Understanding and Simplifying One-Shot Architecture Search

Gabriel Bender [1] Pieter-Jan Kindermans [1] Barret Zoph [1] Vijay Vasudevan [1] Quoc Le [1]

Abstract

There is growing interest in automating neural network architecture design. Existing architecture search methods can be computationally expensive, requiring thousands of different architectures to be trained from scratch. Recent work has explored *weight sharing* across models to amortize the cost of training. Although previous methods reduced the cost of architecture search by orders of magnitude, they remain complex, requiring hypernetworks or reinforcement learning controllers. We aim to understand weight sharing for one-shot architecture search. With careful experimental analysis, we show that it is possible to efficiently identify promising architectures from a complex search space without either hypernetworks or RL.

1. Introduction

Designing neural networks is a labor-intensive process that requires a large amount of trial and error by experts. There is growing interest in automating the search for good neural network architectures (Zoph & Le, 2016; Baker et al., 2016; Real et al., 2017). For instance, Zoph et al. (2017) show that one can find an architecture that simultaneously achieves state-of-the-art performance on the CIFAR-10, ImageNet, and COCO datasets. However these search methods are incredibly resource-hungry. Zoph et al. (2017) used 450 GPUs for four days in order to run a single experiment. They proposed an RL-based approach in which a neural network (the *controller*) enumerates a set of architectures to evaluate. Each architecture is trained from scratch on CIFAR-10 for a fixed number of epochs and then evaluated on a validation set. The weights of the controller are subsequently updated based on the validation accuracies of the trained models.

Training thousands of models is difficult or impossible for a typical machine learning practitioner. To address this weakness of architecture search, new methods have been proposed (Brock et al., 2017; Pham et al., 2018; Elsken et al., 2017; Liu et al., 2017a; Cai et al., 2017; Liu et al., 2017b). One promising direction is sharing weights between models (Brock et al., 2017; Pham et al., 2018): rather than training thousands of separate models from scratch, one can train a single large network capable of emulating any architecture in the search space.

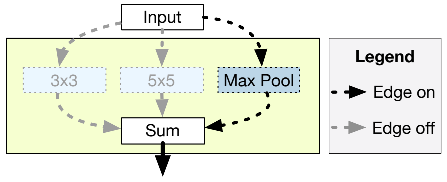

Figure 1. Example building block used in a one-shot model. The search space contains three different operations; the one-shot model adds their outputs together. Some of the operations are removed or zeroed out at evaluation time.

A simple example is shown in Figure 1, where we have the option of applying either a 3x3 convolution, a 5x5 convolution, or a max-pooling layer at a particular position in the network. Instead of training three separate models, we can train a single model containing all three operations (the *one-shot model*). We selectively zero out two of the three operations' outputs at evaluation time in order to determine which operation leads to the best prediction accuracy. In more complex examples, a search space may include choices at many different positions within a network. The size of the search space grows exponentially with the number of choices, while the size of the one-shot model grows only linearly. The same weights are used to evaluate many different architectures, reducing the resources required to run an architecture search by orders of magnitude.

Despite the improvements in efficiency, it is natural to wonder whether there are inherent limits to weight sharing across models. Why should a heterogeneous set of architectures be able to share a single set of weights? One-shot models are typically only used to *rank* architectures in the search space; the best-performing architectures are retrained from scratch after the search is completed. But even with that restriction, the idea that a single fixed set of weights can work well across a wide range of architectures is counterintuitive. SMASH (Brock et al., 2017) tries to address this concern by using a hypernetwork to generate a large fraction of the weights in each candidate architecture. Efficient Neural

[1]Google Brain, Mountain View, CA. Correspondence to: Gabriel Bender <gbender@google.com>.

Proceedings of the 35^{th} *International Conference on Machine Learning*, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

Architecture Search (ENAS) (Pham et al., 2018) addresses the same concern by alternating between training the shared model weights and training a controller that identifies a subset of architectures from the search space to focus on.

