Code Offloading Solutions for Audio Processing in Mobile Healthcare Applications: A Case Study

Pablo Sanabria, Jose I. Benedetto, Andres Neyem, Jaime Navon
Pontificia Universidad Católica de Chile
Computer Science Department
Santiago, Chile
{psanabria,jibenede,aneyem,jnavon}@uc.cl

Christian Poellabauer
University of Notre Dame
Computer Science and Engineering Department
Notre Dame, IN, USA
cpoellab@nd.edu

ABSTRACT
In this paper, we present a real-life case study of a mobile healthcare application that leverages code offloading techniques to accelerate the execution of a complex deep neural network algorithm for analyzing audio samples. Resource-intensive machine learning tasks take a significant time to complete on high-end devices, while lower-end devices may outright crash when attempting to run them. In our experiments, offloading granted the former a 3.6x performance improvement, and up to 80% reduction in energy consumption; while the latter gained the capability of running a process they originally could not.

KEYWORDS
Mobile Cloud Computing, Mobile Cloud-Based Systems, Mobile Code Offloading, Cloud-Based Mobile Augmentation

ACM Reference Format:
Pablo Sanabria, Jose I. Benedetto, Andres Neyem, Jaime Navon and Christian Poellabauer. 2018. Code Offloading Solutions for Audio Processing in Mobile Healthcare Applications: A Case Study. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3197231.3197256

1 INTRODUCTION
Deep learning applied to audio processing, mostly in the context of speech recognition, has become commonplace in the industry ever since the availability of high-performing processors made the execution of very large neural networks feasible in reasonable amounts of time. However, the significant hardware requirements for loading and running these large models have, so far, limited their use to high-end servers or specialized mainframes. The most accurate (and therefore typically very large) neural networks are deemed unfit to be run on resource constrained mobile devices. Clients interested in taking advantage of them usually do so through a network enabled API.

Nevertheless, the increase of the hardware capabilities of mobile devices, and the increasing need of intelligent applications to function offline, are changing the trend towards enabling local execution of deep neural networks (DNN) on mobile platforms. In line with this paradigm shift, both Google and Apple have recently released their own frameworks (Tensorflow Mobile and CoreML respectively) to facilitate the integration of these algorithms into mobile applications. As a consequence, more and more software developers are releasing their own DNN enabled software with offline support. Yet despite these advancements, large DNNs hosted on mobile applications are still a rare sight. Mostly, we only see lightweight DNNs deployed on smartphones or tablets due to the high memory requirements for running larger models.

When depending on APIs hosted on web, we would argue that it is preferable for mobile applications to always fall back to a local implementation when connectivity is lost, even at the cost of reduced performance. End users would rather have limited functionality than none at all when offline. In this sense, network availability is not seen as a hard requirement for an application, but rather as a means for accelerating processes and reducing battery consumption. Applications built this way that make use of large DNNs would then have the best of both worlds, but correctly implementing such a system represents a significant engineering challenge.

In this paper, we present an Android-based mobile application that makes use of large DNN-based algorithms that may be fully executed locally when offline, or offloaded to a more powerful surrogate if a network interface is available. The original server-based DNN model used as the basis of our application could not be directly loaded on a mobile device due to its very high resource requirements. Therefore, it had to be ported to the Android environment by applying several optimizations. While the resulting application can be used both online and offline, the availability of network access significantly improves its performance and energy efficiency, thanks to the greater capabilities of the server counterpart. This code offloading behavior was achieved with very little additional software engineering effort as we made use of an existing code offloading framework that greatly simplifies the task of replicating the logic on the server.

Our contributions are twofold: first, we describe a process that reduces the resource consumption for large DNNs so we can run them locally on the device; and then we describe how this algorithm may be integrated with a code offloading solution to significantly increase performance whenever a network connection is available.

