MS PROJECT REPORT
SPEECH ENHANCEMENT FOR ROBUST AUTOMATIC SPEECH RECOGNITION

Sreeganesh Valathara Rajendran
BU ID: U28069221

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
`www.bu.edu/ece`

Fall 2024

Summary

In this project, we present a audio-enhancement model to improve transcription accuracy in audio-inputs containing multiple speakers. We train a U-Net network with a custom composite loss-function in an adversarial training paradigm, to suppress lower-volume speakers and maintain higher-volume speakers when fed inputs containing overlapped speech. We present the improvements in the ASR systems with the addition of this model in the processing pipeline.

Contents

List of Figures

List of Tables

1 Introduction

Noise refers to any additional unwanted signal that interferes with the clarity, accuracy, and quality of a larger signal. It has plagued audio recordings since the inception of microphones and other audio recording devices. Noise is of many types: white noise, pink noise, babble noise, shot noise and so on.

In addition to being unpleasant to hear, it can also cause problems in digital systems. In telecommunication systems, noise can make conversations difficult to follow. In hearing aids, noise can interfere with the user's ability to focus on speech, thereby reducing the effectiveness of the device. In speech-based diagnostic tools, the presence of noise can result in false classifications.

In this project report, we focus on Automatic Speech Recognition systems. ASR systems take in audio inputs containing speech and output text corresponding to the transcription of the speech in the audio clip. Thus, this technology is an important stage in pipelines of larger systems like virtual assistants (Siri, Alexa, Google), real-time transcription services, language translation tools and so on.

The presence of noise in the input audio often results in the transcription produced by the ASR being faulty or error-filled. This is problematic since faulty transcription in the ASR stage will result in the error propagating through the rest of the stages. For example, faulty transcription will result in the virtual assistant implementing the wrong command. Thus, it is imperative that such pipelines actively suppress corruptive noise.

Thanks to the increasing R&D in this direction, modern-day ASRs are quite robust to noise. However, upon my investigation, I found that this robustness held primarily for noise with different statistical characteristics from that off speech, such as traffic noise. However, when the input speech was corrupted with noise that had similar statistical characteristics, there was an increase in transcription errors. Especially when there were multiple speakers in the audio clip. The transcription contained words from both speakers. This is rather problematic since overlapping speech is ubiquitous in daily life: cafes, office spaces, trains or any other public space, and having a system that fails in such scenarios would mean a system whose use is highly limited.

To overcome this challenge, we develop a method to suppress the background speakers to assist the ASR in accurately transcribing the words of the loudest speaker, which is assumed to be the one that is closest to the device.

2 Background

The three main pillars of this project are the architecture, loss function and the training paradigm.

2.1 Convolutional U-Net

A neural network, fundamentally, is a function approximator. The set of functions it can represent depends on the architecture of the neural network. So the choice of architecture should depend on things like: dimensionality of the function we wish to approximate ("target-function", moving forward), properties of the target-function and properties of the data. This is important as the function we want to approximate must be within the set of functions the neural network architecture can approximate.

Prior to the onset of transformer [11] based architectures, U-Nets [7] used to be state-of-the-art for audio tasks. This was primarily due to the fact that operating on the STFT-magnitude and recombining with the original phase, produced reliable outputs.

In addition to the track-record of U-Nets in the audio-space, we also consider U-Nets due to its reduced computational load compared to auto-regressive methods [9] and token-based[11] methods . While such methods occupy the current state-of-the-art approaches in audio, they come with the trade-off of increased computational load. Those methods are not appropriate for our tasks because our project is intended to be run on edge-devices, and in real-time. So we shall be using U-Net for this project.

The U-Net is a deep convolutional neural network designed for tasks involving image-like data. This is ideal for us because we'll be using the model to deal with STFTs which has an image-like dimensions. The model follows the U-Net framework, consisting of two main parts: a down-sampling (encoder) path and an up-sampling (decoder) path. The encoder progressively extracts hierarchical features from the input using convolutional layers with a stride of 2 to reduce spatial dimensions. Each encoder block comprises a convolutional layer, a batch normalization layer (except the first), and a LeakyReLU activation function. The network begins with an input channel of 1 and doubles the number of feature channels at each subsequent layer, starting from 16 up to 512 in the deepest layer. These skip connections are important for proper gradient-flow. It also enables the model to decide the correct hierarchical level features that is ideal for the task.

