HyperPower: Power- and Memory-Constrained Hyper-Parameter Optimization for Neural Networks

Dimitrios Stamoulis*, Ermao Cai*, Da-Cheng Juan†, Diana Marculescu*
*Department of ECE, Carnegie Mellon University, Pittsburgh, PA
†Google Research, Mountain View, CA
Emails: dstamoul@andrew.cmu.edu, ermao@cmu.edu, dacheng@google.com, dianam@cmu.edu

***Abstract*—While selecting the hyper-parameters of Neural Networks (NNs) has been so far treated as *an art*, the emergence of more complex, deeper architectures poses increasingly more challenges to designers and Machine Learning (ML) practitioners, especially when power and memory constraints need to be considered. In this work, we propose *HyperPower*, a framework that enables efficient Bayesian optimization and random search in the context of power- and memory-constrained hyper-parameter optimization for NNs running on a given hardware platform. *HyperPower* is the first work (i) to show that power consumption can be used as a low-cost, *a priori* known constraint, and (ii) to propose predictive models for the power and memory of NNs executing on GPUs. Thanks to *HyperPower*, the number of function evaluations and the best test error achieved by a constraint-unaware method are reached up to 112.99× and 30.12× faster, respectively, while never considering invalid configurations. *HyperPower* significantly speeds up the hyper-parameter optimization, achieving up to 57.20× more function evaluations compared to constraint-unaware methods for a given time interval, effectively yielding significant accuracy improvements by up to 67.6%.**

1. Introduction

Hyper-parameter optimization of Machine Learning algorithms, and especially of Neural Networks (NNs), has emerged as an increasingly challenging and expensive process, dubbed by many researchers to be *more of an art than science*. A surprisingly high number of state-of-the-art methodologies heavily relies on *human experts* and this knowledge separates a useless model from cutting edge performance [1]. Nonetheless, as the design space of hyper-parameters to be tuned grows, the task of proper hand-tailored tuning can become daunting [2].

Moreover, the ability of a *human expert* could be hampered further if we are to consider platform specific test-time power and memory constraints. We visualize the complexity of the design space in Figure 1, by reporting testing error and GPU power consumption for different NN variations of the AlexNet (CIFAR-10 with Caffe [3] on Nvidia GTX-1070). We observe that, for a given accuracy level, power could differ significantly by up to $55.01W$ (*i.e.*, more than a third of the GPU Thermal Design Power). Hence, the *more of an art than science* rationale could become non-trivial to exploit in a hardware constrained design space, necessitating a significant, yet often unavailable, familiarity of the researcher with the hardware architecture. In addition, the evaluation of each possible architecture could take hours, if not days, to train. Thus, traditional techniques for hyper-parameter optimization, such as grid search, yield poor results in terms of performance and training time [2].

Bayesian optimization and *random search* methods have been shown to outperform human experts in hyper-parameter optimization for NNs [4] [2] [5]. Nevertheless, both methods have not been extensively studied in the context of hardware-constrained hyper-parameter optimization. On the one hand, Bayesian optimization has increased complexity for cases where constraints are not available *a priori* [6]. On the other hand, random methods without hardware-aware enhancements could wastefully consider infeasible points that are randomly selected. These observations constitute the key **motivation** behind our work. As a motivating example, hardware-aware hyper-parameter optimization (based on our framework presented later on) can find an iso-error NN with power savings of $12.12W$ compared to AlexNet, or an iso-power NN with error decreased to 21.16 from 24.74%.

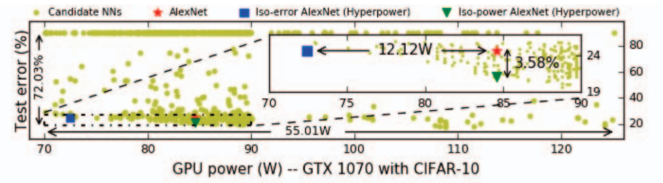

Figure 1. Power differs for a given accuracy level, hampering a human expert's ability to identify the optimal NN configuration under hardware constraints (similarly for memory, not shown due to space limitations).

To this end, we propose *HyperPower*, a framework that enables efficient Bayesian optimization and random search in the context of power- and memory-constrained hyper-parameter optimization for NNs. Our work makes the following **contributions**:

1) *HyperPower* is the *first work* to show that power consumption can be treated as an *a priori* known constraint for NN model selection. This insight for low-cost constraint evaluation paves the way towards enabling efficient Bayesian optimization and random search methods.

