OPERATING NEURAL NETWORKS ON MOBILE DEVICES

by

Peter Bai

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Electrical and Computer Engineering

School of Electrical & Computer Engineering

West Lafayette, Indiana

August 2019

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Saurabh Bagchi, Chair

School of Electrical and Computer Engineering

Dr. Sanjay Rao

School of Electrical and Computer Engineering

Dr. Jan Allebach

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Head of the Graduate Program

To my parents and teachers who encouraged me to never stop pushing on.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Author: Bai, Peter. MSECE
Institution: Purdue University
Degree Received: August 2019
Title: Operating Neural Networks on Mobile Devices
Committee Chair: Dr. Saurabh Bagchi

Machine learning is a rapidly developing field in computer research. Deep neural network architectures such as Resnet have allowed computers to process unstructured data such as images and videos with an extremely high degree of accuracy while at the same time managing to deliver those results with a reasonably low amount of latency. However, while deep neural networks are capable of achieving very impressive results, they are still very memory and computationally intensive, limiting their use to clusters with significant amounts of resources. This paper examines the possibility of running deep neural networks on mobile hardware, platforms with much more limited memory and computational bandwidth. We first examine the limitations of a mobile platform and what steps have to be taken to overcome those limitations in order to allow a deep neural network to operate on a mobile device with a reasonable level of performance. We then proceed into an examination of ApproxNet, a neural network designed to be run on mobile devices. ApproxNet provides a demonstration of how mobile hardware limits the performance of deep neural networks while also showing that these issues can be to an extent overcome, allowing a neural network to maintain usable levels of latency and accuracy.

CHAPTER 1. INTRODUCTION

1.1 Background

Machine learning is a field that has taken great strides in recent years. Deep Neural Network architectures such as ResNet have proven capable of learning to analyze great numbers of features from their input data streams. As a result, such networks are able to process complex inputs such as images and video with very high degrees of accuracy. Using techniques such as reinforcement learning, neural networks can be made to accomplish more intricate tasks. A recent study utilized multiple neural networks to train a reinforcement learning model to navigate a maze based on a visual feed along with a map of the maze environment [1].

While machine learning has managed to accomplish impressive feats in recent years, most state of the art architectures share a common downfall in that they are very computationally intensive. Neural networks are, at their core, linear algebra problems. While simple architectures may be relatively lightweight, running a state of the art network for even a single iteration will generally indicate a not insignificant amount of calculations. This is without taking into account the time and power needed to adequately train one of these networks in the first place. As a result, most state of the art neural networks are only practical to run on very high-end computing systems. While less powerful systems could in theory run such networks, they would run much slower, limiting their practicality. Latency is not necessarily a concern in simpler tasks performed offline, but for a large number of use cases, fast response times are key.

This issue is a particular problem for mobile devices. Mobile devices are often used to capture and view a large number of images and videos. As such, networks for processing such data would be of great utility on such devices; however, mobile hardware does not have nearly as much power as a non-mobile system. This means that any neural network running on a mobile device must contend with strictly limited computational and memory bandwidth all the while maintaining reasonable levels of accuracy, latency, and overall practicality.

CHAPTER 2. HARDWARE SETUP

Before any testing could be done on ApproxNet, we first had to produce a setup on mobile hardware capable of properly running a deep learning model. This consisted of two major steps. The first was selecting an embedded system with specifications that allowed it to run a neural network of reasonable size under reasonable accuracy and latency constraints. The second was make the adjustments needed to allow common machine learning frameworks to properly run on the selected hardware.

2.1 Hardware Selection

Since no one involved with the project had previously attempted to run a machine learning model on mobile hardware, we looked to other research on mobile machine learning to see what sort of hardware was commonly used. *MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints*, a paper which ApproxNet drew inspiration from, performed all of its tests using the NVIDIA Tegra K1 board [4]. Another recent project, DeepX, also utilized the NVIDIA Tegra K1 while also performing tests on the Qualcomm Snapdragon 800 board [7]. DXTK as well performed experiments with the Tegra K1 board while also running tests on the Qualcomm Snapdragon 400 along with the Cortex M3 and Cortex M0 chips [9]. Deepmon, rather than using development boards, performed its tests on mobile phones, namely the Samsung S7, Samsung Note 4, and Sony Z5 [8], all of which utilize various Snapdragon models [14,15,16].

The NVIDIA Tegra K1's popularity in recent mobile machine learning research meant that it was our first choice for the ApproxNet project, while the ubiquity of the Snapdragon series in mobile phones made it a close second. The Cortex M3 and Cortex M0 were not taken into consideration due to being low powered chips which were shown to be incapable of running more complex networks [9]. At the time, the NVIDIA Tegra K1 along with all the Snapdragon models used in the works we referenced were obsolete, with the NVIDIA Tegra X2 and the Qualcomm Snapdragon 835 being the most up to date models of each series.

