ChamNet: Towards Efficient Network Design through Platform-Aware Model Adaptation

Xiaoliang Dai[1], Peizhao Zhang[2], Bichen Wu[3], Hongxu Yin[1], Fei Sun[2], Yanghan Wang[2], Marat Dukhan[2], Yunqing Hu[2], Yiming Wu[2], Yangqing Jia[2], Peter Vajda[2], Matt Uyttendaele[2], Niraj K. Jha[1]
1: Princeton University, Princeton, NJ
2: Facebook Inc., Menlo Park, CA
3: University of California, Berkeley, Berkeley, CA

Abstract

This paper proposes an efficient neural network (NN) architecture design methodology called Chameleon that honors given resource constraints. Instead of developing new building blocks or using computationally-intensive reinforcement learning algorithms, our approach leverages existing efficient network building blocks and focuses on exploiting hardware traits and adapting computation resources to fit target latency and/or energy constraints. We formulate platform-aware NN architecture search in an optimization framework and propose a novel algorithm to search for optimal architectures aided by efficient accuracy and resource (latency and/or energy) predictors. At the core of our algorithm lies an accuracy predictor built atop Gaussian Process with Bayesian optimization for iterative sampling. With a one-time building cost for the predictors, our algorithm produces state-of-the-art model architectures on different platforms under given constraints in just minutes. Our results show that adapting computation resources to building blocks is critical to model performance. Without the addition of any bells and whistles, our models achieve significant accuracy improvements against state-of-the-art hand-crafted and automatically designed architectures. We achieve 73.8% and 75.3% top-1 accuracy on ImageNet at 20ms latency on a mobile CPU and DSP. At reduced latency, our models achieve up to 8.5% (4.8%) and 6.6% (9.3%) absolute top-1 accuracy improvements compared to MobileNetV2 and MnasNet, respectively, on a mobile CPU (DSP), and 2.7% (4.6%) and 5.6% (2.6%) accuracy gains over ResNet-101 and ResNet-152, respectively, on an Nvidia GPU (Intel CPU).

1. Introduction

Neural networks (NNs) have led to state-of-the-art performance in myriad areas, such as computer vision, speech recognition, and machine translation. Due to the presence of millions of parameters and floating-point operations (FLOPs), NNs are typically too computationally-intensive to be deployed on resource-constrained platforms. Many efforts have been made recently to design compact NN architectures. Examples include NNs presented in [24, 21, 28], which have significantly cut down the computation cost and achieved a much more favorable trade-off between accuracy and efficiency. However, a compact model design still faces challenges upon deployment in real-world applications [33]:

- Different platforms have diverging hardware characteristics. It is hard for a single NN architecture to run optimally on all the different platforms. For example, a Hexagon v62 DSP prefers convolution operators with a channel size that is a multiple of 32, as shown later, whereas this may not be the case on another platform.
- Real-world applications may face very different constraints. For example, real-time video frame analysis may have a very strict latency constraint, whereas Internet-of-Things (IoT) edge device designers may care more about run-time energy consumption for longer battery life. It is infeasible to have one NN that can meet all these constraints simultaneously. This makes it necessary to adapt the NN architecture to the specific use scenarios before deployment.

There are two common practices for tackling these challenges. The first practice is to manually craft the architectures based on the characteristics of a given platform. However, such a trial-and-error methodology might be too time-consuming for large-scale cross-platform NN deployment and may not be able to effectively explore the design space. Moreover, it also requires substantial knowledge of the hardware details and driver libraries. The other practice focuses on platform-aware neural architecture search (NAS) and sequential model-based optimization (SMBO) [37].

Both NAS and SMBO require computationally-expensive network training and measurement of network performance metrics (e.g., latency and energy) throughout the entire search and optimization process. For example, the latency-driven mobile NAS (MNAS) architecture requires hundreds of GPU hours to develop [28], which becomes unaffordable when targeting numerous platforms with various resource budgets. Moreover, it may be difficult to implement the new cell-level structures discovered by NAS because of their complexity [37].

In this paper, we propose an efficient, scalable, and automated NN architecture adaptation methodology. We refer to this methodology as Chameleon. It does not rely on new cell-level building blocks nor does it use computationally-intensive reinforcement learning (RL) techniques. Instead, it takes into account the traits of the hardware platform to allocate computation resources accordingly when searching the design space for the given NN architecture with existing building blocks. This adaptation reduces search time. It employs predictive models (namely accuracy, latency, and energy predictors) to speed up the entire search process by enabling immediate performance metric estimation. The accuracy and energy predictors incorporate Gaussian process (GP) regressors augmented with Bayesian optimization and imbalanced quasi Monte-Carlo (QMC) sampling. It also includes an operator latency look-up table (LUT) in the latency predictor for fast, yet accurate, latency estimation. It consistently delivers higher accuracy and less run-time latency against state-of-the-art handcrafted and automatically searched models across several hardware platforms (e.g., mobile CPU, DSP, Intel CPU, and Nvidia GPU) under different resource constraints. For example, on a Samsung Galaxy S8 with Snapdragon 835 CPU, our adapted model yields 1.7% higher top-1 accuracy and 1.75× speedup compared to MnasNet [28].