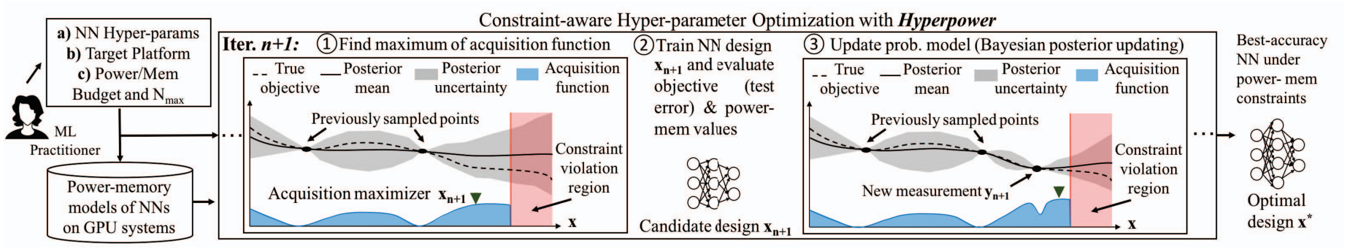
Constraint-aware Hyper-parameter Optimization with Hyperpower
a) NN Hyper-params
b) Target Platform
c) Power/Mem Budget and N_{max}
ML Practitioner
Power-memory models of NNs on GPU systems
Iter. n+1:
① Find maximum of acquisition function
True objective
Posterior mean
Posterior uncertainty
Acquisition function
Previously sampled points
Constraint violation region
Acquisition maximizer x_{n+1}
x
② Train NN design x_{n+1} and evaluate objective (test error) & power-mem values
Candidate design x_{n+1}
③ Update prob. model (Bayesian posterior updating)
New measurement y_{n+1}
Best-accuracy NN under power- mem constraints
Optimal design x^*

Figure 2. Overview of *HyperPower* flow and illustration of the Bayesian optimization procedure during each iteration $n + 1$. The ML designer only provides the NN design space, the target platform, the power/memory budget values, and the number of iterations $N_{\max}$. Our goal is to find the NN configuration with minimum test error under hardware constraints. Bayesian optimization uses a surrogate probabilistic model $\mathcal{M}$ to approximate the objective function; the plots show the mean and confidence intervals estimated with the model (the true objective function is shown for reference, but it is unknown in practice). At each iteration $n + 1$, an acquisition function $\alpha(\cdot)$ is expressed based on the model $\mathcal{M}$ and the maximizer of $\alpha(\cdot)$ is selected as the candidate design point $\mathbf{x}_{n+1}$ to evaluate. *HyperPower* incorporates the power-memory models directly into the acquisition function formulation, thus inherently preventing sampling from constraint-violating regions (shaded red). The objective function (NN test error) is evaluated, *i.e.*, the candidate NN design $\mathbf{x}_{n+1}$ is trained and tested. Then, the probabilistic model $\mathcal{M}$ is refined via Bayesian posterior updating based on the new observation. After $N_{\max}$ iterations, *HyperPower* returns the design $\mathbf{x}^*$ with optimal accuracy that satisfies the hardware constraints.

2) To the best of our knowledge, *HyperPower* is the *first* work to propose predictive models for the power and memory consumption of NNs running on GPUs.
3) *HyperPower* reaches the number of function evaluations and the best test error of a constraint-unaware method up to **112.99**$\times$ and **30.12**$\times$ faster, respectively, while **never** considering invalid configurations.
4) *HyperPower* allows for up to **57.20**$\times$ more function evaluations compared to a constraint-unaware method for a given time interval, yielding a significant accuracy improvement by up to **67.6%**.

2. Related work

Modeling hardware metrics: Prior work relies on simplistic proxies of the memory consumption (*e.g.*, counts of the NN's weights [6]), or on extrapolation based on technology node energy tables per operation [7] [8] [9]. Consequently, existing modeling assumptions are either overly simplifying and have not been compared against real platforms, or they reflect outdated technology nodes that are not representative of modern GPU architectures. On the contrary, we train our models on commercial Nvidia GPUs and we achieve accurate predictions against actual hardware measurements. Our recent work also introduces more elaborate (layer-wise) predictive models for runtime and energy, which can be incorporated into *HyperPower* [10] and which could be flexibly extended to account for process variations [11], thermal effects [12], and aging [13].

Bayesian optimization under constraints: Prior art has proposed formulations for constrained Bayesian optimization that generalize the model-based treatment of the objective to the constraint functions [6]. Hernández-Lobato *et al.* have developed a general framework for employing Bayesian optimization with unknown constraints or with multiple objective terms [14]. This framework has been successfully used for the co-design of hardware accelerators and NNs [15] [16], and the design of NNs under runtime constraints [14]. However, existing methodologies evaluate only MNIST on hardware simulators [15] [16], do not consider power as key design constraint [14], and use a simplistic count of the network's weights as a proxy for the memory constraint. Instead, in our work, we propose an accurate model for both power and memory that is trained and tested on different commercial GPUs and datasets. *HyperPower* and the proposed power and memory models can be flexibly incorporated into generic formulations that support constrained multi-objective optimization [14].

