Honkling: In-Browser Personalization for Ubiquitous Keyword Spotting

Jaejun Lee, Raphael Tang, and **Jimmy Lin**
David R. Cheriton School of Computer Science
University of Waterloo

Abstract

Used for simple commands recognition on devices from smart speakers to mobile phones, keyword spotting systems are everywhere. Ubiquitous as well are web applications, which have grown in popularity and complexity over the last decade. However, despite their obvious advantages in natural language interaction, voice-enabled web applications are still few and far between. We attempt to bridge this gap with Honkling, a novel, JavaScript-based keyword spotting system. Purely client-side and cross-device compatible, Honkling can be deployed directly on user devices. Our in-browser implementation enables seamless personalization, which can greatly improve model quality; in the presence of underrepresented, non-American user accents, we can achieve up to an absolute 10% increase in accuracy in the personalized model with only a few examples.

1 Introduction

With the rapid proliferation of voice-enabled devices such as the Amazon Echo and the Apple iPhone, speech recognition systems are becoming increasingly prevalent in our daily lives. Importantly, these systems improve safety and convenience in hands-free interactions, such as using Apple's Siri to dial contacts while driving. However, a prominent drawback is that most of these systems perform speech recognition in the cloud, where a remote server receives from the device all audio to be transcribed. Clearly, the privacy and security implications are significant: servers may be accessed by other people, authorized or not. Thus, it is important to capture only the relevant speech and not all incoming audio, while providing a pleasant hands-free experience.

Enter keyword spotting systems. They solve the aforementioned issues by implementing an on-device mechanism to awaken the intelligent agent, e.g., "Okay, Google" for triggering the Google Assistant. This then allows the device to record and transmit only a limited segment of speech, obviating the need to send everything to the cloud. The task of keyword spotting (KWS) is to detect the presence of specific phrases in a stream of audio, often with the end goal of wake-word detection or simple command recognition on the device. Currently, the state of the art uses lightweight neural networks (Sainath and Parada, 2015; Tang and Lin, 2018), which can perform inference in real-time, even on low-end devices (Fernández-Marqués et al., 2018; Tang et al., 2018).

Despite the popularity of voice-enabled products, web applications have yet to make use of KWS. This is surprising, since modern web applications are supported on billions of devices ranging from desktops to smartphones. We close the gap between KWS systems and web applications by building and evaluating a JavaScript-based, in-browser KWS system. Exploiting the pervasiveness of JavaScript, our system can be deployed directly on user devices, facilitating the development of JavaScript-based, voice-enabled applications (Lee et al., 2019).

We observe, however, that the quality of our models suffers on various accents that are scarcely represented in our dataset—a problem common in speech recognition (Huang et al., 2004; Humphries et al., 1996). Fortunately, our JavaScript-based application runs completely client-side, enabling the development of personalized models. To further improve the universality of our system, we explore the benefits and costs of fine-tuning an existing KWS model on a few user-provided recordings, personalizing the application for a given user.

Our main demonstration is Honkling,[1] a novel

[1] http://honkling.ai

Proceedings of the 2019 EMNLP and the 9th IJCNLP (System Demonstrations), pages 91–96
Hong Kong, China, November 3 – 7, 2019. ©2019 Association for Computational Linguistics

in-browser KWS system running previous state-of-the-art models (Tang and Lin, 2018). We provide a set of comprehensive experimental results for the latency of an in-browser KWS system on a broad range of devices. We also evaluate the accuracy of KWS on various user accents and present a mechanism for in-browser user accent adaptation. On the Google Speech Commands dataset (Warden, 2018), our most accurate in-browser model achieves an accuracy of 94% while performing inference in less than 30 milliseconds. With only five user-recorded audio clips per keyword, Honkling can be fine-tuned to improve accuracy by up to an absolute 10%. Using hardware acceleration, users only need to wait for eight seconds before their Honkling becomes personalized.

2 Background and Related Work

KWS is the task of detecting a spoken phrase in audio, applicable to simple command recognition (Warden, 2018) and wake-word detection (Arik et al., 2017). Typically, KWS systems must be small footprint, since the target platforms are mobile phones, Internet-of-Things (IoT) devices, and other portable electronics. To achieve this goal, resource-efficient architectures using convolutional neural networks (CNNs) (Tang and Lin, 2018; Sainath and Parada, 2015) and recurrent neural networks (RNNs) (Arik et al., 2017) have been proposed, while other techniques make use of low-bitwidth weights (Fernández-Marqués et al., 2018; Zhang et al., 2017). However, despite the pervasiveness of modern web browsers on a broad range of devices and the availability of deep learning toolkits in JavaScript, a personalizable, on-device KWS system in web applications has not to our knowledge been explored.

