Honkling: In-Browser Personalization for Ubiquitous Keyword Spotting

by

Jaejun Lee

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Used for simple voice commands and wake-word detection, keyword spotting (KWS) is the task of detecting pre-determined keywords in a stream of utterances. A common implementation of KWS involves transmitting audio samples over the network and detecting target keywords in the cloud with neural networks because on-device application development presents compatibility issues with various edge devices and provides limited supports for deep learning. Unfortunately, such an architecture can lead to unpleasant user experiences because network latency is not deterministic. Furthermore, the client-server architecture raises privacy concerns because users lose control over the audio data once it leaves the edge device. In this thesis, I present Honkling, a novel, JavaScript-based KWS system. Unlike previous KWS systems, Honkling operates purely on the client-side—Honkling is decentralized and serverless. Given that it is implemented in JavaScript, Honkling can be deployed directly in the browser, achieving higher compatibility and efficiency than the existing client-server architecture. From a comprehensive efficiency evaluation on desktops, laptops, and mobile devices, it is found that in-browser keyword detection only takes 0.5 seconds and achieves a high accuracy of 94% on the Google Speech Commands dataset.

From an empirical study, the accuracy of Honkling is found to be inconsistent in practice due to different accents. To ensure high detection accuracy for every user, I explore fine-tuning the trained model with user personalized recordings. From my thorough experiments, it is found that such a process can increase the absolute accuracy up to 10% with only five recordings per keyword. Furthermore, the study shows that in-browser fine-tuning only takes eight seconds in the presence of hardware acceleration.

Acknowledgements

First of all, I would like to thank Prof. Jimmy Lin. I truly believe that only through his supervision, I could accomplish this much amount of work in such a short period of time. As I worked closely with Jimmy, I have become a better researcher. I now know how to set up a meaningful hypothesis and conduct rigorous experiments which allow me to derive logical conclusions. Outside of research, he changed the way I balance my work and personal life. Specifically, I learned how to make my time most meaningful regardless of what I do. Overall, my time with Jimmy has definitely been one of the most valuable experience in my life. I really appreciate him for the opportunities he has provided me and wish he also enjoyed the time he spent with me.

The second person I would like to thank is Raphael Tang. He was, still is, and will continue to be my mentor, teacher, colleague and friend. I am really thankful that I have him along side me working on the same project. He made me realize the importance of proper guidance, which is now one of the most important criteria when I consider future career opportunities. Once I graduate, I am going to miss him a lot. I have one advice for him though, I hope he takes care of himself a little better because he has another three or four years of Ph.D ahead of him.

Next, I would like to thank Prof. Khuzaima Daudjee, who has watched my growth as a researcher closely throughout my Master's degree. Other than Jimmy, he is the professor who provided the most lessons on how to become successful in graduate school. I really appreciate his counsel and encouragement.

I thank Prof. Jeff Avery for the numerous words of wisdom he gave throughout my time at Waterloo. Though we met as instructor and student, I always found him as someone who I could easily talk to. When I was stressed out, I visited him in his office to take a little break from my overbearing work. I am really happy to have him as one of my thesis reviewers and be there to share in my accomplishments.

I also had a close relationship with Prof. Ken Salem. Ever since I worked with him as an undergraduate research assistant, he continuously provided me with pointers on how to make my time as a graduate student most fruitful. I would like to take a chance to thank him for his kind words.

My feelings for my girlfriend, Erica, is grateful. Throughout my studies, there have been times where I put my work above her. Though she didn't complain, I knew that it was hard on her. I am sorry for those heart-breaking moments and thankful for her unwavering support.

I have met amazing people throughout my journey as a graduate student.

First of all, my time in Data Systems Group (DSG) wouldn't be this memorable without Jayden, Peng, Ashutosh, Zeynep, Ryan, Achyudh, Leo, Mavis, and the rest of the DSG group. Coming from a wide variety of backgrounds, we have inspired each other through intense discussions, filling the knowledge gaps that each one of us had. It was an honor to work with them and I am going to miss my time in DSG.

I thank Nam, who always had to bare with my complaints and stupid math questions. I also appreciate Max for taking me out for great food whenever I am stressed out. Shoutout to Kevin for his careful editing and proofreading. I am so grateful to know him and to have him in my life. Last but not least, my little brother Jay, for being around and supporting me in a multitude of ways.

Special thanks to Youngbin, who introduced me to Jimmy. He was the first one I talked to when I had any concerns about my career. As my senior, who has went through the same difficulties, he shared his know-hows and guided me to the right direction. I would like to take this opportunity to thank Youngbin from the bottom my heart.

Lastly, I want to thank my parents, who have sacrificed much for my education. I still remember the time when my family moved to Canada. Despite being only fourteen, I knew that it wasn't an easy decision for them. Though this thesis only has my name, I would not be writing this thesis today without their steadfast love and endless support.

Dedication

This thesis is dedicated to my parents for their love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ranging from simple voice commands to virtual assistants, automatic speech recognition (ASR) systems are becoming increasingly prevalent in our daily lives; for example, they enable us to dial contacts while driving or search for a recipe while cooking. Due to the complexity of speech recognition, current ASR systems transmit transcribed audio to perform speech recognition in the cloud. Unfortunately, such a client-server architecture has a major drawback. Since network latency is an unpredictable measure, transmitting an audio block makes the response time of the system non-deterministic [54, 45]. Also, the privacy and security implications are significant: servers may be accessed by other people, authorized or not [59]. Thus, ASR systems often use keyword spotting (KWS) on edge devices to capture the user's intention for interaction and transmits only the relevant speech and not all incoming audio [39].

Concretely, KWS is the task of detecting pre-trained keywords in a stream of utterances. In the above use case, the three most important measures for KWS are accuracy, latency and power consumption. First, accuracy is desired because the server-side ASR process triggered by incorrect wake-word detection can lead to unpleasant user experiences and unnecessary resource usage. Next, latency is often evaluated because users expect real-time response from a speech-based interaction. Power consumption is important because KWS systems must be highly available on low power, performance-limited devices. Therefore, state-of-the-art KWS systems use neural networks [49, 1, 57, 9] with a small number of parameters and floating operations [9, 58].

Unfortunately, deployment of on-device KWS systems is challenging because each platform requires platform-specific development. Consider the two dominant operating systems (OS) for mobile devices: iOS by Apple and Android by Google. iOS application involves

Swift and Objective-C while Android applications are implemented using Java and Kotlin. In the domain of personal computers, there exist Windows, MacOS, and many more, limiting the applications in different ways. Therefore, developers need to put in extra efforts to provide a consistent KWS interface across devices; this hinders the rapid deployment of a unified KWS framework.

Enter JavaScript. JavaScript is a scripting language for web application development. Unlike the issue of platform specificity in the aforementioned domains, many developers have further taken the universality of JavaScript to extremes and enabled development of desktop applications, backend services, and mobile applications with Electron, Node.js, and React, respectively—JavaScript allows the efficient development of cross-platform applications as it fulfills the philosophy of "write once, run anywhere".

Exploiting the universality of JavaScript, I present Honkling, a JavaScript-based KWS system. Implementing the residual networks introduced by Tang and Lin [57] using TensorFlow.js,[1] target keywords can be detected purely in browser; voice-enabled user interfaces on mobile and desktop applications come for free. Furthermore, as it has a serverless and decentralized architecture, Honkling does not suffer from variable network latency and keeps user data secured on the client side.

When I evaluate performance of Honkling on various devices, I have consistently observed a high accuracy of 94% on the test set. However, it is found that such high accuracy is not guaranteed in practice due to various accents that are scarcely represented in the training dataset—when the same model is evaluated on the user recordings, 80.0% accuracy is observed from a Chinese user and 77.6% is observed from a British user. The accuracy degradation in the presence of underrepresented accents is a common issue in speech recognition and numerous techniques have been introduced to minimize the accuracy drop [56, 18, 21, 24, 20, 42]. However, they are mostly specific to the underlying system and little work is known for KWS.

1.1 Contributions

The main contribution of this work is Honkling,[2] a JavaScript-based KWS system that is accurate, real-time, cross-platform, and serverless. This work takes one more step toward accent adaptation and studies how to preserve the high accuracy for users of different accents. Overall, the contributions of this thesis can be summarized as follows.

[1] https://js.tensorflow.org
[2] http://honkling.ai

- By comprehensively evaluating the inference latency of Honkling, I empirically show that JavaScript is capable of providing real-time keyword detection, refuting the belief that browsers are not suitable for computationally expensive tasks.
- I extend Honkling to different domains of web applications, desktop software, and smart home devices and demonstrate the flexibility of JavaScript-based KWS system.
- I present a study on the effectiveness of fine-tuning KWS model with user recordings for accent adaptation—it is found that fine-tuning can increase absolute accuracy up to 10% only with five recordings per keyword. Bringing the findings to fruition, I implement accent adaptation on Honkling with in-browser fine-tuning. My study shows that Honkling can be personalized within 8 seconds with hardware acceleration.

1.2 Thesis Organization

I start with background information. The three topics I discuss in Chapter 2 are: the history of KWS, application development with JavaScript, and accent adaptation in the field of speech recognition. In the next chapter, Chapter 3, I describe how Honkling is implemented step by step: data preparation, model architecture, and challenges I have faced throughout the development. Then, I conduct a set of efficiency evaluation with Honkling to understand the feasibility of in-browser keyword detection. In the same chapter, I include the three applications that support hand-free interaction through Honkling, which demonstrate the flexibility of JavaScript-based system.

Chapter 4 talks about accent adaptation for KWS. First of all, I evaluate the accuracy of honkling for users with an accent that the training data do not capture. Next, I study the effectiveness of fine-tuning for accent adaptation and implement an in-browser fine-tuning process on Honkling using the best hyperparameter setting found. Chapter 4 closes with a set of inference latency evaluations on in-browser fine-tuning.

In Chapter 5, I conclude my thesis with future work.

