POWERML: A FRAMEWORK FOR OPTIMIZING THE POWER CONSUMPTION OF DEEP LEARNING SYSTEMS

KIRK SWANSON AND LOGAN NOEL

ABSTRACT. Some of the most important advances in machine learning over the past five years are due to innovation in neural network architecture design and have enabled the rise of applications such as autonomous driving and personalized medicine. However, deep learning algorithms are very power intensive, so they are typically run on specially designed clusters and are rarely deployed on mobile devices. The power consumption of deep learning systems, for both training and inference tasks, is a key limiting factor inhibiting the transition to embedded and mobile devices, yet it has received relatively little attention. In this work, we propose a framework for minimizing power consumption in the training and inference of generalized neural network topologies. An important byproduct of this framework is a tool that measures the cache performance of training and inference for a given model topology.

1. INTRODUCTION

1.1. **Motivation.** Some of the most revolutionary advances in machine learning over the past five years, for applications such as autonomous driving and personalized medicine, are due to innovation in neural network architecture design. Convolutional neural networks (CNNs), for example, are now state-of-the-art amongst computer vision algorithms. However, systems like CNNs and other deep learning algorithms tend to consume a lot of power, so they are typically run on the cloud [1]. They are rarely deployed on mobile devices subject to severe battery constraints, such as smartphones, even though there are innumerable real-world applications that could be achieved if this were easily done. For example, typical smartphones have trouble running AlexNet in real-time for more than an hour [2].

The power consumption of deep learning systems, for both training and inference tasks, is a key limiting factor inhibiting the transition to mobile devices, and it is therefore an increasingly important issue. Further research on power consumption of neural networks will increase our understanding of the efficiency of different types of network architectures and may help catalyze the transfer of deep learning from the cloud to mobile and handheld devices.

1.2. **Previous Work.** Studying power consumption of neural networks is a relatively nascent field. Some recent advances in chip design have dramatically reduced

Date: December 10, 2018.

Key words and phrases. power consumption, neural networks, deep learning, hyperparameter optimization.

Thanks to professors Ian Foster and Rick Stevens for supporting this project during their course, Deep Learning Systems.

power consumption of neural networks [3], but most recent efforts have focused on optimizing network architectures. There has been a lot of previous work on streamlining network architectures by reducing model size or amount of computation, for example by pruning weights of existing networks (e.g. removing less important weights to make CNN filters sparse) [4, 5], designing networks with bitwidth-reduced weights and operations [6], or making the layers more compact with fewer weights. However, reducing model size or number of computations is not directly related to power consumption, because models with a smaller size or fewer operations can still have higher overall power consumption.

In order to directly address the problem of energy consumption, Yang et al. [2] recently developed an approach to reducing CNN energy consumption with minimial loss of accuracy, called Energy-Aware Pruning. Essentially, they preferentially prune (set to zero) filter weights in CNN layers that contribute the most to energy consumption and then re-tune the network in order to preserve high accuracy. Using this method, they are able to reduce energy consumption in AlexNet and GoogLeNet by 3.7x and 1.6x, respectively, with less than a 1% top-5 accuracy loss.

In order to model the energy consumption of CNNs, they use the key insight that energy consumption of neural networks is driven not only by computations but also by memory access. In fact, according to [7], the energy cost of memory access is orders of magnitude higher than that of computations: "The energy cost of a DRAM access (1 to 2 nJ) is a couple of orders-of-magnitude higher than the cost...[of a] functional operation (10 pJ). Part of this comes from the very energy-inefficient I/O that DRAM systems use...." Yang et al.'s energy estimation framework, which is built off of a framework proposed in [8], takes into account memory access as a primary contributer to energy consumption. In this framework, they extrapolate energy numbers for each operation and memory access from actual hardware measurements.

The Energy-Aware Pruning algorithm is extremely useful but has a couple of limitations. First, it is designed only for CNNs and is not directly generalizable to other network topologies. Second, it assumes that a network architecture has already been developed and simply needs to be pruned, or optimized. For custom neural networks however, especially ones that are being built specifically for use on mobile devices, it might be useful to have a way of taking energy consumption into account during the process of developing the network topology from the start.