The Snapdragon 835 was designed for more general purposes, while the Tegra X2 was explicitly designed with deep learning in mind [10]. To support this claim, NVIDIA also tested the Tegra X2 on several deep learning models. Some of the results of these tests are shown in Table 2.1.

Table 2.1 Tegra X2 Benchmark Results

Model	Maximum FPS
GoogLeNet, batch size = 2	201
GoogLeNet, batch size = 128	290
AlexNet, batch size = 2	250
AlexNet, batch size = 128	692

Due to this design philosophy behind the Tegra X2, we were initially somewhat hesitant to keep it in consideration since it meant that it was not particularly representative of typical mobile devices. The Tegra X2 was designed to operate AI algorithms in locations with poor network connectivity to allow low-latency operation to be possible [10]. This fact worried us since utilizing a piece of hardware tailor made for mobile operation of neural networks would potentially weaken our arguments in favor of ApproxNet.

We then decided to see how much more powerful the Tegra X2 was than the Snapdragon 835. The CPU and GPU specifications of each device are listed in Table 2.1. Since most of the computation for neural networks happens on the GPU, the fact that the Tegra X2 had a significantly more powerful GPU than the Snapdragon 835 was not unexpected. On the other hand, the CPU performance of both systems was much more similar, with the Snapdragon 835 actually having a slight edge over the Tegra X2. Since the performance of the Tegra X2 was ultimately comparable to that of the Snapdragon 835, we decided that it could be seen as an analogue of the best case scenario for running a neural network on a mobile device.

Table 2.2 Hardware Specifications

	NVIDIA Tegra X2 [10]	Qualcomm Snapdragon 835 [11,12]
CPU	Cortex A-57 (quad-core) – 2 GHz Denver2 (dual-core) – 2 GHz	Kryo 280 (octa-core) – 2.45 GHz
GPU	256-Core NVIDIA Pascal – 1300 MHz	Adreno 540 – 710 MHz

Ultimately, what tipped our decision in favor of the Tegra X2 was the issue of practicality. Development boards existed for both sytems, with the Tegra X2's board being called the Jetson TX2. Since the TX2 was already designed to support deep learning, we deduced that it would take less effort to set up the system to run ApproxNet. More importantly, the TX2 had a much more robust development community and had more extensive documentation than the Snapdragon 835. Given this larger pool of knowledge to draw from, it would be easier to debug any issues that arose during experimentation, since it would be likely that another developer would have encountered a similar problem in the past.

As a result, we ultimately opted for the NVIDIA Jetson TX2 as our system of choice for the ApproxNet project.

2.2 Software tools

For all of the experiments detailed in this document, we depended on Python libraries for quantitative measurements. The psutil library was used to collect data on memory and CPU utilization while the time library was used to measure network latency. External tools such as the 'top' command and the Tegrastats tool were also often used alongside the Python measurements. Tegrastats, a TX2 tool that collected numerous statistics including GPU utilization, CPU utilization, memory usage, and power consumption, was utilized extensively in the early stages of experimentation. While these external tools were capable of collecting a great number of data points, they were used primarily for qualitative assessment since their lack of integration into Python meant that they were ill-suited for identifying the causes of fluctuations in usage statistics. These external tools were typically used to initially identify general trends and potential issues since they required less time to configure and deploy than Python tools. Once an issue or trend was noticed, Python tools were then used to take more exact measurements.

2.3 Hardware Limitations

Before any work could be done on ApproxNet itself, the first order of business was to understand how to properly run a neural network on the TX2 board. Even though the TX2 was designed to run machine learning models, it still suffered from the limitations of a mobile system and had a

different architecture than typical stationary system. As a result, most of what was done in the early stages of experimentation was discovering and working around the problems inherent to the TX2 board itself.

2.3.1 Compatibility

While we were prepared to work on solving the problems of limited memory and CPU bandwidth, the first problem encountered was actually that of compatibility. Machine learning frameworks such as Tensorflow and Caffe are designed to support neural networks in both training and testing environments; however, the TX2 was not intended to be used to train networks. As such, the TX2 did not come packaged with any of the traditional machine learning frameworks, instead featuring TensorRT, a platform for optimizing and running pre-trained networks [13]. While we experimented with TensorRT, its use was ultimately dismissed. The full details of these experiements can be found in Chapter 3.

We initially assumed a machine learning framework could easily be installed; however, this did not turn out to be the case. While the TX2 ran on an Ubuntu operating system, it was custom build rather than a standard distrubution. Multiple components that were typically in an Ubunty distribution were either altered to better suit the architecture or removed altogether. As a result of these alterations, our attempt to install machine learning frameworks on the TX2 board resulted in numerous compatibility errors.

The first framework we attempted to install on the TX2 was PyTorch. While the installation script itself did not suffer from compatibility issues, we soon ran into prerequisite errors. We were able to resolve some of these issues by installing the necessary packages, but in many cases, the required package simply was not available for the version of Ubuntu the TX2 was running. We were able to find information online from another Tegra user who had managed to install PyTorch on a Tegra board; however, the information was for an older model and the instructions did not translate properly to the TX2 system. Since we were able to find no further leads, we ultimately decided to abandon PyTorch to see if a different framework would be more cooperative.

