

PROBABILISTIC NEURAL ARCHITECTURE SEARCH

Francesco Paolo Casale*

Microsoft Research
Cambridge, MA, USA

frcasale@microsoft.com

Jonathan Gordon*

University of Cambridge
Cambridge, UK

jg801@cam.ac.uk

Nicoló Fusi

Microsoft Research
Cambridge, MA, USA

fusi@microsoft.com

ABSTRACT

In neural architecture search (NAS), the space of neural network architectures is automatically explored to maximize predictive accuracy for a given task. Despite the success of recent approaches, most existing methods cannot be directly applied to large scale problems because of their prohibitive computational complexity or high memory usage. In this work, we propose a Probabilistic approach to neural ARchitecture SEarCh (PARSEC) that drastically reduces memory requirements while maintaining state-of-the-art computational complexity, making it possible to directly search over more complex architectures and larger datasets. Our approach only requires as much memory as is needed to train a single architecture from our search space. This is due to a memory-efficient sampling procedure wherein we learn a probability distribution over high-performing neural network architectures. Importantly, this framework enables us to transfer the distribution of architectures learnt on smaller problems to larger ones, further reducing the computational cost. We showcase the advantages of our approach in applications to CIFAR-10 and ImageNet, where our approach outperforms methods with double its computational cost and matches the performance of methods with costs that are three orders of magnitude larger.

1 INTRODUCTION

Identifying a good neural network architecture for a given problem can be a difficult and time-consuming process, usually consisting in trying different combinations of layers and connection patterns until a "good" validation accuracy is finally reached. Due to the size of the optimization space, manually performing this complex optimization task can be daunting: one could choose different sizes or types of convolution, skip connections, layer sizes, number of layers, number of filters and so on.

In recent years, there has been a surge of interest in automatically identifying neural network architectures, effectively replacing the expense of human time with the expense of computational time. These approaches have often reached or exceeded the level of accuracy obtained by architectures that were tuned manually (Zoph et al., 2017; Cai et al., 2018b; Liu et al., 2017a; Zhong et al., 2018; Zoph and Le, 2016). Most of the early work in this area was focused on defining the search space (*e.g.* which set of operation and connection patterns to consider) and the search method (*e.g.* reinforcement learning, evolutionary approaches), and required to fully train and evaluate each architecture considered in the search. Given that a typical search using these methods involved training hundreds to thousands of candidate architectures, the computational cost of these searches was in the order of hundreds to thousands of GPU-days.

A more recent line of work has focused on sharing weights across multiple architectures (Pham et al., 2018; Cai et al., 2018a). This is typically achieved by defining an over-parameterized parent network containing all candidate paths in the architecture search space (Liu et al., 2018; Xie et al., 2019). In this setting, NAS reduces to estimate the importance of the candidate paths in the parent network and high-performing architectures can then be obtained

*These authors contributed equally to this work.

by pruning unimportant paths in it. While on one hand this approach drastically reduced the computational cost of NAS to a few GPU days, these methods have much higher memory requirements with respect to training an architecture of the search space, because of the much higher number of intermediate feature maps in the parent network.

The trade-off between high computational cost and high memory cost has resulted in the usage of different types of surrogates (or proxies) during architecture search. The main two types of surrogates are *architecture surrogates* and *dataset surrogates*. Architecture surrogates are small versions of the final network that are cheaper to store in memory and faster to train. For example, in image classification, this can be achieved by searching over the architecture of a recurrent convolutional unit, the *cell* (Liu et al., 2017a,b; Real et al., 2018; Cai et al., 2018b; Liu et al., 2017c; Tan et al., 2018; Luo et al., 2018), rather than directly on the global architecture. Global networks with different complexities can then be obtained by stacking cells while varying the number of stacks, the number of filters, *etc.* One advantage of this framework is that it allows to reuse cell architectures that are found for small global networks for larger ones. The second type of surrogates, dataset surrogates, are datasets that can be used as a proxy to perform a quicker search. For example, one could perform architecture search on CIFAR-10 and "transfer" the resulting architecture (with some modifications, e.g. changing the number of filters) to ImageNet (Liu et al., 2017a, 2018; Xie et al., 2019). While good performance on a surrogate task does not guarantee good performance on the final task (*i.e.* fully-sized network on the target dataset), the vast majority of architecture search methods use at least one, if not both, of these surrogates.

In this work, we address these problems by casting neural network architecture search in a probabilistic modelling framework and propose a sampling-based optimization method to learn a probability distribution over high-performing architectures for a specified supervised task. Our probabilistic framework, called PARSEC (Probabilistic neural ARchitecture SEArCh), has multiple advantages over existing NAS methods:

- our search procedure is memory-efficient, as only the feature maps associated with the sampled architectures are loaded on the GPU. This allows us to directly search over fully-sized architectures and larger datasets;
- because of PARSEC's search space and probabilistic foundations, our approach can transfer probability distributions over architectures learnt on small surrogates to larger networks and datasets, enabling us to further reduce the computational cost;
- our search procedure is computationally efficient and can be run in less than a day on a single GPU on CIFAR-10;
- in experiments on CIFAR-10 and ImageNet, we show that PARSEC outperforms other methods that consider the same architecture search space, while drastically reducing the search time. When evaluating architectures on ImageNet, the best architecture found by PARSEC on CIFAR-10 in less than one GPU-day outperforms architectures identified by other methods with comparable computational cost, while matching the accuracy of architectures found by methods with computational costs that are two to three orders of magnitude larger.