Our contributions can be summarized as follows:

1. We show that computation distribution is critical to model performance. By leveraging existing efficient building blocks, we adapt models with significant improvements over state-of-the-art hand-crafted and automatically searched models under a wide spectrum of devices and resource budgets.
2. We propose a novel algorithm that searches for optimal architectures through efficient accuracy and resource predictors. At the core of our algorithm lies an accuracy predictor built based on GP with Bayesian optimization that enables a more effective search over a similar space to RL-based NAS.
3. Our proposed algorithm is efficient and scalable. With a one-time building cost, it only takes minutes to search for models under different platforms/constraints, thus making them suitable for large-scale heterogeneous deployment.

2. Related work

Efficient NN design and deployment is a vibrant field. We summarize the related work next.

Model simplication: An important direction for efficient NN design is model simplification. Network pruning [11, 29, 6, 34, 32, 7] has been a popular approach for removing redundancy in NNs. For example, NetAdapt [33] utilizes a hardware-aware filter pruning algorithm and achieves up to 1.2× speedup for MobileNetV2 on the ImageNet dataset [8]. AMC [13] employs RL for automated model compression and achieves 1.53× speedup for MobileNet_v1 on a Titan XP CPU. Quantization [10, 17] has also emerged as a powerful tool for significantly cutting down computation cost with no or little accuracy loss. For example, Zhu et al. [36] show that there is only a 2% top-5 accuracy loss for ResNet-18 when using a 3-bit representation for weights compared to its full-precision counterpart.

Compact architecture: Apart from simplifying existing models, handcrafting more efficient building blocks and operators for mobile-friendly architectures can also substantially improve the accuracy-efficiency trade-offs [18, 30]. For example, at the same accuracy level, MobileNet [15] and ShuffleNet [31] cut down the computation cost substantially compared to ResNet [12] by utilizing depthwise convolution and low-cost group convolution, respectively. Their successors, MobileNetV2 [24] and ShuffleNetV2 [21], further shrink the model size while maintaining or even improving accuracy. In order to deploy these models on different real-world platforms, Andrew et al. propose linear scaling in [15]. This is a simple but widely-used method to accommodate various latency constraints. It relies on thinning a network uniformly at each layer or reducing the input image resolution.

NAS and SMBO: Platform-aware NAS and SMBO have emerged as a promising direction for automating the synthesis flow of a model based on direct metrics, making it more suitable for deployment [14, 22, 3, 19, 20, 4]. For example, MnasNet [28] yields an absolute 2% top-1 accuracy gain compared to MobileNetV2 1.0x with only a 1.3% latency overhead on Google Pixel 1 using TensorFlow Lite. As for SMBO, Stamoulis et al. use a Bayesian optimization approach and reduce the energy consumed for VGG-19 [25] on a mobile device by up to 6×. Unfortunately, it is difficult to scale NAS and SMBO for large-scale platform deployment, since the entire search and optimization needs to be conducted once per network per platform per use case.

3. Methodology

We first give a high-level overview of the Chameleon framework, after which we zoom into predictive models.

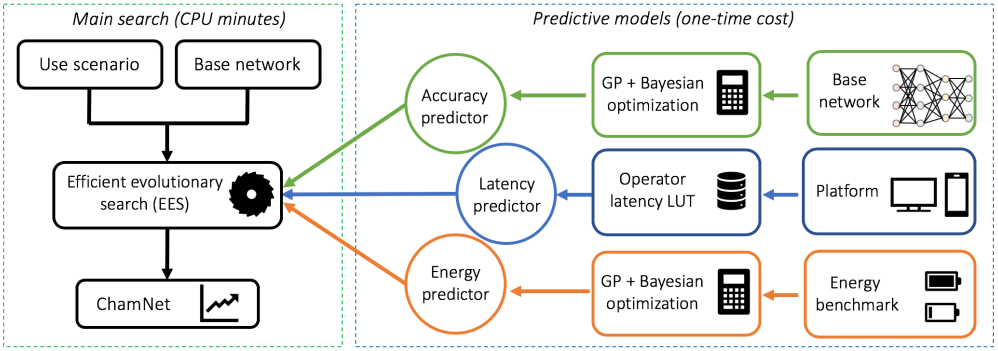


Figure 1. An illustration of the Chameleon adaptation framework

3.1. Platform-aware Model Adaptation

We illustrate the Chameleon approach in Fig. 1. The adaptation step takes a default NN architecture and a specific use scenario (i.e., platform and resource budget) as inputs and generates an adapted architecture as output. Chameleon searches for a variant of the base NN architecture that fits the use scenario through efficient evolutionary search (EES). EES is based on an adaptive genetic algorithm [26], where the gene of an NN architecture is represented by a vector of hyperparameters (e.g., #Filters and #Bottlenecks), denoted as $\mathbf{x} \in \mathbf{R}^n$, where n is the number of hyperparameters of interest. In each iteration, EES evaluates the fitness of each NN architecture candidate based on inputs from the predictive models, and then selects architectures with the highest fitness to breed the next generation using mutation and crossover operators. EES terminates after a pre-defined number of iterations. Finally, Chameleon rests at an adapted NN architecture for the target platform and use scenario.

