Learning Network Architectures of Deep CNNs under Resource Constraints

Michael Chan, Daniel Scarafoni, Ronald Duarte, Jason Thornton and Luke Skelly

MIT Lincoln Laboratory, 244 Wood St, Lexington, MA 02421, USA

Abstract

Recent works in deep learning have been driven broadly by the desire to attain high accuracy on certain challenge problems. The network architecture and other hyper-parameters of many published models are typically chosen by trial-and-error experiments with little considerations paid to resource constraints at deployment time. We propose a fully automated model learning approach that (1) treats architecture selection as part of the learning process, (2) uses a blend of broad-based random sampling and adaptive iterative refinement to explore the solution space, (3) performs optimization subject to given memory and computational constraints imposed by target deployment scenarios, and (4) is scalable and can use only a practically small number of GPUs for training. We present results that show graceful model degradation under strict resource constraints for object classification problems using CIFAR-10 in our experiments. We also discuss future work in further extending the approach.

1. Introduction

Deep learning with convolutional neural networks (CNN) has become the method of choice in recent years when it comes to solving image recognition problems [16]. The methodology has the ability to simultaneously learn a hierarchical feature representation of the underlying data and an optimal classifier given labeled training data. Some of the leading models published in the literature had surpassed the performance of human, most notably in the ImageNet 1000-class image classification problem [11].

However, most prior works on deep learning were driven primarily to achieve high accuracy on certain challenge problems [16][23][26][10]. The designer generally has to experiment with many different network architectures as well as trying combinations of hyper-parameters such as learning rate, number of iterations, batch size for normalization, and regularization parameters in order to arrive at good-performing models. The main fundamental issue is that the model design process generally still requires a lot of human intervention. Furthermore, little considerations were paid to resource constraints for subsequent deployment of those models during the design stage. Often times, a high-performing model may only be slightly inferior to other competitive models, but at the expense of much higher computational cost or memory consumption at runtime. Having to find good models from the large space of possible models subject to additional resource constraints make the problem doubly challenging.

This paper proposes a method to address the challenge of network architecture design for applications that may be subject to strict deployment resource constraints. We are not aware of any prior works that attempt to solve this problem in an automated way. We selected *CIFAR-10* to assess the effectiveness of our approach.

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

2. Related Work

Various authors have explored different strategies for optimizing hyper-parameters of machine learning algorithms, including the use software tools to manage the complexity of the process at least in the case of fixed-size configuration spaces [3]. Here, we highlight related works that we think are most relevant.

The idea of using the framework of Bayesian optimization for hyper-parameters search was proposed in [24], in which a Gaussian process was used to model the generalization performance of a learning algorithm. The fundamental idea was to model the objective function as a Gaussian process defined over the parameter space and then use the model to draw successive samples in order to maximize the likelihood of expected improvement in the objective function. However, the method was limited to fixed-size hyper-parameter spaces.

Reinforcement learning (RL) is another class of methods that has been employed to optimize deep network structure. Zoph and Le [29] proposed a neural architecture search method that generated neural networks architectures with another recurrent neural network (RNN). The RNN was trained by *REINFORCE*, searching from scratch in a variable-length architecture space, to maximize the expected accuracy of the generated architectures on a validation set. In the RL formulation, a controller generates hyper-parameters as a sequence of tokens, which are actions chosen from hyper-parameters spaces; each gradient update to the policy parameters corresponds to training one generated network to convergence; and

measured accuracy on a validation set is the reward signal. The authors designed a parameter server approach to speed up training. Compared with state-of-the-art methods, this approach achieved competitive results for an image classification task. Baker *et al*. [1] proposed a meta-learning approach, using Q-learning with ϵ-greedy exploration and experience replay, to generate CNN architectures automatically for a given learning task.