2 RELATED WORK

Using DNN algorithms to solve problems related to audio processing tasks, such as audio and speech recognition, is not new [4, 6, 10], but running this kind of algorithms on mobile devices is challenging and complex. Nevertheless, there do exist some optimizations and techniques that can be applied to DNN models to help reduce the required computational power in mobile devices [11, 13].

Alternatively, mobile platforms may leverage resources from the cloud in order to run DNN algorithms. There are several examples and studies where researchers attempted to run DNN applications to solve image and object recognition problems with the help of a cloud based system [3, 7, 8]. This approach has the disadvantage of being dependent on the network availability, but it can help to reduce energy consumption and execution time when it is available. Additionally, using a cloud based system grants the possibility of running these models on devices with reduced hardware capabilities.

By using a mobile code offloading platform, we can take advantage of both approaches: we can execute DNN models locally on mobile devices if we have no network connectivity and sufficient memory and processing power, or we can leverage cloud resources to reduce the execution time and energy consumption of the mobile device. We found that this approach is viable and is already used to solve other types of problems (e.g., augmented reality [14], object recognition [12]).

3 IMPLEMENTATION

3.1 Audio Processing Application

In this paper we present Contect, an early warning medical application built for the Android operating system that makes use of a deep learning algorithm to analyze a patient's speech and detect abnormalities indicative of possible neurological complications, such as traumatic brain injuries [5]. The workflow of the application consists of three steps: first, an audio sample is recorded based on standardized speech-language pathology tests; then, the sample is processed, transformed into a spectrogram and fed into a DNN; and finally, the application returns a concussion probability.

Contect's neural network is very complex as it tries to detect patterns that are very hard for human beings to notice. For this to work, our deep neural network architecture is based on the residual neural network concept proposed by He et al. [9]. The model consists of 10 parallel 50-layer deep ResNets, with two LSTM layers at the end of each of them (Figure 1). All ResNets are later merged into a series of fully connected layers in order to obtain the final output. In order to train the model, we used the Keras machine learning framework.

The main challenge when deploying a DNN to a mobile device is the device's memory limitation. Current high-end mobile platforms support around 2GB of memory, with only the newest models released from 2017 onwards supporting 4GB or more. This, coupled with the memory restrictions that are inherent to mobile operative systems (OS), means that applications executing these models exceeding or even approaching these thresholds are killed by the OS. This differs from desktop or server OS behavior where the existence of a backing store allows to temporarily swap memory onto disk to run a program under critical memory conditions at the expense of running time. As such, it is imperative to restrict memory consumption as much as possible when running on a mobile device.

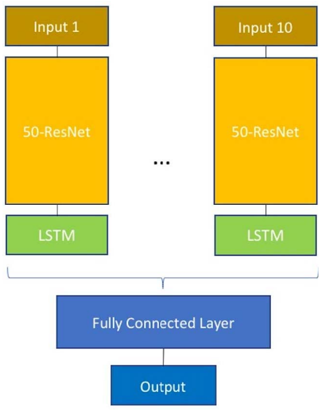

Figure 1: Overview of our DNN Architecture

A second challenge to consider is the resulting app size. Complex DNNs can easily reach many gigabytes in size when stored on disk. For Android, apps published to the Play Store may not exceed 100 MB in size if running on Android 2.3 or higher, but these apps may support up to two expansion files that can store arbitrary assets that are downloaded in conjunction with the main APK, bringing the maximum total app size to 4.1 GB. Despite this maximum app size, developers are strongly urged to minimize app sizes as much as possible.

