Examining Universal Approximators with Use of Convolutional Neural Networks.

Sangjoon Lee
University of California: Los Angeles
Department of Computer Science, CS259
Los Angeles, USA
aaccjjt@ucla.edu

William Chong
University of California, Los Angeles
Department of Computer Science, CS259
Los Angeles, USA
williamchong@ucla.edu

I. Introduction

During this past decade, there has been tremendous effort in creating hardware accelerators to increase speed and energy efficiency of classical computers. But researchers faced limitations as it became increasingly difficult to build efficient hardware that is both fast and generalizable for all applications. So instead, new types of specialized hardwares were developed to increase speed of a specific application. For instance, FPGA enabled programmers to hardcode the circuits to create applications that can benefit from more efficient hardware configuration. In the area of computer networking, specialized hardware that performs specific tasks like Protocol Converter and Network Switches were developed to increase performance for specific kinds of applications.

As recent advances in machine learning gained popularity across diverse fields and applications, these specialized accelerator designs also branched into the area of machine learning platform. Now researchers in this field realized that neural networks were especially useful because they can serve as a general approximator for any kind of function. Hence by creating an accelerator for neural networks, the accelerators can increase speed for all kinds of applications given that only an approximate result is necessary for the function. This idea was put to test by research group, Alternative Computing Technologies (ACT) Lab. First they sampled functions from different fields of interest. Then, they created an artificial neural network for functions in a way such that it can perform faster computation than the original function itself. The approximation from the digital neural network maintained over 90% accuracy and gave x2.4 speed up and x2.1 energy reduction [1]. Similarly, another research group attempted to create a neural network with analogue circuit with similar benchmark functions, by using analog circuit on their neural network, they showed that it is possible to create an accelerator that can replace long lines of code with much shorter, energy efficient neural network substitutes.

This paper attempts to continue the idea of a generalizable accelerator using neural networks. Both of the previous research groups showed convincing results from multilayer perceptrons. They specifically tailored their topology to use 8 input to 1 output neuron units and showed that collection of these simple neurons can approximate all the benchmarks. However, there are more popular neural network architectures that have developed over the decade. These include Convolution neural network, Long Short Term Memory Unit, Transformers to name a few. And because of the popularity of these models, many more specialized accelerators were developed to enhance these units specifically. We wanted to see compatibility of all of these models for the benchmark created by ACT Lab. Then, based on performance of each neural model, it may be possible to evaluate which models are more desirable to create an accelerators for. And Likewise, based on the existing state of the art accelerators for all the neural architecture, it may be possible to evaluate which model is most suited for speed, accuracy, and energy efficiency.

Unfortunately due to the difficulty of formulating every single architecture specifically tailored to each algorithm, this paper only attempts to show compatibility of CNN as a general approximator for the benchmark codes as a first step to this more ambitious goal. It has been shown previously that a fully connected layer can be converted into CNN, and vice versa CNN to fully connected. So, since the previous groups were able to create approximation code with 8 to 1 perceptrons, this paper attempts to show performance and accuracy of CNN, and extrapolate the result to state of the art CNN accelerators to show the amount of energy reduction, and performance gain a CNN approximator can give in comparison to a regular neural network approximator.

II. Related Works

Most notably, ACT Lab created a benchmark function repository for other groups to create their own approximator [1]. University of Washington, Georgia Tech, and University of Texas followed suit in creating analog NPU accelerators [2]. In terms of Accelerators, one of the more renowned accelerators has been Eyeriss v.2 [3]. Its performance measures up to 35 convolutional processing frames per second. This paper will extrapolate results from a CNN

approximator network to Eyeriss v.2, to analytically model the overall speed up gain and energy gain that can happen when CNN approximation is used in combination with state of the art accelerators.

