proceedings

MDPI

Proceedings

Deep Neural Networks on Mobile Healthcare Applications: Practical Recommendations †

Jose I. Benedetto [1,*], Pablo Sanabria [1], Andres Neyem [1], Jaime Navon [1], Christian Poellabauer [2] and Bryan (Ning) Xia [2]

[1] Computer Science Department, Pontificia Universidad Católica de Chile, Santiago 7820436, Chile; psanabria@uc.cl (P.S.); aneyem@uc.cl (A.N.); jnavon@uc.cl (J.N.)
[2] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA; cpoellab@nd.edu (C.P.); nxia@nd.edu (B.N.X.)
* Correspondence: jibenede@uc.cl
† Presented at the 12th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2018), Punta Cana, Dominican Republic, 4–7 December 2018.

Published: 24 October 2018

Abstract: Deep learning has for a long time been recognized as a powerful tool in the field of medicine for making predictions or detecting abnormalities in a patient's data. However, up until recently, hosting of these neural networks has been relegated to the domain of servers and powerful workstations due to the vast amount of resources they require. This trend has been steadily shifting in the recent years, and we are now beginning to see more and more mobile applications with similar capabilities. Deep neural networks hosted completely on mobile platforms are extremely valuable for providing healthcare services to remote areas without network connectivity. Yet despite this, there is very little information regarding the migration process of an existing server-based neural network to a mobile environment. In this work, we describe the various techniques and considerations that should be taken into account when developing a deep-learning enabled mobile application with offline support. We illustrate the above by providing a concrete example through our experience in migrating to mobile an in-house developed medical application for detecting early signs of traumatic brain injuries.

Keywords: machine learning; mobile devices; mobile healthcare; deep learning; keras; tensorflow

1. Introduction

The dramatic expansion of mobile technologies in the world has had a significant impact in our way of life. Estimates indicate that by 2018, over 2.53 billion people will own a smartphone (approximately one third of the world's population). Smartphone global shipments currently exceed 1.4 billion units shipped every year and it is estimated that this number will increase to 1.7 billion by 2020 [1,2]. This change has radically changed many facets of our way of life, including the field of healthcare provisioning. M-health technologies have already altered the way healthcare is delivered, with smartphones enabling delivery of low-cost healthcare assistance to parents, the elderly and residents of remote and/or unconnected areas [3].

Up until recently, mobile healthcare applications were rather limited in their functionalities by the limited capabilities (CPU, memory) of general-purpose mobile devices. It is rare for applications to offer more in the way of healthcare than catalogues of information, passive monitoring and simple reminders. Yet, with the availability of more powerful mobile platforms and APIs that permit developers to harness all the capabilities of the hardware, it is now starting to become more common to see smarter applications.

Deep neural networks (DNN), also known as deep learning, are of particular interest to mobile healthcare applications, as they allow to find patterns useful for early warnings or diagnosis amongst clusters of highly irregular, sparse and multi-dimensional data. The effectiveness of neural networks in healthcare applications has long since been recognized [4–6], however, most implementations found in academia today are limited to standard artificial neural networks (ANN), often featuring only a couple of hidden layers. Deep learning differs significantly from standard ANNs in their number of hidden layers, their connections and the architecture's complexity. This allows them to learn more meaningful abstractions that may be applied to solve a wide variety of problems. Perhaps one of the most interesting demonstrations of the applicability of deep learning in medical healthcare is Google's AI retinal scanner showcased at Google IO 2018, where a deep learning implementation proved itself capable of predicting various pathologies based on nothing but a patient's retinal scan [7]. Deep learning applied to healthcare is rarer that ANNs, even more so when it comes to mobile applications. This is due to both the novelty of the technology and the difficulty in implementing such complex models in resource constrained mobile platforms. Yet precedent does exist, and so far deep learning has proven itself capable of delivering novel healthcare capabilities on stand-alone devices without network connectivity [8].

To date, the most accurate and complex DNN models are relegated to the domain of powerful server environments due to the vast amount of resources they consume. Mobile applications providing deep neural network capabilities usually act as a thin client to a DNN hosted on a server. The few available DNN-enabled mobile applications with offline capabilities make use of heavily optimized, relatively light, neural networks, which in several cases have to sacrifice significant amounts of accuracy in their efforts to remaining mobile-friendly. Offline functionality is of particular importance to mobile apps as it allows them to function in remote and underground locations with poor or no network connectivity. Moreover, it allows them to save the end-user from significant amounts of network traffic, therefore also leading to potential monetary impacts.

