Low Latency Streaming Speech Selection

Christopher Hong

May 15, 2023

1 Introduction

The field of automatic speech recognition (ASR) has experienced great progress within the past decade [5]. Most of this progress is due to the incorporation of deep learning (DL). The success of deep learning in ASR gave rise to popular, general purpose voice assistants backed by big tech companies such as Amazon, Apple, and Google, and the improvement in ASR technology has allowed voice to become a viable medium for human computer interaction. However, voice for human computer interaction has been largely limited to big tech platforms. While big tech voice assistants have proven to be useful, they tend to be deliberately confined to their respective platforms with similar and limited capabilities. Voice for human computer interaction has the potential to become useful across applications outside of big tech products, but so far adoption seems to be limited.

Big tech companies also provide software-as-a-service (SaaS) APIs for ASR so that software developers with either limited expertise or resources can quickly integrate ASR into their applications. However, these APIs have a few considerable downsides. Most importantly from a software development standpoint, they are not transparent in their implementation, they are difficult to customize to fit the needs of the application, and there is no guarantee about the permanency of the API. These factors lead to little control over the quality of the voice recognition system.

In a previous project (CSCI2592-C, Learning with Limited Data, Fall 2022), I showed that a custom ASR implementation using open source software and deep-NN models generally outperformed SaaS ASR APIs in the specific domain of speech selection, which will be described momentarily. In this project, I optimize the custom implementation for speech selection to be suited for online streaming. From the user’s perspective, a viable online streaming ASR system should have high accuracy and low latency.

2 Background

2.1 Initial Problem Description

The premise for this project is based on implementing ASR into a workout logging iOS app. The initial goal is for the user to be able to switch the type of workout they are logging on an Apple Watch by voicing the name of the type into the watch ("bench press", "ab crunch"), without needing to pull out their phone, and searching/swiping for the particular workout type. This problem is closely related to speech command classification problem. However, the key difference between speech command classification and this problem is that, in speech command classification, the possible transcriptions is constrained to a small, static set (less than 50 commands), while in the workout app setting the potential for different types of names for workout types is potentially
unlimited, although highly biased towards a small set of words due to the domain setting. In addition, in the iOS app, the possible set of workout types is constrained to the types that the user has created, and the ASR system should not expect the user to voice a type outside that set. So, while the set is limited as in speech command classification, this set is (potentially infinitely) different between users. I will call the general problem of selecting a transcription from a finite but dynamic set of possible transcriptions the "speech selection" problem. Speech selection can be seen as lying in the middle of a continuum between the full ASR problem and speech command classification.

Speech transcription selection has a set of performance requirements closer to command classification than full ASR, in the sense that users of a speech transcription selection system expect a very high accuracy under a variety of challenging settings (different mics, noise environments) and low latency. Currently, as far as I am aware, there is no readily available software or service that specifically addresses the problem of speech transcription selection. The simplest real world implementation to address this problem is to directly use an ASR SaaS, but the APIs for these ASR SaaS platforms are limited and ill suited to the problem.

2.2 Brief Background on Contemporary ASR

Concretely, the problem of automatic speech recognition (ASR) is to infer a transcription from a speech audio signal (see figure 1 in the Appendix). Most modern production ASR pipelines today are almost fully deep learning from end-to-end. However, they still carry a few hand crafted features at the beginning and end of the pipeline. At the beginning of the pipeline, the raw speech signal is manually transformed into hand crafted features called Mel Frequency Cepstral Coefficients (MFCC). A rough diagram is outlined in figure 2 in the Appendix, the most notable parts being the use of the fast fourier transform (FFT) and mel filter bank processing, which sends the FFT signal through a set of bandpass filters that mimic the human auditory system [5]. The MFCC is then sent through a deep learning model. There are generally two classes of deep learning model outputs: CTC and Transducer. A CTC (Connectionist Temporal Classification) output is a set of probabilities of each phoneme over time. A further decoder step is needed to process this into the most likely transcription, which is usually determined in conjunction with a language model. A transducer output is the direct transcript, and does not require a language model. Figure 3 in the Appendix illustrates the two typical ASR pipelines. It is worth noting that some ASR pipelines such as wav2vec 2.0 [2] eliminate the MFCC pipeline and process the raw speech signal directly in the deep learning model, usually with convolutional layers in the front.