Prior art has motivated optimization cases where the constraints can be expressed as known *a priori* [6]; these formulations enable models that can directly capture candidate configurations as valid or invalid [17]. In this work, we are first to show that both power and memory constraints can be formulated as constraints known *a priori*. We exploit this insight to train predictive models on the power and memory consumption of NNs executing on state-of-the-art platforms and datasets, allowing the *HyperPower* framework to navigate the design space in a constraint "complying" manner. We extend the hyper-parameter optimization models to explicitly account for hardware imposed constraints.

Random (model-free) methods: Random [5] and random-walk [8] hyper-parameter selection has been shown to perform well in problems that have low *effective dimensionality* [1]. Nevertheless, when hardware constraints are to be accounted for, it is as likely for random methods to sample a point inside the invalid region as to select a candidate point outside of it. Our enhancements allow to quickly discard invalid randomly selected points.

3. HyperPower Framework

The *HyperPower* framework is illustrated in Figure 2. The ML practitioner selects the hyper-parameter space (possible NN configurations), the target platform, and the power/memory constraints. After $N_{\max}$ iterations, *HyperPower* returns the NN with optimal accuracy that satisfies the hardware constraints. This problem of interest is a special case of optimizing function $f(\mathbf{x})$ over design space $\mathcal{X}$ and constraints $\mathbf{g}(\mathbf{x})$, *i.e.*, $\min_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}), s.t.\ \mathbf{g}(\mathbf{x}) \leq \mathbf{c}$, where the objective function (*i.e.*, test error of each NN configuration) has no simple closed form and its evaluations are costly. To efficiently solve this problem, *HyperPower* exploits the effectiveness of Bayesian optimization methods.

3.1. Bayesian optimization

Bayesian optimization is a sequential model-based approach that approximates the objective function with a surrogate (cheaper to evaluate) probabilistic model $\mathcal{M}$, based on Gaussian processes (GP). The GP model is a probability distribution over the possible functions of $f(\mathbf{x})$, and it approximates the objective at each iteration $n+1$ based on data $\mathbf{X} := x_i \in \mathcal{X}_{i=1}^{n}$ queried so far. We assume that the values $\mathbf{f} := f_{1:n}$ of the objective function at points $\mathbf{X}$ are jointly Gaussian with mean $\mathbf{m}$ and covariance $\mathbf{K}$, *i.e.*, $\mathbf{f} \mid \mathbf{X} \sim \mathcal{N}(\mathbf{m}, \mathbf{K})$. This formulation intuitively encapsulates our belief about the shape of functions that are more likely to fit the data observed so far. Since the observations $\mathbf{f}$ are noisy with additive noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$, we write the GP model as $\mathbf{y} \mid \mathbf{f}, \sigma^2 \sim \mathcal{N}(\mathbf{f}, \sigma^2\mathbf{I})$. At each point $\mathbf{x}$, GP gives us a cheap approximation for the mean and the uncertainty of the objective, written as $p_{\mathcal{M}}(y|\mathbf{x})$ and illustrated in Figure 2 with the black curve and the grey shaded areas.

Each iteration $n+1$ of a Bayesian optimization algorithm consists of three key steps:

① **Maximization of acquisition function**: We first need to select the point $\mathbf{x}_{n+1}$ (*i.e.*, next candidate NN configuration) at which the objective (*i.e.*, the test error of the candidate NN) will be evaluated next. This task of guiding the search relies on the so-called acquisition function $\alpha(\mathbf{x})$. A popular choice for the acquisition function is the Expectation Improvement (EI) criterion, which computes the probability that the objective function f will exceed (negatively) some threshold y^+, *i.e.*, $EI(\mathbf{x}) = \int_{-\infty}^{\infty} \max\{y^+ - y, 0\} \cdot p_{\mathcal{M}}(y|\mathbf{x})\ dy$. Intuitively, $\alpha(\mathbf{x})$ provides a measure of the direction toward which there is an expectation of improvement of the objective function.

The acquisition function is evaluated at different candidate points $\mathbf{x}$, yielding high values at points where the GP's uncertainty is high (*i.e.*, favoring exploration), and where the GP predicts a high objective (*i.e.*, favoring exploitation) [1]; this is qualitatively illustrated in Figure 2 (blue curve). We select the maximizer of $\alpha(\mathbf{x})$ as the point $\mathbf{x}_{n+1}$ to evaluate next (green triangle in Figure 2). To enable power- and memory-aware Bayesian optimization, *HyperPower* incorporates hardware-awareness directly into the acquisition function (subsection 3.4).

② **Evaluation of the objective**: Once the current candidate NN design $\mathbf{x}_{n+1}$ has been selected, the NN is generated and trained to completion to acquire the test error. This is the most expensive step. Hence, our efforts towards enabling efficient Bayesian optimization, mainly focus on detecting when this step can be bypassed (subsection 3.2).