The decoder path mirrors the encoder, utilizing transposed convolutional layers to up-sample the feature maps and reconstruct the original spatial dimensions. Each decoder block consists of a transposed convolutional layer, batch normalization, ReLU activation, and dropout for regularization in earlier layers. To preserve spatial details and ensure accurate reconstruction, the decoder concatenates the corresponding feature maps from the encoder via skip connections at each stage. This concatenation doubles the input channels of the decoder layers, which is handled in the up-sampling process. The final output passes through a 'Sigmoid' activation function, which maps it to the range [0, 1]. This design decision is to made since we want the model to create a multiplicative mask, which will attenuate different frequencies at different levels. The architecture we've used for our U-Net is given in Table 1

Table 1: U-Net Architecture

Stage	Layer	Input Channels	Output Channels	Details
Down-Sampling Path (Encoder)				
Layer 1	Conv2d + LeakyReLU	1	16	5×5, Stride: 2, Pad: 2
Layer 2	Conv2d + BatchNorm2d + LeakyReLU	16	32	5×5, Stride: 2, Pad: 2
Layer 3	Conv2d + BatchNorm2d + LeakyReLU	32	64	5×5, Stride: 2, Pad: 2
Layer 4	Conv2d + BatchNorm2d + LeakyReLU	64	128	5×5, Stride: 2, Pad: 2
Layer 5	Conv2d + BatchNorm2d + LeakyReLU	128	256	5×5, Stride: 2, Pad: 2
Layer 6	Conv2d + BatchNorm2d + LeakyReLU	256	512	5×5, Stride: 2, Pad: 2
Up-Sampling Path (Decoder)				
Layer 1	ConvTranspose2d + BatchNorm2d + ReLU + Dropout	512	256	2×2, Stride: 2
Layer 2	ConvTranspose2d + BatchNorm2d + ReLU + Dropout	512 (256 + skip)	128	2×2, Stride: 2
Layer 3	ConvTranspose2d + BatchNorm2d + ReLU + Dropout	256 (128 + skip)	64	2×2, Stride: 2
Layer 4	ConvTranspose2d + BatchNorm2d + ReLU	128 (64 + skip)	32	2×2, Stride: 2
Layer 5	ConvTranspose2d + BatchNorm2d + ReLU	64 (32 + skip)	16	2×2, Stride: 2
Layer 6	ConvTranspose2d + BatchNorm2d + Sigmoid	32 (16 + skip)	1	2×2, Stride: 2

2.2 Adversarial Training

Adversarial training [3] is a training paradigm introduced by Goodfellow [3] that involves using discriminator networks to classify samples from a "true" distribution versus out-of-distribution samples. In our context of speech enhancement, the true distribution refers to the distribution of clean-speech signals containing only one speaker, while out-of-distribution refers to samples that are processed outputs of signals that have been altered by multiple speakers.

Adversarial training occurs in two phases: in the first phase, the discriminator is trained to distinguish clean speech (true distribution) from processed speech (out-of-distribution) by reducing its classification error. In the second phase, the generator, in this case a speech-enhancement model, is trained to produce speech samples that resemble clean speech, thus increasing the error of the discriminator and improving the enhancement model's ability to remove background noise.

The primary reason for this training paradigm is to ensure that certain features are present [2]. One of the issues with conventional losses such as $L2$ or $L1$ is that it prioritizes getting right those features that contribute the most to the loss. This means that the higher-frequency features or features that might be seminal but do not contribute as much are ignored because the lower-frequency features contribute more to the loss. This is problematic because these features might be anchor features and thus, more important to the ASR systems. One way to get around this is by using larger models. However, that is not an option since this model is built for edge devices. Thus, using composite loss-functions are more ideal. Thus, we bring in adversarial learning to the project. This technique has been proved to work through MelGAN[5] and Hi-Fi GAN [4].