2 RELATED WORK

Neural architecture search (NAS) is a line of research that has received significant attention over the last few years. The goal of NAS methods is to search for architectures that have good performances on a specific task. This is achieved by performing a heuristic search in a predefined architecture search space.

For image classification tasks, most works define a search space in terms of *cells*, computational graphs of neural primitives (*e.g.*, convolution or pooling operations) that can then be stacked to compose global networks (Zoph and Le, 2016; Liu et al., 2017a). See Figure 1a for an illustration.

Multiple search methods have been considered to traverse the space of architectures, including reinforcement learning (RL; Zoph and Le (2016); Baker et al. (2016)), genetic algorithms

(Liu et al., 2017b; Xie and Yuille, 2017; Zoph et al., 2017), and progressive search methods (Liu et al., 2017a; Negrinho and Gordon, 2017). A significant drawback of these methods is that they require fully training and evaluating hundreds (or thousands) of intermediate models during search, resulting in searches that require thousands of GPU compute hours.

More recently, researchers have observed that the key source of computational overhead is the need to fully train intermediate models during search (Pham et al., 2018; Cai et al., 2018a). This observation lead to the idea of *weight sharing* during training (Pham et al., 2018). Here intermediate models all share the same weights, which are updated during search iterations. In these works, the entire search space is viewed as a single, large parent network containing all possible paths, while instances of architectures are child networks from the parent. Within this framework, Pham et al. (2018) introduce a controller on the large graph that activates subsets of the graph at each iteration. The controller is trained with policy gradient to discover subgraphs with good validation accuracy. Liu et al. (2018) propose instead a continuous relaxation to the exact problem, where hard selections on the paths to select in the parent network are replaced by continuous parameters that weight the importance of each path. The weights are tuned with gradient descent on held out validation cross-entropy, and at the end of search a heuristic is employed to extract a single graph based on the finalized weights. Xie et al. (2019) represent the search space of with a set of one-hot random variables in a fully factorized joint distribution. All these papers produce compelling results in a 1000x reduction in compute time with respect to previous method, relying either on architecture surrogates or on dataset surrogates (or both).

Our work is most closely related to DARTS (Liu et al., 2018) and SNAS (Xie et al., 2019). We use the same search space and computational graph as DARTS, but rather than employing a continuous relaxation, we directly tackle the optimization problem by viewing search evaluations as samples from an underlying distribution over architectures, a procedure that is similar to SNAS (Xie et al., 2019). In contrast to SNAS, we consider a different factorization of the probability distribution over architectures that leads to a dramatic reduction of the memory requirements: the memory requirements of PARSEC are the same as needed for training a single architecture of the search space. Also, we propose leveraging previous searches as prior information to speed up convergence on a new dataset or different global architectures.

Finally, another related method is ProxylessNAS (Cai et al., 2019), which achieves low memory requirements by applying a binary mask to the architecture topology and operations, forcing only a path through the network to be active and thus loaded in GPU memory. The parameters of the binary mask are learned using BinaryConnect (Courbariaux et al., 2015). ProxylessNAS is the first method not requiring surrogates for architecture search. However, in contrast to PARSEC, ProxylessNAS is used for searching over global architectures rather than over a recurrent unit (*i. e.*, the cell) as done here, thereby requiring a strong prior knowledge on which global network search space should be selected for specific datasets. This requirement also prevents transferring knowledge across related tasks. In contrast, our goal in this work is to learn a set of good cells that perform well across different architecture sizes and datasets. Leveraging similarity between problems, we can then efficiently fine-tune these architecture choices to perform well for specific architecture sizes and datasets.

3 METHODOLOGY

In this section, we present our probabilistic approach to neural architecture search. In Section 3.1 we describe the architecture search space considered in our work; in Section 3.2 we summarize the deterministic NAS method proposed in Liu et al. (2018), our model builds upon; in Section 3.3 we introduce our probabilistic framework; finally, in Section 3.4 we give full algorithmic details of our architecture search procedure.

