring high precision.
Faster Computation: Reduces processing time in many tasks.	Limited Dynamic Range: Smaller exponent range than FP32.

iii. FP8: FP8 (8-bit Floating Point) is an extremely low-precision format that is

mainly used in specialized applications where extreme memory efficiency is required. It uses 1 sign bit, 4 exponent bits, and 3 mantissa bits.

Sign Bit	Exponent	Mantissa
1bit	4bits	3bits

The above example is of E4M3 variant. It also has another variant named E5M2 which has 1 sign bit, 5 exponent bits and only 2 mantissa bits.

FP8 is used in contexts like model quantization or inference tasks, where the trade-off between precision and speed can be adjusted.

Calculation of Minimum and Maximum values(E4M3):

$$\text{Value} = (-1)^{\text{sign}} \times 1.\text{Mantissa} \times 2^{\text{Exponent - 7}}$$

For maximum value:

Sign = 0

Exponent = 1111

Mantissa = 110

$$\begin{aligned}\text{Maximum Value} &= (-1)^0 \times 1.110 \times 2^{1110} \\ &= (1) \times 1.75 \times 2^{15-7} \\ &= 1.75 \times 2^8 \\ &= 1.75 \times 128 \\ &= 448\end{aligned}$$

For minimum value:

The minimum value that can be represented in FP8 is a subnormal number. To indicate a subnormal number, all 0s are stored in Exponent.

Sign = 1

Exponent = 0000 (All 1s is reserved for NaN values)

Mantissa = 111

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 0.111 \times 2^{-6} \\ &= (-1) \times 0.875 \times 2^{-6} \\ &= -1.875 \times 2^{-6} \\ &= -1.875 \times 128 \\ &\approx -1.36 \times 10^{-2}\end{aligned}$$

The minimum normal value can also be calculated as :

Sign = 1

Exponent = 0001 (Smallest non-zero exponent for normalized numbers)
Mantissa = 000

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 1 \times 2^{-6} \\ &= -2^{-6} \\ &\approx -1.56 \times 10^{-2}\end{aligned}$$

Table E-3: Advantages and Disadvantages of FP8

Advantages	**Disadvantages**
Extremely Low Memory Usage: Minimizes memory requirements.	Very Low Precision: Inadequate for tasks requiring high accuracy.
Faster Computation: Significantly increases speed in simple operations.	Small Dynamic Range: Can lead to overflows and underflows.
Reduced data movement and simpler arithmetic operations lead to lower power consumption.	Limited Support : FP8 is not yet supported by all hardware and software

iv. INT8: INT8 (8-bit Integer) is a format that uses 8 bits to represent integers in the range from -128 to 127. It is primarily used in tasks requiring integer values, like certain parts of machine learning models, especially in quantization.

Sign Bit	Magnitude
1bit	7bits

int8 is widely used for model quantization, where it is employed to significantly reduce the memory footprint and increase computational efficiency.

Calculation of Minimum and Maximum values:

(a) For Signed INT8:

- Maximum Value:
  Sign = 0
  Magnitude = 1111111

$$\begin{aligned}\text{Maximum Value} &= (-1)^0 \times 1111111 \\ &= 127\end{aligned}$$

- Minimum Value:
  Sign = 1

Magnitude = 0000000

$$\text{Minimum Value} = (-1)^1 \times 128(\text{2s complement of} 10000000)$$
$$= -128$$

(b) For Unsigned INT8:

- Maximum Value:
  Sign = 0
  Magnitude = 11111111

$$\text{Maximum Value} = 11111111$$
$$= 255$$

- Minimum Value:
  Sign = 0
  Magnitude = 00000000

$$\text{Minimum Value} = 0$$

Table E-4: Advantages and Disadvantages of int8

Advantages	**Disadvantages**
Extremely Low Memory Usage: Uses minimal memory, ideal for embedded systems.	Very Low Range: Limited to integer values, with no fractional precision.
High-Speed Computation: Operates very quickly in suitable tasks.	Precision Loss: Can cause inaccuracies in tasks that require floating-point precision.

v. **bfloat16:** BFLOAT16 (Brain Floating Point Format) is a custom 16-bit floating-point format primarily developed by Google. It uses the same 8-bit exponent as FP32 but reduces the mantissa to 7 bits, compared to 23 bits in FP32.

