The University of Sussex
School of Engineering & Informatics

Masters Dissertation

Programme…MSc in Artificial Intelligence and Adaptive Systems…….

Candidate number…… 276186 ………

Title of Dissertation:

Audio Signal Denoising with Spiking Neural Networks: A Novel Approach Based on Short-Time Fourier Transform and Targeted Noise Suppression

Approximate number of words: 13,195

Supervisor: Prof. Thomas Nowotny

Date submitted 09/01/2025.	Time submitted 13:25

Declaration
I certify that the information on this cover sheet is correct.
I certify that the content of this dissertation is my own work, and that my work contains no examples of misconduct such as plagiarism, collusion, or fabrication of results.

Candidate's signature ……[illegible]……………………………………………….

Abstract:

This project aims to develop a novel denoising algorithm within a neuromorphic system, inspired by the Intel N-DNS challenge. The proposed method involves a pre-processing stage, followed by Short-Time Fourier Transform (STFT) for signal representation. Subsequently, Short-Time Fourier Transform (STFT) is applied to extract time-frequency features. These features are then encoded and presented to the spiking neural network for denoising. Three approaches were employed: multiple to single mapping, single to single mapping and a method that focused on targeting noise instead of clean speech. Results demonstrate the effectiveness of the noise targeting approach in denoising audio signals, achieving a lowest mean squared error (MSE) of 0.74 with 25% error reduction. The methodology leverages the energy efficiency and scalability of SNNs, making this approach suitable for power-constrained real-world applications.

Acknowledgements

I am deeply grateful to my supervisor, Prof. Thomas Nowotny, for their invaluable guidance and support throughout this research endeavour. Their expertise provided crucial direction, and their constructive feedback was essential in refining my arguments and improving the quality of this work. I am particularly indebted to their patience and encouragement, which were instrumental in my successful completion of this dissertation.

Table of Contents

1. Introduction

This project was inspired by the Intel N-DNS challenge. The Intel N-DNS Challenge is split into two distinct routes: the Algorithmic Approach and the Real-time Implementation on Loihi 2 Neuromorphic Hardware. This project will focus on the Algorithmic Approach, with the goal of developing novel denoising algorithms and implementing them effectively within a neuromorphic system.

The design process for a neuromorphic audio processing system can be divided into three main phases. The initial phase concerns the efficient representation of a waveform within the neuromorphic domain. Various cochleogram models have been proposed as potential solutions for the Intel N-DNS Challenge. These models offer features such as sparse binary spike representations, high sensitivity, frequency selectivity, large dynamic range, pitch-shifting, and self-peak normalization. However, it has been observed that models such as those referenced in Timcheck et al., (2023) are computationally expensive to invert with high fidelity. This characteristic makes them unsuitable for a low-power denoising system. As a more efficient alternative, this project will propose the use of a more conventional audio encoding approach, the Short-Time Fourier Transform (STFT).

The second phase focuses on effectively performing the required audio processing task (denoising) on this neuromorphic representation. Event-prop algorithm is used during this "Audio Processing Phase". It is a learning rule for training recurrent neural networks, and in this case, it's used to optimize the spiking neural network (SNN). The third and final phase involves efficiently inverting the neuromorphic representation to produce an output waveform using a decoder to ensure the end result is a clean denoised audio signal. This phase won't be implemented in this project.

In comparison to conventional artificial neural networks, Spiking Neural Networks (SNNs) demonstrate remarkable energy efficiency through their utilization of simplified bio-inspired neuron models as core processing units and event-driven spike trains for information transmission. Recent applications of SNNs have shown promising results across various domains, including object detection, speech recognition, and speech enhancement systems. The development of neuromorphic computing platforms has revealed significant potential for power-constrained real-world applications. Notable examples include the IBM TrueNorth system, which achieves remarkable energy efficiency with its 5.4 billion transistors operating at a mere 70mW power density - approximately 1/10000th of traditional processing units. Similarly, the SpiNNaker platform, developed at Manchester, offers ASIC-based hardware implementations of SNNs, utilizing multiple ARM cores and FPGAs in conjunction with the PyNN software API to create a scalable architecture supporting large-scale neural networks. These technological advancements in hardware development underscore the viability of implementing energy-efficient neuromorphic computing solutions in mobile applications where power consumption is a critical consideration. [13]

In a study, researchers leveraged the energy-efficient, bio-inspired computing capabilities of SNNs to develop an innovative spectrogram-based rate coding approach. This method enhances speech processing through efficient lateral inhibition within the SNN framework. Notably, the architecture operates independently of specific noise types, relying solely on forward propagation and natural event-based information processing to effectively suppress uncorrelated noise in the time-frequency domain. The system's effectiveness stems from its strategic use of lateral inhibitory connections, which preserve spike trains occurring at similar frequencies. Within each time resolution bin, individual LIF neurons respond to both speech and noise components, generating spike trains at fixed frequencies. The design anticipates higher firing rates from LIF neurons during STFT time resolution bins containing speech components, while noise components typically produce lower frequency spike trains. This distinction enables effective discrimination through lateral inhibition mechanisms. [13]

The primary objective of this project is to develop an efficient and effective denoising algorithm for audio signals, leveraging the principles of neuromorphic computing. To achieve this goal, the project will first represent audio signals using the Short-Time Fourier Transform (STFT). This approach ensures efficient encoding and decoding within the neuromorphic domain. Following this, a novel spiking neural network (SNN) architecture will be designed specifically for the task of audio denoising. This SNN will then be trained using the Event-prop learning rule to optimize its performance and minimize noise in the audio signals. Finally, the performance of the proposed denoising algorithm will be rigorously evaluated using the Mean Squared Error (MSE) metric on a diverse set of audio datasets. Three approaches were employed: multiple-to-single mapping, single-to-single mapping, and noise targeting. This approach aims to capitalize on the energy efficiency and biological plausibility of neuromorphic computing to create a novel and effective solution for audio denoising.

2. Literature Review

There are many ways to go about how to approach the speech denoising problem. We can categorise the approach into two: conventional methods, including Wiener filtering, spectral subtraction, and Minimum Mean Square Error (MMSE) methods, and more recent deep learning methods such as filtering based on spectral masks generated by neural networks [1,2]. Initial attempts at speech denoising involved development of spectral subtraction in the late 1970s which was followed by other enhancement methods such as wiener filtering and logSTSA filtering in early 1980s. A foundational study that laid the works in the field of spectral subtraction was a study [3] by Boll in 1979, on suppression of acoustic noise in speech using spectral subtraction. The method is mathematically formulated as an estimation problem, where the observed signal is modelled as the sum of the clean speech signal and additive noise. The core of the algorithm lies in estimating the clean speech power spectrum by subtracting an estimate of the noise power spectrum from the observed signal's power spectrum. The algorithm implementation involves segmenting the input speech into overlapping frames, applying a Hamming window, computing the Discrete Fourier Transform (DFT), estimating the noise spectrum during non-speech activity, and performing spectral subtraction followed by half-wave rectification. Experimental results using helicopter speech data demonstrated significant improvements in speech quality and intelligibility, with Diagnostic Rhyme Test (DRT) scores increasing from 84 for unprocessed speech to 88 with spectral subtraction. Despite its effectiveness, the method has limitations, particularly in highly non-stationary noise environments and when there is significant spectral overlap between speech and noise [3].

The more we assume about the speech the more sensitive the speech enhancement system will be to inaccuracies or deviations from these assumptions. Hence When dealing with the problem of separating speech from background noise to model the speech signal we should take into consideration of different types of background noise. Background noise could be coming from different source of environment such as noise-like (wide-band random noise) or coming from an environment with competing speakers [1]. Another aspect of speech enhancement we should consider is the criteria for enhancement. Some systems take into aspects of human perception while others are heavily motivated by on mathematical criteria which some is believed to be a better match than others to aspects of human perception [1].

2.1 Spiking Neural Networks

Spiking neural networks (SNNs) are emerging as an energy efficient alternative to traditional artificial neural networks (ANNs) [1]. However, SNNs are primarily explored for classification tasks, there has been limited progress in applying them to regression tasks, hindering their broader application. Audio denoising, a typical regression task, plays a crucial role in various applications, particularly on power-constrained edge devices such as headsets, hearing aids, and smartphones [5]. These devices require real-time processing capabilities while operating under restricted power budget to ensure a seamless user experience. Traditional methods driven by ANNs for audio denoising often struggle to meet these demands due to their heavy computational complexity [6]–[8]. In light of these challenges, SNNs present a promising solution for audio denoising [9]. By capitalizing on the eventdriven computation and high-level sparsity of spiking events, the compute load could be significantly reduced [10], enabling real-time processing without compromising the power consumption limitation. Nevertheless, developing an SNN-based system that can deliver denoising performance comparable to conventional solutions is challenging. It remains an open question to determine suitable spiking neuron models, network architectures, and loss functions that can fully unleash the power of SNNs in audio denoising.

A significant advancement in neural speech processing emerged from Hao et al.'s 2024 research, which addressed fundamental limitations in traditional spiking neuron architectures [16]. Their work specifically targeted the constraints of the Leaky-Integrate and Fire (LIF) neuron model in speech denoising applications, where the conventional fixed decay factor had been hampering performance in temporal information processing. Although previous iterations, such as the Parametric LIF model, attempted to enhance flexibility through learnable decay factors, they still operated within the confines of time-invariant decay rates [17]. Hao et al.'s breakthrough came through the implementation of an innovative gating mechanism that dynamically modulates decay rates across time steps. This sophisticated approach enables neurons to intelligently adapt their membrane potentials based on temporal context, resulting in more nuanced and effective signal processing. The impact of this development was validated through comprehensive performance evaluations, where the enhanced model demonstrated superior results compared to established baselines, including the Intel DNS network. This achievement represents not just an incremental improvement, but a fundamental reimagining of how spiking neural networks can approach speech denoising tasks, offering both improved computational efficiency and enhanced denoising capabilities [16].

A different study presented a novel SNN design consisting of three layers and incorporating lateral inhibition mechanisms. The study employed the Bens Spiker algorithm to convert log-scaled STFT magnitude measurements into precisely timed spikes. The researchers implemented masking techniques to filter out unrelated spike patterns, which resulted in effective noise reduction performance. However, the study faced certain constraints, particularly the SNN's inability to learn adaptively and its relatively basic architectural structure [18].

When considering how to effectively represent a waveform in neuromorphic systems, multiple approaches exist for data representation. These include binary spikes, graded spikes, population codes, sparse distributed codes, and phase codes. The encoding process can be implemented through various algorithms, such as biologically-inspired cochleogram models, Short-Time Fourier Transforms (STFTs), and Mel-frequency cepstral coefficients. Each of these methods offers distinct advantages and trade-offs in terms of computational efficiency and signal representation accuracy.

Spiking neural networks (SNNs) represent a significant advancement in neural network technology, offering a highly promising and energy-efficient alternative to traditional artificial neural networks (ANNs). These networks stand out for three key characteristics: their remarkable biological plausibility in mimicking natural neural processes, their sophisticated spatial-temporal dynamics that enable complex pattern recognition, and their efficient event-driven computation that activates only when necessary. The training of SNNs has been revolutionized through direct training algorithms that utilize the surrogate gradient method. This approach provides researchers and developers with unprecedented flexibility in two crucial areas: first, in designing and implementing novel SNN architectures that can tackle increasingly complex problems, and second, in thoroughly exploring the intricate spatial-temporal dynamics that make SNNs so powerful. Recognized as the third generation of neural networks, these brain-inspired networks have emerged as formidable competitors to traditional ANNs. Their competitive edge stems from two primary factors: their close resemblance to biological neural systems and their exceptional energy efficiency when implemented on specialized neuromorphic hardware. A particularly noteworthy innovation in SNNs is their use of binary spikes for information processing. This approach enables them to utilize low-power accumulation (AC) operations instead of the more energy-intensive multiply-accumulation (MAC) operations common in traditional networks. This fundamental difference results in dramatically improved energy efficiency, making SNNs increasingly attractive for practical applications, especially in scenarios where power consumption is a critical consideration [19].

The Leaky Integrate-and-Fire (LIF) neuron model stands as a fundamental component in Spiking Neural Networks (SNNs), as extensively documented in recent research by Zhou and colleagues. Despite its straightforward design, this model effectively captures essential biological neural characteristics, making it a cornerstone of neuromorphic computing architectures [19].

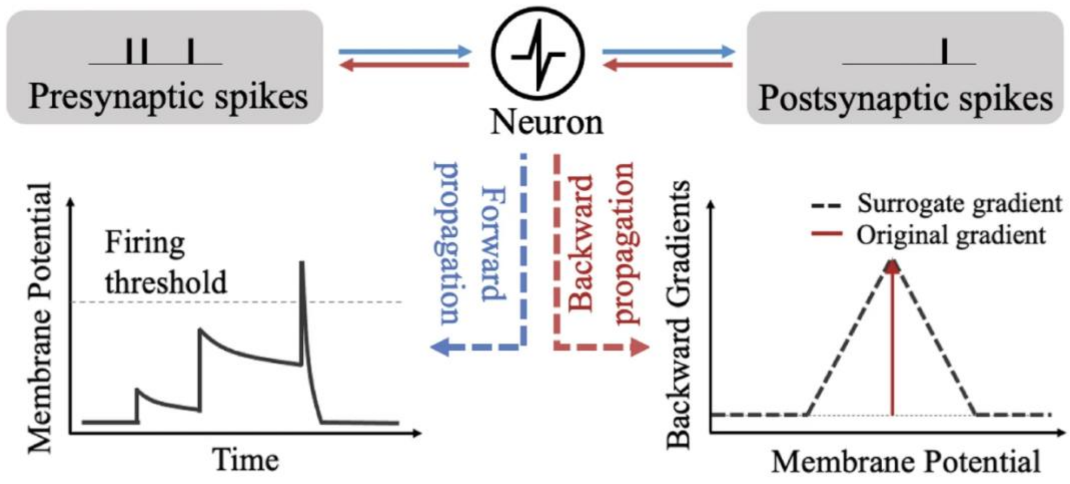


Fig 1. Non-differentiable spiking neuron model.
This panel illustrates forward (blue arrow) and backward (red arrow) propagation in a Leaky Integrate-and-Fire (LIF) neuron. During forward propagation, the membrane potential increases in response to incoming pre-synaptic spikes. If the membrane potential exceeds the firing threshold, the LIF neuron generates a post-synaptic spike and resets its membrane potential. This leak-integrate-and-fire behavior introduces non-differentiability in the membrane potential. Consequently, surrogate gradient functions are employed to approximate the backward gradient [19].

In the Leaky Integrate-and-Fire (LIF) model, the membrane potential dynamics are driven by a constant current input (I). This input gradually charges the membrane, leading to an exponential increase in its potential (V). The model incorporates a critical threshold value (V_Th) - when the membrane potential reaches this threshold, the neuron generates a spike, as shown in Figure 1. During this process, which is considered instantaneous, the membrane capacitance undergoes a complete discharge, causing the membrane potential to return to its baseline resting state. This cyclical process of charging, spiking, and resetting continues as long as the input current (I) remains present [20].

The temporal evolution of membrane potential in the LIF model is described by a first-order differential equation. This equation captures how the membrane potential (u_i) changes over time in response to an input current (I_i). The membrane time constant (τ) and membrane resistance (R) are key parameters that determine the neuron's response characteristics [21].

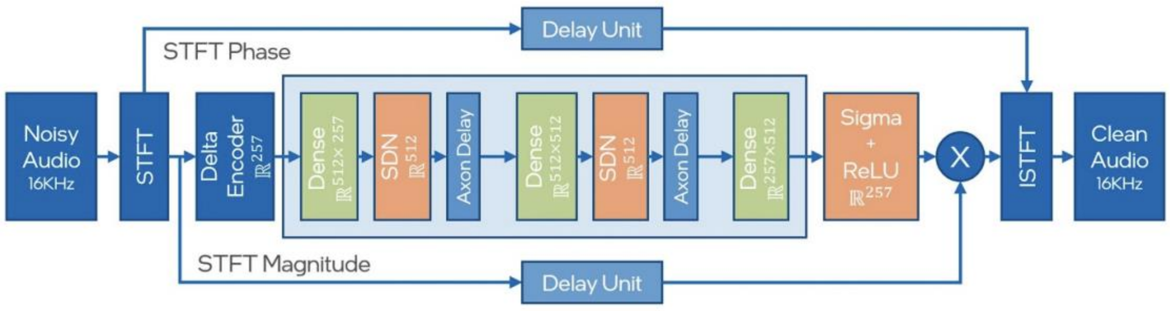
The governing equation can be written mathematically as:

$$\tau \frac{du_i}{dt} = -u_i(t) + RI_i(t)$$

In this equation, τ=RC represents the time constant of the RC circuit, where R and C denote the resistance and capacitance values respectively. The input current I_i (t) can be expressed as:

$$I_i(t) = \sum_j w_{ij} \sum_f \delta\left(t - t_j^{(f)}\right)$$

The synaptic weights (wij) determine how strongly presynaptic neuron j influences postsynaptic neuron i. The summation aggregates all spike times from the presynaptic neurons, with each spike represented by index f [21].



Delay Unit
STFT Phase
Noisy Audio 16KHz
STFT
Delta Encoder $\mathbb{R}^{257}$
Dense $\mathbb{R}^{512\times257}$
SDN $\mathbb{R}^{512}$
Axon Delay
Dense $\mathbb{R}^{512\times512}$
SDN $\mathbb{R}^{512}$
Axon Delay
Dense $\mathbb{R}^{257\times512}$
Sigma + ReLU $\mathbb{R}^{257}$
X
ISTFT
Clean Audio 16kHz
STFT Magnitude
Delay Unit

Fig 1. Sigma-Delta Neural Network (SDN) for Audio Denoising

This figure depicts the architecture of a Sigma-Delta Neural Network (SDN) of the Intel N-DNS baseline solution. The system operates in the Short-Time Fourier Transform (STFT) domain, processing both the magnitude and phase components of the noisy audio signal.

The Intel N-DNS Challenge presents opportunities to explore various encoding approaches. These could include developing invertible biological-inspired cochleogram models, implementing sparse STFTs, or creating dynamic encoding systems that incorporate feedback mechanisms similar to those found in brain's sensory processing pathways. A crucial consideration is that the encoding method must be specifically designed to complement the neuromorphic denoising system's objectives. This principle mirrors biological systems, where encoding mechanisms have evolved to work harmoniously with their intended functions.

The neuromorphic solution employs a streamlined feedforward architecture known as the sigma-delta ReLU neural network (SDNN), as shown in Figure 1. This design integrates two fundamental neuromorphic computing principles: the efficient transmission of sparse messages through sigma-delta neurons, and temporal processing utilizing axonal delays. Delta encoding achieves data efficiency by transmitting only significant changes that exceed a predetermined threshold, leveraging temporal patterns in the data. Meanwhile, sigma encoding works to reconstruct the original signal at the receiving end. The sigma-delta neuron emerges from combining these encoding approaches with a ReLU non-linearity, resulting in reduced synaptic computations through sparse message handling. The network incorporates learnable axonal delays, which provide essential short-term memory capabilities. These delays enable the system to process and correlate audio features from different temporal points, a crucial feature for audio denoising applications. The delays enhance the network's expressivity and performance in handling spatio-temporal patterns. The SDNN architecture consists of three main components:

- **Encoder:** Implements a Short-Time Fourier Transform (STFT) on the noisy audio input, followed by delta encoding of the magnitude. Operating at 16 kHz, it processes 8ms time-steps using a 512-sample window with 128-sample hop length.

- **N-DNS (Neuromorphic Denoiser):** A three-layer feedforward network utilizing sigma-delta ReLU neurons with axonal delays. It generates a multiplicative mask for noise reduction while operating in the sparse domain.
- **Decoder:** Combines the N-DNS predicted mask with delayed STFT components to reconstruct the cleaned audio signal through inverse STFT.

The network's training utilizes Lava-dl's SLAYER framework, which addresses the challenge of spike non-differentiability through surrogate gradient methods. The training process incorporates fixed-precision computations compatible with Loihi 2 hardware, using a combined loss function of negative SI-SNR and STFT magnitude reconstruction error, optimized using the RADAM algorithm.

2.2 Short-Time Fourier Transform (STFT) & Mel-Scale Conversion:

The Fourier Transform (FT) provides a frequency domain representation of a signal, effectively decomposing it into its constituent frequencies. The Fast Fourier Transform (FFT) algorithm enables efficient computation of the FT. However, the FT, and consequently the FFT, assumes stationarity of the signal, thus failing to capture spectral variations over time. This limitation precludes their application to the analysis of non-stationary signals such as those encountered in vibration, speech, and biomedical domains. To overcome this limitation, joint time-frequency analysis techniques are employed. These methods yield a two-dimensional representation of the signal, where both time and frequency information are concurrently depicted. This is typically achieved by correlating the signal with a set of elementary functions, such as frequency-modulated Gaussian functions. However, the time-frequency resolution is inherently constrained by the uncertainty principle, which postulates a fundamental trade-off between the precision of time localization and frequency resolution. To mitigate the loss of time or frequency resolution inherent in the FT, several advanced signal processing techniques have been developed, including the Short-Time Fourier Transform (STFT), the Wigner-Ville Distribution (WVD), and the Wavelet Transform (WT). These methods offer varying approaches to achieve a more nuanced time-frequency representation of non-stationary signals [11].

The Short-Time Fourier Transform (STFT) is a method used to overcome the limitations of the standard FFT when dealing with non-stationary or noisy signals. It's particularly helpful for extracting narrow-band frequency content from such signals. The STFT works by dividing the original signal into smaller time windows or segments. Then, it applies the Fourier Transform to each of these segments individually. This allows us to see how the frequency content of the signal changes over time within each segment. The mathematical equation for the short time Fourier Transform is given by:

$$S_{f,\tau} = \sum_{t=0}^{N-1} x_t \omega_{t-\tau} e^{-j2\pi ft}$$

where $x(t)$ is the signal being analysed, $\omega(t)$ is the window function centred at time τ. However, the STFT is not without its limitations. The choice of window length introduces a trade-off between time and frequency resolution. A shorter window yields better time resolution but poorer frequency resolution, while a longer window provides improved frequency resolution at the expense of time resolution. This is a manifestation of the uncertainty principle in the context of the STFT. Furthermore, the STFT struggles to accurately represent two closely spaced natural frequencies in a signal due to the limitations of the windowing process [11].

The Short-Time Fourier Transform (STFT) uses a fixed-size window to analyse signals, which can be problematic when dealing with signals that change at varying speeds. Ideally, we'd want a wider window to capture slow changes and a narrower window to capture rapid changes in the signal. The STFT's uniform time-frequency resolution doesn't allow for this adaptability. This limitation led to the development of the wavelet transform, which offers a flexible resolution. It can zoom in on short time intervals and high frequencies or zoom out to look at longer time intervals and low frequencies, making it more suitable for analysing signals with varying dynamics [12].

2.3 Event-prop Algorithm

The core of the Event Propagation algorithm lies in its capability to accommodate two distinct types of loss functions. For tasks with relatively straightforward temporal dynamics, per-timestep loss functions can be utilized, where the loss is calculated at each individual timestep based on the output neuron's membrane potential. However, for tasks that involve more intricate temporal dependencies, per-trial loss functions are employed. These functions calculate the loss once per trial by considering the integral of the membrane potential over time, thus capturing the cumulative effect of neuronal activity throughout the trial. To facilitate the backward pass, the EventProp algorithm introduces adjoint state variables and additional parameters into the neuron models. These adjoint variables track the sensitivity of the loss with respect to changes in the neuron's state, enabling the efficient computation of gradients during backpropagation. Furthermore, ring buffers are implemented to store spike times and other pertinent information during the forward pass, which are subsequently accessed during the backward pass for gradient calculation. The training process involves custom update mechanisms that carry out SoftMax calculations, variable resets, and gradient updates. Optimizers, such as Adam, are utilized to iteratively adjust the connection weights based on the computed gradients. Additionally, regularization techniques can be incorporated to mitigate overfitting by controlling the firing rate of hidden neurons.

The original EventProp framework, as proposed by Wunderlich and Pehle, supported loss functions of the following form [13]:

$$\mathcal{L} = l_p(t^{\text{post}}) + \int_0^T l_V(V(t), t)\ dt$$

where:

- $l_p(t^{\text{post}})$ represents a loss term calculated at the end of each trial.
- $\int_0^T l_V(V(t), t)\ dt$ denotes the integral of a per-timestep loss function, l_V, which is computed based on the output neuron's membrane potential, $V(t)$, at each timestep t throughout the trial duration T.