III. Methods

A. Benchmark Choice and Profiling Overview

We selected a few benchmark algorithms/workloads from different sectors to compare our models against the traditional computational algorithms. AxBench provided 7 main benchmark workloads. Due to time constraints, we narrowed it down to three. We tried to select workloads that were intuitive to benchmark and avoided workloads that had significant overlap with another. For now, we settled on benchmarking Sobel filter or image edge detection, Discrete Fast Fourier Transform (FFT), and JPEG Image compression and decompression. With each benchmark, we selected an appropriate corresponding evaluation criterion based on existing work; Root Mean Squared Error (RMSE) for JPEG, and Mean Squared Error for FFT and Sobel.

We decided to use Python Jupyter Notebooks in Google Colab to implement our models and benchmark the existing algorithms. This aided in documenting our design process as well as assisting with collaboration. For each benchmarked workload, we first used a "traditional" method to generate a set of input and output data. This allowed us to run the evaluation criterion and establish a ground truth. It also gave us data to train and design our models with.

B. Sobel Edge Detector

Sobel Edge Detector was created using the cv2 module. The algorithm itself performs a Sobel algorithm across both an x and y axis of image. The dataset was created by using preexisting image datasets of animals found online. The image was scaled to 512 by 512 wide and fed across the algorithm to create output that was equivantly 512 by 512.

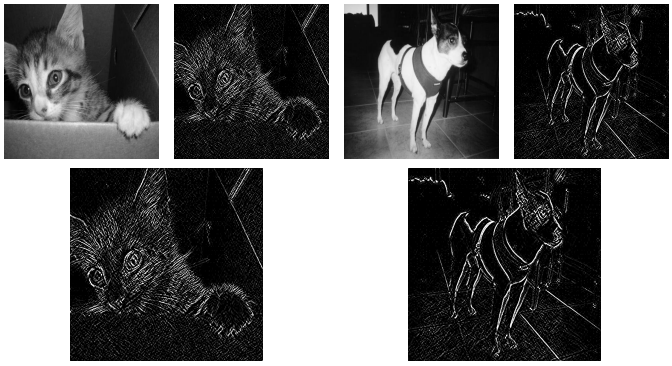

Figure 1: Examples of Sobel Image Edge Detection. In the top row, the original image and ground truth images are shown; the bottom shows the result of our Sobel approximating model.

In this format we generated 1500 data points (due to limitation of memory) and trained the CNN network in which 20% of the data was used for validation. And overall trained the model with 40 epochs with batch size of 25 images where approximated output is given in Figure 1.

C. Fast Fourier Transform

Fast Fourier Transform was profiled by feeding a single dimensional vector of random floating points. The output of this function was the same sized vector, in which the total amount of data points numbered 10,000. Because each value was composed of both real number and imaginary number, the inputs and the label were separated into two values and fed into the CNN neural network as a set of single 2 x N vectors. About 10% of the data was used for validation. It used a batch size of 25 with 80 epochs and used Means Square Error as loss and validation metric.

D. JPEG Image Compression and Decompression

We selected the JPEG Image compression and decompression as it seemed like a task perfectly suited for approximation, and one where it was easily visualized. Profiling this workload for input and output was fairly simple as it meant sourcing a set of PNG formatted images and running it through an OpenCV2-based image encoder/decoder. We decided to use the set of images from Reference [4] as they stress test different aspects of compressors; they also include the same image but in both 8-bit and 16-bit image quality.

To fit the CNN model theme, we based our model off of Jiang et al.'s "End-to-End Compression Framework Based on Convolutional Neural Networks" [5]. This model has two parts; first, the Compact CNN to learn structural information and encode it into JPEG, and second the Reconstructional CNN to decode it. Both parts are trained simultaneously with the loss defined as the sum of their MSE. We fed in our training set to the model and ran it for 200 epochs (< 100 epochs performed poorly).