3.1 ARCHITECTURE SEARCH SPACE

In this work, we consider the same architecture search space as in DARTS (Liu et al., 2018). Neural network architectures are obtained by stacking a recurrent convolutional unit,

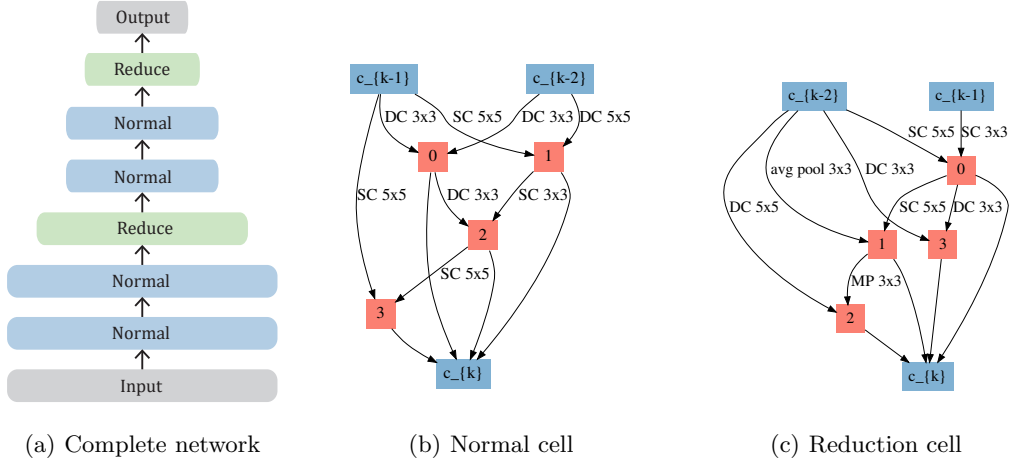


Figure 1: (a) Global network obtained stacking normal and reduction search. (b) Example of normal and (c) reduction cells.

denoted as *cell* (Zoph et al., 2017). A cell is defined as a directed acyclic graph of N ordered nodes, $\{z_1, \dots, z_N\}$, taking the output of the two previous cells as inputs and returning the concatenation of all its intermediate nodes as output. An intermediate node, z_k , is defined as the sum of two of the previous nodes, z_i and z_j with $i, j < k$, after applying primitive operations $o_{i,k}$ and $o_{j,k}$, respectively:

$$z_k = o_{i,k}(z_i) + o_{j,k}(z_j), \quad i < k, \quad j < k. \quad (1)$$

A node is thus fully specified by the two previous nodes selected as inputs and the two primitive operations that are applied to either of them. As in Liu et al. (2018), we consider the following $P = 7$ primitive operations:

- identity;
- 3×3 average pooling;
- 3×3 max pooling;
- 3×3 separable convolutions;
- 5×5 separable convolutions;
- 3×3 dilated separable convolutions;
- 5×5 dilated separable convolutions.

All convolutions are preceded by a ReLU transformation (Xu et al., 2015) and followed by batch normalization (Ioffe and Szegedy, 2015), and padding is added to preserve spatial dimensions of feature maps. A cell is fully specified when the inputs and operations of each intermediate node are defined.

For a cell with $N = 4$ nodes, this corresponds to $\prod_{n=1}^N P^2 \times (n+1)^2 \approx 10^{10}$ possible choices (see Liu et al. (2017a)).

Finally, following previous work (Zoph and Le, 2016; Liu et al., 2018), we also consider two different types of cells: *normal* cells and *reduction* cells. Operations in a *normal* cell have stride one and do not change the shape of the feature maps, whereas operations in a *reduction* cell have stride two and halve the spatial dimensions of the feature maps. When stacking cells, every third cell is a reduction cell, and the rest are normal (Figure 2a). The architectures of these two cells may be different and are learned jointly during search (Figure 2b-c shows an example of normal and reduction cell, respectively).

3.2 DETERMINISTIC APPROACH TO ARCHITECTURE SEARCH

DARTS (Liu et al., 2018) is a deterministic approach to neural architecture search that uses a continuous relaxation on the categorical choices of inputs and operations, so that the architecture can be optimized using gradient descent. Specifically, Liu et al. (2018) consider an over-parametrized parent network containing all possible paths between nodes:

$$\mathbf{z}_k = \sum_{i,p} \alpha_{ik}^p \cdot o_{i,k}^p(\mathbf{z}_i), \quad (2)$$

where indices i and p run over all possible inputs and operations for node k respectively, $o_{i,k}^p$ denotes primitive operation p on input node i to contribute to output node k , and $\alpha_{i,k}^p$ is the weight of the path. The path weights $\alpha_{i,k}^p$ of all operations between two nodes are set to sum up to 1, *i. e.*, $\sum_p \alpha_{i,k}^p = 1$. In order to enable the automatic pruning of unimportant input nodes during search, a null operation is added to the set of primitive operations. After this relaxation, architecture search reduces to an optimization problem over the continuous architecture parameters $\boldsymbol{\alpha} = \{\alpha_{i,k}^p\}$, which measure the importance of the different paths in the parent network. Specifically, the authors propose to jointly optimize the network weights \mathbf{v} of the parent (*i. e.*, the union of the weights associated to each operation $o_{i,k}^p$) and the architecture parameters $\boldsymbol{\alpha}$ using gradient descent on different splits of the data, namely a train and a search set. After the search, an architecture of the original search space is obtained by (i) selecting for each node the two strongest input nodes and (ii) selecting the strongest operation at each remaining edge. The obtained architecture is then trained from scratch (*i. e.*, weights are re-initialized) for the supervised task at hand. For further details on the optimization procedure, we refer to the original DARTS paper (Liu et al., 2018).