2.3 Streaming Description

A streaming session for speech selection is defined as follows: A user initiates a streaming session on a device with a microphone and access to the ASR server through an internet connection. When a user initiates a streaming session, the device begins recording the user’s speech through its microphone and sends equal blocks of contiguous audio data to the server. The server receives the block data, and every time it receives a block it sends back its best transcription to the user based on the audio data it has received up to that block.

3 Streaming Problem Description

In order to evaluate the quality of the streaming service, we use two measures; the accuracy of the transcription and the latency between intermediate block requests and server responses. Accuracy
is determined using the transcript response at the end of the user’s utterance of the intended phrase. Since latency is approximately deterministic if disregarding the internet connection at the user endpoint, latency is measured using a single streaming session, and the latency measures for each block request is graphed versus the timepoint of the block request relative to the start of the streaming session.

4 Preliminary Methods

As described above, I compared the performance of ASR SaaS APIs to a custom ASR solution. The custom solution is to use an existing end-to-end audio to phoneme-level CTC encoder model \[3\] and perform a highly customized greedy search at the phoneme level to select the transcription with the highest probability among the valid set. The greedy search algorithm uses a trie to prune the search space, and this trie is initialized dynamically by the valid transcription set. This algorithm will be hereafter referred to as CTC Speech Selection Greedy Search.

The CTC encoder model we use is CTC-Conformer \[4\], adopted from the open source NLP platform Nvidia NeMo \[1\]. CTC-Conformer uses both transformer and convolution components to achieve high transcription accuracy on audio data. Figure 6 in the Appendix illustrates the CTC-Conformer architecture.

4.1 CTC Speech Selection Greedy Search

The simple idea of CTC speech selection greedy search is to take the greedy search algorithm from \[3\] and modify it such that the search space is restricted to the valid transcription set. We closely follow the terminology laid out in \[3\].

Suppose there exists an input sequence \(x\) of length \(T\) and a deep learning model with parameters \(\omega, N_\omega\). Let \(y = N_\omega(x)\) be the sequence of model outputs and denote \(y^t_k\) the activation of output unit \(k\) at time \(t\). then \(y^t_k\) is interpreted as the probability of observing label \(k\) at time \(t\), which defines a distribution over the set \(L^T\) of length \(T\) sequences over the phoneme alphabet \(L' = L \cup \text{blank}\).

Note the addition of the \(\text{blank}\) token to the phoneme alphabet \(L\):

\[
p(\pi|x) = \sum_{t=1}^{T} y^t_{\pi_t}, \forall \pi \in L^T
\]

Please note that we will be using the subtle distinction between the phoneme alphabet with and without the \(\text{blank}\) token \(L'\) and \(L\). Now suppose that we have a transcription set \(T\) composed of transcriptions \(T = \{\tau_1, ..., \tau_n\}\). Each transcription is associated through a one to one function \(z : T \rightarrow L^{\leq T}\) such that if \(z(\tau_i) = \pi_i \in L^{\leq T}\), then the probability of transcription given phoneme sequence \(\pi \in L^T\), \(p(\pi_i | \pi_i) = 1\) and \(p(\pi_i | \pi) = 0, \pi \neq \pi_i\). The function, or many-to-one map \(c : L^T \rightarrow L^{\leq T}\) describes the set of possible phoneme labeling sequences. The function \(c\) removes all blanks and repeated labels. So for example, if the \(\text{blank}\) token is denoted “\_”, then \(c(“\_ccaa–t”) = “cat”\) (see \[3\] for more details). To define the conditional probability of a given labeling \(l \in L^{\leq T}\) we have:

\[
p(l|x) = \sum_{\pi \in c^{-1}(l)} p(\pi|x)
\]