We next attempted to install Tensorflow to the TX2. As before, the installation script itself had no compatibilty issues. Unlike PyTorch, Tensorflow was actually able to successfully install; however, while the installation itself was able to complete, Tensorflow was unable to be launched. While we were able to find instructions on how to modify Tensorflow to work on the TX2, we were also able to locate a Github repository containing a third-party modified version of Tensorflow designed to work on the Tegra X1 and Tegra X2 systems. We were able to install this Tensorflow distribution without issue, and a simple "Hello World!" program verified that Tensorflow was launching properly.

2.3.2 Memory

The next problems encountered were those caused by interactions with the TX2's memory systems. After verifying the successful installation of Tensorflow, we performed verification tests using very simple neural networks. These included a very simple pre-trained MNIST model and an untrained prototype of ApproxNet. While a Tensorflow instance was able to start, the networks themselves failed to run and instead crashed after hanging for several minutes. This was initially believed to be a faulty installation, but reinstalling Tensorflow did not resolve the issue. In subsequent tests, we ran the networks alongside the Tegrastats utility. The output of Tegrastats revealed that during each experiment, memory usage steadily climbed until the program ultimately crashed. From this, we were able to conclude that the networks were causing Out of Memory errors when launched. Some research revealed that by default, when a Tensorflow model is run, it allocates all available memory on the GPU device it is run on to itself, then uses the allocated memory as it sees fit. While this would not be a problem on a system with separate GPU and CPU memory, the TX2 board does not feature dedicated GPU memory, instead sharing the same 8 GB of memory between its CPUs and GPU. As a result, when Tensorflow attempted to allocated all available GPU memory, it consumed all 8 GB of available system memory. This resulted in there being no available memory for background tasks on the CPU, including tasks vital to the operations of Tensorflow. Since these tasks could not run, Tensorflow would crash.

This issue was resolved by setting Tensorflow to incrementally allocate GPU memory as needed through the 'allow_growth' property in its GPUOptions class. Afterwards, the MNIST model was able to run to completion however, it suffered from extremely long loading times and very high

latency. The ApproxNet prototype still crashed after hanging, but it occasionally would successfully run, albeit with a latency of several seconds per inference. Tegrastats monitoring revealed that the memory usage of the ApproxNet prototype followed the same trend as before. The MNIST model's memory usage also grew rapidly; however, the growth stopped before the program crashed. Once the growth stopped, the network then ran, albeit very slowly.

Fortunately, this problem was due only to an oversight when migrating pre-trained networks to the TX2. The networks being tested had originally been created on a computing cluster with generous memory constraints. As a result, memory efficiency had not been a concern, and the networks had not been stripped of the various nodes used during the training process. These nodes consumed a significant amount of memory while having no function outside of training. While this could be tolerated in a system with higher memory bandwidth, the TX2's limited resources were unable to handle the excess load. After removing the nodes, memory usage dropped to more reasonable levels, and both the MNIST model and ApproxNet prototype ran at reasonably high framerates.

2.3.3 GPU & CPU

Once the networks were properly pruned, we encountered some minor problems in regards to the CPUs and GPU on the board. While the ApproxNet prototypes were now functional, they still ran extremely slowly. Examination using Tegrastats showed that during these tests, CPU usage was more limited than expected while the GPU was seeing little if any utilization. This was merely a problem with system settings. By default, the TX2 board was set to throttle it's performance as a safety precaution. Changing these default settings resulted in more reasonable utilization levels in later experiments.

CHAPTER 3. APPROXNET

3.1 Introduction

Once basic testing concluded, the experimental then shifted to the ApproxNet DNN. ApproxNet was a DNN designed with operation on mobile devices in mind. The main limitation ApproxNet sought to address was the fact that neural networks on mobile devices were generally quite inflexible. Due to the limited amount of memory present on most mobile devices, a very limited number of models can be loaded at once. This limits the number of usage scenarios that can be addressed at any time without loading new models from memory. While doing so is an option, the long loading times of models limits the usefulness of this approach. MCDNN, a paper that ApproxNet was inspired by, sought to address this by optimizing memory usage through model specialization (creating a variant model to recognize only the most common output classes) and model sharing (having low level layers be shared among different models)[4].

ApproxNet takes a different approach by opting not to optimize separate networks for different use cases and instead focuses on creating a single network that can be adjusted to suit the current usage scenario. Unlike traditional neural networks, ApproxNet is able to adjust the size of its input as well as the depth of the network. In many applications, latency and accuracy requirements will change over time. For a traditional neural network, the only way to adapt to these changes is switch to a different network altogether. Launching a new network will incur a latency network as the new network is loaded from memory while keeping multiple networks loaded at once once again runs in the problem of limited memory on mobile devices. By including the ability to adjust input size and network depth, ApproxNet is able to adapt to changing needs without the need for a second network.

