UNAS: Differentiable Architecture Search Meets Reinforcement Learning

Arash Vahdat, Arun Mallya, Ming-Yu Liu, Jan Kautz
NVIDIA
{avahdat, amallya, mingyul, jkautz}@nvidia.com

Abstract

Neural architecture search (NAS) aims to discover network architectures with desired properties such as high accuracy or low latency. Recently, differentiable NAS (DNAS) has demonstrated promising results while maintaining a search cost orders of magnitude lower than reinforcement learning (RL) based NAS. However, DNAS models can only optimize differentiable loss functions in search, and they require an accurate differentiable approximation of non-differentiable criteria. In this work, we present UNAS, a unified framework for NAS, that encapsulates recent DNAS and RL-based approaches under one framework. Our framework brings the best of both worlds, and it enables us to search for architectures with both differentiable and non-differentiable criteria in one unified framework while maintaining a low search cost. Further, we introduce a new objective function for search based on the generalization gap that prevents the selection of architectures prone to overfitting. We present extensive experiments on the CIFAR-10, CIFAR-100 and ImageNet datasets and we perform search in two fundamentally different search spaces. We show that UNAS obtains the state-of-the-art average accuracy on all three datasets when compared to the architectures searched in the DARTS [18] space. Moreover, we show that UNAS can find an efficient and accurate architecture in the ProxylessNAS [28] search space, that outperforms existing MobileNetV2 [28] based architectures.

1. Introduction

Since the success of deep learning, designing neural network architectures with desirable performance criteria (*e.g.* high accuracy, low latency, *etc*.) for a given task has been a challenging problem. Some call it alchemy and some refer to it as intuition, but the task of discovering a novel architecture often involves a tedious and costly process of trial-and-error for searching in an exponentially large space of hyper-parameters. The goal of neural architecture search (NAS) [6] is to find novel networks for new problem domains and criteria automatically and efficiently.

Early work on NAS used reinforcement learning [1, 3, 24, 42, 43], or evolutionary algorithms [17, 27, 26, 37] to obtain state-of-the-art performance on a variety of tasks. Although, these methods are generic and can search for architecture with a broad range of criteria, they are often computationally demanding. For example, the RL-based approach [43], and evolutionary method [26] each requires over 2000 GPU days.

Recently, several differentiable neural architecture search (DNAS) frameworks [18, 39, 36, 4] have shown promising results while reducing the search cost to a few GPU days. However, these approaches assume that the objective function is differentiable with respect to the architecture parameters and cannot directly optimize non-differentiable criteria like network latency, power consumption, memory usage, *etc*. To tackle this problem, DNAS methods [36, 4, 40] approximate network latency using differentiable functions. However, these approximations may fail when the underlying criteria cannot be accurately modeled. For example, if compiler optimizations are used, methods such as layer fusion, mixed-precision inference, and kernel auto-tuning can dramatically change latency, making it challenging to approximate it accurately. In addition to the loss approximation, DNAS relies on the continuous approximation of discrete variables in search, introducing additional mismatch in network performance between discovered architecture and the corresponding continuous relaxations.

In this paper, we introduce UNAS, a unified framework for NAS that bridges the gap between DNAS and RL-based architecture search. **(i)** UNAS offers the best of both worlds and enables us to search for architectures using both differentiable objective functions (*e.g.*, cross-entropy loss) and non-differentiable functions (*e.g.*, network latency). UNAS keeps the search time low similar to other DNAS models, but it also eliminates the need for accurate approximation of non-differentiable criteria. **(ii)** UNAS training does not introduce any additional biases due to the continuous relaxation of architecture parameters. We show that the gradient estimation in UNAS is equal to the estimations obtained by RL-based frameworks that operate on discrete variables. Finally, **(iii)** UNAS proposes a new objective function based on the generalization gap which is empirically shown to find

architectures less prone to overfitting.

We perform extensive experiments in both DARTS [18] and ProxylessNAS [4] search spaces. We show that UNAS achieves the state-of-the-art average performance on all three datasets in comparison to the recent gradient-based NAS models in the DARTS space. Moreover, UNAS can find architectures that are faster and more accurate than architectures, searched in the ProxylessNAS space.

1.1. Related Work

Zoph and Le [42] introduced the paradigm of NAS, where a controller recurrent neural network (RNN) was trained to output the specification of a network (filter sizes, number of channels, *etc*.). The controller was trained using REINFORCE [35] to maximize the expected accuracy of the output network on the target validation set, after training on the target task. Requiring the method to specify every layer of the network made it challenging to deepen or transfer an obtained network to other tasks. Based on the observation that popular manually-designed convolutional neural networks (CNNs) such as ResNet [10] or Inception [30] contained repeated generic blocks with the same structure, Zoph *et al*. [43] trained the RNN to output stackable '*cells*'. The task of NAS was thus reduced to learning two types of cells, the *Normal Cell* - convolutional cells that preserve the spatial dimensions, and the *Reduce Cell* - convolutional cells that reduce spatial dimensions while increasing feature maps.

Recently, DARTS [18] relaxed the architecture search space to be continuous by using a weighted mixture-of-operations and optimized the candidate architecture through gradient descent. Using weight-sharing [2, 24], they brought search down to a few GPU days. As the final architecture is required to be discrete, DARTS only retained the top two operations based on the weight assigned to each operation. Building upon DARTS, SNAS [39] used weights sampled from a trainable Gumbel-Softmax distribution instead of continuous weights. Both DARTS and SNAS assume that the objective function for search is differentiable. We extend these frameworks by introducing unbiased gradient estimators that can work for both differentiable and non-differentiable objective functions.

Recent works [4, 36, 40, 11, 31] consider latency in architectures search. ProxylessNAS [4], FBNet [36] and NetAdapt [40] convert the non-differentiable latency objective to a differentiable function by learning an accurate latency approximation. However, these approximations may fail when latency cannot be predicted by a trainable function. MnasNet [31] does not require a differentiable approximation of the latency as it relies on an RL-objective, however, it requires ∼300 TPU-days for each architecture search. Our framework bridges the gap between differentiable and RL-based NAS; it can search with differentiable and non-differentiable functions and it does not require an accurate approximation of non-differentiable terms in the objective. Our work is compared against previous works in Table. 1.

	DARTS [18]	SNAS [39]	P-DARTS [16]	ProxylessNAS [4]	FBNet [36]	MnasNet [31]	UNAS (ours)
Differentiable loss	✓	✓	✓	✓	✓	✓	✓
Non-differentiable loss	✗	✗	✗	✗	✗	✓	✓
Latency optimization	✗	✗	✗	✓	✓	✓	✓
Low search cost	✓	✓	✓	✓	✓	✗	✓

Table 1: Comparison with differentiable NAS methods.

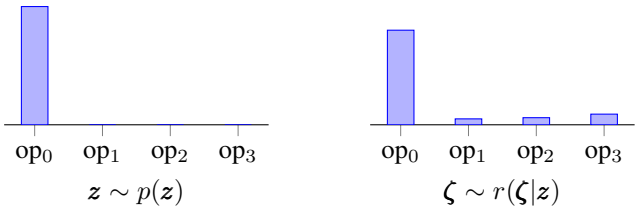
Recently, P-DARTS [5] proposes a progressive version of DARTS and shows that by gradually increasing the depth of the network during the search, deeper cells can be discovered. UNAS explores an orthogonal direction to P-DARTS and it proposes generic gradient estimators that work with both differentiable and non-differentiable losses and new generalization-based search objective functions.

2. Background

In differentiable architecture search (DARTS) [18], a network is represented by a directed acyclic graph, where each node in the graph denotes a hidden representation (*e.g*., feature maps in CNNs) and each directed edge represents an operation transforming the state of the input node. The nth node x_n is connected to its predecessors (*i.e*., P_n) and its content is computed by applying a set of operations to the predeceasing nodes, represented by $x_n = \sum_{x_m \in P_n} O_{m,n}(x_m)$, where $O_{m,n}$ is the operation applied to x_m. The goal of architecture search is then to find the operation $O_{m,n}$ for each edge (m, n). Representing the set of all possible operations that can be applied to the edge $e := (m, n)$ using $\{O_e^{(1)}, O_e^{(2)}, \dots, O_e^{(K)}\}$ where K is the number of operations, this discrete assignment problem can be formulated as a *mixed operation* denoted by $O_e(x_m) = \sum_{k=1}^{K} z_e^{(k)} O_e^{(k)}(x_m)$, where $\boldsymbol{z}_e = [z_e^{(1)}, z_e^{(2)}, \dots, z_e^{(K)}]$ is a one-hot binary vector (*i.e*., $z_e^{(k)} \in \{0, 1\}$) with a single one indicating the selected operation. Typically, it is assumed that the set of operations also includes a *zero* operation that enables omitting edges in the network, and thus, learning the connectivity as well.