Our goal in this paper is to understand the role that weight sharing plays in efficient architecture search methods. Perhaps surprisingly, we show that neither a hypernetwork nor an RL controller is necessary to achieve good results. To do this, we train a large one-shot model containing every possible operation in the search space. We then zero out some of the operations and measure the impact on the model's prediction accuracies. When trained carefully, we show that the network automatically focuses its capacity on the operations that are most useful for generating good predictions. Zeroing out the less important operations has only a small impact on the model's predictions. In contrast, zeroing out the more important operations has an exaggerated effect on both the model's predictions and its validation set accuracy. In fact, it is possible to predict an architecture's *validation set accuracy* by looking at its behavior on *unlabeled* examples from the *training set*. This behavior is an implicit consequence of weight sharing, and requires neither a hypernetwork nor an explicit controller.

2. Related Work

The use of meta-learning to improve machine learning has a long history (Schmidhuber, 1987; Hochreiter et al., 2001; Thrun & Pratt, 2012). Beyond architecture search, meta-learning has been used to optimize other components of learning algorithms such as update rules (Andrychowicz et al., 2016; Wichrowska et al., 2017; Bello et al., 2017) and activation functions (Ramachandran et al., 2017).

Our work is most closely related to SMASH (Brock et al., 2017), which in turn is motivated by NAS (Zoph & Le, 2016). In NAS, a neural network controller is used to search for good architectures. The training of the NAS controller requires a loop: The controller proposes *child model architectures*, which are trained and evaluated. The controller is then updated by policy gradient (Williams, 1992) to sample better architectures over time. Once the controller is done training, the best architectures are selected and trained longer to improve their accuracies. The main bottleneck of NAS is the training of the child model architectures; SMASH aims to amortize this cost. In SMASH, a hypernetwork is trained a priori to generate suitable weights for every child model architecture in the search space. The same fixed hypernetwork is then used to evaluate many different child model architectures.

NAS and SMASH both treat architecture search as a black-box optimization problem, which can be optimized using off-the-shelf techniques. (Bergstra et al., 2011; Bergstra & Bengio, 2012; Bergstra et al., 2013; Snoek et al., 2012; 2015) In hyperparameter optimization, the idea of sharing parameters between models has been also explored in the context of Population Based Training (Jaderberg et al., 2017).

Genetic and neuro-evolution algorithms have also been used for designing good neural network architectures, e.g., Stanley & Miikkulainen (2002); Bayer et al. (2009); Jozefowicz et al. (2015); Miikkulainen et al. (2017); Xie & Yuille (2017). For these methods, parameter sharing between models, also known as weight inheritance, has been explored with positive effects (Real et al., 2017).

Black-box methods measure the accuracies of trained architectures on a held-out *validation set*. In contrast, MorphNets (Gordon et al., 2017) make architectural decisions directly on the *training set*, applying L1 regularization to induce sparsity. Like MorphNets, we start with an overcomplete network architecture and then prune the least useful parts. But while Gordon et al. (2017) focus on filter sizes, we focus on pruning ops and skip-connections. Experiments from SMASH and ENAS suggest our method might be extended to search over filter sizes. We could also apply MorphNets to the models found by our architecture search.

3. One-Shot Architecture Search

The proposed approach for one-shot architecture search consists of four steps: (1) Design a search space that allows us to represent a wide variety of architectures using a single one-shot model. (2) Train the one-shot model to make it predictive of the validation accuracies of the architectures. (3) Evaluate candidate architectures on the validation set using the pre-trained one shot model. (4) Re-train the most promising architectures from scratch and evaluate their performance on the test set. We describe these steps in the remainder of this section.

3.1. Search Space Design

Designing a good search space for one-shot architecture search is a challenging problem, as it requires us to balance a number of competing requirements. First: the search space should be large and expressive enough to capture a diverse set of interesting candidate architectures. Second: the validation set accuracies produced by the one-shot model must be predictive of the accuracies produced by stand-alone model training. Third: the one-shot model must be small enough to train using limited compute resources (i.e., memory and time). The best architectures in the search space must also have competitive accuracies. However, since our main goal is to understand the role of weight sharing, our search space is not yet fully optimized for quality, and we believe that further improvements are possible.