ApproxNet at its core is based on the ResNet architecture as outlined in [3]. As such, the layers up until the output layer are all either convolutional layers or pooling layers. These layers do not require a set input size. ApproxNet takes advantage of this fact to feed downsampled images into the network when latency requirements become tighter or when accuracy requirements are loosened. A smaller input image would result in lower accuracy, but would also be processed

faster due to fewer operations being performed at each layer. Before the final fully-connected layer, ApproxNet uses a spatial pyramid pooling layer [17] to change the variable sized output of the convolutional layers into a fixed size input for the fully-connected layer.

ApproxNet's other major difference from typical neural networks is the presence of multiple output layers at varying depths throughout the network. A similar structure can be found in the GoogLeNet network; however, these output ports were only used for backpropagation and were disabled outside of training [5]. In contrast, ApproxNet's various output layers are all functional during inference. For each image, ApproxNet sets how deep into the network the image will travel. Once it reaches the designated layer, rather than being passed further into the network, it is instead redirected to an output port at that layer. This effectively makes it so that the depth of the network can be adjusted for each individual inference.

A more thorough discussion of ApproxNet's capabilites can be found in 'ApproxNet: Content-Aware Approximation for Video Classification on Mobile Platforms' [2]. In this paper, our discussion will focus on the experiments covering the performance of ApproxNet on the TX2 board under various conditions. Specifically, we will cover ApproxNet's memory usage and latency patterns along with its response to resource contention.

3.2 TensorRT

As mentioned previously, the TX2 board came with a platform known as TensorRT. NVIDIA advertises TensorRT as a platform capable of optimizing deep learning models and producing a runtime that can maintain both low latency and high framerate during inference [13]. Seeing as performance was a metric we sought to maximize for ApproxNet, we decided to test whether running ApproxNet via TensorRT yielded any benefits over running the model through Tensorflow as per usual.

Despite the similar name, TensorRT is not capable of running a network off of the .pb or checkpoint files that are used to store Tensorflow models. Instead, TensorRT initially requires a pre-existing network to be dumped to a .uff file. This is due to TensorRT having support for models originating from various machine learning frameworks. While converting a model to a .uff

file is a very straightforward process that requires but a single line of code, it needs to be done on a separate machine. Creating a .uff file requires the associated library which is not available on the Ubuntu distribution the TX2 runs.

Once the .uff file is created, it is then transferred to the TX2 board at which point TensorRT can then be used to run the network. TensorRT uses this .uff file to then build an inference engine runtime to operate the network. After an inference engine has been created, it can be serialized and saved directly to a .engine file which can be then used to relaunch the application, bypassing the need for the original .uff file. This method is faster than launching from a .uff file since TensorRT does not have to build and optimize the engine from scratch, but since an inference engine is optimized for the device it is built on, .engine files cannot to migrated to other devices running TensorRT while .uff files can.

Due to unfamiliarity with TensorRT, initial tests were run with an MNIST classifier rather than ApproxNet. This was due to the fact that when running a network in TensorRT, the user is required to identify input and output layers by name and explicitly tie those port to input and output buffers. The MNIST model possesses a single input port and a single output port, whereas ApproxNet possesses inputs for both the inference image and settings along with a number of output ports throughout the network. Thus, for simplicity's sake, we opted to first run tests on the MNIST model to gain a basic understanding of how TensorRT was operated before moving on to the more complex ApproxNet model.

As expected, we encountered some difficulties with learning TensorRT's syntax while adapting given code for our model; however after successfully setting up an inference engine from the MNIST network, we discovered a major limitation of the TensorRT runtime. While TensorRT was designed to run a trained network, it did not, unlike Tensorflow, contain methods for handling media. This meant that the task of reading in image files fell to external libraries. The Cimg library was readily available for this task, but it was when we attempted to feed the image data into the inference engine that we encountered problems. While the engine itself was functional, for any particular image, the results it produced were different from those produced by the network when run in Tensorflow. The issue lay in the fact that when using TensorRT, inputs and outputs

to the engine are handled as 1-dimensional arrays. This design ensures that the engine is agnostic to the original input format of the network since any numerical data can be reformatted to be 1-dimensional. For data that is originally 1-dimensional, this is no issue; however, for higher dimensional data such as images and video, the question arises as to how the data is to be alligned when it is collpased into a 1-dimensional format. As it turned out, the default alignment used by the Cimg library when collapsing an image did not match that of the alignment Tensorflow used to collapse images. Ultimately, the proper alignment was determined through trial and error by manipulating the input image until the output values matched those of the Tensorflow version.