In the past, DNNs were rarely considered as an option for execution on mobile devices. This has changed recently with the release of more and more powerful mobile devices from industry giants such as Apple, Google, and Samsung, and the surge of a trend to facilitate the local hosting and execution of complex machine learning (ML) models on mobile devices through official promotion from the aforementioned companies and the release of new software optimized for mobile ML [9,10]. This trend was particularly highlighted with the announcement of TensorFlow Lite and a Neural Network API for Android (at Google IO 2017), and the Core ML framework for iOS (at WWDC 2017).

Yet despite this paradigm shift, there is still a lack of understanding of the limits of this technology. As such, developers are unsure up to what point it is possible to migrate a large DNN model to a mobile platform, and the implications of such migration. There is also a lack of knowledge of the various existing techniques to optimize arbitrary DNN models for mobile environments.

In this paper, we seek to provide detailed insight into the technical aspects involved in the migration process of arbitrary DNN models to mobile environments, for both Android and iOS. Furthermore, we illustrate these concepts through a practical implementation of a mobile healthcare application that makes use of deep learning to analyze a patient's speech and provide early warnings of possible concussion symptoms following a trauma to the head.

2. Related Work

There are many fields in healthcare where deep learning has proven to be particularly effective. One of the earliest areas in which this technology was applied was in clinical imaging. In [11], MRI scans are analyzed by a deep neural network for the early detection of Mild Cognitive Impairment (MCI) and Alzheimer's Disease (AD). Deep learning was similarly used in the analysis of lung CT scans and breast ultrasound images for detecting benign and malignant nodules or lesions [12]. Deep learning has also shown success in analyzing patients' electronic medical records. DeepCare [13] is an example of such a technology, wherein a Recursive Neural Network (RNN) with Long-Short Term Memory (LSTM) hidden units is used to predict a patient's future medical trajectory based on his past medical history. Genomics has also seen wide application of deep neural networks. For example,

DeepBind [14] is a framework that makes use of Convolutional Neural Networks (CNNs) to predict sequence specificities of DNA and RNA binding proteins.

A common trend amongst all of these applications is that they all operate on either servers or powerful workstations. Despite the wide variety of applications of deep learning in the field of medical healthcare, related mobile applications are rare, with surprisingly few results published in academia. A couple of interesting applications we found are: a mobile-hosted CNN that takes input from triaxial accelerometers and heart-rate sensors to predict Energy Expenditure (EE), a valuable metric for preventing chronic diseases [15]; and an Android application that identifies skin diseases by taking a picture of a suspected skin abnormality with the smartphone's camera, and then processing the image with a shallow neural network hosted on-device [16]. The latter example however makes use of an ANN, so it does not fully qualify as deep-learning.

The lack of results in this area, despite the usefulness of deep learning in providing robust mobile diagnosis tools, is mostly due to the difficulty in porting a server-based DNN to a mobile environment, and the novelty of mobile hardware powerful enough to run them, and the software to properly make use of it. There do exist however some publications that address the issue of migrating large DNNs to mobile.

Chen et al. introduce a framework for migrating deep neural networks to iOS devices while reducing execution time and the overall neural network size with negligible loss in accuracy. They accomplish this by maximizing data reusability and pruning redundant kernels [17]. Their work is focused on execution time speedups; as such, results are limited to relatively light neural networks (16-layer deep CNNs in their examples). In [18], the authors shift their focus to energy optimizations for migrating deep neural networks into generic embedded systems. They do this by applying redundant connection pruning, weight sharing and quantization techniques. Quantized CNN [19] also focuses on the benefits on quantization, with reports of up to 6× speed-ups and 20× compression rates, with less than 1% accuracy loss. MCDNN is a neural network execution frameworks that introduces a compiler for optimizing models, and client/cloud runtime that allows for resource sharing between multiple DNNs across different contexts [20]. Following a similar idea, MoDNN [21] focuses on local partitioning of already trained DNNs across multiple devices to obtain speedups.

Our contribution in this paper differs from all the above in that we focus on models inherently too large to even be run on a mobile environment. We then seek to optimize them with the main focus being memory reduction, even at the cost of execution time, to get them to run on stock mobile devices available in today's market. As such, we mainly aim to provide production-ready DNN-enabled mobile applications based on existing deep learning models.