We can construct a network architecture given the set of all operation assignments for all edges denoted by $\boldsymbol{z} = \{\boldsymbol{z}_e\}$. Therefore, the objective of the architecture search is to find a distribution over architecture parameters, $\boldsymbol{z}$ such that it minimizes the expected loss $\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}(\boldsymbol{z})]$ where $p_{\boldsymbol{\phi}}$ is a $\boldsymbol{\phi}$-parameterized distribution over $\boldsymbol{z}$ and $\mathcal{L}(\boldsymbol{z})$ is a loss function measuring the performance of the architecture specified by $\boldsymbol{z}$ using a performance measure such as classification loss.

We assume that the architecture distribution is a facto-



(a) categorical sample (b) smoothed sample

Figure 1: (a) Operation selection corresponds to sampling from a categorical distribution that selects an operation. (b) Sampling from the conditional Gumbel-Softmax distribution $r(\boldsymbol{\zeta}|\boldsymbol{z})$ acts as a smoothing distribution that yields continuous samples ($\boldsymbol{\zeta}$), correlated with the discrete samples ($\boldsymbol{z}$).

rial distribution with the form $p_{\boldsymbol{\phi}}(\boldsymbol{z}) = \prod_e p_{\boldsymbol{\phi}_e}(\boldsymbol{z}_e)$ where $p_{\boldsymbol{\phi}_e}(\boldsymbol{z}_e)$ is a $\boldsymbol{\phi}_e$-parameterized categorical distribution defined over the one-hot vector $\boldsymbol{z}_e$. Recently, SNAS [39] proposed using the Gumbel-Softmax relaxation [20, 14] for optimizing the expected loss. In this case, the categorical distribution $p_{\boldsymbol{\phi}}(\boldsymbol{z})$ is replaced with a Gumbel-Softmax distribution $p_{\boldsymbol{\phi}}(\boldsymbol{\zeta})$ where $\boldsymbol{\zeta}$ denotes the continuous relaxation of the architecture parameter $\boldsymbol{z}$. SNAS assumes that the loss $\mathcal{L}(\boldsymbol{z})$ is differentiable with respect to $\boldsymbol{z}$ and it uses the reparameterization trick to minimize the expectation of the relaxed loss $\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{\zeta})}[\mathcal{L}(\boldsymbol{\zeta})]$ instead of $\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}(\boldsymbol{z})]$.

3. Method

As discussed above, the problem of NAS can be formulated as optimizing the expected loss $\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}(\boldsymbol{z})]$. In this section, we present our framework in two parts. In Sec. 3.1, we start by presenting a general framework for computing $\frac{\partial}{\partial \boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}(\boldsymbol{z})]$ which is required for optimizing the expected loss. Then, we present our formulation of the loss function $\mathcal{L}(\boldsymbol{z})$ in Sec. 3.2.

3.1. Gradient Estimation

The most generic approach for optimizing the expected loss is the REINFORCE gradient estimator

$$\frac{\partial}{\partial \boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}(\boldsymbol{z})] = \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[\mathcal{L}(\boldsymbol{z})\partial_{\boldsymbol{\phi}} \log p_{\boldsymbol{\phi}}(\boldsymbol{z})\right], \quad (1)$$

where $\partial \log p_{\boldsymbol{\phi}}(\boldsymbol{z})$ is known as the score function and $\mathcal{L}(\boldsymbol{z})$ is a loss function. As we can see, the gradient estimator in Eq. 1 only requires computing the loss function $\mathcal{L}(\boldsymbol{z})$ (not the gradient $\partial_{\boldsymbol{z}}\mathcal{L}(\boldsymbol{z})$), so it can be applied to any differentiable and non-differentiable loss function. However, this estimator is known to suffer from high variance and therefore a large number of trained architecture samples are required to reduce its variance, making it extremely compute intensive. The REINFORCE estimator in Eq. 1 can be also rewritten as

$$\frac{\partial}{\partial \boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}(\boldsymbol{z})] = \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[(\mathcal{L}(\boldsymbol{z}) - c(\boldsymbol{z}))\,\partial_{\boldsymbol{\phi}} \log p_{\boldsymbol{\phi}}(\boldsymbol{z})] + \partial_{\boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[c(\boldsymbol{z})], \quad (2)$$

where $c(\boldsymbol{z})$ is a control variate [22]. The gradient estimator in Eq. 2 has lower variance than Eq. 1, if $c(\boldsymbol{z})$ is correlated with $\mathcal{L}(\boldsymbol{z})$, and $\partial_{\boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[c(\boldsymbol{z})]$ has a low-variance gradient estimator [21, 23, 25].[1] Without loss of generality, we assume that the loss function is decomposed into $\mathcal{L}(\boldsymbol{z}) = \mathcal{L}_d(\boldsymbol{z}) + \mathcal{L}_n(\boldsymbol{z})$ where $\mathcal{L}_d(\boldsymbol{z})$ contains the terms that are differentiable with respect to $\boldsymbol{z}$ and $\mathcal{L}_n(\boldsymbol{z})$ includes the non-differentiable terms. We present a baseline function $c(\boldsymbol{z}) = c_d(\boldsymbol{z}) + c_n(\boldsymbol{z})$, where $c_d(\boldsymbol{z})$ and $c_n(\boldsymbol{z})$ are for $\mathcal{L}_d(\boldsymbol{z})$ and $\mathcal{L}_n(\boldsymbol{z})$ respectively. Intuitively, the baseline is designed such that the term $\partial_{\boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[c(\boldsymbol{z})]$ in Eq. 2 is approximated using the low-variance reparameterization trick.

Gradient Estimation for Differentiable Loss $\mathcal{L}_d$: Following REBAR [33], in order to construct $c_d(\boldsymbol{z})$, a control variate for $\mathcal{L}_d$, we use stochastic continuous relaxation $r_{\boldsymbol{\phi}}(\boldsymbol{\zeta}|\boldsymbol{z})$ that samples from a conditional Gumbel-Softmax distribution given the architecture sample $\boldsymbol{z}$. Here, $\boldsymbol{\zeta}$ can be considered as a smooth architecture defined based on $\boldsymbol{z}$ as shown in Fig. 1. Hence, it is highly correlated with $\boldsymbol{z}$ (see REBAR [33] for details). With the definition $c_d(\boldsymbol{z}) := \mathbb{E}_{r_{\boldsymbol{\phi}}(\boldsymbol{\zeta}|\boldsymbol{z})}[\mathcal{L}_d(\boldsymbol{\zeta})]$, the gradient in Eq. 2 can be written as

$$\frac{\partial}{\partial \boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_d(\boldsymbol{z})] = \underbrace{\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[(\mathcal{L}_d(\boldsymbol{z}) - c_d(\boldsymbol{z}))\,\partial_{\boldsymbol{\phi}} \log p_{\boldsymbol{\phi}}(\boldsymbol{z})\right]}_{\text{(i) reinforce}} - \underbrace{\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[\partial_{\boldsymbol{\phi}} c_d(\boldsymbol{z})\right]}_{\text{(ii) correction}} + \underbrace{\partial_{\boldsymbol{\phi}}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[c_d(\boldsymbol{z})]}_{\text{(iii) Gumbel-Softmax}}. \quad (3)$$

The gradient estimator in Eq. 3 consists of three terms: (i) is the reinforce term, which is estimated using the Monte Carlo method by sampling $\boldsymbol{z} \sim p_{\boldsymbol{\phi}}(\boldsymbol{z})$ and $\boldsymbol{\zeta} \sim r_{\boldsymbol{\phi}}(\boldsymbol{\zeta}|\boldsymbol{z})$. (ii) is the correction term due to the dependency of $c_d(\boldsymbol{z})$ on $\boldsymbol{\phi}$. This term is approximated using the reparameterization trick applied to the conditional Gumbel-Softmax $r_{\boldsymbol{\phi}}(\boldsymbol{\zeta}|\boldsymbol{z})$. (iii) is the Gumbel-Softmax term that can be written as

$$\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[c_d(\boldsymbol{z})] = \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[\mathbb{E}_{r_{\boldsymbol{\phi}}(\boldsymbol{\zeta}|\boldsymbol{z})}[\mathcal{L}_d(\boldsymbol{\zeta})]\right] = \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{\zeta})}[\mathcal{L}_d(\boldsymbol{\zeta})], \quad (4)$$

which is the expected value of loss evaluated under the Gumbel-Softmax distribution $p_{\boldsymbol{\phi}}(\boldsymbol{\zeta})$. Thus, its gradient can be computed also using the low-variance reparameterization trick. In practice, we only need two function evaluations for estimating the gradient in Eq. 3, one for computing $\mathcal{L}_d(\boldsymbol{z})$, and one for $\mathcal{L}_d(\boldsymbol{\zeta})$. The gradients are computed using an automatic differentiation library.