We formulate EES as a constrained optimization problem. The objective is to maximize accuracy under a given resource constraint on a target platform:

$$\text{maximize } A(\mathbf{x}) \tag{1}$$

$$\text{subject to } F(\mathbf{x}, plat) \leq thres \tag{2}$$

where A, F, $plat$, and $thres$ refer to mapping of $\mathbf{x}$ to network accuracy, mapping of $\mathbf{x}$ to the network performance metric (e.g., latency or energy), target platform, and resource constraint determined by the use scenario (e.g., 20ms), respectively. We merge the resource constraint as a regularization term in the fitness function R as follows:

$$R = A(\mathbf{x}) - [\alpha H(F(\mathbf{x}, plat) - thres)]^w \tag{3}$$

where H is the Heaviside step function, and α and w are positive constants. Consequently, the aim is to find the network gene $\mathbf{x}$ that maximizes R:

$$\mathbf{x} = \underset{\mathbf{x}}{argmax}(R) \tag{4}$$

We next estimate F and A to solve the optimization problem. Choices of F depends on constraints of interest. In this work, we mainly study F based on direct latency and energy measurements, as opposed to an indirect proxy, such as FLOPs, that has been shown to be sub-optimal [33]. Thus, for each NN candidate $\mathbf{x}$, we need three performance metrics to calculate its $R(\mathbf{x})$: accuracy, latency, and energy consumption.

Extracting the above metrics through network training and direct measurements on hardware, however, is too time-consuming [19]. To speed up this process, we bypass the training and measurement process by leveraging accuracy, latency, and energy predictors, as shown in Fig. 1. These predictors enable metric estimation in less than one CPU second. We give details of our accuracy, latency, and energy predictors next.

3.2. Efficient Accuracy Predictor

To significantly speed up NN architecture candidate evaluation, we utilize an accuracy predictor to estimate the final accuracy of a model without actually training it. There are two desired objectives of such a predictor:

1. Reliable prediction: The predictor should minimize the distance between predicted and real accuracy, and rank models in the same order as their real accuracy.

2. Sample efficiency: The predictor should be built with as few trained network architectures as possible. This saves computational resources in training instance generation.

Next, we explain how we tackle these two objectives through GP regression and Bayesian optimization based sample architecture selection for training.

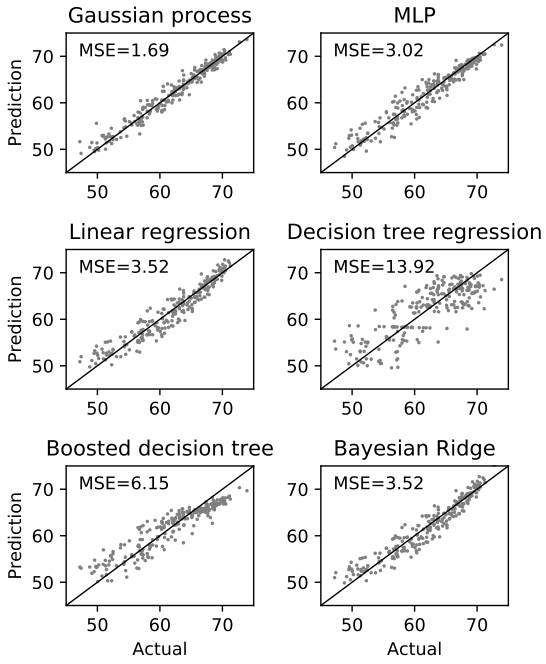


Figure 2. Performance comparison of different accuracy prediction models, built with 240 pretrained models under different configurations. MSE refers to leave-one-out mean squared error.

3.2.1 Gaussian Process Model

We choose a GP regressor as our accuracy predictor to model A as:

$$\begin{aligned} A(\mathbf{x}_i) =& f(\mathbf{x}_i) + \epsilon_i,\ i = 1, 2, ..., s \\ f(\cdot) &\sim \mathcal{GP}(\cdot|0, K) \\ \epsilon_i &\sim \mathcal{N}(\cdot|0, \sigma^2) \end{aligned} \tag{5}$$

where *i* denotes the index of a training vector among *s* training vectors and ϵ_i's refer to noise variables with independent $\mathcal{N}(\cdot|0, \sigma^2)$ distributions. $f(\cdot)$ is drawn from a GP prior characterized by covariance matrix K. We use a radial basis function kernel for K:

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma||\mathbf{x} - \mathbf{x}'||^2) \tag{6}$$

A GP regressor provides two benefits. First, it offers reliable predictions when training data are scarce. As an example, we compare several regression models for MobileNetV2 accuracy prediction in Fig. 2. The GP regressor has the lowest mean squared error (MSE) among all six regression models. Second, a GP regressor produces predictions with uncertainty estimations, which offers additional guidance for new sample architecture selection for training. This helps boost the convergence speed and improves sample efficiency, as shown next.