Chapter 2

Related Work and Background

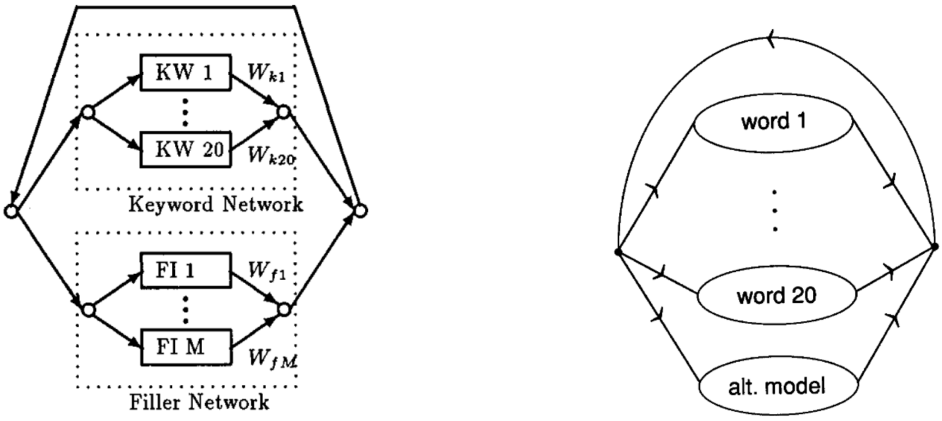
2.1 Keyword Spotting

Used for simple voice commands and wake-word detection, KWS refers to the task of detecting a set of keywords from an audio stream. Traditionally, speech transcription and keyword-filler hidden Markov model (HMM) have been the two popular algorithms for KWS. However, as neural networks show their effectiveness in speech recognition, many researchers have applied neural networks to KWS and achieve state-of-the-art accuracy.

2.1.1 Speech Transcription

Dynamic-Time-Warping-based Speech Transcription

The first approach taken for KWS involves speech transcription; the input audio is transcribed and target words are searched in the transcription. In order to achieve highly accurate speech transcription, the audio stream is broken down into blocks of same length, and each block is queried for the most similar annotated audio sample. To find the most similar sample, dynamic time warping (DTW) is used, which measures the similarity between two sequential data [60, 50]. After constructing the transcription, KWS becomes a simple search problem.



(a) Rose and Paul [48]

(b) Rohlicek et al. [47]

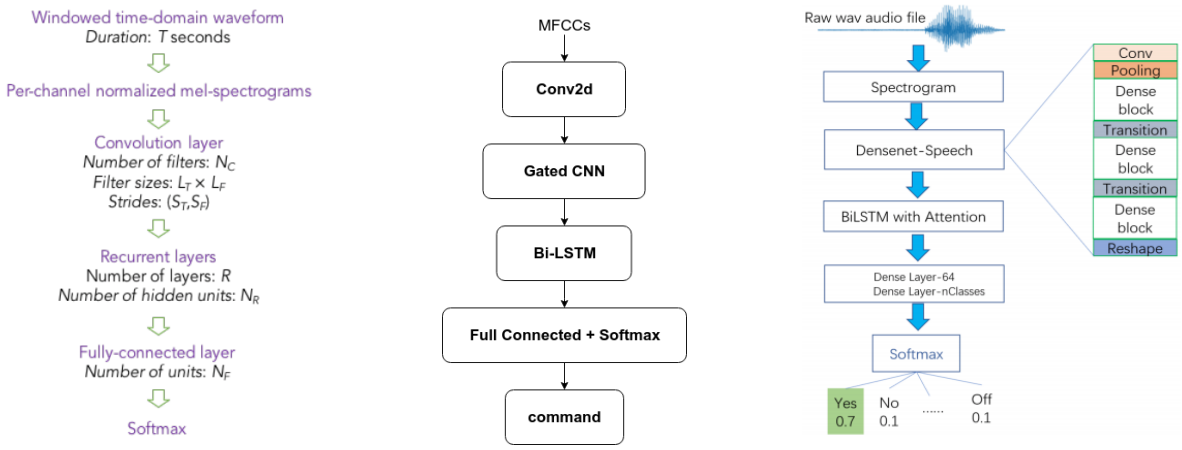
Figure 2.1: An illustration of keyword-filler HMM.

HMM-based Speech Transcription

From the 1970s to the 1990s, HMM-based speech transcription has been the most popular technique for KWS. This technique builds a statistical state machine which captures audio features of each word. There are two types of HMM-based speech transcription differing the representation of each state: word-based [64, 51] and phonetic-based [32, 31]. Once HMM is trained, a sequence of hidden states that describes the input audio stream best can be constructed using Viterbi algorithm [10]. The transcription is simply a concatenation of the states and the target keywords are searched in the transcription [63, 36, 37, 40].

2.1.2 Keyword-Filler HMM

Even though transcribing an entire audio stream leads to a successful keyword detection, it transcribes unnecessary words; speech-transcription-based KWS is computationally inefficient. The following algorithm, called keyword-filler HMM, overcomes this issue. In this type of HMM, there exists a state for each target keyword. In addition to keyword states, some models have extra states called filler states which represent non-target words (see Figure 2.1). Given a set of speech data where start and end times of each word are known, training keyword-filler HMM results in capturing different audio features for each word. Again, the best sequence is constructed using Viterbi algorithm and the sequence is checked for whether it contains the target keyword or not [48, 47, 53].



(a) Arik et al. [1] (b) Wang et al. [61] (c) Zeng and Xiao [65]

Figure 2.2: Different combinations of CNN and RNN for KWS.

2.1.3 Neural Networks

Chen et al. [3] propose the first neural networks for KWS and demonstrate that deep neural networks (DNN) are more effective for KWS than the standard HMM. Since this finding, different types of DNN have been applied for KWS.

Convolutional Neural Networks

Inspired by the connectivity pattern of neurons, convolutional neural networks (CNN) are designed to extract relevant features better than vanilla DNN [27]. In 2015, Sainath and Parada [49] introduce the first CNN-based KWS. Tang and Lin [57] then integrate residual connections and achieves higher accuracy by handling the vanishing gradient issue better. In the domain of image classification, the idea of residual connection has been taken to extremes; DenseNet is a network architecture where every layer is connected to each other by residual connection [19]. It is found that such a high degree of information sharing is also beneficial for KWS [65, 7].

Recurrent Neural Networks

Though CNN have increased the accuracy of KWS systems remarkably, some researchers have argued that recurrent neural networks (RNN) are more suitable for KWS because

they are designed to operate on sequential data and the input for KWS is an audio signal. Consequently, RNNs of different architectures—vanilla RNN, long short-term memory (LSTM), and gated recurrent unit (GRU)—have been explored for KWS [8, 4, 55, 66]. It is found that all of the RNN variations are better at detecting target keywords than vanilla DNN implementations.

Combination of CNN and RNN

As CNN and RNN have both shown their effectiveness in KWS, the combination of the two network architectures has been studied to extract meaningful feature effectively and to utilize the temporal aspects of the speech data [61, 65, 23, 1, 52]. As listed in Figure 2.2, the architectures presented in these papers are all very similar; first, extract audio features using CNN and improve detection accuracy using RNN. Though combining CNN and RNN often guarantees higher accuracy than using one of the two, such an architecture often ignores constraints on resource usage as it consists of a large number of parameters.

2.2 Application Development with JavaScript

JavaScript is a programming language developed to modify the state of web components written in HTML and CSS. Therefore, browsers are designed to behave in a uniform way for a JavaScript instruction. In other words, web applications written in JavaScript are guaranteed to provide the same functionality across browsers. This also means that the web applications are ubiquitous; they are available on any device that runs a browser.

2.2.1 Universality of JavaScript

Cross-platform functionality is fairly difficult to achieve because there are different OSes for edge devices. For desktops, there are Windows, MacOS, and Ubuntu. The two dominant OSes for mobile are Android and iOS. Each one of these OSes provides a unique environment to their application and therefore, developers often duplicate their code to support multiple platforms.

Fortunately, the development of several JavaScript frameworks provides a workaround for the platform-specificity issue [17, 5, 43]. First of all, Node.js permits the development

of server-side applications in JavaScript. Before the advent of Node.js, server-side applications are often developed in Java, C#, or PHP.[1] Node.js has enabled front-end developers to exploit their JavaScript knowledge for server-side development, blurring the boundary between front-end and back-end developer. Similarly, React[2] and Apache Cordova[3] support the development of mobile application in JavaScript; code written in HTML, CSS and JavaScript can be compiled into platform-specific language, reducing the duplicated codebases. There is a similar library for desktop development: Electron.[4] With this library, a web application can be turned into applications for Windows, MacOS, and Ubuntu. Many applications we use in our daily lives are developed with Electron—Slack, Atom, WhatsApp, Skype, as well as Discord.

2.2.2 Serverless Architecture

Standard implementation of a web application involves client-server architecture. Client applications focus on user interaction, interpreting the user's intention and responding to it in an appropriate manner. Server applications, on the other hand, are responsible for keeping state information of an individual application and executing computationally expensive tasks that edge devices lack the resources to support. However, browsers are now capable of exploiting various accelerations available from the underlying hardware and are found to be much more powerful than what we used to believe.

Previously, Gebaly and Lin [13] explore a JavaScript-based analytical relational database management system (RDBMS) that runs completely inside a browser with no external dependencies; their RDBMS implementation demonstrates comparable performance to MonetDB, running natively on the same machines. The authors then collaborate with Golab and further explore in-browser data cube exploration [14]. Similar to these work, Chen and Xu [2] implement a browser-based online game purely in JavaScript. In this work, they use Three.js[5] for rendering 3D graphics and WebSocket for exchanging messages among players to minimize the dependencies between client and server.

In the domain of deep learning, Liang et al. [33] have deployed a CNN model directly in the browser and evaluate the feasibility of in-browser natural language processing; by conducting a comprehensive evaluation on a wide range of devices including desktops,

[1] https://nodejs.org
[2] https://reactjs.org
[3] https://cordova.apache.org
[4] https://electronjs.org
[5] https://threejs.org

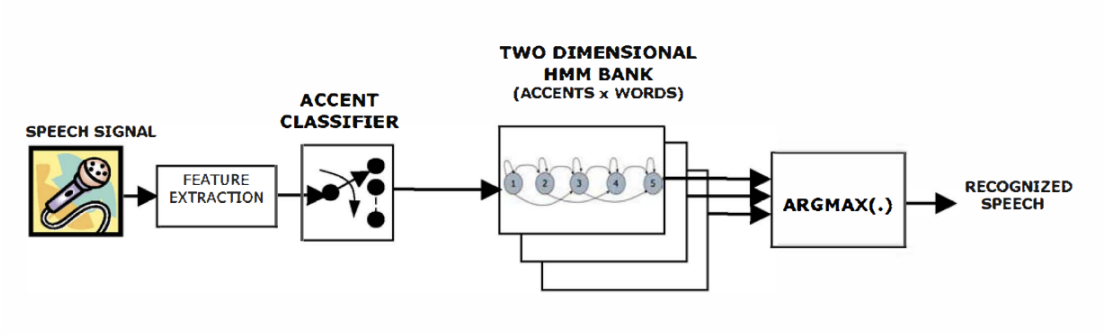


Figure 2.3: Accent-clustering-based accent adaptation [56].

laptops, as well as mobile devices, the authors show that in-browser sentence classification based on sentiment is feasible with hardware acceleration.