The next step we took was to attempt to run ApproxNet on TensorRT. Since ApproxNet had multiple input and output ports, we needed to test whether TensorRT could even support such a setup. While we were fairly confident we were able to properly identify the ports, our plans ran into a roadblock we found that we were unable to dump ApproxNet into a .uff file. Multiple layers critical to the functionality of ApproxNet could not be converted into a .uff format. Further research revealed that this was an issue only with Tensorflow. While TensorRT primarily supported conversions from Tensorflow and Caffe, support for Caffe models was more advanced. This meant that while we were unable convert a Tensorflow model thanks to layer incompatibility, an equivalent model in Caffe would not have had the same issues. While we briefly discussed creating a version of ApproxNet in Caffe, the fact was that there did not exist a tool as far as we were aware to convert Tensorflow models to Caffe models, meaning the conversion would have to be done by hand. Since no one in the project had done such a thing, we decided in the interest of practicality to shelve plans for TensorRT for the time being.

3.3 Memory Usage and Latency

With TensorRT out of the question, the focus then shifted to evaluating ApproxNet itself. The first task I was assigned was to assess the memory usage and latency of ApproxNet on its own. One of the problems facing neural networks on mobile devices is that even if a network can be run, user requirements are not necessarily going to be static. Over time, a user's latency and accuracy needs may change. With a traditional neural network, this would be handled by switching to a different network, either by loading a new network or swapping to one that had been previously loaded. The former option would incur a large switching overhead since the user would have to

wait while the new network was loaded. The latter would not be practical on a mobile device since loading additional networks would further strain the already limited memory resources.

ApproxNet addresses this issue by allowing the user both the depth of the network as well as the complexity of the input on the fly, thus allowing them to adapt to changing requirements without having to switch networks entirely. The following experiments compare the memory profile of ApproxNet against those of our baseline. The baseline models were based off of the cascade models used in MCDNN. For a particular network, two variants existed: the full original variant and a simplified version that was trained only to identify the most common output classes. When an inference is run, the data is first passed to the simpler model, and if it fails to identify the image, it is then passed on to the full model [4].

For this experiment, w used several configurations, each of which performed inference on 300 images with the latency of each inference and Tensorflow's memory usage immediately following each inference being recorded. For ApproxNet, w used three different complexity settings. LL was a setting for low latency and low accuracy, MM medium latency and accuracy, and HH high latency and accuracy. Image size and network depth increased as latency and accuracy increased. We ran four configurations of ApproxNet, one each for every setting where only a single setting was used, and one where the setting changed from LL to MM after 100 images, then from MM to HH after another 100 images. For the MCDNN baselines, we had two models: one corresponding to the LL and MM ApproxNet settings, and one corresponding to the HH setting. We ran four configurations for MCDNN: one where only the LL/MM model was used, one where only the HH model was used, and two where both models were used. For one of the dual configurations, the two MCDNN models were loaded simultaneously, while for the other, only the model currently in use was loaded into memory. Each configuration was run multiple times and the results were aggregated across the various runs.

The memory usage patterns of these experiments are summarized in Figure 3.1. The blue bars are for ApproxNet while the red bars are for various MCDNN configurations. LL, MM, and HH denote runs where only a single complexity was tested. AN Mixed denotes runs where ApproxNet settings were changed during inference. MCDNN All denotes when multiple MCDNN models

were simultaneously loaded to memory and run sequentially while MCDNN Replace is for the strategy of replacing one network with another once requirements change. The results quite clearly show the memory efficiency of ApproxNet over the baseline MCDNN. The most complex ApproxNet configuration, ApproxNet Mixed, consumed approximately as much memory as the least complex MCDNN configuration. To match the flexibility of the ApproxNet Mixed setting, MCDNN requires an additional gigabyte of memory, as shown by the difference between ApproxNet Mixed and MCDNN all.