③ **Probabilistic model update**: As the new objective value y_{n+1} becomes available at the end of iteration $n+1$, the probabilistic model $p_{\mathcal{M}}(y)$ is refined via Bayesian posterior updating (the posterior mean $\mathbf{m}_{n+1}(\mathbf{x})$ and covariance covariance $\mathbf{K}_{n+1}$ can be analytically derived). This step is quantitatively illustrated in Figure 2 with the black curve and the grey shaded areas. Please note how the updated model has reduced uncertainty around the previous samples and newly observed point. For an overview of GP models the reader is referred to [1].

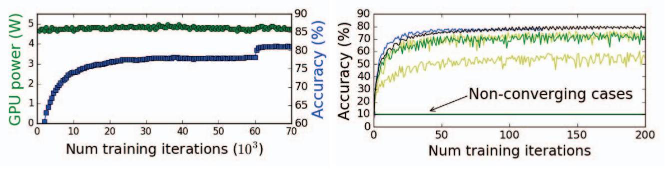

Figure 3. Visualizing our **insights**: how power varies vs accuracy with the number of training epochs (left); how accuracy can indicate configurations that do not converge to high-accuracy values ($> 10\%$) (right).

3.2. *HyperPower* enhancements

Early termination of the NN training at step ②: First, we observe that candidate architectures that diverge during training can be *quickly identified* only after a few training epochs (Figure 3 (right)). Please note that this is different than predicting the final test error of a network, which could suffer from overestimation issues [18], introducing artifacts to the probabilistic model. Instead of predicting for converging cases, we identify diverging cases, allowing the optimization process to discard low-performance samples.

Power and memory as low-cost constraints: To enable an efficient formulation with *a-priori* known constraints, we observe that power and memory are low-cost constraints to evaluate. That is, as motivated in prior art for runtime [6] [15], the power-memory characteristics of an NN are not affected by the quality of the trained model itself. In Figure 3 (left), we observe that the NN power values on Nvidia TX1 with MNIST do not heavily change even if the NN is trained for more iterations (unlike accuracy, obviously). We are *first* to exploit this insight to train predictive models for the power and memory of NN architectures. More importantly, we use the predictive models to formulate a power- and memory-constrained acquisition function.

3.3. Power and memory models

To enable *a priori* power and memory constraint evaluations that are decoupled from the expensive objective evaluation, we propose to model power and memory consumption of an network as a function of the J discrete (structural) hyper-parameters $\mathbf{z} \in \mathbb{Z}_{+}^{J}$ (subset of $\mathbf{x} \in \mathcal{X}$); we train on the structural hyper-parameters $\mathbf{z}$ that affect the NN's power and memory (*e.g.*, number of hidden units), since parameters such as learning rate have negligible impact.

To this end, we employ offline random sampling by generating different configurations based on the ranges of the considered hyper-parameters $\mathbf{z}$ (discussed in Section 4). Since the Bayesian optimization corresponds to function evaluations with respect to the test error [4], for each candidate design $\mathbf{z}_l$ we measure the hardware platform's power P_l and memory M_l values during inference and not during the NN's training. Given the L profiled data points $\{(\mathbf{z}_l, P_l, M_l)\}_{l=1}^{L}$, we train the following models that are linear with respect to both the input vector $\mathbf{z} \in \mathbb{Z}_{+}^{J}$ and model weights $\mathbf{w}, \mathbf{m} \in \mathbb{R}^J$, *i.e.*:

$$\textbf{Power model}: \quad \mathcal{P}(\mathbf{z}) = \sum_{j=1}^{J} w_j \cdot z_j \tag{1}$$

$$\textbf{Memory model}: \quad \mathcal{M}(\mathbf{z}) = \sum_{j=1}^{J} m_j \cdot z_j \tag{2}$$

We train the models above by employing a 10-fold cross validation on the dataset $\{(\mathbf{z}_l, P_l, M_l)\}_{l=1}^{L}$. While we experimented with nonlinear regression formulations which can be plugged-in to the models (*e.g.*, see our recent work [10]), these linear functions provide sufficient accuracy (as shown in Section 5). More importantly, we select the linear form since it allows for the efficient evaluation of the power and memory predictions within the acquisition function (next subsection), computed on each sampled grid point of the hyper-parameter space.

3.4. Constraint-aware acquisition function

HW-IECI: In the context of hardware-constraint optimization, EI allows us to directly incorporate the *a priori* constraint information in a representative way. Inspired by constraint-aware heuristics [6] [17], we propose a power and memory constraint-aware acquisition function:

$$a(\mathbf{x}) = \int_{-\infty}^{\infty} \max\{y^{+} - y, 0\} \cdot p_{\mathcal{M}}(y|\mathbf{x}) \cdot \mathbb{I}[\mathcal{P}(\mathbf{z}) \leq \text{PB}] \cdot \mathbb{I}[\mathcal{M}(\mathbf{z}) \leq \text{MB}] \; dy \tag{3}$$

where $\mathbf{z}$ are the structural hyper-parameters, $p_{\mathcal{M}}(y|\mathbf{x})$ is the predictive marginal density of the objective function at $\mathbf{x}$ based on surrogate model M. $\mathbb{I}[\mathcal{P}(\mathbf{z}) \leq \text{PB}]$ and $\mathbb{I}[\mathcal{M}(\mathbf{z}) \leq \text{MB}]$ are the indicator functions, which are equal to 1 if the power budget PB and the memory budget MB are respectively satisfied. Typically, the threshold y^{+} is adaptively set to the best value $y^{+} = \max_{i=1:n} y_i$ over previous observations [1] [6].