Others recently attempted to use evolutionary algorithms for the architecture learning problem. Real *et al*. [22] used a variety of novel and intuitive mutation operators to navigate the large search spaces, and was able to discover competitive CNN models that rival state-of-the-art results on *CIFAR-10*. The process was reported to require no human participation once evolution starts from trivial initial conditions. However, the authors did note that there was a large computational cost involved requiring many hundreds of GPU's to produce the model. In another case, it took thousands of CPU nodes for two months to produce a competitive model for the single-channel MNIST benchmark using an evolution-based approach [5].

An adaptive structural learning method based on the idea of boosting was recently proposed with theoretical guarantees on generalization performance, but was applied only to unconstrained binary classification problems [4]. The applicability to constrained multi-class problems remains to be future work.

While none of the hyper-parameter learning methods above considered resource constraints of the end applications, there are related works that address the question of resource constraints separately from the architecture learning problem. Existing methods include limiting numerical precision [7][21], network pruning or slimming [8][9][17][20], using specialized network components [13], and implementing sparse convolutions [20], all of which either assumed that a network architecture had already been determined, or required significant human input to design one. To control memory resources required by a CNN, the authors in [6] incorporated in their optimization a penalty term formulated as a function of sum of bit-depths of all parameters. This work is similar in spirit to our work in that hardware resource constraints were considered upfront, but they assumed a fixed architecture (a 4-layer model in their experiments) and the penalty term cannot enforce a hard constraint.

In our work, we address the constrained architecture learning problem with variable-dimension parameter spaces in a number of ways that we believe are novel. First, our method uses a sequential combination of broad randomized searches and stochastic coordinate descent optimization that finds good solutions from a very large architecture space. This is in contrast to evolutionary approaches, where they perform a large number of random adaptations and may not be the most efficient. It is also in contrast to RL approaches, where they perform more targeted search based on policy gradients that typically need to be estimated from many more coordinate dimensions to be effective. Second, our method takes into account deployment resource constraints upfront and is incorporated into the optimization framework in a more integral way. It is in contrast to other works that address resource constraints using a post-learning approximation strategy like model compression. Third, we demonstrated the effectiveness of our approach in generating competitive solutions on a well-known dataset that generally needs far fewer number of full backpropagation training runs than the above architecture learning techniques. The main goal of the current paper is not to deliberately match state-of-the-art accuracy reported elsewhere, but to demonstrate the feasibility and practicality of the overall constrained network architecture optimization methodology.

3. Methods

Since the overall problem of model architecture selection is formulated as an optimization task, we begin with a description of the objective function we seek to optimize and describe the network architecture representation that we use. We then describe the proposed *Monte Carlo* approach that generates successively refined network architectures via a random sampling procedure before the associated weights are learned. We also discuss a few implementation-specific variants of the overall approach.

3.1. Objective Function

Let A denote the set of CNN architecture parameters (i.e., the architecture specification), and w_k denote the collection of weights assigned to the model after k iterations of backpropagation using a training dataset. Our objective is to select a model architecture A such that, when realized with a set of trained weights, minimizes an objective function in the following form:

$$\hat{A} = \arg\min_{A,\, w_k} J(A, w_k) \quad s.t. \quad \bigwedge_j (\, C_j(A) < \tau_j). \quad (1)$$

The objective function is essentially a weighted sum of a loss term L given the labeled data $\{x_i, y_i\}$ and a regularization term R as shown below:

$$J(A, w_k) = L(A, w_k | \{x_i, y_i\}) \; + \; \beta \cdot R(A), \quad (2)$$

and C_j represents the cost of the jth resource of a particular model architecture, which together with thresholds τ_j represent the hard resource constraint. The loss term measures the cross-entropy error of the model with respect to a labeled data set while the regularization term measures the complexity of the architecture in some way; for example those that favor smaller number weights in the model. The constraint in Equation (1) essentially defines a restricted solution space, for example to meet certain requirements on memory usage and computational budget. To directly

Algorithm I: Constrained Architecture Sampling

```
k ← 0
modelPool ← {}
WHILE k < NRANDOM
  LOOP:
    Ak = GetRandomArch(MODEL_SPACE)
    IF FailValidArch(Ak)
      CONTINUE LOOP
    IF FailConstraint(Ak, THRESH)
      CONTINUE LOOP
    ELSE
      BREAK LOOP
  modelPool ← Append(modelPool, Ak)
  k ← k + 1
```

measure classification accuracy on validation data, L can be formulated as such, which is what we used as stop criteria for backpropagation training and model selection.

