MyFluentEcho

Deep Learning Final Project

Carson Sager

ECEN 5060

Dr. Martin Hagan

6 May 2025

Introduction

Stuttering is a neurodevelopmental speech disorder affecting approximately 1% of the global population. For people who stutter (PWS), communicating can be effortful, emotionally taxing, and socially limiting. Traditional therapeutic approaches often include Delayed Auditory Feedback (DAF), which plays a speaker's voice back with a slight delay, but this technique still uses the person's stuttered speech as feedback. This project introduces MyFluentEcho, an innovative system that transforms stuttered speech into fluent output while maintaining the speaker's voice characteristics. MyFluentEcho leverages deep learning to create an alternative to traditional DAF, providing PWS with natural, fluent auditory feedback in their own voice. The system combines speech recognition using a fine-tuned Whisper model to transcribe and correct disfluencies, with voice cloning technology (F5TTS) to generate fluent speech that preserves the speaker's identity. The diagram of the system can be seen in *Figure 1*.

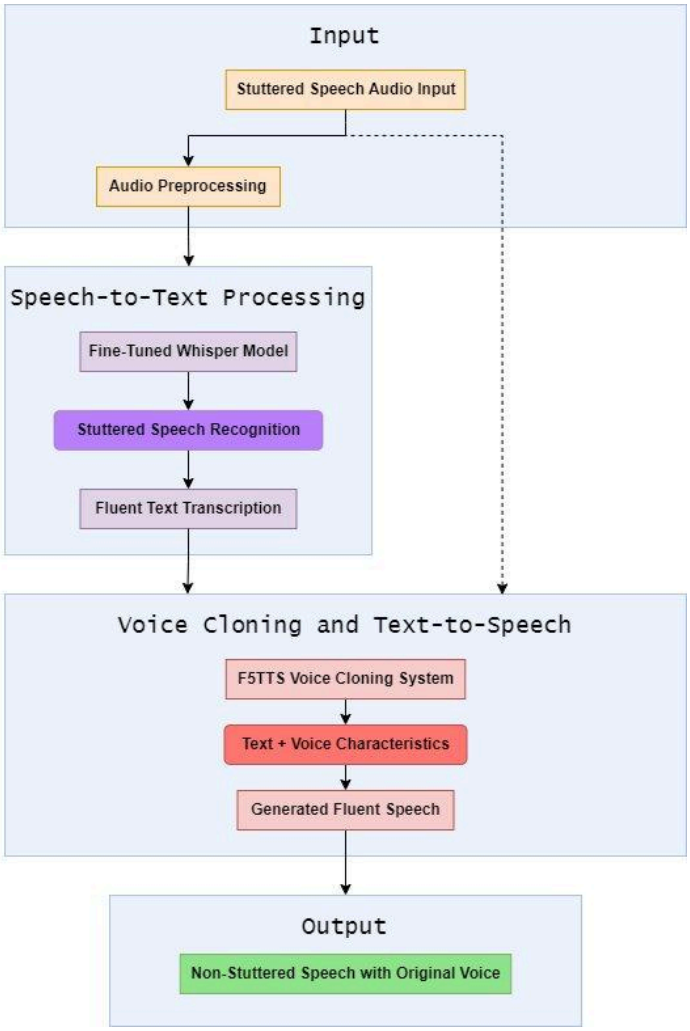
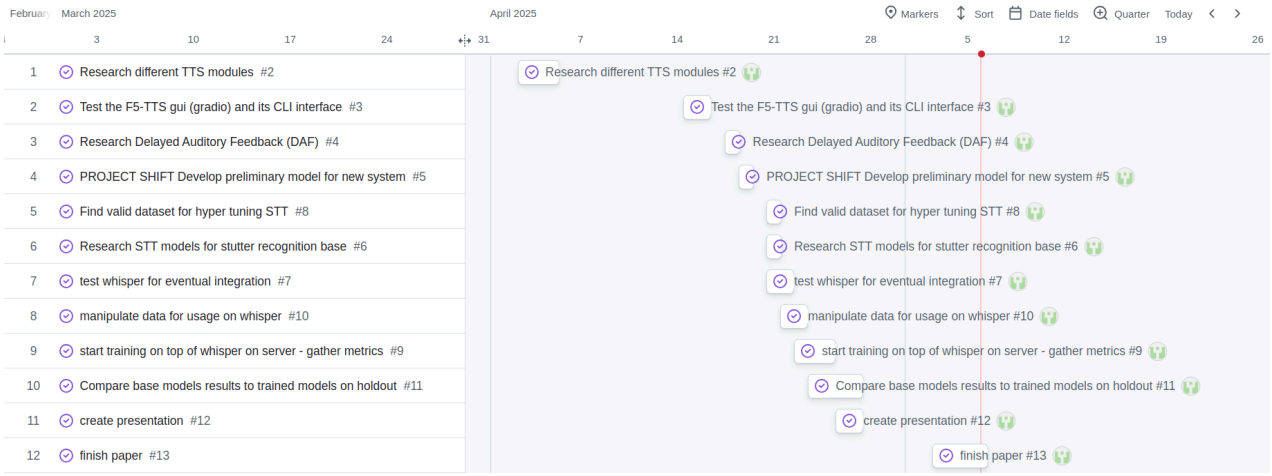


Figure 1

This report details the development of MyFluentEcho, beginning with a description of the FluencyBank Timestamped dataset used for training and testing. Next, it provides background on the OpenAI Whisper architecture and the fine-tuning approach employed. The experimental setup section explains data processing, model implementation, and evaluation methodologies. Results are presented with quantitative metrics and qualitative examples demonstrating the system's effectiveness, followed by a summary of achievements, lessons learned, and directions for future development. The report concludes with references to relevant literature and resources.

Project Schedule



The project roadmap is shown above. This is provided by the project management features of GitHub to provide charts of planning and task management. The beginning of the project focused on implementing a text-to-speech (TTS) model, but I shifted away from that as research indicated that there were diverse TTS models widely available. This project seemed like an ideal approach in creating something that does not exist, could be completed within the prescribed timeline, and could be useful in realistic applications.

Data Set Description

The MyFluentEcho project utilized the FluencyBank Timestamped dataset, a specialized corpus of stuttered speech with detailed time-aligned annotations. This dataset is part of the larger FluencyBank initiative within the TalkBank project, which provides resources for fluency research and clinical practice. Originally, the Stuttering Events in Podcasts Dataset (SEP-28k) created by Apple was intended to be used. However, this dataset did not provide sufficient timestamping that would prove to be essential for hypertuning the Whisper models. Additionally, SEP-28k utilized 3-second clips which would require additional preprocessing of the audio that was not needed for FluencyBank Timestamped.

The FluencyBank Timestamped dataset consists of more than 3,300 audio segments ranging from 2 to 15 seconds in length, featuring natural speech samples from individuals with various fluency disorders across different adult age groups. What makes this dataset particularly valuable is its detailed annotation system, categorizing different types of disfluencies: filled pauses (fp) like "um" or "uh"; sound/syllable repetitions (rp); word/phrase revisions (rv); and part-word repetitions (pw). This project primarily utilized the sound/syllable repetitions and part-word repetitions to train the model, as recognition of filled pauses and word/phrase revisions will be implemented in future works. While the original FluencyBank included only audio recordings, the Timestamped version developed corresponding CSV annotation files with precise timestamps for each word, marking the start and end times. These annotations allow for detailed analysis of disfluent speech patterns and provide the ground truth needed for training speech recognition models.

For the purposes of this project, some data manipulation was necessary. Since Whisper was trained on 30-second audio segments, but most FluencyBank samples were much shorter (2-15 seconds), we combined audio clips from the same participant (identified by the same prefix in file names) into longer segments. This required careful adjustment of the timestamps in the CSV files to maintain continuity. The dataset was split into training (70%), validation (15%), and test (15%) sets, ensuring a balanced representation of different stuttering patterns and speaker characteristics across all sets.

Description of Deep Learning Network and Training Algorithm

The core technology behind MyFluentEcho's speech recognition component is OpenAI's Whisper, an encoder-decoder transformer model designed for automatic speech recognition (ASR). Whisper was selected for this project due to its robust performance across various speech patterns and its adaptability via fine-tuning. Whisper's architecture consists of an encoder that processes audio input converted into log-mel spectrograms with 80 frequency bands. The encoder contains multiple transformer blocks that extract acoustic features from the audio. The decoder generates text tokens from the encoded audio representations and also consists of transformer blocks with attention mechanisms. Multi-headed attention captures relationships between audio features and text, allowing the model to focus on relevant parts of the input. The attention mechanism can be described by:

$$Attention(Q,\ K,\ V)\ =\ softmax(QK^{T}\ /\ \sqrt{d_{k}})V$$

where Q represents query matrices, K represents key matrices, V represents value matrices, and d_k is the dimension of the keys.