3.3 PROBABILISTIC NEURAL ARCHITECTURE SEARCH

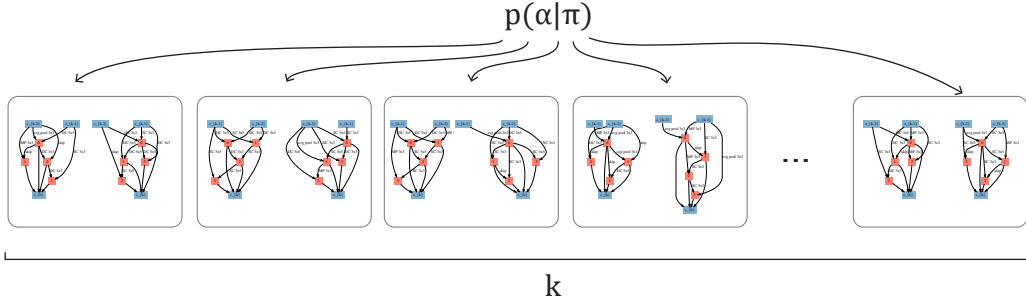


Figure 2: Pictorial representation of architecture sampling. Each sample consists of a normal and a reduction cell, which are shown side-to-side.

As opposed to the deterministic framework in DARTS, we consider a Probabilistic Approach to Neural ARchitecture SEarCh (PARSEC). We introduce a prior $p(\boldsymbol{\alpha}|\boldsymbol{\pi})$ on the choices of inputs and operations that define the cell (Figure 3a), where hyper-parameters $\boldsymbol{\pi}$ are the probabilities corresponding to the different choices. In this framework, child networks from the search space are directly obtained as samples from the specified architecture distribution, and thus neural architecture is reduced to inferring the probability distribution $p(\boldsymbol{\alpha}|\boldsymbol{\pi})$ of high-performing architectures on the task at hand. Given a supervised task and denoting with \mathbf{y} the targets and with \mathbf{X} the input features, this is achieved by optimizing the continuous prior hyper-parameters $\boldsymbol{\pi}$ through an empirical Bayes Monte Carlo procedure (Carlin and Louis, 2000). Specifically, we optimize:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) = \int p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\alpha}) p(\boldsymbol{\alpha}|\boldsymbol{\pi}) d\boldsymbol{\alpha}, \quad (3)$$

with respect to the network weights \mathbf{v} , which are shared across all child architectures (*i. e.*, samples), and the prior hyper-parameters $\boldsymbol{\pi}$, *i. e.*, the architecture parameters.

Architecture distribution. In the search space we introduced in Section 3.1, each intermediate node is specified by defining (i) the two input predecessor nodes and (ii) the two primitive operations to apply to each of the input nodes. We assume independent categorical distributions over each input/operation pair. Specifically, for normal and reduction cells we set

$$p(\boldsymbol{\alpha}^{(n)}|\boldsymbol{\pi}^{(n)}) = \prod_{n=1}^N \prod_{i=1}^2 \text{Cat}(\boldsymbol{\alpha}_{n,i}^{(n)}|\boldsymbol{\pi}_{n,i}^{(n)}), \quad (4)$$

$$p(\boldsymbol{\alpha}^{(r)}|\boldsymbol{\pi}^{(r)}) = \prod_{n=1}^N \prod_{i=1}^2 \text{Cat}(\boldsymbol{\alpha}_{n,i}^{(r)}|\boldsymbol{\pi}_{n,i}^{(r)}), \quad (5)$$

where n runs over the nodes, i runs over the inputs of each node (in our search space each node has two inputs), $\boldsymbol{\pi}_{n,i}^{(n)}$ and $\boldsymbol{\pi}_{n,i}^{(r)}$ are the vectors of probabilities for all possible incoming node/operation pairs and finally, we introduced $\boldsymbol{\alpha}^{(\cdot)} = \{\boldsymbol{\alpha}_{n,i}^{(\cdot)}\}$ and $\boldsymbol{\pi}^{(\cdot)} = \{\boldsymbol{\pi}_{n,i}^{(\cdot)}\}$. Note that each sample from this distribution is a child architecture from the architecture search space. As in our search procedure all computations are done on child networks sampled from the architecture distribution (as explained in detail in the next section), the resulting NAS algorithm has the same memory requirements of training a single architecture from the search space. This is not the case, for example, for either SNAS or DARTS.