1.3. **Project Summary.** In this work, we propose a framework for minimizing power consumption while maximizing accuracy in the training and inference of generalized neural network topologies. Specifically, we perform a simplified neural architecture search for two network types, Dense Feed Forward Rectangle (DFFR) and Convolutional Neural Network (CNN), operating on a subset of MNIST containing images of 0s and 1s. The search yields a suggestion for a topology that maximizes accuracy while minimizing power consumption. This network topology can also serve as a starting point for a more careful neural architecture search.

An important byproduct of this framework is a tool that measures the cache performance of training and inference for a given model topology. All code is available at https://github.com/lmnoel/powerML. We begin by discussing the methodology for this framework in §2 and then describe our rseults in §3

2. Methods

2.1. **Power Consumption and Memory Access.** As suggested by Yang et. al, we use processor cycles required for memory access as a proxy for power consumption. This proxy is useful, because measuring actual power consumption is a manual and slow process (which we have attempted using the Kill-A-Watt multimeter system), rendering it impractical for large-scale use. Instead, we use the Linux utility Valgrind (specifically the cachegrind tool) to build a profile of cache hits and misses for a given training or inference process [12]. This profile gives us three critical values:

(1) the number of L1 cache hits
(2) the number of LL hits (lower level cache, as Valgrind tracks only L1 and either L2 or L3 cache, depending on how many layers the processor architecture uses)
(3) LL misses, i.e. memory access

We use the latency, in units of processor cycles, associated with each level of memory access, which we get from Intel's specification sheets[1]. We then construct a single metric for the "cost" of a process according to the following equation[2]:

$$\begin{aligned} \text{cost} = {} & \text{L1 hits} \times \text{L1 latency} + \\ & \text{LL hits} \times \text{LL latency} + \\ & \text{LL misses} \times \text{memory latency}. \end{aligned} \tag{2.1}$$

We constructed a suite of synthetic loads (matrix multiplications ranging from 400 x 400 to 8000 x 6000, run in Python's NumPy library) and gathered the actual power consumption of those tasks using the Kill-A-Watt system. We looked at the increase in power consumption when running the synthetic loads, netting out power consumption of the OS. We then measured the same benchmarks using our software power consumption proxy, processor cycles for memory access. As shown in Figure 1, our predictor is very good (r=0.991), which gives clear proof of concept that processor cycles for memory access is an accurate proxy for power consumption.

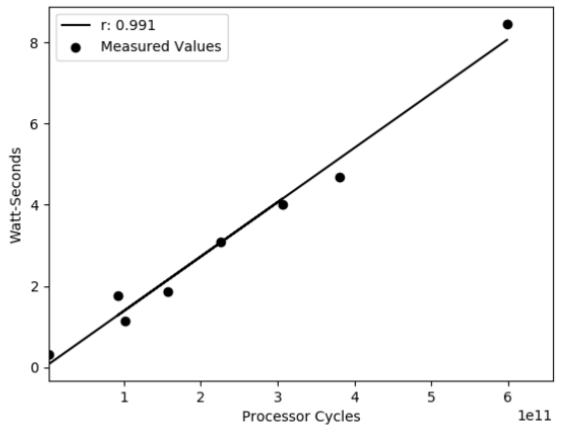


FIGURE 1. Power consumption vs processor cycles for cache and memory access

2.2. **Hardware and OS.** All trials for this paper were conducted using Ubuntu 18.04 LTS running on two Intel Xeon x5650 Processors (2.66Ghz).