2.3 Discriminator

The discriminator being used for adversarial training is a time-domain discriminator. It takes in the time-domain signals and classifies whether it was sampled from the "true" distribution or out-of-distribution. We use two instantiations of this discriminator. We use two discriminators operating on two scales of raw waveforms [4].

Layer	Type	In Channels	Out Channels	Kernel Size	Stride	Groups	Activation
1	Conv1d	1	16	15	1	1	LeakyReLU
2	Conv1d	16	64	41	4	4	LeakyReLU
3	Conv1d	64	256	41	4	16	LeakyReLU
4	Conv1d	256	1024	41	4	64	LeakyReLU
5	Conv1d	1024	1024	41	4	256	LeakyReLU
6	Conv1d	1024	1024	5	1	1	LeakyReLU
7	Conv1d	1024	1	3	1	1	-

Table 2: Discriminator Architecture

2.4 Whisper

The Whisper [6] model is a large-scale, weakly supervised speech recognition system designed to perform robustly across multiple languages and tasks without fine-tuning. It utilizes a Transformer-based [11] encoder-decoder architecture trained on 680,000 hours of diverse audio-transcript data sourced from the internet, including multilingual and translation datasets. Whisper eliminates the need for pre-processing steps like inverse text normalization by relying on sequence-to-sequence learning. The model represents tasks using special tokens, enabling transcription, translation, voice activity detection, and language identification within a unified framework. To enhance generalization, the dataset was filtered for quality and de-duplicated, ensuring diversity in audio sources while maintaining reliable transcripts.

Whisper offers six model sizes, trading off accuracy for speed. Table 3 lists the models and their approximate memory requirements and inference speeds relative to their largest model, *large*.

2.5 Loss Function Design

For training this model, we use a function of L1-losses modulated with cosine embeddings, to train the model.

- The primary loss-function is based on the $L1$ loss. For each FFT in the STFT, the loss is calculated in the following manner. The loss essentially pushes the

Size	**Parameters**	**Model**	**Required VRAM**	**Relative speed**
tiny	39 M	`tiny`	~1 GB	~10x
base	74 M	`base`	~1 GB	~7x
small	244 M	`small`	~2 GB	~4x
medium	769 M	`medium`	~5 GB	~2x
large	1550 M	`large`	~10 GB	1x
turbo	809 M	`turbo`	~6 GB	~8x

Table 3: Specifications of different Whisper sizes.

processed STFT further away from that of the input and pulls closer to that of the ideal STFT.

$$\begin{aligned}\text{loss}_1 = &|\text{STFT}_{\text{ideal}} - \text{STFT}_{\text{processed}}| \\ &- \min\Big(|\text{STFT}_{\text{processed}} - \text{STFT}_{\text{noisy}}|, |\text{STFT}_{\text{ideal}} - \text{STFT}_{\text{noisy}}|\Big)\end{aligned}$$

- The second loss-function is based on the observation that the amplitude scaling of the FFT-Magnitude only changes the amplification factor of the signal obtained by IFFT. This is an important observation because this means that if the output of our processing produces a scaled down version of the ideal-STFT, if we do not consider this information, the error will still be high. So in order to account for this, we bring in the cosine loss between the magnitudes. This is multiplied with the first loss-function .

$$\text{loss}_2 = \frac{\text{STFT}_{\text{processed}}}{||\text{STFT}_{\text{processed}}||} * \frac{\text{STFT}_{\text{ideal}}}{||\text{STFT}_{\text{ideal}}||}$$

- Finally, we put the two together to get our loss-function

$$\text{loss}_{\text{final}} = \frac{1}{M} \sum_{\text{for each FFT in STFT}} \text{loss}_1 * \text{loss}_2$$

2.6 TIMIT Database

We use the TIMIT database throughout this project. TIMIT Acoustic-Phonetic Continuous Speech Corpus was a joint effort among the Massachusetts Institute of Technology (MIT), SRI International (SRI) and Texas Instruments, Inc. (TI). It comprises of approximately five hours of English speech along with time-aligned transcriptions. TIMIT contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences [1].