We begin with an example of a search space (shown in

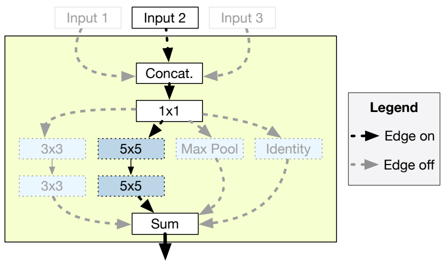

Figure 2. Example of a cell during one-shot model evaluation. Although the one-shot model contains four separate operations, we can emulate a cell containing a max-pooling op by removing the other operations from the network without retraining the weights.

Figure 3.1) that incorporates non-trivial decisions about both the network structure and the operations that are applied at different positions within the network.

At training time, the one-shot model contains three different inputs which are concatenated together. At evaluation time, however, we can simulate a network containing only Input 2 by zeroing out or removing the incoming connections from Input 1 and Input 3 from the trained network.

More generally, we have the option of enabling or disabling any combination of the incoming connections. In this way, the the size of the search space grows exponentially with the number of incoming skip-connections, while the size of the one-shot model grows only linearly. The concatenation is always followed by a 1x1 convolution; the number of output filters in the convolution remains constant no matter how many incoming skip-connections there are.

The one-shot model then applies several different operations to the output of the 1x1 convolution and adds the results together. At evaluation time, we zero out or remove some of these operations from the network. In our running example, we have four possible operations: a pair of 3x3 convolutions, a pair of 5x5 convolutions, a max pooling layer, or an identity operation. However, only the 5x5 convolutions' outputs are used when the architecture is evaluated.

This approach is applied to a much larger model as shown in Figure 3. Following Zoph et al. (2017), our network is composed of several identical cells which are stacked on top of each other. Each cell is divided into a fixed number of *choice blocks*. The inputs to a given choice block come from (1) outputs of previous cells, and (2) outputs of previous choice blocks within the same cell.

The number of choice blocks within each cell, N_{choice}, is a hyper-parameter of the search space. In our experiments, we set $N_{\text{choice}} = 4$. Each choice block can consume the outputs of the two most recent cells in the network. This means that each choice block can select from up to five possible inputs: two from previous cells and up to three from previous choice blocks within the same cell.

Each choice block can select up to two operations from a menu of seven possible options: (1) identity, (2) a pair of depthwise separable 3x3 convolutions, (3) a pair of depthwise separable 5x5 convolutions, (4) a pair of depthwise separable 7x7 convolutions, (5) a 1x7 convolution followed by a 7x1 convolution, (6) a max pooling layer, and (7) an average pooling layer.

Search Space Size. Each architecture in our search space consists of a stack of identical cells; we now estimate the number of possible cells in our search space. Each cell has four choice blocks. For $i = 0, 1, 2, 3$, the ith choice block takes at least 1 and at most $2 + i$ inputs. This means that there are $2^{2+i} - 1$ possible combinations of inputs to the ith choice block. Furthermore, each choice block applies either one or two different operations out of 7 possible options. There are therefore $\binom{7}{1} + \binom{7}{2} = 7 + 21 = 28$ possible combinations of operations that we can apply in each block. Across the entire search space, there are therefore $(2^2 - 1) \cdot (2^3 - 1) \cdot (2^4 - 1) \cdot (2^5 - 1) \cdot 28^4 \approx 6 \cdot 10^9$ possible cells.

3.2. Training the One-Shot Model

Many architecture search methods use a proxy metric to find promising models efficiently. If the proxy metric provides a strong relative ranking of models, this enables the discovery of high performing models when trained to convergence. For example, one common proxy metric is the accuracy on the validation set after a short amount of training. This proxy metric was successfully used in the original NAS paper. In this work, as well as in SMASH and ENAS, the proxy metric is the validation accuracy obtained by activating individual architectures in the one shot model.

The one-shot model is a standard large neural network trained using SGD with Momentum. To make sure that the one-shot model accuracies for specific architectures correlate well with stand-alone model accuracies we have to consider the aspects discussed below.

Robustness to Co-adaptation. At evaluation time, we zero out large portions of the one-shot model to evaluate specific architectures. If we train the one-shot architecture naively, the components can co-adapt. Removing operations – even unimportant ones – from the network can cause the quality of the model's predictions to degrade severely. The correlations between one-shot and stand-alone model accuracies also degrade.