Algorithm 1 Steps for building an accuracy predictor

Input: k: sample architecture pool size, p: exploration sample count, q: exploitation sample count, e: MSE threshold
Pool = Get k QMC samples from the adaptation search space
All_samples = Randomly select samples from Pool
Train (All_samples)
Predictor = Build a GP predictor with all observations
while Eval(Predictor) $\geq e$ **do**
  for Architecture in (Pool $\setminus$ All_samples) **do**
    $\mathrm{a_i}$ = Predictor.*getAccuracy*(Architecture)
    $\mathrm{v_i}$ = Predictor.*getUncertainty*(Architecture)
    $\mathrm{flop_i}$ = *getFLOP*(Architecture)
  end for
  $\mathrm{S_{explore}}$ = {Architectures with p highest $\mathrm{v_i}$}
  $\mathrm{S_{exploit}}$ = {Architectures with q highest $\frac{\mathrm{a_i}}{\mathrm{flop_i}}$}
  All_samples = All_samples $\cup$ $\mathrm{S_{explore}}$ $\cup$ $\mathrm{S_{exploit}}$
  Train ($\mathrm{S_{explore}} \cup \mathrm{S_{exploit}}$)
  Predictor = Build a GP predictor with all observations
end while
Return Predictor

3.2.2 Iterative Sample Selection

As mentioned earlier, our objective is to train the GP predictor with as few NN architecture samples as possible. We summarize our efficient sample generation and predictor training method in Algorithm 1. Since the number of unique architectures in the adaptation search space can still be large, we first sample representative architectures from this search space to form an architecture pool. We adopt the QMC sampling method [2], which is known to provide similar accuracy to Monte Carlo sampling but with orders of magnitude fewer samples. We then build the accuracy predictor iteratively. In each iteration, we use the current predictor as a guide for selecting additional sample architectures to add to the training set. We train these sample architectures and then upgrade the predictor based on new architecture-accuracy observations.

To improve sample efficiency, we further incorporate Bayesian optimization into the sample architecture selection process. This enables Algorithm 1 to converge faster with fewer samples [27]. Specifically, we select both exploitation and exploration samples in each iteration:

- Exploitation samples: We choose sample architectures with high accuracy/FLOPs ratios. These desirable architectures, or 'samples of interest,' are likely to yield higher accuracy with less computation cost. They typically fall in the top left part of the accuracy-FLOPs trade-off graph, as shown in Fig. 3.

- Exploration samples: We choose samples with large uncertainty values. This helps increase the prediction confidence level of the GP regressor over the entire adaptation search space [27].

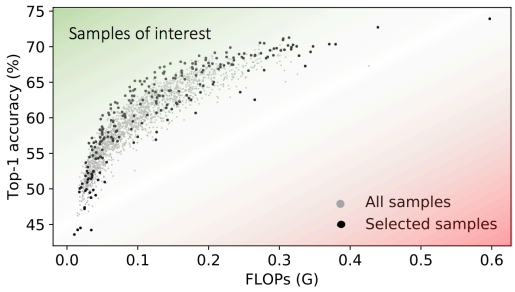
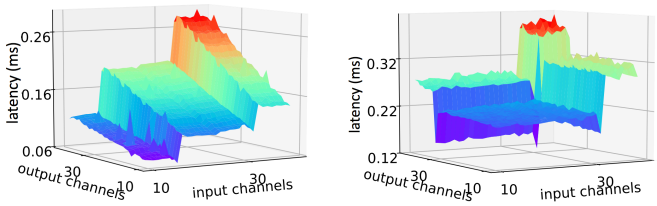


Figure 3. An illustration of 'samples of interest' and sample selection result.



(a) (b)

Figure 4. Latency vs. #Channels for a 1×1 convolution on an input image size of 56×56 and stride 1 on (a) Snapdragon 835 GPU and (b) Hexagon v62 DSP.

Based on these rules, we show the selected sample architectures from the architecture space in Fig. 3. It can be observed that we have higher sampling density in the area of 'samples of interest,' where adaptation typically rests.

3.3. Latency Predictor

Recently, great efforts have been made towards developing more efficient and compact NN architectures for better accuracy-latency trade-offs. Most of them optimize NNs based on FLOPs, which is an often-used proxy [37, 35]. However, optimization based on direct latency measurement instead of FLOPs can better explore hardware traits and hence offer additional advantages. To illustrate this point, we show the measured latency surface of a 1×1 convolution operator with varying numbers of input and output channels in Fig. 4. The latency is measured using the Caffe2 library on a Snapdragon 835 mobile CPU and a Hexagon v62 DSP. It can be observed that FLOPs, though generally effective in providing guidance for latency reduction, may not capture desired hardware characteristics upon model deployment.

Extracting the latency of an NN architecture during EES execution through direct measurement, however, is very challenging. Platform-specific latency measurements can be slow and difficult to parallelize, especially when the number of available devices is limited [33]. Therefore, large-scale latency measurements might be expensive and become the computation bottleneck for Chameleon. To

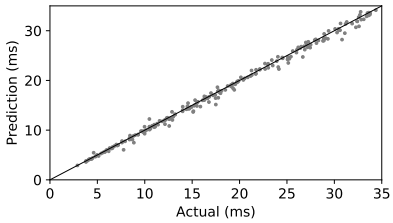


Figure 5. LUT-based latency predictor evaluation on Snapdragon 835 CPU.

speed up this process, we construct an operator latency LUT for the target device to enable fast and reliable latency estimations of NN candidates throughout EES execution. The LUT is supported by an operator latency database, where we benchmark operator-level latency on real devices with different input dimensions. For an NN model, we sum up all its operator-level latencies as an estimate of the network-level latency:

$$t_{net} = \Sigma\, t_{operator} \tag{7}$$

Building the operator latency LUT for a given device is just a one-time cost, but can be substantially reused across various NN models, different tasks, and different applications of architecture search, model adaptation, and hyperparameter optimization.