[1] The low variance of Eq. 2 comes from fact that $\mathrm{Var}(X - Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) - 2\mathrm{Cov}(X, Y)$ for any random variable X and Y. If X and Y are highly correlated the negative contribution from $-2\mathrm{Cov}(X, Y)$ reduces the overall variance of $X - Y$.

Eq. 3 unifies the differentiable architecture search with policy gradient-based NAS methods [42, 39, 31]. This estimator does not introduce any bias due to the continuous relaxation, as in expectation the gradient is equal to the REINFORCE estimator that operates on discrete variables. Moreover, this estimator uses the Gumbel-Softmax estimation of the differentiable loss for reducing the variance of the estimate. Under this framework, it is easy to see that SNAS [39] is a biased estimation of the policy gradient as it only uses (iii) for search, ignoring other terms. On the other hand, policy gradient-based NAS [24, 42, 43] assumes a constant control variate ($c_d(\boldsymbol{z}) = C$) which only requires computing (i) as $\partial_{\boldsymbol{\phi}} \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[C] = 0$.

Gradient Estimation for Non-Differentiable Loss $\mathcal{L}_n$: The gradient estimator in Eq. 3 cannot be applied to non-differentiable loss $\mathcal{L}_n(\boldsymbol{z})$ as the reparameterization trick is only applicable to differentiable functions. For $\mathcal{L}_n(\boldsymbol{z})$, we use RELAX [8] that lifts this limitation by defining the baseline function $c_n(\boldsymbol{z}) := \mathbb{E}_{r_{\boldsymbol{\phi}}(\boldsymbol{\zeta}|\boldsymbol{z})}[g(\boldsymbol{\zeta})]$, where $g(.)$ is a surrogate function (*e.g.*, a neural network) trained to be correlated with $\mathcal{L}_n(\boldsymbol{z})$. The gradient estimator for $\mathcal{L}_n$ is obtained by replacing c_d in Eq. 3 with c_n:

$$\frac{\partial}{\partial \boldsymbol{\phi}} \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_n(\boldsymbol{z})] = \underbrace{\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[(\mathcal{L}_d(\boldsymbol{z}) - c_n(\boldsymbol{z}))\, \partial_{\boldsymbol{\phi}} \log p_{\boldsymbol{\phi}}(\boldsymbol{z})\right]}_{\text{(i) reinforce}} - \underbrace{\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[\partial_{\boldsymbol{\phi}} c_n(\boldsymbol{z})\right]}_{\text{(ii) correction}} + \underbrace{\partial_{\boldsymbol{\phi}} \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{\zeta})}[g(\boldsymbol{\zeta})]}_{\text{(iii) Gumbel-Softmax}}, \quad (5)$$

However, the main difference is that here the reparameterization trick is applied to $\mathbb{E}_{r_{\boldsymbol{\phi}}(\boldsymbol{\zeta}|\boldsymbol{z})}[g(\boldsymbol{\zeta})]$ in (ii) and similarly to $\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{\zeta})}[g(\boldsymbol{\zeta})]$ in (iii). Here, to make $g(\boldsymbol{z})$ be correlated with $\mathcal{L}_n(\boldsymbol{z})$, we train g by minimizing $||g(\boldsymbol{z}) - \mathcal{L}_n(\boldsymbol{z})||_2^2$. In the case of latency, this corresponds to training g to predict latency on a set of randomly generated architectures before search. Similar to FBNet [36] and ProxylessNAS [4], we use a simple linear function to represent $g(\boldsymbol{z})$.

It is worth noting that the Gumbel-Softmax term, (iii) in Eq. 5, minimizes the expectation of the approximation of the non-differentiable loss (e.g., latency) using the Gumbel-Softmax relaxation. This gradient estimator was used in FBNet [36] for optimizing latency. In Eq. 5, we can see that if g cannot predict latency correctly, $\mathcal{L}_d(\boldsymbol{z}) - c_n(\boldsymbol{z})$ will be large, thus, optimizing only (iii) will suffer from additional bias due to the approximation error. However, even if $g(\boldsymbol{z})$ cannot approximate $\mathcal{L}_n(\boldsymbol{z})$ accurately, for example in the case of compile-time performance optimizations, our gradient estimator is equal to the REINFORCE estimator, and it optimizes the true expected latency. Hence, UNAS does not suffer from any bias introduced due to the approximation of non-differentiable criteria.

3.2. Training Objective

Several recent works on differentiable NAS have proposed bi-level training of architecture parameters and network parameters. In the architecture update, either training loss [39], or validation loss [18] given the current network parameters $\boldsymbol{w}$, are used to update architecture parameters using

$$\min_{\boldsymbol{\phi}} \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_{\text{train}}(\boldsymbol{z}, \boldsymbol{w})], \text{ or } \min_{\boldsymbol{\phi}} \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_{\text{val}}(\boldsymbol{z}, \boldsymbol{w})]. \quad (6)$$

Then, the network parameters $\boldsymbol{w}$ are updated given samples from the architecture by minimizing

$$\min_{\boldsymbol{w}} \mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_{\text{train}}(\boldsymbol{z}, \boldsymbol{w})]. \quad (7)$$

The parameters $\boldsymbol{\phi}$ and $\boldsymbol{w}$ are updated iteratively by taking a single gradient step in Eq. 6 and Eq. 7. It has been shown that by sharing network parameters among all the architecture instances, we gain several orders of magnitude speedup in search [18, 24]. However, this comes with the cost of updating architecture parameters at suboptimial $\boldsymbol{w}$. Intuitively, this translates to making decision on architecture without considering its optimal performance.

To avoid overfitting, we base our objective function on the generalization gap of an architecture. The rationale behind this is that the selected architecture not only should perform well on the training set, but also, should generalize equally well to the examples in the validation set, even if network weights are suboptimal. This prevents search from choosing architectures that do not generalize well. Formally, we define the generalization loss in search $\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_{\text{gen}}(\boldsymbol{z}, \boldsymbol{w})]$ by:

$$\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_{\text{train}}(\boldsymbol{z}, \boldsymbol{w}) + \lambda |\mathcal{L}_{\text{val}}(\boldsymbol{z}, \boldsymbol{w}) - \mathcal{L}_{\text{train}}(\boldsymbol{z}, \boldsymbol{w})|], \quad (8)$$

where λ is a scalar balancing the training loss and generalization gap. We observe that $\lambda = 0.5$ often works well in our experiments.[2] For training, we iterate between updating $\boldsymbol{\phi}$ using Eq. 8 and updating $\boldsymbol{w}$ using Eq. 7. In each parameter update, we perform a simple gradient descent update.

Latency Loss: In resource-constrained applications, we might be interested in finding an architecture that has a low latency as well as high accuracy. In this case, we can measure the latency of the network specified by $\boldsymbol{z}$ in each parameter update[3]. Representing the latency of the network using $\mathcal{L}_{lat}(\boldsymbol{z})$, we augment the objective function in Eq. 8 with $\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\lambda_{lat} \mathcal{L}_{lat}(\boldsymbol{z})]$, where λ_{lat} is a scalar balancing the trade-off between the architecture loss and the latency loss. Although $\mathcal{L}_{lat}(\boldsymbol{z})$ is not differentiable w.r.t. $\boldsymbol{z}$, we construct a low-variance gradient estimator using Eq. 5 for optimizing this term.