2.3. **Bayesian Optimization Scheme.** When choosing hyperparameters[3] that determine the network topology and other network characteristics, such as number

[1]For instance, the spec sheet for the Broadwell architecture family is givey at https://www.7-cpu.com/cpu/Broadwell.html

[2]To replicate this project using the code provided, one needs to compile Python 2.7 from source after enabling memory debugging, as Valgrind does not support Python by default. See http://svn.python.org/projects/python/trunk/Misc/README.valgrind

[3]In this paper, hyperparameter refers to any parameter in the network that is adjusted during the optimization process. So, batch size and number of network layers are both examples of hyperparameters.

of layers and batch size, it is common to perform a grid search or a random search. The grid search involves preselecting discrete values of hyperparameters to test, while random search involves selecting these values at random in a series of trials. In both of these cases, the optimal set of hyperparameters, and therefore the optimal network topology, is chosen from these trials based on which one yields the best result, a minimum (or maximum) of some objective function. In our case, the objective function would involve a combination of minimizing power consumption while maximizing accuracy (see §2.4).

Instead of grid or random search, we choose to use a Bayesian optimization scheme to perform the hyperparameter search. Grid and random search are both uninformed by past trials and so do not converge over time, but Bayesian approaches keep track of past evaluation results in order to make increasingly well-informed guesses about which set of hyperparameters to test next in order to find the objective function minimum.

One popular class of Bayesian optimization schemes is called Sequential Model-based Global Optimization (SMBO). Essentially, these methods form a probabilistic model that maps the hyperparameters to a probability of a score on the objective function, $p(y|x)$. This probabilistic model of the objective function is called a "surrogate." As described in [9], SMBO methods work by choosing the next set of hyperparameters to test on the objective function by selecting hyperparameters that perform best on the surrogate function. As each set of hyperparameters is evaluated, the method updates the surrogate probability model in order to make increasingly well-informed guesses. After going through a specified number of iterations, the method uses the final version of the surrogate to suggest the optimal set of hyperparameters for the objective function.

There are different ways of identifying which hyperparameters to select based on the surrogate model, but one of the most effective is a metric called Expected Improvement, otherwise known as an "exploration-exploitation" criterion. Given a desired threshold value for the objective function, y^*, and some set of hyperparameters x, the Expected Improvement is given by

$$\mathrm{EI}_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y)p(y|x)dy. \quad (2.2)$$

The first factor in the integrand promotes values in regions that are likely to contain objective function minima (exploitation), while the second term promotes regions that have greater uncertainty (exploration). When this integral is positive, it means that the hyperparameter set x is expected to yield an improvement relative to the threshold value y^*.

The specific SMBO method that we use in this work is called a Tree-structured Parzen Estimator (TPE), which is thoroughly described in [10]. Instead of modeling the surrogate directly as $p(y|x)$, this method uses Bayes rule, $p(y|x) = \frac{p(x|y)p(y)}{p(x)}$, to model $p(x|y)$ and $p(y)$ instead. $p(x|y)$ is broken down into $l(x)$ and $g(x)$, such that

$$p(x|y) := \begin{cases} l(x) & y < y^* \\ g(x) & y \geq y^*. \end{cases} \quad (2.3)$$

In other words, we create two different distributions for the hyperparameters: one where the objective function value is less than the threshold, $l(x)$, and one where the objective function value is greater than the threshold, $g(x)$. These non-parametric densities are constructed after some number K of evaluations of the objective function. y^* is chosen to be slightly greater than the best observed objective function score.

In this approach, the Expected Improvement is given by

$$\text{EI}_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) \frac{p(x|y)p(y)}{p(x)} dy, \tag{2.4}$$

which can be rearranged as

$$\text{EI}_{y^*}(x) \propto \left(\gamma + \frac{g(x)}{l(x)}(1-\gamma)\right)^{-1}, \tag{2.5}$$

where $\gamma = p(y < y^*)$ (no specific $p(y)$ is necessary). So, the TPE works by drawing sample hyperparameters from $l(x)$, evaluating them in terms of $g(x)/l(x)$, and returning the set x that gives the best expected improvement value.

To implement the TPE algorithm, we used the Python package, "hyperopt" [11], by Bergstra et al.