The TIMIT corpus includes time-aligned orthographic, phonetic and word transcriptions as well as a single channel, 16-bit, 16kHz speech waveform file for each utterance.

Speaker metadata includes gender, dialect, birth date, height, race, and education level. Of the 630 speakers, about 70% are men and 30% are women[1]. The dataset provides a training-test split. The training set contains 4620 single-speaker samples and the test set contains 1680 single-speaker samples.

3 Method

3.1 Signal Processing Pipeline

The algorithm begins by computing the Short-Time Fourier Transform (STFT) of the input audio signal, transforming it into the time-frequency domain. From the resulting STFT, the magnitude and phase components are extracted. The magnitude is then passed through a U-Net architecture to generate a mask that captures the relevant features for enhancement or suppression. The mask is a tensor of the same dimensions as that of the STFT, with values ranging between 0 and 1. This mask is element-wise multiplied with the magnitude of the input STFT to produce a processed magnitude spectrum. The processed magnitude is then combined with the original phase component to reconstruct the processed STFT. Finally, the inverse STFT is applied to the processed STFT, converting it back to the time domain and producing the enhanced output audio signal. This process effectively leverages the U-Net to refine the magnitude spectrum while preserving the phase information.

The block diagram for the signal processing pipeline is given by Figure 1 and the algorithm by Algorithm 1.

Algorithm 1 Audio Processing Pipeline

```
1: Calculate the Short-Time Fourier Transform (STFT) of the input signal.
2: Compute the magnitude and phase of the STFT.
3: Feed the STFT magnitude to the U-Net to produce a mask.
4: Multiply the mask with the magnitude of the input STFT to obtain the processed
   magnitude.
5: Combine the processed magnitude with the phase of the input STFT to recon-
   struct the processed STFT.
6: Apply the inverse STFT to obtain the output signal.
```

3.2 Training

The model-training follows the adversarial training paradigm. Thus, the training is split into two phases. In the first phase, the U-net is trained to produce a mask that would accurately reproduce the ideal STFT-magnitude when multiplied with the STFT-magnitude of the input. And the discriminators are trained to classify audio samples that are clean vs those that are not clean (processed or noisy).

