PROXY-LESS ARCHITECTURE SEARCH VIA BINARIZED PATH LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural architecture search (NAS) has great impact by automatically designing efficient neural network architectures. However, the computation demand is prohibitive with conventional NAS algorithms (e.g. 10^4 GPU hours), making it difficult to *directly* search the architecture on large-scale tasks (e.g. ImageNet). As a result, NAS needs to utilize *proxy* tasks, such as training on a smaller dataset (e.g. CIFAR-10), or learning with only a few blocks rather than the full depth, or training only for a few epochs rather than till convergence. These architectures optimized on proxy tasks are not guaranteed to be optimal on target task. In this paper, we present *Proxy-less Architecture Search* that can directly learn the architectures on a large-scale target task. We reduce the computational cost of architecture search by two orders of magnitude (to the same level of normal training). Additionally, by using the measured hardware latency (rather than FLOPs) as a *direct* objective, we can specialize neural network architectures for different hardware architectures. Experiments on CIFAR-10 and ImageNet demonstrate the effectiveness of directness. On CIFAR-10, our model achieves 2.08% test error with only 5.7M parameters, better than the previous state-of-the-art architecture AmoebaNet-B, while using 6$\times$ fewer parameters. On ImageNet, our model achieves 3.1% better top-1 accuracy than MobileNetV2, while being 1.2$\times$ faster with measured GPU latency. Compared with previous architecture search algorithms, we saved the GPU hours by two orders of magnitude, GPU memory by one orders of magnitude, both on ImageNet. We also analyzed the insights of efficient CNN models specialized for different hardware platforms.

1 INTRODUCTION

Recently neural architecture search (NAS) has demonstrated much success in automating neural network architecture design for various deep learning tasks, such as image recognition (Zoph et al., 2018; Cai et al., 2018a; Liu et al., 2018a; Zhong et al., 2018) and language modeling (Zoph & Le, 2017). Despite the remarkable results, conventional NAS algorithms are prohibitively computation-intensive, requiring thousands of models to be trained on the target task in a single experiment. Therefore, directly applying NAS to a large-scale task (e.g. ImageNet) is computationally difficult or impossible. As a trade-off, Zoph et al. (2018) propose to search for building blocks on proxy tasks, such as training for fewer epochs, starting with a smaller dataset (e.g. CIFAR-10), or learning with fewer blocks. Then top-performing blocks are stacked and transferred to the target large-scale task. This paradigm has been widely adopted in subsequent NAS algorithms (Liu et al., 2018a;b; Real et al., 2018; Cai et al., 2018b; Liu et al., 2018c; Tan et al., 2018; Luo et al., 2018).

However, these blocks optimized on proxy tasks are not guaranteed to be optimal on target task. For example, when proxy strategy "fewer epochs" is adopted, lower weight decay may be preferred because of faster convergence on early training stage. But it might lead to a worse final result due to overfitting. Also, competitive blocks on CIFAR-10 can have less compelling results on ImageNet (Liu et al., 2018c). Moreover, to enable transferability, such methods typically search for only a few architectural motifs and then repeatedly stack the same pattern, which restricts the block diversity and thereby harms performance (Tan et al., 2018).

In this work, we propose a simple and effective solution to aforementioned limitations, called *Proxy-less Architecture Search*, which directly learns the architectures on the target task rather with proxy (Figure 1) while allowing blocks to have different structures. To achieve this, we reduce the computation cost (GPU hours and GPU memory) of architecture search to the same level of normal training in the following ways.

Figure 1: Conventional NAS vs our proxy-less NAS.

GPU hour-wise, inspired by recent works (Bender et al., 2018; Liu et al., 2018c), we directly train an over-parameterized network that contains all candidate paths (Figure 2). And during training we explicitly introduce architecture parameters to learn which paths are redundant, while these redundant paths are pruned at the end of training to get a compact optimized architecture. In this way, we simplify architecture search to be a single training process of the over-parameterized network without any meta-controller (or hypernetwork).

However, naively including all the candidate paths leads to GPU memory explosion (Liu et al., 2018c; Bender et al., 2018), as the memory consumption grows linearly w.r.t. the number of choices. Thus, GPU memory-wise, we binarize the architecture parameters (1 or 0) and force only one path to be active at run-time, which reduces the required memory to the same level of training a compact model. These binarized architecture parameters can be optimized via gradient decent, using the BinaryConnect technique (Courbariaux et al., 2015). Furthermore, to handle non-differentiable objectives (e.g. latency), we propose a REINFORCE-based (Williams, 1992) algorithm as an alternative strategy to update these binary parameters.

In our experiments on CIFAR-10 and ImageNet, benefit from the efficiency and directness, our method can achieve strong empirical results. On CIFAR-10, our model reaches 2.08% test error with only 5.7M parameters. On ImageNet, our model achieves 75.1% top-1 accuracy which is 3.1% higher than MobileNetV2 (Sandler et al., 2018) while being 1.2$\times$ faster.

Beside strong empirical results, our method also allows specializing neural network architectures for a certain device. Conventionally, the same network architecture is employed for different hardware architectures. For example, ResNet-50 is popular among various platforms, CPU, GPU and TPUs[1]. However, the specialized network architecture is needed for different hardware architectures to achieve best accuracy-efficiency trade-offs. Our study shows that CPU and GPU have totally different preferences on efficient neural network architectures: CPU prefers slim and deep structures, while GPU prefers wide and shallow ones, which matches the degree of their parallelisms (Figure 4). To the best of our knowledge, this is the first work to study the preferences of different platforms.