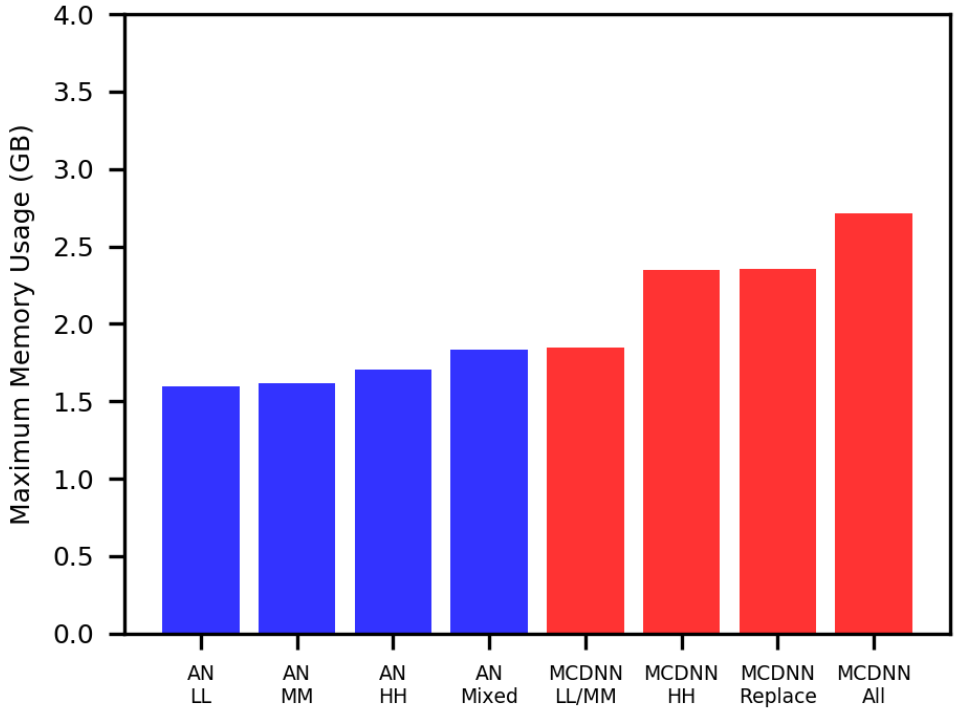

Figure 3.1 Memory Consumption of Various Models

One point which stood out was that the differernt ApproxNet configurations all had different levels of memory consumption. Since all ApproxNet tests used the exact same network model, it seemed strange that different configurations would have different memory requirements. We ran some further tests and took a closer look at the log files produced by each experiment. Figure 3.2 shows the memory usage trend over the course of one of the experiments on the ApproxNet Mixed configuration.

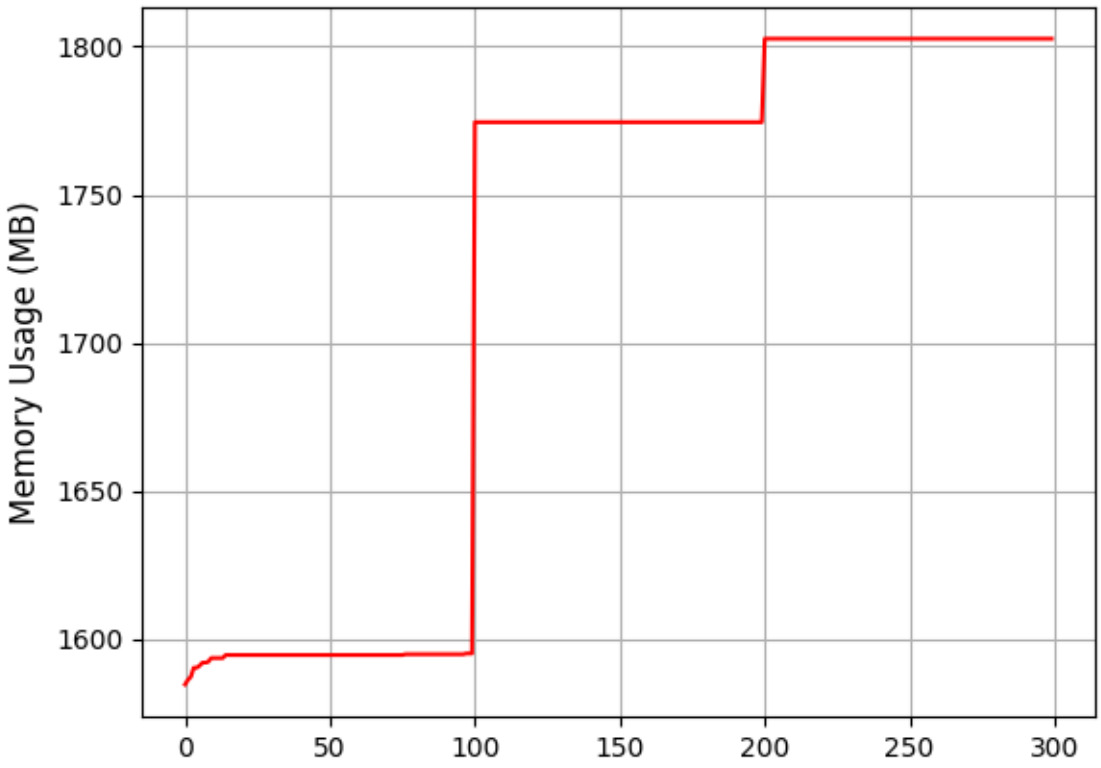

Figure 3.2 ApproxNet Mixed Memory Usage Over Time

The initial gradual climb in memory usage was a common feature in every experiment. In every experiment, the network needed time to 'warm-up'. During this period, memory usage gradually climbed, while latency was significantly higher than normal. Afterwards, memory usage leveled out and latency dropped back to typical levels. This performance loss was the switching overhead associated with loading a new network from memory. The other two memory spikes in Figure 2 correspond to when the setting changed from LL to MM and then from MM to HH. These spikes were also accompanied by a brief spike in latency, but these spikes were less severe and of shorter durations than the one incurred when the network was first launched.

Seeing this, we ran several more experiments where we switched between different ApproxNet settings. Unlike in the ApproxNet Mixed configuration, settings were revisited in these experiments. In these experiments, the first time a setting was used, we incurred similar switching costs to those shown in the ApproxNet Mixed runs. More importantly, when a setting was revisited, no switching cost was incurred. From these results, we concluded that whenever a setting was

used for the first time, the relevant structures for it would be loaded into memory where they would remain until the program terminated. Since different setting share network components, there is less loading to be done when switching to a new setting, thus explaining the smaller latency spikes.