Then, there is the matter of execution time. Simple neural networks often return results practically instantly and are suited for real time applications. DNNs on the other hand may require tens of seconds to process a single input sample. GPU support for machine learning frameworks on Android is currently very limited. The two most popular technologies for running deep learning algorithms on Android are Tensorflow Mobile and Tensorflow Lite. Tensorflow Mobile is the standard Tensorflow framework built for mobile architectures. It may support all kernels defined in Tensorflow when built with selective registration, but currently lacks GPU support. Tensorflow Lite on the other hand has been extensively optimized for mobile devices and thus features a smaller binary size, fewer dependencies and better performance than Tensorflow Mobile. In particular, it can make full use of an Android device's GPU through the neural network API introduced in Android 8.1. Unfortunately, it currently supports only a subset of Tensorflow Mobile's kernels and rarely will a Keras-built model be compatible with Tensorflow Lite. As such, we used Tensorflow Mobile for building our application with the subsequent penalty in performance.

3.2 Model Optimization

The original DNN model we built for processing audio samples was unfortunately unfit to be run on a mobile device due to its significant memory requirements. We therefore applied several optimizations to adapt it for mobile environments: stripping unused nodes and merging of constants, quantization and partitioning. While the first two techniques are rather well known and documented (there are even publicly available scripts that implement them automatically), the third one is a novel approach we introduced since the former optimizations were insufficient for our purposes.

3.2.1 Stripping Unused Nodes and Merging Constants. Neural network models usually include a variety of nodes and operations that serve a purpose during the training phase only. If we preserve these nodes in the inference phase, memory usage will slightly increase, so it is recommended to remove them if possible. Additionally, it is common for model definitions to be split in two sets of files: one that defines the model architecture and a second one that defines each node's weight. This distinction is important in the training phase, because the weights and biases vary when the neural network model is trained. However, this is no longer the case when doing inference. Model and weights can be merged into a single file, where all trainable nodes are replaced with constant nodes assigned with their respective weight values. This optimization results in a slightly smaller model size and lower loading times.

3.2.2 Quantization. Empirical evidence shows that weight definitions contribute by far the most to a model's on-disk representation. In our experiments, the weights of our built networks accounted for over 99% of the overall model size. It is also true that most neural networks are designed to be resistant to noise, so minor alterations in a node's weight value should not significantly alter a model's accuracy. It is therefore possible to alter a model's weight representation in a lossy way to reduce its size with barely any changes to its output.

Quantization is the process of rounding numerical values to an approximate representation with reduced precision. For all relevant nodes, the interval between the minimum and maximum weight value is divided into 2^N buckets, where N is the number of bits selected in the quantized representation. Next, all input weights W are replaced with an integer K between 0 and $2^N - 1$, such that:

$$K = \frac{W - w_{min}}{w_{max} - w_{min}} * 2^N$$

A mapping for all nodes and their respective minimum and maximum weight values is kept so that the original weight can be approximately restored from the quantized representation.

Most models are trained using 32-bit floating point numbers. Reducing their precision to 16 or 8 bits allows us to reduce model sizes by 50% and 75% respectively with negligible loss of accuracy. Not only will this reduce the application's size, but our experiments show that significant memory savings can be achieved in this fashion. In our particular case, we used 8 bit quantization.

3.2.3 Model Partitioning. By default, neural network inference engines operate by loading an entire model into memory and running the entire input set layer by layer until we obtain an output. It is an efficient mechanism that puts performance first at the cost of higher memory requirements. Model partitioning refers to determining a partition of a model's underlying graph so that instead of executing the entire model at once, every subgraph in the partition is run sequentially, with memory being collected in between. Intermediate results are preserved and later fed to the subsequent iterations. Doing this can lead to significant memory savings at the cost of a higher execution time, as long as the number of tensors to keep track of in between iterations is small. If a model's memory requirements exceed the host platform's limits, partitioning is a good alternative if a proper balance between memory consumption and execution time is attained. In our particular case, we partitioned our model into 11 different subsets: one for each ResNet, and a final one for the fully connected layers.