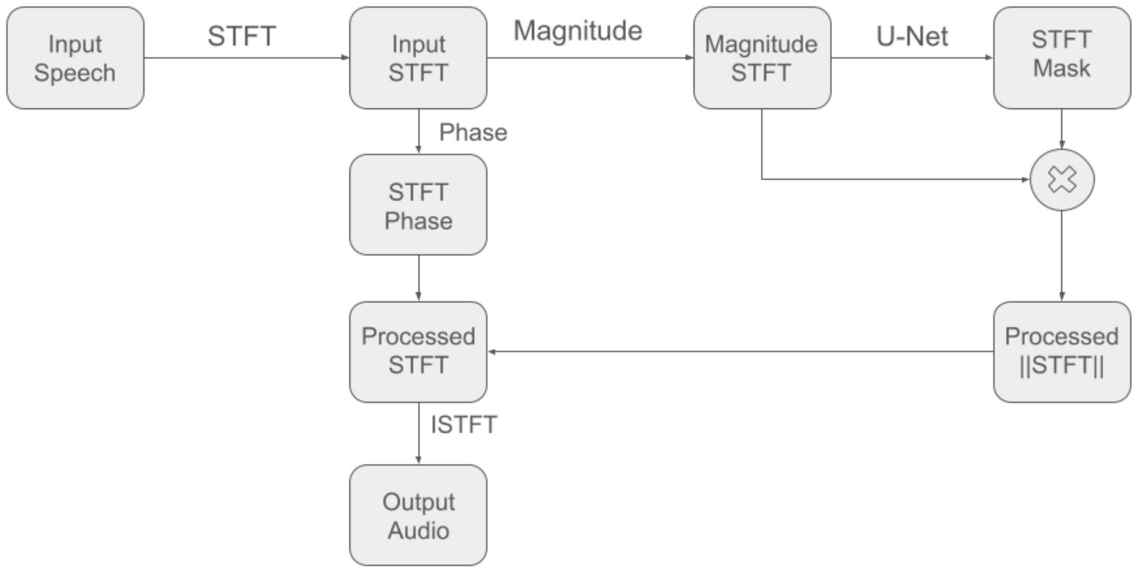


Figure 1: Signal Processing Pipeline

In the second phase, we train the pipeline (and therefore the U-Net) to maximize the mistake of the discriminators. By doing so, we force the U-Net to produce masks, which when multiplied with the STFT-magnitude of input, produce signals that follow the same distribution as that of the clean-speech. Note that the discriminators are not trained in this phase and they're usually put in the frozen mode using PyTorch's *.eval()* mode.

In our training, the two phases are switched every 400 iterations. Theoretically, this period is irrelevant as long as both the phases get equal time but generally, it is good practice to spend some iterations in one phase before moving to the other.

3.3 Inference

For inference, only the U-Net model is required. During inference, the models are usually put in *eval* mode to make sure the weights are not accidentally updated. To produce the output, we follow the same process as that shown in the signal processing pipeline section and the black diagram given in Figure 1.

4 User Guide

4.1 Dataset Setup

The dataset must be present locally. And the training and inference script will run as long as the absolute path to the training-directory and test-directory are explicitly provided in *fFetchListOfTrainTestFiles()*.

Since the inputs to the model are essentially speech-clips with two speakers, the number of possible two-speech combinations is given by $^{n}C_2$. This means that the training-set and the test-set possible for our context, are 21,339,780 and 2,820,720 samples, respectively. Creating and storing the whole dataset is not feasible as the train-set would take 4234 GB and test-set would take 559 GB.

To get around this limitation, we create a python function, *fCreateBatchTwoSpeech()*. This function does the following

- Sample pairs of wav-files from the list.
- Mix the pairs at argument SNR.
- Calculate and store the STFT of noisy-speech and primary-speech (the louder speech).
- Return two tensors containing the STFT of the noisy-speech and clean-speech.

4.1.1 Dataset path

If TIMIT [1] is the dataset to be used, make sure the paths to the training-directory and test-directory are copied appropriately to the body of the function, *fFetchListOfTrainTestFiles*.

However, if one wishes to use another dataset for the task, the script should handle that too. Please follow the following set of tasks to use the new dataset

- Split the dataset into training, validation and test-set. The standard rule of thumb in machine learning is to use 80%-10%-10% if the dataset is moderately sized. If the dataset is huge, then 96%-2%-2% might be more appropriate.
- Ensure that the training and test set are in different directories.
- Copy the absolute path of the training-set and validation-set (or test-set if you're putting together validation and test-set) to the appropriate variables in the body of the function, *fFetchListOfTrainTestFiles()*.

4.2 Conda Setup

Conda environments are isolated, self-contained spaces to manage software packages, dependencies, and configurations for Python and other languages. Each environment is an independent workspace, ensuring that specific libraries and their versions do not interfere with those in other environments. This isolation is the norm when working on projects with distinct requirements, allowing you to work on multiple projects without worrying about compatibility issues. Environments are created and managed using the "conda" command-line tool, which is part of the Anaconda or Miniconda distribution.

In order to ensure robust reproducibility of this project, we also provide the *yml* file of my conda environment. This file can be used to reproduce the same environment in another PC, irrespective of the platform. In addition to ensuring reproducibility, this also ensures setup taking less time allowing the researchers to spend more time adding to this project.

To reproduce the same conda environment in another system, please step through the following set of steps

- Install conda, if not already installed, at Conda: Getting Started
- Copy *environment.yml* file into the project directory (if not already copied). If not available, please download from environment.yml
- Run the following command in terminal, at the project directory: *conda env create -f environment.yml*. After this command, the conda environment must've installed.
- To enter the conda environment, enter the following in the command-line *conda activate attenuate_env_mlrobot*. To deactivate, enter, *conda deactivate*.
- All training and inference script must be run only after this environment has been activated. Else, there is a good chance of running into dependency issues.