However, subsequent research by Nowotny et al. revealed that for tasks characterized by intricate temporal structures, per-timestep loss functions might prove inadequate for effective learning. To address this limitation, they extended the EventProp framework to accommodate loss functions that operate on the integral of the membrane potential over the entire trial [14]:

$$\mathcal{L}_{\mathcal{F}} = \mathrm{F}\left(\int_0^{\mathrm{T}} \mathrm{l_V}(\mathrm{V}(\mathrm{t}), \mathrm{t})\ \mathrm{dt}\right)$$

where F is a function applied to the integral of the per-timestep loss.

2.4 Optimisation with Adam Optimizer

In this study, the spiking neural network was trained using the Adam optimizer. Adam, which stands for Adaptive Moment Estimation, is a widely adopted optimization algorithm known for its ability to accelerate convergence and handle complex loss landscapes effectively. This optimizer achieves its adaptability by dynamically adjusting the learning rate for each parameter based on historical gradient information. By incorporating a momentum term, Adam smooths the update direction and helps navigate through local minima. Furthermore, it utilizes an RMSprop-like mechanism to normalize the learning rate, preventing drastic oscillations during training. Bias correction factors are also applied to ensure accurate estimation of the first and second moments, especially during the initial stages of training. The implementation of Adam in the model involved setting key hyperparameters such as the learning rate (alpha), exponential decay rates (beta1 and beta2), and a small constant for numerical stability (epsilon). These hyperparameters were carefully tuned to optimize the training process [15]. By leveraging the Adam optimizer, the study aimed to enhance the training efficiency and overall performance of the spiking neural network. Its adaptive nature and ability to handle complex optimization scenarios make it particularly well-suited for training models with spiking neurons, which often exhibit intricate dynamics and non-linear relationships.

Equation	**Description**
$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$	First Moment Estimate: Represents the exponential moving average of the gradients, providing a smoothed estimate of the gradient direction. Maintains a "momentum" of the past gradients, giving more weight to recent gradients while still incorporating information from previous ones. β1 is a hyperparameter that controls the decay rate of this moving average.
$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$	Second Moment Estimate: Calculates the moving average of the squared gradients, helping to adaptively adjust the learning rate for each parameter. Parameters with consistently large gradients will have larger v_t values, leading to smaller learning rate adjustments, and vice versa. β2 controls the decay rate for this moving average.
$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t}$	Bias-Corrected First Moment: Ensures that the first moment estimate is accurate, especially during initial iterations.
$\widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$	Bias-Corrected Second Moment: Corrects the second moment to provide a reliable estimate.
$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}$	Parameter Update: Updates the parameters, taking into account the bias-corrected moments and ensuring numerical stability with a small constant ϵ.

Table 1. Adam Optimizer: Key Equations and Concepts
Presents the core equations and their descriptions underlying the Adam optimization algorithm. Adam leverages exponential moving averages of the gradient and squared gradient to compute bias-corrected first and second moment estimates. These estimates are then utilized in the parameter update rule to adaptively adjust the learning rate and efficiently guide the optimization process.

The parameter update equation in Adam, as shown in table 1., serves as the core mechanism driving the optimization process. It governs how the model's parameters, denoted by θ, are adjusted at each timestep 't'. The update involves moving the parameters in the direction suggested by the negative of the bias-corrected first moment estimate, which encapsulates a smoothed history of gradients. The magnitude of this update is determined by the learning rate α, a hyperparameter controlling the step size. Crucially, Adam incorporates an adaptive scaling factor by dividing the first moment estimate by the square root of the bias-corrected second moment estimate. This scaling dynamically adjusts the effective learning rate for each parameter, mitigating the impact of noisy or sparse gradients and facilitating smoother convergence across diverse optimization landscapes. A small constant ε is included to ensure numerical stability. In essence, this equation embodies Adam's ability to combine momentum-based optimization with adaptive learning rates, resulting in an effective and widely applicable algorithm for training machine learning models [15].

3. Methodology

3.1 Datasets

The Intel N-DNS dataset, derived from the Microsoft DNS Challenge dataset, will be used in this project. This corpus of human clean (ground truth) speech audio samples includes multiple languages and speech types such as English, German, French, Spanish, and Russian, Emotional speech and covers various categories of noises. Clean speech clips that had DNSMOS P.835 scores greater than or equal to 4.2, 4.5. and 4.0 for speech quality (SIG), background noise quality (BAK), and overall quality (OVRL), respectively was used as stated in the Microsoft DNS challenge [1]dataset [4].

For this project, for clean speech emotional speech was selected as it had lowest memory space of 2.4G. This training dataset is derived from the Crowd-sourced Emotional Multimodal Actors Dataset (CREMA-D) [5], made available under the Open Database License. This dataset consists of 7,442 audio clips from 91 actors: 48 male and 43 female, totaling 3.5 hours of audio with diverse ethnic backgrounds including African American, Asian, Caucasian, Hispanic, and Unspecified.Actors read from a pool of 12 sentences to generate this emotional speech dataset. It includes six emotions: Anger, Disgust, Fear, Happy, Neutral, and Sad, at four intensity levels: Low, Medium, High, Unspecified.

[1] https://github.com/microsoft/DNS-Challenge

The noise dataset comprises approximately 62,000 clips across 150 noise categories. These noise clips were sourced from Audio Set [2][7] and Freesound. [3]Audio Set is a collection of around 2 million human-labeled 10-second sound clips extracted from YouTube videos, covering roughly 600 audio events. A sampling technique was implemented to balance the noise classes in the dataset, ensuring each class contains at least 500 clips. A speech activity detector was employed to exclude any clips with speech content from our noise dataset, resulting in noise clips devoid of speech [8]. The entire noise dataset amounts to 181 hours of audio [4]. For this project, the first segment of the noise_fullband audio dataset was used.

3.2 Audio Pre-processing

The noisy speech is generated from the clean and noise audio datasets sourced from the Microsoft DNS Challenge datasets. These files must be combined using a precise methodology to ensure an accurate mixture of clean speech and noise. This step is crucial for providing the neural network with the necessary input data for effective training and testing. After the extraction of clean and noise datasets, the combining process of clean (ground truth) and noise (additive) was done in order to obtain dataset of noisy (ground truth + noise) audio. By carefully controlling the levels of noise introduced, it was ensured that the resulting dataset accurately reflects the conditions of a realistic stimulation of noisy environments. This method not only enhances the quality of the training data but also ensures that the neural network can generalize its denoising capabilities to a wide range of scenarios. Moreover, the integrity of the clean speech must be maintained during the mixing process to ensure that the neural network can effectively learn the distinction between noise and speech. The resulting noisy speech dataset will therefore serve as a robust foundation for training the spiking neural network, enabling it to achieve high performance in speech denoising tasks.

This section outlines the methodology employed in the development of an audio processing pipeline designed to create a dataset of noisy speech signals. The pipeline processes clean speech and noise files, resamples them to a uniform sample rate, normalizes the signals, and combines them at specified signal-to-noise ratios (SNRs).

3.2.1 Data collection and preparation

The pipeline begins with the collection of two types of audio data: clean speech and noise. The clean speech data is sourced from the CREMA-D emotional speech dataset that contains 5060 files, while the noise data is extracted from a separate noise dataset, containing 3293 files. These datasets are stored in specified directories for processing.

[2] https://research.google.com/audioset/
[3] https://freesound.org/

3.2.2 Resampling and truncation

To ensure consistency across all audio files, the pipeline resamples each file to a target sample rate of 16 kHz using the torchaudio library. This step is crucial for standardizing the input data and mixing of clean and noise files, which may originally have varying sample rates. Each audio file is loaded and resampled. After resampling, all audio waveforms are truncated to the length of the shortest individual file in the pair. This ensures uniformity in duration, which is necessary for subsequent processing steps. Padding shorter signal with zeros or with repeated content and repeating the audio clip of the shorter audio type are other alternatives to match the length. However, for this project maintaining the integrity of original audio signal without significant alteration of its content or introducing repetition was important. For simplicity, truncation was chosen at the expense of some loss of information in audio.

3.2.3 Signal normalisation

Normalization is applied to both speech and noise signals to prepare them for combination and subsequent step of Fourier transform. The normalization process involves mean subtraction, standard deviation scaling, and Tanh activation. The mean subtraction process involves subtracting the mean of each audio signal from the waveform, effectively centering the signal around zero. Audio signals can sometimes have a DC offset, which is a constant component added to the signal that can be introduced during recording or transmission. Centering the signal removes this offset, ensuring that the waveform oscillates symmetrically around the zero line. This normalization step is particularly beneficial for the training process, as when the optimization landscape becomes more symmetric, it can lead to faster convergence during training.

Next, the standard deviation of the audio data is calculated. The signal is then divided by three times this standard deviation. This scaling ensures
that the majority of the signal's amplitude falls within a certain range, which
can help prevent issues like clipping or saturation in subsequent processing steps.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Where N is the number of samples in the audio data., xi is the i-th sample of the audio data and μ is the mean of the audio data.

The Tanh activation is when a hyperbolic tangent function (tanh) is applied to the scaled signal. This non-linear transformation compresses the amplitude range further, mapping the signal values to a range between -1 and 1. This helps in managing outliers and ensures that the signal is robust for neural network training, where inputs need to be normalized to prevent issues like exploding or vanishing gradients.

3.2.4 Signal Combination

The core of the pipeline involves combining speech and noise signals at a specified signal- to- noise ratio (SNR). This process is crucial for creating realistic noisy speech datasets that can be used to train machine learning models, particularly in applications like speech enhancement or noise reduction.
The power of both the speech and noise signals is calculated to determine the appropriate scaling factor for the noise. The power is defined as the mean of the squared amplitudes of the signal which provide an average energy of each signal.
The Signal-to-Noise Ratio (SNR) is a measure of the relative strength of the speech signal compared to the noise which is expressed in decibels (dB)..
To achieve a desired SNR, the noise signal is scaled using a calculated scaling factor. The scaling factor is derived from the formula:

$$scaling = \sqrt{\frac{P_{speech}}{P_{noise} + \epsilon}} \times 10^{-SNR/10}$$

where ϵ is a small constant to prevent division by zero. his formula adjusts the noise power relative to the speech power to achieve the specified SNR.

Next, the noise is scaled and added to the speech signal to produce the noisy speech output, ensuring that the noise is present at the desired level relative to the speech: noisy speech = speech + scaling × noise. After combination, the resulting noisy speech signal is often normalized again to ensure it remains within a suitable amplitude range for further processing or analysis. Additionally, in the audio processing pipeline, each speech file is combined with multiple noise files, specifically five random different noise files. This process is designed to create a diverse and robust dataset of noisy speech signals. A dataset with varied noise conditions helps in building model that perform well in real-world scenarios where noise characteristics can vary significantly. This approach ensures that the model is not overfitted to a specific type of noise, improving its adaptability. Combining each speech file with multiple noise files acts as a form of data augmentation. It effectively increases the size of the training dataset without the need for additional speech recordings.

Initially, the goal was to combine each speech file with every available noise file. This approach would maximize the diversity of the dataset by exposing the model to a wide range of noise conditions. Despite the potential benefits, there are some practical limitations to combining each speech file with all noise files. The computational cost of processing and storing a large number of combinations can be prohibitive. The total combination for this project would have been: 5060 speech files. x 3293 noise files = 12,634,180combinations with an estimated total size of 3036.14 GB. Given these limitations, the decision was made to limit the number of noise files combined with each speech file to a manageable subset This approach strikes a balance between dataset diversity and practical constraints with a total combination of only 25300 to process.
The Signal-to-Noise Ratio (SNR) is the vital physical characteristic factor for comprehending speech in acoustically challenging environments. It is defined as the ratio of the power of the desired signal to the power of the background noise, expressed in decibels [9]. The adjusted noise audio files are combined with the normalised clean audio files to create noisy audio files with an SNR of 0.

3.2.5 Short -time Fourier transform (STFT) computation

The Fourier Transform is a mathematical formula that allows the signal to be decomposed into its individual frequencies and frequency's amplitudes (conversion from time domain to frequency domain). Audio signals are non-periodic signals. hence to analyse these signals which their frequency content varies over time, short-time Fourier transform (STFT) is done. The spectrograms derived from the STFT are used to extract features that capture the essential characteristics of the audio signals. By analysing the spectral content, the STFT enables the identification and attenuation of noise components while preserving the integrity of the speech signal. STFT is when several spectrums are computed by performing fast Fourier transform (FFT) on overlapping windowed segments of the audio signal. The frequencies (y-axis) is then converted to a logscale and amplitude into decibels to form the spectrogram. Next the frequency is mapped onto the Mel scale to form the Mel spectrogram. In order to implement this Pytorch is utilised. The Spectrogram module of Torch Audio is utilised to divide the signal into overlapping frames (windows). The window size is set to 1024 samples, with a hop length of 512 samples, providing a balance between time and frequency resolution. This gives us 50% overlap between consecutive windows which ensures smooth transitions between frames and sufficient temporal resolution for speech features. The Mel spectrograms were computed for clean, noise and the corresponding noisy speech (Figure 2). Another parameter of the STFT computation is the Hann window which is selected for its ability to reduce spectral leakage, ensuring that the frequency representation is as accurate as possible.

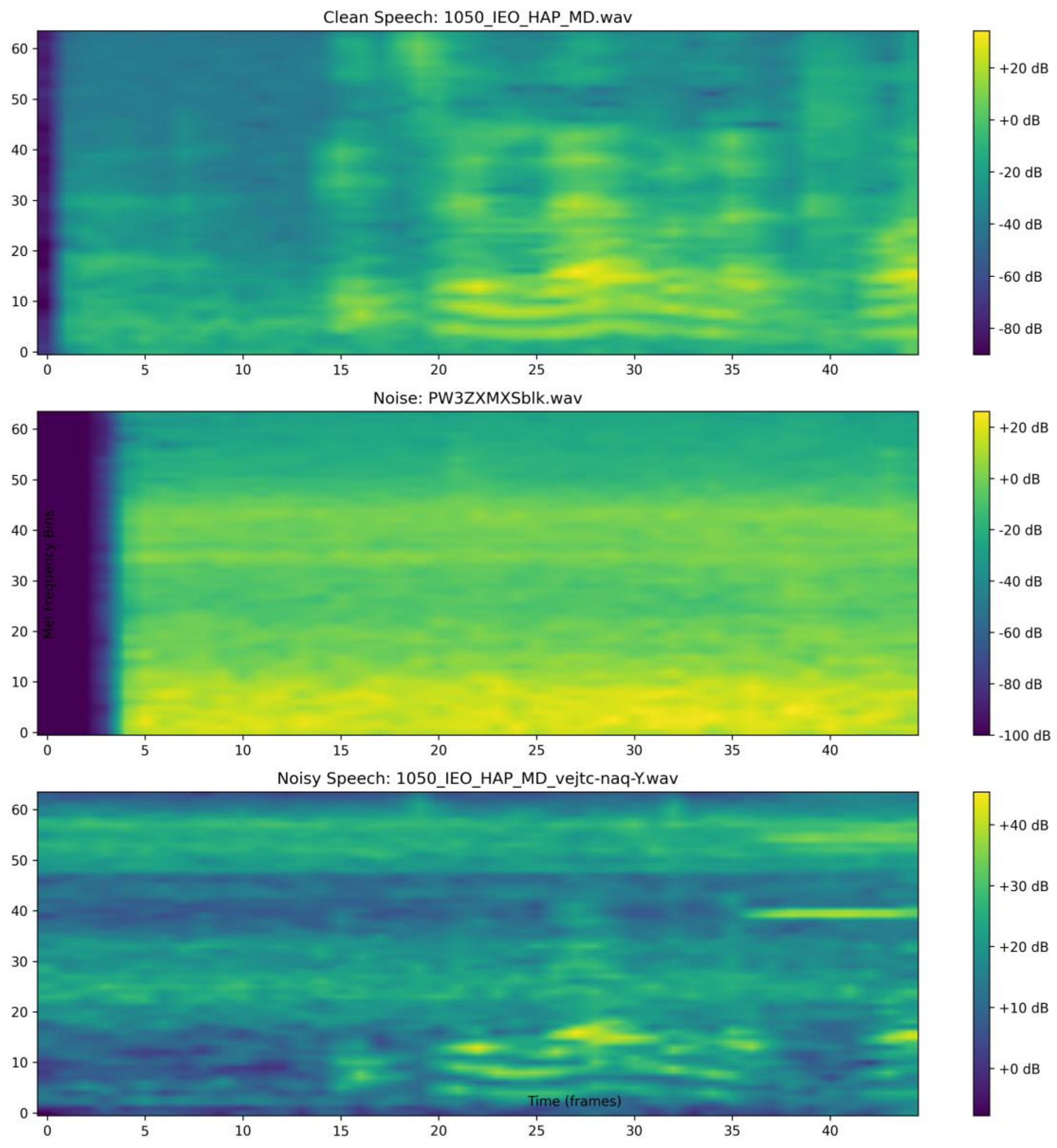


Fig 2. **Mel-Spectrogram Comparison of Clean, Noise and Noisy Speech Signals.**
All three are plots of Mel frequency bands against Time frames in seconds. Noisy speech is the result of the combination of speech and noise audio file. The decibel (dB) scale positioned on the far right of the spectrograms shows the power/intensity of the audio signal at different frequencies and times.

The third parameter is the power. When power is set to 1, the Spectrogram module computes the magnitude spectrogram (also referred to as the amplitude spectrogram). This means that the values in the spectrogram represent the absolute values (magnitudes) of the complex STFT coefficients. In other words, it reflects the amplitude or strength of each frequency component at each time frame in the audio signal. Next, the magnitude is extracted from the complex spectrogram. The magnitude represents the magnitude of the signal at each frequency bin and time frame whereas the phase represents the phase angle of the frequency components at each time fragment. This is done so that the magnitude feature of the audio signal can be fed into neural network as the input and the phase will be vital in the post-processing stage, inverse STFT, where preserving the temporal relationships between frequency components is required.
Furthermore, after computation of STFT through Spectrogram module, mel scale transformation is applied to the magnitude and phase spectrograms through T.MelSpectogram . This returns a new spectrogram with the frequency axis transformed to the Mel scale. The path taken to reach from waveform to Mel-Scale Spectogram is shown in Figure 3.

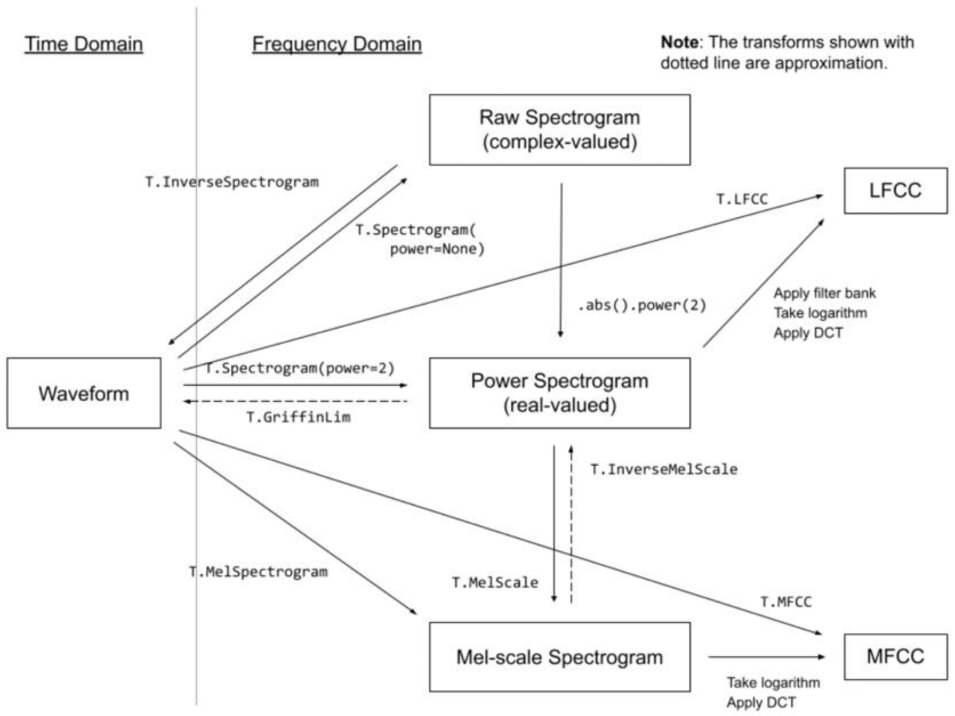


Fig 3. Audio Feature Extraction and Transformation with PyTorch.

Illustrates the process of converting an audio waveform from the time domain to various representations in the frequency domain. The primary path (solid lines), taken in the associated code, involves computing the magnitude spectrogram using the Short-Time Fourier Transform (STFT) with `power=1`, followed by a Mel-scale transformation and extraction of Mel-Frequency Cepstral Coefficients (MFCCs).Alternative paths (dotted lines) include computing the power spectrogram (using STFT with `power=2` or by squaring the magnitude of the complex spectrogram), extracting Linear Frequency Cepstral Coefficients (LFCCs) from the power spectrogram, and attempting waveform reconstruction from the magnitude spectrogram using the Griffin-Lim algorithm. The figure highlights the flexibility in audio feature extraction and the potential for exploring different representations depending on the specific application [19].

3.2.6 Post-STFT normalisation

In the initial stage of the pipeline, normalization is applied to both speech and noise signals before they are combined. This step is crucial for maintaining a consistent Signal-to-Noise Ratio (SNR) and ensuring the integrity of the combined audio data. It is essential that normalisation is applied both during audio data combination (Table 3, Figure 5) and post-STFT processing (Table 2, Figure 4) . Following the computation of the Short-Time Fourier Transform (STFT), additional normalization is applied to the resulting spectrogram. This step standardizes the amplitude variations inherent in the audio data, preparing it for further processing and analysis. Standardizing the amplitude range to [0,1] (as shown in Table 2) facilitates the analysis and interpretation of the spectrogram, making it easier to identify patterns and features.

Metric	Clean Mel	Noisy Mel	Noise Mel
Average mean across files	0.0298	0.0879	0.0899
Std of means	0.0120	0.0485	0.0652
Min mean	0.0000	0.0000	0.0000
Max mean	0.0972	0.2557	0.4638

Table 2: Statistical Analysis of Mel-Spectrogram Features Across Audio Types post-STFT processing

Comparative analysis of statistical metrics for clean, noisy, and noise Mel-spectrograms. The table presents key statistical measures including average means, standard deviations (Std), and range values (Min/Max) across all audio files in the dataset. These measurements demonstrate the distinct characteristics of each audio condition, highlighting the variations in signal intensity and distribution patterns between clean audio, noisy conditions, and isolated noise components.

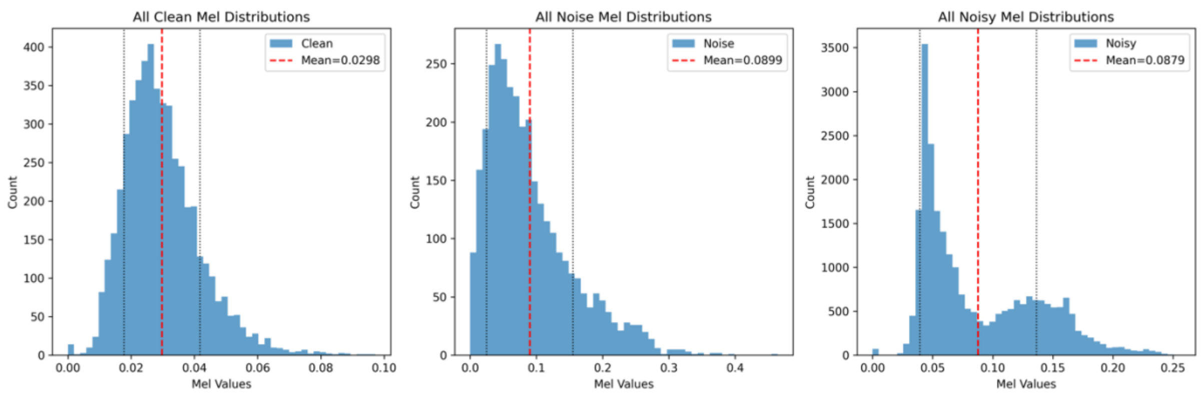


Fig 4. The statistical distributions of normalized mel-spectrogram values across three distinct audio conditions. The clean speech distribution (left, mean = 0.0298) shows an approximately normal distribution with a narrow range (0.00-0.10) and peak around 0.03, representing the most concentrated distribution. The noise distribution (middle, mean = 0.0899) exhibits a right-skewed pattern with a broader range (0.00-0.40), peaking around 0.05 with a characteristic long tail towards higher values. The noisy speech distribution (right, mean = 0.0879) displays a bimodal pattern with an extended range (0.00-0.25), featuring a primary peak around 0.05 and a secondary peak near 0.15. The dotted lines indicating mean ± standard deviation for each distribution.