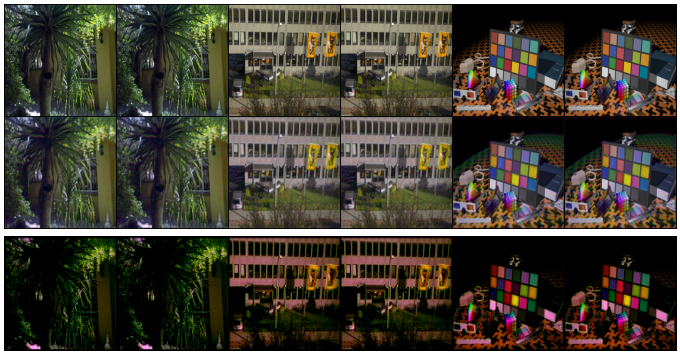

Figure 2: A Comparison of Original, Reconstructed, and Compressed Images. On the top, displayed is a set of six source png images (both 8 and 16 bit versions), on the second row, is the same set of images after passing through the End-to-End CNN based encoder/decoder. The bottom row is the set of images after just the model's compression stage.

This model performed admirably, with test result examples shown in Figure 2. Artifacts were kept to a minimum and the images were remarkably intact. In less trained model versions, the resulting images showed banding around edges/polygons. This perhaps gives some insight into how the model decides to optimize compression — it finds structural elements then condenses them.

E. Accelerator Selection and Simulation

For our purposes, we settled on using a state of the art CNN accelerator simulator named SCALE (Systolic CNN Accelerator Simulator) and tested our models on a simulated Eyeriss v.2 accelerator [6]. We chose SCALE as it presented an easy-to-use interface where we provided a given network topology and specified an accelerator architecture, and it returned cycle counts and memory usage.

After running our models on SCALE, we recorded the simulated performance. SCALE returns the cycles for each CNN layer, along with their DRAM read and writes. The Eyeriss paper presented memory accesses, latencies, and the corresponding energy costs given a core clock frequency of 200 MHz [7]. From this we extrapolated a linear relation between memory accesses and computation latency with energy cost, shown in **Figure 3**. We used clock frequency and the number of cycles in each of our CNN models' layers to compute the latency for our convolutional neural network (a simple multiplication operation. For energy and power consumption, we used the aforementioned relation extrapolated from the Eyeriss paper data, adjusted for nonnegative values. Using this as a model, we used the number of DRAM access computed by SCALE to calculate Energy and Power for each layer.

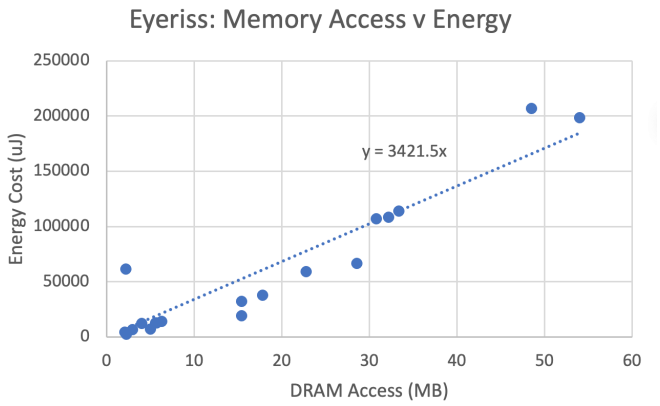

Figure 3: Extrapolating Energy Cost based on Memory Accesses. The Eyeriss paper presented DRAM accesses, time latencies, inference measurements/speed per NN layer. We plotted this against the paper's corresponding energy costs and adjusted it to have no negative energy values for nonnegative DRAM access values. The fitted linear function is y = 3421.5x, where y is energy cost in uJ and x is DRAM accessed in MB.

IV. Evaluation

A. Methodology and Result

The Evaluation of our model involves three criteria: accuracy, latency, and energy consumption. The model should retain reasonable accuracy and return better performance in comparison to the original functions. The original functions are Fast Fourier Transform from SciPy library, and JPEG Compression and Sobel edge detection from the OpenCV library. While these libraries' accuracy and latency was straightforward to get, their energy/power consumption was hard to derive. These values weren't given in the original paper. The original paper by AxBench and other groups merely had improvement scales instead of the actual values. So after hours of unfruitful research, we found a system file in linux: /sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj. This file contains energy (in uJ) that is used up by the computer over time. We ran the functions over a fixed time interval to get a rough estimate of the power used by os and the background processes. Then we ran our code to measure its power and subtracted the baseline power created by the os. It is far from obtaining perfect energy consumption measurement due to many factors such as other student's project process colliding with energy consumption. But out of five different methods we considered. This was by far the most viable option that was left out to us. Hence with this understanding. In **Table 1**, we now present the comparison of the original libraries on a set of test benchmarks against our designed CNN models'.