Since state-of-the-art techniques for KWS focus on light-weighted neural networks, the natural direction of research is to explore in-browser KWS with a study on the inference latency and power consumption.

2.3 Accent Adaptation for ASR

The recent advances in technology has brought ASR systems into reality, redefining the way we interact with electronic devices around us. However, current ASR systems fail to provide consistent accuracy for every individual due to the variety in user accents; Huang et al. [18] show that different accents can degrade the accuracy by 15∼30%. Unfortunately, building an accent-robust ASR system is difficult because there are billions of users, while existing systems have limited capability in distinguishing different accents.

2.3.1 Accent-Clustering-based Accent Adaptation

Figure 2.3 illustrates a traditional solution for accent adaptation; identifying similar accents prior to ASR and building accent-specific ASR models [56, 18, 21, 24]. Throughout the initial process, accents from the same age group or gender are often grouped together [6, 44]. This reduces the variance in accents that the following model needs to deal with, increasing the accuracy of each model. Tanabian and Goubran [56] have found that this process can improve the accuracy by 24%. However, the drawback of this solution is clear: the

performance depends on the quality of accent clusters and training multiple models is unavoidable.

2.3.2 Neural-Networks-based Accent Adaptation

For ASR systems based on neural networks, Huang et al. [20] propose a network architecture with multiple output layers; each output layer corresponds to a different group of accents. It is found that such a network architecture helps in building an accent-robust system and reduces the error rate by 22%. Another neural-network-based accent adaptation technique is introduced by Najafian et al. [42]; the key idea of technique is to fine-tune the trained model using audio of a similar voice traits. Though they observe a 12% reduction in the error rate, they train their model for hours to obtain a noticeable gain.

Surprisingly, I am not aware of any previous study on accent adaptation for KWS. Therefore, I study how sensitive KWS is to different accents and demonstrate the effectiveness of fine-tuning with user recordings.

Chapter 3

Honkling

In this chapter, I present Honkling, a novel, JavaScript-based KWS system that runs in the browser without any server-side support [29]. By conducting a comprehensive efficiency evaluation, I demonstrate that Honkling can support real-time keyword detection on desktops, laptops, tablets, as well as mobile devices. Exploiting the flexibility of JavaScript, I then extend Honkling to different web applications: desktop software, mobile applications, and smart home controllers.

3.1 Data and Implementation

3.1.1 Google Speech Commands Datasets

For training a model for KWS, I use a speech commands dataset released by Google [62]. The dataset contains 65,000 one-second long utterances of 30 short words recorded by thousands people of different genders and ages. The dataset also consists of various background noise samples such as pink noise, white noise, and human-made sounds.

Most experiments conducted on the Google Speech Commands (GSC) dataset aim to distinguish the following 12 classes: "yes", "no", "up", "down", "left", "right", "on", "off", "stop", "go", unknown, or silence [1, 57]. The data is split into three; 80% for training, 10% for validation, and the other 10% for test. This split results in roughly $22,000$ samples for training and $2,700$ samples for both validation and testing. To be consistent with prior work, models for Honkling are trained with the same training and validation set.

type	m	r	n	Par.	Mult.
conv	3	3	45	405	1.80M
avg-pool	2	2	45	-	45K
res $\times$ 3	3	3	45	109K	28M
avg-pool	-	-	45	-	45
softmax	-	-	12	540	540
Total	-	-	-	110K	30M

type	m	r	n	Par.	Mult.
conv	3	3	19	171	643K
avg-pool	4	3	19	-	6.18K
res $\times$ 3	3	3	19	19.5K	5.0M
avg-pool	-	-	19	-	19
softmax	-	-	12	228	228
Total	-	-	-	19.9K	5.7M

Table 3.1: Parameters used for `res8` (left) and `res8-narrow` (right).

In order to train models that are robust against noise, 80% of the training audios are randomly selected and augmented with noise. For each target audio, a noise sample is randomly chosen from the background noise set. I first reduce the volume of the noise by factor of ten and mix it with the target audio. The last step is shifting; the augmented audio is shifted randomly by Y milliseconds, where $Y \sim \textsc{Uniform}[-100, 100]$.

3.1.2 Feature Extraction

As the first step of preprocessing, I remove noises captured in both low and high frequencies by applying band-pass filters of 20Hz and 4kHz. Then, I construct forty-dimensional Mel-Frequency Cepstrum Coefficient (MFCC) frames, which is a standard feature extraction technique for speech recognition. MFCCs represent short-term power spectrums of the audio in mel frequency scale. With a 30ms window with 10ms shift in frame, one-second long utterances result in a matrix of size 101×40.

3.1.3 Model Architecture

Previously, Tang and Lin [57] integrate residual connections to a CNN architecture and achieve state-of-the-art accuracy. This type of architecture is called a ResNet and has been first explored by He et al. [16]; for an input x, it learns the residual $H(x) = F(x) + x$ instead of the true mapping $F(x)$. In their work, it is found that the direct connection can speed up the learning process by reducing the impact of vanishing gradients; ResNet achieves the best accuracy in the ILSVRC 2015 Image classification task with 3.57% error rate and shows 28% relative improvement on the COCO object detection dataset.

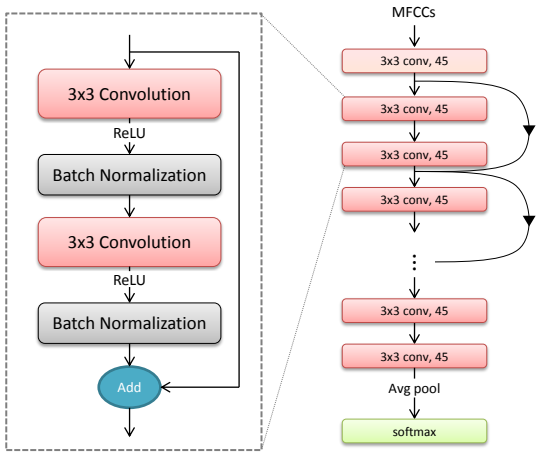


Figure 3.1: An illustration of `res8`, taken from Tang and Lin [57].

Figure 3.1 shows the ResNet of Tang and Lin. The first two layers are the convolutional layer and the average pooling layer. The following components are residual blocks. A residual block consists of two convolutional layers with rectified linear unit activation followed by a batch-normalization layer. After the residual blocks, the output is average-pooled and fed into a fully-connected softmax layer to construct output of the target size. In the original paper, the authors introduce three variations of ResNet which differ by the number of residual blocks: `res8`, `res15`, and `res26`—with 3, 6, and 12 residual blocks, respectively. Surprisingly, three residual blocks are sufficient to achieve a high accuracy of 94% and the benefit of additional residual blocks is found to be small.

A convolutional layer in this network has weights $w \in \mathbb{R}^{(m \times r) \times n}$, where m, r, and n represent the width, height, and number of feature maps, respectively. With 45 feature maps, `res8` requires about 110K parameters and 30 million multiplications (see Table 3.1, left). To find models that require less computation, the authors also study how accuracy changes with respect to the number of feature maps. It is found that 90% accuracy can be achieved with `res8-narrow`, a variation of `res8` with 19 feature maps. With fewer feature maps, `res8-narrow` only has 19K parameters and 5.7 million multiplications (see Table 3.1, right). In this work, my goal is to support KWS on a wide range of user-facing devices including desktops, laptops, tablets, and mobile devices. Since hand-held devices are not built for computationally expensive tasks, Honkling focuses on the shallowest models, `res8` and its variation, `res8-narrow`.

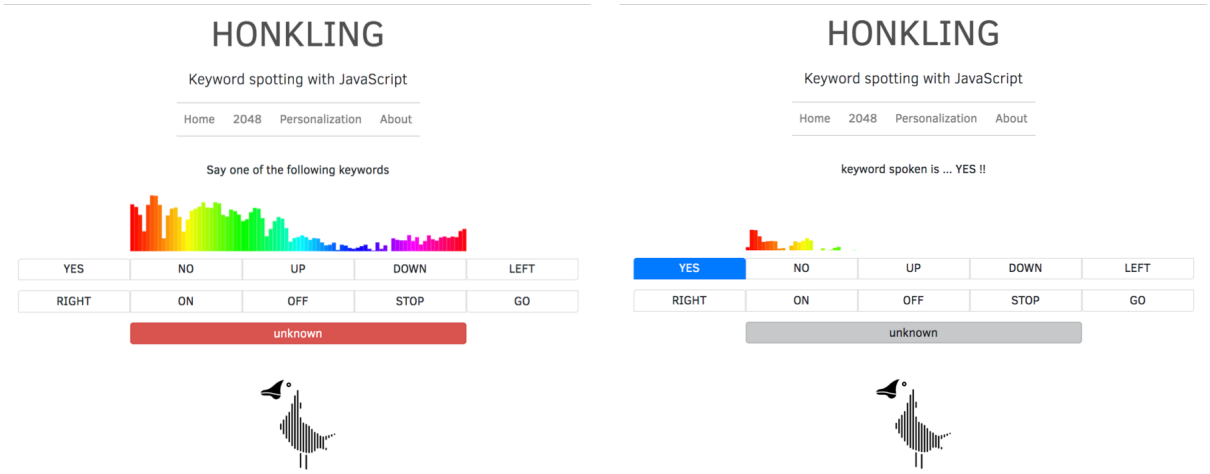


Figure 3.2: Keyword spotting on Honkling (silence is considered as unknown class).