4.3 Path Setup

4.3.1 Whisper Path

The ASR system we're using for this project is Whisper [6]. When we're instantiating the model, in addition to choosing the model, we also get to decide where the model must be saved. It is good practice to explicitly choose where the models are saved as this enables successul troubleshooting and project-moving.

So in the section of the code where we're instantiating the Whisper model, please ensure to provide the absolute path to where you wish the model to be saved, using the argument `download_root`.

4.3.2 Data Path

As the training progresses, we save a number of python objects. This enables us to "check in" and assess the trajectory of our training. This is especially useful when the code is being run on a headless server or through batchjobs. To enable this, we also need to provide the paths where these must be saved.

So to ensure this goes as expected, please edit the entries to the variables,

- `modelDirectoryPath`: Path to directory where trained models are saved.
- `directory_to_save_figures`: Path to directory where plots are saved.
- `directory_to_save_audio`: Path to directory where sample audios are saved.
- `directory_to_save_data`: Path to directory where pickle-data is saved.

Note: Due to the sheer compute required during training, it is a good idea that GPUs be used for this project. Thus, before beginning training, please ensure that the torch has access to the GPUs by running.

```
print("torch.cuda.is_available()=", torch.cuda.is_available())
```

If the output is `True`, then torch has access to the GPU. If not, then there is no CUDA GPU access. This usually calls for installing the CUDA Toolkit. Torch can also run with other GPUs. Please refer to appropriate documentation.

4.4 Inference

This script is to primarily take an existing input-audio, process it, save the output-audio and print the transcript for the input and the processed. Step through the following steps to run the script

1. Follow the steps outlined in the setup.
2. Run the following code from terminal: `python asr_enhancer_read_process_write.py`.

4.5 Training

In this script, we initialize the architecture with random weights and start training the network using the adversarial framework. Step through the following to train a new network

1. Please follow the steps outlined in the earlier parts of this section.
2. Run the code `python asr_enhancer_train.py`

4.6 Demo

This script is primarily for demonstration. It randomly creates a two-speech audio with the required SNR, processes it and saves the audio and produces transcripts. To run this script, step through the following

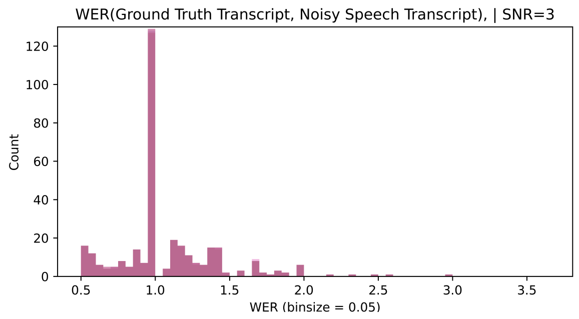
1. First, please follow the steps in the earlier parts of this section.
2. Place the trained models somewhere locally, ideally in the project directory. If not available, please download from here.
3. In the inference, code, assign the path to the variable, `modelMagnitude_path`.
4. Run the code from terminal using: `python asr_enhancer_demo.py`

5 Analysis & Conclusion

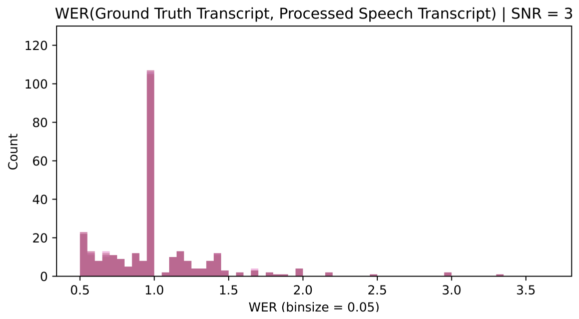
5.1 Outlier Reduction

Figures 2, 3, 4 shows the number of outliers before and after processing with our pipeline. An outlier is defined as those signals that have a word-error-rate greater than 50%. From the figures, the first observation is that the number of outliers have gone down by a significant number. And more importantly, we see that the outliers have moved closer to the lower WER.