2 Related Work

The use of machine learning techniques, such as reinforcement learning or neuro-evolution, to replace human experts in designing neural network architectures, usually referred to as neural architecture search, has drawn an increasing interest (Zoph & Le, 2017; Liu et al., 2018a;b;c; Cai et al., 2018a;b; Pham et al., 2018; Brock et al., 2018; Bender et al., 2018; Elsken et al., 2017; 2018b). In NAS, architecture search is typically considered as a meta-learning process. And a meta-controller (e.g. a recurrent neural network (RNN)), is introduced to explore a given architecture space with training a network in the inner loop to get an evaluation for guiding exploration. Consequently, such methods are computationally expensive to run, especially on large-scale tasks, e.g. ImageNet.

Some recent works (Brock et al., 2018; Pham et al., 2018) try to improve the efficiency of this meta-learning process by reducing the cost of getting an evaluation. In Brock et al. (2018), a hypernetwork is utilized to generate weights for each sampled network and hence can evaluate the architecture without training it. Similarly, Pham et al. (2018) propose to share weights among all sampled networks under the standard NAS framework (Zoph & Le, 2017). These methods speed up architecture search by orders of magnitude, however require a hypernetwork or an RNN controller, thus remain complicated and hard to optimize. Moreover, they mainly focus on small-scale tasks (e.g. CIFAR) rather than large-scale tasks (e.g. ImageNet).

Our work is most closely related to One-Shot (Bender et al., 2018) and DARTS (Liu et al., 2018c), both of which get rid of the meta-controller (or hypernetwork) by modeling architecture search as a single training process of an over-parameterized network that comprises all candidate paths. Specifically, One-Shot trains the over-parameterized network with DropPath (Zoph et al., 2018) that drops out each path with some fixed probability. Then they use the pre-trained over-parameterized network

[1] https://dawn.cs.stanford.edu/benchmark/
https://mlperf.org

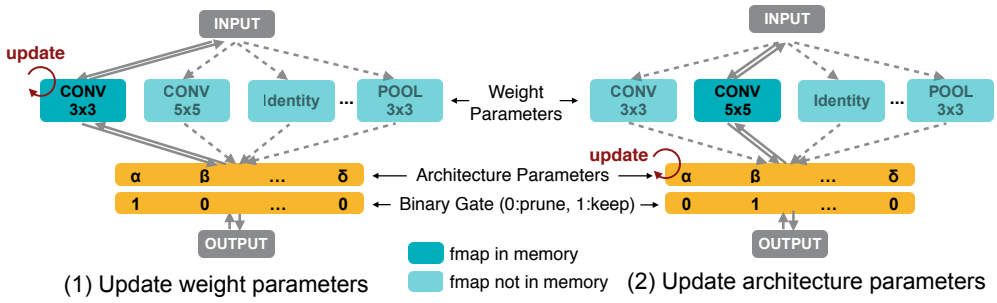


Figure 2: Learning both weight parameters and binarized architecture parameters.

to evaluate architectures, which are sampled by randomly zeroing out paths. DARTS additionally introduces a real-valued architecture parameter for each path and jointly train weight parameters and architecture parameters via standard gradient decent. In this work, we address the large memory issue in these two methods through path binarization.

Another relevant topic is network pruning (Han et al., 2016) that aims to improve the efficiency of neural networks by removing insignificant neurons (Han et al., 2015) or channels (Liu et al., 2017). Similar to these works, we start with an over-parameterized network and then prune the redundant parts to derive the optimized architecture. The distinction is that they focus on layer-level pruning that only modifies the filter (or units) number of a layer but fails to change the topology of the network, while we focus on learning effective network architectures through path-level pruning.

3 Method

In this section, we first describe the construction of the over-parameterized network with all candidate paths, then introduce how we leverage binarized architecture parameters to reduce the memory consumption of training the over-parameterized network to the same level as normal training. Finally, we present two algorithms for learning the binarized architecture parameters, based on BinaryConnect (Courbariaux et al., 2015) and REINFORCE (Williams, 1992) respectively.

3.1 Construction of Over-Parameterized Network

A neural network can be expressed as a directed acyclic graph (DAG), where each edge is associated with an operation $o(\cdot)$ that transforms the feature map. As the lack of connection between two nodes in DAG can be indicated using the zero operation, the task of learning both the connections and operations that are applied at different positions within the network can thereby reduce to choosing operations on its edges (Liu et al., 2018c). For simplicity, we omit nodes and denote a neural network as $\mathcal{N}(e, \cdots, e_n)$ where e_i represents a certain edge in DAG in following discussions.

Let $\mathcal{O} = \{o_i\}$ be the set of N candidate primitive operations (e.g. convolution, pooling, identity, zero, etc). To construct the over-parameterized network that includes any architecture in the search space, instead of setting each edge to be a definite primitive operation, we set each edge to be a mixed operation that has N parallel paths (Figure 2), denoted as $m_{\mathcal{O}}$. As such, the over-parameterized network can be expressed as $\mathcal{N}(e = m_{\mathcal{O}}^1, \cdots, e_n = m_{\mathcal{O}}^n)$. Here, we assume that the candidate set keeps the same throughout the network for ease of discussion, while we can actually arbitrarily adjust the candidate set for each certain position.

Given input x, the output of a mixed operation $m_{\mathcal{O}}$ is defined based on the outputs of its N paths. In One-Shot, $m_{\mathcal{O}}(x)$ is the sum of $\{o_i(x)\}$, while in DARTS, $m_{\mathcal{O}}(x)$ is weighted sum of $\{o_i(x)\}$ where the weights are calculated by applying softmax to N real-valued architecture parameters $\{\alpha_i\}$:

$$
\begin{aligned}
m_{\mathcal{O}}^{\text{One-Shot}}(x) &= \sum_{i=1}^{N} o_i(x), \\
m_{\mathcal{O}}^{\text{DARTS}}(x) &= \sum_{i=1}^{N} p_i o_i(x) = \sum_{i=1}^{N} \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)} o_i(x).
\end{aligned} \tag{1}
$$

To derive the compact optimized architecture from the over-parameterized network, we firstly train the over-parameterized network, then identify and prune $N-1$ redundant paths for each mixed operation. However, straightforwardly training such an over-parameterized network can be problematic. As shown in Eq. (1), the output feature maps of all N paths are calculated and stored in the memory, while training a compact model only involves one path. Therefore, One-Shot and DARTS

roughly need N times GPU memory and GPU hours compared to training a compact model. On large-scale dataset, this can easily exceed the memory limits of hardware with large design space. In the following section, we solve this memory issue based on the idea of path binarization.