3.1.4 Implementation

Honkling achieves KWS in browser using TensorFlow.js, a JavaScript library for numerical computation and large-scale machine learning. From my initial evaluation, it is found that training ResNet with TensorFlow.js requires much longer time than training with the original PyTorch implementation for ResNet, Honk.[1] Hence, I train the model for Honkling offline; its PyTorch parameters are stored as a JSON object and loaded to initialize its TensorFlow.js model when the page is loaded. Unfortunately, PyTorch and TensorFlow.js use different names for each of their network components. Therefore, Honkling initialization involves renaming each of the PyTorch components.

Manipulating audio with JavaScript is found to be difficult; Web Audio API is the only stable library for in-browser audio processing and many standard browsers restrict the sample rate of the input audio to 44.1kHz. Since the official PyTorch implementation extracts MFCCs using LibROSA, a well-known audio processing library for Python [38], Honkling is implemented with Meyda [46], a JavaScript implementation of LibROSA. Unfortunately, even though Meyda claims to provide the same functionality, the values of MFCCs extracted with Meyda are different from the values generated with LibROSA due to the aforementioned restrictions. Therefore, I have patched Meyda comprehensively to minimize the mismatches.

[1] https://github.com/castorini/honk

Device	RAM	CPU	GPU
Desktop	16 GB	i7-4790k (quad)	GTX 1080 Ti
MacBook Pro (2017)	16 GB	i5-7287U (quad)	Intel Iris Plus 650
MacBook Air (2013)	4 GB	i5-4260U (dual)	Intel HD 6000
Galaxy S8 (2017)	4 GB	Snapdragon 835 (octa)	Adreno 540

Table 3.2: The four devices used to evaluate in-browser inference latency.

3.2 Efficiency of In-Browser Keyword Detection

When designing a system that involves many network requests, developers often focus on reducing the number of network requests because they are unpredictable. For instance, when I ping the Google server using my university WiFi connection, I experience the average latency of 25ms with a standard deviation of 20ms. When a network call carries data, latency increases linearly as the amount of data being transferred over the network increases. With a Python server located in Newark, New Jersey, I measure an average latency of 481ms with a standard deviation of 183ms for one second of 16kHz mono-channel audio data sent from Waterloo.

Written purely in JavaScript, Honkling achieves KWS in the browser without any server-side support. In other words, it does not suffer from the network delays. Furthermore, users are now freed from security and privacy implications, such as eavesdropping of network traffic or collection of personal speech data [59]. In this section, I further show the stability of the serverless architecture by conducting a comprehensive evaluation on the efficiency of in-browser keyword detection.

3.2.1 Evaluation Setup

To be consistent with the previous experiments, this evaluation is conducted on the GSC test set and detects the same 12 classes of keyword. The two metrics I report are accuracy and inference latency. Since inference latency can be a processor dependent measure, I include different desktop, laptop, and smartphone configurations; Table 3.2 describes the hardware specifications of each device. Among the wide range of browsers, Chrome (v78.0) and Firefox (v71.0) are chosen for the evaluation because they are the most popular. Since TensorFlow.js provides optimized operations when the browser supports WebGL acceleration, the evaluation is conducted in both settings, with and without hardware acceleration.

	Device	Processor	Platform	res8		res8-narrow	
				Lat. (ms)	Acc. (%)	Lat. (ms)	Acc. (%)
GPU	Desktop	GTX 1080 Ti	PyTorch	1	94.3	1	91.2
	Desktop	GTX 1080 Ti	Firefox	8	94.1	7	90.9
	Desktop	GTX 1080 Ti	Chrome	9	94.0	7	90.8
	MacBook Pro (2017)	Intel Iris Plus 650	Firefox	17	94.0	10	90.8
	MacBook Pro (2017)	Intel Iris Plus 650	Chrome	24	94.0	11	90.8
	MacBook Air (2013)	Intel HD 6000	Firefox	36	94.0	20	90.8
	MacBook Air (2013)	Intel HD 6000	Chrome	25	94.0	12	90.8
	Galaxy S8 (2017)	Adreno 540	Firefox	60	94.1	43	89.0
	Galaxy S8 (2017)	Adreno 540	Chrome	54	93.9	33	90.9
CPU	Desktop	i7-4790k (quad)	PyTorch	10	94.3	2	91.2
	MacBook Pro (2017)	i5-7287U (quad)	PyTorch	12	94.2	3	91.2
	Desktop	i7-4790k (quad)	Firefox	354	94.1	86	90.9
	Desktop	i7-4790k (quad)	Chrome	109	94.0	30	90.8
	MacBook Pro (2017)	i5-7287U (quad)	Firefox	348	94.0	112	90.8
	MacBook Pro (2017)	i5-7287U (quad)	Chrome	91	94.0	48	90.8
	MacBook Air (2013)	i5-4260U (dual)	Firefox	566	94.0	120	90.8
	MacBook Air (2013)	i5-4260U (dual)	Chrome	125	94.0	37	90.8
	Galaxy S8 (2017)	Snapdragon 835 (octa)	Firefox	1105	94.1	265	89.0

Table 3.3: Latency (lat.; 90$^{\text{th}}$ percentile) and accuracy (acc.) results on different platforms for the `res8-*` models.

Unfortunately, Chrome evaluations on CPU are missing a number for Galaxy S8 because hardware acceleration cannot be disabled for Chrome running on a mobile device.

3.2.2 Evaluation Results

Table 3.3 summarizes 90$^{\text{th}}$ percentile latency and accuracy for both `res8` and `res8-narrow` on various devices. Along with Firefox and Chrome, the table includes baseline performance measured with the original PyTorch implementation. Even though there are small differences in accuracy among the entries, JavaScript implementation reports accuracy of around 94% with `res8` and around 91% with `res8-narrow` on every device. This concludes that the JavaScript implementation is correct and the mismatches between LibROSA and Meyda are negligible.

Inference latency is found to be strongly dependent on the underlying platform and processor. First, Firefox is found to be about 4 times slower than Chrome across devices. This is due to the difference in JavaScript engine; V8 of Chrome is faster than SpiderMonkey of Firefox [11]. Another finding is that inference latencies are much smaller on

GPUs. In the case of `res8`, the only CPU configuration that responds within 100ms is the 2017 MacBook Pro (91ms). On the other hand, the highest latency observed from GPU configurations is 60ms of Galaxy S8, a mobile device. I have observed a similar pattern with `res8-narrow`; the latency ranges from 7 to 43ms on GPUs and while it ranges from 30 to 265ms on CPUs.

In the experiments conducted by Miller [41], it is found that humans expect some communicative response within 2 seconds with the most effectiveness at about half a second. In other words, these delays are perceived by humans to be near instantaneous; Honkling supports real-time interactions on a wide range of devices including mobile devices.

3.3 Applications

Interestingly, JavaScript is not limited to web application development; with the development of several recent frameworks, we can develop applications for other platforms in JavaScript. Exploiting the flexibility of the JavaScript-based KWS system, I present three voice-enabled applications in vastly different domains [30].

3.3.1 Voice-enabled 2048

2048 is a popular game with four directional commands: "up", "down", "left", and "right". Recognizing its simplicity, I have integrated Honkling with an open-source JavaScript implementation of 2048 (see Figure 3.3).[2] Achieving hands-free interaction with the user, voice-enabled 2048 provides a unique experience. Such integration illustrates how Honkling can easily transform a traditional web application into voice-enabled application. Furthermore, by the nature of JavaScript, it is guaranteed to behave uniformly on every standard-compliant web browsers; the application is accessible through various hands-on devices ranging from desktops to mobile devices.

3.3.2 JavaScript-based Virtual Assistant

As a library developed for Node.js, Electron enables cross-platform desktop application development with JavaScript, HTML, and CSS. Here, I introduce a cross-platform voice-enabled virtual assistant. The virtual assistant supports a wide range of actions such as

[2] https://github.com/gabrielecirulli/2048

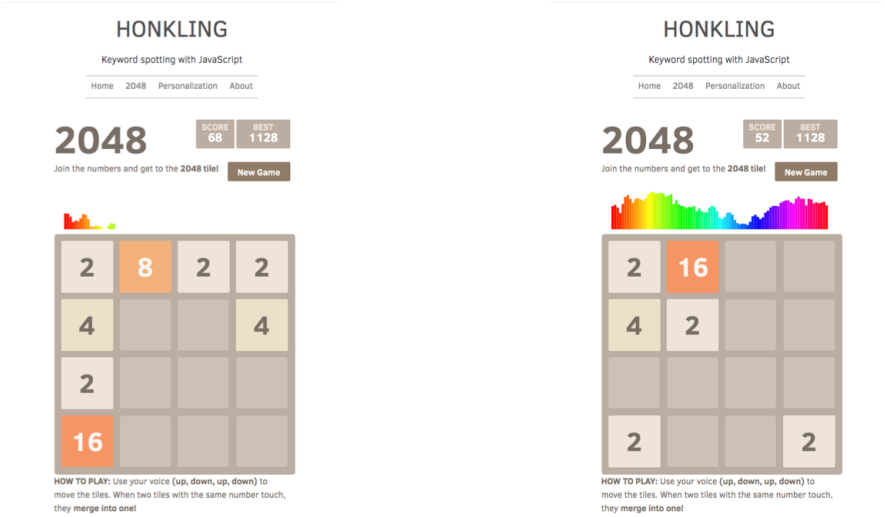


Figure 3.3: Screenshots of voice-enabled 2048 game.

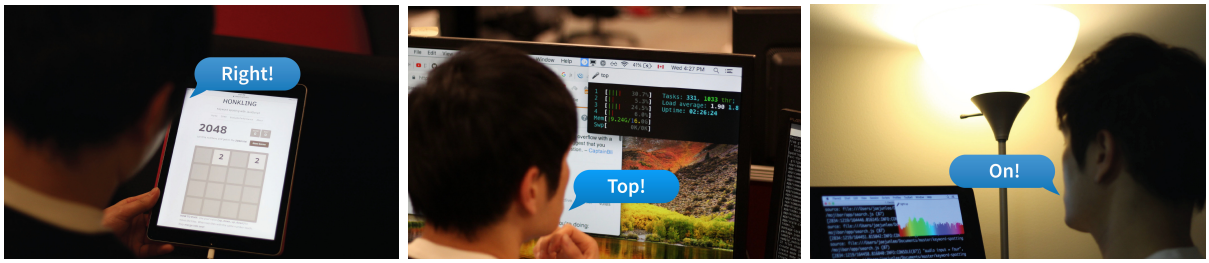


Figure 3.4: From left to right, photographs of the 2048 game, the desktop virtual assistant, and the smart home controller.

displaying the current memory and CPU usage, opening search results in the browser, controlling the volume, and filesystem navigation—needless to say, all operations are triggered by voice commands. While different virtual assistants already exist for desktop environments (e.g., Siri and Cortana), they only provide a fixed set of functionalities. On the other hand, a Honkling-based virtual assistant enables arbitrary customization since everything is transparently implemented in JavaScript.