3.2. Architecture Representation

While a variety of training frameworks exist, we use a framework-agnostic representation for generality. In such a scheme, each CNN architecture A with C convolutional layers and F fully-connected layers can be represented by an n-tuple descriptor namely: ($size_{input}$, $params_{conv}$, $params_{fc}$, $size_{output}$), where

$$params_{fc} = \left(N_{fc}^{1}, \dots, N_{fc}^{F}\right), \quad (3)$$

$$params_{conv} = \left(var_{cv}^{1}, \dots, var_{cv}^{C}\right), \quad (4)$$

$$var_{cv}^{i} = \left(N_{filt}^{i}, filt_{x}^{i}, filt_{y}^{i}, pool_{x}^{i}, pool_{y}^{i}, sub_{x}^{i}, sub_{y}^{i}\right) \quad (5)$$

Here N_{fc}^{i} and N_{filt}^{i} represent the number of fully-connected (or hidden) layers and filters, whereas $filt^{i}$, $pool^{i}$, and sub^{i} represent the sizes (in x and y directions) of the conv filters, pooling neighborhood and subsampling factor in a particular layer i, respectively. The output of each convolution layer is passed through a standard ReLU activation function and followed by a batch normalization step.

Even though we use this simplified template for CNN architectures for the experiments in the current paper, nothing in the optimization approach we describe later would prevent it from being applied to more complex architecture types as well. Other architecture elements (e.g., skips and braches) could be incorporated in future, but we want to first understand what can be achieved without those generalizations. Nonetheless, a very large architecture space can be represented by the above.

3.3. Architecture Optimization Approach

Since there are no known closed-form solutions for the non-convex architecture optimization problem, we propose

Algorithm II: Adaptive Architecture Learning

```
k ← 0
WHILE k < NADAPT
  (Aopt, scoreopt) ← BestModel(modelPool)
  type ← SelRandom({meta, layer})
  IF type == meta
    LOOP_M:
      Li ← SelRandomLayer(Aopt)
      action = SelRandom({add, remove})
      IF action == remove
        Ak ← RemoveLayer(Aopt, Li)
      IF action == add
        Ak ← AddLayer(Aopt, Randomize(Li))
      IF FailConstraint(Ak, THRESH)
        CONTINUE LOOP_M
      scorek = EvalObjective(Ak)
      modelPool ← Append(modelPool, Ak)
      k ← k + 1
  IF type == layer
    LOOP_L:
      Li ← SelRandomLayer(Aopt)
      α ← SelRandom({inc, dec})
      IF LayerType(Li) == conv
        key ← SelRandomParam({n_filt, …
              filt_sz, pool_sz, sub_sz} )
      IF LayerType(Li) == fc
        key ← n_hidden
      LOOP_DESCENT:
        Val(Li, key)  ← α * Val(Li, key)
        Ak ← UpdateArch(Aopt, Li)
        IF FailConstraint(Ak, THRESH)
          BREAK LOOP_DESCENT
        scorek ← EvalObjective(Ak)
        modelPool ← Append(modelPool, Ak)
        IF scorek > scoreopt
          scoreopt ←  scorek
          k ← k + 1
          CONTINUE LOOP_DESCENT
        ELSE
          k ← k + 1
          BREAK LOOP_DESCENT
```

a stochastic optimization method that depends on two main components: (1) random architecture sampling, and (2) adaptive architecture sampling.