This squeezing of outliers is important because most ASR involved pipelines today, incorporate NLUs. These make the system particularly robust because it tries to make sense of the text-output even if there are word replacements. This means that if our system reduces the error down to a WER that the NLU can handle, the system's robustness goes up.



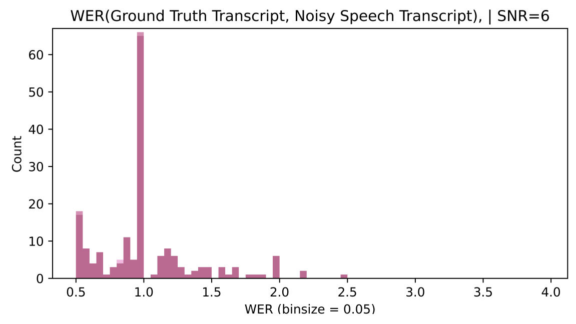
(a) Num Outliers with WER $\geq 50\% = 321$



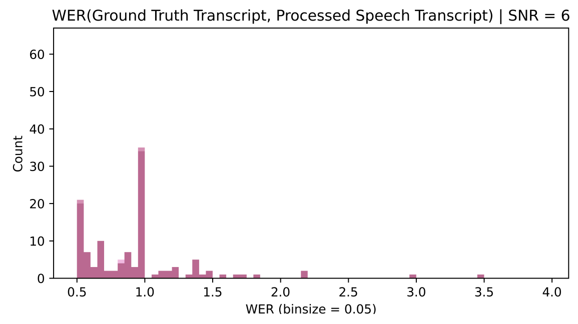
(b) Num Outliers with WER $\geq 50\% = 275$

Figure 2: Outliers for SNR = 3

Table 4 shows the reduction in outliers after processing with our audio enhancement model. This results were obtained when using the "Large" model and evaluated for SNR = 3, 6 and 9.

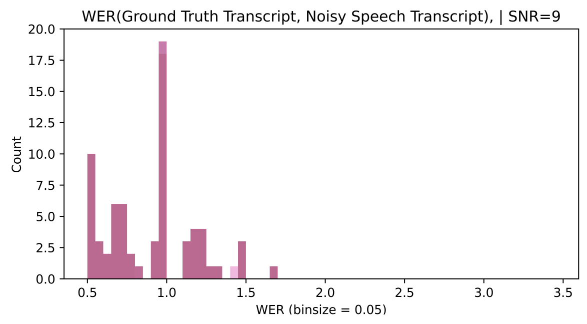


(a) Num Outliers with WER $\geq 50\% = 163$

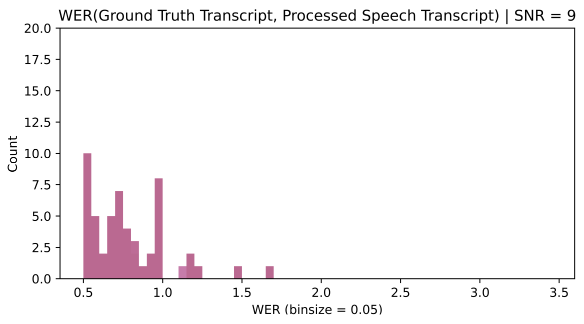


(b) Num Outliers with WER $\geq 50\% = 101$

Figure 3: Outliers for SNR = 6



(a) Num Outliers with WER $\geq 50\% = 60$



(b) Num Outliers with WER $\geq 50\% = 43$

Figure 4: Outliers for SNR = 9

SNR (in dB)	Raw Input Audio	Processed Audio	Reduction in Outliers
3	321	275	46
6	163	101	62
9	60	43	17

Table 4: Outlier Reduction for Whisper - Large

5.2 Average Improvements

The following presents average WER before processing, after processing and the improvements provided by the model. The average word-error-rates are presented for the four biggest models: Large, Turbo, Medium and Small. As expected we notice that the error rates are smallest for the largest models and largest for the smaller models. After our processing, we see that there is an average improvement of 6.7% at SNR = 3, 4.2% at SNR=6, 1.37% at SNR=9.