3.2 Learning Binarized Path

In this work, inspired by BinaryConnect (Courbariaux et al., 2015), we employ a binarization approach for reducing the cost of training the over-parameterized network. Specifically, similar to DARTS, we also introduce N real-valued architecture parameters $\{\alpha_i\}$, but instead of directly using the real-valued weights $\{p_i\}$ (referred to as path weights), we additionally incorporate a binarization process that transforms the real-valued path weights to binary gates:

$$g = \text{binarize}(p_1, \cdots, p_N) = \begin{cases} [1, 0, \cdots, 0] & \text{with probability } p_1, \\ \cdots & \\ [0, 0, \cdots, 1] & \text{with probability } p_N. \end{cases} \tag{2}$$

Based on the binary gates g, the output of the mixed operation is given as:

$$m_{\mathcal{O}}^{\text{Binary}}(x) = \sum_{i=1}^{N} g_i o_i(x) = \begin{cases} o_1(x) & \text{with probability } p_1 \\ \cdots & \\ o_N(x) & \text{with probability } p_N. \end{cases} . \tag{3}$$

And the expectation of $m_{\mathcal{O}}^{\text{Binary}}(x)$ equals to $m_{\mathcal{O}}^{\text{DARTS}}(x)$:

$$\mathbb{E}[m_{\mathcal{O}}^{\text{Binary}}(x)] = \sum_{i=1}^{N} p_i o_i(x) = m_{\mathcal{O}}^{\text{DARTS}}(x). \tag{4}$$

As illustrated in Eq. (3) and Figure 2, by using the binary gates rather than real-valued path weights (Liu et al., 2018c), only one path of activation is active in memory at run-time and the memory requirement of training the over-parameterized network is thus reduced to the same level of training a compact model.

Notice that setting the binary gate value to 0 corresponds to dropping out the path while setting to 1 corresponds to keeping the path. Therefore, when training the weight parameters of the over-parameterized network (Figure 2 left), we are actually training the network with DropPath regularization (Zoph et al., 2018), as is done in One-Shot (Bender et al., 2018). While in our case, the probability of dropping out the path is not regarded as a hyperparameter but a trainable parameter. Furthermore, we are also not aiming to make the over-parameterized network to be a good reference for ranking sampled architectures but directly learn the architecture by learning architecture parameters. Additionally, we allow only one path to be activated during training, while in One-Shot most of the paths are activated, beginning with all paths activated.

3.3 Training Binarized Architecture Parameters

An illustration of the training procedure of the weight parameters and binarized architecture parameters in the over-parameterized network is provided in Figure 2. When training weight parameters, we freeze the architecture parameters and stochastically sample binary gates according to Eq. (2) for each batch of input data. Then the weight parameters of active paths are updated via standard gradient decent on the training set (Figure 2 left). When training architecture parameters, the weight parameters are fixed, then we reset the binary gates and update the architecture parameters on the validation set (Figure 2 right). These two update steps are performed in an alternating manner. Once the training of architecture parameters is finished, we can then derive the compact architecture by pruning redundant paths. In this work, we simply keep the path with the highest path weight.

Unlike weight parameters, the architecture parameters are not directly involved in the computation graph and thereby cannot be updated using the standard gradient decent. In this section, we introduce two algorithms for learning the architecture parameters.

3.3.1 Gradient-Based Updates

The gradient-based algorithm for learning binarized architecture parameters is built upon BinaryConnect (Courbariaux et al., 2015) that proposes to update the real-valued weight using the gradient w.r.t. its corresponding binary gate. In our case, analogously, the gradients w.r.t. architecture

parameters can be approximately estimated using $\partial L/\partial g_i$ in replace of $\partial L/\partial p_i$:

$$\frac{\partial L}{\partial \alpha_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial p_j}\frac{\partial p_j}{\partial \alpha_i} \approx \sum_{j=1}^{N} \frac{\partial L}{\partial g_j}\frac{\partial p_j}{\partial \alpha_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial g_j}\frac{\partial\left(\frac{\exp(\alpha_j)}{\sum_k \exp(\alpha_k)}\right)}{\partial \alpha_i} = \sum_{j=1}^{N} \frac{\partial L}{\partial g_j} p_j(\delta_{ij} - p_i), \tag{5}$$

where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$. Since the binary gates g are involved in the computation graph, as shown in Eq. (3), $\partial L/\partial g_j$ can be calculated through backpropagation. However, computing $\partial L/\partial g_j$ actually requires to calculate and store $o_j(x)$. Therefore, directly using Eq. (5) to update the architecture parameters would also require roughly N times GPU memory compared to training a compact model.

To address this issue, we consider to factorize the task of choosing one path out of N candidates into multiple binary selection tasks. The intuition is that if a path is the best choice at a certain position, it should be the better choice when solely compared to any other path.