3.4 IMPORTANCE-WEIGHTED MONTE CARLO EMPIRICAL BAYES

In this section, we develop an importance weighted EB procedure for jointly optimizing $\boldsymbol{\pi}$ and \mathbf{v} . We begin by introducing the following estimator:

$$\begin{aligned} p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) &= \int p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\alpha}) p(\boldsymbol{\alpha}|\boldsymbol{\pi}) d\boldsymbol{\alpha} \\ &\approx \frac{1}{K} \sum_k p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\alpha}_k), \end{aligned} \quad (6)$$

with $\boldsymbol{\alpha}_k \sim p(\boldsymbol{\alpha}|\boldsymbol{\pi})$. Note that the gradients can be written as

$$\begin{aligned} \nabla_{\mathbf{v}, \boldsymbol{\pi}} \log p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) &= \frac{1}{p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi})} \int \nabla_{\mathbf{v}, \boldsymbol{\pi}} p(\mathbf{y}, \boldsymbol{\alpha}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) d\boldsymbol{\alpha} \\ &= \frac{1}{p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi})} \int \nabla_{\mathbf{v}, \boldsymbol{\pi}} \log p(\mathbf{y}, \boldsymbol{\alpha}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) p(\mathbf{y}, \boldsymbol{\alpha}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) d\boldsymbol{\alpha} \end{aligned} \quad (7)$$

Finally, plugging the estimator in Eq. (6) into Section 3.4 and taking an importance sampling estimator to the expectation, we obtain tractable gradient estimators for $\boldsymbol{\theta}$ and $\boldsymbol{\pi}$:

$$\begin{aligned} \nabla_{\mathbf{v}} \log p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) &\approx \sum_{k=1}^K \tilde{\omega}_k \nabla_{\mathbf{v}} \log p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\alpha}_k) \triangleq \tilde{\nabla}_{\mathbf{v}}, \\ \nabla_{\boldsymbol{\pi}} \log p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\pi}) &\approx \sum_{k=1}^K \tilde{\omega}_k \nabla_{\boldsymbol{\pi}} \log p(\boldsymbol{\alpha}_k|\boldsymbol{\pi}) \triangleq \tilde{\nabla}_{\boldsymbol{\pi}}, \end{aligned} \quad (8)$$

where $\boldsymbol{\alpha}_k \sim p(\boldsymbol{\alpha}|\boldsymbol{\pi})$ and $\tilde{\omega}_k = \frac{p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\alpha}_k)}{\sum_j p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\alpha}_j)}$.

Note that the gradient estimators in Section 3.4 are weighted average of the gradients computed on K child networks sampled from $p(\boldsymbol{\alpha}|\boldsymbol{\pi})$, where the importance weights $\tilde{\omega}_k$ are proportional to the model likelihood of child network k , $p(\mathbf{y}|\mathbf{X}, \mathbf{v}, \boldsymbol{\alpha}_k)$. The update rules and procedure derived here are similar to the ones used in the reweighted wake-sleep procedure (Bornschein and Bengio, 2015; Le et al., 2018), where rather than updating variational parameters of an approximate posterior distribution we are updating the parameters of the prior. Following the approach taken in DARTS (Liu et al., 2018), in order to avoid over-fitting during the joint optimization of the architecture parameters and the shared weights, we consider different splits of the data (specifically, a *search* and a *train* sets) for the estimation of the importance weights and the estimation of the network weight updates. A

Algorithm 1 Importance weighted Monte-Carlo EB algorithm used for joint training of the network and architecture parameters.

```

1: Define train and search splits of the data
2:  $\theta, \pi \leftarrow$  Initial values
3: Initialize  $\theta$  and  $\pi$ 
4: repeat
5:   Sample batch  $(\mathbf{X}^{(s)}, \mathbf{y}^{(s)})$  from search set
6:   for  $k \in \{1, \dots, K\}$  do
7:     Sample architecture  $\alpha_k$  from  $p(\alpha|\pi)$ 
8:      $\omega_k \leftarrow p(\mathbf{y}^{(s)}|\mathbf{X}^{(s)}, \mathbf{v}, \alpha)$ 
9:   end for
10:  Compute normalized weights  $\tilde{\omega}_k = \frac{\omega_k}{\sum_{j=1}^K \omega_j}$ 
11:  Sample batch  $(\mathbf{X}^{(t)}, \mathbf{y}^{(t)})$  from train set
12:  for  $k \in \{1, \dots, K\}$  do
13:     $g_{\mathbf{v},k} \leftarrow \nabla_{\mathbf{v}} \log p(\mathbf{y}^{(t)}|\mathbf{X}^{(t)}, \mathbf{v}, \alpha_k)$ 
14:     $g_{\pi,k} \leftarrow \nabla_{\pi} \log p(\alpha_k|\pi)$ 
15:  end for
16:  Update  $\mathbf{v}$  by  $\tilde{\nabla}_{\mathbf{v}} = \sum_k \tilde{\omega}_k g_{\mathbf{v},k}$ 
17:  Update  $\pi$  by  $\tilde{\nabla}_{\pi} = \sum_k \tilde{\omega}_k g_{\pi,k}$ 
18: until Number of epochs is reached or convergence is achieved
19: Return:  $(\theta, \pi)$ 

```

full description of the derived importance weighted Monte-Carlo EB discussed in this section is given in Algorithm 1.