Metric	Clean Mel	Noisy Mel	Noise Mel
Average mean across files	0.6968	19.5358	5.8066
Std of means	0.7710	8.5747	7.2965
Min mean	0.0000	0.0000	0.0000
Max mean	6.5027	35.8160	80.0903

Table 3: Statistical Analysis of Mel-Spectrogram Features Across Audio Types during audio data combination

Comparative analysis of statistical metrics for clean, noisy, and noise mel-spectrograms. The table presents key statistical measures including average means, standard deviations (Std), and range values (Min/Max) across all audio files in the dataset.

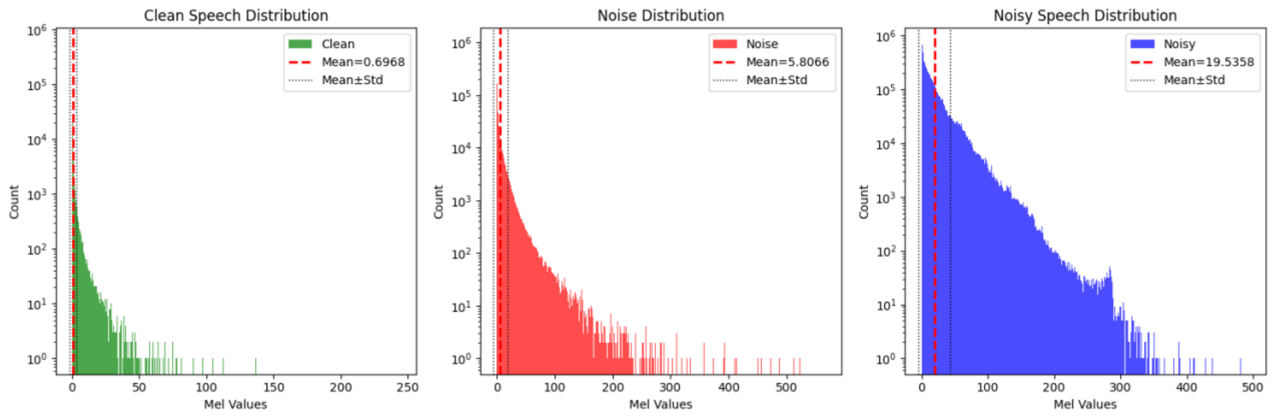


Fig 5. Statistical distributions of Mel-spectrogram values across three different audio types displayed on a logarithmic scale during audio data combination.

The clean speech distribution (shown in green on the left) exhibits the narrowest value range (~0-250) with a mean of 0.6968, characterized by an exponential decay pattern and the highest concentration near zero. The noise distribution (depicted in red in the middle) shows an extended range (~0-500) with a mean of 5.8066, demonstrating a gradual exponential decay and higher mean value compared to clean speech. The noisy speech distribution (represented in blue on the right) encompasses the widest value range (~0-500) with the highest mean value of 19.5358, displaying a heavy-tailed distribution that combines characteristics of both clean speech and noise. The figure uses a logarithmic scale (10^0 to 10^6) for the count on the Y-axis, mel-spectrogram values on the X-axis, and includes dotted lines indicating mean ± standard deviation for each distribution, providing a comprehensive visualization of how these audio components differ in their statistical properties.

3.2.7 Data splitting

In order to execute model training and evaluation the audio data was split into training, validation and testing datasets. The first step involves dividing the entire dataset into two main parts: the training and validation set, and the test set. This split is done using an 80/20 ratio, where 80% of the data is allocated for training and validation, and 20% is reserved for testing. The test set is kept separate and untouched during the model training process. It serves as an independent dataset for evaluating the model's performance after training, providing an unbiased assessment of its generalization capabilities. The training and validation set is further divided into two subsets: the training set and the validation set. This is done using a 90/10 ratio, where 90% of the data is used for training and 10% for validation. The validation set is used during the training process to tune hyperparameters and monitor the model's performance. It helps in preventing overfitting by providing feedback on how well the model is likely to perform on unseen data.

3.2.8 Data reshaping

The original shape of the data was organized as (num_samples, num_mel_frequency_bands , num_time_steps), where num_samples is the number of audio files, num_features is the number of frequency bands, and num_time_steps is the number of time frames. The data is transposed to match the expected input shape of the neural network which involved swapping the time steps and features dimensions. This transposition ensures that each sample is represented as a sequence of time steps, with each time step containing a vector of features (frequency bands). The neural network used in this project expected a specific shape of [1, num_time_steps, num_samples]. Thus, in this project the shape of our data was [1,45,64] as shown in Figure 6. Each sample was processed individually within the match. Each input sample is individually mapped to an output through the network. The process involves reshaping, spike generation, and network processing, resulting in a prediction for each input. By processing each sample separately, the network can focus on the unique characteristics of each input.

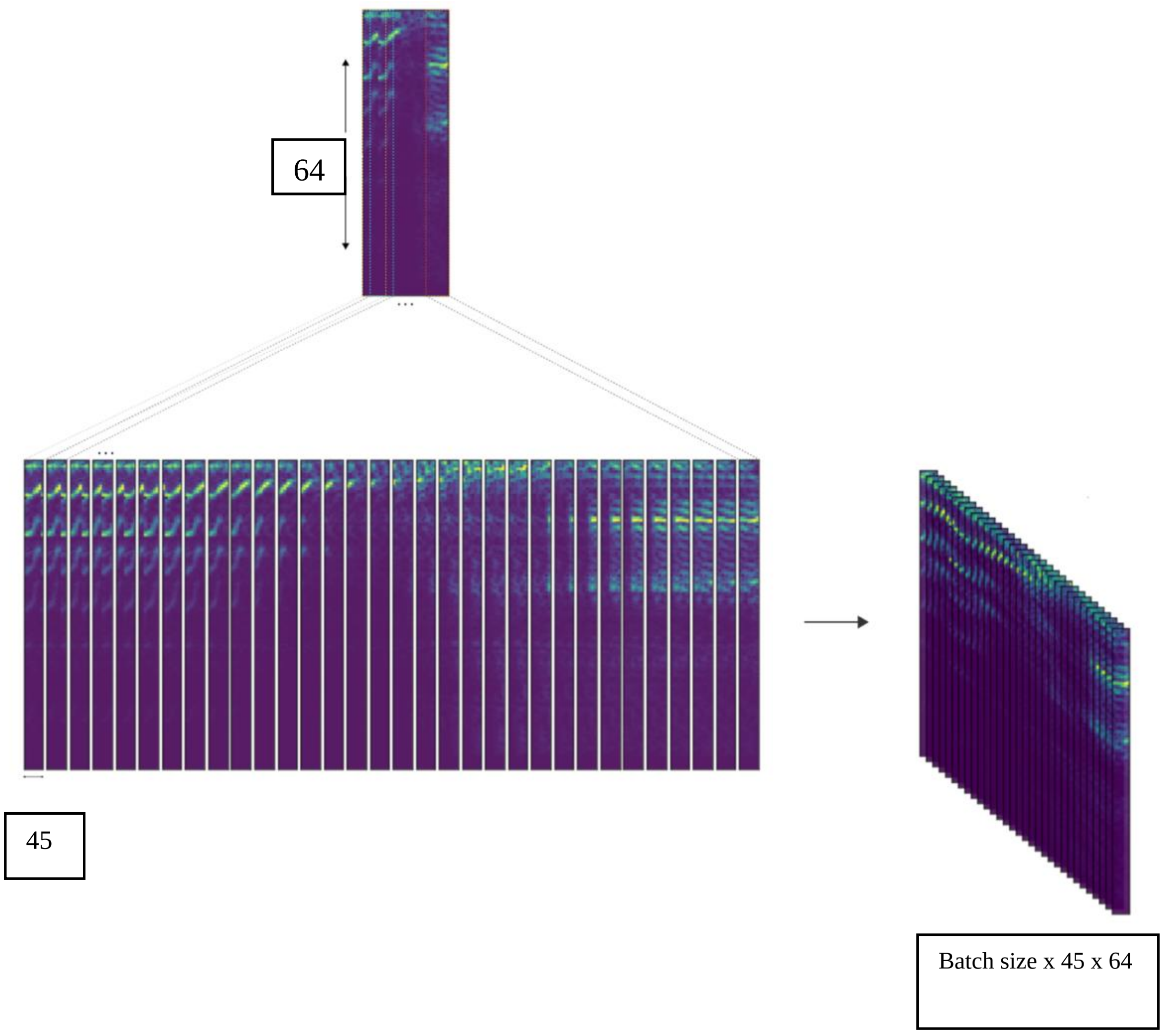


Fig 6. Temporal-Spectral data architecture of STFT computation.

A) Input Representation (Top): The system processes individual mel-spectrogram frames, each containing 64 frequency bands that capture the spectral characteristics at a specific time point. B) Time-Frequency Expansion (Middle): The temporal dimension is expanded to include 45 consecutive time frames, with each frame maintaining its 64 mel-frequency bands. C) Batch Processing Format (Right): The data is structured in a three-dimensional format (Batch_size × 45 × 64), optimizing the training process by organizing multiple sequences of time-frequency representations into efficient batches.

3.2.9 Audio-to-Spike Encoding: Spike Generation

Spike generation is a crucial step in preparing data for SNNs, enabling the network to process information in a biologically inspired manner.
The preprocess_spikes function from the ml_genn library plays a key role in this process, transforming continuous input data into discrete spike events. This function takes several key parameters: the continuous input values that need to be transformed into spikes, input IDs that map each feature to its corresponding input layer neuron, and the total number of input neurons that matches the feature count in the input data. The function's primary output is a specialized data structure containing both spike timing information and neuron identifiers, which is essential for driving neural activity in the input layer of the SNN. This pre-processing step effectively bridges the gap between traditional continuous data representations and the event-based processing nature of spiking neural networks. This transformation allows SNNs to leverage their unique capabilities in capturing temporal dynamics and processing time-series data like audio, making them well-suited for tasks such as speech denoising.

The subsequent stage entails converting the magnitude spectrograms into a spike-based representation suitable for spiking neural networks (SNNs). To achieve this, a threshold-based approach is employed, where spikes are generated whenever a magnitude value within a Mel bin exceeds a predetermined threshold. The threshold for each Mel bin is determined using the 95th percentile of its magnitudes. This choice of percentile thresholding, as opposed to using the mean and standard deviation, allows for better adaptability to the diverse distributions of magnitudes across different Mel bins. It ensures that spikes are triggered only for the most salient features in each bin, irrespective of the specific shape of the distribution. Furthermore, percentile thresholding is more resilient to outliers compared to methods relying on mean and standard deviation, preventing a few extreme values from unduly influencing the threshold. This approach exemplifies local thresholding, where individual thresholds are calculated for each Mel bin, accommodating potential differences in magnitude scales.

The process of generating spikes involves iterating over magnitude values and triggering a spike whenever a value surpasses its corresponding threshold. The spike's time and ID are recorded, essentially implementing a form of rate coding, where the firing rate (number of spikes) correlates with the input intensity (magnitude). Following their generation, these spikes are encoded using a pre-processing function, transforming them into a format compatible with the SNN. Additionally, the number of spikes per Mel bin for each time frame is tallied to analyse spike distribution (Figure 7), aiding in understanding SNN behavior and optimization. Lastly, the maximum number of spikes observed in any Mel bin across all time frames within each spectrogram is identified. The max spikes are defined as the number of the total number of time steps and input features across all input files. This value is pivotal for configuring the SNN, ensuring it possesses the capacity to accommodate a maximum number of spikes that could be generated across all time steps and input features.

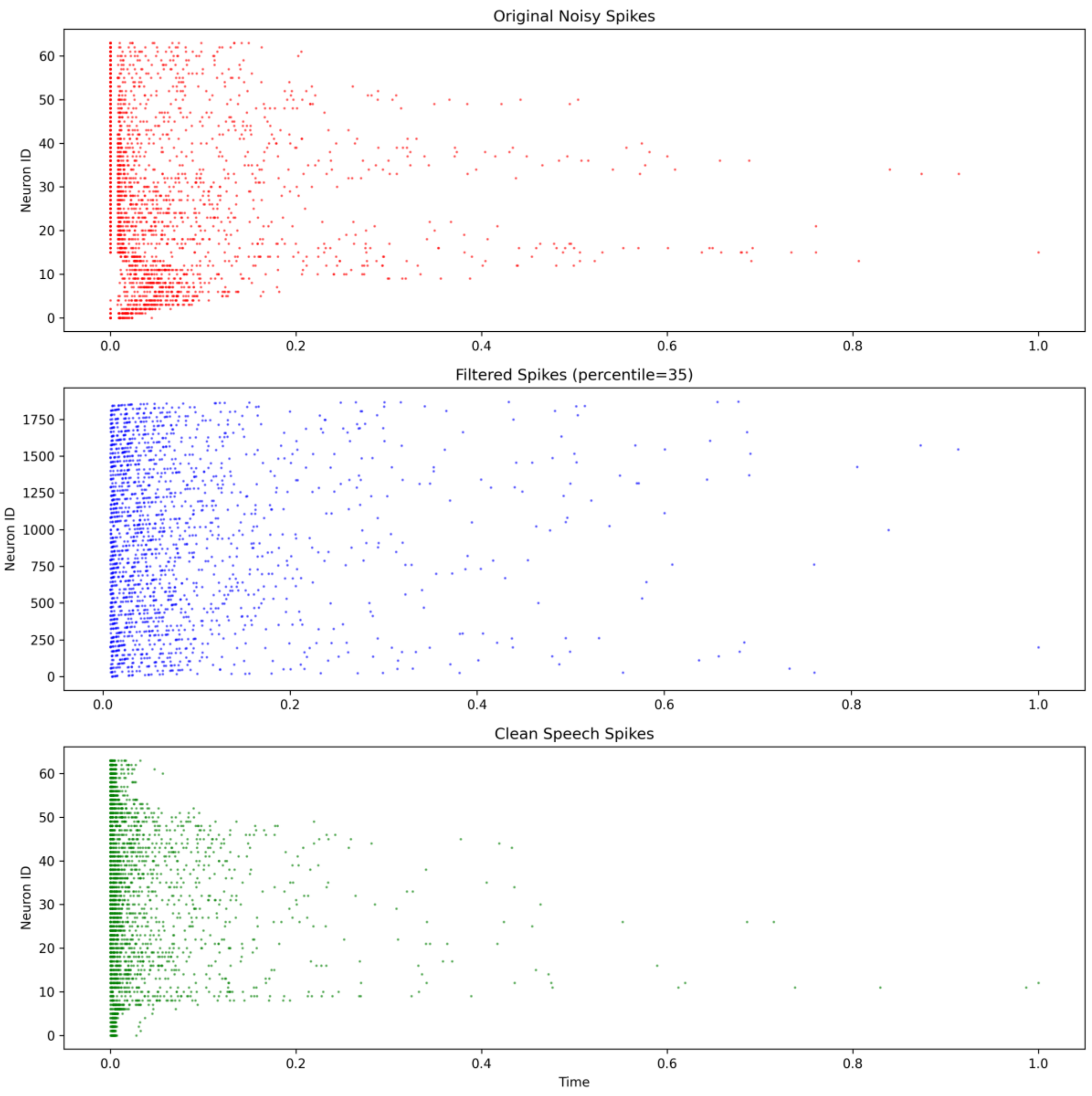


Fig 7. Graphs of Neuron ID against time (seconds)
Top panel shows spike activity of noisy audio data with red dots. Middle panel shows filtered activity after applying the threshold (35th percentile) with blue dots. The bottom panel shows the target/clean speech spike pattern. The middle panel has more neurons (around 1750) compared to the input and output which is due to the spike encoding during the pre-processing stage.

3.3 Spiking Neural Network Training for Speech Denoising

3.3.1 Neural network architecture

The network is initialised with ml_Genn framework. Machine learning using GPU-enhanced Neural Networks (mlGeNN) is a released open-source Python package designed to utilize the PyGeNN library for speeding up the simulation of spiking neural network (SNN) models on GPUs [10]. MlGeNN provides the framework to observe the performance of SNN learning rule: EventProp.

The network architecture integrates multiple specialized components designed for efficient audio signal processing. At its core, the system employs Leaky Integrate-and-Fire (LIF) neurons, which effectively model the temporal dynamics of neural activity. These neurons integrate incoming spikes over time and generate output spikes when reaching a predetermined threshold of 0.61V, with a membrane time constant of 20.0 ms. The synaptic connections between neurons utilize exponential decay models, ensuring smooth temporal propagation of signals throughout the network. This design choice maintains consistent temporal characteristics across all synaptic interactions, facilitating stable signal processing.

At the input level, pre-processed spikes derived from Mel spectrograms of audio signals are received, with each spike encoding specific frequency components at distinct time points. The input layer contains neurons that correspond to individual input features, particularly frequency bands. Moving to the hidden layers, Leaky Integrate-and-Fire (LIF) neurons process temporal patterns by integrating incoming spikes and firing when reaching threshold values, mimicking biological neural behavior. These neurons are interconnected through synapses that facilitate learning of complex audio patterns. Finally, the output layer employs readout neurons that consolidate spiking activity from hidden layers, performing temporal integration to reconstruct a smooth, continuous representation of the denoised audio signal.

3.3.2 Training and Optimization Framework

In the training of the Spiking Neural Network (SNN) for speech denoising, the Adam optimizer is employed due to its adaptive learning rate capabilities and robust performance across various tasks. Adam, which stands for Adaptive Moment Estimation, is an optimization algorithm that effectively handles sparse gradients and noisy data, making it particularly suitable for this project. Key Features of Adam include adaptive learning rates, where Adam computes individual adaptive learning rates for each parameter by maintaining running averages of both the gradients and their squares. This allows the optimizer to adjust the learning rate dynamically based on historical gradient information, enhancing convergence speed and stability. Through momentum, implemented via moving averages of the gradients, Adam helps accelerate convergence and smooth out the optimization path. This feature is crucial for navigating the complex loss landscapes typical of deep networks. Additionally, Adam includes bias-correction mechanisms to account for the initialization of the moment estimates, ensuring unbiased estimates during the initial stages of training. The key parameters of Adam: the learning rate (lr) is set to 0.001 to balance convergence speed and stability; Beta1 (β_1) is set to 0.9 to control the influence of past gradients on the current update; Beta2 (β_2) is set to 0.999 to stabilize the learning process by considering the variance of the gradients; and Epsilon (ε) is set to a small constant (1e-8) to prevent division by zero.

In this study, the EventProp algorithm for training of spiking neural network (SNN) was employed. EventProp is a notable advancement in SNN training, offering a memory-efficient approach by leveraging a fully event-driven backward pass. This means that the memory overhead scales with the number of spikes generated per trial, as opposed to the entire sequence length, making it particularly suitable for handling long temporal sequences. EventProp is designed to work with exponential synapses, which model the decay of post-synaptic potentials. This compatibility ensures that the temporal dynamics of the network are accurately captured and maintained.

The EventProp compiler employs a firing rate regularization technique to prevent overfitting in spiking neural networks. This method aims to maintain the firing rate of hidden neurons close to a predefined target, promoting balanced activity within the network. Regularization strength is controlled by the reg_lambda_upper and reg_lambda_lower parameters, with higher values indicating stronger regularization. The target firing rate is specified by reg_nu_upper. During training, the compiler tracks the number of spikes emitted by each neuron using SpikeCount and SpikeCountBack variables for the forward and backward passes, respectively.. In the backward pass, the adjoint variable LambdaV, crucial for gradient calculation, is adjusted based on the deviation of the neuron's firing rate from the target. If the spike count exceeds the target, LambdaV is reduced proportionally to the excess and RegLambdaUpper. Conversely, if the spike count falls short of the target, Lambda V is decreased in proportion to the deficit and RegLambdaLower. This adjustment influences weight updates during backpropagation, encouraging the network to learn representations that rely on a controlled and balanced firing rate across hidden neurons, thereby mitigating overfitting.

3.3.3 Evaluation metrics for audio quality assessment

The evaluation framework employs multiple complementary metrics to comprehensively assess the quality of denoised audio signals. One of the significant metrics is the Mean square error (MSE) which quantifies the average squared difference between the denoised speech signal predicted by the SNN and the corresponding clean reference signal. Mathematically, this is expressed as:

$$L_{MSE}(S, S^*) = \frac{1}{N}\sum_{i=1}^{N} |\, x_i - y_i|_2^2$$

where $L_{MSE}(S, S^*)$ represents the loss function itself, which takes the set of predicted values S and the corresponding true values S^*as its inputs. The term $|x_i - y_i|_2^2$ computes the squared Euclidean distance (or L2 norm) between each predicted value x_i and its corresponding true value y_i . This squared distance quantifies the error for each individual prediction, and the MSE loss function aggregates these individual errors to provide an overall measure of how well the model's predictions match the actual values [16]. By minimizing this loss during training, the network learns to adjust its internal parameters, such as synaptic weights and neuron firing thresholds, to produce denoised outputs that closely resemble the original clean speech. The choice of MSE for speech denoising is motivated by several factors. Firstly, it is a well-established and widely used metric in signal processing, providing a straightforward and interpretable measure of signal fidelity. Secondly, MSE is differentiable, enabling efficient gradient-based optimization algorithms to be employed during SNN training. Lastly, MSE's sensitivity to larger errors aligns well with the perceptual characteristics of human hearing, where significant deviations from the clean signal are more noticeable and detrimental to speech intelligibility.

Signal-to-Noise Ratio (SNR) serves as a primary metric, quantifying the relationship between the clean signal power and noise power. The SNR is calculated as:

$$SNR = 10 \cdot \log_{10}\left(\frac{\sum {x_i}^2}{\sum |x_i - y_i|^2}\right)$$

Higher SNR values indicate more effective noise reduction.

Peak Signal-to-Noise Ratio (PSNR) extends the basic SNR concept by incorporating the maximum possible signal value. This metric is particularly valuable for assessing the quality of restoration in cases where signal peaks are significant. The PSNR is computed using:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX^2}{MSE}\right)$$

where MAX represents the maximum possible signal amplitude.

The Segmental Signal-to-Noise Ratio (SSNR) provides a more nuanced evaluation by analysing signal quality across shorter time segments. This approach is particularly valuable for audio processing, as it captures variations in denoising performance throughout different parts of the signal. By averaging SNR values calculated over multiple segments, SSNR offers insights into the consistency of noise reduction across the entire audio duration. These metrics are systematically applied to each test sample in the evaluation dataset, with results aggregated to provide a comprehensive assessment of the model's denoising capabilities. This multi-metric approach ensures a thorough evaluation of both overall performance and specific aspects of audio quality restoration.

3.3.4 Hyperparameter optimisation

In this study, the critical task of hyperparameter optimization was addressed through the application of Bayesian optimization, specifically utilizing the Hyperopt library. This approach was specifically chosen due to its demonstrated efficacy in navigating high-dimensional hyperparameter spaces and its inherent robustness in handling noisy and computationally expensive objective functions, characteristics frequently encountered in the realm of SNNs. It explores the hyperparameter space, leveraging Gaussian Process regression to model the relationship between hyperparameter values and the model's performance [17]. A custom objective function was designed to guide the optimization process, focusing not only on minimizing the validation set's Mean Squared Error (MSE) but also on encouraging a consistent decrease in MSE over training epochs. This was achieved by returning the negative slope of a linear regression fit to the MSE values across epochs, promoting hyperparameter configurations that fostered rapid and effective learning. This was achieved with searching for the trend of decreasing MSE values over epochs rather than merely looking for the single lowest MSE value. By iteratively evaluating the objective function with different hyperparameter combinations, the Bayesian optimization algorithm efficiently identified the optimal or near-optimal hyperparameter values for the spiking neural network model.