Independent of the transcription set, we would like the most probable labeling

\[
h(x) = \arg \max_{l \in L^{\leq T}} p(l|x)
\]
But this is generally not tractable so we approximate with the most probable path

\[ h(\mathbf{x}) \approx c(\pi^*) \]

where \( \pi^* = \arg \max_{\pi \in \mathcal{N}} p(\pi | \mathbf{x}) \)

Now we incorporate the transcription set constraint. We would like the most probable transcription

\[ h'(\mathbf{x}) = \arg \max_{\tau \in \mathcal{T}} p(\tau | \mathbf{x}) \]
\[ = \arg \max_{\tau \in \mathcal{T}} p(\tau | \mathcal{I}) p(\mathcal{I} | \mathbf{x}) \]

However as explained above this is intractable so we approximate with

\[ h'(\mathbf{x}) \approx \arg \max_{\tau \in \mathcal{T}} p(\tau | \pi) p(\pi | \mathbf{x}) \]

We can then use \( p(\tau | \pi) \) to prune the greedy search of the phoneme sequence space.

5 Initial Experiments

5.1 Accuracy Experiment: Setup Description

In the previous project, we evaluate the speech selection transcription accuracy on the ASR SaaS APIs vs the custom solution. We use Apple’s SFSpeech API and Google’s Speech-To-Text API. Both SaaS services offer an offline and online streaming API. We evaluate the transcription accuracy for both for Apple’s service. During evaluation on both APIs, we send the speech audio to the SaaS service. The service returns a set of possible transcription alternatives. We normalize the transcriptions by setting all the characters to lower case and removing punctuation. Finally we compare each transcription alternative to the ground truth for a match. If a match occurs within the first 10 most likely alternatives, we consider the transcription to be a success. Otherwise it is a failure. Accuracy is the ratio of successes to the total number of samples. Sometimes the APIs will throw a “no speech detected” error. This case we also consider a failure.

5.2 Datasets

We evaluate the two SaaS baselines and the custom model using three datasets. The first dataset is the Google Speech Commands dataset. This dataset was originally intended for evaluating speech classification. However, in this work we use it to evaluate speech selection using the set of possible commands as the transcription set. This dataset contains 6798 samples among 30 classes that are approximately uniformly distributed among the commands and is by far the largest dataset in this experiment.

The second dataset is similar to the Google Speech Commands dataset, except this dataset is a personal set of recordings, with the commands set being replaced by 19 workout types phrases motivated by the workout app among. I recorded 3 samples per workout type using a custom built iOS app. The phrases are listed in table 3 in the Appendix.

The third dataset is similar to the second dataset except the phrases are generated by Google’s Text to Speech SaaS API. For each phrase, 9 different state-of-the-art voices were used with two different realistic voice speed settings for a total of 18 samples per workout type (same 19 workout types).
5.3 Accuracy Experiment: Results

The following table shows the accuracy and precision results for the baselines/models evaluated on the datasets. Note that for each baseline the first row and second row indicate the non-streaming and streaming APIs.

<table>
<thead>
<tr>
<th>Model</th>
<th>Google Speech Commands</th>
<th>Workout Types Self Recorded</th>
<th>Workout Types Synthesized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple SFSpeech</td>
<td>.946</td>
<td>.544</td>
<td>.857</td>
</tr>
<tr>
<td>&quot; (streaming)</td>
<td>.709</td>
<td>.500</td>
<td>.719</td>
</tr>
<tr>
<td>Google TTS</td>
<td>.773</td>
<td>.632</td>
<td>.941</td>
</tr>
<tr>
<td>Custom</td>
<td>.919</td>
<td>.860</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 1: Accuracy

We note that the custom row is updated from the table in the presentation to reflect a bug fix in the decoder.

5.4 Preliminary Latency Analysis

In order to evaluate the bottleneck of streaming speech selection, we measure the latency between each request and response during a streaming session as defined in section 2.3, where the streaming service is implemented in a naive manner using components that are intended for offline use and not optimized for streaming. The naive implementation is as follows: The user initiates a streaming session. As the user sends blocks of audio data, the server keeps a record of the complete audio since the beginning of the session by concatenating the blocks. For each block request, the complete audio received up to the current block request is first passed to the CTC-Encoder that is trained and infers using unlimited context in its attention heads. The CTC encodings for the complete audio are then passed to the speech selection decoder implemented in a single-threaded manner. The transcript obtained by the speech selection decoder is then sent to the user.