4. Experiments in DARTS Search Space

In this section, we apply the proposed UNAS framework to the problem of architecture search for image classifica-

[2] We also explored with the objective function without the absolute value, *i.e.*, $\mathcal{L}_{\text{train}}(\boldsymbol{z}, \boldsymbol{w}) + \lambda(\mathcal{L}_{\text{val}}(\boldsymbol{z}, \boldsymbol{w}) - \mathcal{L}_{\text{train}}(\boldsymbol{z}, \boldsymbol{w}))$. We observed that this variants does not perform as good as Eq. 8.

[3] We measure latency on the same hardware that the model is being trained.

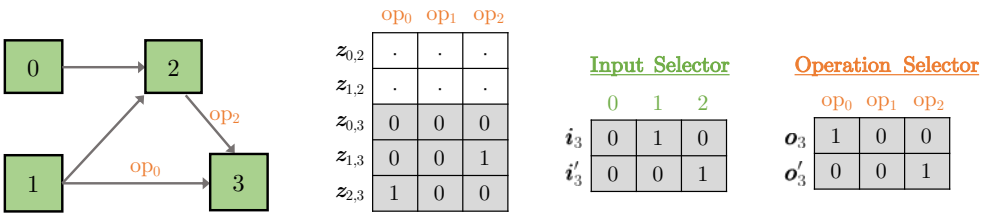


Figure 2: The factorized cell structure ensures that each node depends on two previous nodes. On the left, a small graph with 4 nodes is visualized. In the middle, $\boldsymbol{z} = \{\boldsymbol{z}_e\}$, the operation assignment for the incoming edges to node 3 is shown. On the right, the input and operation selectors for these edges are shown. The shaded matrix on $\boldsymbol{z}$ is parameterized by the outer product $\boldsymbol{i}_3 \otimes \boldsymbol{o}_3 + \boldsymbol{i}'_3 \otimes \boldsymbol{o}'_3$.

tion using DARTS [18] search space, which was also used in [43, 24, 39, 5]. We closely follow the experimental setup introduced DARTS [18]. In the search phase, we search for a normal and reduction cell using a network with a small number of feature maps and/or layers. Given the stochastic representation of the architecture, the final cells are obtained by taking the configuration that has highest probability for each node as discussed below. Then, in the evaluation phase, the cells are stacked into a larger network which is retrained from scratch. Sec. 4.1 discuses a simple approach for factorizing cells that eliminates the necessity of post-search heuristics. Sec. 4.2 provides comparisons to previous work on three datasets.

4.1. Factorized Cell Structure

Training the cell structure introduced in DARTS [18] may result in a densely connected cell where each node depends on the output of all the previous nodes. In order to induce sparsity on the connectivity, prior work [5, 18, 43] heavily relies on post-search heuristics to limit the number of incoming edges for each node. DARTS [18] uses a heuristic to prune the number of input edges to two by choosing operations with the largest weights. P-DARTS [5] uses an iterative optimization to limit the number of skip-connections and the number of incoming edges to two. The main issue with such post-search methods is that they create inconsistency between search and evaluation by constructing a cell structure without directly measuring its performance [39].

In order to explicitly induce sparsity, we factorize the operation assignment problem on the edges using two selection problems: i) an *input selector* that selects two nodes out of the previous nodes and ii) an *operation selector* that selects two operations that are applied to each selected input. We name this structure a *factorized cell* as it enables us to ensure that the content of each node depends only on two previous nodes without relying on any post-search heuristic. Formally, we introduce $\boldsymbol{i}_n$ and $\boldsymbol{i}'_n$, two one-hot vectors for the nth node representing the input selectors as well as two one-hot vectors $\boldsymbol{o}_n$ and $\boldsymbol{o}'_n$ denoting the operation selectors. The architecture is specified by the sets $\{\boldsymbol{i}_n, \boldsymbol{i}'_n\}_{n=1}^{N}$ and $\{\boldsymbol{o}_n, \boldsymbol{o}'_n\}_{n=1}^{N}$, where N is the number of nodes in a cell. This formulation is easily converted to the operation assignment problem on edges (*i.e.* $\{\boldsymbol{z}_e\}$) in Sec. 2 using the outer product $\boldsymbol{i}_n \otimes \boldsymbol{o}_n + \boldsymbol{i}'_n \otimes \boldsymbol{o}'_n$, as shown in Fig. 2. We use the product of categorical distributions in the form $\prod_n p(\boldsymbol{i}_n)p(\boldsymbol{i}'_n)p(\boldsymbol{o}_n)p(\boldsymbol{o}'_n)$ to represent the distribution over architecture parameters.

4.2. Comparison with the Previous Work

The current literature on NAS often reports the final performance obtained by the best discovered cell. Unfortunately, such qualitative metric fails to capture i) the number of searches conducted before finding the best cell, ii) the performance variation resulted from different searches, iii) the effect of each model component on the final performance, and iv) the effect of post-search heuristics used for creating the best architecture. To better provide insights into our framework, we conduct extensive ablation experiments on the CIFAR-10, CIFAR-100 and ImageNet datasets. We run the search and evaluation phases end-to-end four times on each dataset and we report mean and standard deviation of the final test error as well as the best cell out of the four searches. We do not use any post-search heuristic, as our factorized cell structure always yields two-incoming edges per node in the cell. This stands in a stark contrast to DARTS [18] and P-DARTS [5] that use post-search heuristics to sparsify the discovered cell.

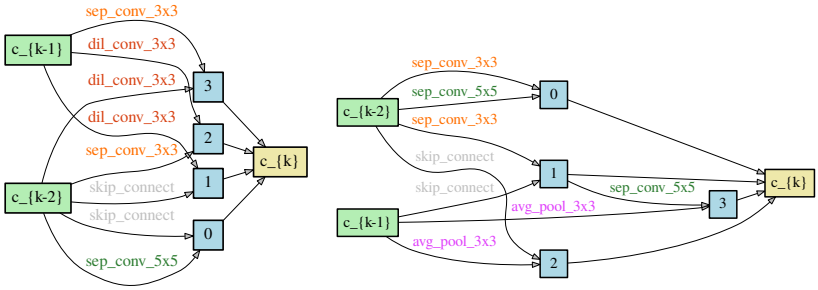
Here, we only consider the differentiable cross-entropy loss functions as the search objective function (*i.e.*, we do not optimize for latency). Since the direct search on ImageNet is computationally expensive, we reduce the search space on this dataset to five operations including skip connection, depthwise-separable 3×3 convolution, max pooling, dilated depthwise-separable 3×3 convolution, and depthwise-separable 3×3 convolution. Prior work on ResNets [10], DenseNets [13], as well as the recent RandWire [38] suggest that it should be possible to achieve high accuracy by using only these three operations.

Below, we discuss the different baselines summarized in Table 2. Additional details of search and evaluation can be found in Appendix A, and Appendix B respectively.

The state-of-the-art: The previous works closest to our

Table 2: Comparison against the state-of-the art methods. Different objective functions for updating architecture parameters and different gradient estimators are examined for UNAS. We run UNAS and the original publicly-available source code for DARTS [18] and P-DARTS [5] end-to-end four times with different initialization seeds. Mean±standard deviation of all four discovered architectures as well as the best architecture at the end of the evaluation phase are reported. For other techniques, the original best results are reported. The search cost is reported on CIFAR-10. UNAS with $\mathcal{L}_{\text{gen}}$ and REBAR significantly outperforms gradient-based methods on all three datasets.