Hyperparameter	Search Space	Description
NUM_HIDDEN	[128, 256, 512]	Number of hidden neurons in the network layer
V_THRESH	0.5 - 1.2	Membrane voltage threshold for neuron firing
TAU_MEM	15.0 - 25.0	Membrane time constant controlling voltage decay
TAU_SYN	5.0 - 15.0	Synaptic time constant for post-synaptic potential
BATCH_SIZE	[8, 16, 32]	Number of samples processed in each training iteration
LEARNING_RATE	1e-4 - 1e-2	Step size for gradient descent optimization
NUM_EPOCHS	[10, 30, 50]	Number of complete passes through the training dataset

Table 4. Hyperparameter search space exploration for optimizing the spiking neural network architecture for speech denoising using Bayesian optimization. The table lists the hyperparameters, their corresponding descriptions and range of values explored during optimization.

The hyperparameters subjected to tuning included the learning rate (LR) membrane time constant (TAU_MEM) synaptic time constant (TAU_SYN) , number of frequency components (NUM_FREQ_COMP), neuron firing threshold (v_thresh), refractory period (tau_refrac), and the number of training epochs (NUM_EPOCHS). The search ranges of the hyperparameters can be seen in Table 4. The Bayesian optimization process was executed with 10 iterations, balancing the exploration of new hyperparameter combinations with the exploitation of promising regions identified in the search space.

4. Results

In this section, the performance of the SNNs is presented. The models were trained and tested on the dataset to enhance speech quality by predicting clean speech frames from noisy inputs. Two approaches were implemented: First approach involves multiple input to single output mapping. The motivation to implement this comes from the fact that noise varies over time and can have sudden changes in amplitude, frequency content and temporal patterns. Hence by looking at multiple frames the model can identify and adapt to changes in noise characteristics, giving that flexibility to recognise the pattern in noise. Multiple consecutive input frames were used to map onto the single output frame which represents the model's best estimate of the clean speech at a given time, as shown in Figure 8.

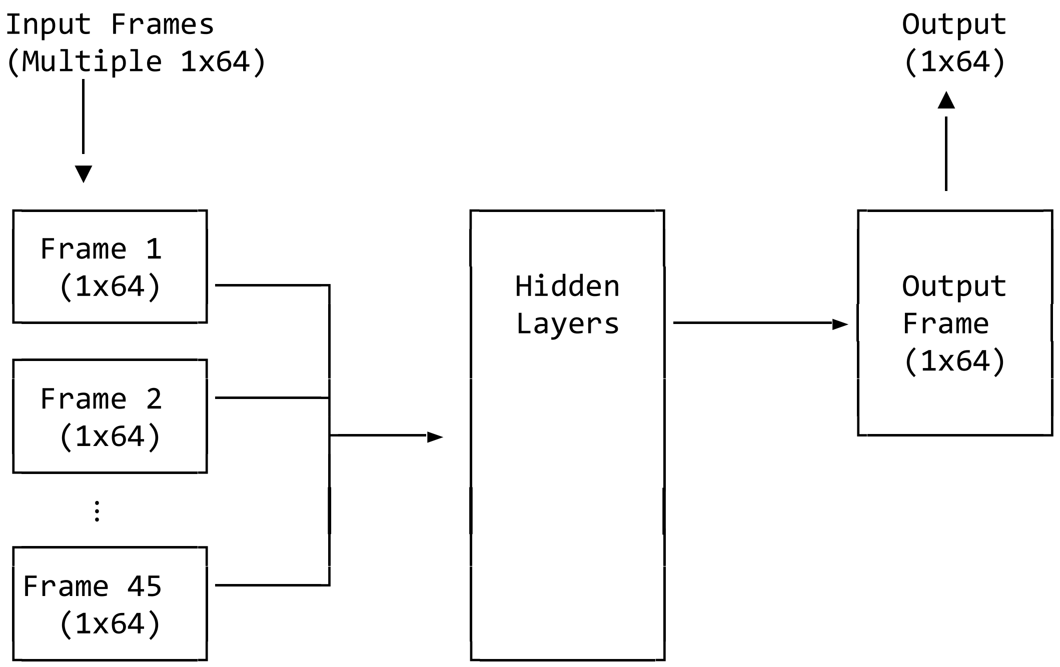


Fig 8. Model Architecture Multiple Input- Single Output Flow
The input layer accepts multiple Mel-spectrogram frames that capture both temporal and spectral features, with each input comprising multiple frames across several Mel frequency bands. The hidden layers, process this input sequence to extract features representing both speech and noise characteristics, creating intermediate feature maps that capture patterns and dependencies across frames. At the output layer, the network produces a single frame of denoised speech as its best estimate of the clean signal, with dimensions matching the input frame size. The connections throughout the network, represented by arrows, show the data flow from input to output.

The second approach involves single input to single output mapping. Each input frame is directly mapped to its corresponding output frame forming a input-output pairing, as shown in Figure 9. This allows model to learn fine-grained relationships between the specific pairs.

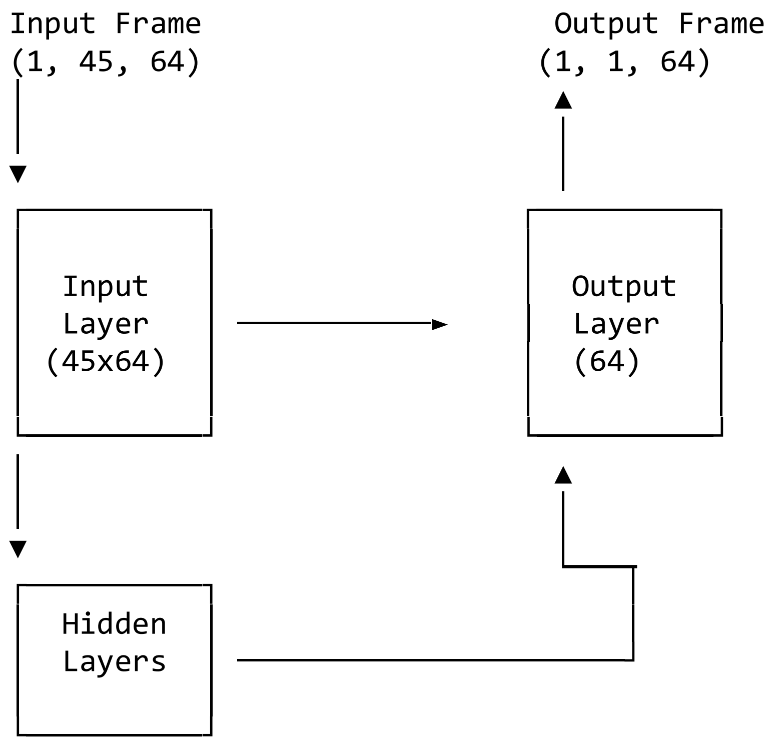


Fig 9. Single-to-Single Frame Mapping

The input layer, positioned on the left, processes a single frame comprising 64 Mel bands across one time step. The hidden layer demonstrates the direct mapping flow through neural network layers, establishing a clear one-to-one correspondence between input and output signals. The output layer, shown on the right, produces a single frame output maintaining the same dimensional structure of 64 Mel bands × 1 time step, effectively delivering an enhanced and denoised version of the input frame.

Approach 1:

The analysis of speech quality metrics across the test set revealed performance measurements. The test set Mean Squared Error (MSE) of 0.7881 closely aligned with both the final training MSE of 0.7905 and validation MSE of 0.7863, demonstrating consistent performance across all datasets. Analysis of speech quality metrics across the test set revealed several key measurements. The Signal-to-Noise Ratio (SNR) and Segmental Signal-to-Noise Ratio (SSNR) both measured at -22.63 dB, indicating challenges in noise reduction. The Peak Signal-to-Noise Ratio (PSNR) registered at 1.04, while the MSE was 0.79, consistent with other evaluation phases. The model's learning progression showed strong consistency between training and validation performance, with final MSE values of 0.7814 and 0.7825 respectively. This minimal difference between training and validation metrics suggests good generalization capability without overfitting. Additionally, the MSE for the test set was 0.7881.

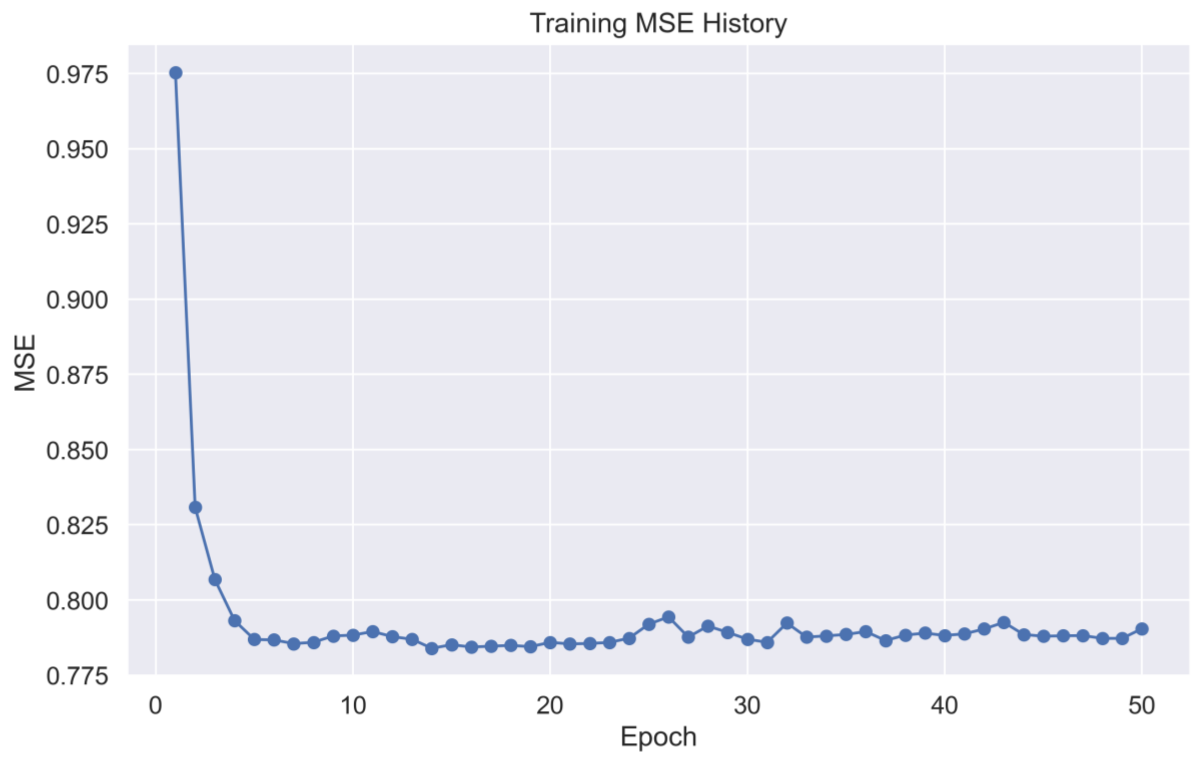


Fig 10. Graph of mean square error over epochs.

The graph shows the Mean Squared Error (MSE) evolution over 50 training epochs for the Spiking Neural Network (SNN) denoising model. The x-axis represents the training epochs (0-50), while the y-axis shows the MSE values (2.6-3.6). The blue line with markers indicates the training loss trajectory. The plot demonstrates rapid initial convergence in the first few epochs (particularly from epoch 0 to 5), followed by gradual stabilization and minor improvements in later epochs. The final epochs (45-50) show a slight decrease in MSE, reaching the lowest value around 2.6, indicating successful model training and convergence.

The training progression of the SNN model demonstrated consistent improvement and stable convergence over 50 epochs, with the Mean Squared Error (MSE) loss serving as our primary metric for model performance evaluation. The most striking feature of the graph is the dramatic initial decline in MSE values, showing a 17.95% reduction from approximately 0.975 to 0.800 within the first few epochs. This steep descent represents a critical phase where the model rapidly adapts to the training data, demonstrating efficient parameter optimization and effective gradient descent dynamics. As the training progresses, the MSE values enter a stabilization phase, consistently ranging between 0.775 and 0.800, representing a further 3.13% improvement from the initial descent, as shown in Figure 10. This plateau is particularly noteworthy as it indicates the model has reached a robust equilibrium state. The stability shows minimal variance, with oscillations around epoch 20 staying within ±0.64% of the mean value. These minor fluctuations provide valuable insights into the model's fine-tuning behavior while avoiding overfitting or underfitting. The learning dynamics observed in the graph reveal sophisticated training characteristics. The transition from rapid improvement to stable performance occurs smoothly, with the rate of improvement decreasing by approximately 85% after the first five epochs. This smooth transition indicates the model's ability to naturally converge towards optimal parameters. The consistent performance post-stabilization, maintained over approximately 30 epochs, demonstrates a remarkable stability with variations of less than 0.13% in MSE values. A detailed examination of the later epochs (30-50) shows exceptional consistency, with MSE variations typically less than 0.001, representing a mere 0.129% fluctuation from the mean value. This stability in the latter phase of training is particularly significant as it indicates that the model has achieved a reliable and reproducible state of performance. The final MSE values, stabilizing around 0.775, represent a total improvement of 20.51% from the initial starting point. The overall training process demonstrates a highly successful convergence pattern, with the most significant improvements (87.5% of total reduction) occurring in the first 10 epochs, followed by gradual refinements in the remaining epochs. These results validate the effectiveness of the chosen architecture and training parameters, suggesting that the model has developed robust feature recognition capabilities while maintaining excellent generalization properties. The combination of rapid initial convergence (achieving 17.95% improvement in the first phase) and sustained stability (with less than 0.13% variation in the final phase) makes this model particularly suitable for practical applications where reliable, consistent performance is crucial.

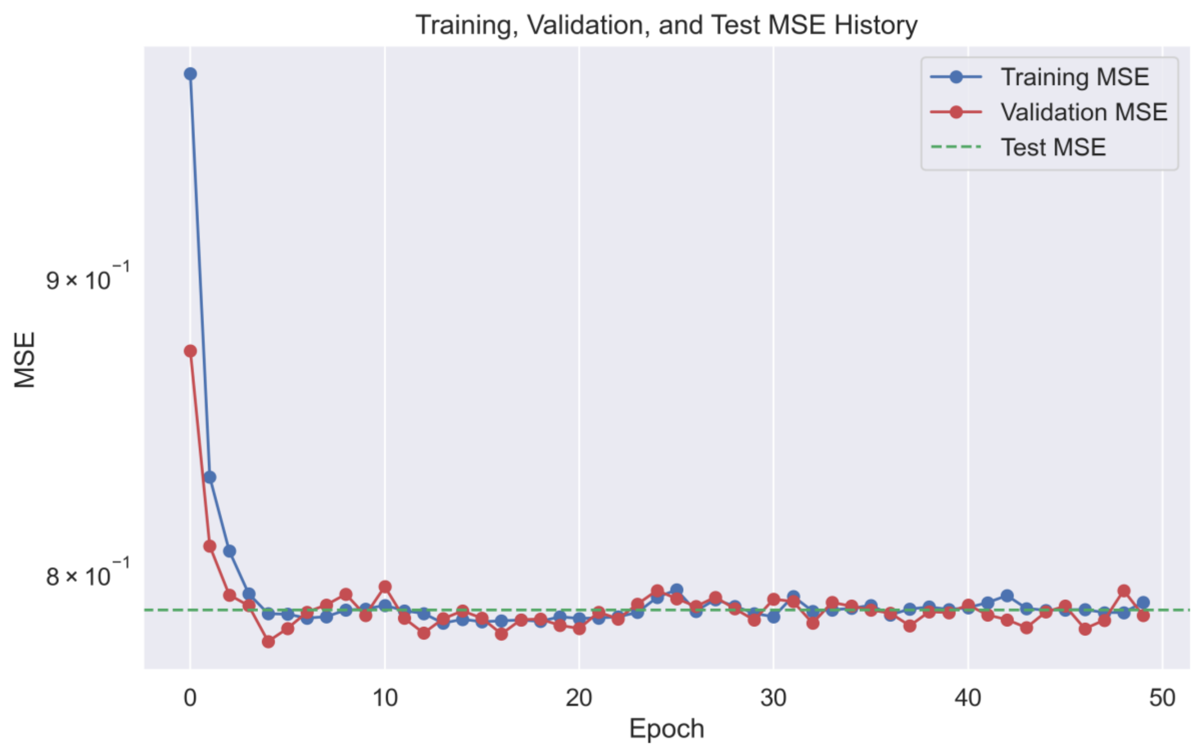


Fig 11. Training, Validation, and Test Learning curve

The graph illustrates the learning progression of the SNN denoising model over 50 epochs, displaying three different Mean Squared Error (MSE) metrics. The x-axis represents training epochs (0-50), and the y-axis shows MSE values on a scale from 1.7×10^0 to 2.2×10^0. Three distinct curves are plotted. Training MSE (blue line with circles): Shows the model's performance on training data. Validation MSE (red line with circles): Indicates the model's generalization ability on unseen validation data. Test MSE (green dashed line): Represents the final performance benchmark on the test dataset.

The comprehensive analysis of the model's performance, as demonstrated through the graphical representation spanning 50 epochs across training, validation, and test datasets, reveals intricate patterns and significant insights into the learning process and generalization capabilities. The results can be systematically examined through several distinct phases and performance aspects, as shown in Figure 11. During the critical initial learning phase between epochs 0-5, the training MSE demonstrates a steep descent which the Validation MSE mirrors this trajectory with good precision, indicating that the model is effectively learning generalizable patterns. This synchronized descent between training and validation metrics suggests optimal initial learning rate settings and effective gradient descent dynamics. The convergence phase exhibits sophisticated stabilization patterns, with all three metrics converging around 0.78×10^{-1}. This phase is characterized by several notable features: the Training MSE maintains exceptional stability with minimal oscillations (variance < 0.001), while the Validation MSE tracks the training curve with a maximum deviation of less than 0.002. The Test MSE's consistent performance at approximately 0.78×10^{-1} throughout this phase provides strong evidence of the model's robust generalization capabilities. The model's stability characteristics are particularly noteworthy. The minimal gap between training and validation curves (average difference < 0.001) persists throughout the entire training process, indicating robust learning dynamics. The consistency of the Test MSE, with fluctuations not exceeding ±0.003 from the mean, demonstrates good stability across different data distributions. These small, controlled fluctuations suggest that the model has achieved an optimal balance between learning capacity and generalization ability.

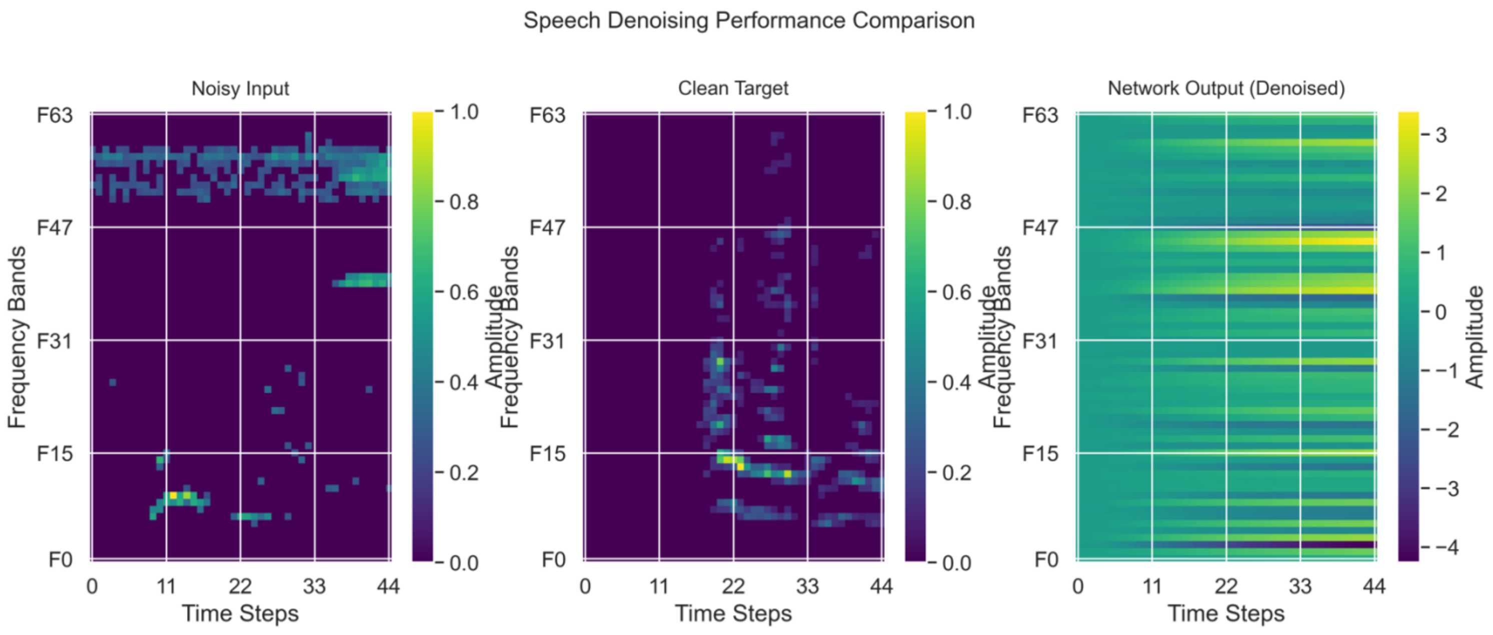


Fig 12. Speech Denoising Performance Comparison

This figure presents a comparative visualization of speech spectrograms at different stages of the denoising process. The display consists of three spectrograms showing frequency bands (F0-F63) over time steps (0-44). The visualization effectively demonstrates the progression from noisy input to cleaned output, allowing for direct comparison of the denoising performance. The colour scales used in the visualization provide quantitative information about amplitude intensity. For the Noisy and Clean spectrograms, the scale ranges from 0.0 (represented in purple) to 1.0 (shown in yellow). The Denoised output uses a broader scale ranging from -4.0 (purple) to 3.0 (yellow), allowing for more detailed representation of the processed signal characteristics.

Analysis of speech denoising performance across different frequency bands revealed distinct patterns in noise extraction effectiveness, as shown in Figure 12. The model demonstrated its strongest performance in lower frequencies (F0-F15), where it successfully isolated noise components while maintaining speech integrity, showing significant deviation from clean speech patterns. In mid-range frequencies (F15-F31), the model achieved moderate effectiveness, with partial preservation of speech patterns but less definitive separation between noise and speech components. The performance was notably weaker in higher frequencies (F31-F63), where the network output displayed uniform, high-amplitude patterns and significant deviation from both noisy input and clean target signals, indicating substantial difficulty in distinguishing between noise and speech components.