The latency graph for a simulated single streaming session the naive implementation is shown in figure 4 in the Appendix. The simulation assumes zero communication latency, so it is a lower bound on real world latency.

6 New Methods

6.1 Parallel and Cached CTC Decoder Implementation

Figure 4 shows that the main bottleneck is in the speech selection decoder. The speech selection decoder is a dynamic programming algorithm that is highly parallelizeable. Therefore, for this project, we implement a multithreaded version of the decoder that spreads the dynamic programming work across multiple threads. No change is made to the basic algorithm, only its implementation in code. To see why the decoder is parallelizeable, we diagram an example case of the decoder in action. Suppose that, in one pass of the speech selection decoder, we only have one transcript option of "CAT". Figure 8 in the Appendix diagrams the possible paths that transcribe to "CAT".

The horizontal axis represents time and the vertical axis represents the phonemes of "CAT" in order from top to bottom with the BLANK token replicated in between. The arrows show the valid transitions, and a node $n_{it}$ at phoneme index $i$, time $t$ then represents the current phoneme in all the possible paths that transcribe to "CAT" where the phoneme index at time $t$ is $i$. Each
A node represents the maximum log probability of some path that ends at that node, where the log probability of a path of length \( l \), \( n_{i_1,t_1}, n_{i_2,t_2}, \ldots, n_{i_l,t_l} \), is the sum of the log probabilities outputted by the CTC encoder at those positions \( \sum_{j=1}^{l} p_{i_j,t_j} \). Suppose the maximum log probability for the node \( n_{it} \) is \( m_{it} \) and \( S \) is the set of nodes with an edge to \( n_{it} \). Then we can calculate by induction

\[
m_{it} = \max_{n_{jk} \in S} m_{jk} + p_{it}
\]

Thus for each time step \( t \) from 0, ..., \( T \), \( m_{it} \) can be calculated for all \( i \) in parallel using the inductive relationship above.

In addition, the interface for the decoder can be optimized for streaming by storing the values of \( m_{it} \) calculated using the blocks of audio already received, suppose up to time \( T \), and new values of \( m_{it} \) where \( t > T \) can be calculated using the already computed values of \( m_{iT} \). Thus we remove duplicate computation in the streaming setting compared to the naive implementation.

### 6.2 Cached CTC Encoder

Another method to optimize the custom ASR system for streaming is to optimize the encoder. Specifically, we can cache computations made in the attention heads and convolution kernels in the CTC-Conformer model so that duplicate computations are not made during a streaming session, and also we can limit the context length of the attention heads to minimize the amount of computation needed per audio block. One way to limit the context length is to divide the time steps into fixed size blocks, and limiting the causal context for a particular attention head by a fixed number of blocks relative to the attention head and the lookahead context to only within the block of the attention head. An example of the context mask for the attention heads and the dependent attention values for the value at \( f_{10} \) is illustrated in figure 7 in the Appendix.

### 7 New Experiments

#### 7.1 Latency Analysis for Optimized Decoding

The multithreaded implementation of the dynamic programming algorithm is implemented in C++. The latency simulation of a streaming session using the unlimited context encoder and multithreaded decoder with the cache interface is shown in figure 5 in the Appendix. The simulation is run on a 16-core, 32-thread Ryzen 5950x CPU (the simulation for the naive implementation is run on the same platform), and all 32 threads are utilized in the multithreaded implementation. We see that the multithreaded implementation with the cache interface dramatically improves the decoding time by as much as 1/6 as well as overall latency. The load balancing in the multithreaded implementation is not fully optimized, so there exists room for further improvement. Overall, the optimized decoder mostly relieves the decoder bottleneck, and so more attention can be made to the encoder should the overall latency still be unacceptable or for longer streaming sessions.