3.3.3 Smart Home Controller

To further demonstrate the benefits of an efficient customization, I have integrated a smart home device with the virtual assistant. Among the various devices on the market, I have chosen Philips Hue light bulbs because they expose an API for controlling various features. With these light bulbs, users can turn light on and off and adjust the brightness through a mobile application. Therefore, I provide reliable hands-free interaction between the user and smart home devices by explicitly training a model that can detect four commands: “on”, “off”, “up”, and “down”.

Chapter 4

Personalized Keyword Spotting

Achieving KWS purely in JavaScript, Honkling enables speech-based interaction for a wide range of applications. However, KWS is sensitive to accents and therefore, it may struggle to detect keywords for some users. In this chapter, I study how Honkling behaves in the presence of underrepresented accents. To minimize the accuracy drop, I propose training a model with the GSC dataset and fine-tuning with user recordings. In this chapter, I conduct a set of experiments to find the best hyperparameter setting that gives the biggest gain in accuracy. Bringing the results to fruition, I enable personalization of Honkling with in-browser accent adaptation.

4.1 KWS Accuracy with Underrepresented Accents

Implementing `res8-narrow`, Honkling achieves an accuracy of 91% on the GSC dataset. However, such a high accuracy is not guaranteed in practice because the users are not always native speakers of English. To determine the effect of accents, I evaluate the accuracy of Honkling on the other accents.

4.1.1 User Recordings

For the following evaluation, I have collected recordings of 8 different accents: two Canadian, one British, two Korean, two Chinese, and one American. Since Honkling depends on the GSC dataset, each participant is asked to record the 30 keywords of the GSC dataset;

Dataset	Accent	Accuracy (%)
Google Speech Commands	American	91.5
User *A*	Canadian	89.2
User *B*	Canadian	86.9
User *C*	British	77.6
User *D*	Korean	91.4
User *E*	Korean	81.9
User *F*	Chinese	91.7
User *G*	Chinese	80.0
User *H*	American	89.2

Table 4.1: Accuracy of `res8-narrow` on GSC test set and user test sets.

50 samples are collected for the 10 positive classes and 10 samples are collected for the other 20 classes—total of 700 samples.

4.1.2 Evaluation Setup

First, I train `res8-narrow` the same way as described in Section 3.1. I report accuracy on the 8 test sets constructed for each participant. The test set has 40 samples for each class; with 12 classes, the total number of samples is 560. Since we have 50 recordings for each of the positive classes, 40 recordings are randomly selected from them. For silence, 40 samples are selected from the original GSC background noise set. Lastly, unknown samples are randomly chosen from the 200 recordings of negative classes.

In the previous section, I have mentioned that training a model with TensorFlow.js is slower than training with the original PyTorch implementation. Since I have already demonstrated that Honkling successfully replicates the functionality of the original implementation, this evaluation is conducted in PyTorch.

4.1.3 Evaluation Results

Table 4.1 summarizes the average accuracy collected from 100 evaluations. The first row is the accuracy measured on the GSC test set. While the model has an accuracy of 91.5% on the GSC test set, its accuracy on the user test sets varies from 77.6% to 91.7%—the high accuracy is not observed from every user.

Unsurprisingly, the model can detect Canadian and American accents with high accuracy; Accuracy of 89.2% and 86.9% are reported for the two Canadian accents and the American user also shows 89.2% detection accuracy. A British accent, on the other hand, introduces absolute accuracy degradation of 13.9%. The accuracy varies most when the user is Asian—possibly due to the fact that they are not native speakers of English. For the Korean accents of user *D* and *E*, `res8-narrow` reports accuracy of 81.9% and 91.4%, respectively. Accuracy reported from the Chinese users also have high variance; while one user shows an accuracy of 91.7%, the other user demonstrates limited accuracy of 80.0%.

4.2 User Accent Adaptation by Fine-Tuning

The main topic of this section is fine-tuning with user recordings for minimizing the accuracy degradation. The key assumption is that, as the base model is fine-tuned, the model will learn the difference in accents, increasing the accuracy for the target user.

In order to achieve accent adaptation by fine-tuning, the system needs two things. First, the system needs to collect audio samples from the user. Second, the system needs time and resources for fine-tuning. Since these have a strong correlation with the amount of effort the user needs to devote, the optimal accent adaptation process must require minimal recordings and efficiently fine-tune the model. Needless to say, the fine-tuned model should demonstrate higher detection accuracy for the target user.

With the goal of minimizing user interaction throughout the accent adaptation process, I conduct a set of hyperparameter experiments and find a configuration that adapts to the user accent best.

4.2.1 Experimental Setup

Again, I start from `res8-narrow` trained with the GSC dataset for the 12 classes (see Section 3.1 for details). Then, the model is fine-tuned with the user recordings collected for the accuracy evaluation in Section 4.1. From my initial evaluation, I found that 10 recordings are sufficient for adapting the target accent. Therefore, I set aside 10 samples from each class for constructing the fine-tuning set. The remaining samples are used to construct the test set. To reduce the effect of outliers, I repeat the experiment 100 times and report averaged accuracy along with 95% confidence interval. For each trial, the fine-tuning set is reconstructed with a different random seed. On the other hand, I keep the test set static for fair comparisons among the fine-tuned models; the test set consists of

the remaining 40 recordings for each positive class and 40 samples are randomly selected for the unknown class.

Since the size of the fine-tuning set refers to the number of recordings that Honkling need to collect from users, it is important to understand how accuracy changes with respect to the size of the fine-tuning set. Hence, I evaluate the three different sizes: one, three, and five samples per keyword. For each experiment, the target number of samples are randomly selected from the ten samples set aside for fine-tuning set construction.

Another goal of this experiment is to understand the relationship between the accuracy and different types of hyperparameters. The three types of hyperparameters I examine are: number of epochs, optimizer, and learning rate. Since the learning rate and the optimizer are often closely related, I first evaluate different optimizers and provide a study on the learning rate for the best optimizer.

In the following graphs, I have two types of accuracy reported: `original-*` and `personalized-*`, denoting the accuracy on the original test set and the user test set, respectively. In Figures 4.1 and 4.3, dashed lines refer to `original-*` and solid lines refer to `personalized-*`. Similarly, transparent bars are used for `original-*` and opaque bars are used for `personalized-*` in Figure 4.2. The number that follows each label denotes the number of recordings per keyword.

4.2.2 Experimental Results

Fine-Tuning Progress across Epochs

First, I describe how the accuracy changes as the number of epochs increases. In this experiment, the learning rate is fixed to 0.01 and each model is trained with the Stochastic Gradient Descent (SGD) optimizer. Unsurprisingly, the accuracy on the user test set increases across epochs while the accuracy on the original test set decreases. In Figure 4.1, diminishing returns are observed with more epochs; 50 epochs seem to be sufficient to maximize the accuracy.

At convergence, every user shows 4∼10% increase in absolute accuracy on the corresponding user test set. Before fine-tuning, `res8-narrow` achieves an accuracy of 77.6% on user C's test set. However, when the model is fine-tuned with five recordings per keyword, the model accuracy increases to 86.7%. Surprisingly, 86.7% is the lowest fine-tuning accuracy reported from the eight participants for the fine-tuning set of five recordings per keyword. On the other hand, user H shows the highest accuracy of 95.6% with 6.4% increase in absolute accuracy.

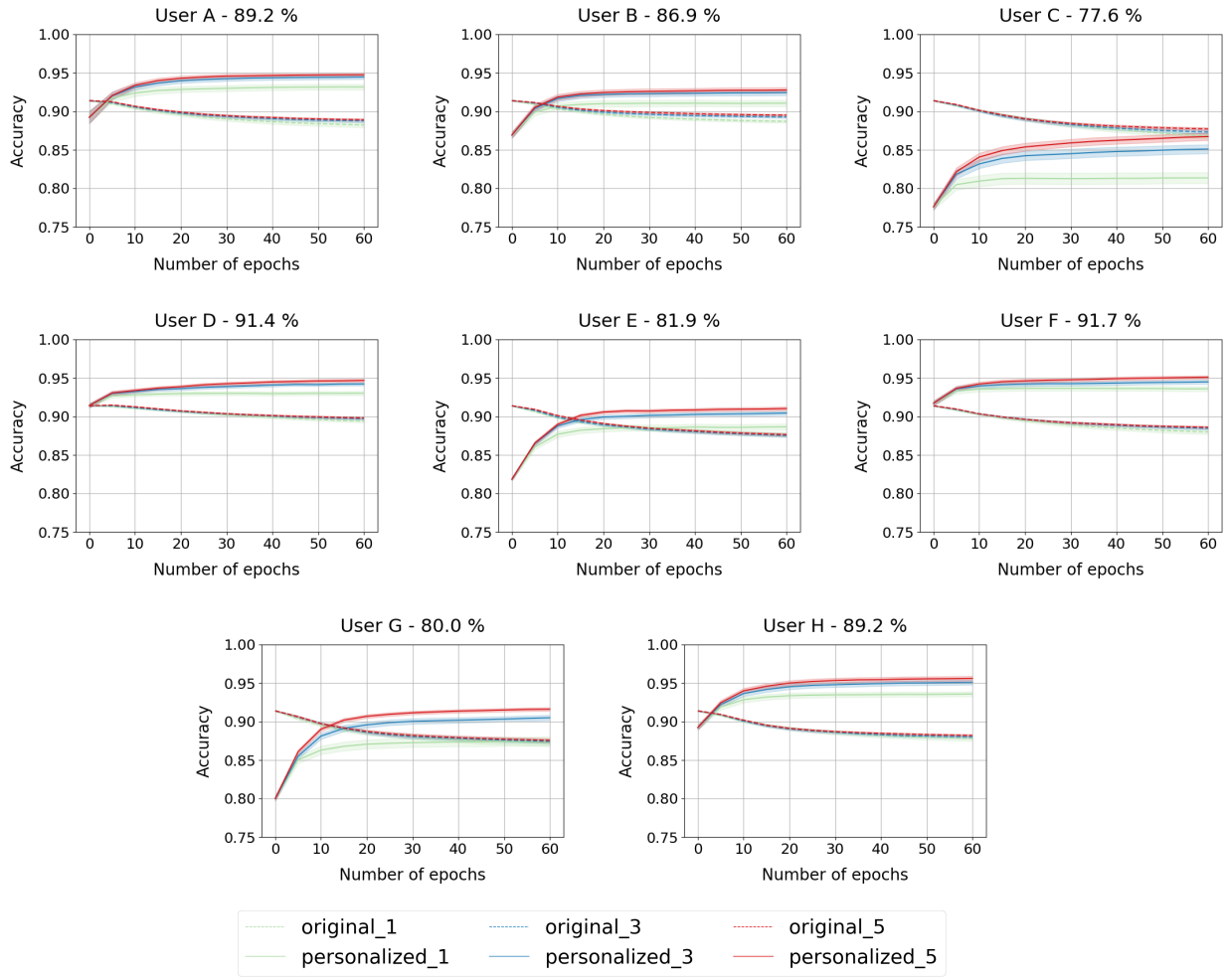


Figure 4.1: Change in accuracy across epochs, along with 95% confidence interval (shaded).