Intuitively, we capture the fact that improvement should not be possible in regions where the constraints are violated. Inspired by the integrated expected conditional improvement (IECI) [17] formulation, we refer to this proposed methodology as *HW-IECI*. We leave the systematic exploration of other acquisition functions for future work. Note that uncertainty can be also encapsulated by replacing the indicator functions with probabilistic Gaussian models as in [17], whose implementation is already supported by the used tool [4] and whose analysis we leave for future work.

3.5. Alternative methods supported by *HyperPower*

Constraints as Gaussian Processes – *HW-CWEI*: We also consider the case where the constraints are modeled by GPs [6] using a latent function $\hat{g}(\mathbf{x}) = g_0 - g(\mathbf{x})$ per constraint. Each GP models the probability of the constraint being satisfied $Pr(\mathcal{C}(\mathbf{x})) = Pr(g(\mathbf{x}) \leq g_0)$. In the context of our approach, to enable efficient constraint evaluation, we evaluate the latent functions based on our models: $Pr(\mathcal{M}(\mathbf{z}) \leq \text{MB})$ and $Pr(\mathcal{P}(\mathbf{z}) \leq \text{PB})$. Inspired by the Constraint Weighted EI (CWEI) [6] function, we refer to this second methodology as *HW-CWEI*.

Random search – *Rand*: We consider random search as the popular model-free alternative [5]. Once again, we exploit the insight of power modeling and early termination, by replacing the GP-based selection with with random selection. We denote this method as *Rand*.

Random walk – *Rand-Walk*: Random walk methods, denoted here as *Rand-Walk*, aim to "tame" the randomness by tuning the exploitation-exploration trade-off [8]; the next random point $\mathbf{x}_{n+1}$ is selected around the point $\mathbf{x}^{+}$ with the best objective value y^{+} over previous observations. Formally, at any step we select from within "neighborhood" controlled by σ_0^2, *i.e.*, $x_{n+1} \sim \mathcal{N}(\mathbf{x}^{+}, \sigma_0^2)$.

In Section 5, we show that by exploiting our insights, the *HyperPower* implementations of these methods, *i.e.*, *Rand*, *Rand-Walk*, *HW-WCEI*, and *HW-IECI*, significantly outperform their default, previously published hardware-unaware counterparts [5] [8] [6] [17], in the context of power- and memory-constrained hyper-parameter optimization.

4. Experimental Setup

We employ power- and memory-constrained optimization with the four discussed methods, on two different machines, *i.e.*, a server machine with an NVIDIA GTX 1070 and a low-power embedded board NVIDIA Tegra TX1. To train the predictive models, we profile the CNNs offline using `Caffe` [3] on both the CIFAR-10 and MNIST.

We extend and implement the aforementioned four methods on top of `Spearmint` [4]. We implement wrapper scripts around the objective/constraint functions that are queried by `Spearmint`, that automate the generation of `Caffe` simulations, and power/memory model evaluations. We employ hyper-parameter optimization on variants of the AlexNet network for MNIST and CIFAR-10, with six and thirteen hyper-parameters respectively. For the convolution layers we vary the number of features (20-80) and the kernel size (2-5), for the pooling layers the kernel size (1-3), and for the fully connected layers the number of units (200-700). We also vary the learning rate (0.001-0.1), the momentum (0.8-0.95), and the weight decay (0.0001-0.01) values. While the considered experimental setup serves as a comprehensive basis to evaluate *HyperPower*, we are currently considering larger networks on the state-of-the-art ImageNet dateset as part of future work.

5. Experimental Results

Proposed power and memory models: First, we assess the accuracy of the power and memory models (Equations 1-2). In Figure 5, we plot the predicted and actual power values, trained on the MNIST and CIFAR-10 networks for both GTX 1070 and Tegra TX1. Alignment across the blue line indicates good prediction results. We observe good prediction for all tested platforms and datasets, with a Root Mean Square Percentage Error (RMSPE) value always less than 7% (Table 1) for both power and memory models. It is worth noticing the power value ranges per device and that our proposed models can accurately capture both the high-performance and low-power design regimes.

Fixed number of function evaluations: We first assess the four methods for fixed number of function evaluations. We apply each algorithm on the MNIST and CIFAR-10 NNs with power constraints of 90W and 85W, respectively. As typical values for the experiments [6] [4] [18], we select a maximum number of 50 iterations per run (30 for MNIST); we execute each method five times, and we optimize for and we report the test error results per iteration.