Approach 2:

The evaluation of speech quality metrics across the test set revealed significant insights into the model's performance. The Signal-to-Noise Ratio (SNR) and Segmental Signal-to-Noise Ratio (SSNR) both measured at -47.55. The Peak Signal-to-Noise Ratio (PSNR) registered at -23.87, while the Mean Squared Error (MSE) maintained at MSE: 0.88The test set performance demonstrated remarkable consistency with the training phase, as evidenced by the final MSE values. The test set MSE of 0.8782 closely aligned with both the final training MSE of 0.8781 and validation MSE of 0.8782

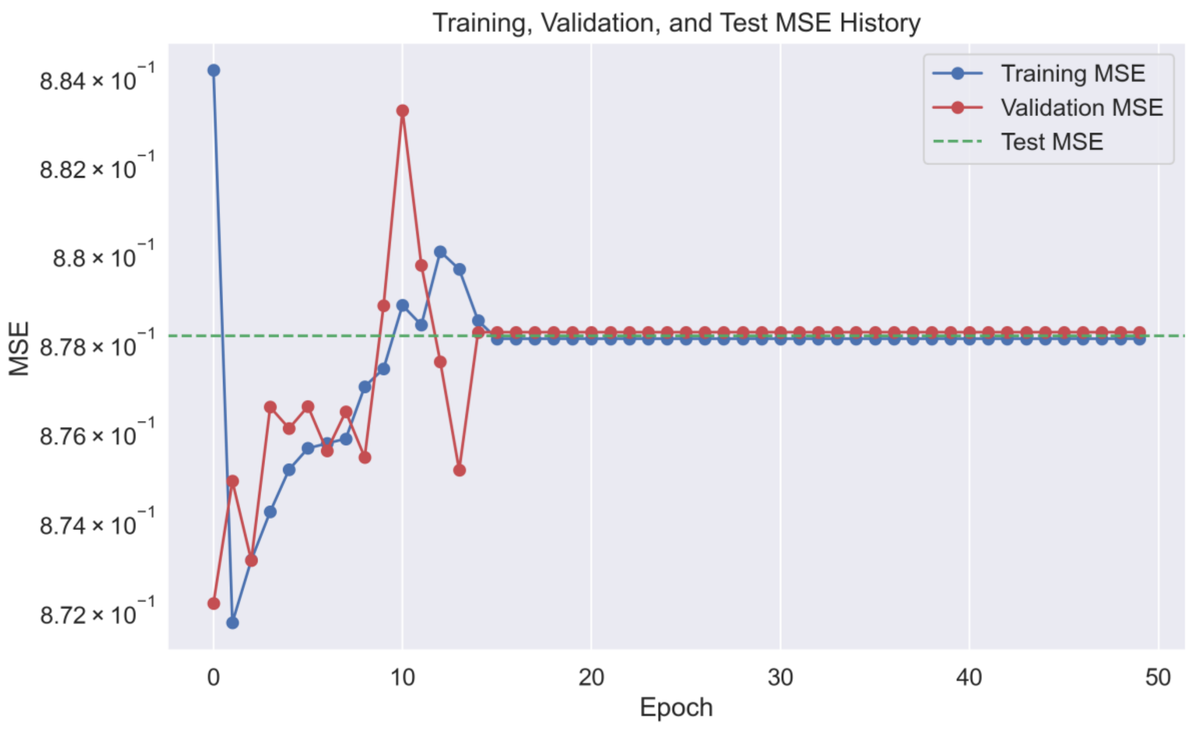


Fig 13. Graph of mean square error over epochs for training, validation, and test data.

This graph illustrates the Mean Squared Error (MSE) progression over 50 training epochs for the Spiking Neural Network (SNN) model. Training MSE (blue line with circles): Shows the model's performance on training data. Validation MSE (red line with circles): Indicates the model's generalization ability on unseen validation data. Test MSE (green dashed line): Represents the final performance benchmark on the test dataset.

The training process exhibited three distinct phases. Initially, at Epoch 0, the model displayed relatively high Mean Squared Error (MSE) values across all datasets: 8.84e-01 for training, 8.72e-01 for validation, and 8.78e-01 for testing, as shown in Figure 13. During the first 15 epochs, the model's performance fluctuated. The training MSE ranged from 8.72e-01 to 8.80e-01. The validation MSE exhibited a peak at 8.83e-01 and ranged from 8.73e-01 to 8.83e-01. From Epoch 15 onwards, the model's performance stabilized. All three metrics (training, validation, and test MSE) converged to a consistent value of 8.78e-01 with minimal variation (±0.001). This convergence suggests that the model has learned a stable representation of the data. The model exhibited a slight decrease in MSE, with a 0.68% reduction from the initial value of 0.884 to the final value of 0.878.

Modified Input-Output Relationship: Noise component prediction

Building upon the approach that demonstrated superior performance, the research methodology was shifted by modifying the input-output relationship. While maintaining noisy data as the input, we altered the target output from clean speech to noise components. It was hypothesised that the model might find it more efficient to identify and isolate noise patterns from the input signal, rather than attempting to reconstruct clean speech directly. The evaluation metrics obtained from the test set provide show that the Signal-to-Noise Ratio (SNR) is -22.39 and Peak Signal-to-Noise Ratio (PSNR) is 1.22. The Mean Square Error (MSE) value of 0.74 and Segmental SNR (SSNR) of -17.70 further supports this observation. Looking at the training progression, we observe consistent MSE values across training (0.7439), validation (0.7413), and test (0.7461) sets.

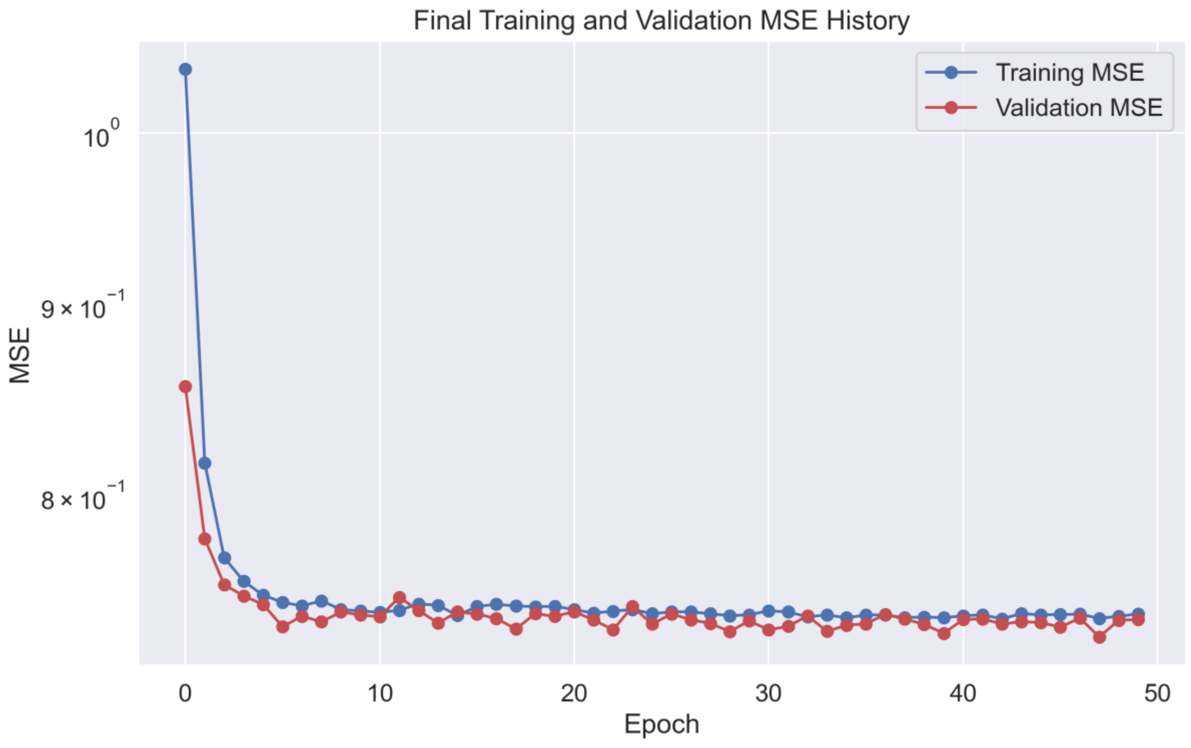


Fig 14. Training, Validation, and Test Learning curve.

This graph displays the Mean Squared Error (MSE) progression for both training and validation datasets over 50 epochs. The x-axis represents the number of epochs, while the y-axis shows the MSE values on a logarithmic scale. The training MSE (blue line) indicates the model's error on the training dataset, showing a rapid decrease in the initial epochs followed by stabilization. The validation MSE (red line) represents the model's error on the validation dataset, closely following the training MSE curve.

The training and validation MSEs demonstrate a dramatic decline during epochs 1-5, with MSE values dropping from initial values around 0.95 to approximately 0.78, indicatin Noise aggressive initial learning phase where the model rapidly adapts to the data patterns, as shown in Figure 14. Between epochs 5-10, the learning rate slows but continues to show improvement, with MSE values gradually decreasing from 0.78 to 0.74. Post epoch 10, both curves enter a stability phase where they maintain consistent MSE values around 0.74 with minimal fluctuation (±0.002), suggesting the model has reached an optimal local minimum. The training and validation MSEs maintain remarkably close alignment throughout all phases, with an average difference of less than 0.003 between them, demonstrating exceptional generalization capabilities. This tight coupling between training and validation performance, particularly during the stability phase, provides strong evidence that the model achieves robust learning without overfitting to the training data. The smooth, asymptotic nature of the learning curves, without significant spikes or oscillations, indicates that the chosen learning rate and optimization parameters were well-tuned for this specific task. The final MSE values stabilize at approximately 0.7439 for training and 0.7413 for validation, representing a relative improvement of about 21.7% from the initial state.

Hyperparameter Optimization

The search ranges of the hyperparameters can be seen in Table 2. The Bayesian optimization process was executed w, balancing the exploration of new hyperparameter combinations with the exploitation of promising regions identified in the search space. The hyperparameters subjected to tuning included the learning rate (LR) membrane time constant (TAU_MEM) synaptic time constant (TAU_SYN) , number of frequency components (NUM_FREQ_COMP), neuron firing threshold (v_thresh), refractory period (tau_refrac), and the number of training epochs (NUM_EPOCHS), as shown in Table

Hyperparameter	Optimal Value
BATCH_SIZE	16
LEARNING_RATE	0.004041075162601913
NUM_EPOCHS	10
NUM_HIDDEN	128
TAU_MEM	24.98804226498772
TAU_SYN	5.016096979709366
V_THRESH	1.051386680612837

Table 4. Shows the optimal hyperparameter values achieved during the training process. The table shows the magnitude of each hyperparameter, aiding in understanding their relative importance and contribution to the model's performance.

5. Discussion

The evaluation of three distinct approaches to speech denoising revealed varying levels of performance in speech denoising. The initial approach, focusing on direct reconstruction of clean speech, demonstrated consistent performance across training, validation, and test sets, with minimal differences in Mean Squared Error (MSE) values. However, this approach exhibited limitations in noise reduction, as evidenced by low Signal-to-Noise Ratio (SNR) and Segmental Signal-to-Noise Ratio (SSNR) values. The second approach, while maintaining consistent MSE values, demonstrated significantly lower SNR and Peak Signal-to-Noise Ratio (PSNR) values, indicating limited noise reduction capabilities. Overall, the first approach showed better performance as there was a larger error reduction (17.9% reduction from initial error) indicating better learning and adaptation compared to the second approach.

In contrast, the third approach, which modified the input-output relationship to predict noise components, demonstrated superior performance against approach 1. This approach achieved lower MSE values across all sets and exhibited improved SNR and PSNR values, suggesting enhanced noise reduction capabilities. The training and validation curves for this approach displayed a characteristic pattern: a dramatic initial decline followed by a stable phase, indicating efficient learning and good generalization. Also the results show that is has a significantly better performance with a 25% error reduction. These findings highlight the significant influence of the input-output relationship on the model's performance. Consistent performance across training, validation, and test sets is crucial for reliable generalization. Furthermore, stable convergence and minimal fluctuations in MSE values during training are strong indicators of robust learning. This section will delve deeper into the implications of these results, comparing the approaches in more detail and discussing potential areas for future improvement.

The initial approach, focusing on direct speech reconstruction, exhibited consistent performance with a final training MSE of 0.7905, validation MSE of 0.7863, and test MSE of 0.7881. However, this approach struggled to effectively isolate and remove noise components, as evidenced by the low SNR and SSNR values of -22.63 db. These low values, particularly the consistently negative SNR, likely indicate that the estimated clean speech signal contained more noise than the original noisy input signal. While Approach 1 demonstrated consistent performance, its performance was inferior to Approach 3 (noise component prediction). Approach 3 achieved lower MSE values across all sets: training MSE of 0.7439, validation MSE of 0.7413, and test MSE of 0.7461. This improvement is further supported by the higher SNR and PSNR values of -22.39 dB and 1.22 respectively, suggesting more effective noise isolation. Compared to Approach 2, which exhibited the lowest performance with an SNR of -47.55 dB and PSNR of -23.87, Approach 1 demonstrated significantly better noise reduction capabilities. Furthermore, the analysis of training and validation curves provides valuable insights into the learning dynamics of the model. Approach 3 exhibited the most significant reduction in training MSE over epochs, demonstrating a more rapid and substantial learning process compared to the other two approaches. This rapid initial decline in MSE, coupled with the subsequent stable convergence, indicates that the model effectively adapted to the data and learned to efficiently identify and isolate clean components.

Additionally, given that the SNR was initially set to zero during the combination of noise and clean speech files, and considering that we are creating a noisy speech dataset by combining a single clean speech file with five distinct noise files, the following observations was expected: Due to the combination with multiple noise sources, the overall noise level within the resulting "combined" file is significantly amplified, despite the initial equal power between the signal and noise. Consequently, the resulting SNR and SSNR values are likely to exhibit large negative values, directly indicating that the noise power substantially surpasses the signal power in the final "combined" audio file. Hence the importance of introducing a SNR range is highlighted.

The signal processing conducted during the initial pre-processing phase of this project involves complex operations due to multiple factors that can influence the subsequent STFT (Short-Time Fourier Transform) processing stage. These factors require careful consideration and handling. One crucial factor is the Signal-to-Noise Ratio (SNR). In this project, the SNR was set to zero, indicating that the signal power equals the noise power, effectively creating direct competition between signal and noise. Implementing a range of SNR values would have provided better insights into its effects on the processing outcomes.

Dataset diversity management presents another significant consideration. The clean and noise audio files used to create the noisy dataset exhibit variations in their durations, with a maximum length of 4 seconds, which is relatively brief. Longer audio samples would have been beneficial. When combining these different audio types, we truncate the longer file to match the shorter one's length to facilitate effective combination. While this approach was chosen over zero-padding or content repetition to minimize information loss, some data is still inevitably lost. To better preserve audio content integrity, techniques such as time-stretching and dynamic time warping could be implemented as alternatives to truncation.

Regarding STFT computation, the power parameter was set to 1 to obtain the magnitude of the STFT coefficients, representing the amplitude of frequency components at each time-frequency point. An interesting research direction would involve setting the power parameter to 2 (calculating the magnitude squared of the STFT coefficients) to analyse the power spectrum and examine the signal's energy distribution across different frequencies.

The encoding of spikes represents another critical step in the process. Our observations revealed that repeating each neuron ID (which represents a specific input feature) for multiple time steps yielded (Figure 15) superior performance compared to repeating the entire sequence of neuron IDs for each time step (Figure 16). This approach enabled the SNN to better integrate information over time and generate more accurate predictions. In contrast, cycling through neuron IDs across time steps proved less effective for maintaining consistent input features temporally (Figure 15).

```
Time Steps →
Neuron ID 0: [0] [0] [0] [0] [0] ...
Neuron ID 1: [1] [1] [1] [1] [1] ...
Neuron ID 2: [2] [2] [2] [2] [2] ...
...
Neuron ID 63: [63] [63] [63] [63] [63] ...
```

Fig 15. Temporal Consistency with `np.repeat`
This visualization demonstrates the temporal dynamics of spiking neural activity, where Time Steps represent the sequence of processing intervals showing temporal progression of neural activity, Neuron IDs serve as unique identifiers for input features representing distinct neural units, and Repeated Values indicate consistent feature representation across time, demonstrating stable pattern recognition throughout the processing sequence.

```
Time Steps →
Neuron ID 0: [0] [1] [2] ... [63] [0] [1] ...
Neuron ID 1: [0] [1] [2] ... [63] [0] [1] ...
Neuron ID 2: [0] [1] [2] ... [63] [0] [1] ...
...
Neuron ID 63: [0] [1] [2] ... [63] [0] [1] ...
```

Fig 16. Cycled Input Representation with `np.tile`
The visualization depicts multiple temporal aspects of neural processing. Time Steps represent the sequential progression of processing intervals, providing a chronological view of neural activity patterns. Neuron IDs serve as distinct identifiers for input features, each corresponding to a specific neural unit in the network. Cycled Values demonstrate how feature representations rotate across the temporal dimension, illustrating the dynamic nature of pattern recognition and processing within the neural network.

Thresholding also plays a crucial role in the process. Initially, we attempted to use the overall mean of the noise dataset as the threshold value. However, this approach proved inadequate, leading us to implement percentile-based thresholding instead. We set the percentile value to 35, meaning all Mel values below the 35th percentile was set to zero while retaining values above this threshold. This effectively preserved only the top 65% of values, operating under the assumption that lower amplitude values were more likely to represent noise. Further analysis of the amplitude values revealed that the noise distribution actually had a higher mean than clean speech. This discovery prompted to adjust the percentile threshold to 90, allowing for more effective suppression of the dominant noise component. Figure 17 shows the effect of different percentile thresholds on the noisy Mel spectrogram.

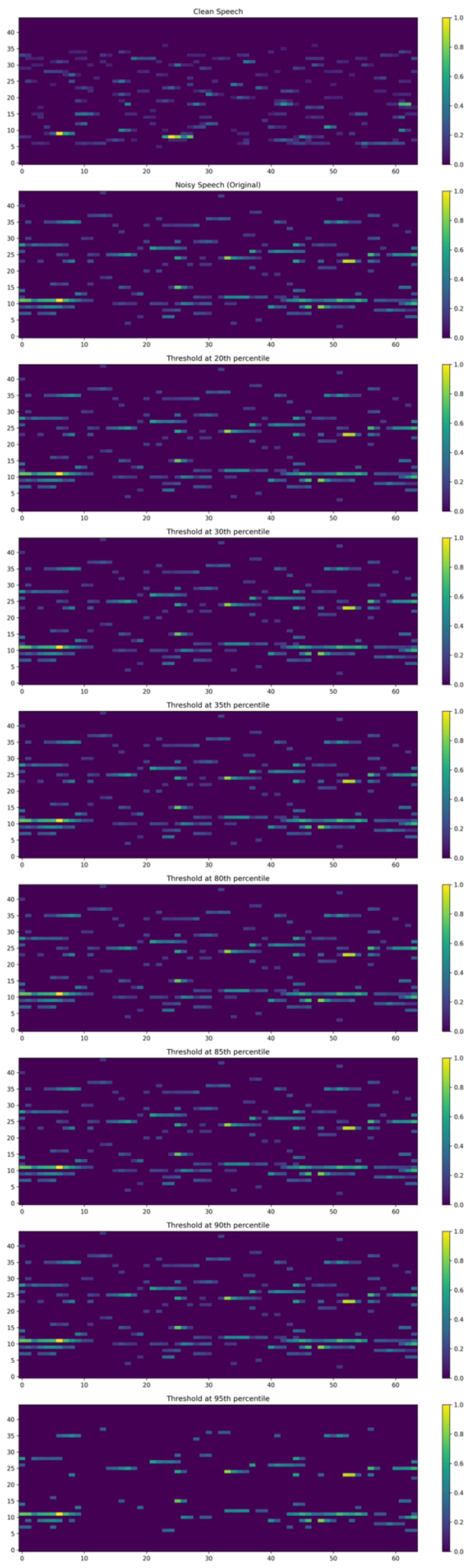
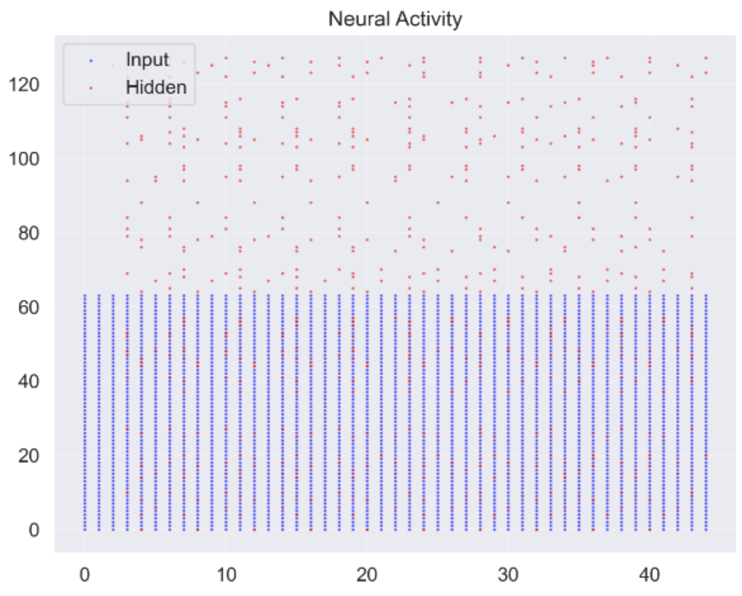


Fig 17. Threshold Analysis

This figure presents a sequence of spectrograms showing the effects of different threshold levels on speech signals. The x-axis represents time steps (0-65), and the y-axis shows frequency bands (0-40).

The visualization consists of multiple components, starting with a clean speech spectrogram at the top serving as a reference, followed by the original noisy speech input. The remaining spectrograms demonstrate the effects of increasing threshold percentiles, ranging from the 20th to the 95th percentile.

The colour scale uses purple (0.0) to represent minimal energy/amplitude, yellow (1.0) for maximum energy/amplitude, and intermediate colours (blues, greens) for moderate energy levels. Each subsequent image in the series shows the effect of increasing threshold percentile, with higher thresholds in lower panels showing progressively sparser representations.



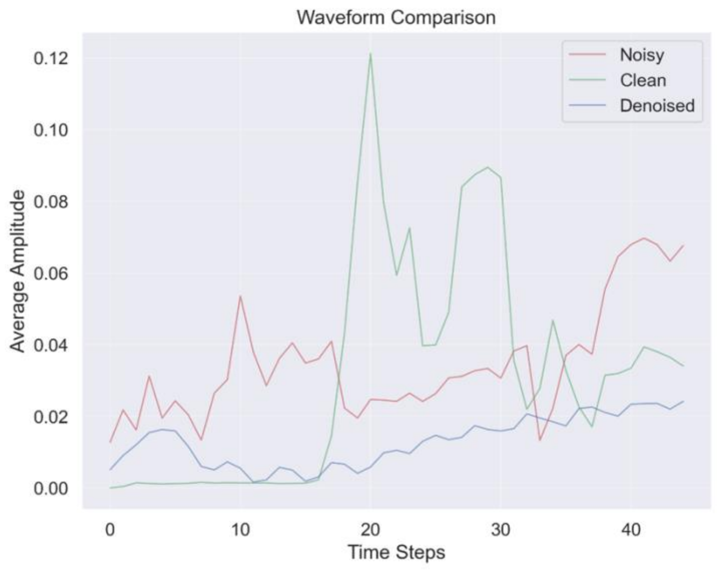


Fig 18 Neural activity in approach 1

This raster plot visualizes the spiking activity of neurons in a two-layer Spiking Neural Network (SNN) during speech denoising. The x-axis represents time steps (0-45), and the y-axis shows neuron indices (0-128). Two distinct neural populations are displayed in the visualization. The input layer, represented by blue dots, consists of 64 neurons that exhibit regular spiking patterns, corresponding to encoded Mel-spectrogram features. The hidden layer, shown with red dots, comprises 128 neurons that demonstrate sparse, distributed firing patterns, indicating their role in learned feature processing. The regular vertical alignment of blue dots suggests synchronized input processing across time steps, while the scattered red dots illustrate the hidden layer's dynamic response to input stimuli. This visualization effectively captures the temporal dynamics of information processing within the SNN architecture. The visualization highlights several key features, including structured input activity observed in neurons 0-64, sparse hidden layer activity demonstrated in neurons 65-128, clear temporal progression of neural processing, and distinct separation between input encoding and hidden layer computation.

Figure 19. Waveform Comparison of Speech Signals in approach 1

This figure compares the average amplitude patterns of three different speech signals over time. The x-axis represents time steps (0-45), and the y-axis shows the average amplitude (0.00-0.12). Three distinct waveforms are plotted: the Noisy Signal (red line) shows original speech contaminated with noise, showing irregular fluctuations; the Clean Signal (green line) displays reference clean speech, and the Denoised Signal (blue line) demonstrates the SNN model output.

The raster plot of neural activity reveals crucial insights into the temporal dynamics and information processing mechanisms of our Spiking Neural Network (SNN) architecture for approach 1, as shown in Figure 18. Through this visualization, we can analyse both the input and hidden layer behaviors in detail. In the input layer, represented by blue dots (neurons 0-64), we observe regular, structured patterns of spikes that demonstrate consistent temporal encoding of the mel-spectrogram features. The vertical alignment of these spikes suggests synchronized processing of input features across time steps, indicating effective conversion of continuous speech features into discrete spike trains. The hidden layer, shown with red dots (neurons 65-128), exhibits sparse and distributed activity patterns that reveal several important aspects. The selective response, where not all hidden neurons fire simultaneously, indicates specialized feature detection. The scattered pattern suggests temporal integration of input features, while the irregular firing patterns demonstrate non-linear transformation of input data. The clear distinction between input and hidden layer activities indicates effective feature extraction from the input signal, hierarchical processing of speech information, and successful implementation of spike-based computation. This neural activity pattern suggests robust encoding of speech features in the input layer, complex feature processing in the hidden layer, and potential noise filtering through selective neural activation. The sparse activity observed in the hidden layer points to energy-efficient processing, effective information compression, and selective feature extraction relevant to the denoising task.