We incorporate path dropout at model training time in order to ensure that the model is robust to such changes. When training the one-shot model, we randomly zero out a subset of the ops for each batch of examples. We achieved good results by disabling path dropout at the beginning of training and gradually increasing the rate of dropout over time using a linear schedule. The dropout rate at the end of training is set to $r^{1/k}$ where $0 < r < 1$ is a hyper-parameter of

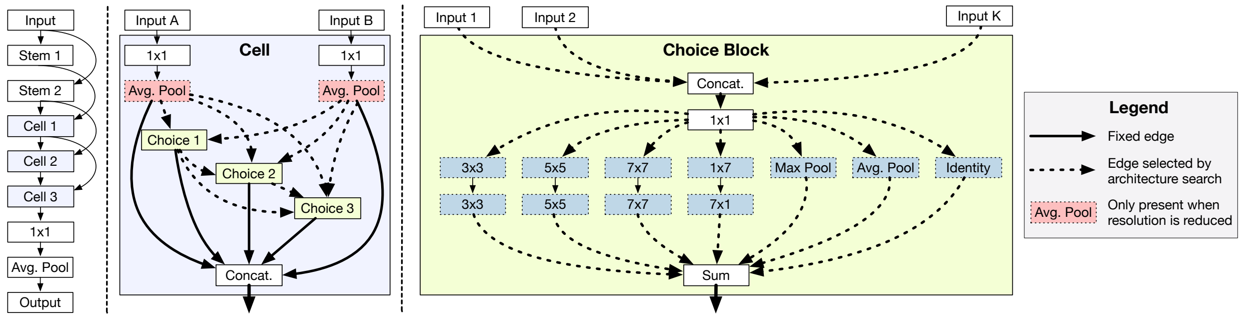

Figure 3. Diagram of the one-shot architecture used in our experiments. Solid lines indicate components that are present in every architecture, while dashed lines indicate optional components that are part of the search space.

the model and k is number of incoming paths to a given operation in the network. The higher the fan-in, the more likely each possible input is to be dropped out. However, the probability of dropping out *all* inputs to a node is kept constant regardless of its fan-in. Suppose $r = 0.05$. If a node has $k = 2$ inputs then each one will independently be dropped out with probability $0.05^{1/2} \approx 0.22$ and will be retained with probability 0.78. If a node has $k = 7$ incoming paths then each one will independently be dropped out with probability $0.05^{1/7} \approx 0.65$ and will be retained with probability 0.35. In both cases, the probability of dropping out all of the op's incoming paths is 5%.

Within a single cell, different operations are dropped out independently of each other. If a model contains multiple cells, however, the same operations will be dropped out in each one. We found that independently dropping out different paths within the same cell was beneficial, but did not investigate the behavior of dropout across cells.

Stabilizing Model Training. One-shot model training was highly unstable in early experiments. We found that a careful application of batch normalization could be used to stabilize training. We experimented with the orders BN-Relu-Conv and Relu-BN-Conv. While both showed promise, we use the former for the experiments reported in this paper. When the one-shot model is used to evaluate a candidate architecture from the search space, we zero out a subset of its operations. Doing so changes the batch statistics at each layer. Because we do not know the batch statistics for a candidate architecture in advance, batch normalization is applied exactly the same way at evaluation time as during training – computing the batch statistics on the fly.

A variant of ghost batch normalization (Hoffer et al., 2017) further stabilized training. One-shot model training tended to become unstable if we dropped out the same subset of paths for every example within a single batch. We found, however, that it could be stabilized if we dropped out different paths on different subsets of the examples. We initially tried dropping different paths independently for every example in the batch. However, this approach didn't work well with batch normalization, which is better able to compute batch statistics when the same paths are dropped out for multiple examples. As a compromise, we partition each training batch into multiple ghost batches. A single training batch might contain 1024 examples, which can be partitioned into 32 ghost batches of size 32. We drop out the same paths for each example within a ghost batch, but drop out different paths for different ghost batches.

Preventing Over-regularization. A given convolutional layer might only be used for a subset of the architectures in the search space. During training, L2 regularization is applied only to parts of our model that are used by the current architecture. Without this change, layers that are dropped out frequently are regularized more.