2.4. **Objective Function for Optimization.**

The Bayesian approach allows us to use a custom objective function that balances each of our three objectives: minimizing training cost, minimizing inference cost, and maximizing accuracy. Specifically, the objective function we use is given by

$$\begin{aligned} \text{cost} = \alpha \cdot (\text{training cost})^2 &+ \\ \beta \cdot (\text{inference cost})^2 &- \\ \gamma \cdot (\text{model score})&, \end{aligned} \tag{2.6}$$

where $\gamma = 1 - \alpha - \beta$ and costs are normalized by the first iteration to 1. This function has several desirable features. It allows the experimenter to weight each objective differently to find an optimal balance of the objectives. It also penalizes extreme values much more harshly than it rewards optimal values, which should reduce the probabiliity of converging to a local minimum as opposed to the global minimum.

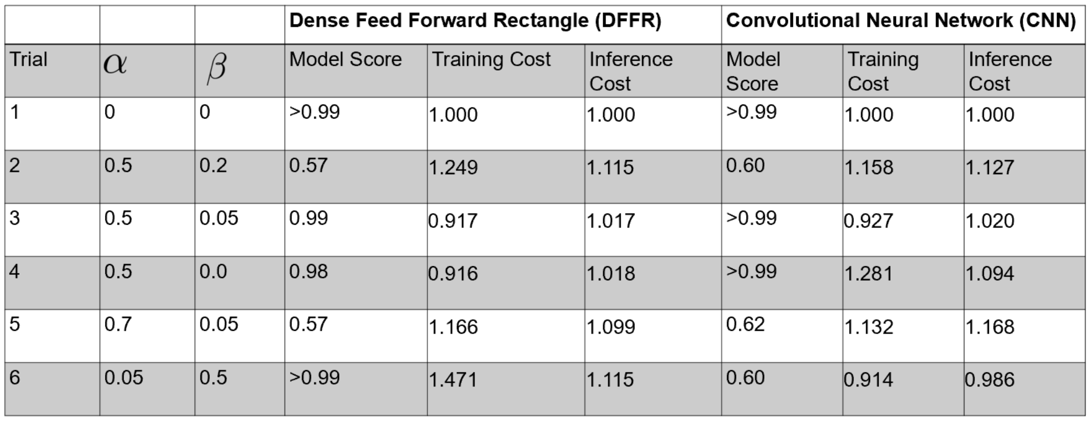
2.5. **Search Spaces.**

In this work, we explore two different types of simple network architectures: Dense Feed Forward Rectangle networks (DFFR) and Convolutional Neural Networks (CNN). For the DFFR networks, the hyperparameter search space includes the number of layers, the width of these layers, and the batch size. The width value, which is the number of nodes in a layer, is a single value that applies to all of the layers in the network (hence the network is a "rectangle"). The number of layers ranges from 1 to 10, the width value ranges from 1 to 20, and the batch size ranges from 2 to 128. Each DFFR network has a fixed final layer equal to the number of classes in the dataset.

For the CNNs, the hyperparameter search space includes the number of convolutional layers, the number of filters for each layer, the filter size for each layer, and the batch size. Similar to the width value for DFFRs, the number of filters and filter size applies uniformly to all convolutional layers in a given CNN. The number of convolutional layers ranges from 1 to 10, the number of filters range from 1 to 5, the filter size ranges from 1 to 10, and the batch size ranges from 2 to 128. Each CNN has two fully connected layers at the end, the first of size 128 and the second of size equal to the number of classes in the dataset. We decided to start with these specific bounds on the network topologies in order to test our search space algorithm effectively without needing GPUs to speed up the process.

In our code, the TPE algorithm begins by placing a discrete uniform distribution over the hyperparameter values and then reweights these distributions appropriately as the algorithm forms the non-parametric densities $l(x)$ and $g(x)$. All networks were run using Keras.

2.6. **Dataset.** Our code currently provides two options for the dataset, "mnist" and "mnist_small." "mnist" is the full MNIST dataset, while "mnist_small" is a subset of the MNIST dataset containing 0s and 1s (100 samples for training and 100 for testing). All computations in this paper were performed using the "mnist_small" dataset in order to speed up computation time.