Sign Bit	Exponent	Mantissa
1bit	8bits	7bits

BFLOAT16 was introduced to address computational and memory bottlenecks in deep learning and high-performance computing. Many machine learning tasks do not require the precision of FP32, leading to unnecessary computational overhead. By reducing the mantissa size, bfloat16 allows for faster computation and

reduced memory usage while maintaining the dynamic range of FP32.

Calculation of Minimum and Maximum values:

$$\text{Value} = (-1)^{\text{sign}} \times 1.\text{Mantissa} \times 2^{\text{Exponent - 127}}$$

For maximum value:
Sign = 0
Exponent = 11111110 (All 1s is reserved for NaN values)
Mantissa = 1111111

$$\begin{aligned}\text{Maximum Value} &= (-1)^0 \times 1.1111111 \times 2^{11111110} \\ &\approx (1) \times 1.9921875 \times 2^{254-127} \\ &\approx 1.9921875 \times 2^{127} \\ &\approx 3.3895314 \times 10^{38}\end{aligned}$$

For minimum value:
The minimum value in BFLOAT16 is a subnormal number. To indicate a subnormal number, all 0s are stored in Exponent.
Sign = 1
Exponent = 00000000 (Indicating a subnormal number)
Mantissa = 0000001

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 0.0000001 \times 2^{-126} \\ &= (-1) \times 2^{-7} \times 2^{-126} \\ &= -2^{-133} \\ &\approx -9.1835496 \times 10^{-41}\end{aligned}$$

The minimum normal value can also be calculated as:
Sign = 1
Exponent = 00000001 (Smallest non-zero exponent for normalized numbers)
Mantissa = 0000000

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 1.0000000 \times 2^{-126} \\ &= (-1) \times (1) \times 2^{-126} \\ &= -2^{-126} \\ &\approx -1.175494 \times 10^{-38}\end{aligned}$$

Table E-5: Advantages and Disadvantages of BFLOAT16

Advantages	**Disadvantages**
Wide Dynamic Range: Retains the same exponent range as fp32, reducing overflow and underflow.	Reduced Precision: Smaller mantissa affects operations requiring high accuracy.
Computational Efficiency: Reduces memory usage and increases computation speed.	Software/Hardware Dependency: Requires specialized support.

vi. TF32: TF32 (TensorFloat32) is a custom numeric floating point format introduced by NVIDIA. It uses the same 8-bit exponent as FP32 but reduces the mantissa to 10 bits.

Sign Bit	Exponent	Mantissa
1bit	8bits	10bits

TF32 is specifically designed to accelerate tensor computations in machine learning while maintaining sufficient precision for stable training. TF32 bridges the gap between performance and precision in deep learning. While FP32 operations are computationally expensive and FP16 lacks the precision needed for complex model training, TF32 offers a middle ground by maintaining the exponent range of FP32 and reducing the mantissa size to speed up tensor operations.

Calculation of Minimum and Maximum values:

$$\text{Value} = (-1)^{\text{sign}} \times 1.\text{Mantissa} \times 2^{\text{Exponent - 127}}$$

For maximum value:
Sign = 0
Exponent = 11111110 (Maximum exponent for normalized numbers)
Mantissa = 1111111111

$$\begin{aligned}\text{Maximum Value} &= (-1)^0 \times 1.1111111111 \times 2^{11111110} \\ &\approx (1) \times 1.9990234 \times 2^{254-127} \\ &\approx 1.9990234 \times 2^{127} \\ &\approx 3.4011621 \times 10^{38}\end{aligned}$$

For minimum value:

Sign = 1

Exponent = 00000001
Mantissa = 1111111111

$$\begin{aligned}\text{Minimum Value} &= (-1)^1 \times 1.1111111111 \times 2^{-126} \\ &= (-1) \times (1) \times 2^{-126} \\ &= -2^{-126} \\ &\approx -1.175494 \times 10^{-38}\end{aligned}$$

Table E-6: Advantages and Disadvantages of TF32

Advantages	**Disadvantages**
Performance Gains: Faster tensor operations and matrix multiplications.	Less Precision than FP32: Unsuitable for applications needing high numerical accuracy.
Precision-Performance Balance: Sufficient precision for most training tasks.	Limited to NVIDIA GPUs: Proprietary format tied to specific hardware.