Method	Param $\times 10^6$	Accuracy
ENAS General	34.9	95.8
ENAS masks	12.6	95.7
ENAS skip	14.1	95.0
ENAS skip large	38.0	96.1
ENAS Cell search	4.6	96.5
NASNet-A	3.3	96.6
SMASHv2	16.0	96.0
One-Shot Top ($F = 16$)	0.7 ± 0.1	94.6 ± 0.2
One-Shot Top ($F = 32$)	2.7 ± 0.3	95.5 ± 0.1
One-Shot Top ($F = 64$)	10.4 ± 1.0	95.9 ± 0.2
One-Shot Top ($F = 128$)	41.3 ± 4.0	96.1 ± 0.2
One-Shot Small ($F = 16$)	0.4 ± 0.01	94.6 ± 0.2
One-Shot Small ($F = 32$)	1.3 ± 0.04	95.6 ± 0.2
One-Shot Small ($F = 64$)	5.0 ± 0.2	96.0 ± 0.1
One-Shot Small ($F = 128$)	19.3 ± 0.6	96.1 ± 0.2
All On ($F = 16$)	1.3	95.0
All On ($F = 32$)	4.8	95.6
All On ($F = 64$)	18.5	96.0
All On ($F = 128$)	72.7	96.2
Random ($F = 16$)	0.5 ± 0.2	94.1 ± 0.5
Random ($F = 32$)	1.7 ± 0.7	95.0 ± 0.5
Random ($F = 64$)	6.7 ± 2.6	95.6 ± 0.2
Random ($F = 128$)	26.4 ± 10.5	95.8 ± 0.2

Table 1. Architecture search on CIFAR-10. We evaluate ten models and report the mean x and standard deviation y as $x \pm y$.

3.3. Evaluating Candidate Architectures

Once the one-shot model is trained, we use it to evaluate the performance of many different architectures on a held-out validation set. In our experiments, architectures are sampled independently from a fixed probability distribution, following Brock et al. (2017). We note that random search could be replaced by other search methods, such as evolutionary algorithms or neural network-based reinforcement learning.

3.4. Final Model Selection and Training

The output of the search is a list of candidate architectures ranked by one-shot accuracy. After completing a search, one can retrain the best-performing architectures from scratch. Depending on the amount of compute resources available and the model accuracy requirements, one can additionally screen and hyperparameter tune many of the best-performing models as a post-processing step. However, other efficient architecture search methods elide this accuracy-boosting step, and we follow suit for purposes of comparison with these methods. To evaluate the use of a one-shot model, we sample the best-performing architectures found by our search. On CIFAR-10, each architecture is trained from scratch for 300 epochs on the full training set, then evaluated on the test set. On ImageNet, each architecture is trained for 200 epochs. Experiments were implemented using TensorFlow (Abadi et al., 2016).

It is possible to either scale up an architecture (to increase its accuracy) or scale down (to reduce its inference cost). In our experiments, we scale up architectures by increasing the number of filters. It may be possible to increase the number of cells in the model to further improve performance, but we use a fixed depth of six cells in all of our CIFAR-10 experiments and eight cells in our ImageNet experiments.

4. One-Shot Model Experiments

In this section, we explain and analyze the steps of architecture search. The goal is twofold: (1) to show that our approach is competitive with existing approaches for one-shot architecture search and (2) to provide insights into what makes one-shot architecture search possible.

On CIFAR-10 we used a 45,000 element training set, 5,000 element validation set, and 10,000 element test set. ImageNet was partitioned into a 1,281,167 training set, 50,046 element validation set, and 50,000 element test set. One-shot model accuracies and accuracies from abbreviated training were computed on the validation sets, while the final accuracies of architectures were computed on the test sets.

4.1. Experiments on CIFAR-10

Training the One-Shot Model. Given the search space detailed in the previous section, we begin by training a one-shot model on CIFAR-10. Each one-shot model was trained for 5,000 - 10,000 steps (113 - 225 epochs) on a cluster of 16 P100 GPUs. Each worker used a batch size of 64, which was divided into two ghost batches of size 32. We used a global learning rate of 0.1 and Nesterov momentum 0.9.[1] Increasing the number of training steps improved the correlations between one-shot and stand-alone model accuracies in our experiments, but only slightly. We therefore used a shorter training period for our initial hyperparameter tuning experiments and a longer period for the model that was used in our large-scale architecture search.