Following this idea, within an update step of the architecture parameters, we first sample two paths according to the multinomial distribution $(p_1, \cdots, p_N)$ and mask all the other paths as if they do not exist. As such the number of candidates temporarily decrease from N to 2, while the path weights $\{p_i\}$ and binary gates $\{g_i\}$ are reset accordingly. Then we update the architecture parameters of these two sampled paths using the gradients calculated via Eq. (5). Finally, as path weights are computed by applying softmax to the architecture parameters, we need to rescale the value of these two updated architecture parameters by multiplying a ratio to keep the path weights of unsampled paths unchanged. As such, in each update step, one of the sampled path is enhanced (path weight increases) and the other sampled path is attenuated (path weight decreases) while all other paths keep unchanged. In this way, regardless of the value of N, only two paths are involved in each update step of the architecture parameters, and thereby the memory requirement is reduced to the same level of training a compact model.

3.3.2 REINFORCE-Based Updates for Non-Differentiable Objectives

The gradient-based algorithm relies on the gradients of the loss function w.r.t. binary gates. But not all objectives are differentiable, such as latency, energy and memory. In this work, we consider another solution for updating binary weights based on REINFORCE (Williams, 1992), as an alternative to BinaryConnect (Courbariaux et al., 2015).

Consider a network that has both weight parameters and binarized architecture parameters α. Here we assume the network only has one mixed operation for ease of illustration. Since only one path will be retained when deriving the final optimized architecture, the goal of updating architecture parameters is actually to find the optimal binary gates g that maximizes a certain reward, denoted as $R(\cdot)$. Therefore, according to REINFORCE (Williams, 1992), we have the following updates for architecture parameters:

$$\begin{aligned} J(\alpha) &= \mathbb{E}_{g\sim\alpha}[R(\mathcal{N}_g)] = \sum_{i=1}^{N} p_i R(\mathcal{N}(e = o_i)), \\ \nabla_\alpha J(\alpha) &= \sum_{i=1}^{N} R(\mathcal{N}(e = o_i))\nabla_\alpha p_i = \sum_{i=1}^{N} R(\mathcal{N}(e = o_i)) p_i \nabla_\alpha \log(p_i), \\ &= \mathbb{E}_{g\sim\alpha}[R(\mathcal{N}_g)\nabla_\alpha \log(p(g))] \approx \frac{1}{M}\sum_{i=1}^{M} R(\mathcal{N}_{g^i})\nabla_\alpha \log(p(g^i)), \end{aligned} \tag{6}$$

where g^i denotes the ith sampled binary gates, $p(g^i)$ denotes the probability of sampling g^i according to Eq. (2) and $\mathcal{N}_{g^i}$ is the compact network according to the binary gates g^i. Since Eq. (6) does not require $R(\mathcal{N}_g)$ to be differentiable w.r.t. g, it can thus handle non-differentiable objectives. An interesting observation is that Eq. (6) has a similar form to the standard NAS (Zoph & Le, 2017), while it is not a sequential decision making process and no RNN meta-controller is used in our case. Furthermore, since both gradient-based updates and REINFORCE-based updates are essentially two different update rules to the same binarized architecture parameters, it is possible to combine them to form a new update rule for the architecture parameters.

4 Experiments and Results

We demonstrate the effectiveness of our proposed method on two benchmark datasets (CIFAR-10 and ImageNet) for the image classification task. Unlike previous NAS works (Zoph et al., 2018; Cai

Model	Params	Test error
DenseNet-BC (Huang et al., 2017)	25.6M	3.46
PyramidNet (Han et al., 2017)	26.0M	3.31
Shake-Shake + c/o (DeVries & Taylor, 2017)	26.2M	2.56
PyramidNet + SD (Yamada et al., 2018)	26.0M	2.31
ENAS + c/o (Pham et al., 2018)	4.6M	2.89
DARTS + c/o (Liu et al., 2018c)	3.4M	2.83
NASNet-A + c/o (Zoph et al., 2018)	27.6M	2.40
PathLevel EAS + c/o (Cai et al., 2018b)	14.3M	2.30
AmoebaNet-B + c/o (Real et al., 2018)	34.9M	2.13
Proxyless-R + c/o (ours)	5.8M	2.30
Proxyless-G + c/o (ours)	5.7M	**2.08**

Table 1: Proxyless NAS achieves state-of-the-art performance on CIFAR-10.

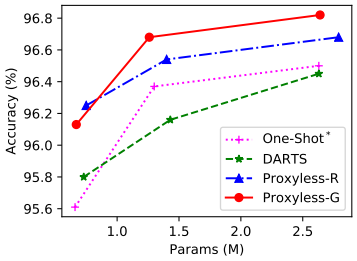


Figure 3: Proxyless NAS outperforms DARTS and One-Shot* on CIFAR-10.

et al., 2018b; Liu et al., 2018c) that first learn CNN blocks on CIFAR-10 under small-scale setting (e.g. fewer blocks), then transfer the learned block to ImageNet or CIFAR-10 under large-scale setting by repeatedly stacking it, we directly learn the architectures on the target task (either Cifar or ImageNet) and allow each block to be different.

4.1 Experiments on CIFAR-10

Architecture Space. For CIFAR-10 experiments, we use the tree-structured architecture space that is introduced by Cai et al. (2018b) with PyramidNet (Han et al., 2017) as the backbone. Specifically, we replace all 3×3 convolution layers in the residual blocks of a PyramidNet with tree-structured cells, each of which has a depth of 3 and the number of branches is set to be 2 at each node (except the leaf nodes). The allocation scheme and merge scheme of a non-leaf node are set to be $replication$ and add respectively. For further details about the tree-structured architecture space, we refer to the original paper (Cai et al., 2018b). In our experiments, instead of using a meta-controller to predict which operation should be chosen at each edge in a tree-structured cell, we set each edge to be a mixed operation over the candidate set[2] and use the proposed method to learn the cell structures. Moreover, by using distinct architecture parameters for different cells, they are free to be have different structures. Additionally, we use two hyperparameters to control the depth and width of a network in this architecture space, i.e. B and F, which respectively represents the number of blocks at each stage (totally 3 stages on CIFAR-10) and the number of output channels of the final block.