Latency estimation based on operator latency LUT can be completed in less than one CPU second, as opposed to real measurements on hardware that usually take minutes. Moreover, it also supports parallel query, hence significantly enhancing simultaneous latency extraction efficiency across multiple NN candidates. This enables latency estimation in EES to consume very little time. We compare the predicted latency value against real measurement in Fig. 5. The distance between the predicted value and real measurement is quite small. We plan to open-source the operator latency LUT for mobile CPUs and DSPs.

3.4. Energy Predictor

Battery-powered devices (e.g., smart watches, mobile phones, and AR/VR products) have limited energy budgets. Thus, it is important to adjust and fine-tune the NN model to fit the energy constraint before deployment [5]. To solve this problem, we incorporate energy constraint-driven adaptation in Chameleon for different platforms and use scenarios.

We build the energy predictor in a similar manner to the accuracy predictor. We build a GP energy predictor that incorporates Bayesian optimization and acquire the energy values from direct measurements on hardware. However, for the energy predictor, we only select exploration samples in each iteration (i.e., samples with large uncertainty). The concept of 'samples of interest' is not applicable in this sce-

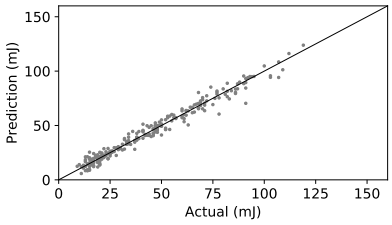


Figure 6. Energy predictor evaluation on Snapdragon 835 CPU.

nario. We show the performance of our energy predictor for MobileNetV2 in Fig. 6.

4. Experiments

In this section, we apply Chameleon to various NN architectures over a wide spectrum of platforms and resource budgets. We use PyTorch [23] and Caffe2 for the implementation. For mobile platforms, we train the model with full precision (float32) and quantize to int8. We report full precision results for accuracy comparison but we see no or minimal loss of accuracy for quantized models with fake-quantization fine-tuning. We benchmark the latency using Caffe2 int8 back-end with Facebook AI Performance Evaluation Platform [1]. We report results on the ImageNet dataset [8], which is a well-known benchmark consisting of 1.2M training and 50K validation images classified into 1000 distinct classes. Thus, all our models share the same 1000 final output channels. We randomly reserve 50K images from the training set (50 images per class) to build the accuracy predictor. We measure latency and energy with the same batch size of 1.

In the EES process, we set $\alpha = 10/\text{ms}$ and $\alpha = 10/\text{mJ}$ for latency- and energy-constrained adaptation, respectively, and $w = 2$. We set the initial QMC architecture pool size to $k = 2048$. We generate the accuracy and energy predictors with 240 samples selected from the architecture pool. Latency estimation is supported by a operator latency LUT with approximately 350K records. In evolutionary search, the population size of each generation is set to 96. We pick the top 12 candidates for the next generation. The total number of search iterations is set to 100. We present our experimental results next.

4.1. Adaptation for Mobile Models

This section presents the adaptation results leveraging the efficient inverse residual building block from MobileNetV2, which is the state-of-the-art hand-crafted architecture for mobile platforms. It utilizes inverted residual and linear bottleneck to significantly cut down on the number of operations and memory needed per inference [24]. We first show the adaptation search space used in our experiments in Table 1, where t, c, n, and s refer to the expansion factor, number of output channels, number of repeated blocks, and stride, respectively. The adaptation search range for each hyperparameter is denoted as $[a, b]$, where a denotes the lower bound and b the upper bound. The default values used in MobileNetV2 1.0x are also shown next to our search ranges, following the notation rule used in [24].

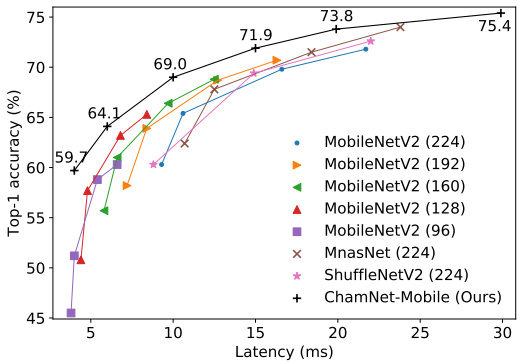


Figure 7. Performance of ChamNet-Mobile on a Snapdragon 835 CPU. Numbers in parentheses indicate input image resolution.

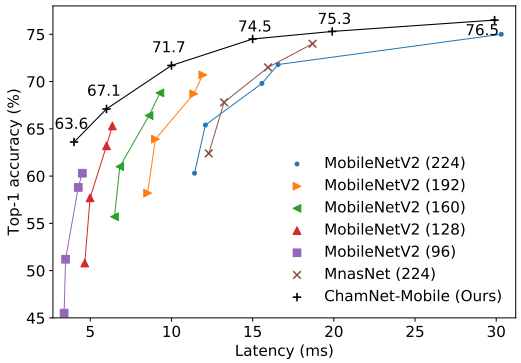


Figure 8. Performance of ChamNet-Mobile on a Hexagon v62 DSP. Numbers in parentheses indicate input image resolution.