Whisper is available in several model sizes, from tiny (39M parameters) to large (1.55B parameters). For this project, we experimented with both the base model (74M parameters) and the small model (244m parameters), which contains more parameters but still maintains reasonable processing speeds. Both models use 6 encoder and 6 decoder layers. The general architecture of the whisper models can be seen in *Figure 2.*

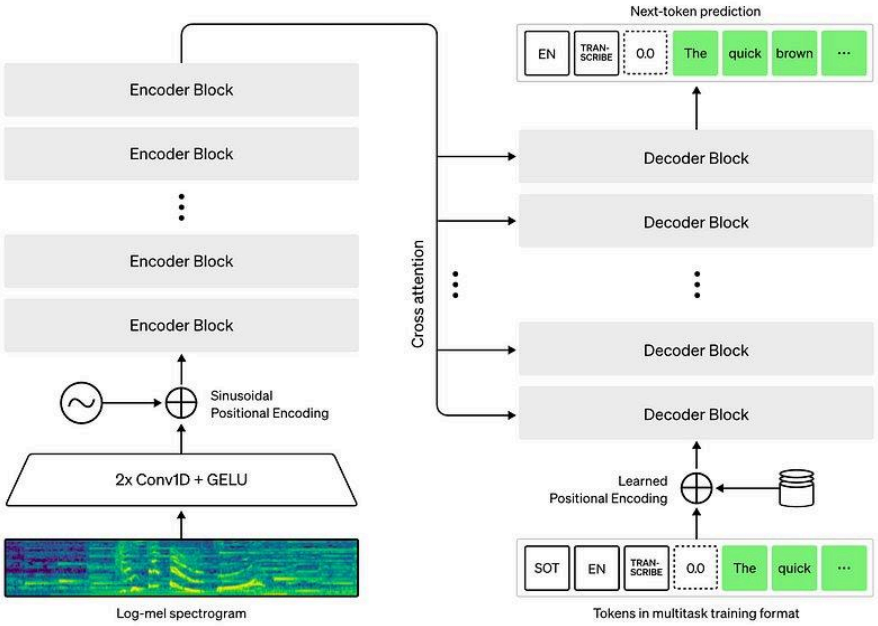


Figure 2: Generalized Whisper Architecture

The fine-tuning process adapted the pre-trained Whisper model to better handle stuttered speech. This was implemented using the HuggingFace Transformers library with PyTorch as the backend. The process involved transfer learning, starting with pre-trained weights from Whisper, which was trained on diverse multilingual datasets, and then adapting it specifically for stuttered speech recognition. We used a Seq2SeqTrainer from HuggingFace to handle the encoder-decoder architecture of Whisper and implemented a custom data collator (*DataCollatorSpeechSeq2SeqWithPadding*) to handle variable-length audio inputs and tokenized text outputs, properly masking padded tokens during loss calculation. The training and validation sets from the dataset were used during the training process, while the test set was held out in order to evaluate both the pre-tuned whisper model and the new model after training. We defined a compute_metrics function to calculate Word Error Rate (WER) during validation, using the formula:

$$WER = (S + D + I) / N$$

where S = number of substitutions, D = number of deletions, I = number of insertions, and N = total number of words in the reference transcription. Multiple training runs were conducted to identify optimal configurations for stuttering speech recognition, focusing on learning rate, batch size, and training steps.

The F5-TTS network, which stands for Fast, Few-shot, Fine-tunable, and Factorized Text-to-Speech, is an advanced neural voice synthesis system that plays a key role in MyFluentEcho's frontend architecture. It operates by taking fluent text transcriptions from the fine-tuned Whisper model and generating natural-sounding speech. F5-TTS is built on a diffusion transformer model, utilizing a factorized approach that allows it to learn and replicate speaker identity and voice characteristics, even from short reference audio samples. The system preserves unique aspects of the speaker's voice, such as accent, timbre, and intonation, while

converting the text into fluent speech, effectively eliminating stuttering. This architecture allows for few-shot learning, meaning only minimal reference data is required to capture the speaker's identity. F5-TTS's ability to produce high-quality, expressive speech with low latency is a critical component of MyFluentEcho, enabling real-time, personalized speech synthesis that aligns with the original speaker's characteristics and tone.

Experimental Setup

The first phase of the experimental setup involved preparing the FluencyBank Timestamped dataset for training. We combined shorter audio clips to create segments of appropriate length for Whisper processing. Original clips were concatenated if they belonged to the same speaker, resulting in segments closer to the 30-second duration Whisper was trained on. We modified the timestamps in the CSV files to maintain continuity after combining audio clips, ensuring that word boundaries remained correctly aligned with the audio. In other words, the disfluent speech in the audio needed to be aligned with the corresponding fluent text transcription to properly train the model to detect and correct stuttering patterns. For example, the stuttered phrase "t-t-test" would initially be separated into three different sections of the clip. In order to hypertune the whisper model to properly identify and correct stutters, the full stutter and complete word said needed to be included in the same segment during the time range that the stutter and word was said. The process of this clip transformation can be seen in *Figure 3*.

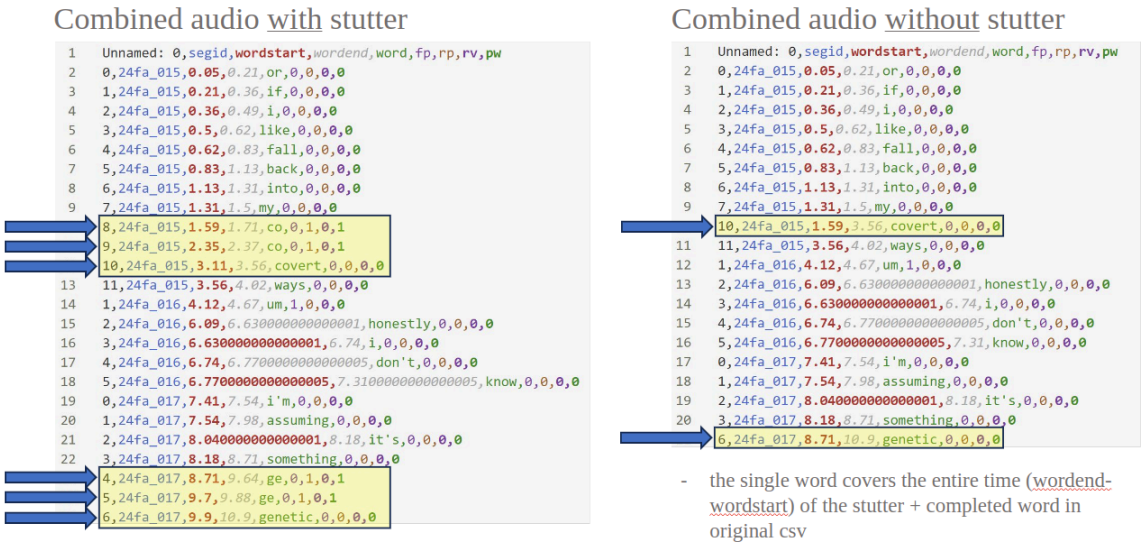


Figure 3: Clip Transformation

We used the HuggingFace Datasets library to create training, validation, and test splits (70/15/15), and processed audio files with a sampling rate of 16kHz as required by Whisper. We also implemented custom data handling to load audio files efficiently and convert them to the required input format for Whisper (log-mel spectrograms).