The accuracy decrease on the original test set also converges around 50 epochs. Interestingly, the lowest accuracy is 86.8% indicating that the fine-tuned model can still detect keywords for non-target users with decent accuracy.

Fine-Tuning Dataset Size

Surprisingly, a single recording per keyword is found to be sufficient for personalization as the models achieve higher accuracy on the user test set than on the original test set for

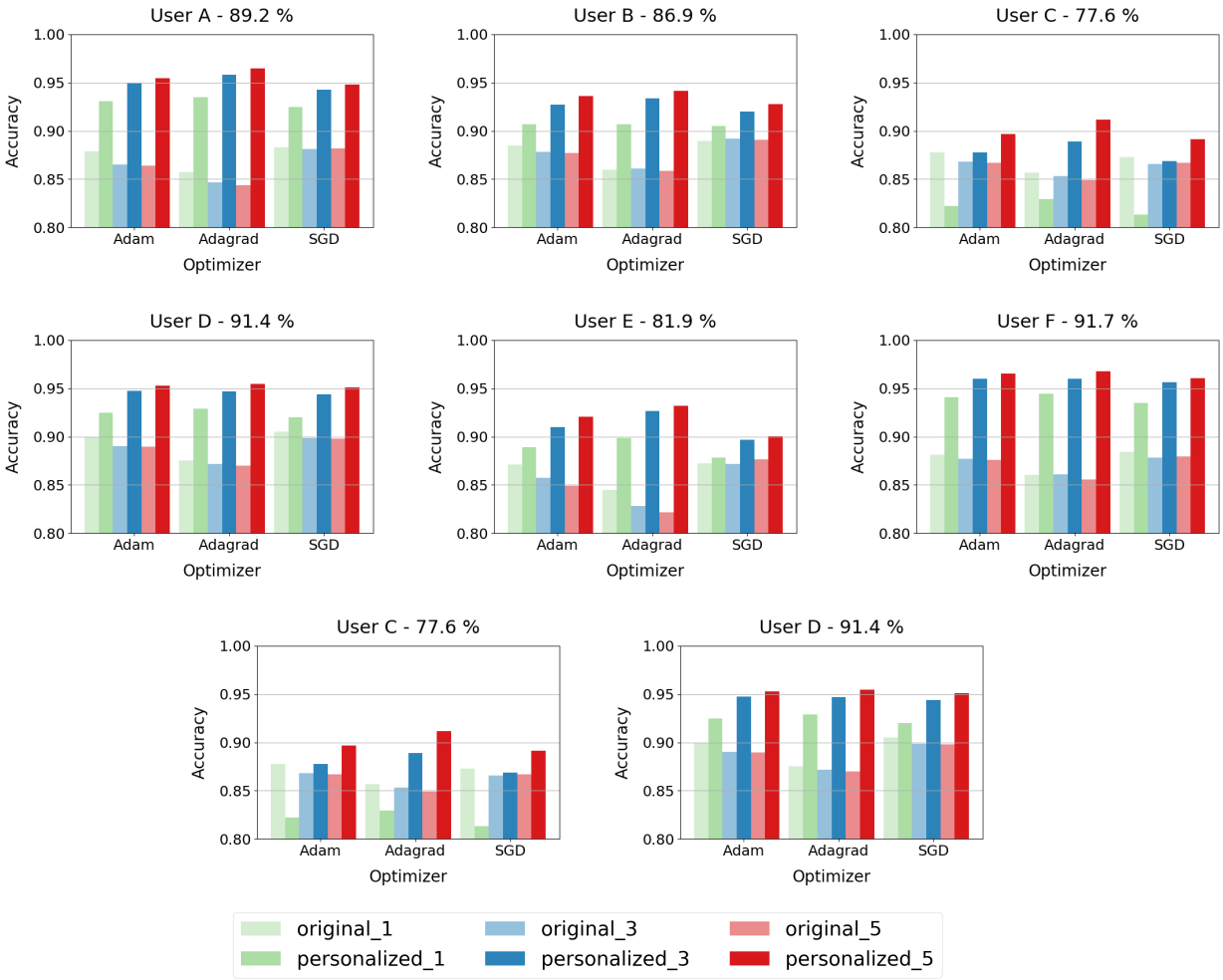


Figure 4.2: Change in accuracy for Adam, Adagrad and SGD optimizer.

every user except user C. In general, an increase in accuracy is observed as more recordings are added to the fine-tuning set; more training data leads to a better representation of a user's speech patterns. However, diminishing returns are observed with the accuracy after a mere five recordings per keyword; such a trend is evident for every user (compare `personalized_{1,3,5}` in Figures 4.1, 4.2, and 4.3). Concretely, the accuracy gap between one and three recordings is substantially greater than the gap between three and five, suggesting that each additional recording provides rapidly diminishing returns. Such a trend is well captured in the graphs of user C and G. In Figure 4.1, the final accuracy of

user C on the test set are 81%, 85%, and 87% for one, three, and five samples per keyword. Similarly, I observe accuracy of 87%, 91%, and 92% from user G, respectively. Since the marginal benefit of two more recordings is quite large, results suggest that having at least three recordings is desirable although using one sample per keyword helps.

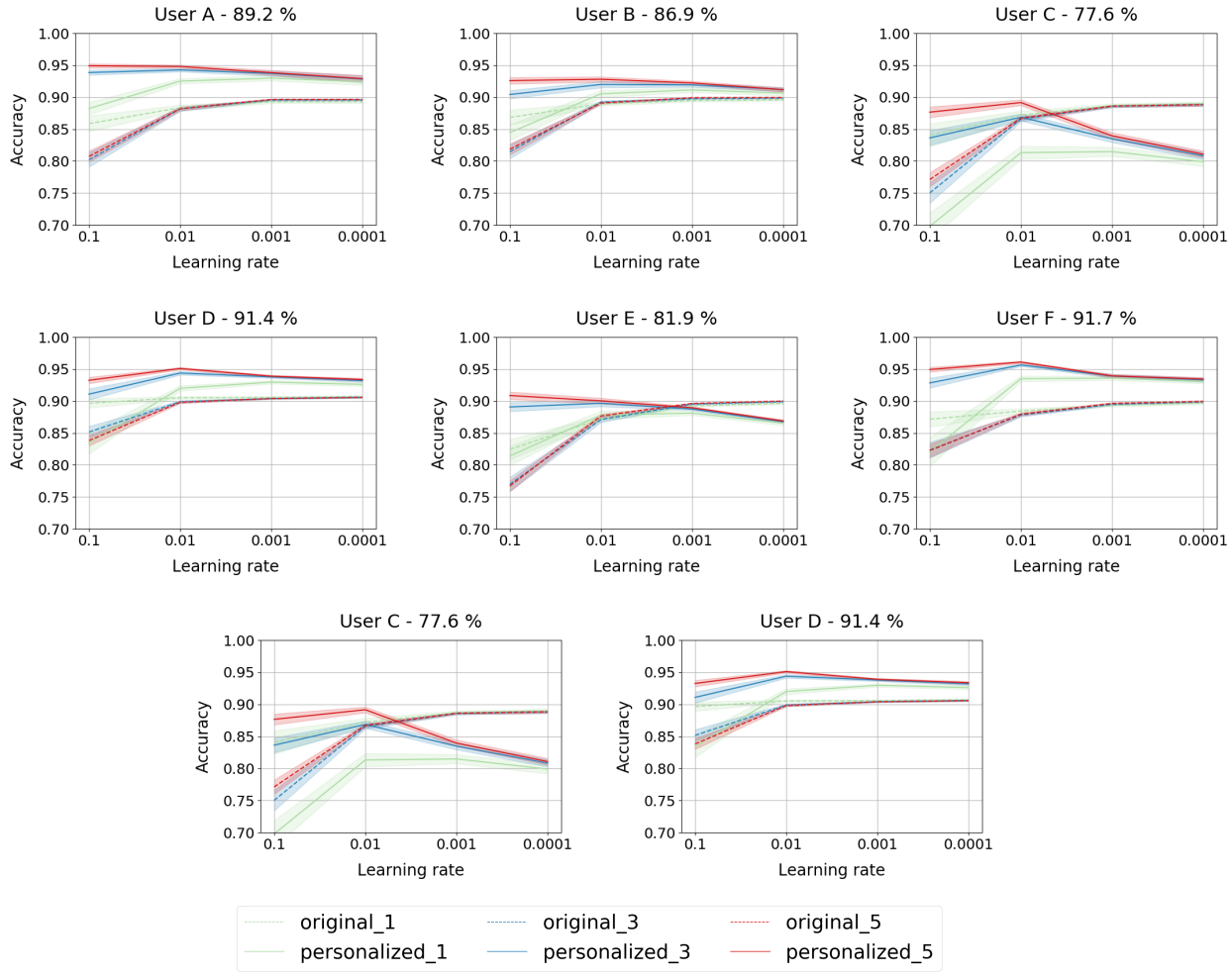


Figure 4.3: Change in accuracy while varying learning rate, along with 95% confidence interval (shaded).