Training Details. CIFAR-10 consists of 50,000 training images and 10,000 test images. We randomly sample 5,000 images from the training set as a validation set for learning architecture parameters which are updated using the Adam optimizer with an initial learning rate of 0.006 for gradient-based algorithm (Section 3.3.1) and 0.01 for REINFORCE-based algorithm (Section 3.3.2). In the following discussions, we refer to these two algorithms as **Proxyless-G** (gradient) and **Proxyless-R** (REINFORCE) respectively.

After the training process of the over-parameterized network completes, a compact network is derived according to the architecture parameters, as discussed in Section 3.3. Next we train the compact network using the same training settings except that the number of training epochs increases from 200 to 300. Additionally, when the DropPath regularization (Zoph et al., 2018; Huang et al., 2016) is adopted, we further increase the number of training epochs to 600 (Zoph et al., 2018).

Results. We first apply the proposed method to learn architectures in the tree-structured architecture space with $B = 18$ and $F = 400$. Since we do not force cells to share the same structure and each cell has 12 learnable edges, totally $12 \times 18 \times 3 = 648$ decisions are required to fully determine the architecture. Furthermore, each edge has 7 possible options, so there are $7^{648} \approx 10^{547}$ possible architectures in the entire search space.

The test error rate results of our proposed method and other state-of-the-art architectures on CIFAR-10 are summarized in Table 1, where "c/o" indicates the use of Cutout (DeVries & Taylor, 2017). Compared to these state-of-the-art architectures, our proposed method can achieve not only lower test error rate but also better parameter efficiency. Specifically, Proxyless-G reaches a test error rate of 2.08% which is slightly better than AmoebaNet-B (Real et al., 2018) (the previous best architecture on CIFAR-10). Notably, AmoebaNet-B uses 34.9M parameters while our model only uses 5.7M parameters which is $6\times$ fewer than AmoebaNet-B. Furthermore, compared with PathLevel EAS (Cai et al., 2018b) that also explores the tree-structured architecture space, both Proxyless-G

[2]The list of operations in the candidate set is provided in the appendix.

Model	Top-1	Top-5	GPU latency
MobileNetV2 (Sandler et al., 2018)	72.0	91.0	6.1ms
ShuffleNetV2 (1.5) (Ma et al., 2018)	72.6	-	7.3ms
ResNet-34 (He et al., 2016)	73.3	91.4	8.0ms
NASNet-A Zoph et al. (2018)	74.0	91.3	38.3ms
MnasNet (Tan et al., 2018)	74.0	91.8	6.1ms
Proxyless (GPU)	**75.1**	**92.5**	**5.1ms**

Table 2: Accuracy (%) and measured latency on ImageNet.

Method	GPU hours	GPU memory
NASNet	10^4	$\mathbf{10^1}$
DARTS	$\mathbf{10^2}$	10^2
Mnas	10^4	$\mathbf{10^1}$
Ours	$\mathbf{10^2}$	$\mathbf{10^1}$

Table 3: Proxyless NAS save GPU hours and GPU memory (GB) by 1-2 orders of magnitude on ImageNet.[3]

and Proxyless-R achieves similar or lower test error rate results with half fewer parameters. The strong empirical results of our proposed method demonstrate the benefits of removing the constraint of forcing all cells to be the same and directly exploring a large architecture space instead of repeatedly stacking the same cell which is learned on a small architecture space. It also shows that our proposed method can effectively handle such a huge architecture space.

To further justify our proposed method, we directly compare our method to two most related works, i.e. One-Shot (Bender et al., 2018) and DARTS (Liu et al., 2018c), using the same architecture space under three different settings, i.e. $(B = 4, F = 256)$, $(B = 8, F = 256)$ and $(B = 8, F = 400)$. Additionally, to save time, the DropPath regularization is not used when training all the derived architectures. So they are trained for 300 rather than 600 epochs under all three settings. For DARTS, due to its high GPU memory requirement (Section 3.1), we apply it to learn the cell structure on the first setting, i.e. $(B = 4, F = 256)$, and then transfer the learned cell to the other two settings. For One-Shot, we implement a modified version, denoted as One-Shot*. We set the probability of keeping a path to be $1/N$, where N is the number of candidate operations, rather than a high value in the original paper. As such, One-Shot* can be viewed as a reduced version of our proposed method, where the architecture parameter updates are disabled. Also by setting a low keep probability, which reduces the GPU memory consumption, One-Shot* can be directly applied to all of the three settings.

The results are reported in Figure 3 where we can observe that: (i) Both Proxyless-G and Proxyless-R perform better than One-Shot* under all settings, which demonstrates the importance of architecture parameter updates; (ii) DARTS performs better than One-Shot* on the first setting where DARTS is directly learning on. But on the other two settings, the transferred cell of DARTS performs worse than the directly learned cell of One-Shot*; (iii) Proxyless-G performs worse than Proxyless-R on the first setting while performs better on the other two settings. Notice that the architecture space is larger in the last two settings, we suppose it might because that gradient-based algorithm is more efficient in handling large architecture space than REINFORCE-based algorithms.