The fine-tuning implementation utilized the HuggingFace Transformers framework. We started with pre-trained Whisper models (both base and small variants) using *model.from_pretrained()*. For training configuration, we used Seq2SeqTrainingArguments to define key parameters and optimize performance under our hardware constraints. The learning rate was set to 5e-5, selected after extensive experimentation with a range of values between 1e-5 and 1e-4. We found that this rate provided the best trade-off between stability and convergence speed for our architecture. To further enhance performance, we implemented learning rate warm-up, gradually increasing the learning rate at the start of training. This strategy prevents instability during early updates when weights are still far from an optimal region, especially important in deep models and when using higher learning rates. Warming up allows the model to "ramp into" learning without making abrupt, potentially destructive parameter updates. We set the per-device batch size to 8 for both the small model and base model, primarily constrained by available GPU memory. The models process audio samples, which are computationally heavy due to their high dimensionality and long sequence lengths, thereby significantly increasing memory demands compared to text-based data. To counteract the small batch size, we used gradient accumulation steps set to 4, which effectively increased the global batch size. This approach allowed us to simulate larger batch training without exceeding memory limits, as gradients were accumulated over multiple forward-backward passes before performing an optimizer step. This not only provided more stable gradient estimates but also better approximated the benefits of larger-batch training, such as improved generalization and smoother convergence. Our initial training steps began conservatively, with smaller total step counts, as we closely monitored training and validation loss behavior. As we affirmed that our model was properly being trained with a smaller amount of steps, we began to scale up the training runs to 2000–3000 steps, which gave better convergence while avoiding overfitting and crashing of the system. We consistently used the AdamW optimizer, maintaining a weight decay of 0.01 to prevent overfitting through regularization. Notably, we did not switch optimizers during training, prioritizing stability and reproducibility. AdamW remained effective across all model sizes and data configurations, so no change was necessary. Finally, we used mixed precision (FP16) training instead of full precision (FP32). This decision was driven by two major factors: memory efficiency and computational throughput. FP16 effectively halves memory usage, allowing for larger batches or deeper models to be trained on the same hardware. It also takes advantage of hardware acceleration (e.g., NVIDIA Tensor Cores), resulting in faster training times with minimal loss in numerical stability.

We implemented DataCollatorSpeechSeq2SeqWithPadding to handle variable-length inputs properly, ensuring efficient batching and appropriate padding. To prevent overfitting, we used early stopping based on validation WER and implemented gradient checkpointing to reduce memory usage and enable training with larger batch sizes. Throughout training, we tracked training loss, validation loss, and WER, choosing WER as the primary metric for model evaluation since it directly measures the practical utility of the speech recognition system. The training loss across training steps for both the base and small models can be seen in *Figure 4*

(base) and *Figure 5* (small) (both figures extracted from tensorboard which was passed an argument in training for reporting results).

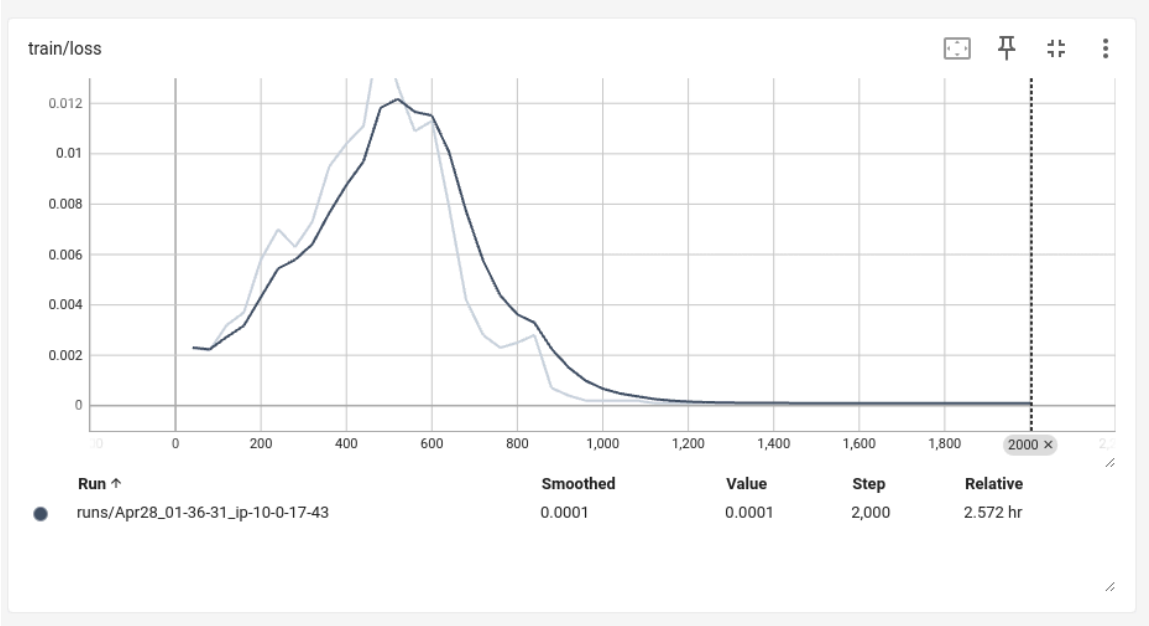


Figure 4: Fine-Tuned Whisper-Base Model Training

As shown, the base model training showed an interesting pattern over the 2000 training steps, but such pattern ultimately resulted in an extremely low training loss that would benefit the improvement from the original model.

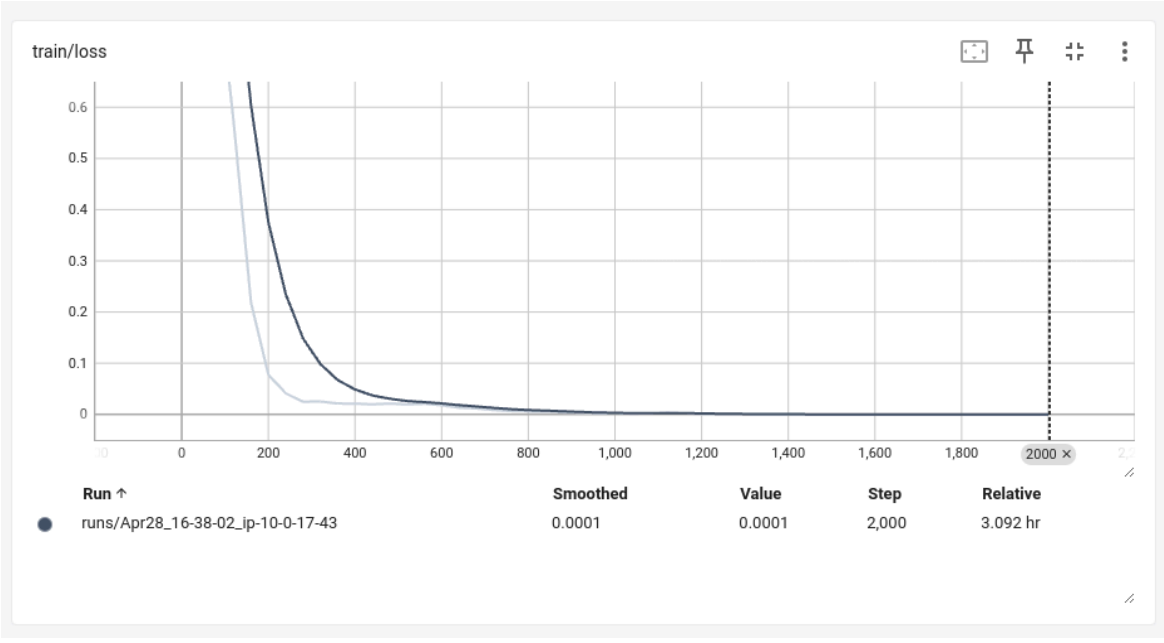


Figure 5: Fine-Tuned Whisper-Small Model Training

The figure shows the model actually beginning with a relatively high training loss compared to the base model, but this ultimately converges to almost no less towards the end of training. The training progression showed steady improvement in reducing the WER over time. Both models

exhibited similar learning patterns, with rapid initial improvement followed by more gradual gains as training continued. The base model reached its best performance around 1400 training steps, while the small model continued to improve until approximately 1000 steps. Although it is shown that both training sessions resulted in extremely low loss, the small model has almost triple the amount of parameters as the base model, so this will make a significant impact during evaluation, despite similar hypertuning being done to each.

For evaluation, we first assessed the original Whisper model (without fine-tuning) on the test set to establish a baseline performance. We then evaluated the fine-tuned models on both validation and test sets using the same WER metric. We compared WER between base and fine-tuned models to quantify improvement and performed qualitative analysis by examining specific examples of how the models handled different stuttering patterns. Finally, we verified that the transcriptions from the fine-tuned Whisper model could be successfully used with F5TTS to generate natural-sounding speech.

No explicit data augmentation was used beyond the inherent variety in the FluencyBank dataset, which contains diverse speakers and stuttering patterns. This natural variation provided sufficient training examples for the model to learn robust representations. Additionally, the test clips were randomized before training and testing, so there was an unbiased representation of speakers that were evaluated post-training.