Impact of Dropout Rate. Compared with vanilla SGD, one-shot model training introduces just one new hyperparameter: the dropout rate. The value of this hyperparameter is important, however, and it must be tuned carefully in order to achieve good correlations between one-shot and stand-alone model accuracies.

To demonstrate its importance, we trained one-shot models with varying dropout rates. Following the setup described at the beginning of the section, each one-shot model was trained for 5,000 steps (113 epochs) using Synchronous SGD with 16 workers. The dropout rates in these experiments were kept constant throughout training.

The results of our experiment are shown in Figure 4. When the dropout rate is very low (i.e., most of the paths in the one-shot model are retained at each training step), the correlation plots develop a “snout.” A few of the architectures in the search space receive relatively high accuracies from the one-shot model. But while these architectures are usually good, the ones that receive the highest accuracies are not necessarily the best ones in the search space. Most of the architectures, however, receive very low accuracies. Due to the low dropout rate at training time, the one-shot model is ill-prepared for large portions of the model to be zeroed out at evaluation time.

When the dropout rate is very high (i.e., most of the paths in the one-shot model are dropped out at each training step), we encounter a different problem: the extent to which the one-shot model focuses its capacity on the most useful paths in the network is greatly diminished. Instead of ranging from 0.3 to 0.9, the one-shot model accuracies now range between 0.66 to 0.78. In early experiments, we found that high dropout rates yielded respectable results when the one-shot models were relatively shallow (e.g., networks containing

[1]In early experiments where the dropout rate was kept constant, using at least 16 ghost batches helped stabilize training. Gradually increasing the dropout rate over time may make it possible to use fewer workers, but we did not explore this in depth.

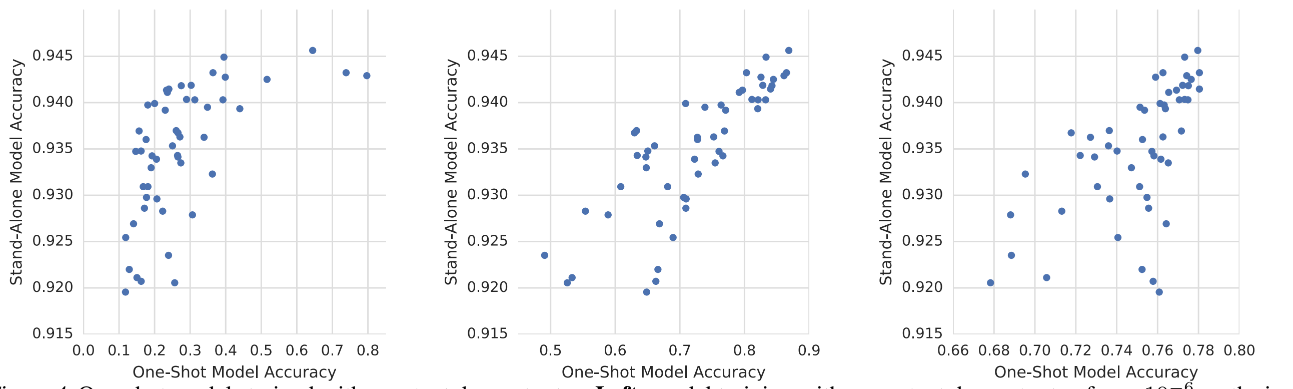

Figure 4. One-shot models trained with constant dropout rates. **Left**: model training with a constant dropout rate of $r = 10^{-6}$; paths in the network are retained more than 85% of the time. **Middle**: model trained with a constant dropout rate of $r = 0.01$, a moderate value. **Right**: model trained with a constant dropout rate of $r = 0.3$, a high value where some paths are retained only 15% of the time.

up to 12 layers in total). However, using a high dropout rate became increasingly problematic as the search space grew more complex and the one-shot model grew deeper.

One-Shot Model Training and Evaluation. For the one-shot model used in our large-scale architecture search experiments, we made two changes from the smaller-scale experiments described above. First: we increased the number of training steps from 5,000 to 10,000. Second: we allowed the dropout rate to increase over time. At the start of training, dropout was effectively disabled, while at the end of training, we had a dropout rate determined by the coefficient $r = 0.1$. Between these two points, the rate of dropout for each operation in the network was increased linearly over the course of training.