The random sampling step first selects all architecture parameters from a uniform distributions over the possible values defined by an initial model space. There is evidence in the literature that random search actually performed better than deterministic grid search strategies in machine learning problems with a large search space [2]. Our method is a chained sampling process; first the meta-parameters determining layer composition are generated, followed by the layer-specific parameters for each layer. If a resource constraint is optionally provided, the constraint will be evaluated before proceeding (more on constraints

later). If the constraint is not satisfied, another random sample is drawn and the process is repeatedly until the constraint is satisfied. This step draws a total of *NRANDOM* samples and evaluates each with respect to the objective function as described by Algorithm I.

Once random sampling is complete, we proceed to the adaptive sampling step, where information about previous samples and their objective function values are used to determine where each new sample should be taken. It is worth noting that since the existence of some (layer-specific) parameters depends on the values of other meta-parameters, sampling methods that work with fixed-size parameter spaces [24] are not applicable. Instead, we use a coordinate descent formulation that is designed to work in variable-dimension parameter spaces.

The algorithm we apply during the adaptive sampling phase is given by Algorithm II. First we identify the optimal architecture discovered so far. Then it randomly decides whether to modify meta-parameters (by adding or removing layers from the architecture) or modify a layer-specific parameter within a randomly selected layer. In the latter case, we choose a coordinate direction for modifying the parameter value (either increasing or decreasing). If the resulting modification leads to a performance improvement we continue to sample in that direction until improvement stops; this is analogous to performing coordinate descent optimization [19]. In fact, it is a form of stochastic coordinate descent method where only one coordinate dimension is randomly chosen and updated at a time [28]. The adaptive samples can be drawn outside of the initial model space at this stage. The process of alternating between randomly modifying the dimensions of the parameter space to look for improvement and conducting coordinate descent within a fixed-dimension parameter space is repeated multiple times. The routine finishes after evaluating a total of *NADAPT* adaptive samples in this phase.

In contrast to evolutionary algorithms, our algorithm begins with what could be considered random mutations of fit architectures, which is followed by a coordinate descent procedure to focus our use of model architecture evaluations where they are more likely to improve performance.

3.4. Early Assessment of Model Traction

A key insight for efficient optimization is that we do not need to let backpropagation weight training run until convergence to assess the fitness of a candidate model architecture. Typically, error backpropagation will make multiple passes (or iterations) through the training data until the accuracy measured on the training data or validation data levels off. However, the accuracy measured at early iterations in the process can be considered an indicator of model "traction," or likelihood to converge to high accuracy. There is a significant advantage to assessing early indicators during training, since it saves on the computational load required to explore the model architecture space. Empirically, we have seen good results even if we run the random sampling step and the initial phase of the adaptive sampling step to only partial convergence.

3.5. Depth-first vs. Breath-first and N-Best Variants

The two algorithm components, namely random sampling and adaptive sampling, are roughly analogous to breadth-first and depth-first search. By altering the ratio of random and initial adaptive architectures, we can make the optimization process more depth-first and less breadth-first or vice versa. *N*-best, by contrast, uses the top *N* architectures for generating subsequent architectures. This is in contrast to the baseline method, which only looks at the current top architecture. This serves as somewhat of a hybrid between depth- and breadth-first, in that multiple "threads" of depth-first search are pursued at the same time, reducing the possibility of selecting an architecture that leads to a shallow local optimum.

3.6. Parallel and Asynchronous Operations

In a parallel asynchronous optimization process, all workers that finish optimizing an architecture are immediately given a new one to optimize. This is in contrast to the sequential version, in which architectures are trained and evaluated one at a time. There is no difference algorithmically in the random sampling stage. However, in the adaptive sampling stage, the sequential version is theoretically more optimal (given a fixed number of architecture evaluations) than the parallel asynchronous variant because waiting for one architecture to finish allows the best architecture for the next round to be chosen from a larger pool.

3.7. Sampling with Resource Constraints

We choose to use intrinsic model properties of CNNs (e.g., number of weights, filter size, etc.) to estimate the size and computational efficiency of the networks. Much of the work in the literature use the number of weights as the standard for measuring the size of CNN models. Indeed, the number of weights is the main contributor to both memory and number of floating point operations (FLOP) in CNNs. We implemented a memory and FLOP estimation function that reflects more closely the runtime requirement of CNNs. We can constrain the architecture sampling routine (e.g., in Algorithm I) by specifying these metrics in order to find the best performing architecture that meets a particular hardware constraint. We describe their derivations below.