Appendix F: Quantization with Example

Quantization is a technique used in machine learning to reduce the precision of numbers used in computations and storage. There are two primary types of quantization: symmetric and asymmetric. Below is a detailed explanation of asymmetric quantization.

F.1 Asymmetric Quantization

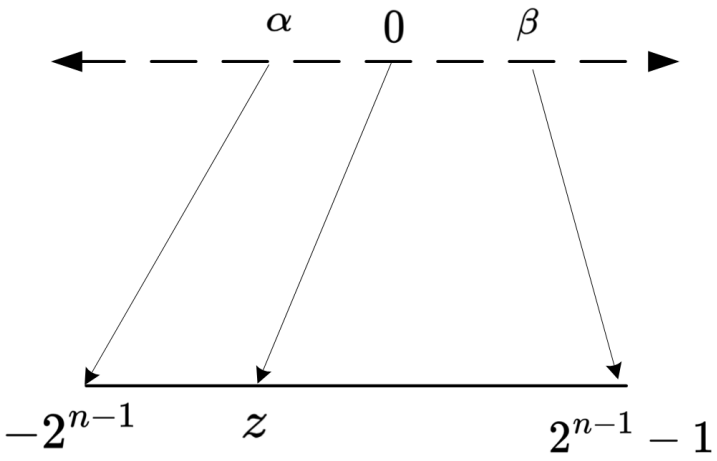


Figure F-1: Asymmetric Quantization

In asymmetric quantization, the range of values is not necessarily symmetric around zero. This allows the representation of values that are skewed toward either positive or negative numbers.

Characteristics:

i. The zero point is not fixed at zero; it is a non-zero number that provides offset of zero.

ii. Both scale and zero point are required to calculate quantized value.

$$\text{quantized value} = \text{round}\left(\frac{\text{original value}}{\text{scale}} + \text{zero-point}\right)$$

where,

$$\text{scale} = \frac{w_{\max} - w_{\min}}{2^n - 1}$$

$$\text{zero-point} = q_{\min} - \frac{w_{\min}}{\text{scale}}$$

Then the dequantized value can be calculated as:

$$\text{dequantized value} = (\text{quantized value} - \text{zero-point}) \cdot \text{scale}$$

Table F-1: Comparison of Symmetric and Asymmetric Quantization

Aspect	**Symmetric Quantization**	**Asymmetric Quantization**
Use Cases	- Neural network weights (often symmetrically distributed). - Scenarios requiring simple computations. - Signed data distributions centered around zero.	- Input activations in neural networks. - Data with skewed or biased distributions (e.g., image pixel values).
Advantages	- Simpler arithmetic due to zero-point being zero. - Efficient for hardware implementations using signed integers (e.g., int8). - Ideal for symmetric data distributions.	- Better range utilization for skewed data. - Reduces quantization error for asymmetric distributions. - Improved accuracy for biased data.
Disadvantages	- Inefficient for asymmetric data distributions. - May lead to higher quantization error for skewed data.	- More complex arithmetic due to non-zero zero-point. - Slightly higher computational overhead. - Hardware may require additional complexity to implement.