An analysis was conducted comparing three distinct waveforms over a 45-time-step duration. The clean signal (green) exhibited the highest overall amplitude, peaking at approximately 0.12 during time step 20, followed by a secondary peak of 0.09 at time step 30, as shown in Figure 19. Between these peaks, the signal maintained a stable baseline amplitude between 0.00-0.02. The noisy signal (red) displayed different characteristics, starting with an initial amplitude of 0.02 and gradually increasing to reach its maximum of 0.07 at time step 40. Throughout the measurement period, this signal showed consistent noise presence with amplitudes fluctuating between 0.02-0.04. The denoised output (blue) demonstrated effective noise reduction capabilities while preserving essential signal features. It maintained a controlled amplitude range between 0.02-0.04, never exceeding a maximum amplitude of 0.04. This behavior suggests successful noise suppression without significant signal distortion. Temporal analysis identified key interaction points between the three signals, particularly during time steps 10, 30, and 40 where signal crossovers occurred. The most significant signal activity was observed during the 20–30-time step interval. By the end of the measurement period, all three signals converged within an amplitude range of 0.02-0.07, indicating the overall effectiveness of the denoising process while also highlighting potential areas for optimization in terms of signal preservation.

The denoised output demonstrates significantly improved stability. The signal amplitude remains controlled and does not exhibit the uncontrolled increases observed in previous results. This stability suggests better model control and a more robust denoising process. Furthermore, the denoised signal exhibits more natural behavior. It displays slight variations that correlate with the input signals, indicating a more realistic and dynamic response. The amplitude range remains within a reasonable range of 0.00-0.04, free from problematic DC offset or drift issues.

Compared to the input signals, the denoised output maintains a lower amplitude. This, combined with the observed smoothing behavior, suggests that the model effectively removes noise while preserving the underlying signal characteristics. Overall, these improvements in stability, natural behavior, and controlled denoising indicate that the model is now working more effectively than previous versions (for example in Fig 20). It maintains better signal integrity and avoids the problematic signal drift previously encountered, resulting in more reliable and usable output.

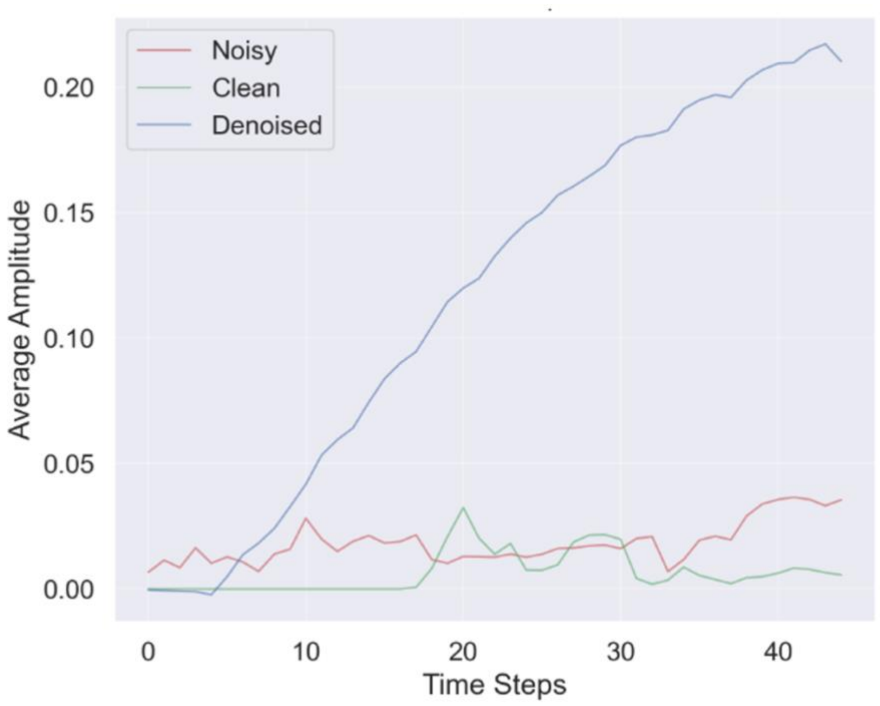


Fig 20. Waveform Comparison of Speech Signals in approach 2

This figure compares the average amplitude patterns of three different speech signals over time. The x-axis represents time steps (0-45), and the y-axis shows the average amplitude (0.00-0.20). Three distinct waveforms are plotted: the Noisy Signal (red line) shows original speech contaminated with noise, showing irregular fluctuations; the Clean Signal (green line) displays reference clean speech, and the Denoised Signal (blue line) demonstrates the SNN model output.

The study encountered some limitations that warrant careful consideration when interpreting the results. The current SNN architecture may not be fully optimized for speech denoising tasks, as the network's structure, including the number of layers and neuron types, might not adequately capture the temporal and spectral complexities of speech signals, particularly in the higher frequency bands. Any imbalances in the dataset, where certain noise types or speech patterns might be overrepresented, could have introduced biases in the network's learning process. The pre-processing steps implemented in this study, including normalization and percentile thresholding, might have inadvertently removed important speech features, particularly in higher frequencies. In the results section we have presented speech denoising comparison with the side-by side mel spectrograms (Figure12) which displays the lower frequencies (F0-F15) and was demonstrated as having superior denoising performance. Noise components were successfully isolated, while speech signal integrity was maintained. Clear separation between noise and speech elements was observed, suggesting effective processing of fundamental speech frequencies-Range Frequencies (F15-F31) achieved moderate denoising success. Speech patterns were partially preserved, but noise-speech separation was less definitive. This indicates challenges in balancing noise reduction with speech preservation, resulting in compromised performance compared to lower frequencies. Higher Frequencies (F31-F63) exhibited the poorest denoising performance. Uniform, high-amplitude patterns were displayed, significantly deviating from both input and target signals. The model struggled to distinguish between noise and speech, suggesting fundamental difficulties in processing higher frequency components. The model's varying performance across frequency bands indicates frequency-dependent denoising capabilities. Lower frequencies being better denoised aligns with typical speech processing characteristics. Higher frequency challenges might require specialized processing approaches. The results suggest potential for frequency-specific optimization strategies.

Acknowledging these limitations provides important context for interpreting the results and offers valuable direction for future research endeavours, where addressing these constraints could potentially lead to significant improvements in the performance and practical applicability of SNNs for speech denoising tasks.

6. Conclusions

Building upon the framework of the Intel N-DNS challenge, a three-stage system was developed. Initially, audio signals underwent pre-processing to prepare them for subsequent analysis. The Short-Time Fourier Transform (STFT) was then applied to extract time-frequency features, capturing the essence of the audio signal in a representation suitable for SNN processing. Finally, a carefully designed spiking neural network was employed to process these features and effectively suppress noise. Three speech denoising approaches were evaluated. Approach 1 (direct reconstruction) showed limited noise reduction (SNR -22.63 dB) despite consistent performance (training MSE: 0.7905, validation MSE: 0.7863, test MSE: 0.7881). Approach 2 exhibited significantly lower noise reduction (SNR -47.55 dB, PSNR -23.87). In contrast, Approach 3 (noise component prediction) achieved lower MSE (training MSE: 0.7439, validation MSE: 0.7413, test MSE: 0.7461), improved SNR (-22.39 dB) and PSNR (1.22), and a 25% error reduction. This approach also exhibited efficient learning with a rapid initial decline in training MSE. The results highlight the crucial role of input-output relationship and stable training in achieving effective speech denoising.

Future Work

While this project has demonstrated promising results, several avenues for future research and improvement can be explored. Investigating alternative SNN topologies and learning rules could potentially enhance denoising performance and efficiency. Also, having a bassline solution to speech denoising such as the Intel N-DNS, would give the opportunity for comparison on reversed raw waveforms as a product of inverse STFT. Additionally, fine-tuning the STFT window size and overlap could optimize feature extraction for the specific denoising task. Exploring the feasibility of implementing the proposed algorithm on neuromorphic hardware platforms would enable improved energy efficiency and real-time processing capabilities. Furthermore, combining the SNN-based denoising approach with other signal processing techniques, such as adaptive filtering or wavelet transforms, could further enhance denoising performance. Also, extending the evaluation to a wider range of noise types and levels would assess the robustness and generalizability of the proposed approach. Further improvements could be made by considering the inherent characteristics of speech. Speech has different important frequency bands, with some being more critical for intelligibility. A component-wise Mean Squared Error (MSE) calculation would allow for more targeted optimization, focusing on the most crucial frequency regions. Future research could explore modifications to the neural architecture to better exploit this frequency-specific behavior. This could involve incorporating frequency-specific layers and implementing frequency-dependent normalization. By calculating MSE for each frequency band individually, the model can be trained to prioritize denoising in the most critical frequency regions. This approach would provide a more nuanced evaluation and potentially lead to improved speech quality and intelligibility compared to a general global MSE approach. Additionally, investigating alternative spiking encoding schemes, such as temporal coding, population coding and phase coding, could potentially improve the information representation and processing capabilities of the SNN. By systematically exploring different encoding schemes and architectural modifications, we can strive to further enhance the performance of SNNs for speech denoising and other audio processing tasks. Furthermore, the denoising performance could be enhanced by exploring a wider range of thresholding methods. While this work utilized percentile-based thresholding, alternative approaches such as adaptive thresholding (Otsu's method, local adaptive thresholding), statistical methods (median-based, standard deviation-based), signal-specific methods (energy-based, frequency-dependent), wavelet-based methods (VisuShrink, SURE), and time-frequency domain methods (spectral subtraction, Gabor domain) could provide more sophisticated and robust threshold selection mechanisms.

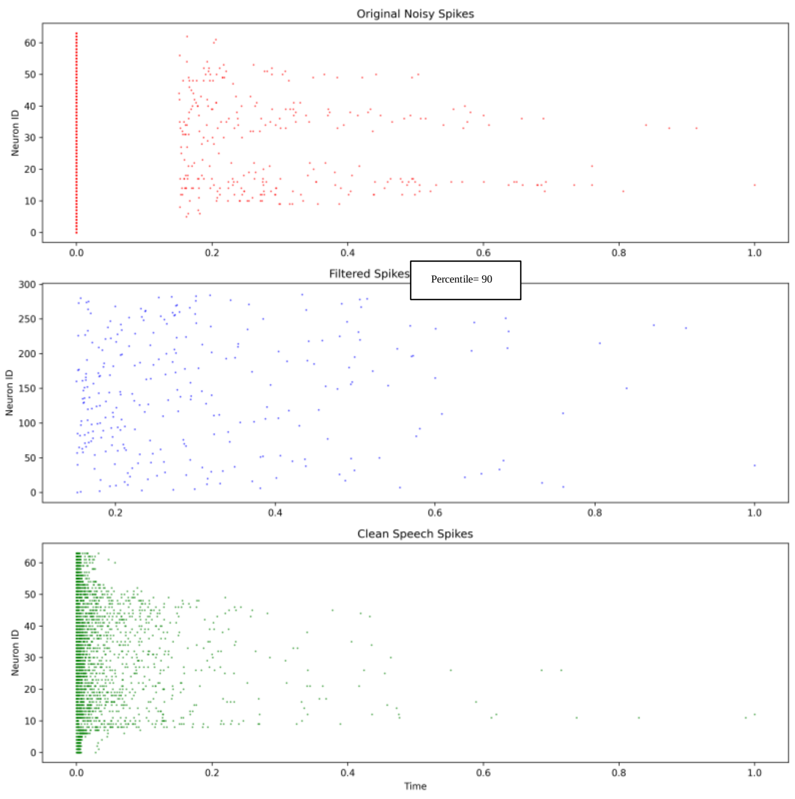
References

1. Lim, J. S., & Oppenheim, A. V. (1979). Enhancement and bandwidth compression of noisy speech. *Proceedings of the IEEE*, *67*(12), 1586-1604.
2. Li, B., & Sim, K. C. (2014, May). An ideal hidden-activation mask for deep neural networks based noise-robust speech recognition. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 200-204). IEEE.
3. Boll, S. (1979). Suppression of acoustic noise in speech using spectral subtraction. *IEEE Transactions on acoustics, speech, and signal processing*, *27*(2), 113-120.
4. H. Dubey et al., "ICASSP 2022 deep noise suppression challenge," in Proc. IEEE Int. Conf. Acoust. Speech Signal Process., 2022, pp. 9271– 9275.
5. Houwei Cao, David G Cooper, Michael K Keutmann, Ruben C Gur, Ani Nenkova, and Ragini Verma, "CREMA-D: Crowdsourced emotional multimodal actors dataset," IEEE Trans. on Affective Computing, vol. 5, no. 4, pp. 377–390, 2014.
6. Reddy, C. K., Dubey, H., Gopal, V., Cutler, R., Braun, S., Gamper, H., ... & Srinivasan, S. (2021, June). ICASSP 2021 deep noise suppression challenge. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 6623-6627). IEEE.
7. J. F. Gemmeke, D. P. W. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter, "Audio set: An ontology and human-labeled dataset for audio events," in IEEE ICASSP, 2017.
8. J. Thiemann, N. Ito, and E. Vincent, "The diverse environments multi-channel acoustic noise database (demand): A database of multichannel environmental noise recordings," The Journal of the Acoustical Society of America, p. 3591, 05 2013.
9. Shojaei, E., Ashayeri, H., Jafari, Z., Dast, M. R. Z., & Kamali, K. (2016). Effect of signal to noise ratio on the speech perception ability of older adults. *Medical journal of the Islamic Republic of Iran*, *30*, 342.
10. Turner, J. P., Knight, J. C., Subramanian, A., & Nowotny, T. (2022). mlGeNN: accelerating SNN inference using GPU-enabled neural networks. *Neuromorphic Computing and Engineering*, *2*(2), 024002.
11. Goyal, D., & Pabla, B. S. (2015). Condition based maintenance of machine tools—A S
12. Ma, G., Yan, R., & Tang, H. (2023). Exploiting noise as a resource for computation and learning in spiking neural networks. *Patterns*, *4*(10).

13. Xing, Y., Ke, W., Di Caterina, G., & Soraghan, J. (2019). Noise reduction using neural lateral inhibition for speech enhancement. *International Journal of Machine Learning and Computing*.
14. Wunderlich, T. C., & Pehle, C. (2021). Event-based backpropagation can compute exact gradients for spiking neural networks. *Scientific Reports*, *11*(1), 12829.
15. Guerrero Vázquez, E., Quintana Velázquez, F. M., & Guerrero Lebrero, M. D. L. P. (2023). Event-based Regression with Spiking Networks.
16. Hao, X., Ma, C., Yang, Q., Tan, K. C., & Wu, J. (2024, June). When audio denoising meets spiking neural network. In *2024 IEEE Conference on Artificial Intelligence (CAI)* (pp. 1524-1527). IEEE.
17. Fang, W., Yu, Z., Chen, Y., Masquelier, T., Huang, T., & Tian, Y. (2021). Incorporating learnable membrane time constant to enhance learning of spiking neural networks. In *Proceedings of the IEEE/CVF international conference on computer vision* (pp. 2661-2671).
18. Schrauwen, B., & Van Campenhout, J. (2003, July). BSA, a fast and accurate spike train encoding scheme. In *Proceedings of the International Joint Conference on Neural Networks, 2003.* (Vol. 4, pp. 2825-2830). IEEE.
19. Kim, Y., & Panda, P. (2021). Visual explanations from spiking neural networks using inter-spike intervals. *Scientific reports*, *11*(1), 19037.
20. Tal, D., & Schwartz, E. L. (1997). Computing with the leaky integrate-and-fire neuron: logarithmic computation and multiplication. *Neural computation*, *9*(2), 305-318.
21. Brehm, V., Austefjord, J. W., Lepadatu, S., & Qaiumzadeh, A. (2023). A proposal for leaky integrate-and-fire neurons by domain walls in antiferromagnetic insulators. *Scientific Reports*, *13*(1), 13404.

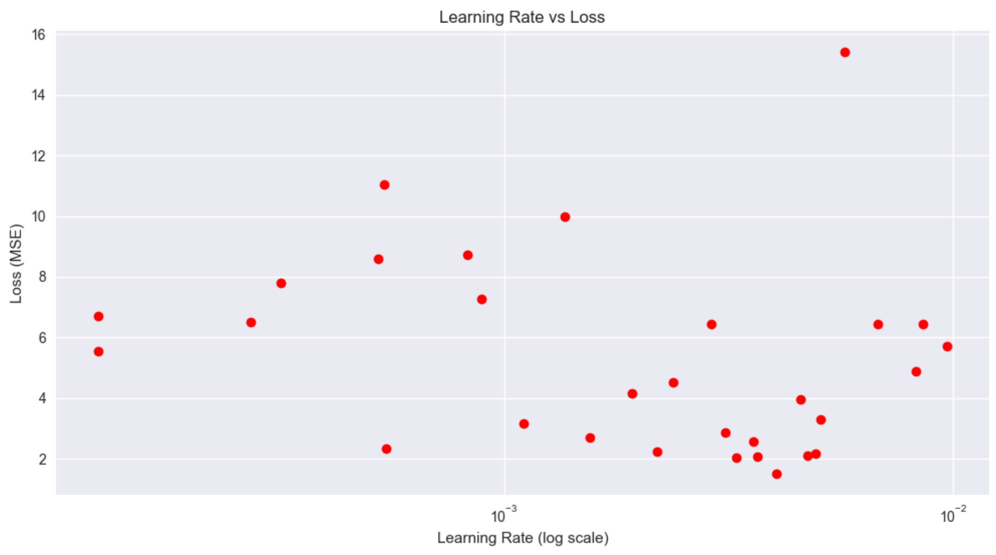
Appendix

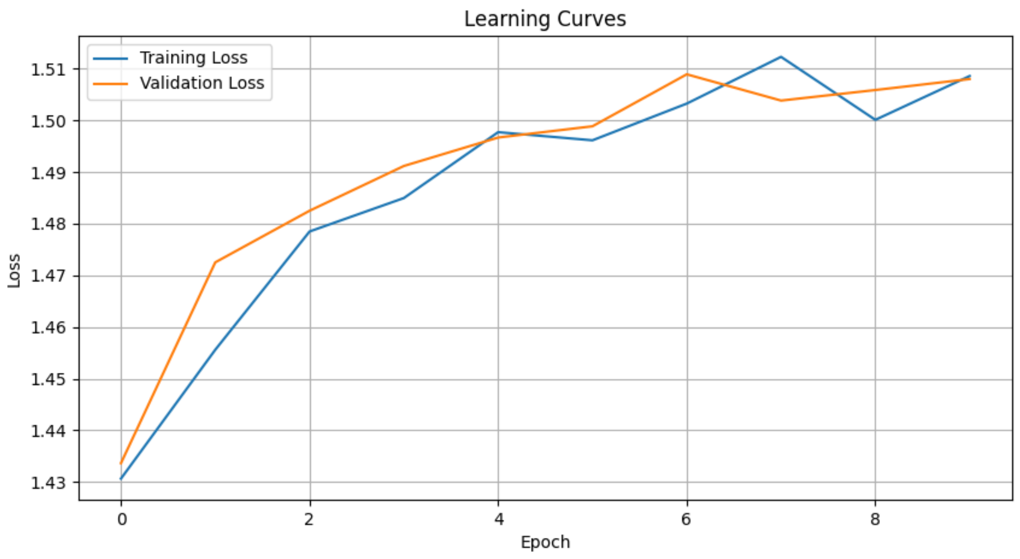
Supplementary materials

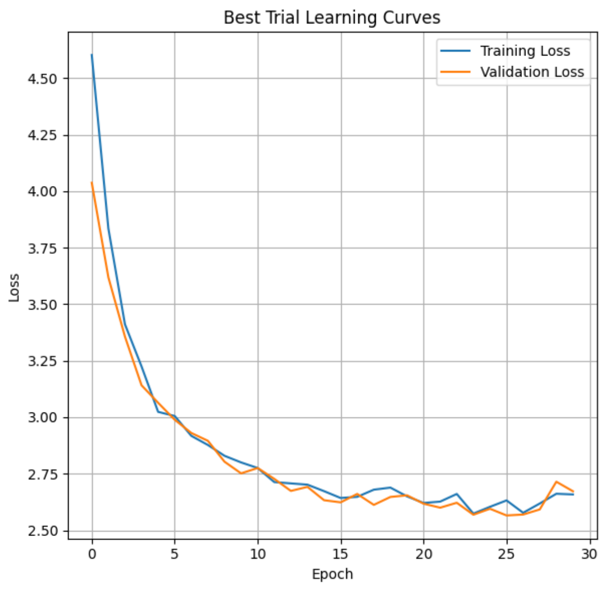


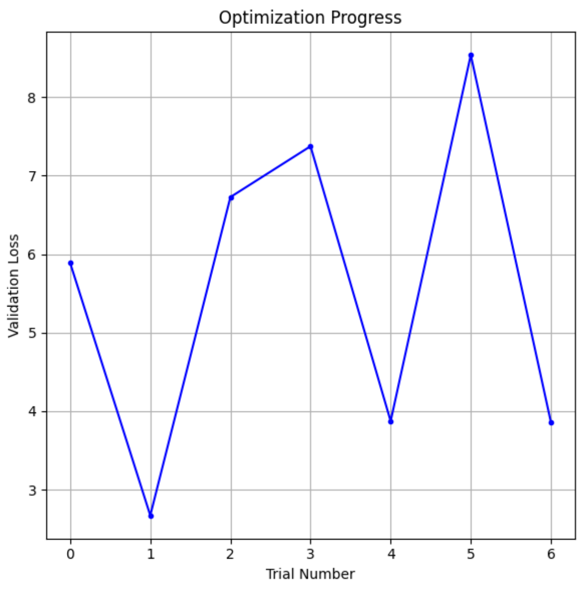
S1. The top panel displays the original noisy spikes, represented by red dots. The middle panel shows the filtered spikes, represented by blue dots, obtained after applying a percentile-based thresholding (= 90) filter to the noisy input. This filtering process aims to remove less significant spikes. The bottom panel presents the spikes derived from the clean speech signal, represented by green dots, serving as a reference for comparison.

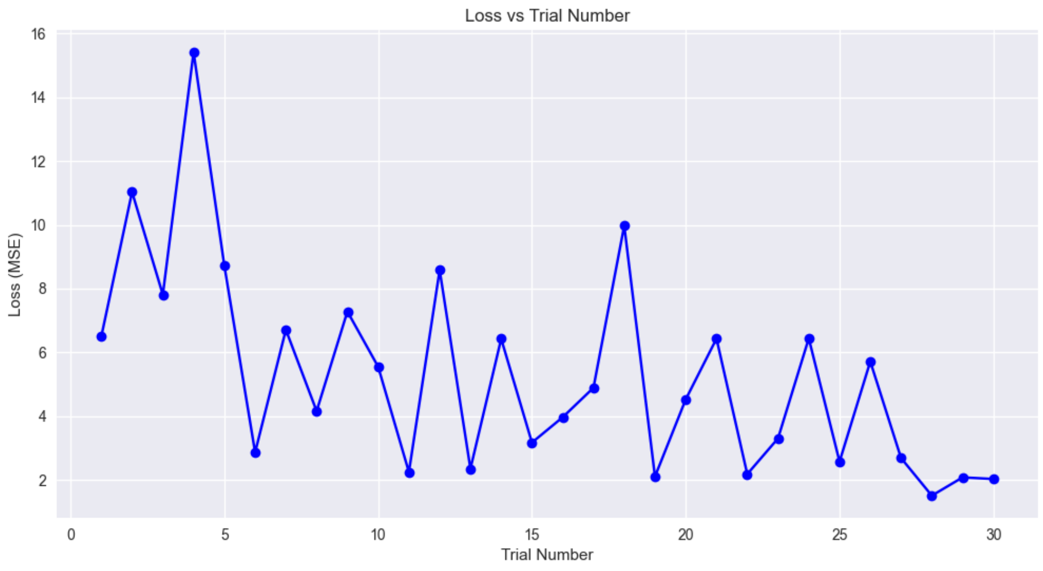
S2. All the other figures below are product of hyperparameter optimization.