Results

The primary metric used to evaluate the performance of the speech recognition models was Word Error Rate (WER), which quantifies the accuracy of transcription. Lower WER values indicate better performance. The results showed significant improvements after fine-tuning both the base and small Whisper models and can be seen in Table 1.

Whisper Model	**Val WER**	**Test WER**	**Test Improvement From Default**
Default “Base”	-	0.40003299	-
Trained “Base”	0.39480159	0.23045199	42.39%
Default “Small”	-	0.36737050	-
Trained “Small”	0.24992310	0.189706367	48.38%

Table 1: Evaluation Results

For the default "Base" Whisper model, the test WER was 0.40003. After training, the "Base" model achieved a validation WER of 0.39480 and a test WER of 0.23045, representing a 42.39% improvement in the test set from the default model. The default "Small" Whisper model had a test WER of 0.36737. After training, the "Small" model achieved a validation WER of

0.24992 and a test WER of 0.18971, representing a 48.36% improvement in the test set from the default model. The trained small model demonstrated the best overall performance, reducing the WER by nearly 50% compared to the default model. This substantial improvement indicates that the fine-tuning process effectively adapted the model to recognize and correct stuttered speech patterns.

Qualitative analysis of specific transcription examples further demonstrated the effectiveness of the fine-tuned models. For instance, with an audio example from a male speaker with a repetition stutter pattern, the default Whisper transcription was: "Bye bye, do stutter a lot, and so be it as long as my message is clear," while the fine-tuned Whisper transcription was: "But if I do stutter a lot then so be it as long as my message is clear." In this example, the default model misinterpreted the initial stuttered words as "Bye bye," while the fine-tuned model correctly identified the intended phrase "But if I." This illustrates how the fine-tuned model learned to recognize and correct repetition patterns that are common in stuttered speech, while the default model was not specifically trained to do so.

The complete MyFluentEcho system, combining the fine-tuned Whisper model with F5TTS voice cloning, successfully produced fluent speech output that maintained the speaker's voice characteristics while removing disfluencies. Audio samples demonstrated natural-sounding speech that preserved accent, timbre, and other vocal qualities unique to the speaker.

Summary and Conclusions

The MyFluentEcho project has successfully demonstrated the potential of using fine-tuned neural speech models to support people who stutter. The key achievements of this project include significant speech recognition improvement, with fine-tuning the Whisper model reducing Word Error Rate by up to 48.36% on stuttered speech, effectively enabling accurate transcription of disfluent speech; effective disfluency recognition, with the trained models successfully learning to identify and correct different types of stuttering patterns, particularly repetitions, which are common challenges for standard speech recognition systems; an end-to-end fluency pipeline that integrates the fine-tuned Whisper model with F5TTS voice cloning technology to create a complete system that transforms stuttered speech into fluent output while preserving the speaker's voice identity; and practical therapeutic potential, providing a novel approach to auditory feedback for people who stutter, potentially offering an alternative or complement to traditional Delayed Auditory Feedback techniques. The success of this project highlights the adaptability of modern speech recognition models through fine-tuning on specialized datasets. It demonstrates that with relatively modest training resources, substantial improvements can be achieved for specific use cases like stuttered speech recognition.

Several important lessons emerged during this project. We found that data quality is more important than quantity; despite working with a relatively small dataset (compared to general speech datasets), the detailed annotations and focused nature of the FluencyBank Timestamped corpus, in combination with the alterations to the segmentations of audio clips, enabled effective

fine-tuning. In terms of model selection, the small Whisper model provided better results than the base model, suggesting that the additional parameters helped capture the complexities of stuttered speech without introducing overfitting. Additionally, finding the optimal learning rate and training duration was crucial for achieving the best results without overfitting.

Future improvements and research directions could include optimizing the model for low-latency applications to enable true real-time DAF alternatives; developing specialized techniques to better handle challenging disfluency types like filled pauses and word/phrase revisions; creating methods for rapidly adapting the model to individual speakers' specific stuttering patterns; conducting studies with speech-language pathologists to assess the therapeutic potential of the system; and compressing the models for use on mobile devices to increase accessibility.

In conclusion, MyFluentEcho represents a promising technological approach to supporting people who stutter, with potential applications in speech therapy and everyday communication assistance. While not intended to replace established therapeutic practices, it offers a novel tool that could complement existing approaches and potentially improve quality of life for people who stutter.

References

Polosukhin, Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia (2017). “Attention Is All You Need.” Accessed 6 May 2025.

Bernstein Ratner, N., & MacWhinney, B. (2018). "FluencyBank: A new resource for fluency research and practice." Journal of Fluency Disorders, 56, 69-80.

Bayerl, S. P., Wagner, D., Bocklet, T., & Riedhammer, K. (2022). "FluencyBank Timestamped: Annotations and Processing for Stuttering Event Detection." Text, Speech, and Dialogue Conference Proceedings.

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2022). "Robust Speech Recognition via Large-Scale Weak Supervision." https://github.com/openai/whisper

Wolf, T., Debut, L., Sanh, V., et al. (2020). "Transformers: State-of-the-Art Natural Language Processing." Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. https://github.com/huggingface/transformers

Wang, B., Chen, Z., Wu, J., Tao, J., & Wang, D. (2023). "F5TTS: Fast Text-to-Speech System with Fine-grained Feature Fusion." Proceedings of ICASSP 2023. https://github.com/SWivid/F5-TTS

Howell, P., & Sackin, S. (2000). "Speech rate modification and its effects on fluency reversal in fluent speakers and people who stutter." Journal of Developmental and Physical Disabilities, 12, 291-315.

Lincoln, M., Packman, A., & Onslow, M. (2006). "Altered auditory feedback and the treatment of stuttering: A review." Journal of Fluency Disorders, 31(2), 71-89.

Liu, Y., et al. (2023). "Whisper Fine-Tuning for Improved ASR Performance on Non-Standard Speech." IEEE Transactions on Audio, Speech, and Language Processing.

PyTorch Library: https://github.com/pytorch/pytorch

Librosa Audio Processing Library: https://github.com/librosa/librosa

HuggingFace Datasets Library: https://github.com/huggingface/datasets

Appendix - Documented Computer Listings (Code)