F.2 Quantization to 8-bit Integer Format

We will demonstrate an example of asymmetric quantization, and symmetric quantization can then be easily derived from this example. Given the following values:

$$\text{original value} = 0.14893225 \tag{F-1}$$

$$\text{scale} = 0.002625829565758799 \tag{F-2}$$

$$\text{zero-point} = -0.23390677845551977 \tag{F-3}$$

Substitute the values into the formula:

$$\text{quantized value} = \text{round}\left(\frac{0.14893225}{0.002625829565758799} - 0.23390677845551977\right)$$

Calculate step-by-step:

$$\frac{0.14893225}{0.002625829565758799} \approx 56.7181708753$$

Now, adding zero-point,

$$56.69868081 + -0.23390677845551977 \approx 56.4842640968$$

Thus we get,

$$\begin{aligned} \text{quantized value} &= \text{round}(56.46477403) \\ &= \boxed{56} \end{aligned} \tag{F-4}$$

F.3 Dequantization to 32-bit Floating-Point Format

Given the formula for dequantization:

$$\text{dequantized value} = (\text{quantized value} - \text{zero-point}) \times \text{scale}$$

Substitute the values:

$$\text{dequantized value} = (56 - (-0.23390677845551977)) \times 0.002625829565758799$$

Calculate step-by-step:

$$56 - (-0.23390677845551977) = 56 + 0.23390677845551977 \approx 56.23390678$$

Thus we get,

$$\begin{aligned} \text{dequantized value} &= 56.23390678 \times 0.002625829565758799 \\ &\approx 0.14766065501 \\ \text{dequantized value} &\approx \boxed{0.14766} \end{aligned} \tag{F-5}$$

F.4 Quantization Error

The quantization error is given by:

$$\text{quantization error} = \text{original value} - \text{dequantized value}$$

Substituting the values:

$$\begin{aligned} \text{quantization error} &= 0.14893225 - 0.14766065501 \\ &= 0.00127159499 \\ \text{quantization error} &\approx \boxed{0.00127} \end{aligned} \tag{F-6}$$

F.5 Mean Squared Quantization Error (MSQE)

For the given data, we can now calulate Mean Squared Quantization Error (MSQE) and see how quantization error increases as integer bit-width decreases as shown in Table F-2.

$$\text{Original Tensor:} \quad \mathbf{T} = \Big[-0.41067, \ldots, -0.92187\Big]$$

Table F-2: Increase in Quantization Error with Lower Bit-width Integers

Bit-width	**Scale**	**Zero Point**	**Quantized Tensor**	**MSQE**
32	3.48×10^{-10}	4.99×10^{8}	$[-679681118, \ldots, -2147483648]$	6.82×10^{-21}
16	2.28×10^{-5}	7620.56	$[-10371, \ldots, -32768]$	3.86×10^{-11}
8	5.87×10^{-3}	29.15	$[-41, \ldots, -128]$	1.67×10^{-6}

As the bit-width decreases, the quantization error increases due to the reduced precision in representing the original floating-point values. With fewer bits available to represent the data, the quantization process introduces more significant rounding errors, leading to a higher MSQE. This trade-off between precision and memory efficiency is a common consideration in quantization techniques, where the goal is to minimize the error while reducing the memory footprint of the data.