3. The Migration Process

The main challenge when deploying a deep neural network to a mobile device is the device's memory limitations. Most low to mid end mobile devices available in the market today feature around 1 to 2 GB of RAM, with only the latest models released from 2017 onwards supporting up to 6 GB of RAM. This, coupled with the memory restrictions inherent to mobile operating systems, means that programs executing these models exceeding or even approaching these thresholds are killed by the OS. Empirical evidence shows that for mobile operating systems, programs are extremely likely to being pre-emptively killed, even when running in the foreground, whenever they use more than a certain threshold that varies per device (usually, a percentage of the maximum amount of RAM, although this value varies per device and per OS). Even when remaining below that number, we still noticed rare cases when the background threads doing the heavy lifting would get killed. This differs from desktop and server OS behavior where the existence of a backing store allows to temporarily swap memory into 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 mobile.

A second challenge to consider is the resulting app size. Complex deep neural networks can easily reach many gigabytes in size when stored on disk. When included on a mobile software package, there is very little compression that can be achieved and for the most part, the model's

original size will be added to the resulting app size. Developers are strongly recommended to minimize app sizes as much as possible in whatever platform they may be working on in order to improve user experience.

Then, there is the matter of execution time. Simple neural networks often return results practically instantly and are suited for real time applications. Deep neural networks on the other hand may require tens of seconds to complete a single input sample. Current multiplatform neural network technologies do not support all mobile devices' GPUs (although there do exist iOS and Android exclusive technologies that support hardware accelerated neural networks). Consequently, all processing must be done exclusively on the CPU, thereby requiring orders of magnitude more time to complete than its server counterpart.

Finally, mobile compatible versions of the aforementioned multiplatform neural network APIs feature several optimizations meant to accelerate mobile computation. By doing so, they actually exclude support for several operations in the full version and may not be able to run arbitrary models that were not originally targeted for mobile environments. Careful consideration must be made to make sure the mobile neural network library includes support for all operations defined in the model of interest.

Next, we discuss several methods we have successfully applied and tested that allow us to tackle these challenges: stripping unused nodes and merging of constants, selective registration, quantization and partitioning. While the first three techniques are rather well known and documented (there are even publicly available scripts that implement them automatically), the last one is a novel approach we hereby introduce. From here onwards, we will refer to the set of the first three optimizations as: Standard Deep Learning Optimizations for Mobile.

3.1. Stripping Unused Nodes and Merging of 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. Selective Registration

Neural network frameworks can be compiled with support for a variety of different operations (in TensorFlow, these are referred to as Kernel Ops). The mobile distribution of the frameworks may strip off certain operations to limit the resulting library size. This has two important consequences: the default mobile distribution may lack support for certain operations included in certain models, therefore being impossible to run on mobile devices; and lastly, the framework's library may include definitions for several operations that are not included in a particular model, thus resulting in wasted space. Selective registration (Figure 1) is the process by which a deep learning framework library is built with support for a specific subset of operations. Developers may use scripts to enumerate all operations defined in a specific set of models they intend to load, and then build the deep learning library with support for only those operations in particular. This significantly reduces the library's binary size, while also ensuring full compatibility with the related models.

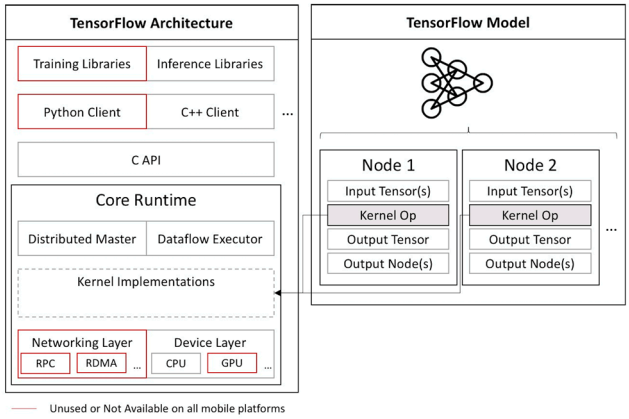

Figure 1. Overview of a TensorFlow Library built with Selective Registration.

3.3. 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:

$$K = \frac{W - \text{min\_weight}}{\text{max\_weight} - \text{min\_weight}} \cdot 2^N$$

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.

3.4. 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.

4. Evaluation

In order to evaluate these optimizations, we sought to migrate a complex deep neural network developed for medical purposes. The DNN in question has been trained to detect abnormalities in a patient's speech stemming from traumatic brain injuries (TBI). It was originally developed for server environments with the Keras framework, using Tensorflow as backend. At first, a light mobile client

application was intended to make use of this model through a network-enabled API, but this approach proved problematic in various scenarios where no network connection is available (such as in remote locations with no mobile network connectivity, and facilities with no Wi-Fi hotspot in range).