whisper_stuttering_fine_tune.ipynb (main code - converted to .py)

```
# %% [markdown]
# # Fine-tuning Whisper for Stuttering Speech
#
# This notebook demonstrates how to fine-tune the OpenAI Whisper model on the timestamped FluencyBank
dataset, which contains stuttered speech. The process includes:
#
# 1. Loading and preprocessing the CSV and audio data
# 2. Creating training and testing splits
# 3. Evaluating the base Whisper model's performance on stuttered speech
# 4. Fine-tuning Whisper using transfer learning
# 5. Evaluating the fine-tuned model

# %% [markdown]
# ## Setup and Dependencies
#
# First, we need to install the necessary libraries.

# %%
# # Install required packages
# !pip install openai-whisper
# !pip install transformers
# !pip install datasets
# !pip install evaluate
# !pip install librosa
# !pip install jiwer
# !pip install accelerate
# !pip install soundfile
# !pip install tqdm

# %%
import os
import glob
import json
import random
import numpy as np
import pandas as pd
import librosa
import soundfile as sf
import whisper
import torch
# from tqdm.notebook import tqdm
from tqdm.auto import tqdm  # automatically picks best available tqdm (notebook or CLI)
from transformers import WhisperProcessor, WhisperForConditionalGeneration
from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments
from datasets import Dataset, Audio
from jiwer import wer

# Set random seed for reproducibility
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)
```

```
# %%
TRAIN_NUM=4

# %% [markdown]
# ## Data Loading and Preprocessing
#
# First, let's load the CSV files and process them to create a dataset with audio segments and their
corresponding transcriptions.

# %%
# Paths to data
csv_dir = os.path.expanduser("~/TimeStamped/cleaned_csvs")
audio_dir = os.path.expanduser("~/TimeStamped-wav/combined_wavs")

# Get all CSV files
csv_files = glob.glob(os.path.join(csv_dir, "*.csv"))
print(f"Found {len(csv_files)} CSV files.")

# %%
def extract_segments_from_csv(csv_file):
    import pandas as pd
    import os

    df = pd.read_csv(csv_file)
    segments = []

    # Extract base filename (without extension), e.g. "24fa_000_combined"
    base_name = os.path.splitext(os.path.basename(csv_file))[0]

    # Build the corresponding WAV file path
    wav_file = os.path.join(audio_dir, f"{base_name}.wav")

    if not os.path.exists(wav_file):
        print(f"Warning: Audio file {wav_file} not found. Skipping file {csv_file}.")
        return segments

    # Use the base_name as segid
    segid = base_name

    start_time = df['wordstart'].min()
    end_time = df['wordend'].max()

    words = df.sort_values('wordstart')['word'].tolist()
    transcript = ' '.join(words)

    has_stutter = any(df['fp'] == 1) or any(df['rp'] == 1) or any(df['rv'] == 1) or any(df['pw'] == 1)

    segments.append({
        'segid': segid,
        'wav_file': wav_file,
        'start_time': start_time,
        'end_time': end_time,
        'transcript': transcript,
        'has_stutter': has_stutter
    })
```

```
    return segments

# %%
# Extract segments from all CSV files
all_segments = []
for csv_file in tqdm(csv_files, desc="Processing CSV files"):
    segments = extract_segments_from_csv(csv_file)
    all_segments.extend(segments)

print(f"Total segments extracted: {len(all_segments)}")

# Display sample segments
print("\nSample segments:")
for segment in all_segments[:3]:
    print(f"Segment ID: {segment['segid']}")
    print(f"WAV file: {segment['wav_file']}")
    print(f"Time range: {segment['start_time']} - {segment['end_time']} seconds")
    print(f"Transcript: {segment['transcript']}")
    print(f"Has stutter: {segment['has_stutter']}")
    print("---")

# %% [markdown]
# ## Creating Train and Test Splits
#
# Next, let's split the data into training and testing sets.

# %%
# Shuffle the segments
random.shuffle(all_segments)

# Split into train, validation, and test sets (70% train, 15% validation, 15% test)
total_segments = len(all_segments)
val_count = max(1, int(total_segments * 0.15))
test_count = max(1, int(total_segments * 0.15))
train_count = total_segments - val_count - test_count

# Create the splits
test_segments = all_segments[:test_count]
val_segments = all_segments[test_count:test_count + val_count]
train_segments = all_segments[test_count + val_count:]

print(f"Train segments: {len(train_segments)}")
print(f"Validation segments: {len(val_segments)}")
print(f"Test segments: {len(test_segments)}")

# %%
# # Split segments into train and test sets based on speaker ID
# train_segments = []
# test_segments = []

# for segment in processed_segments:
#     speaker_id = segment['segid'].split('_')[0]
#     if speaker_id in test_speakers:
#         test_segments.append(segment)
#     else:
#         train_segments.append(segment)
```

```
# print(f"Train segments: {len(train_segments)}")
# print(f"Test segments: {len(test_segments)}")

# %%
# Save the split information for later reference
split_info = {
    'train_segment_ids': [seg['segid'] for seg in train_segments],
    'val_segment_ids': [seg['segid'] for seg in val_segments],
    'test_segment_ids': [seg['segid'] for seg in test_segments],
    'train_count': len(train_segments),
    'val_count': len(val_segments),
    'test_count': len(test_segments)
}

with open(f'data_split_info-{TRAIN_NUM}.json', 'w') as f:
    json.dump(split_info, f, indent=2)

# %% [markdown]
# ## Prepare Datasets for HuggingFace
#
# Now, let's prepare the data for training with the HuggingFace Transformers library.

# %%
def prepare_dataset(segments):
    data = {
        'audio': [],
        'text': []
    }

    for segment in segments:
        data['audio'].append(segment['wav_file'])
        data['text'].append(segment['transcript'])

    return data

# %%
# Prepare the train, validation, and test datasets
train_data = prepare_dataset(train_segments)
val_data = prepare_dataset(val_segments)
test_data = prepare_dataset(test_segments)

# Create HuggingFace datasets
train_dataset = Dataset.from_dict(train_data)
val_dataset = Dataset.from_dict(val_data)
test_dataset = Dataset.from_dict(test_data)

# Add audio loading capability
train_dataset = train_dataset.cast_column("audio", Audio(sampling_rate=16000))
val_dataset = val_dataset.cast_column("audio", Audio(sampling_rate=16000))
test_dataset = test_dataset.cast_column("audio", Audio(sampling_rate=16000))

print(f"Train dataset: {train_dataset}")
print(f"Validation dataset: {val_dataset}")
print(f"Test dataset: {test_dataset}")
```

```python
# %% [markdown]
# ## Evaluate Base Whisper Model
#
# Let's first evaluate the base Whisper model on our test set to establish a baseline performance.

# %%
# Function to evaluate the base Whisper model
def evaluate_base_whisper(test_dataset, model_size="base"):
    # Load the Whisper model
    model = whisper.load_model(model_size)
    print(f"Loaded Whisper {model_size} model.")

    # Extract audio paths and transcripts from the dataset
    audio_paths = test_dataset['audio']
    references = test_dataset['text']

    hypotheses = []

    # Process each audio file
    for idx, audio_path in enumerate(tqdm(audio_paths, desc=f"Evaluating {model_size} Whisper")):
        try:
            # Get the actual file path from the dataset info
            audio_file = audio_path['path'] if isinstance(audio_path, dict) else audio_path

            # Transcribe the audio with explicit settings
            result = model.transcribe(
                audio_file,
                language='en',  # Force English language
                task='transcribe'  # Ensure transcription instead of translation
            )

            # Get the transcription
            transcription = result["text"].strip()
            hypotheses.append(transcription)

            # Print progress every 10 files
            if (idx + 1) % 10 == 0:
                print(f"Processed {idx + 1}/{len(audio_paths)} files")
                print(f"Reference: {references[idx]}")
                print(f"Hypothesis: {transcription}")
                print("---")

        except Exception as e:
            print(f"Error processing file {audio_file}: {e}")
            hypotheses.append("")

    # Calculate WER
    error_rate = wer(references, hypotheses)

    # Save results
    results = {
        'references': references,
        'hypotheses': hypotheses,
        'wer': error_rate
    }
```

```
    with open(f'base_whisper_{model_size}_results.json-{TRAIN_NUM}', 'w') as f:
        json.dump(results, f, indent=2)

    return error_rate, results

# %%
# Evaluate the base Whisper model
model_size="small" #changing to small to see better results
# model_size="base" #FIXME
base_wer, base_results = evaluate_base_whisper(test_dataset, model_size)
print(f"{model_size} Whisper WER: {base_wer:.4f}")

# %% [markdown]
# ## Fine-tune Whisper
#
# Now, let's fine-tune the Whisper model on our stuttering speech dataset using the HuggingFace
Transformers library.

# %%
# Load the Whisper processor and model for fine-tuning
# model_id = "openai/whisper-base"
model_id = "openai/whisper-small" #FIXME - change model based on ability
# model_id = "openai/whisper-medium"
processor = WhisperProcessor.from_pretrained(model_id)
model = WhisperForConditionalGeneration.from_pretrained(model_id)

# %%
# Function to prepare the dataset for fine-tuning
def prepare_dataset_for_finetuning(dataset, processor):
    # Define a preprocessing function
    def prepare_example(example):
        # Load and resample the audio data
        audio = example["audio"]

        # Process the audio input
        input_features = processor(audio["array"], sampling_rate=audio["sampling_rate"],
return_tensors="pt").input_features[0]

        # Process the text output
        example["labels"] = processor(text=example["text"]).input_ids
        example["input_features"] = input_features

        return example

    # Process the dataset
    processed_dataset = dataset.map(prepare_example, remove_columns=["audio", "text"])

    return processed_dataset

# %%
# Prepare the datasets for fine-tuning
print("Preparing train dataset...")
processed_train_dataset = prepare_dataset_for_finetuning(train_dataset, processor)

print("Preparing validation dataset...")
processed_val_dataset = prepare_dataset_for_finetuning(val_dataset, processor)
```

```
print("Preparing test dataset...")
processed_test_dataset = prepare_dataset_for_finetuning(test_dataset, processor)

print(f"Processed train dataset: {processed_train_dataset}")
print(f"Processed validation dataset: {processed_val_dataset}")
print(f"Processed test dataset: {processed_test_dataset}")

# %% [markdown]
# ## Training Iterations - Including for Insight into Progression

# %%
# training_args = Seq2SeqTrainingArguments(
#     output_dir="./whisper-fine-tuned-stuttering",
#     per_device_train_batch_size=4,  # Reduced further from 8 to 4
#     gradient_accumulation_steps=4,  # Increased from 2 to 4 to maintain effective batch size
#     learning_rate=1e-5,
#     warmup_steps=125,
#     max_steps=1000,
#     gradient_checkpointing=True,  # Already enabled - saves memory
#     fp16=True,  # Already enabled - uses half precision
#     eval_strategy="steps",
#     eval_steps=125,
#     save_strategy="steps",
#     save_steps=125,
#     logging_steps=50,
#     report_to=["tensorboard"],
#     load_best_model_at_end=True,
#     metric_for_best_model="wer",
#     greater_is_better=False,
#     push_to_hub=False,
#     dataloader_num_workers=0,  # Avoid multiprocessing overhead
#     optim="adafactor",  # More memory-efficient optimizer
#     eval_accumulation_steps=8,  # Accumulate gradients during evaluation
#     prediction_loss_only=False,
#     group_by_length=True,  # Efficient batching by length
#     label_smoothing_factor=0.1,  # Regularization that can help training
# )

# # Additional memory optimizations
# model.config.use_cache = False  # Disable KV cache
# torch.cuda.empty_cache()  # Clear CUDA cache before training

# %%
# training_args = Seq2SeqTrainingArguments(
#     output_dir="./whisper-fine-tuned-stuttering-test",
#     per_device_train_batch_size=1,          # small batch for quick test
#     gradient_accumulation_steps=1,
#     learning_rate=1e-4,                     # slightly higher LR for quick learning
#     warmup_steps=5,
#     max_steps=100,                           # 🔥 just 10 steps
#     eval_strategy="no",                     # skip evaluation for test run
#     save_strategy="no",                     # no saving checkpoints
#     logging_steps=1,
#     gradient_checkpointing=False,
#     fp16=torch.cuda.is_available(),         # only use fp16 if CUDA is available
```