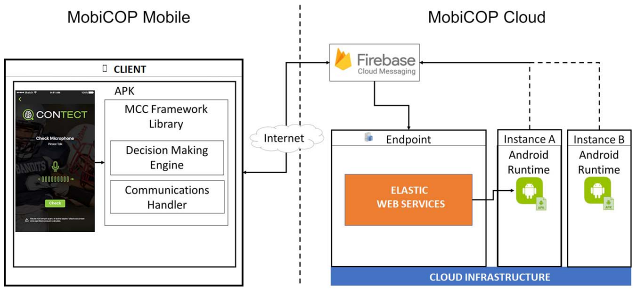

Figure 2: MobiCOP architecture overview

3.3 Code Offloading Platform

Code offloading is a popular cloud computing technique by which tasks are transparently migrated from a resource constrained environment to a resource rich offsite surrogate that may complete said task more efficiently. Code offloading may be implemented in mobile devices through one of various mobile code offloading frameworks available in academia. In this paper, we used the MobiCOP framework to add code offloading capabilities [1] [2].

MobiCOP consists of two major components: an Android library designed for the client to define arbitrary tasks to offload, and a server component in charge of executing the offloaded tasks and relaying the results back to the client. An overview of the architecture is shown in Figure 2. MobiCOP was designed to run long-running background tasks in modern Android applications. It isolates offloaded tasks into small components through an application programming interface analogous to Android services. Its design grants it compatibility with stock Android devices without the need to modify the mobile OS, in contrast to most other mobile code offloading frameworks currently available.

In Contect, the flow of an offloaded task starts when the audio is recorded and converted into an spectrogram. A decision engine, based on network quality and previous executions, determines if the task should be run locally or in the cloud. If the task is to be offloaded, a communication layer first sends the input parameters to the cloud (in this case the speech sample spectrogram), and then a server-based Android x86 environment replicates the DNN algorithm defined by the client. Once the task is completed, a push notification including the concussion probability is sent back to the client through Firebase Cloud Messaging and lastly the end-user is notified of the result.

4 EXPERIMENTATION

We measured both performance and energy consumption to test in three different configurations:

- An Lenovo A319 with a dual-core 1.3 GHz Cortex-A7 CPU and 512 MB of RAM.

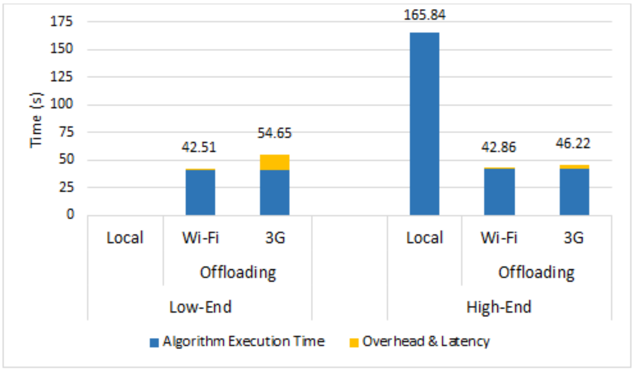

Figure 3: Mobile application performance results

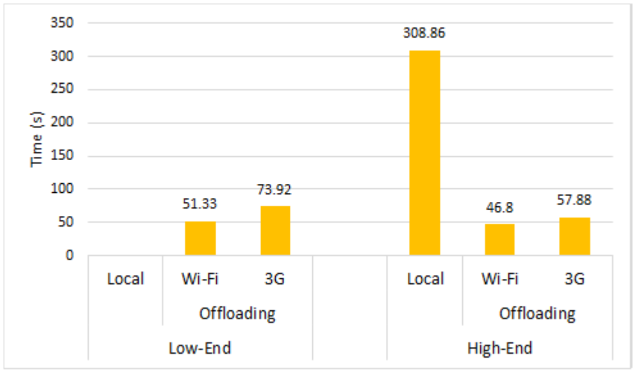

Figure 4: Mobile application battery consumption results

- A Samsung Galaxy S5 with a quad-core 2.5GHz Krait 400 CPU and 2GB of RAM (high end device) for local execution.
- Either device with the usage of our platform with the cloud component running Genymotion for AWS on a c4.2xlarge instance.