After joint optimization of the network weights \mathbf{v} and the architecture parameters π , one can extract a specific configuration of hyper-parameters in several ways. In this work, we consider the mode of the architecture distribution and train it from scratch. An alternative strategy is to train multiple samples from the learnt distribution from scratch and consider model ensembling. We plan to explore this direction in future work.

4 EXPERIMENTS AND RESULTS

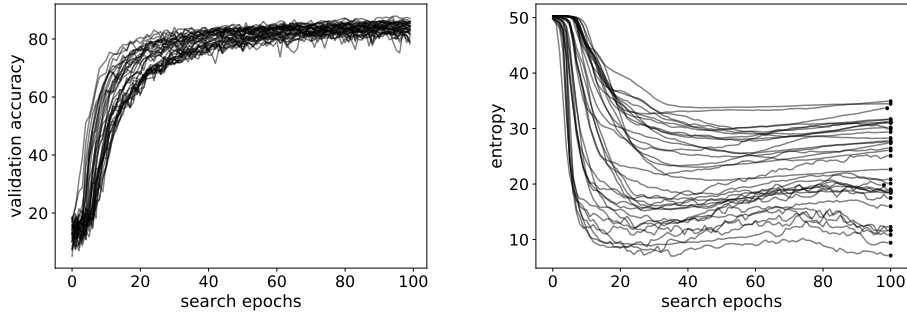
We validate our method by performing architecture search using CIFAR-10 and evaluating performance both on CIFAR-10 and ImageNet. In Section 4.1, we evaluate the performance of our approach using the same procedure considered in (Liu et al., 2018): (i) search a good cell architecture using a small global networks, (ii) evaluate the best cell architecture on a fully-sized global network. In Section 4.2, we use our probabilistic framework to tune the distribution over architectures identified using the small global network using the fully-sized global network and show that this further improves performance. Finally, in Section 4.3 we evaluate the networks searched on CIFAR-10 using ImageNet.

Overall, our results show that:

1. when searching over small network surrogates, our method matches or exceeds (depending on the dataset) the performance of methods with similar search space and comparable computational cost.
2. fine-tuning without network surrogates on CIFAR-10 (*i. e.*, directly operating over fully-sized networks) further improves the results in both datasets.

4.1 NEURAL ARCHITECTURE SEARCH USING CIFAR-10

We start by evaluating the performance of our approach in the same exact setting as DARTS (Liu et al., 2018). Specifically, we first search good cell architectures using a global small network, then we evaluate the best cell when using a fully-sized global network.



(a) Validation accuracy of the mode of the architecture distribution vs search epochs (b) Entropy of the architecture distribution vs search epochs

Figure 3: Validation accuracy of the mode of the architecture distribution and its entropy during independent architecture searches on CIFAR-10.

To perform the search we hold out 50% of the CIFAR-10 training set and use it as a search set, *i. e.*, we use it to compute the architecture importance weights (Algorithm 1). We learn the architecture parameters using Adam (Kingma and Ba, 2014) with learning rate 0.02 and $\beta = (0.5, 0.999)$ for 100 epochs with $k = 16$. All other parameter and settings are identical to the ones used in (Liu et al., 2018). Note that in PARSEC, k acts as a fidelity parameter: higher k will result in more exploration of the architecture space at the expense of higher computational costs.

Figure 3a shows the validation accuracy of the mode of the architecture distributions during search as a function of the number of epochs across multiple random seeds, learning rates and numbers of Monte Carlo samples K (we considered $K = 8, 16$, see also Section 3.4). Figure 3b shows the entropy of $p(\alpha|\pi)$ as a function of the number of epochs in the same settings. Overall, these figures show that the entropy of the architecture probability distribution steadily decreases over time (Figure 3b) and that the mode of the architecture distributions (*i. e.*, the network we ultimately select) improves during search.

As reported in (Liu et al., 2018), we also noticed variance in search results arising from subtle differences in initialization. For this reason, we run searches with 2 random seeds and select the architecture that reached highest validation accuracy. Search takes approximately 15 hours on a single V100 GPU.

To evaluate the accuracy of the resulting network, we train a large network of 20 cells. All the hyperparameters follow Liu et al. (2018). To summarize briefly: we train a network of 20 cells for 600 epochs and consider a batch size of 96. We use cutout in the data augmentation, consider a path dropout probability of 0.4 and auxiliary towers with weight 0.4. Since networks evaluated on CIFAR-10 exhibit high variance across random seeds (Liu et al., 2017b), we report the mean and standard deviation across 5 random weight initializations.

As shown in table 1, PARSEC achieves results that are comparable to the state-of-the-art (Liu et al., 2018; Xie et al., 2019) while requiring half of the computational cost.