	Objective Function	Gradient Estimator	CIFAR-10 mean	CIFAR-10 best	CIFAR-100 mean	CIFAR-100 best	ImageNet mean	ImageNet best	Search Cost (GPU days)
UNAS	$\mathcal{L}_{\text{val}}$	Gumbel-Soft.	$2.79_{\pm 0.10}$	2.68	$17.11_{\pm 0.38}$	16.80	$26.06_{\pm 0.51}$	25.41	-
	$\mathcal{L}_{\text{gen}}$	Gumbel-Soft.	$2.81_{\pm 0.01}$	2.74	$16.98_{\pm 0.34}$	16.59	$24.64_{\pm 0.13}$	**24.46**	-
	$\mathcal{L}_{\text{gen}}$	REBAR	**2.65**$_{\pm 0.07}$	**2.53**	**16.72**$_{\pm 0.76}$	**15.79**	**24.60**$_{\pm 0.06}$	24.49	4.3
Gradient	DARTS [18]		$3.03_{\pm 0.16}$	2.80	$27.83_{\pm 8.47}$	20.49	$25.27_{\pm 0.06}$	25.20	4
	P-DARTS [5]		$2.91_{\pm 0.14}$	2.75	$18.09_{\pm 0.49}$	17.36	$24.98_{\pm 0.44}$	24.49	0.3
	SNAS [39]		-	2.85	-	-	-	27.3	1.5
Reinforce	NASNet-A [43]		-	2.65	-	-	-	26.0	2000
	BlockQNN [41]		-	3.54	-	18.06	-	-	96
	ENAS [24]		-	2.89	-	-	-	-	0.45
Evolution	AmoebaNet-A [26]		-	3.12	-	-	-	25.5	3150
	AmoebaNet-B [26]		-	2.55	-	-	-	26.0	3150
	AmoebaNet-C [26]		-	-	-	-	-	24.3	3150
	Hierarchical. Evolution [17]		-	3.75	-	-	-	-	300



(a) Normal Cell (b) Reduce Cell

Figure 3: The best performing cell on CIFAR-10.

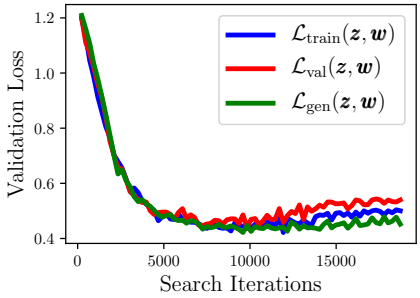


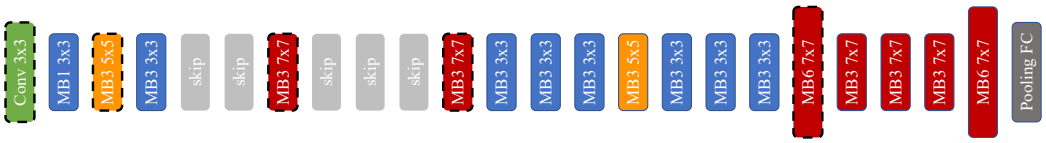
Figure 4: Validation loss in search.

work DARTS [18], P-DARTS [5] and SNAS [39] have unfortunately reported the performance for the best discovered cell. Since DARTS and P-DARTS implementations are publicly available, for a fair comparison, we run their original source code end-to-end four times similar to our model with different random initialization seeds using hyperparameters and commands released by the authors on CIFAR-10 and CIFAR-100.[4] For the ImageNet datasets, we transfer the discovered cells from CIFAR-10 to this dataset as described in [18, 5]. The implementation of SNAS [39] is not publicly available. So, we compare against this work using the original published results. Finally, in order to better contextualize our results, we compare UNAS against previous methods that use reinforcement learning or evolutionary search. On ImageNet, we only consider the mobile-setting (FLOPS $<$ 600M) which is often used to compare NAS models.

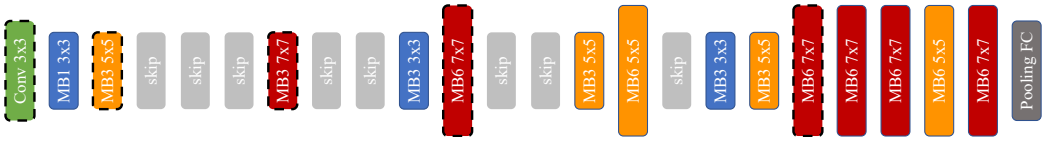
UNAS baselines: We also explore the different variants of UNAS. The objective function column in Table 2 represents the loss function used during search for updating $\boldsymbol{\phi}$. Here, $\mathcal{L}_{\text{val}}$ (Eq. 6) and $\mathcal{L}_{\text{gen}}$ (Eq. 8) are considered. The gradient estimator column represents the gradient estimator used for updating $\boldsymbol{\phi}$ during search. We examine Gumbel-Softmax and REBAR (Eq. 3).

Observations: From the first group of Table 2, we ob-

[4] We exactly followed the hyperparameters and commands using the search/eval code provided by the authors. We only set the initialization seed to a number in $\{0, 1, 2, 3\}$.



(a) Cell discovered by UNAS in the ProxylessNAS [4] search space with 9.8 ms GPU latency and 24.7% top-1 error



(b) Cell discovered by ProxylessNAS [4] with 10.1 ms GPU latency and 24.9% top-1 error

Figure 5: Visualization of the network discovered by UNAS in the ProxylessNAS [4] search space. MBe $K \times K$ denotes a mobile inverted residual block with expansion ratio e and kernel size K. UNAS, in contrast to ProxylessNAS, keeps the cells at the deeper layers (on the right side) computationally inexpensive by using a small expansion ratio, enabling more MBConv layers in the shallower layers. Although UNAS architecture is deeper, it has a lower latency with the same network width.

serve that architecture search with the generalization loss yields a better model often in terms of both average performance and best results. The improvement obtained by the generalization is especially profound in ImageNet as this loss function improves $\mathcal{L}_{\text{val}}$ by 1.4% in average. We can also see that our REBAR gradient estimator often improves the results across all datasets. From the second group of Table 2, we observe that our UNAS framework with REBAR estimator and the generalization loss significantly outperforms DARTS, P-DARTS, and SNAS on all three datasets.[5] Interestingly, our full model ($\mathcal{L}_{\text{gen}}$ with REBAR) exhibits a low variance on CIFAR-10 and ImageNet, showing the robustness of the framework in discovering high-performing architectures. Finally, comparing UNAS against the evolutionary and RL-based models shows that UNAS outperforms these models. The only exception is AmoebaNet-C [26] on ImageNet. However, note that this method requires 700x more GPU time to search.

Why does the generalization loss help in search: Recall that in differentiable architecture search, we often update the architecture distribution parameters $\boldsymbol{\phi}$ using suboptimal $\boldsymbol{w}$. We hypothesize that even if validation loss is used in search due to the suboptimality of $\boldsymbol{w}$, the architecture is not discovered using the the true generalization of the network to unseen examples. To illustrate this, the validation loss during architecture search visualized in Fig. 4 for different loss functions. We observe that even using the validation loss for updating architecture parameters does not prevent the network from overfitting.

One question is whether our generalization loss is required in the case of the original RL-based NAS [24, 42, 43], which updates architecture parameters using $\boldsymbol{w}$ closer to optimality. To answer this, we also examine with ENAS [24]-like training where network parameters $\boldsymbol{w}$ are updated for half epoch in every $\boldsymbol{\phi}$ update (*i.e.*, the network parameters $\boldsymbol{w}$ are brought closer to the optimum). In this case, the architectures found by generalization loss in average obtains test error 2.92% on CIFAR-10 compared with the validation loss based search that achieves 3.12%. This provides another evidence that architecture search can potentially benefit from considering generalization, opening up new research directions in NAS.

Cell visualization: The best cell discovered on the CIFAR-10 dataset is visualized in Fig. 3. See Appendix C for the visualization of best cells on other datasets.

More comparisons: We provide in-depth comparisons against the state-of-the-art techniques with more detailed information including the number of parameters, search cost, and the number of floating point operations in Appendix. D.

5. Experiments on Latency-based Search

In this section, we examine our proposed framework for searching architectures with low latency directly on the ImageNet dataset. Unfortunately, the DARTS search space results in high-latency networks due to the parallel branches and concatenation in each cell. So, here, we change the building blocks of our search space to the mobile inverted bottleneck convolution (MBConv) [28] that has been used in ProxylessNAS [4] and FBNet [36] for discovering low-latency networks. For this section, we closely follow the search space introduced in ProxylessNAS [4] for ImageNet in which a 21-layer network with seven choices of operations in each layer is searched. Specifically, for each layer, an MBConv is selected among various kernel sizes $\{3, 5, 7\}$ and expansion ratios $\{3, 6\}$. To allow layer removal, an additional skip-connection is used in ProxylessNAS yielding seven operations per layer. For search and evaluation

[5] The significance test between UNAS and any other approach passes on all the datasets with p-value < 0.05, except on ImageNet between UNAS and P-DARTS which yields p-value $= 0.18$.

Table 3: Latency-based architecture search. Models are sorted based on their top-1 error. For a better illustration, Fig. 6 compares the models visually.