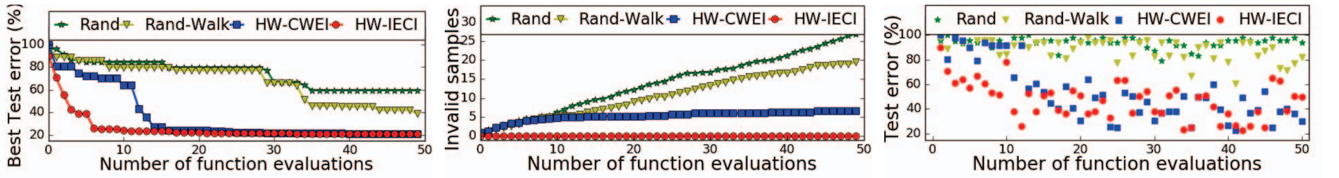

Figure 4. Assessment of the four methods on hyper-parameter optimization on CIFAR-10 CNN. (left) Best observed test error against the number of function evaluations. (center) Number of constraint-violating samples against the number of function evaluations. (right) Test error per function evaluation.

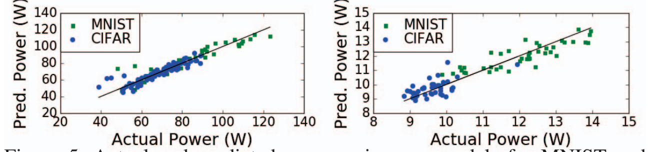

Figure 5. Actual and predicted power using our models for MNIST and CIFAR-10, executing on GTX 1070 (left) and Tegra TX1 (right).

TABLE 1. Root Mean Square Percentage Error (RMSPE) values of the proposed power and memory models.

Proposed Model	MNIST GTX 1070	CIFAR-10 GTX 1070	MNIST Tegra TX1	CIFAR-10 Tegra TX1
Power	5.70%	5.98%	6.62%	4.17%
Memory	4.43%	4.67%	– –[1]	– –[1]

For the CIFAR-10 case, we plot the results in Figure 4 and we make the following observations. We confirm in Figure 4 (center) that *HW-IECI* does not select samples that violate the constraints. This is significantly beneficial, since it allows *HW-IECI* to reach the region around the average best error in a **fifth** of function evaluations, as shown in Figure 4 (left). Finally, the Bayesian optimization methods outperform both random (model-free) methods. That is, in Figure 4 (right), it is easy to observe that most points queried by Bayesian optimization (red circles and blue squares) are in high-performance regions, while random methods tend to select points in low-performance regions.

Efficient hyper-parameter optimization via power modeling and early termination (fixed runtime): Next, we evaluate the hardware-constrained hyper-parameter optimization under maximum wall-clock runtime budget; this scenario is important in a more commercial-standard context when executing on a cluster [1] and under pricing schemes in Infrastructure as a Service systems, where speeding up the expensive function evaluation addresses not only practical but also financial limitations related to hyper-parameter optimization [2], [18]. We therefore repeat the exploration for three runs per method for each considered device-dataset pair with the following constraints constraints: 85W and 1.15 for MNIST on GTX 1070, 90W and 1.25GB for CIFAR-10 on GTX 1070, 10W for MNIST on Tegra TX1, and 12W for CIFAR-10 on Tegra TX1 (no memory constraints on Tegra[1]). To impose upper runtime constraints, each method keeps querying new samples as long as the total wall-clock timestamp is less than two hours and five hours for MNIST and CIFAR-10, respectively; please note that we allow the last sample queried right before the maximum time limit to complete (as seen in Table 3, where the average runtime is slightly above the two and five hours spans).

1. Tegra does not support NVML API for memory measurements, and the `tegrastats` command reports utilization and not memory consumption; for representative comparison, we do not consider memory on Tegra.

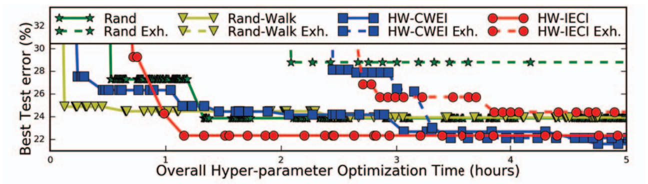

Figure 6. Capturing the benefit of using early termination and the power/memory models to all four considered methods; best test error on CIFAR-10 NN against the total hyper-parameter optimization runtime.

First, we visualize in Figure 6 the benefit that the power/memory models and the early termination offer in *HyperPower*. For CIFAR-10 on GTX 1070, we repeat the 5-hour execution for each method in an exhaustive manner, where these two enhancements are deactivated (the exhaustive versions are shown with dotted lines in Figure 6). We observe that all four methods reach a high-performance region faster that the default (exhaustive) methods, which can be seen with all solid lines lying to the left of the dotted ones. Second, we observe the density of the samples along the solid lines; this is to be expected, since low-performance or violating samples can be quickly discarded.