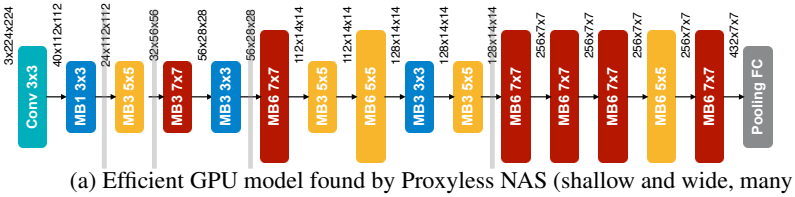
4.2 Experiments on ImageNet

Not all rewards are differentiable. For ImageNet experiments, we aim to learn efficient CNN architectures that not only have high accuracy but also low latency on specific devices (e.g. GPU, CPU, mobile phone, etc.). Therefore, it is a multi-objective NAS task (Hsu et al., 2018; Dong et al., 2018; Elsken et al., 2018a; He et al., 2018; Tan et al., 2018), where one of the objectives is non-differentiable (i.e. latency). Thus, we employ Proxyless-R for ImageNet experiments. Same as Tan et al. (2018), we use $ACC(m) \times [LAT(m)/T]^w$ as the optimization goal, where $ACC(m)$ denotes the accuracy of model m, $LAT(m)$ denotes the latency of m, T is the target latency and w is a hyperparameter for controlling the trade-off between accuracy and latency. We use GPU latency and CPU latency in our experiments as reward. If given mobile farm infrastructure, our algorithm can be easily transferred to the mobile setting[4]. The GPU latency is measured on V100 GPU with a batch size of 8 (single batch will make GPU severely under-utilized). The CPU latency is measured on AWS p3.8 instance with a batch size of 1.

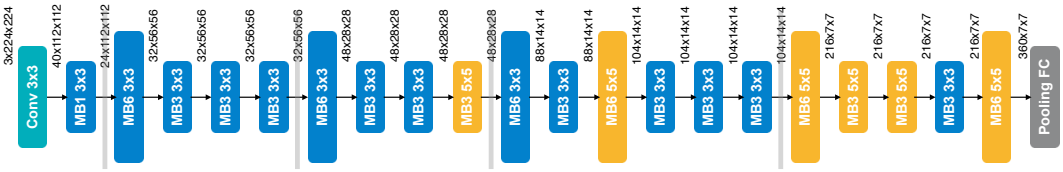
Architecture Space. We use MobileNetV2 (Sandler et al., 2018) as the backbone to build the architecture space. Specifically, rather than repeating the same mobile inverted bottleneck convolution (MBConv), we allow a set of MBConv layers with various kernel sizes $\{3, 5, 7\}$ and expansion ratios $\{3, 6\}$. To enable a direct trade-off between width and depth, we initiate the over-parameterized network to have more blocks than MobileNetV2 and allow a block with the residual connection to be skipped by adding the zero operation to the candidate set of its mixed operation. In this way, with a limited latency budget, the network can either choose to be shallower and wider by skipping more blocks and using larger MBConv layers or choose to be deeper and thinner by keeping more blocks and using smaller MBConv layers.

[3] Further details about the Table 3 is provided in appendix.

[4] In Appendix C, we further extend our method to the mobile setting and remove the need for an expensive mobile farm infrastructure.



(a) Efficient GPU model found by Proxyless NAS (shallow and wide, many 7x7, early pooling).



(b) Efficient CPU model found by Proxyless NAS (deep and thin, only 3x3 and 5x5, late pooling).

Figure 4: Efficient models optimized for different hardware. "MBConv3" and "MBConv6" denote mobile inverted bottleneck convolution layer with an expansion ratio of 3 and 6 respectively.

Model	Top-1 (%)	GPU latency	CPU latency
Proxyless NAS searched on GPU	75.1	5.1ms	204.0ms
Proxyless NAS searched on CPU	75.3	7.4ms	134.8ms

Table 4: Hardware prefers specialized models. Models optimized for GPU does not run fast on CPU, vice versa. Proxyless NAS provide an efficient, automated way to design specialized models for different hardware.

Training Details. We randomly sample 50,000 images from the training set as a validation set during the architecture search. The settings for updating architecture parameters are the same as CIFAR-10 experiments except the initial learning rate is 0.001. The over-parameterized network is trained for 100 epochs on the remaining training images with batch size 256.

ImageNet Classification Results. Table 2 reports the results of our searched architecture with accuracy and GPU latency as objectives. We use the GPU latency of MobileNetV2 as the target latency. Compared to MobileNetV2 and ResNet-34, our model improves the top-1 accuracy by 3.1% and 1.8% respectively while maintaining lower GPU latency. While compared with MnasNet, our model can achieves 1.1% higher top-1 accuracy with 1.2$\times$ fewer GPU latency. More importantly, we are much more sample efficient: the GPU-hour is 100$\times$ fewer than MnasNet (Table 3).

Discussion on Our Searched Models for Different Platforms. Beside GPU latency, we also apply our method to learn architectures with CPU latency as an objective. The results of our GPU model and CPU model are reported in Table 4, while their detailed architectures are provided in Figure 4. We notice that the architecture is totally different when targeting different platforms: (i) The GPU model is shallower and wider, especially in early stages where the feature map has higher resolution; (ii) The GPU model prefers large MBConv operations (e.g. 7×7 MBConv6), while the CPU model would go for smaller MBConv operations. This is because GPU has much higher parallelism than CPU so it can take advantage of large MBConv operations. Beside distinctions, another interesting observation is that both GPU model and CPU model prefer to choose larger MBConv operations in the first block within each stage where the feature map is downsampled. We suppose it might because larger MBConv operations are beneficial for the network to preserve more information when downsampling. Notably, such kind of patterns can not be captured in previous NAS methods as they force the blocks to share the same structure throughout the network (Liu et al., 2018a) or within each stage (Tan et al., 2018).

5 Conclusion

In this work, we introduce Proxy-less Architecture Search, which is able to directly learn architectures on the target task without any proxy. We also reduces the cost (GPU-hours and GPU memory) of NAS to the same level of normal training using path binarization. Benefit from the direct search, we achieve strong empirical results on CIFAR-10 and ImageNet. Furthermore, we allow specializing network architectures for different platforms by directly incorporating the measured hardware latency into optimization objectives. We compared the optimal model on CPU/GPU and raised the awareness that specialized neural network architecture is needed for different hardware architectures.

REFERENCES

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. In *ICLR*, 2018.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, 2018a.