For measuring performance, we measured the time it takes to process a given audio sample and return the concussion probability. For measuring energy consumption, we used a Monsoon Power Monitor to collect real-time precise battery usage readings on the devices through the tasks execution. As the network is a determinant factor in the quality of an offloading solution, we considered a network connection through both Wi-Fi and 3G in our experiments.

The original Keras-built DNN model required about 4GB of memory to be loaded on a server, and about 7 GB to be loaded on a mobile device due to the lack of GPU support. Through our various optimizations, we managed to reduce this amount to 1.3 GB, enough to be run on our high-end device, but still insufficient for low-end devices. We therefore omit local execution times for low-end devices in our results, as loading the DNN in that context would crash the application. Our experiments show that by migrating the task to the cloud, performance improves by a factor of up to 3.6 times, and energy consumption is reduced by up to 5 times on a high-end device. The results are shown in figures 3 and 4.

5 DISCUSSION AND CONCLUSIONS

In this paper, we presented a real-life case study that shows the advantages in both performance and energy consumption of code offloading when applied to applications with DNN algorithms. DNNs are seldom found locally on mobile applications, but this trend is slowly starting to change. So far, DNN models commonly found on popular mobile apps are mostly lightweight. However, we show through Contect that this is not an absolute necessity. Larger models may be adapted to run fairly well on constrained mobile devices and produce interesting results, despite relatively high wait times for inference. But even though this last constraint is still an issue, it can be greatly amortized through the use of code offloading. Not only does it allow to increase performance and battery, but even adds compatibility with lower end devices with insufficient hardware requirements.

Future work involves adding edge computing support to our offloading framework to further increase performance by leveraging the capabilities low-latency proximate servers.

ACKNOWLEDGMENTS

This work was partially supported by a DCC-UC research grant, the CONICYT-PCHA/National Ph.D./2016 Grant no. 21161015, and AWS Cloud Credits for Research. The authors would also like to thank the author of the Contect application, Bryan (Ning) Xia, for granting them access to their code base.

REFERENCES

[1] Jose I Benedetto, Andres Neyem, Jaime Navon, and Guillermo Valenzuela. 2017. Rethinking the mobile code offloading paradigm: from concept to practice. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 63–67.
[2] Jose I Benedetto, Guillermo Valenzuela, Pablo Sanabria, Andres Neyem, Jaime Navon, and Christian Poellabauer. 2018. MobiCOP: A Scalable and Reliable Mobile Code Offloading Solution. *Wireless Communications and Mobile Computing* (2018), 18 pages.
[3] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 155–168.
[4] Li Deng, Geoffrey Hinton, and Brian Kingsbury. 2013. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 8599–8603.
[5] Michael Falcone, Nikhil Yadav, Christian Poellabauer, and Patrick Flynn. 2013. Using isolated vowel sounds for classification of mild traumatic brain injury. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),*. IEEE, 7577–7581.
[6] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 6645–6649.
[7] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 68–81.
[8] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 123–136.
[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[10] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech

recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.

[11] Xin Lei, Andrew W Senior, Alexander Gruenstein, and Jeffrey Sorensen. 2013. Accurate and compact large vocabulary speech recognition on mobile devices.. In *Interspeech*, Vol. 1.

[12] Dawei Li, Theodoros Salonidis, Nirmit V Desai, and Mooi Choo Chuah. 2016. DeepCham: Collaborative Edge-Mediated Adaptive Deep Learning for Mobile Object Recognition. In *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 64–76.

[13] Mohammed Kyari Mustafa, Tony Allen, and Kofi Appiah. 2017. A comparative review of dynamic neural networks and hidden Markov model methods for mobile on-device speech recognition. *Neural Computing and Applications* (2017), 1–9.

[14] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. 2017. Delivering Deep Learning to Mobile Devices via Offloading. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*. ACM, 42–47.