4.1. Contect

Contect is an early warning medical application built natively for the Android and iOS operating systems that makes use of a deep learning algorithm to analyze a patient's speech and detect abnormalities indicative of possible neurological complications [22,23]. The workflow of the application consists of three steps: first, an audio sample is recorded based on standardized speech-language pathology (SLP) tests; then, the sample is processed, transformed into a spectrogram and fed into a DNN; and finally, the application returns a concussion probability. Contect features five types of SLP tests: enunciation of multisyllabic words, enunciation of sentences with consonant and vowel phonemes with front and back articulation, repetition of sentences while applying stress and intonation in different words, rapid repetition and alternation of the "Pa", "Ta", and "Ka" phonemes to assess diadochokinetic alternating motion rate and diadochokinetic sequential motion rate, and sustained vowel enunciation to capture voice quality and tremor. A signal-to-noise ratio analysis is then conducted on the resulting audio samples to discard those with significant environmental noise. Screenshots of the application can be seen in Figure 2.

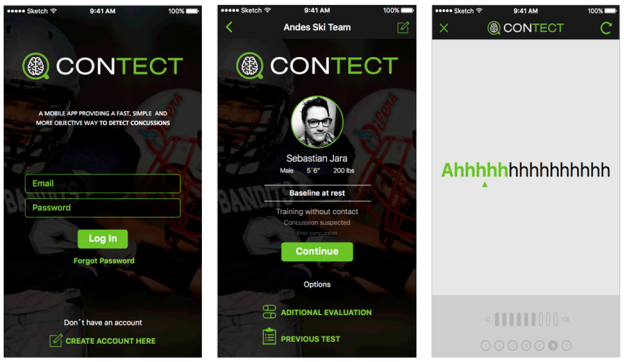

Figure 2. Contect Screenshots.

Contect's deep neural network architecture is based on the residual neural network concept proposed by He et al. [24]. The model consists of 10 parallel 50-layer deep ResNets, with two LSTM layers at the end of each of them (Figure 3). All ResNets are later merged into a series of fully connected layers in order to obtain the final output.

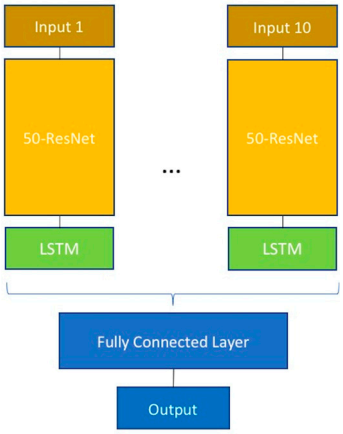

Figure 3. Overview of Contect's Deep Neural Network Architecture.

In total, the resulting optimized neural network features 25,474 nodes and 252,739,743 weight parameters.

4.2. Results

We now present the results obtained when running this model with all the aforementioned optimizations under TensorFlow version 1.1 (Table 1). We consider the following cases: no optimizations whatsoever; all standard deep learning optimizations for mobile, but no partitioning; and all optimizations, including partitioning. For quantization, we used 8-bit quantization; and for partitioning, we divided the execution of our neural network into 11 sequential steps: one per each residual neural network and one final step for running the fully-connected layer. Further partitioning is possible by splitting each individual residual neural network into multiple subgraphs.

The following results consider the maximum RAM consumption and execution time for a single inference. Test execution times are split into three phases: graph loading time (the time it takes to load the neural network model into memory and initialize a TensorFlow session with it), inference time (the time between feeding the input and getting an output tensor back) and general overhead. In our experiments, the latter is mostly due to setting the input tensors. Results were averaged for 10 distinct test runs in each case. While it may be possible to further reduce overhead times by applying different engineering strategies and more efficiently setting the tensors, there is very little improvement possible regarding the other two metrics. Also, do note that TensorFlow has been optimized for batch processing, therefore the processing time for multiple input sets is not proportional to their number.

Table 1. Results when Running our Deep Neural Network under Different Scenarios.