First, one implicit assumption we make in practice is that

all weights need to be stored in memory for the duration of the classification process in order to minimize the number of memory operations and thus achieve a higher throughput. The number of weights for each convolutional layer $|\vec{\theta}_{conv}|$ is a function of the kernel size f, number of input x, and the number of output features k (including the bias). For fully-connected layers, the number of weights $|\vec{\theta}_{fc}|$ is the product of the number hidden nodes h, and the number of inputs x (including the bias). We estimate the total memory requirement for a particular model, based on the GEMM (General Matrix-Matrix Multiplication) algorithm. By summing the number of weights across the network, and multiplying the sum by the number of bytes per element (B_{pe}), one can obtain a good approximation of the total memory requirement. We further assume that in an optimized implementation of a forward-pass, one would need at least two memory buffers to store the inputs and outputs during computation. We consider a ping-pong buffering approach in which sufficient memory is allocated to store the largest input and output volumes (this memory is reused throughout the network) for all layers l. A better overall memory estimation is therefore given by:

$$mem = B_{pe} \cdot \left(\sum|\vec{\theta}_{conv}| + \sum|\vec{\theta}_{fc}| + \max_{l}(|\overrightarrow{x_l}|, |\overrightarrow{y_l}|) \right) \quad (6)$$

The total FLOP of the network is also a function of the number of weights. In a convolutional layer, we convolve weighted filters with the input volume. The total FLOP for convolutional layers, which is equivalent to the number of weights times the height and width of the input (after scaling by 1 over the stride s in both directions):

$$flop_{conv} = \left(\frac{1}{s} \cdot x_w \cdot x_h\right) \cdot (x_d \cdot f_w \cdot f_h + 1) \cdot k \quad (7).$$

In the fully-connected layers, we perform one fused multiply-add (FMA) for every weight; thus, the total FLOP is equivalent to the number of weights:

$$flop_{fc} = h \cdot (x + 1). \quad (8)$$

Finally, the estimated FLOP for the entire network is obtained by summing the FLOP for all the layers:

$$flop_{net} = \sum_l flop_{conv}(l) \; + \; \sum_l flop_{fc}(l) \quad (9)$$

The current state of our framework provides estimations both the memory and FLOP of CNN models. In practice, metrics such as power consumption of models for a given hardware platform, and the speed of executing a forward pass of the resulting CNN model measured in number of inferences per second (IPS) may be desirable. These measurable metrics could be formulated (empirical or otherwise) as a function of the intrinsic model properties such as memory and FLOP. By doing so, we can directly specify higher-level resource constraints on power and IPS, as well as compute and memory, for embedded platforms and for applications that require a minimum number of IPS. Our algorithm is generalizable to handle various needs by constraining the model search using appropriate metrics to set the desire limits on hardware resource and computational efficiency.

4. Experiments and Results

We conducted a number experiments and compared the outcomes from different variants of our algorithm. We assessed their relative effectiveness and the impacts of resource constraints on our architecture optimization results.

As described in Section 3, each architecture learning experiment performed random sampling of the architecture space, followed by adaptive sampling to refine the selection. In particular, we randomly sampled 50 architectures, followed by 50+20 adaptively sampled architectures: the initial 50 adaptive architectures were only trained on a predefined number of iterations (8 in our case), while the last 20 were trained until full convergence. In all experiments, the available annotated data were generally divided into a set of held-out test data, and the rest were further divided into training and validation data used during individual architecture learning runs. In all experiments, we set β to 0.00005, learning rate to 0.0005, and batch size for normalization to 128. The coordinate descent scaling factor α was set to either 1.5 or 0.66 depending on the direction.