Fine-Tuning with Different Optimizers

Since each optimizer has different strategies for updating weights, the three optimizers, Adam, Adagrad, and SGD optimizer, lead to differences in accuracy. From an initial experiments, it is found that the Adagrad optimizer reports the highest fine-tuning accuracy when trained with learning rate of 0.01, while the Adam optimizer achieves the highest fine-tuned accuracy with learning rate of 0.001. Therefore, I set learning rate to 0.001 for the Adam optimizer while 0.01 is used for the Adagrad and SGD optimizers.

Figure 4.2 summarizes how the accuracy changes when `res8-narrow` is trained with different optimizers for 25 epochs. Though the three optimizers show minimal differences, the Adagrad optimizer consistently achieves the highest accuracy on the user test sets. Unfortunately, the high accuracy comes at the cost of low accuracy for non-target users—the lowest accuracy on the original test set is also observed from the Adagrad optimizer. The optimizer with the highest accuracy on the original set after fine-tuning is the SGD optimizer, which also shows a comparable increase on the user test set.

Fine-tuning Progress across Learning Rates

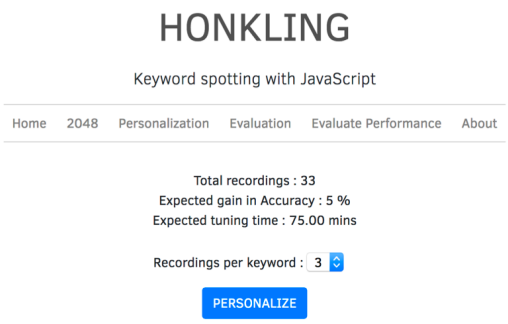
In this experiment, I have trained each model with the SGD optimizer and performed linear search on the learning rate from 0.1 to 0.0001, stepping by a factor of ten. Each model is trained for 25 epochs. As shown in Figure 4.3, learning rate of 0.01 consistently leads to the highest accuracy for the target user. When a model is trained with a learning rate of 0.1, its accuracy on the user test set is shown to be similar to the accuracy from learning rate of 0.01. However, the former model has much lower accuracy than the latter model on the original test set.

4.3 In-Browser Accent Adaptation

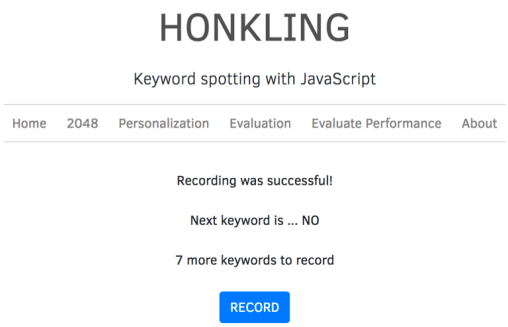
Bringing all the previous threads together, I have implemented in-browser fine-tuning, exploiting the decentralized and serverless nature of the JavaScript-based KWS system. In this section, I measure the efficiency of in-browser fine-tuning on a wide range of devices.

4.3.1 Implementation

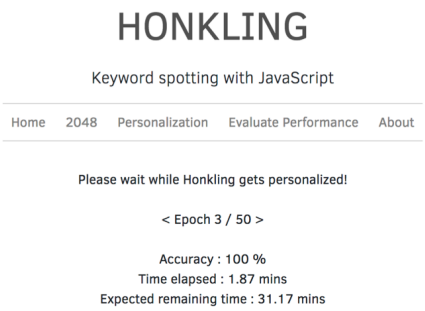
Figure 4.4 illustrates how Honkling supports personalization with accent adaptation. From the previous experiment in Section 4.2.2, I have shown that the number of recordings has



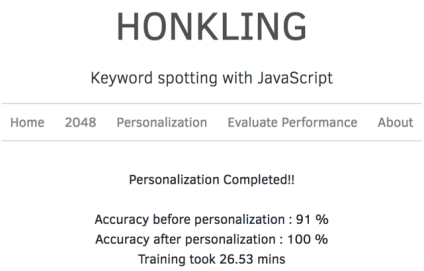
(a) Fine-tuning set size selection



(b) User recording collection



(c) Accent adaptation by fine-tuning



(d) Fine-tuned accuracy report

Figure 4.4: The four steps of accent adaptation with Honkling.

a high correlation with the quality of personalization. Therefore, Honkling first asks the number of recordings that user is willing to provide; users have option to record one, three, or five samples per keyword. Once the recordings are collected, the base model is fine-tuned in the browser, with the SGD optimizer for 50 epochs at a learning rate of 0.01.

To prevent repetitive personalization, I also store the fine-tuned model in the browser. At startup, Honkling loads the stored model if it exists; users can keep the personalized KWS system even after the current browser session ends.

4.3.2 Evaluation Setup

The efficiency of in-browser fine-tuning is evaluated on two types of devices: a desktop and a laptop. The desktop runs Ubuntu 18.04 and has 16GB RAM, an i7-4790k CPU, and a

	Device	Processor	Platform	Number of Recordings		
				1	3	5
GPU	Desktop	GTX 1080 Ti	PyTorch	0.2 sec	0.2 sec	0.2 sec
	Desktop	GTX 1080 Ti	Firefox	3.9 sec	5.9 sec	7.6 sec
	Desktop	GTX 1080 Ti	Chrome	3.0 sec	4.8 sec	6.6 sec
	MacBook Pro (2017)	Intel Iris Plus 650	Firefox	7.2 sec	12.6 sec	27.0 sec
	MacBook Pro (2017)	Intel Iris Plus 650	Chrome	7.2 sec	13.2 sec	19.8 sec
CPU	Desktop	i7-4790k (quad)	PyTorch	3.3 sec	6.0 sec	8.0 sec
	MacBook Pro (2017)	i5-7287U (quad)	PyTorch	2.0 sec	5.9 sec	10.7 sec
	Desktop	i7-4790k (quad)	Firefox	25.4 min	75.8 min	128.1 min
	Desktop	i7-4790k (quad)	Chrome	8.9 min	26.9 min	45.4 min
	MacBook Pro (2017)	i5-7287U (quad)	Firefox	29.5 min	86.3 min	139.2 min
	MacBook Pro (2017)	i5-7287U (quad)	Chrome	6.8 min	20.4 min	34.1 min

Table 4.2: In-browser fine-tuning latency for `res8-narrow` under different configurations.

GTX 1080 Ti GPU. The laptop configuration is 2017 MacBook Pro that runs High Sierra. It has 16GB RAM, a i5-7287U CPU, and Intel Iris Plus 650 GPU. Similar to the previous efficiency evaluation, the evaluation is conducted on Chrome (v78.0) and Firefox (v71.0) with and without hardware acceleration.

To reduce the effects of outliers, I repeat the evaluation ten times. For each trial, I randomly select one of the eight user recording sets and reconstruct the fine-tuning set by selecting the target number of recordings at random. In this evaluation, I use `res8-narrow` to report the fine-tuning time for all three variations of fine-tuning set size—one, three, and five recordings per keyword.

4.3.3 Evaluation Results

Table 4.2 summarizes the averaged in-browser fine-tuning latency. The table also includes a measurement from the original PyTorch implementation for reference. First of all, Chrome is again found to be more optimized than Firefox. From this evaluation, it is also found that personalization time increases with the data size. On CPU, fine-tuning with single sample per keyword takes about 10 minutes on Chrome and 30 minutes on Firefox. When a user provides five samples per keyword, Firefox requires 2.3 hours while only 45 minutes are needed on Chrome. Since training data size correlates with the final accuracy, users have the option to trade off time and quality.

Fortunately, GPU acceleration can significantly decrease the fine-tuning time. On the laptop configuration, the same process that consumes up to 2.3 hours on a CPU can be completed within 27 seconds. Similarly, fine-tuning with GPU only takes eight seconds on the desktop for five recordings per keyword—Honkling can increase the absolute accuracy up to 10% with only eight seconds of fine-tuning (see Section 4.2.2).

Chapter 5

Conclusions and Future Work

Realizing the importance of the "write once, run anywhere" philosophy, I present Honkling, a novel, JavaScript-based KWS system. With the support of different cross-platform development libraries, Honkling provides a reliable KWS-based interaction on a wide range of applications running on desktops, laptops, as well as mobile devices. In my evaluation, I find that Honkling can detect the target keywords in the browser within 0.5 seconds on modern devices including mobile—the JavaScript-based KWS system supports real-time speech-based interaction between the user and the application.

Exploiting the decentralized and serverless architecture of Honkling, I also support in-browser fine-tuning for increasing the detection accuracy for each user. By conducting thorough experiments, I find that a 10% accuracy increase can be achieved with a meager five recordings per keyword, which takes only eight seconds to fine-tune in the browser, in the presence of hardware acceleration.

Following this work, I believe reducing the model size would further improve the user experience. The process of deleting irrelevant parts of a network is called network pruning, and common techniques include network slimming [35] and weight-based pruning [15]. Since fine-tuning can take up to 2.3 hours without hardware acceleration, reducing the model size prior to fine-tuning would greatly speed up the process.

Even though the concept of voice-enabled user interfaces is fairly young, it has changed the shape of our daily lives in many ways. I believe Honkling has contributed to this change by enabling efficient and accurate keyword detection on a wide range of user-facing applications. With the advance in technology, I am excited to see the convenience that Honkling will bring to everyday life.

References

[1] Sercan O. Arik, Markus Kliegl, Rewon Child, Joel Hestness, Andrew Gibiansky, Chris Fougner, Ryan Prenger, and Adam Coates. Convolutional recurrent neural networks for small-footprint keyword spotting. *arXiv:1703.05390*, 2017.

[2] Bijin Chen and Zhiqi Xu. A framework for browser-based multiplayer online games using WebGL and WebSocket. In *IEEE International Conference on Multimedia Technology*, pages 471–474. IEEE, 2011.

[3] Guoguo Chen, Carolina Parada, and Georg Heigold. Small-footprint keyword spotting using deep neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4087–4091. IEEE, 2014.

[4] Guoguo Chen, Carolina Parada, and Tara N. Sainath. Query-by-example keyword spotting using long short-term memory networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5236–5240. IEEE, 2015.