	Optimizations		**Memory Allocated (MB)**	**Execution Time**			**IPA/APK Size (MB)**
	Standard Deep Learning Opt. For Mobile	**Partitioning**		**Tensorflow Graph Loading Time (s)**	**Inference Time (s)**	**Total Execution Time (s)**	
Computer	×	×	4397.6	122.91	3.79	126.7	N/A
iOS Simulator	×	×	6677	2.518	11	24.598	1003
	√	×	3614	0.8821	7.997	20.292	295.5
	√	√	1172	7.9095	8.464	30.906	295.5
iOS Mobile Device	√	√	1087.9	9.965	19.106	38.321	255
Android Mobile Device	√	√	1320	46.458	22.976	77.402	252

Additionally, results also include the final app size for each case. We show these values for the original model running on a computer with no optimization through the Keras interface (control group), the model with and without our optimizations on an iOS simulator, and finally, the model with all optimizations running on both an iPhone 6S mobile device and a Samsung Galaxy S7 smartphone. The simulator used was a 9.7-inch iPad simulator running on a MacBook Pro with 16 GB of RAM and a 2.6 GHz quad-core Intel Core I7 processor. Both the iPhone mobile device and iOS simulator were running iOS 10.3.3, while the Android mobile device was running Android 7.0. Results for the model running without optimizations on an actual mobile device are excluded as memory requirements would exceed those on our test devices (2 GB), therefore getting the application killed ahead of time and making it impossible for us to obtain meaningful results. Results on an Android emulator without optimizations are also excluded as we were unable to successfully execute this scenario, most likely due to the inherent differences between simulators and emulators.

It is to be noted though that while running on the simulator, many test runs were still unsuccessful due to the simulator non-deterministically killing their respective background threads ahead of schedule, even though the computer's memory was adequate. Only successful test runs are considered in the results.

One of the first surprising results we found was the large amount of time it takes a desktop computer to obtain a result. For some reason, it takes a considerable amount of time for Keras to load a complex model for the first time. In our experiments, less than 4% of that time corresponds to the inference process. This situation is reverted on mobile environments, where most of the time is spent on inference. We have to lay stress on the fact however that we are using the Keras high level API for desktop, while only pure TensorFlow for mobile devices and it is possible that Keras introduces an additional overhead that we are unaware of.

The first important thing to notice is that running the models "as is" on a mobile environment without any optimization whatsoever results in a significantly higher memory consumption (around 50% more). Most likely this is due to the lack of GPU support on mobile for the technologies we are evaluating.

Overall, we can see that by applying the standard deep learning optimizations for mobile we can achieve a steady reduction of all metrics: the maximum memory consumption is reduced almost by a factor of almost two, and total execution time is reduced by around 20%. App size is also significantly reduced to a quarter of its original size, mostly due to the benefits of quantization. However, this is far from sufficient to achieve replication on a 2 GB mobile device. Partitioning allows us to reduce memory consumption even further at the cost of increasing the execution time. By doing this, memory consumption is further reduced by 70%, but CPU execution time suffers in consequence and is increased by 50%. It has to be noted however that by partitioning a given model into additional subsets, memory consumption could be further improved at the cost of additional time, therefore a proper balance has to be found by the developer.

Only this final model proved to be suited for execution on mobile devices with 2 GB of RAM. Nevertheless, we also noticed additional differences in our metrics when moving to physical devices, compared to those obtained on a simulator. When deploying to iOS, we noticed a further 8% reduction in memory consumption, which is quite positive for our purposes; however, overall execution times suffered due to the lower specifications of the iPhone's processor and increased by 26%. Surprisingly though, these results were substantially better than those found for the Android platform, considering TensorFlow was developed by Google: on a Samsung Galaxy S7, maximum memory consumption ended up 22% higher than in an iPhone 6S and total execution times actually doubled.

4.3. A Closer Look at Partitioning

We have just analyzed the benefits of one particular partitioning scheme applied to our deep neural network, however, in practice rarely will the partitioning strategy be self-evident. By splitting the neural network's graph into more and more subgraphs, memory consumption will indeed decrease, but at the same time, the time it takes to complete a single inference will also increase. Developers should strive to find a proper balance between these two factors. To give insight into this problem, we provide an analysis of how these two properties correlate through running our deep neural network under different partition strategies.

To this end, we evaluated our model's behavior when run under increasing amounts of sequential steps. For every different strategy, we show the evolution of the overall execution time to run a single sample and the maximum memory allocation (Figure 4).

Our experiments show there is no significant difference in total inference times when increasing the partition size. On the other hand, graph loading times are proportional to the number of items in the partition. This is to be expected as the graph needs to be flushed and reloaded for every iteration in order to recover its allocated memory. At the time of writing, we have yet to discover a more efficient way to do this in TensorFlow. For overhead times, we noticed a small increase in between the first and last partition strategies (~27%). While not negligible, it pays a very minor role in overall time increases when compared to graph loading times. Even more so when considering these values may be further reduced with more engineering. In contrast, memory consumption appears to follow an exponential decrease pattern. Overall, the greatest gain in memory consumption when compared

to the increase in execution time occurs with a size 2 partition. From there onwards, relative gains become less and less significant.