4.1. Unconstrained Optimization Experiments

We start by assessing the performance of several variants of our proposed algorithm with no resource constraints in effect. We used the *CIFAR-10* dataset composed of 60000 32x32 color images with 10 object classes. It contains 50000 training images and 10000 test images [15]. We randomly selected 5000 images from the training set for validation. We applied the same data augmentation process as in [22], where images were flipped with probability of 0.5, padded, cropped into 32x32 sections, and then color enhanced.

Without support for skip connections and branching layers at this time, a reasonable benchmark to compare against is *VGG* [23]. Their work advocated the use of 3x3 convolutions, which is also what we restrict our search space to in this particular experiment. More specifically, we allowed up to 26 convolutional layers and up to 128 filters in each layer, while limiting the size of convolutional kernels up to 3x3 and maxpools up to 2x2 with a stride of 2. For the fully connected layers, we allowed up to 2048 hidden nodes.

We tested the following algorithm variants:

i. *Synchronous:* This is our baseline algorithm where we use 50 random architectures and 50 initial adaptive architectures (with early termination), followed by 20 fully-trained adaptive architectures. The best architecture so far is used as seed for subsequent

Method Variants	Test Set Accuracy		Resulting Model Resource		
	Model: Train	Model: Train+Val	Param Count	Mem (MB)	Flop (GOP)
Synchronous (1 worker)	88.5%	89.0%	$3.9x10^6$	17.6	.72
Async (4 workers)	88.7%	90.2%	$2.6x10^6$	11.9	.65
Async Ensemble (Best 4)	91.3%	92.6%	$1.1x10^7$	51.0	2.7
N-Best (4 workers)	88.6%	88.8%	$2.7x10^6$	12.8	1.4
VGG-19*	89.9%	90.0%*	$2.1x10^7$	82.7	.40

Table 1: Comparing different variants of our architecture learning method on *CIFAR-10*. **VGG-19* based on our implementation and is reachable using our architecture description.

adaptive samples. There is no parallelization and only one worker thread is used after the first 50 random samples.

ii. *Asynchronous:* Similar to variant *i* except that there are multiple worker threads allowing multiple architectures to be trained in parallel (4 in our case). An adaptive sample is generated based the current best model without waiting for other training runs to finish.

iii. *Asynchronous Ensemble:* Similar to variant *ii* except that top *N* models are fused at the very end (4 in our case) using a majority voting scheme when performing classification.

iv. *N-Best:* Similar to variant *i* except that *N* architectures are trained at a time (4 in our case), with the next set of *N* architectures not being decided until all *N* architectures finish training. It also entails some coordination overhead and idle cycles for some GPUs.

All CNN models as-of-yet described were trained on the training data. Validation data was used for evaluating model generalization during architecture optimization. However, once a final architecture has been decided, an additional model can be trained using all training data including the validation data, which tends to boost performance on independent test data.

The results for these experiments can be seen in Table 1. The best non-ensemble method (Async 4 workers) achieved an accuracy of 90%, which matched the performance of the *VGG-19* model [23], which we implemented and trained using our architecture generation framework. The result was 2% lower than the best published results of 92% for *VGG-19*, which was probably due to the fact that we did not employ weight inheritance, a technique that was found to boost results by over 2% [22]. Nonetheless, we were able to match or exceed that with our ensemble classifier derived from the best 4 performers (final and interim) of the 120 produced by a single run of our overall procedure. This was an interesting result on its own because our optimization process seemed to be producing complementary classifiers along its way. The other algorithm variants turned out to have similar performance, averaging in the high eighty-percent, including the *N*-best variant.

While keeping the total number of random samples and initial adaptive samples the same, adjusting the percent mix of random samples vs. initial adaptive samples was found to produce models with different sizes but only minor difference in accuracy (all within about +/-1% in a separate assessment). Overall, these observations suggested that the asynchronous baseline version of our algorithm is a reasonable choice as it leverages parallel computing resource effectively, and using a 50/50 mix for random sampling and initial adaptive sampling stages provides a good balance between accuracy and model size.