4.2 ARCHITECTURE FINE-TUNING USING A FULLY-SIZED NETWORK

Next, we transfer the distribution over architectures identified during the search on small networks and we tune it directly on fully-sized networks, removing the need for architecture surrogates entirely. This experiments showcases (i) the memory efficiency of our method, (ii) its computational efficiency and (iii) its ability to transfer knowledge across searches on different architectures. For this fine-tuning task, we use an Adam learning rate of 0.01 and train for 10 epochs. The distribution $p(\alpha|\pi)$ is initialized using the distribution tuned during the search described in the previous section. Importantly, we fine-tune on the exact network configuration that will be used to train the final network. This includes the use of auxiliary

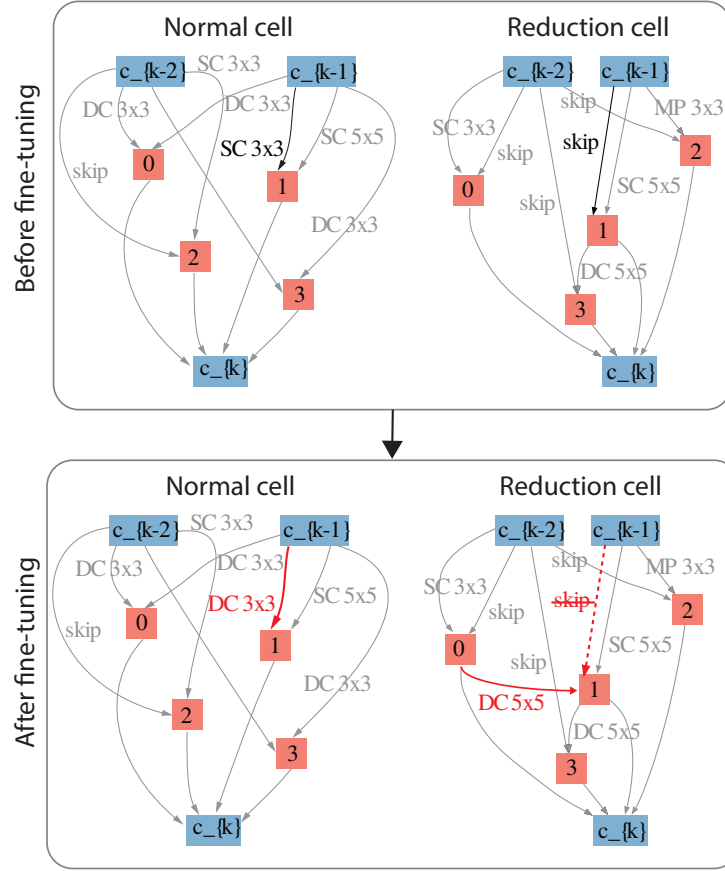


Figure 4: Changes in the architecture in the fine-tuning experiment in CIFAR-10. (top) Best architecture found on CIFAR-10 with the small network search. (bottom) Best architecture after fine-tuning using the fully-sized network on CIFAR-10. Dashed lines represent connections that were removed.

Architecture	Test Error (%)	Params (M)	Search Cost (GPU days)	Search Method
DenseNet-BC (Huang et al., 2017)	3.46	25.6	-	manual
NASNet-A + cutout (Zoph et al., 2017)	2.65	3.3	1800	RL
NASNet-A + cutout (Zoph et al., 2017) [†]	2.83	3.1	3150	RL
AmoebaNet-A + cutout (Real et al., 2018)	3.34 ± 0.06	3.2	3150	evolution
AmoebaNet-A + cutout (Real et al., 2018) [†]	3.12	3.1	3150	evolution
AmoebaNet-B + cutout (Real et al., 2018)	2.55 ± 0.05	2.8	3150	evolution
Hierarchical Evo (Liu et al., 2017b)	3.75 ± 0.12	15.7	300	evolution
PNAS (Liu et al., 2017a)	3.41 ± 0.09	3.2	225	SMBO
ENAS + cutout (Pham et al., 2018)	2.89	4.6	0.5	RL
DARTS (first order) + cutout (Liu et al., 2018)	2.94	2.9	1.5	gradient-based
DARTS (second order) + cutout (Liu et al., 2018)	2.83 ± 0.06	3.4	4	gradient-based
SNAS + mild constraint + cutout (Xie et al., 2019)	2.98	2.9	1.5	gradient-based
SNAS + moderate constraint + cutout (Xie et al., 2019)	2.85 ± 0.02	2.8	1.5	gradient-based
SNAS + aggressive constraint + cutout (Xie et al., 2019)	3.10 ± 0.04	2.3	1.5	gradient-based
Random + cutout	3.49	3.1	-	-
PARSEC (search on small network) + cutout	2.86 ± 0.06	3.6	0.6	gradient-based
PARSEC (fine-tuning on large network) + cutout	2.81 ± 0.03	3.7	1	gradient-based

Table 1: Comparison of different search methods and state-of-the-art architectures on CIFAR-10