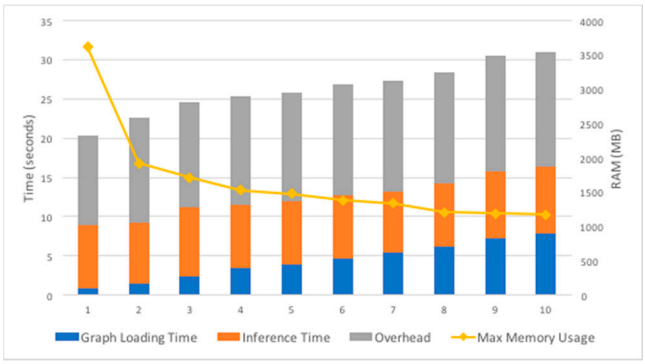

Figure 4. Evolution of Running Time and Memory Consumption when Increasing Partition Size.

4.4. The Effect on Battery Life

While running a deep learning model locally is very beneficial for usability purposes, it does carry with it a cost in terms of battery life. Deep learning inference can be a very process-intensive task; therefore, energy consumption will be affected. In order to gain an insight into the magnitude of this effect, we compared the energy consumption of an Android mobile device invoking a web service that runs our deep learning model on a server, versus that of the same device executing our ported model locally. For this experiment, we used a Samsung Galaxy S5 device with Android Nougat, while energy measurements were carried out with a Monsoon Power Monitor.

Results show that a running our model locally took about 260 Joules, while offloading the same model to a server only took about 6.5 Joules (Figure 5), about 40 times less energy. When porting deep learning models to mobile, developers will need to assess if the benefits of offline availability outweigh the costs in terms of energy.

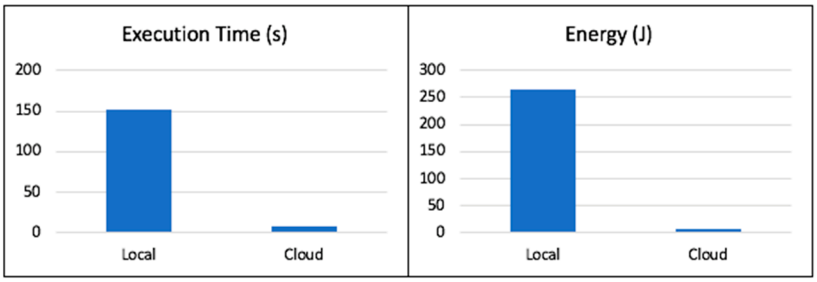

Figure 5. Running Time and Energy Consumption Comparison between Local Execution and Cloud Execution.

5. Lessons Learned

One of the first difficulties we experienced in this project occurred when trying to port our Keras-built model to a mobile environment. While Keras itself is backed by the multiplatform TensorFlow framework, which offers support for iOS, Android and Raspberry Pi, the binary library of the framework distributed for mobile environments is far from being equivalent to its desktop counterpart. Many kernels and data types have been removed from the mobile library to reduce the library size and encourage the execution of models optimized for mobile. As such, models including any of these restricted operations would fail to be loaded by the mobile framework. The library itself would need to be modified at compile-time in order to include support for all operations defined in a given model. Fortunately, selective registration handles this task for us automatically. While we must emphasize that it is always desirable to train models with the specific operations included in the default mobile library to boost performance, our approach will enable developers to run arbitrary models without the need for re-architecting it for a different environment, therefore significantly

reducing time to market. Even though this will come at a certain cost in performance, sometimes this cost will be acceptable.

Next comes the matter of the mobile OS behavior of pre-emptively killing processes when running short of memory. While the general behavior is well understood, the specific thresholds at which the different OS do so is not well documented, so there is always the uncertainty of whether a long-running task requiring large amounts of memory will finish running before the OS decides to kill it. We noticed distinct behaviors on both iOS and Android. On iOS, one of the first things we noticed is that it is possible for the OS to kill individual threads instead of an entire process if the memory consumption is not critical enough. In our experiments, the likelihood of a thread using over 65% of a device's memory being killed was exceedingly high, but even when limiting memory consumption to around 55%, there would still be rare occasions where the process would not be allowed to complete. On Android on the other hand, we need to distinguish two runtime environments: the Java (Dalvik/ART) runtime environment and the native runtime environment accessible through the NDK. On Java, memory limitations are rather tight and depend on a per app heap limit that varies according to the device's screen size and pixel density (curiously, it does not depend on the device's total available memory). As an example, on a Samsung Galaxy S7 running Android 7.0, the maximum heap size is 256 MB. It is however possible to increase this limit by specifying a "large heap" flag on an application's manifest. While the specifics of the memory increase are not documented, in our experiments we noticed the addition of this flag would double the maximum heap size on our setup. By contrast, the Android's native environment allows us to bypass these limitations as long as memory consumption does not exceed the limits imposed by the OS. These limits are again not documented, but in our experiments, we noticed that applications would get killed with a very high likelihood whenever the sum of all allocated memories (native + Java + code, etc.) would exceed 70% of the device's total memory without the overhead of other applications running on the background in parallel.