Table Head	**Algorithms**		
	Sobel	***Image Compression***	***FFT***
Original Algorithm			
Time (s)	1.505 E-3	0.72	1.302 E-4
Energy (J)	1.696 E-09	2.3020 E-4	4.825 E-10
Accuracy	-	RMS: 0.0133	-
CNN Approximation			
Time (s)	1.301 E-3	0.9158	1.555 E-5
Energy (J)	4.220 E-4	0.12704	1.293857 E-3
Accuracy	MSE: 0.0338	RMS: 0.0331	MSE: 0.0189
Improvement			
Time	x1.16	x0.79	x8.59

Table 1: Comparison of original, non-approximated code using conventional libraries against their CNN, approximated counterparts. Here we present our findings and calculated energy and time costs as well as accuracy of the given model. For Image compression we evaluate with Root Mean Squared Error (RMS), while for Sobel and FFT we evaluate accuracy with Mean Squared Error.

We evaluated accuracy of our model in the following way: First, we designed CNN models for the given benchmark workload and evaluated design quality with both quantitative (RMSE, MSE, Accuracy) and qualitative (comparing images) measurements. For the image compression model design, it was faster and easier to evaluate the prototype models' performance with qualitative compression/decompression image quality comparisons. For Sobel and FFT, the model's results were not as easily validated with qualitative observations, so quantitative measurements of model performance were used. RMSE and MSE were preferred over accuracy, especially for FFT, as we aimed for relative, reasonable approximations, rather than strict accuracies.

Evaluation on latency and energy reduction, involved extracting our final models' performance (time, memory, energy) using the SCALE simulation; SCALE simulator provided an analytical model of our models' quality and

validity within Eyeriss Accelerator. As for the design of our neural network model, We wanted to design fair CNN models that could be used reasonably to approximate code, while not obsessing over model implementation details; this evaluation methodology helped us to be more flexible and verify more strictly only when a "final" model was settled on.

B. Discussion and Analysis

In terms of accuracy there seems to be a reasonable error, bounded by some rms and mse. These values were expected as the neural net is naturally an approximator. Notice how the original JPEG image compression itself has some error when compared with uncompressed image (Table 1). In comparison, the neural net compression version isn't as good as the original compression function, but it still does the job pretty well, as it was demonstrated visually in **Figure 2**.

In terms of performance, while all three algorithms saw improvement in energy consumption. two algorithms (FFT, Sobel) experienced some speed up. We noticed that there was the greatest reduction in latency for FFT, then for Sobel, while image compression time increased with the CNN based compression. It seems that, the more the original algorithm is related to the image processing, the less our approximator can optimize. This was a shocking result because we thought CNN will perform best at tasks related to image processing because CNN is used more for image related tasks. One explanation for this phenomenon may be that there are already many existing image processing algorithms that are more accurate and more optimized than the approximator itself in visual areas that are simplistic. So there is not much room for optimization that CNN—a generalized filter for image processing—can do better than the optimized algorithms that leverage perfect, mathematically accurate filters. OpenCV may have already done an optimal job and have used up most of the potential for parallelization optimization. However, this may not be the case for the Fourier Transform. Fourier Transform has very little ties to image processing as it uses both real and imaginary values and is applied more for 1D signals. Hence our understanding is that CNN approximators may provide performance boost in less image filter related areas like the Fourier Transform.

Regarding the image compression performance, we think that while there was lackluster speed performance, it was interesting to explore the novel end-to-end CNN based compression framework that compressed it into a JPEG and extracted/decoded it with a paired CNN model. Perhaps another source of slowdown was our inexperience with the design of the network and how to optimize it for our set of images. The CNN-based model did result in compressed JPEG images that were exceptionally small in size—so it did this aspect well.