Armed with a calibrated one-shot model, we randomly sampled around 20,000 architectures from the search space, and computed the accuracy of each one on the one-shot model. We then divided the architectures into buckets based on their accuracies, and sampled four architectures from each bucket. Finally, we retrained each of the sampled architectures from scratch for around 28 epochs with a batch size of 64 and a ghost batch size of 32. These stand-alone model accuracies were averaged across five runs. Using the one-shot model, each architecture took around 15 seconds to evaluate on a P100 GPU. It took around 80 GPU-hours to evaluate all 20,000 architectures. However, we spent little time optimizing the code, and the search was trivially parallelizable. We believe that with more engineering effort, the cost of a search could be decreased dramatically.

We then compared these stand-alone model accuracies to the one-shot model accuracies. Figure 5 shows that there is a near-monotonic correlation between the two. This confirms that in our experimental setup, both proxy metrics are likely to favor similar network architectures.

On the bottom of Figure 5 we take a look at the histogram

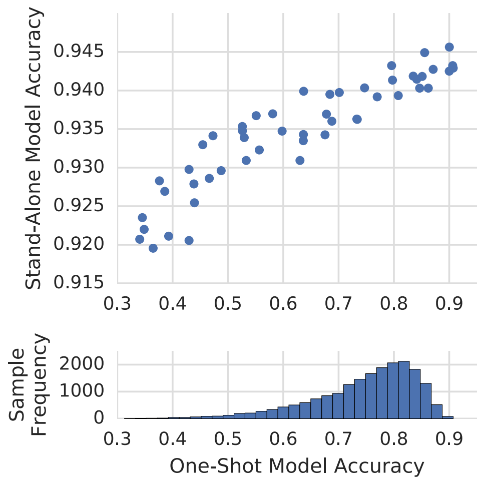

Figure 5. **Top:** Comparison of one-shot and stand-alone model accuracies from a stratified sample of architectures. Stand-alone models were trained for an abbreviated period of around 28 epochs, and stand-alone model accuracies were averaged across 5 runs. **Bottom:** Distribution of sampled one-shot model accuracies.

of one-shot model accuracies. About 34% of architectures receive one-shot model accuracies of 0.8 or above, while only 9% have accuracies of 0.85 or above and only 1% have accuracies of 0.88 or above. The quality improvements of better-performing models are modest but still noticeable: after 25 epochs of training, models in our sample with one-shot accuracies from 0.8 - 0.85 have average validation set accuracies of around 0.941 after 28 epochs of stand-alone training. Meanwhile, architectures with one-shot accuracies of 0.90 or above have average validation set accuracies of around 0.944. This suggests that the truly high performing architectures only make up a limited part of the search space. This will be confirmed by a random sample that is trained until convergence in the next experiment.

Final Model Selection and Training. After screening 20,000 random architectures with the one-shot model, we performed another selection. From the top 100 architectures, we took a stratified sample of 10 for further evaluation. We then increased the number of filters in the models and trained them for 300 epochs without additional hyper-parameter tuning. The results are shown in the "One-Shot Top" section of Table 1. When comparing based on the number of parameters in the model, our approach is competitive with SMASHv2 and almost all ENAS variations. We parameterize our models based on F, the number of filters in the first convolutional layer. When $F = 64$, we obtain an average accuracy of 95.9% with about 10M parameters. SMASH has 96.0% accuracy with 16M parameters. If we make the model even bigger ($F = 128$), we get 96.1% accuracy. The best models get up to 96.5% accuracy with around 41M parameters.

We conclude that our approach is competitive with SMASH and all but one ENAS variation. This shows that our simplified architecture search is solid and can be used to better understand one-shot model behaviour. This we will analyze in more detail now and in the next section.

Comparing our sample of 10 top architectures ("One-Shot Top") against 10 randomly sampled architectures ("Random"), we find that the top architectures have better accuracies by around .5% absolute. However, they also have about 1.6x as many parameters. A natural follow-up question is whether the one-shot model always favors the architectures with the most parameters. To answer this question, we searched for the smallest architectures whose one-shot model accuracies exceeded a certain threshold (roughly in the 90th percentile). The resulting architectures ("One-Shot Small") have nearly the same accuracies as the top models but fewer parameters than both the top and random models.