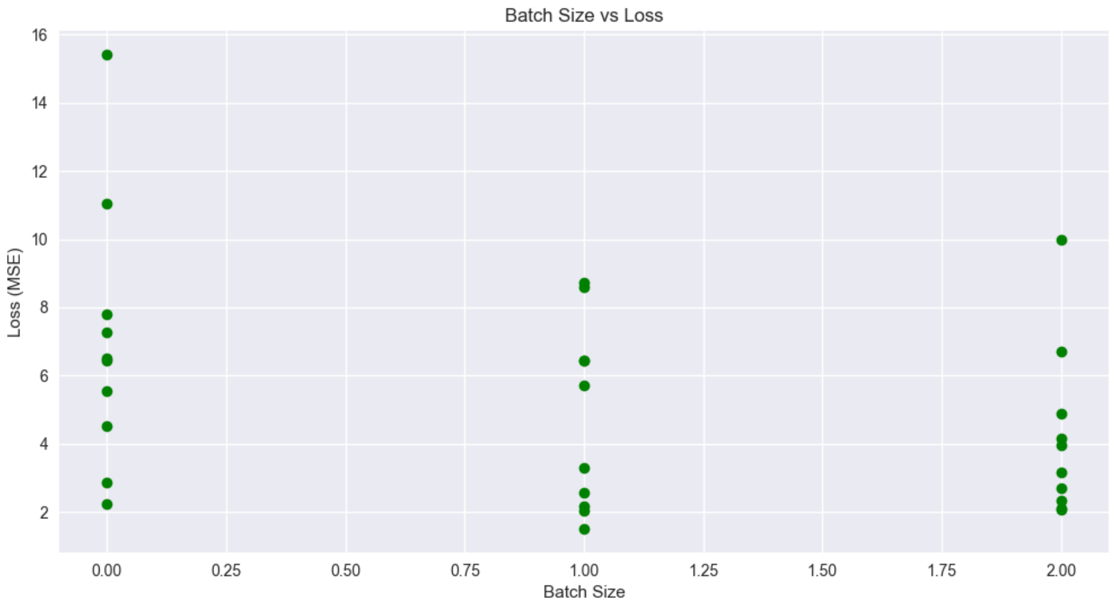


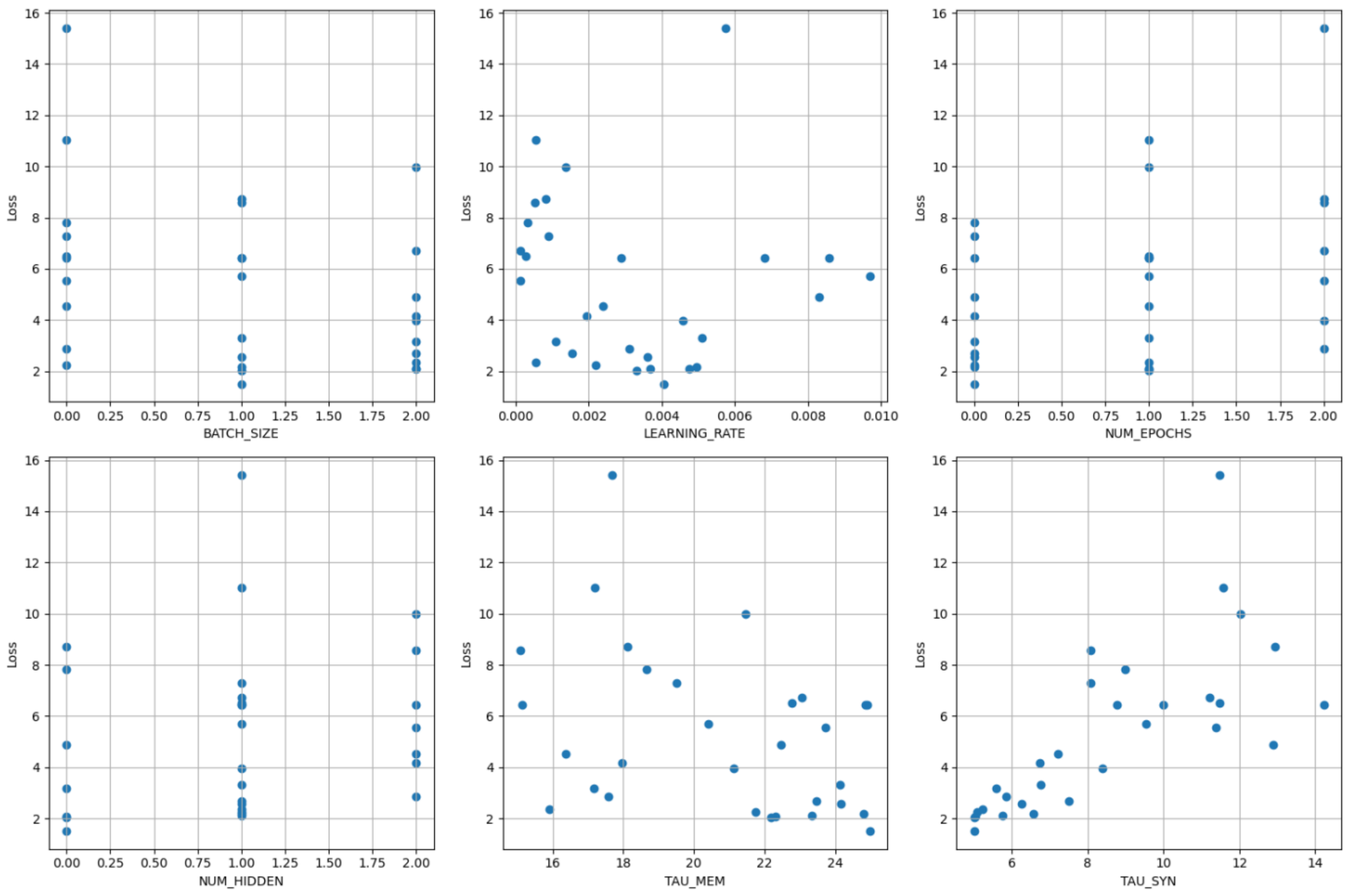

Learning Curves
Training Loss
Validation Loss
Loss
Epoch


Best Trial Learning Curves
Training Loss
Validation Loss
Loss
Epoch


Optimization Progress
Validation Loss
Trial Number


Loss vs Trial Number
Loss (MSE)
Trial Number
16
14
12
10
8
6
4
2
0
5
10
15
20
25
30


Batch Size vs Loss
Loss (MSE)
Batch Size
16
14
12
10
8
6
4
2
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Loss
BATCH_SIZE
LEARNING_RATE
NUM_EPOCHS
NUM_HIDDEN
TAU_MEM
TAU_SYN