In automatic speech recognition (ASR), the presence of user accents often degrades recognition quality (Huang et al., 2004; Humphries et al., 1996). Unfortunately, the solutions proposed in previous work vary depending on the underlying system, and little prior art exists using deep learning. The idea of fine-tuning neural networks to increase accuracy for a group of accents is found in Najafian et al. (2016); however, full ASR involves much larger datasets and models, and thus hours of extra training data are necessary for successful adaptation. Surprisingly, there has been little work on building KWS systems for user accent personalization.

3 Data and Models

For consistency with past results (Tang and Lin, 2018; Tang et al., 2018), we train our models on the first version of the Google Speech Commands dataset (Warden, 2018), comprising 65,000 spoken utterances for 30 short, one-second phrases. As with Tang and Lin (2018), we pick the following twelve classes: “yes”, “no”, “stop”, “go”, “left”, “right”, “on”, “off”, “up”, “down”, unknown, and silence. The dataset contains roughly 2,000 examples per class, including a few background noise samples of both man-made and artificial noises, e.g., washing dishes and white noise. As is standard in the speech processing literature, all audio is in 16-bit PCM, 16kHz mono-channel WAV format. We use the standard 80%, 10%, and 10% splits from the Speech Commands dataset for the training, validation, and test sets, respectively.

3.1 Input Preprocessing

First, for dataset augmentation, the input is randomly mixed with additive noise from the background noise set—this helps to decrease the generalization error and improve the robustness of the model under noise (Ko et al., 2015). Following the official TensorFlow implementation, we also apply a random time shift of $\textsc{Uniform}[-100, 100]$ milliseconds (ms). For feature extraction, we compute 40-dimensional Mel-frequency cepstral coefficients (MFCCs), with a window size of 30ms and a frame shift of 10ms, yielding a final preprocessed input size of 101×40 for each one-second audio sample.

3.2 Model Architecture

We use the `res8` and `res8-narrow` architectures from Tang and Lin (2018) as starting points, which represent the prior state of the art in residual CNNs (He et al., 2016) for KWS. In both models, given the input $\mathbf{X} \in \mathbb{R}^{101 \times 40}$, we first apply a 2D convolution layer with weights $\mathbf{W} \in \mathbb{R}^{C_{out} \times 1 \times (3 \times 3)}$ and a padding of one on all sides. This step results in an output of $\tilde{\mathbf{X}} \in \mathbb{R}^{C_{out} \times 101 \times 40}$, which we then downsample using an average pooling layer with a kernel size of (4×3). Next, the output is passed through a series of three residual blocks comprising convolution and batch normalization (Ioffe and Szegedy, 2015) layers. Finally, we average pool across the channels and pass the features through a softmax across the twelve classes.

	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	12	94.0	10	90.9
	MacBook Pro (2017)	Intel Iris Plus 650	Firefox	29	94.0	15	90.8
	MacBook Air (2013)	Intel HD 6000	Firefox	34	94.0	19	90.8
	Galaxy S8 (2017)	Adreno 540	Firefox	60	94.1	43	89.0
CPU	Desktop	i7-4790k (quad)	PyTorch	10	94.3	2	91.2
	MacBook Pro (2017)	i5-7287U (quad)	PyTorch	12	94.3	3	91.2
	Desktop	i7-4790k (quad)	Firefox	371	94.1	94	90.9
	MacBook Pro (2017)	i5-7287U (quad)	Firefox	361	94.0	107	90.8
	MacBook Air (2013)	i5-4260U (dual)	Firefox	485	94.0	115	90.8
	Galaxy S8 (2017)	Snapdragon 835 (octa)	Firefox	1105	94.1	265	89.0

Table 1: Latency (lat.; 90th percentile) and accuracy (acc.) results on different platforms for the `res8-*` models.

In the previous description, we are free to choose C_{out} to dictate the expressiveness and computational footprint of the model; `res8` and `res8-narrow` choose 45 and 19, respectively. In total, `res8` contains 110K parameters and incurs 30 million multiplies per second of audio, while `res8-narrow` uses 19.9K parameters and incurs 5.7 million multiplies.