[5] Paulo R. M. de Andrade, Adriano B. Albuquerque, Otávio F. Frota, Robson V. Silveira, and Fátima A. da Silva. Cross platform app: a comparative study. *arXiv:1503.03511*, 2015.

[6] Shamalee Deshpande, Sharat Chikkerur, and Venu Govindaraju. Accent classification in speech. In *IEEE Workshop on Automatic Identification Advanced Technologies*, pages 139–143. IEEE, 2005.

[7] Xingjian Du, Mengyao Zhu, Mingyang Chai, and Xuan Shi. End-to-end model for keyword spotting with trainable window function and DenseNet. In *IEEE International Conference on Digital Signal Processing*, pages 1–4. IEEE, 2018.

[8] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. In *International Conference on Artificial Neural Networks*, pages 220–229. Springer, 2007.

[9] Javier Fernández-Marqués, W. Tseng Vincent, Sourav Bhattachara, and Nicholas D. Lane. BinaryCmd: Keyword spotting with deterministic binary basis. In *SysML*, 2018.

[10] David G. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[11] Alexander Fox. Firefox Quantum vs. Chrome: The Verdict. https://www.lifewire.com/firefox-quantum-vs-chrome-4176159, 2019.

[12] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv:1803.03635*, 2018.

[13] Kareem E. Gebaly and Jimmy Lin. Afterburner: The case for in-browser analytics. *arXiv:1605.04035*, 2016.

[14] Kareem E. Gebaly, Lukasz Golab, and Jimmy Lin. Portable in-browser data cube exploration. *KDD Workshop on Interactive Data Exploration and Analytics*, pages 35–39, 2017.

[15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[17] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Evaluating cross-platform development approaches for mobile applications. In *International Conference on Web Information Systems and Technologies*, pages 120–138. Springer, 2012.

[18] Chao Huang, Tao Chen, and Eric Chang. Accent issues in large vocabulary continuous speech recognition. *International Journal of Speech Technology*, 7(2-3):141–153, 2004.

[19] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[20] Yan Huang, Dong Yu, Chaojun Liu, and Yifan Gong. Multi-accent deep neural network acoustic model with accent-specific top layer using the KLD-regularized model adaptation. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[21] Jason J. Humphries, Philip C. Woodland, and David Pearce. Using accent-specific pronunciation modelling for robust speech recognition. In *Proceeding of Fourth International Conference on Spoken Language Processing*, volume 4, pages 2324–2327, 1996.

[22] Biing-Hwang Juang and Lawrence Rabiner. Automatic speech recognition - A brief history of the technology development. 01 2005.

[23] Chieh-Chi Kao, Weiran Wang, Ming Sun, and Chao Wang. R-CRNN: Region-based convolutional recurrent neural network for audio event detection. *arXiv:1808.06627*, 2018.

[24] Liu Wai Kat and Pascale Fung. Fast accent identification and accented speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 221–224, 1999.

[25] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv:1408.5882*, 2014.

[26] Tom Ko, Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. Audio augmentation for speech recognition. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[28] Egor Lakomkin, Sven Magg, Cornelius Weber, and Stefan Wermter. KT-Speech-Crawler: Automatic dataset construction for speech recognition from youtube videos. *arXiv:1903.00216*, 2019.

[29] Jaejun Lee, Raphael Tang, and Jimmy Lin. Honkling: In-browser personalization for ubiquitous keyword spotting. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 2019.

[30] Jaejun Lee, Raphael Tang, and Jimmy Lin. Universal voice-enabled user interfaces using JavaScript. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 81–82, 2019.

[31] Kai-Fu Lee. Context-independent phonetic hidden markov models for speaker-independent continuous speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(4):599–609, 1990.

[32] Kai-Fu Lee and Hsiao-Wuen Hon. Speaker-independent phone recognition using hidden markov models. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(11):1641–1648, 1989.

[33] Yiyun Liang, Zhucheng Tu, Laetitia Huang, and Jimmy Lin. CNNs for NLP in the browser: Client-side deployment and visualization opportunities. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 61–65, 2018.

[34] Jimmy Lin and Kareem El Gebaly. The future of big data is... JavaScript? *IEEE Internet Computing*, 20(5):82–88, 2016.

[35] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[36] John Makhoul, Francis Kubala, Timothy Leek, Daben Liu, Long Nguyen, Richard Schwartz, and Amit Srivastava. Speech and language technologies for audio indexing and retrieval. *Proceedings of the IEEE*, 88(8):1338–1353, 2000.

[37] Jonathan Mamou, David Carmel, and Ron Hoory. Spoken document retrieval from call-center conversations. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 51–58. ACM, 2006.

[38] Brian McFee, Colin Raffel, Dawen Liang, Daniel P. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. LibROSA: Audio and music signal analysis in Python. In *Proceedings of the 14th Python in Science Conference*, pages 18–25, 2015.

[39] Assaf Hurwitz Michaely, Xuedong Zhang, Gabor Simko, Carolina Parada, and Petar Aleksic. Keyword spotting for Google assistant using contextual speech recognition. In *IEEE Automatic Speech Recognition and Understanding Workshop*, pages 272–278. IEEE, 2017.

[40] David R. Miller, Michael Kleber, Chia-Lin Kao, Owen Kimball, Thomas Colthurst, Stephen A. Lowe, Richard M. Schwartz, and Herbert Gish. Rapid and accurate spoken term detection. In *Proceedings of the IEEE International Conference on Computer Vision*, 2007.

[41] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the Fall Joint Computer Conference*, pages 267–277, 1968. doi: 10.1145/1476589.1476628.

[42] Maryam Najafian, Saeid Safavi, John H. Hansen, and Martin Russell. Improving speech recognition using limited accent diverse British English training data with deep neural networks. In *IEEE 26th International Workshop on Machine Learning for Signal Processing*, pages 1–6, 2016.

[43] Robin Nunkesser. Beyond Web/Native/Hybrid: A new taxonomy for mobile app development. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 214–218. IEEE, 2018.

[44] Carol Pedersen and Joachim Diederich. Accent classification using support vector machines. In *IEEE/ACIS International Conference on Computer and Information Science*, pages 444–449. IEEE, 2007.

[45] Changhua Pei, Youjian Zhao, Guo Chen, Ruming Tang, Yuan Meng, Minghua Ma, Ken Ling, and Dan Pei. Wifi can be the weakest link of round trip network latency in the wild. In *IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[46] Hugh Rawlinson, Nevo Segal, and Jakub Fiala. Meyda: an audio feature extraction library for the web audio API. In *The 1st Web Audio Conference*, 2015.

[47] Robin J. Rohlicek, William Russell, Salim Roukos, and Herbert Gish. Continuous hidden markov modeling for speaker-independent word spotting. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 627–630. IEEE, 1989.

[48] Richard C. Rose and Douglas B. Paul. A hidden markov model based keyword recognition system. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 129–132. IEEE, 1990.

[49] Tara N. Sainath and Carolina Parada. Convolutional neural networks for small-footprint keyword spotting. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[50] Hiroaki Sakoe, Seibi Chiba, and Waibel. Dynamic programming algorithm optimization for spoken word recognition. 26:43–49, 1978.

[51] George Saon and Jen-Tzung Chien. Large-vocabulary continuous speech recognition systems: A look at some recent advances. *IEEE Signal Processing Magazine*, 29(6): 18–33, 2012.

[52] Changhao Shan, Junbo Zhang, Yujun Wang, and Lei Xie. Attention-based end-to-end models for small-footprint keyword spotting. *arXiv:1803.10916*, 2018.

[53] Marius-Calin Silaghi and Hervé Bourlard. Iterative posterior-based keyword spotting without filler models. In *IEEE Automatic Speech Recognition and Understanding Workshop*, pages 213–216, 1999.

[54] Beatriz Soret, Preben Mogensen, Klaus I. Pedersen, and Mari C. Aguayo-Torres. Fundamental tradeoffs among reliability, latency and throughput in cellular networks. In *IEEE Globecom Workshops*, pages 1391–1396. IEEE, 2014.

[55] Ming Sun, Anirudh Raju, George Tucker, Sankaran Panchapagesan, Gengshen Fu, Arindam Mandal, Spyros Matsoukas, Nikko Strom, and Shiv Vitaladevuni. Max-pooling loss training of long short-term memory networks for small-footprint keyword spotting. In *IEEE Spoken Language Technology Workshop*, pages 474–480. IEEE, 2016.

[56] Mohammad M. Tanabian and Rafik A. Goubran. Accent adaptation in speech user interface. In *IEEE Symposium on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pages 99–102. IEEE, 2005.

[57] Raphael Tang and Jimmy Lin. Deep residual learning for small-footprint keyword spotting. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5484–5488, 2018. doi: 10.1109/ICASSP.2018.8462688.

[58] Raphael Tang, Weijie Wang, Zhucheng Tu, and Jimmy Lin. An experimental analysis of the power consumption of convolutional neural networks for keyword spotting. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5479–5483, 2018. doi: 10.1109/ICASSP.2018.8461624.

[59] Jaclyn Trop. The spy inside your car. Jan 2019. URL http://fortune.com/2019/01/24/the-spy-inside-your-car/.

[60] Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics and Systems Analysis*, 4(1):52–57, 1968.

[61] Dong Wang, Shaohe Lv, Xiaodong Wang, and Xinye Lin. Gated convolutional LSTM for speech commands recognition. In *International Conference on Computational Science*, pages 669–681. Springer, 2018.

[62] Pete Warden. Launching the speech commands dataset. https://research.googleblog.com/2017/08/launching-speech-commands-dataset.html, 2017.

[63] Mitchel Weintraub. Keyword-spotting using SRI's DECIPHER large-vocabulary speech-recognition system. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 463–466. IEEE, 1993.

[64] Steve Young. Large vocabulary continuous speech recognition: A review. *IEEE Signal Processing Magazine*, 13(5):45–57, 1996.

[65] Mengjun Zeng and Nanfeng Xiao. Effective combination of DenseNet and BiLSTM for keyword spotting. *IEEE Access*, 7:10767–10775, 2019.

[66] Jing-yun Zhang, Lu Huang, and Jia-song Sun. Keyword spotting with long short-term memory neural network architectures. *DEStech Transactions on Computer Science and Engineering*, 2017.