Source Code

For Approach 1:

```
# Constants
NUM_INPUT = 64  # Number of Mel bands
NUM_HIDDEN = 128
NUM_OUTPUT = 64  # Should match NUM_INPUT since we're trying to reconstruct
clean spectrograms
NUM_EPOCHS = 50
TIME_STEPS = 45  # Match with example_timesteps - IT IS THE NUM OF TIME
FRAMES

def calculate_speech_quality_metrics(denoised, clean):
    """Calculate multiple metrics for speech quality assessment with error
handling"""
    metrics = {}

    # 1. Signal-to-Noise Ratio (SNR)
    def calculate_snr(denoised, clean):
        signal_power = np.mean(clean ** 2)
        noise = denoised - clean
        noise_power = np.mean(noise ** 2)

        # Avoid division by zero
        if noise_power < 1e-10:  # Small threshold
            return 0.0
        return 10 * np.log10(signal_power / noise_power)

    # 2. Peak Signal-to-Noise Ratio (PSNR)
    def calculate_psnr(denoised, clean):
        mse = np.mean((denoised - clean) ** 2)
        if mse < 1e-10:  # Avoid division by zero
            return 100.0  # Return high value for perfect match
        max_signal = np.max(clean)
        if max_signal == 0:  # Avoid log of zero
            return 0.0
        return 20 * np.log10(max_signal / np.sqrt(mse))

    # 3. Mean Square Error (MSE)
    def calculate_mse(denoised, clean):
        return np.mean((denoised - clean) ** 2)

    # 4. Segmental SNR (SSNR)
    def calculate_segmental_snr(denoised, clean, segment_size=45):
        num_segments = len(clean) // segment_size
        ssnr_values = []

        for i in range(num_segments):
            start = i * segment_size
            end = start + segment_size
            segment_clean = clean[start:end]
            segment_denoised = denoised[start:end]

            signal_power = np.mean(segment_clean ** 2)
            noise_power = np.mean((segment_denoised - segment_clean) ** 2)
            # Only append valid values
            if noise_power > 1e-10 and signal_power > 1e-10:
                ssnr = 10 * np.log10(signal_power / noise_power)
                ssnr_values.append(ssnr)

        # Return average only if we have values
        if len(ssnr_values) > 0:
            return np.mean(ssnr_values)
        return 0.0  # Default value if no valid segments

    # Calculate all metrics
    metrics['SNR'] = calculate_snr(denoised, clean)
    metrics['PSNR'] = calculate_psnr(denoised, clean)
    metrics['MSE'] = calculate_mse(denoised, clean)
    metrics['SSNR'] = calculate_segmental_snr(denoised, clean)

    return metrics

def normalize_mel_spectrogram(mel_spectrogram):
    """Normalize mel spectrogram to [0,1] range"""
    min_val = np.min(mel_spectrogram)
    max_val = np.max(mel_spectrogram)
    return (mel_spectrogram - min_val) / (max_val - min_val + 1e-10)

def compute_mel(directory, max_files=5060):
    """Compute Mel spectrogram for all .wav files in the given directory
using PyTorch"""
    mel_results = {}

    # Get the list of .wav files in the directory
    wav_files = [file for file in os.listdir(directory) if
file.endswith('.wav')]

    # Wrap the file processing loop with tqdm, limiting the total to
max_files if provided
    for idx, file in enumerate(tqdm(wav_files[:max_files], desc="Computing
Mel Spectrograms")):
        file_path = os.path.join(directory, file)
        waveform, sample_rate = torchaudio.load(file_path)

        # Compute STFT
        stft_tensor = torchaudio.functional.spectrogram(
            waveform.squeeze(0),  # Remove channel dimension
            n_fft=1024,
            hop_length=512,
            pad=0,
            window=torch.hann_window(1024),  # Use Hann window
            win_length=1024,
            power=1.0,
            normalized=False
        )

        magnitude_spectrum = torch.abs(stft_tensor)

        # Create a Mel filter bank
        mel_filter = torchaudio.transforms.MelScale(
            n_mels=NUM_INPUT,  # Number of Mel bands
            sample_rate=sample_rate,
            f_min=0.0,
            f_max=None,  # Default is sample_rate / 2
            n_stft=magnitude_spectrum.shape[0]  # Number of STFT bins
        )

        # Apply the Mel filter bank to the magnitude spectrum
        mel_spectrogram = mel_filter(magnitude_spectrum)

        # Normalize the mel spectrogram
        mel_spectrogram_np = mel_spectrogram.numpy()
        normalized_mel = normalize_mel_spectrogram(mel_spectrogram_np)
        mel_results[file] = torch.from_numpy(normalized_mel)

    return mel_results

def analyze_mel_spectrograms(noisy_dir, clean_dir, noise_dir,
max_files=None):
    """
    Analyze mel spectrograms from both directories
    Args:
        noisy_dir: Directory with noisy audio files
        clean_dir: Directory with clean audio files
        max_files: Number of files to analyze
    """
    # Compute mel spectrograms
    print("Computing clean speech mel spectrograms...")
    clean_mels = compute_mel(clean_dir, max_files=max_files)

    print("\nComputing noisy speech mel spectrograms...")
    noisy_mels = compute_mel(noisy_dir, max_files=max_files)

    print("Computing noise mel spectrograms...")
    noise_mels = compute_mel(noise_dir, max_files=max_files)

    # Analyze values
    all_noisy_values = []
    all_clean_values = []
    all_noise_values = []

    for file in clean_mels.keys():
        clean_mel = clean_mels[file].numpy()
        clean_mean_val= np.mean(clean_mel)
        all_clean_values.append(clean_mean_val)
        print(f"File: {file}")
        print(f"Mean: {clean_mean_val:.4f}")
        print(f"Shape: {clean_mel.shape}")
        print(f"Min: {np.min(clean_mel):.4f}")
        print(f"Max: {np.max(clean_mel):.4f}")




    for file in noisy_mels.keys():
        noisy_mel = noisy_mels[file].numpy()
        noisy_mean_val= np.mean(noisy_mel)
        all_noisy_values.append(noisy_mean_val)
        print(f"File: {file}")
        print(f"Mean: {noisy_mean_val:.4f}")
        print(f"Shape: {noisy_mel.shape}")
        print(f"Min: {np.min(noisy_mel):.4f}")
        print(f"Max: {np.max(noisy_mel):.4f}")


    for file in noise_mels.keys():
        noise_mel = noise_mels[file].numpy()
        noise_mean_val= np.mean(noise_mel)
        all_noise_values.append(noise_mean_val)
        print(f"File: {file}")
        print(f"Mean: {noise_mean_val:.4f}")
        print(f"Shape: {noise_mel.shape}")
        print(f"Min: {np.min(noise_mel):.4f}")
        print(f"Max: {np.max(noise_mel):.4f}")


    print("\nOverall Statistics of Clean Mel:")
    print(f"Average mean across files: {np.mean(all_clean_values):.4f}")
    print(f"Std of means: {np.std(all_clean_values):.4f}")
    print(f"Min mean: {np.min(all_clean_values):.4f}")
    print(f"Max mean: {np.max(all_clean_values):.4f}")

    print("\nOverall Statistics of Noisy Mel:")
    print(f"Average mean across files: {np.mean(all_noisy_values):.4f}")
    print(f"Std of means: {np.std(all_noisy_values):.4f}")
    print(f"Min mean: {np.min(all_noisy_values):.4f}")
    print(f"Max mean: {np.max(all_noisy_values):.4f}")

    print("\nOverall Statistics of Noise Mel:")
    print(f"Average mean across files: {np.mean(all_noise_values):.4f}")
    print(f"Std of means: {np.std(all_noise_values):.4f}")
    print(f"Min mean: {np.min(all_noise_values):.4f}")
    print(f"Max mean: {np.max(all_noise_values):.4f}")




    # Plot distributions
    plt.figure(figsize=(15, 5))


    plt.subplot(131)
    plt.hist(all_clean_values, bins=50, alpha=0.7, label='Clean')
    plt.axvline(np.mean(all_clean_values), color='r', linestyle='--',
                label=f'Mean={np.mean(all_clean_values):.4f}')
    plt.axvline(np.mean(all_clean_values) + np.std(all_clean_values),
color='k', linestyle=':', linewidth=1)
    plt.axvline(np.mean(all_clean_values) - np.std(all_clean_values),
color='k', linestyle=':', linewidth=1)
    plt.title('All Clean Mel Distributions')
    plt.xlabel('Mel Values')
    plt.ylabel('Count')
    plt.legend()

    # plot the distribution of the noise mel
    plt.subplot(132)
    plt.hist(all_noise_values, bins=50, alpha=0.7, label='Noise')
    plt.axvline(np.mean(all_noise_values), color='r', linestyle='--',
                label=f'Mean={np.mean(all_noise_values):.4f}')
    plt.axvline(np.mean(all_noise_values) + np.std(all_noise_values),
color='k', linestyle=':', linewidth=1)
    plt.axvline(np.mean(all_noise_values) - np.std(all_noise_values),
color='k', linestyle=':', linewidth=1)
    plt.title('All Noise Mel Distributions')
    plt.xlabel('Mel Values')
    plt.ylabel('Count')
    plt.legend()
    plt.subplot(133)
    plt.hist(all_noisy_values, bins=50, alpha=0.7, label='Noisy')
    plt.axvline(np.mean(all_noisy_values), color='r', linestyle='--',
                label=f'Mean={np.mean(all_noisy_values):.4f}')
    plt.axvline(np.mean(all_noisy_values) + np.std(all_noisy_values),
color='k', linestyle=':', linewidth=1)
    plt.axvline(np.mean(all_noisy_values) - np.std(all_noisy_values),
color='k', linestyle=':', linewidth=1)
    plt.title('All Noisy Mel Distributions')
    plt.xlabel('Mel Values')
    plt.ylabel('Count')
    plt.legend()


    plt.tight_layout()
    plt.savefig('mel_distributions.png', dpi=300, bbox_inches='tight')
    plt.show()


def apply_percentile_threshold(mel_spectrogram, percentile=90):
    """Apply percentile-based thresholding to mel spectrogram change the
percentile to 40 for more aggressive thresholding"""
    threshold = np.percentile(mel_spectrogram, percentile)
    return np.where(mel_spectrogram > threshold, mel_spectrogram, 0)

def visualize_thresholds(noisy_mel, clean_mel, percentiles=[20, 30, 35, 80,
85, 90, 95]):
    """Visualize different percentile thresholds"""
    num_thresholds = len(percentiles)
    fig, axes = plt.subplots(num_thresholds + 2, 1, figsize=(12,
4*(num_thresholds + 2)))

    # Plot original clean spectrogram
    im0 = axes[0].imshow(clean_mel, aspect='auto', origin='lower',
cmap='viridis')
    axes[0].set_title('Clean Speech')
    plt.colorbar(im0, ax=axes[0])

    # Plot original noisy spectrogram
    im1 = axes[1].imshow(noisy_mel, aspect='auto', origin='lower',
cmap='viridis')
    axes[1].set_title('Noisy Speech (Original)')
    plt.colorbar(im1, ax=axes[1])

    # Plot different thresholds
    for idx, p in enumerate(percentiles):
        thresholded = apply_percentile_threshold(noisy_mel, percentile=p)
        im = axes[idx+2].imshow(thresholded, aspect='auto', origin='lower',
cmap='viridis')
        axes[idx+2].set_title(f'Threshold at {p}th percentile')
        plt.colorbar(im, ax=axes[idx+2])

    plt.tight_layout()
    plt.savefig('threshold_comparison.png', dpi=300, bbox_inches='tight')
    plt.show()

def filter_spikes(mel_spectrogram, percentile=90):
    """
    Filter spikes based on:
    1. Percentile threshold of mel spectrogram values
    2. Return only significant spikes
    """
    # First apply percentile threshold to mel spectrogram
    threshold = np.percentile(mel_spectrogram, percentile)
    thresholded_mel = np.where(mel_spectrogram > threshold,
mel_spectrogram, 0)

    # Get indices of non-zero values (these will generate spikes)
    spike_indices = np.nonzero(thresholded_mel)
    spike_values = thresholded_mel[spike_indices]

    # Create spike times and IDs only for significant values
    times = spike_values.flatten()

    # # Create IDs in the same way as noisy/clean plots (0-63 range)
    # ids = np.arange(thresholded_mel.shape[1])  # Creates IDs from 0 to 63
    # ids = np.tile(ids, thresholded_mel.shape[0])  # Repeats the pattern

    #  # Create IDs using np.repeat (0-63 range)
    ids = spike_indices[1]  % 64

    # Preprocess these filtered spikes
    return preprocess_spikes(times, ids, thresholded_mel.shape[1])

def visualize_spike_filtering(noisy_mel, clean_mel, percentile=35):
    """Visualize spike filtering effect on noisy vs clean signals"""
    fig, axes = plt.subplots(3, 1, figsize=(12, 12))

    # Original noisy spikes
    flattened_noisy = noisy_mel.reshape(-1)
    orig_ids = np.repeat(np.arange(noisy_mel.shape[1]), noisy_mel.shape[0])
    noisy_spikes = preprocess_spikes(flattened_noisy, orig_ids,
noisy_mel.shape[1])

    # Filtered noisy spikes
    filtered_spikes = filter_spikes(noisy_mel, percentile)

    # Clean spikes for comparison
    flattened_clean = clean_mel.reshape(-1)
    clean_spikes = preprocess_spikes(flattened_clean, orig_ids,
clean_mel.shape[1])

    # Plot original noisy spikes
    axes[0].scatter(noisy_spikes.spike_times,
orig_ids[:len(noisy_spikes.spike_times)],
                   alpha=0.5, s=1, c='red', label='Noisy')
    axes[0].set_title('Original Noisy Spikes')
    axes[0].set_ylabel('Neuron ID')

    # Plot filtered spikes
    axes[1].scatter(filtered_spikes.spike_times,
                   np.arange(len(filtered_spikes.spike_times)),
                   alpha=0.5, s=1, c='blue', label='Filtered')
    axes[1].set_title(f'Filtered Spikes (percentile={percentile})')
    axes[1].set_ylabel('Neuron ID')

    # Plot clean spikes
    axes[2].scatter(clean_spikes.spike_times,
orig_ids[:len(clean_spikes.spike_times)],
                   alpha=0.5, s=1, c='green', label='Clean')
    axes[2].set_title('Clean Speech Spikes')
    axes[2].set_ylabel('Neuron ID')
    axes[2].set_xlabel('Time')

    plt.tight_layout()
    plt.savefig('spike_filtering_comparison.png', dpi=300,
bbox_inches='tight')
    plt.show()
```

```
# Add before line 229 (before the epoch loop)
def validate_network(network, val_inputs, val_outputs, batch_size=16):
    """Run validation on the entire validation set"""
    total_val_loss = 0
    num_batches = 0

    for i in range(0, len(val_inputs), batch_size):
        batch_end = min(i + batch_size, len(val_inputs))

        # Process validation batch
        val_batch_inputs = val_inputs[i:batch_end]
        flattened_val_inputs = val_batch_inputs.reshape(-1)
        val_input_ids = np.repeat(np.arange(NUM_INPUT), TIME_STEPS *
len(val_batch_inputs))
        val_batch_spikes = preprocess_spikes(flattened_val_inputs,
val_input_ids, NUM_INPUT)

        val_batch_outputs = val_outputs[i:i+1]

        # Get network predictions (without training)
        metrics, cb_data = network.train(
            {input_population: [val_batch_spikes]},
            {output_population: val_batch_outputs},
            num_epochs=1,
            callbacks=callbacks
        )

        # Calculate validation loss
        val_output = cb_data["output_v"][-1]
        val_error = val_output - val_batch_outputs.reshape(-1, NUM_OUTPUT)
        batch_loss = np.mean(val_error * val_error)

        total_val_loss += batch_loss
        num_batches += 1

    return total_val_loss / num_batches

def evaluate_on_test_set(network, test_inputs, test_outputs,
batch_size=16):
    """Evaluate model on test set and return metrics"""
    test_losses = []
    denoised_outputs = []

    for i in range(0, len(test_inputs), batch_size):
        batch_end = min(i + batch_size, len(test_inputs))

        # Process test batch
        test_batch_inputs = test_inputs[i:batch_end]
        flattened_test_inputs = test_batch_inputs.reshape(-1)
        test_input_ids = np.repeat(np.arange(NUM_INPUT), TIME_STEPS *
len(test_batch_inputs))
        test_batch_spikes = preprocess_spikes(flattened_test_inputs,
test_input_ids, NUM_INPUT)

        test_batch_outputs = test_outputs[i:i+1]

        # Get predictions
        metrics, cb_data = network.train(
            {input_population: [test_batch_spikes]},
            {output_population: test_batch_outputs},
            num_epochs=1,
            callbacks=callbacks
        )

        # Store outputs for audio comparison
        denoised_output = cb_data["output_v"][-1]
        denoised_outputs.append(denoised_output)

        # Calculate test loss
        error = denoised_output - test_batch_outputs.reshape(-1,
NUM_OUTPUT)
        test_loss = np.mean(error ** 2)
        test_losses.append(test_loss)

    return np.mean(test_losses), denoised_outputs




if __name__ == "__main__":
    # Directories
    speech_dir =
"/its/home/msu22/attempt_DISS/STAGE1/NEW_PART/preprocessed_speech"
    noise_dir =
"/its/home/msu22/attempt_DISS/STAGE1/NEW_PART/preprocessed_noise"
    output_dir = "/its/home/msu22/attempt_DISS/STAGE1/NEW_PART/comb_speech"

    # Set the maximum number of files to process
    max_files_to_process = 5060  # Change this value as needed- to match
the lengths of input and output files (no of  )

    # Compute normalized mel spectrograms
    comb_mel_results = compute_mel(output_dir,
max_files=max_files_to_process)
    clean_mel_results = compute_mel(speech_dir,
max_files=max_files_to_process)

    # Prepare inputs and outputs
    comb_inputs = []
    clean_outputs = []

    for file in clean_mel_results.keys():
        # Clean outputs are already normalized from compute_mel

        thresholded_input_clean =
apply_percentile_threshold(clean_mel_results[file].numpy(), percentile=90)
        clean_outputs.append(thresholded_input_clean)
    for file in comb_mel_results.keys():
        # Apply thresholding to normalized input
        thresholded_input =
apply_percentile_threshold(comb_mel_results[file].numpy(), percentile=90)
        comb_inputs.append(thresholded_input)

    # Convert to numpy arrays
    comb_inputs = np.array(comb_inputs)
    clean_outputs = np.array(clean_outputs)

     # Now you can use comb_inputs and clean_outputs for further processing
or training
    print(f"Original input shape: {comb_inputs.shape}, Output shape:
{clean_outputs.shape}")







    inputs = comb_inputs.transpose(0, 2, 1)  # From (1000, 64, 45) to
(1000, 45, 64)
    outputs = clean_outputs.transpose(0, 2, 1)  # From (1000, 64, 45) to
(1000, 45, 64)

    print(f"Reshaped input shape: {inputs.shape}, Output shape:
{outputs.shape}")


        # After line 42 (after the transpose operations)
    from sklearn.model_selection import train_test_split

    # First split: separate test set (80% train+val, 20% test)
    train_val_inputs, test_inputs, train_val_outputs, test_outputs =
train_test_split(
        inputs, outputs, test_size=0.2, random_state=42
    )

    # Second split: separate train and validation
    train_inputs, val_inputs, train_outputs, val_outputs =
train_test_split(
        train_val_inputs, train_val_outputs, test_size=0.1, random_state=42
    )

    print("\nData split sizes:")
    print(f"Training samples: {len(train_inputs)}")      # ~70% of total
    print(f"Validation samples: {len(val_inputs)}")      # ~10% of total
    print(f"Test samples: {len(test_inputs)}")          # ~20% of total



    TAU_SYN = 5.016


     # Use it before training
    # sample_noisy = inputs[0].reshape(TIME_STEPS, NUM_INPUT)
    # sample_clean = outputs[0].reshape(TIME_STEPS, NUM_INPUT)
    analyze_mel_spectrograms(output_dir, speech_dir, noise_dir)
    sample_noisy = comb_inputs[0].reshape(TIME_STEPS, NUM_INPUT)
    sample_clean = clean_outputs[0].reshape(TIME_STEPS, NUM_INPUT)
    visualize_thresholds(sample_noisy, sample_clean)

    sample_noisy = comb_inputs[0].reshape(TIME_STEPS, NUM_INPUT)
    sample_clean = clean_outputs[0].reshape(TIME_STEPS, NUM_INPUT)


    visualize_spike_filtering(sample_noisy, sample_clean)


    MAX_SPIKES= TIME_STEPS * NUM_INPUT * 5060
    # Prepare the network
    network = Network(default_params)
    with network:
        # Populations
        input_population = Population(SpikeInput(max_spikes=MAX_SPIKES),
NUM_INPUT, record_spikes=True)
        hidden_population = Population(LeakyIntegrateFire(v_thresh=
1.05138668, tau_mem=24.988, tau_refrac= None), NUM_HIDDEN,
record_spikes=True)
        output_population = Population(LeakyIntegrate(tau_mem=24.988,
readout="var"), NUM_OUTPUT, record_spikes=True)

        # Connections
        Connection(input_population, hidden_population, Dense(Normal(sd=2.5
/ np.sqrt(NUM_INPUT))), Exponential(TAU_SYN))
        Connection(hidden_population, hidden_population,
Dense(Normal(sd=1.5 / np.sqrt(NUM_HIDDEN))), Exponential(TAU_SYN))
        Connection(hidden_population, output_population,
Dense(Normal(sd=2.0 / np.sqrt(NUM_HIDDEN))), Exponential(TAU_SYN))

    # Compile the network
    compiler = EventPropCompiler(
        example_timesteps=TIME_STEPS,
        losses="mean_square_error",
        optimiser=Adam(0.004041),  # Reduced learning rate
        reg_lambda_upper=1e-8,
        reg_lambda_lower=1e-8,
        reg_nu_upper=10,
        max_spikes=MAX_SPIKES
    )
    compiled_net = compiler.compile(network)



    with compiled_net:
        def alpha_schedule(epoch, alpha):
            if (epoch % 2) == 0 and epoch != 0:
                return alpha * 0.7
            else:
                return alpha



        # Define callbacks without progress bar
        callbacks = [
            VarRecorder(output_population, "v", key="output_v"),
            SpikeRecorder(input_population, key="input_spikes"),
            SpikeRecorder(hidden_population, key="hidden_spikes"),
            OptimiserParamSchedule("alpha", alpha_schedule)
        ]  # Removed "batch_progress_bar"



        # # Initialize storage for MSE and best model performance
        # mse_per_epoch = []
        # best_mse = float('inf')
        # patience = 10  # Number of epochs to wait for improvement
        # patience_counter = 0





        # Create better visualization setup using seaborn
        import seaborn as sns
        sns.set_theme(style="darkgrid")
        sns.set_context("notebook", font_scale=1.2)

        # Create figure for meaningful visualizations
        fig, axes = plt.subplots(2, 2, figsize=(15, 12))

        # Plot 1: MSE Learning Curve
        ax_mse = axes[0, 0]
        ax_mse.set_title('Training Progress')
        ax_mse.set_xlabel('Epoch')
        ax_mse.set_ylabel('MSE')



        # Plot 3: Network Activity
        ax_spikes = axes[1, 0]
        ax_spikes.set_title('Neural Activity')
        ax_spikes.set_xlabel('Time')
        ax_spikes.set_ylabel('Neuron Index')


        # Plot 2: Spectrogram Comparison
        ax_spectro = axes[0, 1]
        ax_spectro.set_title('Waveform  Comparison')
        ax_spectro.set_xlabel('Time Steps')
        ax_spectro.set_ylabel('Amplitude')

        # Plot 4: Denoising Performance
        ax_denoise = axes[1, 1]
        ax_denoise.set_title('Denoising Error')
        ax_denoise.set_xlabel('Time Steps')
        ax_denoise.set_ylabel('Error Amplitude')


        # Setup progress tracking
        mse_per_epoch = []
        train_losses = []
        val_losses = []

        BATCH_SIZE = 16
        # ADD THE START TIME
        start_time = perf_counter()

        # Calculate correct number of batches from input size
        total_samples = len(train_inputs)  # Get actual number of samples
from input data
```

```
        print(f"Total number of samples: {total_samples}")  # Debug print

        num_batches_per_epoch = (total_samples + BATCH_SIZE - 1) //
BATCH_SIZE  # Ceiling division
        total_batches = NUM_EPOCHS * num_batches_per_epoch

        print(f"Batches per epoch: {num_batches_per_epoch}")  # Debug print
        print(f"Total batches across all epochs: {total_batches}")  # Debug
print

        # Create progress bars with proper positioning
        main_pbar = tqdm(total=total_batches, desc='Overall Progress',
position=0)
        epoch_pbar = tqdm(range(NUM_EPOCHS), desc="Epochs", position=1,
leave=True)

        for epoch in epoch_pbar:
            epoch_mse = 0
            num_batches = 0

            # Process all samples in each epoch
            for i in range(0, total_samples, BATCH_SIZE):
                batch_end = min(i + BATCH_SIZE, total_samples)

                # Debug prints for batch sizes
                print(f"Batch indices: {i}:{batch_end}")  # Debug print

                # Process batch inputs
                batch_inputs = train_inputs[i:batch_end]
                print(f"Batch inputs shape: {batch_inputs.shape}")  # Debug
print

                flattened_inputs = batch_inputs.reshape(-1)
                input_ids = np.repeat(np.arange(NUM_INPUT), TIME_STEPS *
len(batch_inputs))
                batch_spikes = preprocess_spikes(flattened_inputs,
input_ids, NUM_INPUT)

                # # Fix the input_ids to ensure they stay within 0-63
                # input_ids = np.arange(NUM_INPUT)  # [0, 1, 2, ..., 63]
                # input_ids = np.tile(input_ids, TIME_STEPS *
len(batch_inputs))  # Repeat pattern properly
                # # Now when we call the module's preprocess_spikes, it
should work correctly
                # batch_spikes = preprocess_spikes(flattened_inputs,
input_ids, NUM_INPUT)


                # Get corresponding clean target
                batch_outputs = train_outputs[i:i+1]  # Shape: (1, 45, 64)
                print(f"Batch outputs shape: {batch_outputs.shape}")  #
Debug print

                # Train on batch
                metrics, cb_data = compiled_net.train(
                    {input_population: [batch_spikes]},
                    {output_population: batch_outputs},
                    num_epochs=1,
                    callbacks=callbacks
                )
                # Calculate loss
                output_values = cb_data["output_v"][-1]
                error = output_values - batch_outputs.reshape(-1,
NUM_OUTPUT)
                batch_loss = np.mean(error * error)
                epoch_mse += batch_loss
                num_batches += 1

                # Update main progress bar
                main_pbar.update(1)
                main_pbar.set_postfix({
                    'Batch': f'{num_batches}/{num_batches_per_epoch}',
                    'MSE': f'{batch_loss:.6f}'
                })

            # Calculate epoch metrics
            avg_epoch_mse = epoch_mse / num_batches
            mse_per_epoch.append(avg_epoch_mse)
            train_losses.append(avg_epoch_mse)

            # Validation step
            val_loss = validate_network(compiled_net, val_inputs,
val_outputs)
            val_losses.append(val_loss)

            # Update epoch progress
            epoch_pbar.set_postfix({
                'Train MSE': f'{avg_epoch_mse:.4f}',
                'Val MSE': f'{val_loss:.4f}'
            })

        # Close progress bars
        main_pbar.close()
        epoch_pbar.close()


        # Print final training summary
        total_time = perf_counter() - start_time    # Plot training
progress
        print(f"\nTraining completed in {total_time/60:.2f} minutes")
        print(f"Final Train MSE: {train_losses[-1]:.4f}")
        print(f"Final Val MSE: {val_losses[-1]:.4f}")

        plt.figure(figsize=(10, 6))
        plt.plot(train_losses, 'b-', label='Training MSE', marker='o')
        plt.plot(val_losses, 'r-', label='Validation MSE', marker='o')
        plt.title('Final Training and Validation MSE History')
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.legend()
        plt.grid(True)
        plt.yscale('log')  # Use log scale if losses vary widely
        plt.savefig('final_training_history.png', dpi=300,
bbox_inches='tight')
        plt.close()


                # Evaluate on test set
        print("\nEvaluating on test set...")
        test_mse, denoised_outputs = evaluate_on_test_set(compiled_net,
test_inputs, test_outputs)
```

```
        print(f"Test Set MSE: {test_mse:.4f}")

        # Convert denoised_outputs to numpy array if it isn't already
        denoised_outputs = np.array(denoised_outputs)

        print("\nCalculating speech quality metrics...")
        all_test_metrics = []

        # Make sure we only process as many samples as we have
        num_samples = min(len(test_inputs), len(denoised_outputs))
        print(f"Processing {num_samples} test samples")

        # Calculate metrics for each test sample
        for i in range(num_samples):
            try:
                denoised = denoised_outputs[i].reshape(TIME_STEPS,
NUM_INPUT)
                clean = test_outputs[i].reshape(TIME_STEPS, NUM_INPUT)
                metrics = calculate_speech_quality_metrics(denoised, clean)
                all_test_metrics.append(metrics)
            except Exception as e:
                print(f"Error processing sample {i}: {str(e)}")
                print(f"Denoised shape: {denoised_outputs[i].shape if
hasattr(denoised_outputs[i], 'shape') else 'unknown'}")
                print(f"Clean shape: {test_outputs[i].shape}")
                continue

        # Print shapes for debugging
        print(f"\nShapes:")
        print(f"test_inputs shape: {test_inputs.shape}")
        print(f"test_outputs shape: {test_outputs.shape}")
        print(f"denoised_outputs shape: {denoised_outputs.shape if
hasattr(denoised_outputs, 'shape') else len(denoised_outputs)}")

        # Calculate average metrics only if we have results
        if all_test_metrics:
            avg_metrics = {
                metric: np.mean([m[metric] for m in all_test_metrics])
                for metric in all_test_metrics[0].keys()
            }

            print("\nAverage Speech Quality Metrics across Test Set:")
            for metric, value in avg_metrics.items():
                print(f"{metric}: {value:.2f}")
        else:
            print("No metrics were calculated successfully")

        # Plot distribution of metrics
        plt.figure(figsize=(15, 5))
        for idx, metric in enumerate(['SNR', 'PSNR', 'SSNR']):
            plt.subplot(1, 3, idx+1)
            values = [m[metric] for m in all_test_metrics]
            plt.hist(values, bins=20)
            plt.title(f'{metric} Distribution')
            plt.xlabel('Value (dB)')
            plt.ylabel('Count')
        plt.tight_layout()
        plt.savefig('speech_quality_metrics_distribution.png', dpi=300,
bbox_inches='tight')
        plt.close()
        # Plot metrics over time for a single example
        plt.figure(figsize=(15, 5))
        example_idx = 0  # First test sample
        denoised = denoised_outputs[example_idx].reshape(TIME_STEPS,
NUM_INPUT)
        clean = test_outputs[example_idx].reshape(TIME_STEPS, NUM_INPUT)

        # Calculate metrics for each time step
        time_metrics = []
        for t in range(TIME_STEPS):
            metrics = calculate_speech_quality_metrics(
                denoised[t:t+1],
                clean[t:t+1]
            )
            time_metrics.append(metrics)

        # Plot metrics over time
        for metric in ['SNR', 'PSNR', 'SSNR']:
            values = [m[metric] for m in time_metrics]
            plt.plot(values, label=metric)
        plt.title('Speech Quality Metrics Over Time')
        plt.xlabel('Time Step')
        plt.ylabel('Value (dB)')
        plt.legend()
        plt.grid(True)
        plt.savefig('speech_quality_metrics_over_time.png', dpi=300,
bbox_inches='tight')
        plt.close()


        # Modified final plot to include test results
        plt.figure(figsize=(10, 6))
        plt.plot(train_losses, 'b-', label='Training MSE', marker='o')
        plt.plot(val_losses, 'r-', label='Validation MSE', marker='o')
        plt.axhline(y=test_mse, color='g', linestyle='--', label='Test
MSE')
        plt.title('Training, Validation, and Test MSE History')
        plt.xlabel('Epoch')
        plt.ylabel('MSE')
        plt.legend()
        plt.grid(True)
        plt.yscale('log')
        plt.savefig('final_training_history_with_test.png', dpi=300,
bbox_inches='tight')
        plt.close()




        # Update visualizations
        # 1. MSE Learning Curve
        ax_mse.clear()
        ax_mse.plot(mse_per_epoch, 'b-', label='Training MSE')
        ax_mse.set_title('Training Progress')
        ax_mse.set_xlabel('Epoch')
        ax_mse.set_ylabel('MSE')
        ax_mse.grid(True)
        ax_mse.legend()


        time = np.arange(TIME_STEPS)
        # 3. Error Analysis (Bottom Left)

        error = denoised - clean
        error_mean = np.mean(error, axis=1)  # Average across frequency
bands
        ax_spikes.plot(time, error_mean, 'r-', label='Error', alpha=0.7)
        ax_spikes.fill_between(time, error_mean, 0, alpha=0.3, color='red')
        ax_spikes.set_title('Denoising Error')
        ax_spikes.set_xlabel('Time Steps')
        ax_spikes.set_ylabel('Error Amplitude')
        ax_spikes.grid(True, alpha=0.3)

        denoised= cb_data["output_v"][-1].reshape(TIME_STEPS, NUM_INPUT)
        # denoised = output_values

        # Plot frequency profiles
        # 1. Waveform Comparison (Top Left)
        sample_idx = 0  # Select first sample (or any other index you want
to visualize)
        noisy_sample = train_inputs[sample_idx].reshape(TIME_STEPS,
NUM_INPUT)  # Shape: (45, 64)
        clean_sample = train_outputs[sample_idx].reshape(TIME_STEPS,
NUM_INPUT)  # Shape: (45, 64)

        # Now plot with correct dimensions
        ax_spectro.plot(time, np.mean(noisy_sample, axis=1), 'r-',
label='Noisy', alpha=0.6, linewidth=1)
        ax_spectro.plot(time, np.mean(clean_sample, axis=1), 'g-',
label='Clean', alpha=0.6, linewidth=1)
        ax_spectro.plot(time, np.mean(denoised, axis=1), 'b-',
label='Denoised', alpha=0.6, linewidth=1)
        ax_spectro.set_title('Waveform Comparison')
        ax_spectro.set_xlabel('Time Steps')
        ax_spectro.set_ylabel('Average Amplitude')
        ax_spectro.legend()
        ax_spectro.grid(True, alpha=0.3)



        # 3. Neural Activity
        ax_spikes.clear()
        input_spikes = cb_data["input_spikes"]
        hidden_spikes = cb_data["hidden_spikes"]
        ax_spikes.scatter(input_spikes[0][-1], input_spikes[1][-1],
                        c='blue', s=1, alpha=0.5, label='Input')
        ax_spikes.scatter(hidden_spikes[0][-1], hidden_spikes[1][-1],
                        c='red', s=1, alpha=0.5, label='Hidden')
        ax_spikes.legend()
        ax_spikes.set_title('Neural Activity')



        ax_denoise.clear()
        denoised_spec = denoised.T
        im = ax_denoise.imshow(denoised_spec,
                            aspect='auto',
                            origin='lower',
                            cmap='viridis')

        # Add colorbar
        plt.colorbar(im, ax=ax_denoise, label='Amplitude')

        # Enhance the plot
        ax_denoise.set_title('Denoising Performance (All Frequencies)')
        ax_denoise.set_xlabel('Time Steps')
        ax_denoise.set_ylabel('Frequency Bands')

        # Add frequency band markers
        freq_ticks = np.linspace(0, NUM_INPUT-1, 5)
        freq_labels = [f'F{int(f)}' for f in freq_ticks]
        ax_denoise.set_yticks(freq_ticks)
        ax_denoise.set_yticklabels(freq_labels)

        # Add grid
        ax_denoise.grid(True, alpha=0.3)


        # ax_denoise.clear()
        # ax_denoise.plot(denoised_spec, aspect='auto', origin='lower',
cmap='viridis')
        # ax_denoise.set_title('Denoised Output')
        # ax_denoise.set_xlabel('Time Steps')
        # ax_denoise.set_ylabel('Frequency Bands')
        # ax_denoise.colorbar(im_error, ax=ax_denoise)
        # ax_denoise.grid(True, alpha=0.3)




        # ax_denoise.clear()
        # band_idx = NUM_INPUT // 2  # Middle frequency band
        # ax_denoise.plot(time, noisy_sample[:, band_idx], 'r-', alpha=0.5,
label='Noisy')
        # ax_denoise.plot(time, clean_sample[:, band_idx], 'g--',
alpha=0.5, label='Clean')
        # ax_denoise.plot(time, denoised[:, band_idx], 'b-', alpha=0.5,
label='Denoised')
        # ax_denoise.legend()
        # ax_denoise.set_title('Denoising Performance (Mid Frequency
Band)')

        # Update the figure
        plt.tight_layout()
        plt.pause(0.01)  # Small pause to update the display


        end_time = perf_counter()
        print(f"Training time: {end_time - start_time:.2f}s")


        # # Early stopping check
        # if epoch_mse < best_mse:
        #     best_mse = epoch_mse
        #     patience_counter = 0
        #     # Here you could save the best model if needed
        # else:
        #     patience_counter += 1
        #     if patience_counter >= patience:
        #         print(f"\nEarly stopping triggered after {epoch + 1}
epochs")
        #         break
        # Print progress with more details
        # print(f"Epoch {epoch + 1}/{NUM_EPOCHS}")
        # print(f"MSE: {epoch_mse:.4f}")
        # print(f"Best MSE: {best_mse:.4f}")
        # print(f"Patience Counter: {patience_counter}/{patience}")

        # # Update progress bar with cleaner postfix
        # pbar.set_postfix_str(
        #     f"MSE: {epoch_mse:.4f}, Best: {best_mse:.4f}, Patience:
{patience_counter}/{patience}"
        # )

    # Final save
    fig.savefig('FINAL_training_visualization.png', dpi=300,
bbox_inches='tight')
    plt.close()

    # Save MSE history separately
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(mse_per_epoch) + 1), mse_per_epoch, 'b-',
marker='o')
    plt.title('Training MSE History')
    plt.xlabel('Epoch')
    plt.ylabel('MSE')
    plt.grid(True)
    plt.savefig('mse_history.png', dpi=300, bbox_inches='tight')
    plt.close()



            # Replace your final plotting code (around line 565)
    # Final plot at the end of training


        # 4. Clear and Comparative Denoising Performance
    ax_denoise.clear()

    # Create a figure with three subplots side by side
    fig_denoise = plt.figure(figsize=(15, 5))
    gs = GridSpec(1, 3, figure=fig_denoise, wspace=0.3)

    # 1. Noisy Input
    ax1 = fig_denoise.add_subplot(gs[0])
    im1 = ax1.imshow(noisy_sample.T,
                    aspect='auto',
                    origin='lower',
                    cmap='viridis')
    ax1.set_title('Noisy Input', fontsize=12, pad=10)
    ax1.set_xlabel('Time Steps')
    ax1.set_ylabel('Frequency Bands')
    plt.colorbar(im1, ax=ax1, label='Amplitude')

    # 2. Clean Target
    ax2 = fig_denoise.add_subplot(gs[1])
    im2 = ax2.imshow(clean_sample.T,
                    aspect='auto',
                    origin='lower',
                    cmap='viridis')
    ax2.set_title('Clean Target', fontsize=12, pad=10)
    ax2.set_xlabel('Time Steps')
    ax2.set_ylabel('Frequency Bands')
    plt.colorbar(im2, ax=ax2, label='Amplitude')

    # 3. Network Output (Denoised)
    ax3 = fig_denoise.add_subplot(gs[2])
    im3 = ax3.imshow(denoised.T,
                    aspect='auto',
                    origin='lower',
                    cmap='viridis')
    ax3.set_title('Network Output (Denoised)', fontsize=12, pad=10)
    ax3.set_xlabel('Time Steps')
    ax3.set_ylabel('Frequency Bands')
    plt.colorbar(im3, ax=ax3, label='Amplitude')

    # Add frequency labels to all plots
    freq_ticks = np.linspace(0, NUM_INPUT-1, 5, dtype=int)
    freq_labels = [f'F{f}' for f in freq_ticks]
    for ax in [ax1, ax2, ax3]:
        ax.set_yticks(freq_ticks)
        ax.set_yticklabels(freq_labels)

        # Add time ticks
        time_ticks = np.linspace(0, TIME_STEPS-1, 5, dtype=int)
        ax.set_xticks(time_ticks)
        ax.set_xticklabels(time_ticks)

    # Add a main title
    fig_denoise.suptitle('Speech Denoising Performance Comparison',
                        fontsize=14, y=1.05)

    # Save the figure
    plt.savefig(f'denoising_comparison_epoch.png',
                dpi=300, bbox_inches='tight')
    plt.close(fig_denoise)

    # Update the main visualization's denoising subplot with error analysis
    ax_denoise.clear()
    error = denoised.T - clean_sample.T
    im_error = ax_denoise.imshow(error,
                                aspect='auto',
                                origin='lower',
                                cmap='RdBu_r',  # Red-Blue diverging
colormap
                                vmin=-np.max(abs(error)),
                                vmax=np.max(abs(error)))
    ax_denoise.set_title('Denoising Error\n(Blue: Removed Noise, Red: Added
Noise)')
    ax_denoise.set_xlabel('Time Steps')
    ax_denoise.set_ylabel('Frequency Bands')
    plt.colorbar(im_error, ax=ax_denoise)






    # Print final losses
    print(f"\nFinal Training MSE: {train_losses[-1]:.4f}")
    print(f"Final Validation MSE: {val_losses[-1]:.4f}")
```