3.4 Co-Location

The second task I was assigned was to determine the effects of contention on the performance of ApproxNet. In practice, it is very unlikely that a neural network will be the only application running on a mobile device. At any given time, any number of other applications may be competing for the same system resources. Since we had already determined that ApproxNet worked well under ideal conditions, we now sought to verify that it was also viable under more realistic usage scenarios.

For these experiments, we used a methodology inspired by Bubble-Up. Bubble-Up is a methodology intended to improve resource utilization in datacenters by safely co-locating tasks. A key component to determining whether tasks could be safely co-located was determining how sensitive a given task was to resource contention. This was done by running the task alongside a program capable of generating an adjustable amount of resource contention, then monitoring the performance of the main task [6].

We adopted this approach by running ApproxNet alongside programs which would allocate a set amount of memory (a ‘bubble’), then perform CPU operations until it was terminated. In initial experiments, the bubble size was held constants, with the number of bubbles created being used as the variable. In each experiment, we first launched the requisite number of bubbles, then launched ApproxNet. ApproxNet iterated through all of its possible settings, running several dozen inferences on each setting.

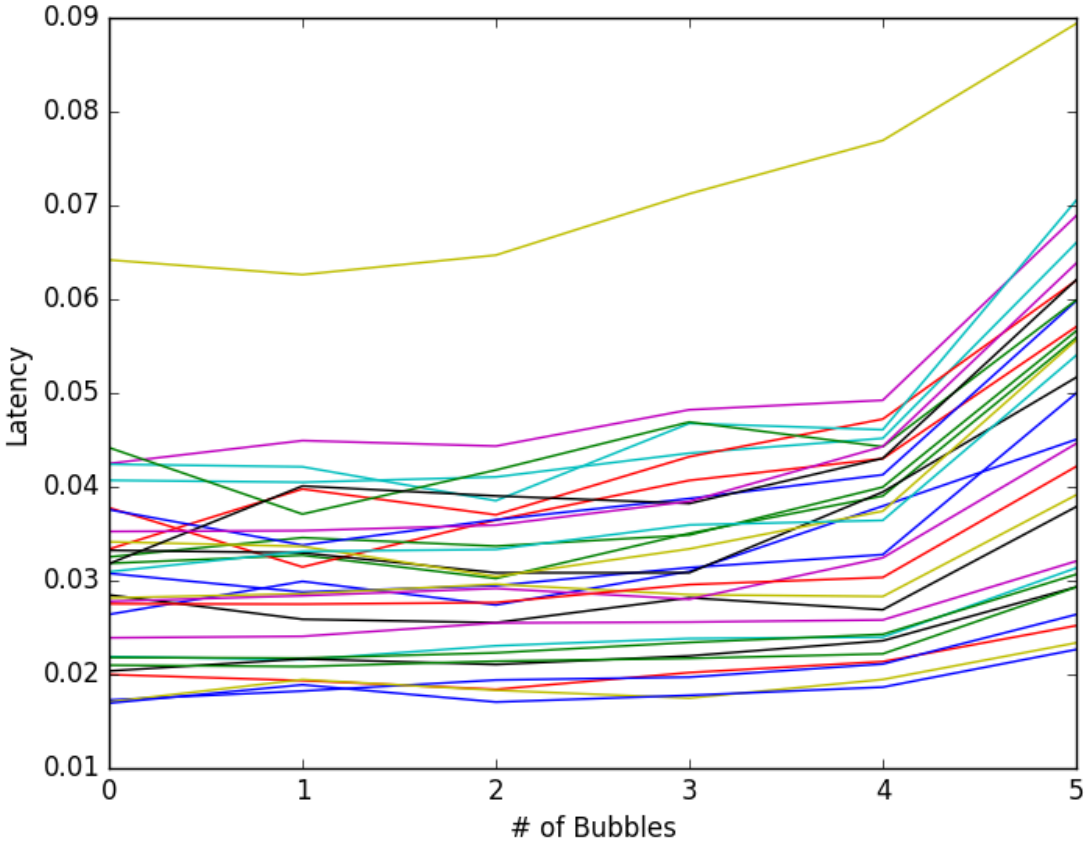

Figure 3.3 Latency vs. Bubble Count

Figure 3.3 shows the results of these experiments. Each colored line corresponds to the average latency experienced by a particular setting as the number of bubbles changes. These lines are not labeled due to space constraints, but an overall trend can be seen. As the number of bubbles increases, latency also increases for all branches. For the first four bubbles, the increase was relatively slight, but the addition of the 5th bubble resulted in a large spike in average latency. We assumed that this was because the addition of the 5th bubble created enough contention to overcome ApproxNet's tolerance for resource competition, causing performance to sharply degrade.

This was corroborated by further experiments done with a single bubble of varying size. As long as the bubble was below a certain size, its effects on the performance of ApproxNet did not vary greatly regardless of how large or small the bubble was. However, once the bubble size exceeded a certain threshold, performance rapidly degraded as bubble size increased until the program became non-functional.