Our results currently had not reached the 95% level of accuracy of the model produced by Real *et al.* [22]. This is because we are currently limiting the "space of all architectures" reachable by our model representation in this first attempt. We expect to reach higher accuracy when we support branching layers or skip connections like those found in architectures defined by *GoogLeNet* or *ResNet*.

However, it is worth noting that the authors in [22] trained thousands of architectures for tens of thousands of steps, whereas we only trained 120 architectures, all but the last 20 for less than 3000 iterations. Our results thus incurred drastically less time and computation. Our learning algorithms involving stochastic coordinate descent was found to be practical (compared to the more expensive form of gradient descent in RL methods) and can produce competitive models with 4 Titan-X class GPUs in about 12 hours, as opposed to requiring hundreds to thousands of GPUs.

We currently used a first-available compute-node allocation strategy for the parallel version of our algorithm, although we may want to consider other strategies in future as we scale up our experiments in heterogeneous (mixed CPU/GPU) cluster environments [14].

4.2. Constrained Optimization Experiments

The goal of this experiment is to assess the performance of our network architecture learning algorithm on *CIFAR-10* data when subject to resource constraints. We used the asynchronous version of the algorithm, but rejected randomly sampled architectures which did not meet resource constraints and continued to generate new ones until constraints were met. In the adaptive stage, we similarly generated architectures until a sufficiently constrained architecture was created, but we also chose a different coordinate dimension to alter the model whenever

Target Const-raints (%)	Test Set Accuracy	Resulting Model Resource				
	Model: Train	Param Count	Mem (MB)	Flop (GOP)	%Mem (actual)	% Flop (actual)
100	88.7%	$2.6x10^6$	11.36	0.65	100	100
90	88.3%	$2.1x10^6$	8.52	0.2	75.0	30.8
70	87.4%	$1.5x10^6$	7.78	0.2	68.5	30.8
50	87.0%	$8.0x10^5$	4.57	0.12	40.2	18.5
30	85.5%	$3.6x10^5$	2.37	0.04	20.9	6.2
10	84.4%	$1.4x10^5$	1.02	0.04	9.0	6.2

Table 2: Performance results on CIFAR-10 subject to varying target resource constraints as % of the unconstrained model. We did not include validation data in the final model training as we only wanted to understand relative trends here.

an architecture was rejected.

We were able to automate the design of CNN models subject to a resource constraint and arriving at optimized models with little loss of accuracy while respecting the constraints. Table 2 shows the relationship between network size reduction and its impact on accuracy, and optimized networks with varying target constraints (for both memory and FLOP in our experiments) given as a percent of the unconstrained baseline. For instance, we produced a CNN model with no less than 2% loss of accuracy when both constraints were 50% of the unconstrained model, and a model with only 5% loss of accuracy when the constraint was only at one tenth.

It is important to mention that the target constraint only provides an upper bound and that the resulting model from the constrained optimization generally will not hit the target constraint exactly. In fact, this is the case for all constrained optimization runs, as one can only define upper (and lower) bounds, which are sufficient to get the desired results.

The memory constraint satisfaction is actually non trivial and reflects realistic memory allocation in optimized runtime software as opposed to just model size. We also experimented with alternative constraint satisfaction strategies by allowing the constraint to be more relaxed in the random sampling stage (e.g., 2X constraint), and linearly reduced to the target constraint through the adaptive sampling stage until the last iteration. We thought we could arrive at a better local minimum solution (higher accuracy), but turned out there was little accuracy impact. That informed the choice of our simpler constraint satisfaction strategy.