			Dense Feed Forward Rectangle (DFFR)			Convolutional Neural Network (CNN)		
Trial	α	β	Model Score	Training Cost	Inference Cost	Model Score	Training Cost	Inference Cost
1	0	0	>0.99	1.000	1.000	>0.99	1.000	1.000
2	0.5	0.2	0.57	1.249	1.115	0.60	1.158	1.127
3	0.5	0.05	0.99	0.917	1.017	>0.99	0.927	1.020
4	0.5	0.0	0.98	0.916	1.018	>0.99	1.281	1.094
5	0.7	0.05	0.57	1.166	1.099	0.62	1.132	1.168
6	0.05	0.5	>0.99	1.471	1.115	0.60	0.914	0.986

FIGURE 2. DFFR and CNN performance across values of α and β.

3. RESULTS

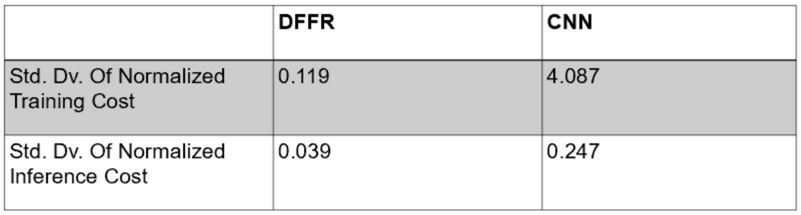
This section describes our results. We performed six Bayesian optimization trials for DFFR and CNN networks, each with a different set of α and β values, and recorded the model score, training cost, and inference cost of the optimized network after 10 iterations of the optimization scheme. These results are shown in Figure 2.

3.1. **DFFR.** Trial 1 serves as the baseline, because it only weights accuracy in the objective function. As expected, this approach produces a highly accurate DFFR after just 10 iterations of Bayesian optimization. In trial 2, we weight α and β highly, but the optimizer fails to converge, leading to a worse measure in each of the objectives. In trials 3 and 4 however, we find the model loses less than one percent and less than two percent in accuracy, respectively, while exhibiting an 8.3% nad 8.4% reduction in training cost, respectively. Inference cost does not substantially change when β is reduced from 0.05 to 0, and exceeds baseline in both cases. In trial 5, we find results which closely match trial 2; in both cases the optimizer fails to converge. This is likely because there is insufficient signal for the optimizer (see §3.3 below). Finally, in trial 6 we use high β values to see if we are able to optimize effectively on inference cost, and we find that the model converges on accuracy but does not improve on inference cost (see §3.4 below).

3.2. **CNN.** Again, we see that the baseline trial performs as expected, reaching >0.99% accuracy. Trials 2 and 5 fail to converge in a remarkably similar pattern to the DFFR trials. Trial 3 yields a 7.3% reduction in training cost while inference cost continues to exceed baseline. Curiously, trial 4 fails to reduce training cost. Trial 6 exhibits behaviour unlike any of the other trials in this experiment: training cost is reduced relative to baseline by 8.6% (despite being a low-alpha trial) and inference cost is reduced by 1.4%–the only reduction in inference cost relative to baseline we observe in all trials. CNN trial 6 is the only case where we see a reduction in cost but failure to converge on accuracy.

3.3. **Failure to Converge.** Empirically, the model appears to fail to converge when γ < 0.4, or (α + β > 0.6), likely because there is insufficient signal from the accuracy metric of the objective function.

3.4. **Variance of Cost.** Figure 3 presents the standard deviations of normalized training and inference costs for trial 6. In the DFFR case, training cost has a standard deviation 3 times higher than inference cost, as is the case for all DFFR trials. Inference costs seem to be much more consistent than training, which reduces signal for the objective function and makes it far more difficult to optimize. Notice in the CNN case, the normalized inference cost is much more variable, perhaps explaining why CNN trial 6 saw a reduction in inference cost relative to baseline. However, the training cost is substantially more variable, a feature that all models which failed to converge on accuracy shared.