4 Honkling

Training neural networks in JavaScript from scratch is ill-advised due to poorly optimized computation routines such as matrix multiplication. Therefore, we use the official PyTorch model implementations[2] at training time. At inference time, weights are transferred from PyTorch to a web application implemented in TensorFlow.js.[3] Since the official implementation utilizes LibROSA (McFee et al., 2015) for audio feature extraction, we instead use Meyda (Rawlinson et al., 2015), a JavaScript version of LibROSA. However, unlike Python, which is well suited for developing audio processing applications, in-browser JavaScript presents challenges in manipulating audio; for example, many browsers restrict the sample rate of input audio to 44.1kHz only. Due to such restrictions, the processed audio in JavaScript differ from MFCCs extracted by LibROSA. Therefore, we have patched Meyda comprehensively to minimize the mismatches.

Overall, we successfully enable KWS functionality in browsers without any server-side inference. Since the audio data is quickly processed within the browser, it is much more efficient than transferring data over the network for inference. Furthermore, users are now freed from security and privacy implications, such as eavesdropping of network traffic and collection of personal speech data.

Measured with our university WiFi connection, the average latency to Google servers is about 25ms with a standard deviation of 20ms. Network latency is much higher for transferring audio data. With a server written in Python, we measure an average latency of 481ms with a standard deviation of 183ms for one second of 16kHz mono-channel audio data. With in-browser inference, we achieve a pure client-side architecture that does not suffer from variable network latency.

The two main metrics for our KWS application are accuracy and inference latency. To be consistent with previous work, our experiments use the same test set. We conduct experiments on desktop, laptop, and smartphone configurations to demonstrate the feasibility of our system on a broad range of devices. These include the following: a desktop with 16GB RAM, an i7-4790k CPU, and a GTX 1080 Ti; two laptop configurations, the MacBook Pro (2017), with a quad-core i5-7287U CPU and an Intel Iris Plus 650 GPU, and the MacBook Air (2013), with a dual-core i5-4260U CPU and an Intel HD 6000 GPU; the Galaxy S8 as our smartphone configuration. Given this wide range of devices, we decide to include results from Firefox only, but Honkling is also available on other browsers. Since TensorFlow.js is GPU capable, experiments are conducted both with and without GPU acceleration, which can be toggled in the browser settings.

Table 1 summarizes 90th percentile latency and recognition accuracy results for both `res8` and `res8-narrow` on various devices. Note that results with the PyTorch implementation on our laptop and desktop setups are included to compare