Table 1. Adaptation space of ChamNet-Mobile

Input resolution →	224 [96, 224]			
stage	t	c	n	s
conv2d	-	32 [8,48]	1	2
bottleneck	1	16 [8,32]	1	1
bottleneck	6 [2,6]	24 [8,40]	2 [1,2]	2
bottleneck	6 [2,6]	32 [8,48]	3 [1,3]	2
bottleneck	6 [2,6]	64 [16,96]	4 [1,4]	2
bottleneck	6 [2,6]	96 [32,160]	3 [1,3]	1
bottleneck	6 [2,6]	160 [56,256]	3 [1,3]	2
bottleneck	6 [2,6]	320 [96,480]	1	1
conv2d	-	1280 [1024,2048]	1	1
avgpool	-	-	1	-
fc	-	1000	-	-

We target two different platforms in our experiments:

Table 2. Adaptation space of ChamNet-Res

Input resolution →	224			
stage	t	c	n	s
conv2d	-	64 [16,64]	1	2
bottleneck	4 [2,6]	64 [16,64]	3 [1,3]	2
bottleneck	4 [2,6]	128 [32,128]	4 [1,8]	2
bottleneck	4 [2,6]	256 [64,256]	6 [1,36]	2
bottleneck	4 [2,6]	512 [128,512]	3 [1,3]	2
avgpool	-	-	1	-
fc	-	1000	-	-

Snapdragon 835 mobile CPU (on a Samsung S8) and Hexagon v62 DSP (800 MHz frequency with internal NN library implementation). We evaluate Chameleon under a wide range of latency constraints: 4ms, 6ms, 10ms, 15ms, 20ms, and 30ms.

We compare our adapted ChamNet-Mobile models with state-of-the-art models, including MobileNetV2, ShuffleNetV2, and MnasNet in Fig. 7 and 8 on mobile CPU and DSP, respectively. Models discovered by Chameleon outperform all the previous manually designed or automatically searched architectures on both platforms consistently. Our ChamNet has an 8.5% absolute accuracy gain compared to MobileNetV2 0.5x with an input resolution of 96×96, while both models share the same 4.0ms run-time latency on the mobile CPU.

4.2. Adaptation for Server Models

We also evaluate Chameleon for server models on both CPU and GPU. We choose residual building blocks from ResNet as the base for adaptation because of its high accuracy and widespread usage in the cloud. The target platforms are the Intel Xeon Broadwell CPU with 2.4 GHz frequency and Nvidia GTX 1060 GPU with 1.708 GHz frequency. We use CUDA 8.0 and CUDNN 5.1 in our experiments.

We show the detailed adaptation search space in Table 2, where the notations are identical to the ones in Table 1. This search space for ChamNet-Res includes #Filters, expansion factor, and #Bottlenecks per layer. Note that the maximum number of layers in the adaptation search space is 152, which is the largest reported depth for the ResNet family [12].

We set a wide spectrum of latency constraints for both Intel CPU and Nvidia GPU to demonstrate the generality of our framework. The latency constraints for the CPU are 50ms, 100ms, 200ms, and 400ms, while the constraints for the GPU are 2.5ms, 5ms, 10ms, and 15ms. We compare the adapted model with ResNets on CPU and GPU in Fig. 9 and 10, respectively. Again, Chameleon improves the accuracy by a large margin on both platforms.

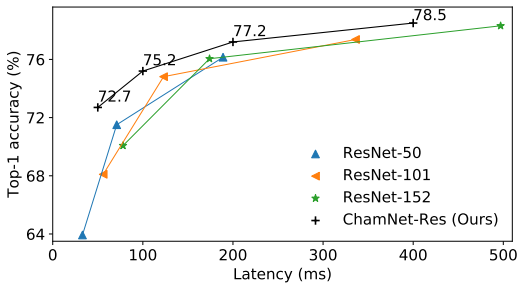


Figure 9. Latency-constrained ChamNet-Res on an Intel CPU.

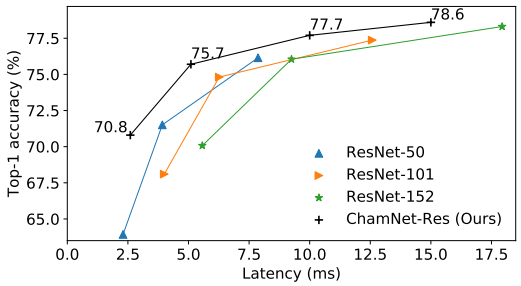


Figure 10. Latency-constrained ChamNet-Res on an Nvidia GPU.

4.3. Energy-Driven Adaptation

Next, we study the energy-constrained use scenario for ChamNet-Mobile on mobile phones. We obtain the energy measurements from the Snapdragon 835 CPU. We first replace the battery of the phone with a Monsoon power monitor with a constant voltage output of 4.2V. During measurements, we ensure the phone is kept in the idle mode for 18 seconds, then run the network 1000 times and measure the current at 200μs intervals. We then deduct the baseline current from the raw data in a post-processing step and calculate the energy consumption per forward pass.

To demonstrate Chameleon's applicability under a wide range of constraints, we set six different energy constraints in our experiment: 15mJ, 30mJ, 50mJ, 75mJ, 100mJ, and 150mJ. We use the same adaptation search space as ChamNet-Mobile.

We show the accuracy of ChamNet and compare it with MobileNetV2 in Fig. 11. We achieve significant improvement in accuracy-energy trade-offs. For example, compared to the MobileNetV2 0.75x with input resolution 96×96 baseline (58.8% accuracy at 19mJ per run), our adapted models achieves 60.0% accuracy at only 14mJ per run. Therefore, our model is able to reduce energy by 26% while simultaneously increasing accuracy by 1.2% on a smartphone.