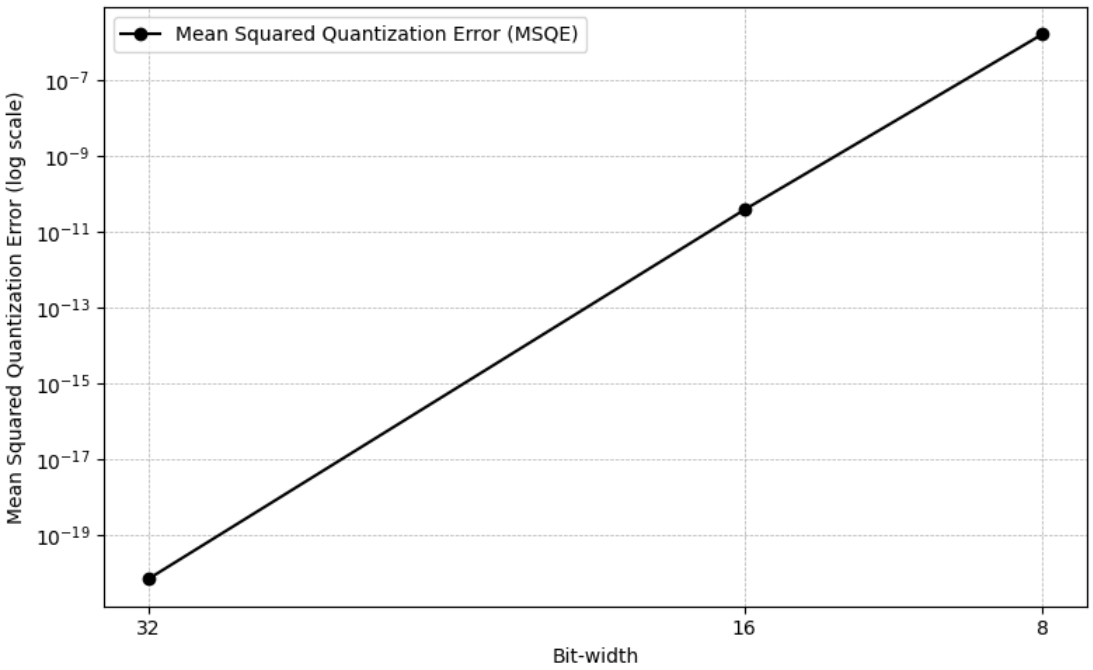


Figure F-2: MSQE vs Bit-width for Quantization

References

[1] Biteable, "Video marketing statistics: The state of video marketing in 2021," 2021, accessed: 2024-06-15. [Online]. Available: https://biteable.com/blog/video-marketing-statistics/

[2] S. Shlien, "Guide to mpeg-1 audio standard," *IEEE Transactions on Broadcasting*, vol. 40, no. 4, pp. 206–218, 1994.

[3] H. Kalva, "The h.264 video coding standard," *IEEE MultiMedia*, vol. 13, no. 4, pp. 86–90, 2006.

[4] M. T. Pourazad, C. Doutre, M. Azimi, and P. Nasiopoulos, "Hevc: The new gold standard for video compression: How does hevc compare with h.264/avc?" *IEEE Consumer Electronics Magazine*, vol. 1, no. 3, pp. 36–46, 2012.

[5] J. Li, B. Li, and Y. Lu, "Deep contextual video compression," 2021.

[6] B. Liu, Y. Chen, S. Liu, and H.-S. Kim, "Deep learning in latent space for video prediction and compression," in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 701–710.

[7] H. Chen, B. He, H. Wang, Y. Ren, S.-N. Lim, and A. Shrivastava, "Nerv: Neural representations for videos," 2021.

[8] K. Panneerselvam, K. Mahesh, V. Josephine, and R. Anandan, "Effective and efficient video compression by the deep learning techniques," *Computer Systems Science and Engineering*, vol. 45, p. 1047–1061, 11 2022.

[9] C. Gomes, R. Azevedo, and C. Schroers, "Video compression with entropy-constrained neural representations," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2023, pp. 18 497–18 506.

[10] J. Li, B. Li, and Y. Lu, "Neural video compression with diverse contexts," 2023.

[11] L. A. Lanzendörfer and R. Wattenhofer, "Siamese siren: Audio compression with implicit neural representations," 2023.

[12] X. Zhang, R. Yang, D. He, X. Ge, T. Xu, Y. Wang, H. Qin, and J. Zhang, "Boosting neural representations for videos with a conditional decoder," 2024.

[13] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit neural representations with periodic activation functions," 2020.

[14] A. Choudhury, P. Singh, and G.-M. Su, "Nerva: Joint implicit neural representations for videos and audios," in *2024 IEEE International Conference on Multimedia and Expo (ICME)*, 2024, pp. 1–6.

[15] "Quantization — pytorch 2.5 documentation," https://pytorch.org/docs/stable/quantization.html, [Accessed 15-10-2024].

[16] J. Han, H. Zheng, and C. Bi, "Kd-inr: Time-varying volumetric data compression via knowledge distillation-based implicit neural representation," *IEEE Trans. Vis. Comput. Graph.*, vol. 30, no. 10, pp. 6826–6838, October 2024. [Online]. Available: https://doi.org/10.1109/TVCG.2023.3345373