Architecture	Val Error top-1	Val Error top-5	Latency (ms)
EfficientNet B0 [32]	23.7	6.8	14.5
MobileNetV3 Large [11]	24.7	7.6	11.0
MnasNet-A1 [31]	24.8	7.5	10.9
Single-Path NAS [29]	25.0	7.8	10.2
FBNet-C [36]	25.1	-	11.5
MobileNetV2 1.4x [28]	25.3	7.5	13.0
MnasNet-B1 [31]	25.5	-	9.4
ShuffleNet V2 2x [19]	26.3	-	9.16
MobileNetV2 1x [28]	28.0	9.	9.2
ProxylessNAS-GPU [4]	24.9	7.5	10.1
UNAS	24.7	7.6	9.8

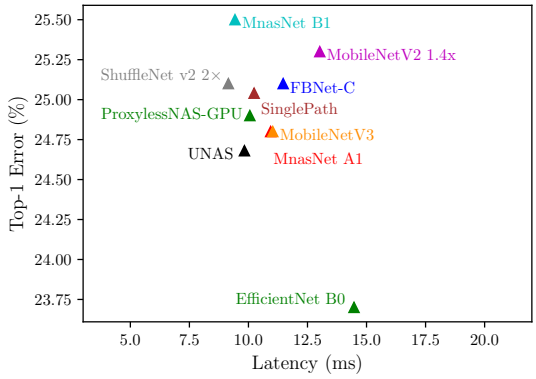


Figure 6: Latency-based architecture search. We seek architectures that are in the bottom-left side of the error vs. latency axes. UNAS discovers an architecture that is more accurate and has a low latency compared to the current state-of-the-art architectures based on MobileNetV2.

we closely follow the settings used in ProxylessNAS (see Appendix E for details).

For gradient estimation of the latency loss in UNAS, we use Eq. 5 with a simple linear function as the surrogate function, *i.e.*, $g(\boldsymbol{z}) = \sum_{i,j} l_{i,j} z_{i,j}$ where $z_{i,j} \in \{0, 1\}$ is a binary scalar indicating if operation i is used in layer j and $l_{i,j}$ is the approximate latency associated with the operation. Similar to ProxylessNAS, we randomly generate 10K network samples before search and we train the parameters of g (*i.e.*, all $l_{i,j}$) by minimizing an L_2 regression loss.

We search for architecture on V100 GPUs, as it allows us to measure the true latency on the device during search. These GPUs were also used in ProxylessNAS [4] which enables us to have a fair comparison against this method. We measure latency using a batch size of 32 images. We empirically observed that smaller batch sizes under-utilize GPUs, resulting in inaccurate latency measurements.

Table 3 and Fig. 6 report the latency and validation set error on ImageNet for our model in comparison to recent hardware-aware NAS frameworks that operate in a similar search space (*i.e.*, MobileNetV2 [28]) and have similar latency (∼10 ms on V100 GPUs). We can see that UNAS finds an architecture that is slightly faster but more accurate than the ProxylessNAS-GPU [4] architecture that uses exactly the same search space and the same target device. EfficientNet B0 [32] is the only architecture that is more accurate than UNAS but it is also 48% slower on the GPU. Although EfficientNet B0 has a low number of mathematical operations, it is not so efficient on TPU/GPU due to the heavy usage of depth-wise separable convolutions [9]. The architectures that are faster than UNAS including ShuffleNet v2 [19], MnasNet B1 [31] and MobileNetV2 1.0x [28] are also less accurate.[6]

In Fig. 5, the architecture discovered by UNAS is compared against ProxylessNAS-GPU that has been discovered for the same type of GPUs. Interestingly, UNAS discovers an architecture that is deeper, *i.e.*, it has 3 more MBConv layers. But, it also faster and more accurate than the architecture discovered by ProxylessNAS.

6. Conclusions

In this paper, we presented UNAS that unifies differentiable and RL-based NAS. Our proposed framework uses the gradient of the objective function for search without introducing any bias due to continuous relaxation. In contrast to previous DNAS methods, UNAS search objective is not limited to differentiable loss functions as it can also search using non-differentiable loss functions. We also introduced a new objective function for search based on the generalization gap and we showed that it outperforms previously proposed training or validation loss functions.

In extensive experiments in both DARTS [18] and ProxylessNAS [4] search spaces, we showed that UNAS finds architectures that 1) are more accurate on CIFAR-10, CIFAR-100, and ImageNet and 2) are more efficient to run on GPUs. We will make our implementation publicly available to facilitate the research in this area.

References

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017. 1

[6] All models are examined in PyTorch. For MobileNetV2 and ShuffleNet V2 the official PyTorch implementations are used. For ProxylessNAS-GPU, the original code provided in [4] is used. Other networks implementations are obtained from EfficientNets repo [34] which are optimized for PyTorch.

[2] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018. 2

[3] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, 2018. 1

[4] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. 1, 2, 4, 7, 8, 14

[5] Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *ICCV*, 2019. 2, 5, 6, 14

[6] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018. 1

[7] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017. 12

[8] Will Grathwohl, Dami Choi, Yuhuai Wu, Geoff Roeder, and David Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. In *ICLR*, 2018. 4

[9] Suyog Gupta and Mingxing Tan. EfficientNet-EdgeTPU. `https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html`, Nov 2019. 8

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 2, 5

[11] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for MobileNetV3. In *ICCV*, 2019. 2, 8

[12] 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. 11, 14

[13] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017. 5

[14] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with Gumbel-Softmax. In *ICLR*, 2017. 3

[15] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019. 13

[16] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive Neural Architecture Search. In *ECCV*, 2018. 2, 13, 14

[17] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *ICLR*, 2018. 1, 6, 13

[18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. In *ICLR*, 2019. 1, 2, 4, 5, 6, 8, 11, 13, 14

[19] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018. 8, 14

[20] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR*, 2017. 3

[21] Andriy Mnih and Karol Gregor. Neural variational inference and learning in belief networks. In *ICML*, 2014. 3

[22] Art B. Owen. *Monte Carlo theory, methods and examples*. 2013. 3

[23] John Paisley, David M Blei, and Michael I Jordan. Variational bayesian inference with stochastic search. In *ICML*, 2012. 3

[24] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018. 1, 2, 4, 5, 6, 7, 13

[25] Rajesh Ranganath, Sean Gerrish, and David Blei. Black box variational inference. In *Artificial Intelligence and Statistics*, 2014. 3

[26] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018. 1, 6, 7, 11, 13, 14

[27] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *ICML*, 2017. 1

[28] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018. 1, 7, 8, 14

[29] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path NAS: Designing hardware-efficient convnets in less than 4 hours. In *arXiv preprint arXiv:1904.02877*, 2019. 8

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015. 2

[31] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 2, 4, 8

[32] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019. 8

[33] George Tucker, Andriy Mnih, Chris J Maddison, John Lawson, and Jascha Sohl-Dickstein. REBAR: Low-variance, unbiased gradient estimates for discrete latent variable models. In *NIPS*, 2017. 3

[34] Ross Wightman. Eficientnet for pytorch. `https://github.com/rwightman/gen-efficientnet-pytorch`, Nov 2019. 8

[35] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992. 2

[36] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 1, 2, 4, 7, 8

[37] Lingxi Xie and Alan Yuille. Genetic CNN. In *ICCV*, 2017. 1

[38] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*, 2019. 5, 11

[39] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: Stochastic Neural Architecture Search. In *ICLR*, 2019. 1, 2, 3, 4, 5, 6, 11, 13, 14

[40] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 285–300, 2018. 1, 2

[41] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *CVPR*, 2018. 6, 13

[42] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017. 1, 2, 4, 7

[43] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018. 1, 2, 4, 5, 6, 7, 11, 13, 14

Appendix

A. Architecture Search Settings

In this section, the implementation details for the search phase are provided.

A.1. Search Space

We use the following 7 operations in our search on CIFAR-10 and CIFAR-100:

1. `skip_connect`: identity connection
2. `sep_conv_3x3`: depthwise-separable 3x3 convolution
3. `max_pool_3x3`: max pooling with 3x3 kernel
4. `dil_conv_3x3`: dilated depthwise-separable 3x3 convolution
5. `sep_conv_5x5`: depthwise-separable 5x5 convolution
6. `avg_pool_3x3`: average pooling with 3x3 kernel
7. `sep_conv_7x7`: depthwise-separable 7x7 convolution

In the case of ImageNet, in order to make the search tractable, we only use the first five operations. All operations use a stride of 1 when part of the Normal Cell, and a stride of 2 when part of the Reduce Cell. Appropriate padding is added to the input features to preserve the spatial dimensions. Each convolution consists of a (`ReLU-Conv-BN`) block, and the depthwise separable convolutions are always applied twice, consistent with prior work [18, 26, 39, 43].