Table 3. Comparisons of different architecture search and model adaptation approaches.

Model	Method	Direct metrics based	Scaling complexity	Latency(ms)	Top-1 accuracy (%)
MobileNetV2 1.3x [24]	Manual	–	–	33.8	74.4
ShuffleNetV2 2.0x [21]	Manual	Y	–	33.3	74.9
ChamNet-A	**EES**	**Y**	$\mathcal{O}(m+n)$	**29.8**	**75.4**
MobileNetV2 1.0x [24]	Manual	–	–	21.7	71.8
ShuffleNetV2 1.5x [21]	Manual	Y	–	22.0	72.6
CondenseNet (G=C=4) [16]	Manual	–	–	28.7*	73.8
MnasNet 1.0x [28]	RL	Y	$\mathcal{O}(m \cdot n \cdot k)$	23.8	**74.0**
AMC [13]	RL	Y	$\mathcal{O}(m \cdot n \cdot k)$	–	–
MorphNet [9]	Regularization	N	$\mathcal{O}(m \cdot n \cdot k)$	–	–
ChamNet-B	**EES**	**Y**	$\mathcal{O}(m+n)$	**19.9**	73.8
MobileNetV2 0.75x [24]	Manual	–	–	16.6	69.8
ShuffleNetV2 1.0x [21]	Manual	Y	–	**14.9**	69.4
MnasNet 0.75x [28]	RL	Y	$\mathcal{O}(m \cdot n \cdot k)$	18.4	71.5
NetAdapt [33]	Pruning	Y	$\mathcal{O}(m \cdot k + n)$	16.6 (63.6$^+$)	70.9
ChamNet-C	**EES**	**Y**	$\mathcal{O}(m+n)$	15.0	**71.9**
MobileNetV2 0.5x [24]	Manual	–	–	10.6	65.4
MnasNet 0.35x [28]	RL	Y	$\mathcal{O}(m \cdot n \cdot k)$	10.7	62.4
ChamNet-D	**EES**	**Y**	$\mathcal{O}(m+n)$	**10.0**	**69.0**
MobileNetV2 0.35x [24]	Manual	–	–	9.3	60.3
ShuffleNetV2 0.5x [21]	Manual	Y	–	8.8	60.3
ChamNet-E	**EES**	**Y**	$\mathcal{O}(m+n)$	**6.1**	**64.1**

We report five of our ChamNet models with A-30ms, B-20ms, C-15ms, D-10ms, and E-6ms latency constraints. m, n, and k refer to the number of network models, distinct platforms, and use scenarios with different resource budgets, respectively. $^+$: Ref. [33] reports 63.6ms latency with TensorFlow Lite on Pixel 1. For a fair comparison, we report the corresponding latency in our experimental setup with Caffe2 on Samsung Galaxy S8 with Snapdragon 835 CPU. *: The inference engine is faster than other models.

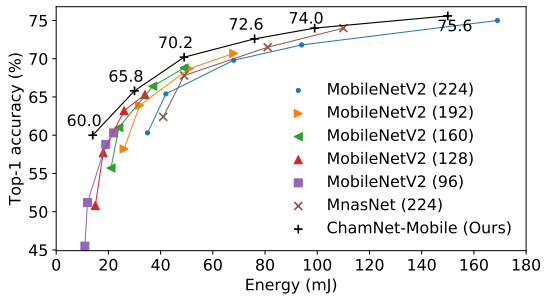


Figure 11. Energy-constrained ChamNet-Mobile on a Snapdragon 835 CPU. Numbers in parentheses indicate input image resolution.

4.4. Comparisons with Alternative Adaptation and Compression Approaches

In this section, we compare Chameleon with relevant work, including:

1. MNAS [28]: this is an RL-based NN architecture search algorithm for mobile devices.
2. AutoML model compression (AMC) [13]: this is an RL-based automated network compression method.
3. NetAdapt [33]: this is a platform-aware filter pruning algorithm that adapts a pre-trained network to a specific hardware under a given latency constraint.
4. MorphNet [9]: this is a network simplification algorithm based on sparsifying regularization.

Table 3 compares different model compression, adaptation, and optimization approaches on the Snapdragon 835 CPU, where m, n, and k refer to the number of network models, distinct platforms, and use scenarios with different resource budgets, respectively. ChamNet yields the most favorable accuracy-latency trade-offs among all models. Moreover, most existing approaches need to be executed at least once per network per device per constraint [28, 33, 9, 13], and thus have a total training cost of $\mathcal{O}(m \cdot n \cdot k)$. Chameleon only builds m accuracy predictors and n resource predictors (e.g., latency LUT), and thus reduces the cost to $\mathcal{O}(m+n)$. The search cost is negligible once the predictors are built. Such one-time costs can easily be amortized when the number of use scenarios scales up, which is generally the case for large-scale heterogeneous deployment.

We compare the FLOPs distribution at each stage (except for avgpool and fc) for MobileNetV2 0.5x and ChamNet with similar latency in Fig. 12. Our model achieves 71.9% accuracy at 15.0ms compared to the MobileNetV2 that has 69.8% accuracy at 16.6ms. We have two observations:

1. ChamNet redistributes the FLOPs from the early stages to late stages. We hypothesize that this is because when feature map size is smaller in the later stages, more filters or a larger expansion factor are needed to propagate the information.