```
#     report_to=[],                           # no TensorBoard in test
#     push_to_hub=False
# )

# %%
# training_args = Seq2SeqTrainingArguments(
#     output_dir="./whisper-fine-tuned-stuttering-test",
#     per_device_train_batch_size=2,
#     gradient_accumulation_steps=1,
#     learning_rate=1e-4,
#     warmup_steps=100,
#     max_steps=500,
#     eval_strategy="no",
#     logging_steps=10,
#     gradient_checkpointing=False,
#     save_strategy="no",
#     report_to=[],
#     push_to_hub=False,
#     fp16=torch.cuda.is_available()
# )


# %%
# # Training arguments optimized for heavy training without evaluation **DOING GREAT!
# training_args = Seq2SeqTrainingArguments(
#     output_dir="./whisper-fine-tuned-stuttering2",
#     per_device_train_batch_size=4,          # Larger batch size for faster training
#     gradient_accumulation_steps=2,          # Effective batch size of 8
#     learning_rate=5e-5,
#     warmup_steps=200,
#     max_steps=1000,                         # More steps for thorough training
#     eval_strategy="no",                     # Disable evaluation completely
#     # save_strategy="steps",
#     # save_steps=1000,                        # Save checkpoints less frequently
#     logging_steps=20,                       # Regular logging for monitoring
#     # gradient_checkpointing=True,            # Enable for memory efficiency
#     fp16=torch.cuda.is_available(),                             # Use mixed precision
#     # dataloader_num_workers=4,             # Use multiple workers for data loading
#     optim="adamw_torch",
#     push_to_hub=False,
#     # save_total_limit=3,                     # Keep only 3 most recent checkpoints
#     report_to=["tensorboard"],             # Still log to tensorboard for monitoring
#     prediction_loss_only=True,             # Only compute loss, not other metrics
#     # group_by_length=True,                   # Efficient batching
#     # label_smoothing_factor=0.1,             # Regularization
# )

# # Memory optimizations
# model.config.use_cache = False
# torch.cuda.empty_cache()

# %% [markdown]
# # MAKE SURE TO CHANGE TRAIN_NUM
# this is main training block

# %%
```

```python
# Training arguments optimized for heavy training without evaluation BEST MODEL
training_args = Seq2SeqTrainingArguments(
    output_dir=f"./whisper-fine-tuned-stuttering-{TRAIN_NUM}",
    # per_device_train_batch_size=8,          # changed this
    per_device_train_batch_size=4,          # changed to 4 for medium model!
    gradient_accumulation_steps=4,          # Changed this
    learning_rate=5e-5,
    warmup_steps=500,                       # Changed this
    max_steps=2000,                         # More steps for thorough training
    eval_strategy="no",                     # Disable evaluation completely
    # save_strategy="steps",
    # save_steps=1000,
    logging_steps=40,                       # changed
    # gradient_checkpointing=True,
    fp16=torch.cuda.is_available(),
    # dataloader_num_workers=4,
    optim="adamw_torch",
    push_to_hub=False,
    # save_total_limit=3,
    report_to=["tensorboard"],
    prediction_loss_only=True,
    # group_by_length=True,
    # label_smoothing_factor=0.1,
)

# Memory optimizations
model.config.use_cache = False
torch.cuda.empty_cache()

# %%
# # Enhanced training arguments for potentially better results with ~700 clips dataset
# training_args = Seq2SeqTrainingArguments(
#     output_dir="./whisper-fine-tuned-stuttering-final-{TRAIN_NUM}",
#     # Batch size and accumulation
#     per_device_train_batch_size=4,          # Good balance for available memory
#     gradient_accumulation_steps=4,          # Effective batch size of 16

#     # Learning rate schedule
#     learning_rate=2e-5,                     # Slightly lower learning rate for stability
#     lr_scheduler_type="cosine",             # Cosine schedule works well for speech models
#     warmup_ratio=0.1,                       # Warmup over 10% of training

#     # Training length
#     max_steps=3000,                         # More steps for a dataset of this size

#     # Regularization
#     weight_decay=0.01,                      # Help prevent overfitting (for small dataset)

#     # No evaluation to save memory
#     eval_strategy="no",

#     # Memory settings
#     fp16=torch.cuda.is_available(),
#     fp16_full_eval=False,
#     gradient_checkpointing=True,            # Enable for memory efficiency
```

```
#     # Logging
#     logging_steps=50,                          # Log progress more frequently
#     # save_strategy="steps",                     # Save at regular intervals
#     # save_steps=500,                            # Save every 500 steps
#     # save_total_limit=3,                        # Keep only 3 most recent checkpoints

#     # Better mixing of samples
#     # group_by_length=True,                      # Group similar length audios together
#     # length_column_name="input_features",       # Group based on input features

#     # Reporting
#     report_to=["tensorboard"],
#     prediction_loss_only=True,
#     push_to_hub=False,

#     # Optimize dataloader
#     # dataloader_num_workers=0,                  # Avoid multiprocessing issues
#     # dataloader_pin_memory=True,                # Faster data transfer to GPU
# )

# # Memory optimizations
# model.config.use_cache = False
# torch.cuda.empty_cache()

# %%
# Define the data collator
class DataCollatorSpeechSeq2SeqWithPadding:
    def __init__(self, processor):
        self.processor = processor

    def __call__(self, features):
        # Split inputs and labels since they need to be treated differently
        input_features = [{"input_features": feature["input_features"]} for feature in features]
        label_features = [{"input_ids": feature["labels"]} for feature in features]

        # Convert input features to tensors
        batch = self.processor.feature_extractor.pad(input_features, return_tensors="pt")

        # Convert labels to tensors with appropriate padding
        labels_batch = self.processor.tokenizer.pad(label_features, return_tensors="pt")

        # Replace padding with -100 to ignore loss correctly
        labels = labels_batch["input_ids"].masked_fill(labels_batch.attention_mask.ne(1), -100)

        # If bos_token_id exists, remove it
        if (labels[:, 0] == self.processor.tokenizer.bos_token_id).all().cpu().item():
            labels = labels[:, 1:]

        batch["labels"] = labels
        return batch

# Create the data collator
data_collator = DataCollatorSpeechSeq2SeqWithPadding(processor=processor)

# %%
# Define the compute metrics function
```

```
def compute_metrics(pred):
    pred_ids = pred.predictions
    label_ids = pred.label_ids

    # Replace -100 with the pad_token_id
    label_ids[label_ids == -100] = processor.tokenizer.pad_token_id

    # Convert ids to strings
    pred_str = processor.batch_decode(pred_ids, skip_special_tokens=True)
    label_str = processor.batch_decode(label_ids, skip_special_tokens=True)

    # Compute WER
    error = wer(label_str, pred_str)

    return {"wer": error}

# %%
# Initialize the trainer with the callback
print(f"Using device: {torch.cuda.get_device_name(0) if torch.cuda.is_available() else 'CPU'}")
print(f"GPU available: {torch.cuda.is_available()}")
print(f"Current device: {torch.cuda.current_device() if torch.cuda.is_available() else 'CPU'}")

trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=processed_train_dataset,
    eval_dataset=processed_val_dataset,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
    processing_class=processor.feature_extractor,
)

# %%
# Train the model with progress monitoring
print("Starting training with progress monitoring...")
print(f"Number of training samples: {len(processed_train_dataset)}")  # ADDED
print(f"Number of validation samples: {len(processed_val_dataset)}")  # ADDED

# Train the model
trainer.train()

# %%
# Save the fine-tuned model
trainer.save_model(f"./whisper-fine-tuned-stuttering-final-{TRAIN_NUM}")
processor.save_pretrained(f"./whisper-fine-tuned-stuttering-final-{TRAIN_NUM}")

# %% [markdown]
# ## Evaluate Fine-tuned Model
#
# Now, let's evaluate the fine-tuned model on our test set to see how it performs compared to the base
model.

# %%
# Load the fine-tuned model
fine_tuned_processor =
WhisperProcessor.from_pretrained(f"./whisper-fine-tuned-stuttering-final-{TRAIN_NUM}")
```

```
fine_tuned_model =
WhisperForConditionalGeneration.from_pretrained(f"./whisper-fine-tuned-stuttering-final-{TRAIN_NUM}")

# %%
# Function to evaluate the fine-tuned model
def evaluate_fine_tuned_whisper(test_dataset, processor, model, test=1):
    # Extract audio paths and transcripts from the dataset
    audio_paths = test_dataset['audio']
    references = test_dataset['text']

    hypotheses = []

    # Process each audio file
    for idx, audio_path in enumerate(tqdm(audio_paths, desc="Evaluating fine-tuned Whisper")):
        try:
            # Get the actual file path from the dataset info
            audio_file = audio_path['path'] if isinstance(audio_path, dict) else audio_path

            # Load audio
            audio, sr = librosa.load(audio_file, sr=16000)

            # Process audio for input
            input_features = processor(audio, sampling_rate=sr, return_tensors="pt").input_features

            # Clear the forced_decoder_ids and set language and task
            model.generation_config.forced_decoder_ids = None

            # Generate transcription with explicit arguments
            predicted_ids = model.generate(
                input_features,
                language='en',
                task='transcribe',
                use_cache=True
            )

            transcription = processor.batch_decode(predicted_ids, skip_special_tokens=True)[0]

            hypotheses.append(transcription)

            # Print progress every 10 files
            if (idx + 1) % 10 == 0:
                print(f"Processed {idx + 1}/{len(audio_paths)} files")
                print(f"Reference: {references[idx]}")
                print(f"Hypothesis: {transcription}")
                print("---")

        except Exception as e:
            print(f"Error processing file {audio_file}: {e}")
            hypotheses.append("")

    # Calculate WER
    error_rate = wer(references, hypotheses)

    # Save results
    results = {
        'references': references,
```

```
        'hypotheses': hypotheses,
        'wer': error_rate
    }

    if test==1:
        with open(f'fine_tuned_whisper_results_test-{TRAIN_NUM}.json', 'w') as f:
            json.dump(results, f, indent=2)
    elif test==0:
        with open(f'fine_tuned_whisper_results_val-{TRAIN_NUM}.json', 'w') as f:
            json.dump(results, f, indent=2)

    return error_rate, results

# %%
# Evaluate the fine-tuned model
fine_tuned_wer, fine_tuned_results = evaluate_fine_tuned_whisper(test_dataset, fine_tuned_processor,
fine_tuned_model, test=1)
print(f"Fine-tuned Whisper WER: {fine_tuned_wer:.4f}")

# %% [markdown]
# ## Compare Results
#
# Let's compare the performance of the base and fine-tuned models.

# %%
print(f"Base Whisper WER: {base_wer:.4f}")
print(f"Fine-tuned Whisper WER: {fine_tuned_wer:.4f}")
print(f"Improvement: {(base_wer - fine_tuned_wer) / base_wer * 100:.2f}%")

# %% [markdown]
# ## Additional Analysis: Validation Set Performance
#
# Let's also check how the model performed on the validation set during training.

# %%
# Evaluate the fine-tuned model on the validation set
print("Evaluating on validation set...")
val_wer, val_results = evaluate_fine_tuned_whisper(val_dataset, fine_tuned_processor, fine_tuned_model,
test=0)
print(f"Fine-tuned Whisper WER on validation set: {val_wer:.4f}")

# %% [markdown]
# ## Summary of Performance
#
# Let's create a summary of the model's performance across all datasets.

# %%
# Create performance summary
performance_summary = {
    'base_model': {
        'test_wer': base_wer
    },
    'fine_tuned_model': {
        'validation_wer': val_wer,
        'test_wer': fine_tuned_wer
    },
```

```
    'improvement': {
        'absolute': base_wer - fine_tuned_wer,
        'relative': (base_wer - fine_tuned_wer) / base_wer * 100
    }
}

# Save performance summary
with open(f'performance_summary-{TRAIN_NUM}.json', 'w') as f:
    json.dump(performance_summary, f, indent=2)

# Print summary
print("\nPerformance Summary:")
print(f"Base Model Test WER: {base_wer:.4f}")
print(f"Fine-tuned Model Validation WER: {val_wer:.4f}")
print(f"Fine-tuned Model Test WER: {fine_tuned_wer:.4f}")
print(f"Improvement: {(base_wer - fine_tuned_wer) / base_wer * 100:.2f}%")
```