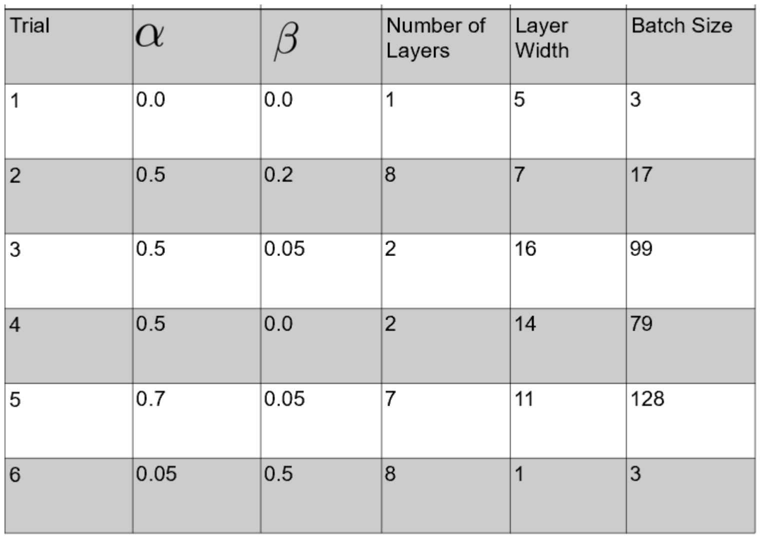


	DFFR	CNN
Std. Dv. Of Normalized Training Cost	0.119	4.087
Std. Dv. Of Normalized Inference Cost	0.039	0.247

FIGURE 3. Standard Deviation of DFFR and CNN costs for Trial 6.

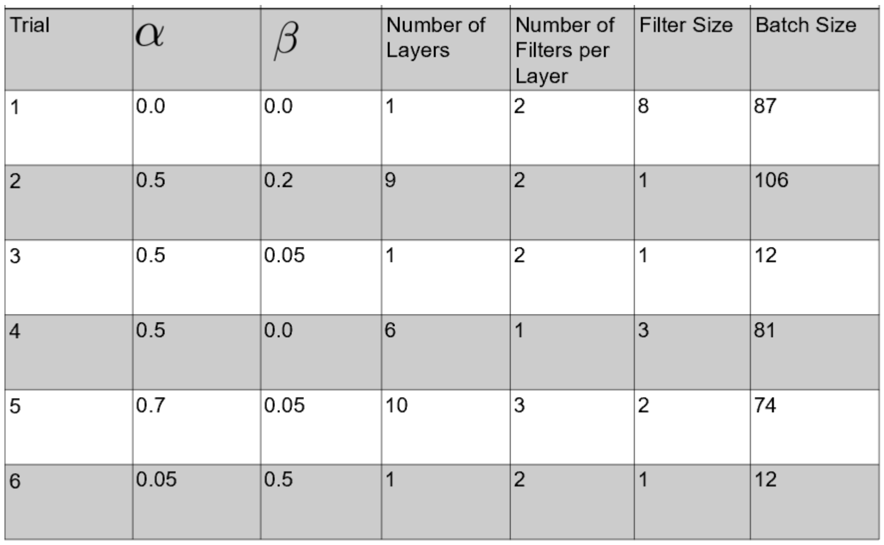
3.5. **Common Features of Optimized Networks.** Figures 4 and 5 present the optimal set of parameters suggested by the optimization process for each trial. For the DFFR, high α trials are associated with higher batch sizes, while low α trials are associated with lower batch sizes. This result matches intuition: a higher batch size results in fewer batches, which requires fewer overall reads from memory. Conversely, batch sizes tend to decrease when optimizing on accuracy alone. We also find that high α and low γ trials have more layers which are wider than baseline.

In the CNN trials, there is less of a clear connection between α and batch size. High-α trials which successfully converged on accuracy had more layers relative to baseline, but smaller filters.



Trial	α	β	Number of Layers	Layer Width	Batch Size
1	0.0	0.0	1	5	3
2	0.5	0.2	8	7	17
3	0.5	0.05	2	16	99
4	0.5	0.0	2	14	79
5	0.7	0.05	7	11	128
6	0.05	0.5	8	1	3

FIGURE 4. Optimal Architectures for DFFR.