In conclusion, to guarantee stability in both platforms, we recommend keeping memory consumption below 50% of the total device's memory and to introduce a routine in code that checks if the relevant task managed to complete. That would allow the app to restart or postpone it, on the off chance the thread was killed prematurely by the OS. In the specific case of Android, it is recommended to handle memory intensive tasks with the NDK to bypass ART or Dalvik's limitations and to try to stay below the 50% of total device memory threshold for the sum of all categories of memory.

Finally, despite our optimizations, deep neural network models of our complexity can easily weigh various hundreds of megabytes. If the models were to be included in the official app releases, end-users are likely to be displeased due to the heavy toll they may take on their internal memory, as the public is yet unaccustomed to business apps this large in size. It is therefore recommended not to include these models in the original application, and instead distribute them through a separate and optional download. An online-only approach could still be included for those end-users who are unwilling to perform these downloads.

6. Conclusions and Future Work

In this study, we introduced a detailed analysis into various optimization techniques for migrating arbitrary large DNNs from server to mobile environments. This includes extensive insight into how various metrics (execution time, memory usage, app size) evolve when applying these optimizations consecutively. Although we illustrate all of the above with a practical healthcare application, we must lay stress on the fact that this study may be extended to other areas as well.

When compared to a model running as is on a simulator, the listed optimizations managed to reduce maximum memory consumption by a factor of between 6 and 7, while increasing total execution time only by around 25%. The deployment on an actual device showed a slight decrease in memory consumption with only 26% increased execution times on iOS. Android ended being surprisingly less efficient than iOS, with 22% increased memory usage and double the execution time of iOS. More modest results are also attainable without partitioning the graph, allowing both a

memory and execution time reduction by a factor of 2 on simulator. Most importantly, all our work was done on stock iOS and Android mobile devices, proving that our method can be easily reproduced and is fit for production-ready applications. Given the multiplatform characteristic of TensorFlow, it is theoretically possible to also apply our findings to the Raspberry Pi platform as well.

However, deploying deep learning applications on standalone mobile platforms does present some drawbacks developers need to be aware of. Due to the high amount of processing required, a significant increase in energy consumption is to be expected throughout the duration of the inference process. In smaller embedded devices, thermal dissipation can also become an issue. Therefore, we recommend developing standalone mobile deep learning applications only when their use is expected to be sporadic.

Nevertheless, our work was focused solely on TensorFlow Mobile, which does not include GPU support for all mobile devices. With the release of the Core ML framework for iOS, TensorFlow Lite, and the Neural Network API for Android 8.1, better metrics should be attainable with these technologies. Alternatively, if we were to focus exclusively on iOS, MPSCNN and BNNS should also allow us to achieve better results by exploiting the capabilities of Metal on devices running iOS 10 forward. The same can be said for Android exclusive deep learning solutions that harness the GPU, such as CNNDroid. However, these technologies are not multiplatform.