F5-TTS_init.py (using the hypertuned model to run on F5TTS)

```
#!/usr/bin/env python3
"""
Simple test script for voice cloning with F5TTS and Fine-tuned Whisper.
"""

import os
import subprocess
import whisper
import argparse
from transformers import WhisperProcessor, WhisperForConditionalGeneration
import librosa
import torch
from whisper_utils import setup_whisper, transcribe_audio
import soundfile as sf  # Add this import at the top with your other imports
import pandas as pd
from pydub import AudioSegment


# Change to F5-TTS directory
os.chdir("../../F5-TTS")
FILE_PREFIX="24fa_000"
# FILE_PREFIX="37m_012"

# Hardcoded parameters
AUDIO_FILE = f"/home/ubuntu/TimeStamped-wav/combined_wavs/{FILE_PREFIX}_combined.wav"  # Using original
wave with stutters for audio ref.
# AUDIO_FILE = f"/home/ubuntu/TimeStamped-wav/original_wavs/{FILE_PREFIX}.wav"
REF_CSV = f"/home/ubuntu/TimeStamped/combined_csvs/{FILE_PREFIX}_combined.csv"
OUTPUT_FILE =
f"/home/ubuntu/Final-Project-11/Code/Audio-Samples/project_audio/{FILE_PREFIX}_cloned.wav"
TEXT_FILE = "/home/ubuntu/Final-Project-11/Code/Audio-Samples/custom_text.txt"
FINE_TUNED_MODEL_PATH = "/home/ubuntu/Final-Project-11/Code/whisper-fine-tuned-stuttering-final-4"  #
Path to your fine-tuned model

def load_text_from_file(file_path):
    try:
        with open(file_path, "r") as f:
```

```
            return f.read().strip()
    except Exception as e:
        print(f"Error reading text file: {e}")
        return None

CUSTOM_TEXT = load_text_from_file(TEXT_FILE)

def append_silence_to_audio(file_prefix, silence_duration_ms=1000):
    """Appends silence to the end of the given audio and saves to silenced_wavs."""
    input_path = f"/home/ubuntu/TimeStamped-wav/combined_wavs/{file_prefix}_combined.wav"
    output_path = f"/home/ubuntu/TimeStamped-wav/silenced_wavs/{file_prefix}_combined_silenced.wav"

    audio = AudioSegment.from_wav(input_path)
    silence = AudioSegment.silent(duration=silence_duration_ms)
    audio_with_silence = audio + silence

    os.makedirs(os.path.dirname(output_path), exist_ok=True)  # Ensure output dir exists
    audio_with_silence.export(output_path, format="wav")

    return output_path

def load_fine_tuned_whisper(model_path):
    """Load the fine-tuned Whisper model."""
    print(f"Loading fine-tuned Whisper model from {model_path}...")
    processor = WhisperProcessor.from_pretrained(model_path)
    model = WhisperForConditionalGeneration.from_pretrained(model_path)
    return processor, model

def transcribe_with_base_model(model, audio_file):
    """
    Transcribe an audio file using Whisper.

    Args:
        model: The loaded Whisper model
        audio_file: Path to the audio file

    Returns:
        The transcribed text
    """
    # Transcribe the audio file
    result = model.transcribe(audio_file)
    return result["text"]

def transcribe_with_fine_tuned_whisper(processor, model, audio_file):
    """Transcribe audio using the fine-tuned Whisper model."""
    # Load and preprocess audio
    audio, sr = librosa.load(audio_file, sr=16000)

    # Process audio for input
    input_features = processor(audio, sampling_rate=sr, return_tensors="pt").input_features

    # Clear the forced_decoder_ids and set language and task
    model.generation_config.forced_decoder_ids = None

    # Generate transcription
    predicted_ids = model.generate(
```

```
        input_features,
        language='en',
        task='transcribe',
        use_cache=True
    )

    transcription = processor.batch_decode(predicted_ids, skip_special_tokens=True)[0]
    return transcription

def test_voice_cloning(reference_audio, output_file=None, custom_text=None):
    """Test voice cloning with F5TTS using fine-tuned Whisper."""
    # Set default output path if not provided
    if not output_file:
        dir_name = os.path.dirname(os.path.abspath(reference_audio))
        base_name = os.path.splitext(os.path.basename(reference_audio))[0]
        output_file = os.path.join(dir_name, f"{base_name}_cloned.wav")

    # Load fine-tuned Whisper model
    fine_tuned_processor, fine_tuned_model = load_fine_tuned_whisper(FINE_TUNED_MODEL_PATH)

    # Step 1: Transcribe the reference audio for voice cloning using fine-tuned model
    print("Transcribing with fine-tuned Whisper model...")
    fluent_text = transcribe_with_fine_tuned_whisper(fine_tuned_processor, fine_tuned_model,
reference_audio)
    print(f"Fine-tuned transcription (fluent): {fluent_text}")

    # Step 2: For comparison, also get standard Whisper transcription
    standard_whisper = setup_whisper("base")
    standard_text = transcribe_audio(standard_whisper, reference_audio)
    print(f"Standard Whisper transcription: {standard_text}")

    # For F5TTS, we'll use the fluent (fine-tuned) transcription as reference
    # ref_text = fluent_text
    ref_text = ' ' + fluent_text.strip().rstrip('.') + '.'
    # ref_text = standard_text #this is using the base model transcription...

    # df = pd.read_csv(REF_CSV) #takes the actual stuttering csv for reference
    # ref_text = ' '.join(df['word'].astype(str)).strip().rstrip('.') + '.'

    # Determine text to generate
    # gen_text = custom_text if custom_text else ref_text #THIS NEEDS TO CHANGE BASED ON MODEL
    gen_text = ' ' + fluent_text.strip().rstrip('.') + '.' #uses the fluent transcription for generating
    if custom_text:
        print(f"Using custom text: {gen_text}")
    else:
        print(f"Using fluent transcription for generation: {gen_text}")

    # Run F5TTS for voice cloning
    print("Running F5TTS...")
    cmd = [
        "f5-tts_infer-cli",
        "--model", "F5TTS_v1_Base",  # or E2TTS_Base
        "--ref_audio", reference_audio,
        "--ref_text", ref_text,
        "--gen_text", gen_text, # Using fluent transcription
        "--output_file", output_file
```

```
        # "--speed", "0.85"
    ]

    print(f"Command: {' '.join(cmd)}")

    try:
        subprocess.run(cmd, check=True)
        print(f"Voice cloning successful! Output saved to: {output_file}")
        return output_file
    except Exception as e:
        print(f"Error during voice cloning: {e}")
        return None

def main():
    # Run the test with hardcoded parameters

    AUDIO_FILE_WITH_SILENCE = append_silence_to_audio(FILE_PREFIX)

    output_path = test_voice_cloning(
        reference_audio=AUDIO_FILE_WITH_SILENCE,
        output_file=OUTPUT_FILE
    )

    # output_path = test_voice_cloning(
    #     reference_audio=AUDIO_FILE,
    #     output_file=OUTPUT_FILE
    # )

    if output_path:
        print(f"\nTest completed successfully. Generated file: {output_path}")
    else:
        print("\nTest failed.")


if __name__ == "__main__":
    main()
```