#### 7.2 Accuracy of Block Context-Limited Streaming Encoder

We also analyze the accuracy of a block-cached CTC Encoder. Unfortunately, while a pretrained CTC-Conformer model is available where the attention heads have unlimited context (and is therefore suited for offline use), Nvidia does not provide a pretrained model where the context is limited. However, Nvidia does provide the model and training configuration for this case, so we train a limited context CTC-Conformer model using NeMo on the LibriSpeech dataset [6], which has 1000
hours of transcribed audiobook speech data. This takes around 6 hours on a machine with 6x A100 Nvidia GPUs to achieve 50 epochs on the dataset, which is a reasonable training time.

We use the block method described above to mask the context for the attention heads. The block size is 34 time samples. The preprocessing uses 10ms windows for FFT and the model uses 4x convolutional downsampling, so each time sample represents 40ms. Therefore, each block represents 40ms * 34 = 1.36s. Attention heads can only lookahead within a block but can look back 3 blocks. The CTC-Conformer model has 18 layers, so each head can look back at most 1.36s * 3 * 18 = 1m13s.

The table below shows the accuracies achieved by running inference on the block-cached CTC encoder on the same datasets as the original accuracy experiment. The accuracies of the original custom model is copied here for convenience.

<table>
<thead>
<tr>
<th></th>
<th>Google Speech Commands</th>
<th>Workout Types Self Recorded</th>
<th>Workout Types Synthesized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>.919</td>
<td>.860</td>
<td>1.0</td>
</tr>
<tr>
<td>New Encoder</td>
<td>.867</td>
<td>.895</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 2: Accuracy of Original Encoder vs. Context-Limited Streaming Encoder

The table suggests that there may be some loss of accuracy in limiting the context. However, we note that we do not use the cached streaming interface of the encoder developed by Nvidia, which we observed further reduces accuracy, likely due to a poor implementation of the caching.

We note that the use case described in the premise involves relatively short phrases that should be uttered within a few seconds, so there may be little need to use an optimized encoder based on the latency analysis. However, the cached and context-limited encoder is of considerable interest for longer online speech transcription.

8 Conclusion (Abstract) and Future Directions

We analyze the latency of a naive implementation of our custom speech selection model for the streaming scenario, and find that the main bottleneck is the decoder. We then experiment with streaming optimizations for both the encoder and decoder. We find that optimizations for the decoder which do not change the nature of the decoding algorithm significantly improve the latency of the decoder. We also find that optimizations for the encoder results in potential significant loss of accuracy in speech selection transcription accuracy, and conclude that for our use case it may not be worth the accuracy trade off to use the context-limited encoder, although for longer streaming sessions it could be worth the tradeoff.

Potential future directions for speech selection research include experimenting with transducer models instead of CTC models, as well as comparing greedy CTC decoding with full CTC decoding, since greedy CTC decoding using the best path is purportedly an approximation to the conditional probability involving the sum of all paths leading to the same transcript, but the claim is not supported with evidence or analysis in [3].
9 Appendix

Figure 1: General ASR problem: raw speech signal to transcription

Figure 2: MFCC pipeline
Figure 3: ASR pipeline [5]
Figure 4: Latency Graph for Naive Implementation

Figure 5: Latency Graph for Cached and Multithreaded Implementation
Figure 6: CTC-Conformer Architecture

Figure 7: Block Context-Limited CTC-Conformer Illustration
**Figure 8: CTC Paths for “CAT”**

<table>
<thead>
<tr>
<th>bench press</th>
<th>pulley row</th>
<th>decline crunches</th>
<th>assisted pullups</th>
<th>barbell squats</th>
<th>dumbbell row</th>
<th>ab crunch machine</th>
<th>dumbbell bicep curl</th>
<th>barbell bench press</th>
<th>dumbbell press</th>
<th>tricep extension</th>
<th>pushups</th>
<th>pulldown</th>
<th>shoulder fly</th>
<th>chest press machine</th>
<th>bicycles</th>
<th>dumbbell curl</th>
<th>dumbbell good mornings</th>
<th>dips</th>
</tr>
</thead>
</table>

**Table 3: List of phrases for workout types datasets**

**References**

[1] URL [https://github.com/NVIDIA/NeMo](https://github.com/NVIDIA/NeMo)