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *ICML*, 2018b.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *ECCV*, 2018.

Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Multi-objective architecture search for cnns. *arXiv preprint arXiv:1804.09081*, 2018a.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018b.

Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep pyramidal residual networks. In *CVPR*, 2017.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NIPS*, 2015.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV*, 2018.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Chi-Hung Hsu, Shu-Huan Chang, Da-Cheng Juan, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Shih-Chieh Chang. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332*, 2018.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018a.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *ICLR*, 2018b.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018c.

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017.

Renqian Luo, Fei Tian, Tao Qin, and Tie-Yan Liu. Neural architecture optimization. *arXiv preprint arXiv:1808.07233*, 2018.

Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*. 1992.

Yoshihiro Yamada, Masakazu Iwamura, and Koichi Kise. Shakedrop regularization. *arXiv preprint arXiv:1802.02375*, 2018.

Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *CVPR*, 2018.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.

A THE LIST OF CANDIDATE OPERATIONS USED ON CIFAR-10

Similar to previous works (Zoph et al., 2018; Cai et al., 2018b; Liu et al., 2018c), we adopt the following 7 operations in our CIFAR-10 experiments:

- 3×3 dilated depthwise-separable convolution
- Identity
- 3×3 depthwise-separable convolution
- 5×5 depthwise-separable convolution
- 7×7 depthwise-separable convolution
- 3×3 average pooling
- 3×3 max pooling

B DETAILS ABOUT TABLE 3

Mnas trains 8,000 mobile-sized models on ImageNet, each of which is trained for 5 epochs for learning architectures. And our method trains a single model for 100 epochs whose cost of training is at the same level as training a mobile-size model. Therefore we use roughly 400× fewer GPU hours than Mnas, while our GPU memory cost also keeps at the same level of training a mobile-sized model. The GPU memory of DARTS is inferred using our architecture space on ImageNet, as the original paper does not directly search for architectures on ImageNet. Additionally, when training a mobile-sized model on ImageNet with a batch size of 64, the GPU memory consumption is roughly 10GB.

C LEARNING SPECIALIZED NEURAL NETWORK ARCHITECTURES FOR MOBILE PHONE

In this section, we present an economical way for learning specialized neural network architectures for mobile phone. By incorporating an accurate mobile latency prediction model, we can use the predicted mobile latency rather than measured mobile latency during architecture search. As such, we remove the need for an expensive mobile farm infrastructure (Tan et al., 2018) that is hard for academic research teams to build.

Model	Top-1	Top-5	Mobile latency
MobileNet V1 (Howard et al., 2017)	70.6	89.5	113ms
MobileNetV2 (Sandler et al., 2018)	72.0	91.0	75ms
NASNet-A (Zoph et al., 2018)	74.0	91.3	183ms
AmoebaNet-A (Real et al., 2018)	74.5	92.0	190ms
MnasNet (Tan et al., 2018)	74.0	91.8	76ms
MnasNet (our impl.)	74.0	91.8	79ms
Proxyless NAS (mobile)	**74.6**	**92.2**	78ms

Table 5: Proxyless NAS achieved state-of-the art accuracy (%) and mobile latency on ImageNet with 100× less search cost in GPU hours.

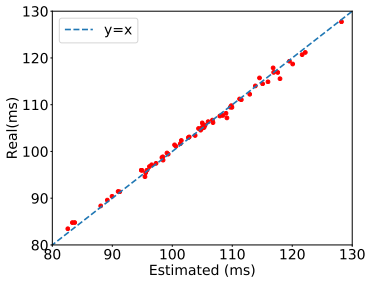


Figure 5: Our mobile latency model is close to $y = x$.

C.1 MOBILE LATENCY PREDICTION

Measuring the latency on-device is accurate but not ideal for scalable neural architecture search. There are two reasons: (i) *Slow.* As suggested in TensorFlow-Lite, we need to average hundreds of runs to produce a precise measurement, approximately 20 seconds. This is far more slower than a single forward / backward execution. (ii) *Expensive.* A lot of mobile devices and engineering work are required to build an automatic pipeline to gather the latency from a mobile farm. Instead of direct measurement, we build a model to estimate the latency. We need only 1 phone rather than a farm of phones, which has only 0.75ms latency RMSE. We use the *latency model* to search, and we use the *measured latency* to report the final model's latency.

We sampled 5k architectures from our candidate space, where 4k architectures are used to build the latency model and the rest are used for test. We measured the latency on Google Pixel 1 phone using TensorFlow-Lite. The features include (i) type of the operator (ii) input and output feature map size (iii) other attributes like kernel size, stride for convolution. As illustrated in Figure 5, we

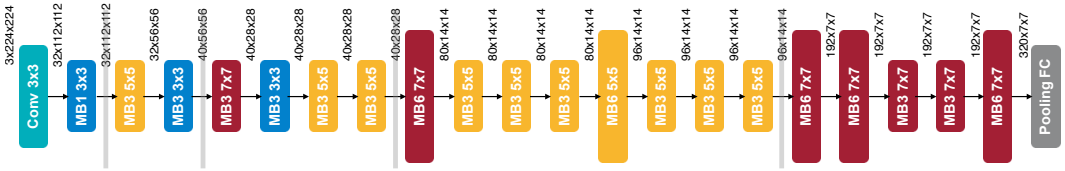


Figure 6: Efficient mobile model discovered by Proxyless NAS.

observe a strong correlation between the predicted latency and real measured latency on the test set, suggesting that the latency prediction model can be used to replace the mobile farm infrastructure with little error introduced.