We present the results for all considered methods and all device-dataset pairs in Tables 2-5, where we compare against the constraint-unaware implementations of those methods (these exhaustive cases are denoted as `default`, and the average speedup values are computed as the geometric mean across all runs per case). First, in Table 2 we report the mean and the standard deviation of the best test error achieved by each method. As expected, the constraint-aware versions of all four methods supported by *HyperPower*, outperform their respective constraint-unaware counterparts, with accuracy increase by up to **67.6%** for the case of *Rand* on CIFAR-10 with Tegra TX1.

It is important to note that *HyperPower* results display less variance compared to all the constraint-unaware versions. For the random-based methods, this is because several runs completely failed to find a high-performance region. This is to be expected since the default exhaustive methodologies are agnostic to constraints, hence they could keep wastefully sampling constraint-violating designs. An extreme case of this inefficiency could be seen for both CIFAR-10 cases solved with *Rand-Walk*, which both failed to reach a feasible solution. This highlights the key disadvantage of Random Walk methods due to the sensitivity of their performance to the selection of the proper σ_0 value, which defeats the purpose of automated hyper-parameter optimization altogether. These observations for vanilla random search methods [5] [8] has significant implications, since a total of 32 hours of server runtime was inefficiently wasted.

TABLE 2. MEAN BEST TEST ERROR (AND STANDARD DEVIATION IN PARENTHESIS) ACHIEVED PER METHOD.

	MNIST – GTX 1070		CIFAR-10 – GTX 1070		MNIST – Tegra TX1		CIFAR-10 – Tegra TX1	
Solver	Default	**HyperPower**	Default	**HyperPower**	Default	**HyperPower**	Default	**HyperPower**
Rand	60.59% (41.46%)	1.01% (0.18%)	69.60% (28.85%)	24.39% (3.08%)	1.06% (0.08%)	0.97% (0.14%)	74.35% (22.13%)	24.09% (1.97%)
Rand-Walk	31.16% (41.55%)	0.84% (0.08%)	–	22.88% (0.87%)	1.04% (0.05%)	0.90% (0.12%)	–	21.90% (0.59%)
HW-CWEI	0.97% (0.19%)	0.85% (0.12%)	22.09% (1.01%)	22.09% (0.35%)	0.98% (0.13%)	0.91% (0.07%)	24.28% (0.60%)	22.99% (0.41%)
HW-IECI	0.81% (0.05%)	**0.81%** (0.02%)	22.35% (1.71%)	**21.81%** (0.38%)	0.81% (0.02%)	**0.79%** (0.03%)	23.35% (1.04%)	21.95% (0.65%)

TABLE 3. RUNTIME (HOURS) FOR *HyperPower* METHODS TO REACH THE NUMBER OF SAMPLES THAT THEIR EXHAUSTIVE COUNTERPARTS QUERIED.

	MNIST – GTX 1070			CIFAR-10 – GTX 1070			MNIST – Tegra TX1			CIFAR-10 – Tegra TX1		
Solver	Default	**HyperPower**	Speedup	Default	**HyperPower**	Speedup	Default	**HyperPower**	Speedup	Default	**HyperPower**	Speedup
Rand	2.14	0.02	101.46×	5.25	0.22	**30.31**×	2.08	0.49	**4.31**×	5.35	0.74	11.78×
Rand-Walk	2.17	0.02	**112.99**×	5.29	0.46	17.45×	2.14	1.00	2.15×	5.31	1.13	**21.00**×
HW-CWEI	2.00	0.30	10.22×	5.06	2.46	2.07×	2.16	1.32	1.65×	5.39	1.10	8.06×
HW-IECI	2.02	1.81	1.13×	5.12	2.97	1.74×	2.02	1.65	1.22×	5.22	1.71	3.48×

TABLE 4. INCREASE IN THE NUMBER OF SAMPLES THAT EACH METHOD WAS ABLE TO QUERY.

	MNIST – GTX 1070			CIFAR-10 – GTX 1070			MNIST – Tegra TX1			CIFAR-10 – Tegra TX1		
Solver	Default	**HyperPower**	Increase	Default	**HyperPower**	Increase	Default	**HyperPower**	Increase	Default	**HyperPower**	Increase
Rand	14.00	796.33	**57.20**×	14.67	405.33	**27.88**×	13.00	35.67	**2.77**×	13.33	262.33	**20.00**×
Rand-Walk	15.00	316.67	19.16×	13.33	118.33	8.86×	14.00	30.67	2.12×	14.33	88.67	5.46×
HW-CWEI	21.67	62.67	2.79×	28.00	38.67	1.38×	11.00	14.67	1.35×	13.00	27.33	1.97×
HW-IECI	53.00	60.33	1.14×	29.00	43.33	1.49×	46.33	54.67	1.18×	11.00	20.00	1.75×

TABLE 5. IMPROVEMENT IN RUNTIME (HOURS) TO ACHIEVE THE BEST ACCURACY THAT THE EXHAUSTIVE METHODS DID.