whisper_utils.py (setup whisper and transcribe using loaded model)

```
# # Install Whisper and dependencies
# pip install openai-whisper
# pip install torch
# pip install ffmpeg-python

# # For handling audio files
# pip install pydub

import whisper
import numpy as np
import torch

def setup_whisper(model_size="base"):
    """
    Initialize the Whisper model.

    Args:
        model_size: Size of the model to use ("tiny", "base", "small", "medium", "large")
```

```
    Returns:
        The loaded Whisper model
    """
    # Load the Whisper model
    model = whisper.load_model(model_size)
    return model

def transcribe_audio(model, audio_file):
    """
    Transcribe an audio file using Whisper.

    Args:
        model: The loaded Whisper model
        audio_file: Path to the audio file

    Returns:
        The transcribed text
    """
    # Transcribe the audio file
    result = model.transcribe(audio_file)
    return result["text"]
```

combine_csvs_clips.py (combine FluencyBank Timestamped clips)

```
import os
import pandas as pd
from pydub import AudioSegment
from collections import defaultdict
import re

# Paths
csv_dir = os.path.expanduser('~/TimeStamped/csvs')
wav_dir = os.path.expanduser('~/TimeStamped-wav')
out_csv_dir = os.path.expanduser('~/TimeStamped/combined_csvs')
out_wav_dir = os.path.expanduser('~/TimeStamped-wav/combined_wavs')

# Create output directories
os.makedirs(out_csv_dir, exist_ok=True)
os.makedirs(out_wav_dir, exist_ok=True)

# Regex to extract prefix and index
pattern = re.compile(r"^([a-zA-Z0-9]+)_(\d+)\.csv$")

# Group files by prefix
groups = defaultdict(list)

for fname in os.listdir(csv_dir):
    match = pattern.match(fname)
    if match:
        prefix, idx = match.groups()
        groups[prefix].append((int(idx), fname))

# Process each group
for prefix, files in groups.items():
```

```
    files.sort()  # Sort by index
    for i in range(0, len(files), 3):
        group = files[i:i+3]
        if len(group) < 3:
            continue  # Skip incomplete groups

        combined_csv = pd.DataFrame()
        combined_audio = AudioSegment.empty()
        total_offset = 0.0
        out_base = f"{prefix}_{str(i).zfill(3)}_combined"

        for idx, fname in group:
            csv_path = os.path.join(csv_dir, fname)
            wav_name = fname.replace('.csv', '.wav')
            wav_path = os.path.join(wav_dir, wav_name)

            # Load and adjust CSV timestamps
            df = pd.read_csv(csv_path)
            df['wordstart'] += total_offset
            df['wordend'] += total_offset
            combined_csv = pd.concat([combined_csv, df], ignore_index=True)

            # Load audio
            audio = AudioSegment.from_wav(wav_path)
            combined_audio += audio
            total_offset += len(audio) / 1000.0  # in seconds

        # Save combined CSV and WAV
        combined_csv_path = os.path.join(out_csv_dir, out_base + ".csv")
        combined_wav_path = os.path.join(out_wav_dir, out_base + ".wav")

        combined_csv.to_csv(combined_csv_path, index=False)
        combined_audio.export(combined_wav_path, format="wav")

        print(f"Saved: {combined_csv_path}, {combined_wav_path}")
```

clean_csvs.py (clean combined clips)

```
import os
import pandas as pd

# Paths
input_dir = os.path.expanduser('~/TimeStamped/combined_csvs')
output_dir = os.path.expanduser('~/TimeStamped/cleaned_csvs')
os.makedirs(output_dir, exist_ok=True)

for fname in os.listdir(input_dir):
    if not fname.endswith('.csv'):
        continue

    df = pd.read_csv(os.path.join(input_dir, fname))

    # Cleaned rows go here
    cleaned_rows = []
    skip_rows = []

    pending_start = None
```

```
for idx, row in df.iterrows():
    if row['rp'] > 0 or row['pw'] > 0:
        # Stutter row - remember its start time and skip it
        if pending_start is None:
            pending_start = row['wordstart']
        skip_rows.append(idx)
        continue

    if pending_start is not None:
        # Adjust start time of fluent word
        row['wordstart'] = pending_start
        pending_start = None

    cleaned_rows.append(row)

cleaned_df = pd.DataFrame(cleaned_rows)
out_path = os.path.join(output_dir, fname)
cleaned_df.to_csv(out_path, index=False)

print(f"Saved cleaned file: {out_path}")
```