A.2. CIFAR-10 and CIFAR-100

The CIFAR-10 and CIFAR-100 datasets consist of 50,000 training images and 10,000 test images. During search, we use 45,000 images from the original training set as our training set and the remaining as the validation set. The final evaluation phase uses the original split. During architecture search, a network is constructed by stacking 8 cells with 4 hidden nodes. Similar to DARTS [18], the cells are stacked in the blocks of 2-2-2 Normal cells with Reduction cells in between. The networks are trained using 4 Tesla V100 GPUs with a batch size of 124, for 100 epochs. For the first 15 epochs, only the network parameters ($\boldsymbol{w}$) are trained, while the architecture parameters ($\boldsymbol{\phi}$) are frozen. This pretraining phase prevents the search from ignoring the operations that are typically slower to train. The architecture parameters are trained using the Adam optimizer with cosine learning rate schedule starting from 2×10^{-3} annealed down to 3×10^{-4}. The network parameters are also trained using Adam with cosine learning rate schedule starting from 6×10^{-4} annealed down to 1×10^{-4}. We use $\lambda = 0.5$, and a Gumbel-Softmax temperature of 0.4.

One issue with the factorized structure is that the architecture search may choose the same input and operation pair for both incoming edges of a node due to the symmetric expression in $\boldsymbol{i}_n \otimes \boldsymbol{o}_n + \boldsymbol{i}'_n \otimes \boldsymbol{o}'_n$. To prevent this, we add an architecture penalty term to our objective function using $\mathcal{L}_{arch}(\boldsymbol{z}) = \mathbb{E}\left[\lambda_{arch} \sum_{n=1}^{N} \text{tr}([\boldsymbol{i}_n \otimes \boldsymbol{o}_n][\boldsymbol{i}'_n \otimes \boldsymbol{o}'_n]^T)\right]$ where λ_{arch} is a trade-off parameter ($\lambda_{arch} = 0.2$). The term inside the summation is one if the same input/op pairs are selected by $(\boldsymbol{i}_n, \boldsymbol{o}_n)$ and $(\boldsymbol{i}'_n, \boldsymbol{o}'_n)$.

A.3. ImageNet

We search using a 14-layer network with 16 initial channels, over 8 V100 GPUs, needing around 2 days. We use a learning rate of 3×10^{-4} with Adam to learn the network parameters of the mixed-op network. We train architecture parameters with a learning rate of 1×10^{-3} using Adam. We parallelize training over 8 GPUs without scaling the learning rate. For the first 5 epochs, we only train the network parameters ($\boldsymbol{w}$), and in the remaining 15 epochs, we update both $\boldsymbol{w}$ and $\boldsymbol{\phi}$. We use $\lambda = 0.5$ and $\lambda_{arch} = 0.2$, the same as CIFAR-10, and a Gumbel-Softmax temperature of 0.4. We use a weight decay of 3×10^{-4} on the weight parameters, and 1×10^{-6} on the architecture parameters. 90% of the ImageNet train set is used to train the weight parameters, while the rest is used as the validation set for training the architecture parameters.

B. Architecture Evaluation Settings

In this section, the implementation details for the evaluation phase are provided.

B.1. CIFAR-10 and CIFAR-100

The final network is constructed by stacking a total of 20 cells. The networks are trained on a V100 GPU with a batch size of 128 for 600 epochs. SGD with momentum 0.9 is used. The cosine learning rate schedule is used starting from 5×10^{-2} annealed down to zero. Similar to DARTS, the path dropout of the probability 0.2 on CIFAR-10 and 0.3 on CIFAR-100, and cutout of 16 pixels are used.

B.2. ImageNet

For data augmentation, we use the same settings as DARTS [18]. We randomly crop training images to a size of 224×224 px along with a random horizontal flip, and jitter the color. During evaluation, we use a single center crop of size 224×224 px after resizing the image to 256×256 px.

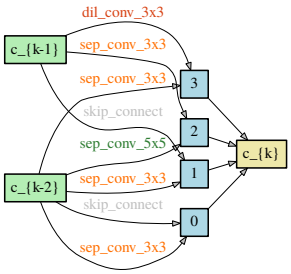
For the final evaluation, we train a 14 layer network for 250 epochs with an initial channel count such that the multiply-adds of the network is <600M, as per the mobile setting proposed by [12]. We train our networks using SGD with momentum of 0.9, base learning rate of 0.1, weight decay of 3×10^{-5}, with a batch size of 128 per GPU. We train our model for 250 epochs in line with prior work [18, 38, 39], annealing the learning rate to 0 throughout the training using a cosine learning rate decay. We scale training to 8 V100

GPUs using the linear scaling rule proposed in [7], with a learning rate warmup for the first 5 epochs.

C. Best Cell Structures

Figure 7: The best performing cell discovered on ImageNet.

(a) Normal Cell



(b) Reduce Cell

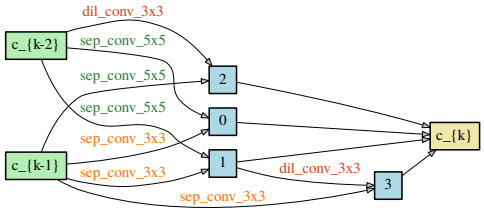
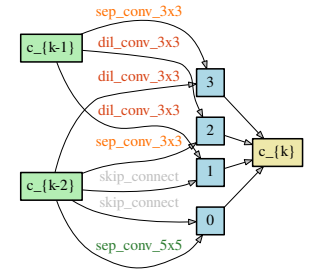


Figure 8: The best performing cell discovered on CIFAR-10.

(a) Normal Cell



(b) Reduce Cell

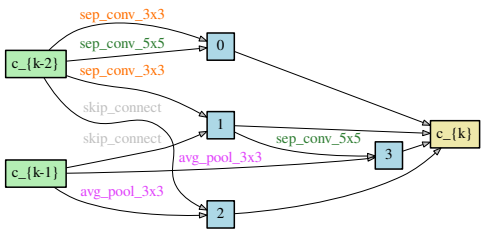
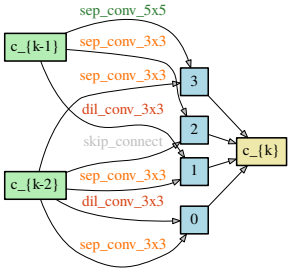


Figure 9: The best performing cell found on CIFAR-100.

(a) Normal Cell



(b) Reduce Cell

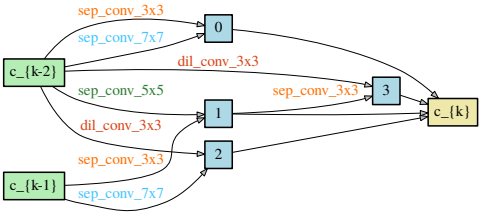
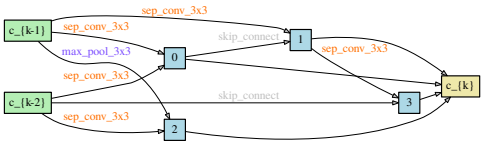
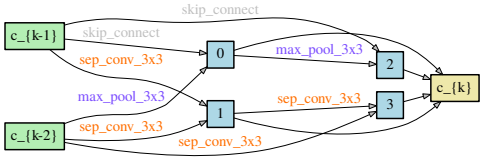


Figure 10: The best performing randomly proposed cell on ImageNet.

(a) Normal Cell



(b) Reduce Cell



D. Comparison with the Previous Work in DARTS Space

In this section, we compare the best cells discovered by UNAS against previously published results on CIFAR-10, CIFAR-100 and ImageNet.