C.2 Results on Mobile Setting

The mobile latency is measured on Pixel 1 phone with a batch size of 1. The results on the mobile setting are summarized in Table 5 and the detailed architecture of the searched model is illustrated in Figure 6. We find that our model can still achieve superior performances compared to both human-designed and automatically searched architectures. Specifically, our model improves the top-1 accuracy by 2.6% and 0.6% respectively with similar mobile latency on Pixel 1 phone ($<$ 80ms), compared to MobileNetV2 and MnasNet. And compared with NASNet-A and AmoebaNet-A, our model runs 2.4$\times$ faster with slightly higher top-1 accuracy.

D Making Latency Differentiable with Latency Prediction Model

On resource-constrained platforms such as mobile phone and embedded devices, beside accuracy, latency (not FLOPs) is another very important objective when designing efficient neural network architectures (Tan et al., 2018). Unfortunately, unlike accuracy that can be optimized using the gradient of the loss function (e.g. cross entropy loss) (Liu et al., 2018c), latency is a non-differentiable objective. In Section 3.3.2, we have presented a REINFORCE-based approach for handling non-differentiable objectives. In this section, we propose another approach to address this issue. Specifically, we introduce a novel latency regularization loss based on our proposed latency prediction model (see Section C.1). By adding the latency regularization loss to the normal loss function, we are able to directly optimize the trade-off between accuracy and latency in a differentiable manner.

D.1 Latency Regularization Loss

The latency regularization loss is based on a key observation that the expected latency of a mixed operation is actually differentiable w.r.t. architecture parameters. Consider a mixed operation with a candidate set $\{o_j\}$ and each o_j is associated with an architecture parameter p_j which represents the probability (or weights) of choosing o_j. As such, we have the expected latency of a mixed operation (i.e. a learnable block) as:

$$\mathbb{E}[\text{latency}_i] = \sum_j p_j^i \times F(o_j^i), \tag{7}$$

where $\mathbb{E}[\text{latency}_i]$ is the expected latency of the ith learnable block, $F(\cdot)$ denotes the latency prediction model and $F(o_j^i)$ is the predicted latency of o_j^i. The gradient of $\mathbb{E}[\text{latency}_i]$ w.r.t. architecture parameters can thereby be given as:

$$\frac{\partial \mathbb{E}[\text{latency}_i]}{\partial p_j^i} = F(o_j^i). \tag{8}$$

For the whole network with a sequence of mixed operations (Figure 7 left), since these operations are executed in a sequential manner when doing inference, the expected latency of the network can be expressed with the sum of the expected latencies of these mixed operations:

$$\mathbb{E}[\text{latency}] = \sum_i \mathbb{E}[\text{latency}_i], \tag{9}$$

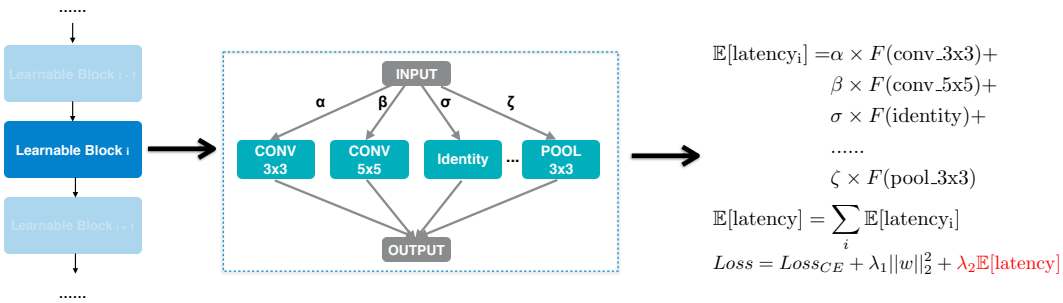


Figure 7: How we make latency differentiable.

where $\mathbb{E}[\text{latency}]$ is the expected latency of the network. As such, $\mathbb{E}[\text{latency}]$ is also differentiable w.r.t. architecture parameters.

We incorporate the expected latency of the network into the normal loss function by multiplying a scaling factor $\lambda_2 (> 0)$ which controls the trade-off between accuracy and latency. The final loss function is given as (also shown in Figure 7 right):

$$Loss = Loss_{CE} + \lambda_1 ||w||_2^2 + \lambda_2 \mathbb{E}[\text{latency}], \tag{10}$$

where $Loss_{CE}$ denotes the cross-entropy loss and $\lambda_1 ||w||_2^2$ is the weight decay term.

D.2 Effectiveness of Latency Regularization Loss

In this section, we present experiments on ImageNet to justify the effectiveness of the proposed latency regularization loss. The summarized results are reported in Table 6, where the results of Proxyless-G without latency regularization loss is also included for comparison.

We observe that Proxyless-G has no incentive to choose computation-cheap operations without latency regularization loss. The resulting architecture initially has 158ms latency on Pixel 1. After rescaling the network using depth multiplier (Sandler et al., 2018; Tan et al., 2018), the latency reduces to 83ms. However, this model can only achieve 71.8% top-1 accuracy on ImageNet, which is 0.2% lower than MobileNetV2. Therefore, we conclude that it is essential to take latency as a *direct* objective when designing efficient (low-latency) neural network architectures.

In contrast, with latency regularization loss, Proxyless-G can make a good trade-off between accuracy and latency. Specifically, with similar latency, it improves the top-1 accuracy on ImageNet by 2.2% and 2.4% respectively, compared to MobileNetV2 and Proxyless-G (without latency loss).

Model	Parameters	FLOPs	Top-1	Top-5	Mobile latency
MobileNetV2 (Sandler et al., 2018)	3.4M	300M	72.0	91.0	75ms
Proxyless-G	2.8M	274M	71.8	90.3	83ms
Proxyless-G + Latency Loss	4.5M	305M	74.2	91.7	79ms

Table 6: Latency regularization loss effectively makes a good trade-off between accuracy and latency. Without it, differentiable NAS has no incentive to choose computation-cheap operations and would fail to reach compelling accuracy results under latency constraints.