Architecture	Test Error (%)		Params (M)	Search Cost	Search Method
	top-1	top-5			
Inception-v1 (Szegedy et al., 2015)	30.2	10.1	6.6	-	manual
MobileNet (Howard et al., 2017)	29.4	10.5	4.2	-	manual
ShuffleNet 2× (v1) (Zhang et al.)	29.1	10.2	~5	-	manual
ShuffleNet 2× (v2) (Zhang et al.)	26.3	-	~5	-	manual
NASNet-A (Zoph et al., 2017)	26.0	8.4	5.3	1800	RL
NASNet-B (Zoph et al., 2017)	27.2	8.7	5.3	1800	RL
NASNet-C (Zoph et al., 2017)	27.5	9.0	4.9	1800	RL
AmoebaNet-A (Real et al., 2018)	25.5	8.0	5.1	3150	evolution
AmoebaNet-B (Real et al., 2018)	26.0	8.5	5.3	3150	evolution
AmoebaNet-C (Real et al., 2018)	24.3	7.6	6.4	3150	evolution
PNAS (Liu et al., 2017a)	25.8	8.1	5.1	225	SMBO
DARTS (Liu et al., 2018)	26.9	9.0	4.9	4	gradient-based
SNAS + mild constraint (Xie et al., 2019)	27.3	9.2	4.3	1.5	gradient-based
PARSEC (search small network on CIFAR10)	26.3	8.4	5.5	0.6	gradient-based
PARSEC (fine-tuning large network on CIFAR10)	26.0	8.4	5.6	1	gradient-based

Table 2: Comparison with state-of-the-art image classifiers on ImageNet.

towers, cutout and larger batch sizes (96 samples instead of 64), which are all settings not considered during search on small networks.

This additional tuning step takes 8 hours on a single V100 GPU, bringing the total time taken by both search and fine-tuning to 23 hours. Once more, we evaluate the accuracy of the resulting network by training from scratch, as done in Liu et al. (2018). As shown in table 1, in exchange for a modest increase in search cost, fine-tuning further improves the accuracy of the networks identified during first search. While the results have relatively high variance, it’s worth noting that on average, PARSEC is more accurate than every method with comparable search cost, and it is only outperformed by methods (Zoph et al., 2017; Real et al., 2018) that have search budgets of thousands of GPU-days.

4.3 ARCHITECTURE EVALUATION ON IMAGENET

Next, we transfer the architectures searched and fine-tuned on CIFAR-10 to ImageNet and train a large network to evaluate their performance. Once more, we follow the same hyperparameters used in Liu et al. (2018) and train networks of 14 cells for 600 epochs.

As shown in Table 2, both architectures (searched on small network and fine-tuned on large network) identified by PARSEC outperform those identified by competing methods with $1.5\times$ to $4\times$ larger computational costs. In particular the fine-tuned architecture also approaches

the performance of the architecture identified by PNAS (Liu et al., 2017a), which has a $225\times$ larger computational cost and matches the performance of NASNet (Zoph et al., 2017), which has a $1800\times$ larger cost.

As observed in CIFAR-10, the fine-tuned architecture outperforms the architecture identified by searching over small architecture surrogates.

5 CONCLUSION

In this work, we have presented PARSEC, a probabilistic approach neural architecture search method with low memory requirements that learns a probability distribution over high-performing neural network architectures. Because of its efficient sampling-based optimization method, PARSEC only requires as much memory as is needed to train a single architecture from its search space. In our framework, the architecture distributions learnt for a specific network and dataset can be transferred and fine-tuned in larger problems. Through experiments on CIFAR-10 and ImageNet, we have shown that when searching over small network surrogates, our method matches or exceeds the performance of methods with same search space and comparable computational cost. Importantly, we have shown that fine-tuning the architecture distribution on fully-sized network further improves performance on both CIFAR-10 and ImageNet. Our best network achieves a test error of 26% on ImageNet, which is comparable to accuracies obtained with search algorithms that have computational costs that are up to three orders of magnitude larger.

REFERENCES

- B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.
- J. Bornschein and Y. Bengio. Reweighted wake-sleep. In *International Conference on Learning Representations (ICLR)*, 2015.
- H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Efficient architecture search by network transformation. AAAI, 2018a.
- H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu. Path-level network transformation for efficient architecture search. *arXiv preprint arXiv:1806.02639*, 2018b.
- H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=HylVB3AqYm>.
- B. P. Carlin and T. A. Louis. Empirical Bayes: Past, present and future. *Journal of the American Statistical Association*, 95(452):1286–1289, 2000.
- M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.
- A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.
- S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.
- D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- T. A. Le, A. R. Kosiorek, N. Siddharth, Y. W. Teh, and F. Wood. Revisiting reweighted wake-sleep. *arXiv preprint arXiv:1805.10469*, 2018.
- C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017a.
- H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017b.
- H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.
- Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2755–2763. IEEE, 2017c.
- R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In *Advances in Neural Information Processing Systems*, pages 7827–7838, 2018.
- R. Negrinho and G. Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.
- H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

- E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.
- C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018.
- L. Xie and A. Yuille. Genetic CNN. *arXiv preprint arXiv:1703.01513*, 2017.
- S. Xie, H. Zheng, C. Liu, and L. Lin. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=rylqooRqK7>.
- B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.
- X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. arxiv 2017. *arXiv preprint arXiv:1707.01083*.
- Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2423–2432, 2018.
- B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.
- B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017.