V. Conclusion and Future Work

In this paper we investigate the followup question to the ACT universal accelerator paper of whether there is a better Neural Network suited for general-code acceleration over a Multi-Layer Perceptron model. To do this, we hand-designed a set of CNN models to apply to three types of algorithms/workloads: Sobel Edge Detection, Discrete Fast Fourier Transforms, and JPEG image compression and decompression. We conclude that CNNs are sometimes appropriate for certain workloads, while in other cases actually lead to non-optimal performance compared to conventional techniques. This may be due to a variety of factors such as non-optimal CNN model design, inexperience with simulating energy and time costs, and high standards set by conventional algorithmic techniques.

A. Limitation

Our project saw major limitations in measuring energy for each of the original algorithms. We considered other methods, such as counting the number of x86 instruction set and evaluating its energy cost, using average TDP applied to the time interval. But none of them remained viable. The estimated energy cost using the linux energy file did lead to a conclusion that there was a significant energy reduction. But we believe that the resulting improvement is too big to be true, and aim to investigate this in future work.

B. Future works

In our original project proposal, we had presented a project goal to test the suitability of various types of neural networks. In this paper, we were only able to focus our resources on assessing the suitability of using CNNs to approximate generalized code/workloads. This was mainly due to the unforeseen amount of time we took to design and implement appropriate models for each benchmark. In future work, we would like to extend both our explored range of models, as well as extend our evaluated benchmarks to include the other approximable workloads from ACT. Some examples of other models that show potential are LSTMs, BERT, auto-encoders, and composite models (e.g. CNN-LSTM)

Another interesting tangent we could explore is the possibility of implementing a model that chooses and designs an appropriate neural network specifically tailored for the profiled workload. This could give us an interesting look into the compilation portion of ACT's general accelerator paper and any areas of improvement that we can add.

Future work also includes taking a more detailed look at energy and area costs of implementing our NN topologies on existing accelerators. Beyond that, another path for future extension could be to use these NN general-code acceleration models to design a better "general" purpose accelerator in hardware and minimize the disadvantages inherent with CPUs, NPUs, and GPUs.

VI. Statement of Work

Profiling, model design, and SCALE evaluation of the Sobel and FFT benchmarks were done by Sangjoon, while work on JPEG and SCALE CNN accelerator research was done by William. Sangjoon and William both ran accelerator simulations on their respective algorithms. Sangjoon wrote Sections I and II, as well as Subsections B, C, and E in III. William wrote A, D and E in III and A in IV. Each of us wrote the respective subsections in Evaluation (IV).

Acknowledgment

We would like to thank Professor Tony Nowatzki for his support from CS259 lectures, the course design, and organization that encouraged creative project ideas. We would also like to thank Kunal Deshmukh for his Pytorch implementation of the End-to-End CNN Compressor.

VII. References

[1] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, H. Esmaeilzadeh, "AXBENCH: A Multi-Platform Benchmark Suite for Approximate Computing", IEEE Design and Test, special issue on Computing in the Dark Silicon Era 2016.

[2] R. S. Amant et al., "General-purpose code acceleration with limited-precision analog computation," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014, pp. 505-516, doi: 10.1109/ISCA.2014.6853213.

[3] Y. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127-138, Jan. 2017, doi: 10.1109/JSSC.2016.2616357.

[4] Image Compression Test Suite. https://imagecompression.info/

[5] Jiang et al., "End-to-End Compression Framework Based on Convolutional Neural Networks," https://arxiv.org/abs/1708.00838.

[6] Samajdar, Ananda and Zhu, Yuhao and Whatmough, Paul and Mattina, Matthew and Krishna, Tushar, "SCALE-Sim: Systolic CNN Accelerator Simulator", https://github.com/ARM-software/SCALE-Sim.

[7] Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. https://arxiv.org/pdf/1807.07928.pdf