[2] `http://honk.ai`
[3] `https://js.tensorflow.org`

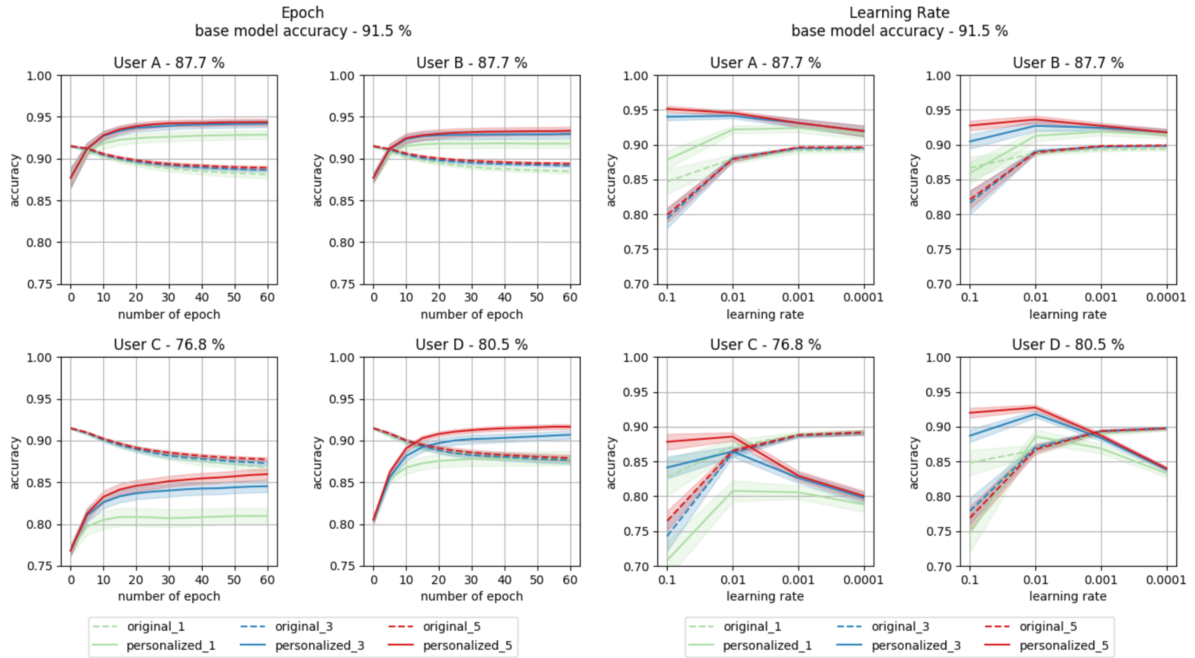

Figure 1: Accuracy varying the number of epochs (left) and the learning rate (right), with 95% confidence intervals (shaded). `original_*` and `personalized_*` denote accuracy on the original and user test sets.

with our in-browser configurations. The original implementation achieves an accuracy of 94.3% for `res8` and 91.2% for `res8-narrow` (see the first few rows in Table 1). Slight differences are observed across platforms due to a mismatch of MFCC computations between LibROSA and Meyda. However, the accuracy for each model is consistent on every platform, confirming that our in-browser implementation is robust.

Even though latency is processor dependent, the `res8-narrow` model performs inference in real time on every platform. Given that these delays are perceived by humans to be near instantaneous (Miller, 1968), our system is sufficient for real-time interactive web applications.

5 Personalization and Accent Adaptation

JavaScript applications run fully client-side and are cross-device compatible, meaning that they can be deployed directly on user devices, which enables seamless personalization. In this paper, to design a cross-device, cross-*human* KWS system, we extend Honkling to adapt to various user accents that are uncommon in the primarily American-accented dataset.

To measure the effects of different user accents on the KWS quality of Honkling, we evaluate the accuracy of `res8-narrow` on datasets comprised of recordings from different people. In these experiments, there are four participants: A, B, C, and D. Users A and B are native speakers of Canadian English while C has a British accent and D has a Korean accent. From each person, we collected 50 recordings for each of the twelve classes, setting aside 40 recordings of each class to construct a test set of 480 samples. We conduct evaluations on the same twelve classes.

To quantify the amount of effort required from the user for personalization, we begin by finding the minimum number of recordings that leads to improvements in accuracy. Next, we experiment with different numbers of epochs and learning rates. Since we have shown that our JavaScript implementation reproduces the PyTorch implementation with minimal differences, unless indicated otherwise the following experiments are conducted using PyTorch for convenience.

We report accuracy metrics under two different sets of conditions: `original_*` and `personalized_*`, denoting accuracy on the original test set and the user test set, respectively (see Figure 1, dashed and solid lines). For each condition, we conduct experiments with three variations of training data size. The number that follows each name denotes the number of recordings per keyword in the fine-tuning dataset. To reduce the effects of outliers, we report averaged results from 60 experiments with different ran-

	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
	MacBook Pro (2017)	Intel Iris Plus 650	Firefox	7.2 sec	12.6 sec	27.0 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
	MacBook Pro (2017)	i5-7287U (quad)	Firefox	29.5 min	86.3 min	139.2 min

Table 2: Average in-browser fine-tuning efficiency for `res8-narrow` under different configurations.

dom seeds, where the training data is constructed by randomly selecting the appropriate number of recordings from the fine-tuning set. The test set stays the same for each experiment.

Fine-tuning progress across epochs. In Figure 1, we include in the labels of the plots both the accuracy of the base model and the accuracy on each user test set. The left half of the figure shows the fine-tuning progress across epochs, with the learning rate fixed to 0.01. As expected, accuracy on the user test set is lower than on the original test set prior to fine-tuning. For users A and B, the differences are only a few percent, which seems acceptable in practice. However, the model achieves an accuracy of only 76.8% and 80.5% for users C and D, respectively, demonstrating the need for personalization.

As fine-tuning proceeds across epochs, accuracy on the user recordings increases while accuracy on the original data decreases. We observe diminishing returns with more epochs; 50 epochs seem to be sufficient to maximize accuracy. After convergence, the models generally achieve higher accuracy on the user recordings than on the original data, thus demonstrating successful adaptation. We find that a single recording per keyword is sufficient for personalization, as the fine-tuned models exhibit higher accuracy on the user recordings for every user except user C.

Fine-tuning dataset size. Since more training data leads to a better representation of a user's speech patterns, it is no surprise that an increase in accuracy is observed as more recordings are added to the dataset; compare `personalized_{1,3,5}` in Figure 1.