CIFAR-10: In Table 4, the best cell discovered by UNAS is compared against the previous work that uses similar search space. For DARTS and P-DARTS, we list the original results reported by the authors, as well as, the best cell we discovered by running the original implementation four times. The best cell discovered by UNAS outperforms DARTS and SANS. In comparison to P-DARTS, UNAS obtains better than the best cell that we discovered by running the original P-DARTS code four times with different seeds. However, UNAS achieves a comparable result to P-DARTS' originally reported result on CIFAR-10. Nevertheless, as we show in Table 2, UNAS outperforms DARTS, P-DARTS, and SNAS in terms of the average performance. As discussed by Li and Talwalkar [15], the average performance is a better representative metric to evaluate the performance of NAS methods, as it is more robust against rare architecture instances that perform well, but, are less likely to be discovered by the method. Such architectures require many search/evaluation runs, making NAS models expensive for practical applications, and more challenging for reproducing the results.

When we ran the original P-DARTS source code with four different initialization seeds[7], we could not find an architecture with accuracy similar to the reported number. We believe this is because i) P-DARTS reports the lowest error observed during the evaluation phase while we report the error at the end of evaluation following DARTS. Taking the minimum of test error values, across small fluctuations towards the end of training, can reduce the error rate by 0.1%, ii) P-DARTS does not report the number of searches performed to obtain the best result. We hypothesize that the reported result is the best architecture obtained from many searches. However, we do not intend to discount the contributions made by P-DARTS. When we evaluate the original discovered cell by P-DARTS on CIFAR-10, we can reproduce the same results in the evaluation phase. Nevertheless, the contributions of UNAS are orthogonal to P-DARTS thesis as discussed in Sec. 1.1. UNAS proposes new gradient estimators that work with differentiable and non-differentiable objective functions and it also introduces a new objective function based on the generalization gap.

CIFAR-100: In Table 5, our best cell discovered using UNAS is compared against previous work. We can see that UNAS outperforms DARTS, SANS, and P-DARTS on this dataset. Similar to CIFAR-10, when we ran P-DARTS code four times, we could not discover a cell as performant as the cell discovered originally on CIFAR-100.

[7]We exactly followed the hyperparameters and commands using the search/eval code provided by the authors. We only set the initialization seed to a number in $\{0, 1, 2, 3\}$.

Table 4: Results on CIFAR-10.

Architecture	Test Error (%)	Params (M)	Search Cost (GPU Days)	Search Method
NASNet-A [43]	2.65	3.3	2000	RL
BlockQNN [41]	3.54	39.8	96	RL
AmoebaNet-A [26]	3.12	3.1	3150	evolution
AmoebaNet-B [26]	2.55	2.8	3150	evolution
H. Evolution [17]	3.75	15.7	300	evolution
PNAS [16]	3.41	3.2	225	SMBO
ENAS [24]	2.89	4.6	0.45	RL
Random [18]	3.29	3.2	4	random
DARTS-1^{st} [18]	3.00	3.3	1.5	grad-based
DARTS-2^{nd} [18]	2.76	3.3	4	grad-based
SNAS [39]	2.85	2.8	1.5	grad-based
P-DARTS [39]	2.50	3.4	0.3	grad-based
Best cell discovered after running the original code 4 times				
DARTS-2^{nd} [18]	2.80	3.6	4	grad-based
P-DARTS [39]	2.75	3.5	0.3	grad-based
UNAS	2.53	3.3	4.3	grad RL

Table 5: Results on CIFAR-100.

Architecture	Test Error (%)	Params (M)	Search Cost (GPU Days)	Search Method
BlockQNN [41]	18.06	39.8	96	RL
P-DARTS [39]	15.92	3.6	0.3	grad-based
Best cell discovered after running the original code 4 times				
DARTS-2^{nd} [18]	20.49	1.8	4	grad-based
P-DARTS [39]	17.36	3.7	0.3	grad-based
UNAS	15.79	4.1	4.0	grad RL

ImageNet: Here, we compare UNAS on the ImageNet dataset against previous works. We also provide a surprisingly strong baseline using randomly generated architectures. Table 6 summarizes the results.

Random Baseline: We provide a strong random baseline, indicated by "Random Cell" in Table 6, that outperforms most prior NAS methods. Random cells are generated by drawing uniform random samples from factorized cell structure. We train a total of 10 networks constructed by randomly generated Normal and Reduce cells. The best network yields top-1 and top-5 errors of 25.55% and 8.06% respectively (see Fig 10 for the cell structure). To the best of our knowledge, we are the first to report performance of a randomly discovered cell on ImageNet that outperforms most previous NAS methods, although not UNAS and P-DARTS.

Direct Search on ImageNet: Searching on ImageNet gives us the cell in Fig. 7. Our cell searched on Ima-

Table 6: Best results on ImageNet in the mobile setting (#Multi.-Adds<600M) [12].

Architecture	Val Error (%)		Params	×+	Search Cost	Search Method
	top-1	top-5	(M)	(M)	(GPU Days)	
MobileNetV2 [28]	25.3	–	6.9	585	–	manual
ShuffleNetV2 2× [19]	25.1	7.8	7.4	591	–	manual
NASNet-A [43]	26.0	8.4	5.3	564	2000	RL
AmoebaNet-B [26]	26.0	8.5	5.3	555	3150	evolution
AmoebaNet-C [26]	24.3	7.6	6.4	570	3150	evolution
PNAS [16]	25.8	8.1	5.1	588	∼255	SMBO
DARTS [18]	26.7	8.7	4.7	574	4	grad-based
SNAS [39]	27.3	9.2	4.3	522	1.5	grad-based
P-DARTS [5]	24.4	7.4	4.9	557	0.3	grad-based
Best cell discovered after running the original code 4 times						
DARTS [18]	25.2	7.7	5.12	595	4	grad-based
P-DARTS [5]	24.5	7.3	5.2	599	0.3	grad-based
Random Cell	25.55	8.06	5.37	598	∼250	random
UNAS	24.46	7.44	5.07	563	16	grad-based RL

geNet obtains a performance, comparable to P-DARTS and AmoebaNet-C [26], giving a top-1 and top-5 error of 24.46% and 7.44% resp. at a fraction of the cost (0.5%) required by the best AmoebaNet-C [26].

E. UNAS with ProxylessNAS Search Space

In this section, we list the implementation details used for the latency based experiments presented in Sec. 5.

E.1. Search Space

We follow ProxylessNAS [4] to construct the search space which is based on MobileNetV2 [28]. During search we seek operations assigned to each layer of a 21-layer network. The operations in each layer are constructed using mobile inverted residual blocks [28] by varying the kernel size in $\{3, 5, 7\}$ and the expansion ratio in $\{3, 6\}$ yielding 6 choices with the addition of a skip connection (i.e., an identity operation) which enables removing layers. For the channel sizes, we followed the ProxylessNAS-GPU architecture. For the first 20 epochs, only the network parameters ($\boldsymbol{w}$) are trained, while the architecture parameters ($\boldsymbol{\phi}$) are frozen. The architecture parameters are trained in 15 epochs using the Adam optimizer with cosine learning rate schedule starting from $1{\times}10^{-3}$ annealed down to $3{\times}10^{-4}$. The network parameters are also trained using Adam with cosine learning rate schedule starting from $3{\times}10^{-4}$ annealed down to $1{\times}10^{-4}$. Batch of 192 images on 8 V-100 GPUs are used for training. For the latency-based search, we use the following objective function:

$$\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[\mathcal{L}_{\text{gen}}(\boldsymbol{z}, \boldsymbol{w})] + \lambda_{lat}\mathbb{E}_{p_{\boldsymbol{\phi}}(\boldsymbol{z})}[f(\mathcal{L}_{lat}(\boldsymbol{z}) - t_{\text{target}})]$$

where t_{target} represents the target latency, $f(u) = max(0, u)$ penalizes the architectures that has latency higher than the target latency.

We linearly anneal λ_{lat} from zero to 0.1 to focus the architecture search on the classification loss initially. However, we empirically observed that the latency loss has a low gradient variance that provides a very strong training signal for selecting low-latency operations such as skip connection. To avoid this, Inspired by P-DARTS [5], we apply dropout to the skip connection during search. We observe that a small amount of dropout with probability 0.1 prevents the search from over-selecting the skip operation.

E.2. Evaluation

After search, the operations in each layer with the highest probability values are chosen for the final network. The training in the evaluation phase is based on the ProxylessNAS evaluation. Batches of 512 images on 8 V-100 GPUs are used for training in 300 epochs. We train our networks using SGD with momentum of 0.9, base learning rate of 0.2, linear learning-rate warmup in 5 epoch, and weight decay of $5{\times}10^{-5}$. The learning rate is annealed to 0 throughout the training using a cosine learning rate decay.