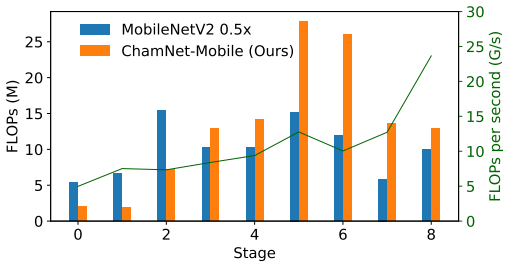


Figure 12. FLOPs distribution and stage-wise CPU processing speed of MobileNetV2 and ChamNet on a mobile CPU.

2. ChamNet has a better utilization of computation resources. We estimate the CPU processing speed at each stage using the ratio of FLOPs and latency, as shown with the green curve in Fig. 12. The operators in early stages with large input image size have significantly lower FLOPs per second, hence incur higher latency given the same computational load. A possible reason is incompatibility between cache capacity and large image size. Through better FLOPs redistribution, ChamNet enables 2.1% accuracy gain while reducing run-time latency by 5% against the baseline MobileNetV2.

5. Conclusions

This paper proposed a platform-aware model adaptation framework called Chameleon that leverages efficient building blocks to adapt a model to different real-world platforms and use scenarios. This framework is based on very efficient predictive models and thus bypasses the expensive training and measurement process. It significantly improves accuracy without incurring any latency or energy overhead, while taking only CPU minutes to perform an adaptation search. At the same latency or energy, it achieves significant accuracy gains relative to both handcrafted and automatically searched models.

References

[1] Facebook AI performance evaluation platform. `https://github.com/facebook/FAI-PEP`, 2018.

[2] S. Asmussen and P. W. Glynn. *Stochastic Simulation: Algorithms and Analysis*, volume 57. Springer Science & Business Media, 2007.

[3] B. Baker, O. Gupta, R. Raskar, and N. Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

[4] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Proc. Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.

[5] 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.

[6] X. Dai, H. Yin, and N. K. Jha. NeST: A neural network synthesis tool based on a grow-and-prune paradigm. *arXiv preprint arXiv:1711.02017*, 2017.

[7] X. Dai, H. Yin, and N. K. Jha. Grow and prune compact, fast, and accurate LSTMs. *arXiv preprint arXiv:1805.11797*, 2018.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 248–255. Ieee, 2009.

[9] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. MorphNet: Fast & simple resource-constrained structure learning of deep networks. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2018.

[10] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[11] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Proc. Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[13] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. AMC: AutoML for model compression and acceleration on mobile devices. In *Proc. European Conf. Computer Vision*, pages 784–800, 2018.

[14] J. M. Hernández-Lobato, M. A. Gelbart, R. P. Adams, M. W. Hoffman, and Z. Ghahramani. A general framework for constrained Bayesian optimization using information-based search. *J. Machine Learning Research*, 17(1):5549–5601, 2016.

[15] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[16] G. Huang, S. Liu, L. van der Maaten, and K. Q. Weinberger. CondenseNet: An efficient DenseNet using learned group convolutions. *arXiv preprint arXiv:1711.09224*, 2017.

[17] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Proc. Advances in Neural Information Processing Systems*, pages 4107–4115, 2016.

[18] 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 <0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[19] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017.

[20] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[21] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. *arXiv preprint arXiv:1807.11164*, 2018.

[22] D. Marculescu, D. Stamoulis, and E. Cai. Hardware-aware machine learning: Modeling and optimization. *arXiv preprint arXiv:1809.05476*, 2018.

[23] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *Proc. Neural Information Processing Systems Workshop on Autodiff*, 2017.

[24] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *arXiv preprint arXiv:1801.04381*, 2018.

[25] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[26] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Trans. Systems, Man, and Cybernetics*, 24(4):656–667, 1994.

[27] D. Stamoulis, E. Cai, D.-C. Juan, and D. Marculescu. Hyperpower: Power-and memory-constrained hyper-parameter optimization for neural networks. In *Proc. IEEE Europe Conf. & Exihibition on Design, Automation & Test*, pages 19–24, 2018.

[28] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. MnasNet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018.

[29] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Proc. Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

[30] B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer. Shift: A zero FLOP, zero parameter alternative to spatial convolutions. *arXiv preprint arXiv:1711.08141*, 2017.

[31] Z. Xiangyu, Z. Xinyu, L. Mengxiao, and S. Jian. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2017.

[32] T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *arXiv preprint arXiv:1611.05128*, 2016.

[33] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam. NetAdapt: Platform-aware neural network adaptation for mobile applications. In *Proc. European Conf. Computer Vision*, volume 41, page 46, 2018.

[34] T. Zhang, K. Zhang, S. Ye, J. Li, J. Tang, W. Wen, X. Lin, M. Fardad, and Y. Wang. ADAM-ADMM: A unified, systematic framework of structured weight pruning for DNNs. *arXiv preprint arXiv:1807.11091*, 2018.

[35] Y. Zhou, S. Ebrahimi, S. Ö. Arık, H. Yu, H. Liu, and G. Diamos. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912*, 2018.

[36] C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[37] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017.