Trial	α	β	Number of Layers	Number of Filters per Layer	Filter Size	Batch Size
1	0.0	0.0	1	2	8	87
2	0.5	0.2	9	2	1	106
3	0.5	0.05	1	2	1	12
4	0.5	0.0	6	1	3	81
5	0.7	0.05	10	3	2	74
6	0.05	0.5	1	2	1	12

FIGURE 5. Optimal Architectures for CNN.

4. DISCUSSION

This project is meant to serve as a proof of concept that deep learning architectures can be optimized to decrease power consumption while maintaining high levels of accuracy. We demonstrated that this is possible by decreasing power consumption of DFFRs and CNNs during training by 8.3 and 7.3%, respectively, with only a marginal decrease in accuracy. This reduction in power consumption is less effective than the pruning approach of Yang et al., but it is substantially more computationally efficient and is generalizable to broad classes of models. We were unable to successfully optimize inference power consumption, most likely because there is much less variability in the cost relative to training cost and we only ran the optimization scheme for a small number of iterations. We also presented a new tool for programmatically and accurately measuring the power consumption of deep learning processes using memory access as a proxy.

5. Future Directions and Limitations

This project is not intended to provide a general answer for how to optimize every deep learning application to reduce power consumption. Rather, these conditions will depend on hardware, data, and the specific needs of the researcher. Optimal topologies will depend on many context-specific factors like cache structure, memory throughput and shape of the data. This framework can suggest architectures which can serve as a starting point for more focused study.

To decrease the search space powerML needed to cover, we decided to limit all layers to the same width (DFFR) and same number of filters and filter size (CNN). This was a necessary concession given the limited computational resources available to us. Varying these parameters, and adding different parameters such as learning rate to the search space, would allow powerML to make even better predictions about optimal architectures and parameters. A better objective function, which is more sensitive to signal, would also increase the efficacy of the system and perhaps allow us to optimize on inference power consumption.

This system will only work on local–i.e. not distributed–systems, and will only work on CPUs due to the limitations of Valgrind. This is a serious constraint in the field of deep learning, but it could be overcome with a mechanism to programmatically measure the power consumption at the hardware level. Future work should also attempt to investigate in more detail why the topology and parameter sets suggested by the optimization are optimal.

6. Project Contributions

6.1. Logan.

(1) Poster: Results, Conclusion, tables, figures, part of Future Directions and Limitations and part of Methods.
(2) Paper: Methods (Power Consumption and Memory Access, Hardware, Objective Function), Results, Discussion and part of Future Directions/Limitations.
(3) Programming and experiments: Implemented the tool to measure power consumption in processor cycles and the objective function for the optimizer.

6.2. Kirk.

(1) Poster: Background, Specific Aims, part of Future Directions and Limitations, part of Methods, References.
(2) Paper: Abstract, Introduction, Methods (Bayesian Optimization Scheme, Search Spaces, Dataset), part of Future Directions/Limitations, editing in LaTex.
(3) Programming and experiments: Implemented the Bayesian optimization scheme and the Keras neural network models.

References

[1] https://techxplore.com/news/2017-07-method-neural-networks-power-consumption.html
[2] https://arxiv.org/pdf/1611.05128.pdf
[3] https://techxplore.com/news/2018-02-chip-neural-networks-power-consumption.html
[4] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *NIPS*, 2015.
[5] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *ICLR*, 2016.
[6] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *ECCV*, 2016.
[7] https://ieeexplore.ieee.org/abstract/document/6757323
[8] "Y. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.
[9] https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f
[10] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl, "Algorithms for Hyper-Parameter Optimization," in *NIPS*, 2011.
[11] https://github.com/hyperopt/hyperopt
[12] http://www.valgrind.org/

Email address: `swansonk1@uchicago.edu`

Institute for Molecular Engineering, University of Chicago, 5640 S Ellis Ave, Chicago, IL 60637

Email address: `lnoel@uchicago.edu`

Department of Computer Science, University of Chicago, 5730 S. Ellis Avenue, John Crerar Library, Chicago, IL 60637