The runtime performance of two selected models along with the *VGG-19* model are also reported in Table 3. The unconstrained model learned from *CIFAR-10* (labeled *CF100*) is 1.9X more efficient on Tegra TX1 compared to *VGG-19* with similar or better accuracy, and the resultant

Target Model	Test Set Accuracy	Resulting Model Resource			Performance (Infer/sec)	
	Model: Train+Val	Param Count	Mem (MB)	Flop (GOP)	Titan Xp	Tegra TX1
CF100	90.2%	$2.6x10^6$	11.36	0.65	1246	354
CF50	87.8%	$0.8x10^6$	4.57	0.12	1745	452
VGG-19	90.0%	$21x10^6$	78.91	0.4	328	183

Table 3: Comparative results of architecture learning on CIFAR-10 subject to varying resource constraints. Our model met explicit constraints and actually ran faster with little loss in accuracy. For reference, the accuracy of our best unconstrained ensemble model (with $1.1x10^7$ parameters) is at 92.6%.

model at a target constraint of 50% (labeled *CF50*) is 2.5X more efficient with only about 2% loss in accuracy.

It is also worth noting that as compared to *VGG-19*, *CF100* measured higher in IPS despite being more demanding based on FLOP. This is due to the sizeable difference in the number of memory operations between the two networks (see Table 3). Typically, GPU memory operations (Global Memory access in particular) are more expensive than ALU (arithmetic logic unit) operations [18]. In addition, memory operations consume much more power than ALU [12]. This has let us to place more emphasis on reducing memory operations in CNNs in order to increase IPS and reduce power consumption.

5. Conclusions

We demonstrated the efficacy of a novel network architecture learning algorithm that has the ability to learn competitive deep CNN models subject to optional but hard resource constraints at deployment time. We found that a combination of random sampling and adaptive sampling of the constrained architecture space can be effective and practical in finding good solutions for the corresponding large-scale variable-dimension parameter optimization problem; it allows us to automate the design of deep CNN with resource constraints. We plan to build on this initial success and further extend our framework to allow richer network representations.

6. Acknowledgement

We like to thank Jennifer Sloboda and William Pughe for their help with data preparation, parallel and embedded computing assistance.

References

[1] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017

[2] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[3] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. Cox. Hyperopt: A Python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 2015

[4] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang. AdaNet: Adaptive structural learning of artificial neural networks. In *ICML*, 2017

[5] T. Desell. Large scale evolution of convolutional neural networks using volunteer computing. *Genetic and Evolutionary Computation Conference*, 2017

[6] R. Doshi, K-W Hung, L. Liang, and K-H Chiu. Deep learning neural networks optimization using hardware cost penalty. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016

[7] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan. Deep learning with limited numerical precision. In *ICML*, 2015

[8] S. Han, H. Mao, and W. J. Dally. Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR*, 2016

[9] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *NIPS*, 2015

[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016

[11] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: surpassing human-level performance on ImageNet classification, In *ICCV*, 2015

[12] M. Horowitz. Energy table for 45nm process. Stanford VLSI wiki

[13] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5MB model size. 2016

[14] J. Kinnison, N. Kremer-Herman, D. Thain, and W. Scheirer. SHADHO: Massively scalable hardware-aware distributed hyperparameter optimization. arXiv preprint arXiv:1707.014282, 2017

[15] Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical Report, Univ. of Toronto, 2009

[16] Krizhevsky. I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012

[17] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017

[18] NVIDIA. *CUDA C Programming Guide 9.0*, 2017

[19] Y. Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22: 341–362, 2010

[20] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey. Faster CNNs with direct sparse convolutions and guided pruning. In *ICLR*, 2017

[21] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *ECCV*, 2016

[22] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *ICML*, 2017

[23] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *CVPR*, 2015

[24] J. Snoek, H. Larochelle, and R. Adams. Practical Bayesian optimization of machine learning algorithms. In *NIPS*, 2012

[25] S. Srinivas and R. V. Babu. Learning neural network architectures using backpropagation. In *BMVC*, 2016.

[26] Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhouce, and A. Rabinovich. Going deeper with convolutions, In *CVPR*, 2015

[27] Q. Tao, K. Kong, D. Chu, and G. Wu. Stochastic coordinate descent methods for regularized smooth and nonsmooth losses. In *European conference of Machine Learning and Knowledge Discovery in Databases*, 2012

[28] L. Yang, P. Luo, C. Loy, and X. Tang. A large-scale car dataset for fine-grained categorization and verification. In *CVPR*, 2015

[29] Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017