	MNIST – GTX 1070			CIFAR-10 – GTX 1070			MNIST – Tegra TX1			CIFAR-10 – Tegra TX1		
Solver	Default	**HyperPower**	Speedup	Default	**HyperPower**	Speedup	Default	**HyperPower**	Speedup	Default	**HyperPower**	Speedup
Rand	0.50	0.16	1.56×	2.09	0.53	**3.97**×	0.72	0.16	3.64×	2.63	0.48	4.54×
Rand-Walk	0.69	0.12	4.72×	–	–	–	1.74	0.27	6.18×	–	–	–
HW-CWEI	3.47	3.06	6.11×	4.12	2.58	2.08×	1.30	0.19	7.39×	5.05	1.24	**4.80**×
HW-IECI	1.61	0.10	**30.12**×	4.45	2.49	2.13×	1.53	0.14	**11.30**×	3.24	1.16	2.69×

Moreover, among all four versions supported by *HyperPower*, we can observe that *HW-IECI* always achieves the best results, which shows the importance of enabling *a-priori* constraint evaluation through our predictive models. Finally, we observe that *HyperPower*'s enhancements allow for up to **57.20**× more function evaluations (see Table 4). In terms of runtime improvement, it takes *HyperPower* up **112.99**× faster to reach the same number of function evaluations that default methods queried (see Table 3), which attests to the importance of hardware-awareness when power/memory violating samples can be quickly discarded. Most importantly, thanks to *HyperPower*, we can reach the best test error achieved by the exhaustive methods up to **30.12**× faster (see Table 5).

6. Conclusion

Accounting for power and memory constraints could significantly impede the effectiveness of traditional hyper-parameter optimization methods to identify optimal NN configurations. In this work, we proposed *HyperPower*, a framework that enables efficient hardware-constrained Bayesian optimization and random search. We showed that power consumption can be used as a low-cost, *a priori* known constraint, and we proposed predictive models for the power and memory of NNs executing on GPUs. Thanks to *HyperPower*, we reached the number of function evaluations and the best test error achieved by a constraint-unaware method up to **112.99**× and **30.12**× faster, respectively, while **never** considering invalid configurations under the proposed *HW-IECI* acquisition function. By significantly speeding up the hyper-optimization with up to **57.20**× more function evaluations compared to constraint-unaware methods for a given time interval, *HyperPower* yielded significant accuracy improvements by up to **67.6%**.

Acknowledgments

This research was supported in part by NSF CNS Grant No. 1564022.

References

[1] B. Shahriari *et al.*, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, 2016.
[2] K. Swersky, J. Snoek, and R. P. Adams, "Multi-task bayesian optimization," in *NIPS*, 2013, pp. 2004–2012.
[3] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.
[4] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *NIPS*, 2012.
[5] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *JMLR*, vol. 13, no. Feb, pp. 281–305, 2012.
[6] M. A. Gelbart, J. Snoek, and R. P. Adams, "Bayesian optimization with unknown constraints," *arXiv preprint arXiv:1403.5607*, 2014.
[7] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Delight: Adding energy dimension to deep neural networks," in *ISLPED*, 2016.
[8] S. C. Smithson *et al.*, "Neural networks designing neural networks: Multi-objective hyper-parameter optimization," in *ICCAD*, 2016.
[9] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.
[10] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu, "Neuralpower: Predict and deploy energy-efficient convolutional neural networks," *arXiv preprint arXiv:1710.05420*, 2017.
[11] D. Stamoulis and D. Marculescu, "Can we guarantee performance requirements under workload and process variations?" in *ISLPED'16*.
[12] E. Cai, D. Stamoulis, and D. Marculescu, "Exploring aging deceleration in finfet-based multi-core systems," in *ICCAD*, 2016.
[13] D. Stamoulis *et al.*, "Efficient reliability analysis of processor datapath using atomistic bti variability models," in *GLSVLSI*, 2015.
[14] J. M. Hernández-Lobato *et al.*, "A general framework for constrained bayesian optimization using information-based search," *JMLR*, 2016.
[15] J. M. Hernández-Lobato *et al.*, "Designing neural network hardware accelerators with decoupled objective evaluations," in *NIPS*, 2016.
[16] B. Reagen *et al.*, "A case for efficient accelerator design space exploration via bayesian optimization," in *ISLPED*, 2017.
[17] R. B. Gramacy and H. K. Lee, "Optimization under unknown constraints," *arXiv preprint arXiv:1004.4027*, 2010.
[18] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *IJCAI*, 2015, pp. 3460–3468.