However, we find that the accuracy converges rapidly after a mere five recordings per keyword; such a trend is evident for every user. 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. Although using one sample per keyword helps, results suggest that having at least three recordings is desirable, since the marginal benefit of two more recordings is quite large; in user C's case (see Figure 1, bottom left), we observe an absolute improvement of almost 10 points.

Learning rate. In this experiment, we fix the number of epochs to 25 and perform grid search on the learning rate from 0.1 to 0.0001, stepping by a factor of ten—see the right half of Figure 1. Among the four different learning rates, we find that choosing 0.01 consistently leads to the best accuracy on user recordings for all three variations of training data size.

Efficiency and application evaluation. Bringing all the previous threads together, Honkling supports in-browser personalization by fine-tuning with user recordings. As we find that the number of recordings has a high correlation with the quality of personalization, users have the option to choose the number of recordings. Once recording is completed, the base model is fine-tuned with the best hyperparameter settings, for 50 epochs with a learning rate of 0.01. The fine-tuned model is then stored in the browser. At startup, Honkling loads the stored model if it exists so users can keep the application personalized even after the current browser session ends.

Table 2 summarizes the in-browser fine-tuning efficiency for `res8-narrow` on the 2017 MacBook Pro and our desktop. We average results over ten trials, where fine-tuning data is randomly selected from the four user datasets from the previous experiment. To be consistent with our previous study on in-browser inference efficiency, we conduct the experiments in Firefox. From the results, we find that, unsurprisingly, personalization time increases with the data size. 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 fine-tuning time. The same process that consumes up to 2.3 hours on a CPU can be completed within 27 seconds on a GPU. With a GTX 1080 Ti, only 8 seconds are necessary to achieve personalization with Honkling.

6 Conclusion

In this paper, we realize a new paradigm for serving neural network applications by implementing Honkling, a JavaScript-based, in-browser KWS system. On the popular Google Speech Commands dataset, our model achieves an accuracy of 94% while maintaining an inference latency of less than 30 milliseconds on modern devices. The purely client-side architecture allows our application to be efficient and cross-device compatible, with the additional benefit of supporting user personalization. Since many speech-based systems suffer from accuracy degradation caused by differences in accents and speaking styles, we support per-user personalization in Honkling. Based on a small-scale study, we observe a substantial increase in accuracy after spending only a short amount of time fine-tuning on a few sample recordings. These results pave the way for future web applications that seamlessly support built-in speech-based interactions.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

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

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

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. *CVPR*.

Chao Huang, Tao Chen, and Eric Chang. 2004. Accent issues in large vocabulary continuous speech recognition. *International Journal of Speech Technology*, 7:141–153.

Jason J. Humphries, Philip C. Woodland, and David J. B. Pearce. 1996. Using accent-specific pronunciation modelling for robust speech recognition. *International Conference on Spoken Language Processing*.

Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*.

Tom Ko, Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. 2015. Audio augmentation for speech recognition. *INTERSPEECH*.

Jaejun Lee, Raphael Tang, and Jimmy Lin. 2019. Universal voice-enabled user interfaces using JavaScript. *IUI*.

Brian McFee, Colin Raffel, Dawen Liang, Daniel P. W. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. 2015. librosa: Audio and music signal analysis in Python. *Python in Science Conference*.

Robert B. Miller. 1968. Response time in man-computer conversational transactions. *Fall Joint Computer Conference, Part I*.

Maryam Najafian, Saeid Safavi, John H. L. Hansen, and Martin Russell. 2016. Improving speech recognition using limited accent diverse British English training data with deep neural networks. *International Workshop on Machine Learning for Signal Processing*.

Hugh Rawlinson, Nevo Segal, and Jakub Fiala. 2015. Meyda: An audio feature extraction library for the web audio API. *Web Audio Conference*.

Tara N. Sainath and Carolina Parada. 2015. Convolutional neural networks for small-footprint keyword spotting. *INTERSPEECH*.

Raphael Tang and Jimmy Lin. 2018. Deep residual learning for small-footprint keyword spotting. *ICASSP*.

Raphael Tang, Weijie Wang, Zhucheng Tu, and Jimmy Lin. 2018. An experimental analysis of the power consumption of convolutional neural networks for keyword spotting. *ICASSP*.

Pete Warden. 2018. Speech Commands: A dataset for limited-vocabulary speech recognition. *arXiv:1804.03209*.

Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2017. Hello edge: Keyword spotting on microcontrollers. *arXiv:1711.07128*.