These experiments demonstrated that ApproxNet was quite capable of operating under reasonable amounts of contention. As long as contention from other tasks remained below a certain threshold, ApproxNet suffered relatively little degradation in its performance. A more thorough exploration of ApproxNet's ability to deal with contention can be found in [2].

CHAPTER 4. CONCLUSION

Machine learning is a field that is rapidly advancing. In recent years, neural networks have proven capable of executing increasingly complex tasks with ever improving degrees of accuracy; however, most of these neural networks also suffer from the fact they require large amounts of computational resources to operate. This limits the use of neural networks on mobile devices, where computational and memory bandwidth are much more limited. We examined the hurdles that one would have to overcome in order to allow a neural network to run in a mobile environment. We first looked at issues of compatibility. Due to differences between mobile and stationary systems, a user would have to make numerous tweaks to a network to better suit it for running in a mobile environment. Secondly, we also examined practical limitations. For this, we looked to ApproxNet and how its design was able to mitigate some of the issues that limit the practicality of running a neural network on a mobile system. ApproxNet's unique architecture allowed to operate with a great deal more flexibility than traditional neural networks, allowing it to adapt to changing latency and accuracy needs. In addition, ApproxNet also demonstrated a reasonable robustness against contention, suffering relatively little degradation in performance below a certain level of contention.

One topic I would like to revisit is TensorRT. TensorRT was packaged with the TX2 board and was the intended method of running neural networks on the board. While we did experiment with TensorRT in the beginning, we ultimately set it aside due it being incompatible with the current iteration of ApproxNet. We could attempt to convert ApproxNet into a Caffe model to ensure compatibility, and it is also likely that future updates to TensorRT will provide great compatibility with Tensorflow models. Since TensorRT is advertised as a method of optimizing neural networks, it is possible that it can help improve the performance of ApproxNet on the TX2 board.

Overall, while there are still obstacles in the way of bringing practical machine learning models to mobile systems, ApproxNet has demonstrated that there are ways to overcome these limitations.

REFERENCES

[1] Brunner, G.; Richter, O.; Wang, Y.; and Wattenhofer, R. Teaching a machine to read maps with deep reinforcement learning. In Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[2] Xu, R.; Koo, J.; Kumar, R.; Bai, P.; Mitra, S.; Meghanath, G.; and Bagchi, S. ApproxNet: Content-Aware Approximation for Video Classification on Mobile Platforms. Under Submission, 2019.

[3] He, K.; Zhang, X.; Ren, S.; and Sun, J. Deep Residual Learning for Image Recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[4] Han, S.; Shen, H., Philipose, M.; Agarwal, S.; Wolman, A.; and Krishnamurthy, A. MCDNN: An Approximatio-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, 2016.

[5] Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. Going Deeper With Convolutions. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[6] Mars, J.; Tang, L.; Hundt, R.; Skadron, K.; and Soffa, M.L. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011.

[7] Lane, N.D.; Bhattacharya, S.; Georgiev, P.; Forlivesi, C.; Jiao, L.; Qendro, L.; and Kawsar, F. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In 2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2016.

[8] Huynh, L.N.; Lee, Y.; and Balan, R.K. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, 2017.

[9] Lane, N.D.; Bhattacharya, S.; Mathur, A.; Forlivesi, C.; and Kawsar, F. DXTK: Enabling Resource-efficient Deep Learning on Mobile and Embedded Devices with the DeepX Toolkit. In Proceedings of the 8th EAI International Conference on Mobile Computing, Applications and Services, 2016.

[10] Franklin, D. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge. *https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/* , 2017.

[11] Snapdragon 835 Mobile Platform. https://www.qualcomm.com/products/snapdragon-835-mobile-platform.

[12] Qualcomm Adreno 540. https://www.notebookcheck.net/Qualcomm-Adreno-540-GPU-Benchmarks-and-Specs.207856.0.html.

[13] NVIDIA TensorRT: Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt.

[14] Samsung Galaxy Note 4. https://www.gsmarena.com/samsung_galaxy_note_4-6434.php.

[15] Samsung Galaxy S7 (USA). https://www.gsmarena.com/samsung_galaxy_s7_(usa)-7960.php.

[16] Sony Xperia Z5. https://www.gsmarena.com/sony_xperia_z5-7534.php.

[17] He, K.; Zhang, X.; Ren, S.; and Sun, J. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. In IEEE Transactions on Pattern Analysis and Machine Intelligence, 2015.

[18] Xu, R.; Koo, J.; Kumar, R.; Bai, P.; Mitra, S.; Misailovic, S.; and Bagchi, S. VideoChef: Efficient Approximation for Streaming Video Processing Pipelines. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18), 2018.

[19] Mitra, S.; Gupta, M.; Misailovic, S.; and Bagchi, S. Phase-Aware Optimization in Approximate Computing. In Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2017.