From the tables we see that the highest error rates are for SNR=3. This is expected as at that SNR, the two speakers are more or less at the same volume and even humans have a hard-time differentiating what is spoken. At SNR = 6, the difference is more noticeable and even better at SNR = 9. For SNRs above 9, it was noticed that the Whisper models were rather robust and the improvements in performance was minute.

Large

SNR	**Noisy**	**Processed**	**Δ WER**
3	24.1%	17.3%	6.8%
6	13.9%	9.3%	4.6%
9	7.1%	6.1%	1.0%

Turbo

SNR	**Noisy**	**Processed**	**Δ WER**
3	27.88%	20.24%	7.64%
6	14.52%	10.34%	4.18%
9	7.79%	6.26%	1.53%

Medium

SNR	**Noisy**	**Processed**	**Δ WER**
3	30.21%	23.72%	6.49%
6	15.57%	11.77%	3.80%
9	10.13%	8.61%	1.52%

Small

SNR	**Noisy**	**Processed**	**Δ WER**
3	34.40%	28.54%	5.85%
6	20.73%	16.50%	4.23%
9	13.96%	12.5%	1.46%

5.3 Future Directions

One of the fundamental challenges of developing an audio-processing model to improve transcription errors is the breakage of computation graphs in the pipeline. The backpropagation algorithm [8] relies on the differentiability of the computation pipeline we build, called the computation graph.

The computation graph becomes non-differentiable when there are stages in it, that is not written using the same tools or contains code that is not open to the framework (torch, in our case). Most ASRs available are blackboxes, which means that the computation graph breaks at that stage. This means that our model cannot be trained to directly reduce the word-error-rate.

Reinforcement Learning is a machine learning field that primarily deals with stochastic and non-differentiable environments, which it receives rewards from. Thus, borrowing a few methods from it should allow us to train the model to directly deal with the errors, especially policy gradient methods.

Policy gradient methods are a class of reinforcement learning methods that are used to optimize policy functions, directly. They aim to learn a policy that maximizes the expected cumulative reward by adjusting the parameters of the policy directly. These methods use gradient ascent to update the policy, where the gradient is computed with respect to the expected return, often using techniques like the likelihood ratio or the REINFORCE [10] algorithm. One of the key features of policy gradient methods is their ability to work with environments where the action space is continuous or large, which makes it appropriate for audio signals.

Thus, reformulating our project as a reinforcement learning problem where the policy is our denoising model, the state-space as the noisy-speech distribution, the action-space as the clean-speech distribution and the rewards as a negative function of the loss, we can directly deal with reducing the transcription errors despite the broken computation graph. This will be investigated later.

5.4 Github Repository

Link to github project page `https://github.com/vrsreeganesh/asr-enhancer`

References

[1] L. D. Consortium, "Timit acoustic-phonetic continuous speech corpus," 1993.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016.

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[4] J. Kong, J. Kim, and J. Bae, "Hifi-gan: Generative adversarial networks for efficient and high fidelity speech synthesis," 2020.

[5] K. Kumar, R. Kumar, T. de Boissiere, L. Gestin, W. Z. Teoh, J. Sotelo, A. de Brebisson, Y. Bengio, and A. Courville, "Melgan: Generative adversarial networks for conditional waveform synthesis," 2019.

[6] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," 2022.

[7] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[9] D. Stoller, S. Ewert, and S. Dixon, "Wave-u-net: A multi-scale neural network for end-to-end audio source separation," 2018.

[10] R. S. Sutton, D. A. McAllester, Y. Mansour, and S. P. Singh, "Policy gradient methods for reinforcement learning with function approximation," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 12, pp. 1057–1063, 2000.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.