Acknowledgments: This work was partially supported by DCC-UC Research Grant and CONICYT-PCHA/National PhD/2016—No. 21161015 Grant.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Global Smartphone Shipments Forecast from 2010 to 2022 (in Million Units). Available online: https://www.statista.com/topics/840/smartphones/ (accessed on 30 May 2018).
2. Smartphones—Statistics & Facts. Available online: https://www.statista.com/topics/840/smartphones/ (accessed on 30 May 2018).
3. West, D. How mobile devices are transforming healthcare. *Issues Technol. Innov.* **2012**, *18*, 1–11.
4. Eggers, K.M.; Ellenius, J.; Dellborg, M.; Groth, T.; Oldgren, J.; Swahn, E.; Lindahl, B. Artificial neural network algorithms for early diagnosis of acute myocardial infarction and prediction of infarct size in chest pain patients. *Int. J. Cardiol.* **2007**, *114*, 366–374.
5. Lewenstein, K. Radial basis function neural network approach for the diagnosis of coronary artery disease based on the standard electrocardiogram exercise test. *Med. Biol. Eng. Comput.* **2001**, *39*, 362–367.
6. Libbrecht, M.W.; Noble, W.S. Machine learning applications in genetics and genomics. *Nat. Rev. Genet.* **2015**, *16*, 6, 321.
7. Google's New "Android Things" OS Hopes to Solve Awful IoT Security. Available online: https://goo.gl/sWHXAw (accessed on 30 May 2018).
8. Miotto, R.; Wang, F.; Wang, S.; Jiang, X.; Dudley, J.T. *Brief. Bioinform.* **2017**, doi:10.1093/bib/bbx044.
9. Lane, N.D.; Warden, P. The Deep (Learning) Transformation of Mobile and Embedded Computing. *Computer* **2018**, *51*, 12–16.
10. Lane, N.D.; Bhattacharya, S.; Mathur, A.; Georgiev, P.; Forlivesi, C.; Kawsar, F. Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Comput.* **2017**, 82–88, doi:10.1109/MPRV.2017.2940968.
11. Liu, S.; Liu, S.; Cai, W.; Pujol, S.; Kikinis, R.; Feng, D. Early diagnosis of Alzheimer's disease with deep learning. In Proceedings of the 2014 IEEE 11th International Symposium on Biomedical Imaging (ISBI), Beijing, China, 29 April–2 May 2014; pp. 1015–1018.
12. Cheng, J.Z.; Ni, D.; Chou, Y.H.; Qin, J.; Tiu, C.M.; Chang, Y.C.; Huang, C.S.; Shen, D.; Chen, C.M. Computer-aided diagnosis with deep learning architecture: Applications to breast lesions in US images and pulmonary nodules in CT scans. *Sci. Rep.* **2016**, *6*, doi:10.1038/srep24454.
13. Pham, T.; Tran, T.; Phung, D.; Venkatesh, S. Deepcare: A deep dynamic memory model for predictive medicine. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, Seoul, Korea, 30 April–2 May 2003; pp. 30–41.
14. Alipanahi, B.; Delong, A.; Weirauch, M.T.; Frey, B.J. Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning. *Nat. Biotechnol.* **2015**, *33*, 8, 831.

15. Jindan Zhu, Amit Pande, Prasant Mohapatra, and Jay J Han. Using deep learning for energy expenditure estimation with wearable sensors. In Proceedings of the 17th International Conference on E-health Networking, Application & Services (HealthCom), Boston, MA, USA, 14–17 October 2015; pp. 501–506.
16. Bourouis, A.; Zerdazi, A.; Feham, M.; Bouchachia, A. M-health: Skin disease analysis system using smartphone's camera. *Procedia Comput. Sci.* **2013**, *19*, 1116–1120.
17. Chen, C.F.; Lee, G.G.; Sritapan, V.; Lin, C.Y. Deep Convolutional Neural Network on iOS Mobile Devices. In Proceedings of the IEEE International Workshop on Signal Processing Systems (SiPS), Dallas, TX, USA, 26–28 October 2016; pp. 130–135.
18. Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M.A.; Dally, W.J. EIE: Efficient inference engine on compressed deep neural network. *SIGARCH Comput. Archit. News* **2016**, *44*, 243–254.
19. Wu, J.; Leng, C.; Wang, Y.; Hu, Q.; Cheng, J. Quantized convolutional neural networks for mobile devices. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 4820–4828.
20. Han, S.; Shen, H.; Philipose, M.; Agarwal, S.; Wolman, A.; Krishnamurthy, A. MCDNN: An execution framework for deep neural networks on resource-constrained devices. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*; ACM: New York, NY, USA, 2016.
21. Mao, J.; Chen, X.; Nixon, K.W.; Krieger, C.; Che, Y.N. MoDNN: Local distributed mobile computing system for Deep Neural Network. In Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 1396–1401.
22. Daudet, L.; Yadav, N.; Perez, M.; Poellabauer, C.; Schneider, S.; Huebner, A. Portable mTBI assessment using temporal and frequency analysis of speech. *IEEE J. Biomed. Health Inform.* **2017**, *21*, 496–506.
23. Yadav, N.; Poellabauer, C.; Daudet, L.; Collins, T.; McQuillan, S.; Flynn, P. Portable neurological disease assessment using temporal analysis of speech. In *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*; ACM: New York, NY, USA, 2015; pp. 77–85.
24. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.