The baseline *All On* trains the model with all paths turned on. This approach gets .1% higher accuracies, with the best models obtaining 96.5%. However, the number of parameters is nearly 2x that of *One-Shot Top* and 4x that of *One-Shot Small*. In this respect, we could see architecture search as a way to prune the less useful parts of the model.

4.2. Experiments on ImageNet

To evaluate our approach on larger datasets, we ran an architecture search directly on ImageNet using Cloud TPUs. The One-Shot model was trained for 15k steps (about 47 epochs or 6 hours) with a batch size of 4,096 on four Cloud TPUs (16 chips). Each candidate architecture took 1-2 TPU chip-minutes to evaluate. The final models were trained and evaluated using 224×224 input images.

Results, shown in Table 2, are consistent with our CIFAR-10 experiments. The top-performing models have better accuracies than random models, but also more parameters. By searching for the smallest models whose one-shot accuracies exceed a predetermined threshold, we are able to improve the trade-off between quality and model size.

Method	Param $\times 10^6$	Accuracy
One-Shot Top ($F = 16$)	3.1 ± 0.4	70.1 ± 0.6
One-Shot Top ($F = 24$)	6.8 ± 0.9	73.8 ± 0.4
One-Shot Top ($F = 32$)	11.9 ± 1.5	75.2 ± 0.4
One-Shot Small ($F = 16$)	1.4 ± 0.4	67.9 ± 0.5
One-Shot Small ($F = 24$)	3.0 ± 0.8	72.4 ± 0.4
One-Shot Small ($F = 32$)	5.1 ± 1.5	74.2 ± 0.3
Random ($F = 16$)	2.0 ± 0.5	67.9 ± 1.0
Random ($F = 24$)	4.4 ± 1.0	72.2 ± 0.7
Random ($F = 32$)	7.7 ± 1.9	74.1 ± 0.6

Table 2. Architecture search results on ImageNet.

We also discovered that although our models have significantly better accuracies than NASNet and MobileNet (Howard et al., 2017) for the same number of parameters, they have higher inference costs (measured in terms of multiply-adds). In follow-up experiments in Appendix A, we tweaked our top models in order to roughly match the computation of these mobile-sized models.

5. Understanding One-Shot Models

We next discuss why the same fixed set of model weights can be shared across many different architectures. One key observation from Figure 5 is that while one-shot model accuracies range from 30% to 90% on CIFAR-10, stand-alone model accuracies range from 92.0% to 94.5%. Although the accuracies of the best models decrease by only 5 - 10 percentage points when we switch from stand-alone to one-shot training, the accuracies of less promising architectures drop by as much as 60 percentage points. Similar behavior is shown in Brock et al. (2017), where one-shot model accuracies range from 10% to 60% on CIFAR-100 while stand-alone model accuracies range from 70% to 75%. Why should the spread of one-shot model accuracies be so much larger than the spread of stand-alone model accuracies?

Our hypothesis is that the one-shot model learns which operations in the network are most useful, and comes to rely on these operations when they are available. Removing the less important operations from the network has relatively little influence on the model's predictions and only a modest effect on its final prediction accuracy. Removing the most important operations from the network, however, can lead to dramatic changes in the model's predictions and a large drop in prediction accuracy.

In order to test this hypothesis, we sampled a collection of architectures where almost all the operations in the one-shot model were enabled (base dropout rate $r = 10^{-8}$). We compared the predictions made by these *reference architec-*

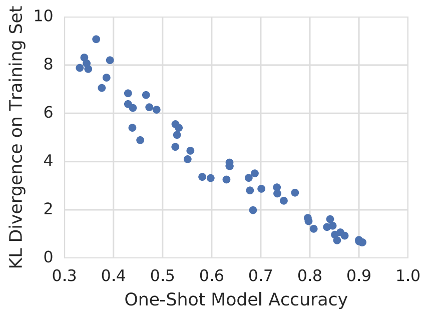

Figure 6. Comparison of one-shot model accuracies on the validation set vs. symmetrized KL divergences on the training set. Lower KL divergences on the training set are strongly correlated with higher validation set accuracies.

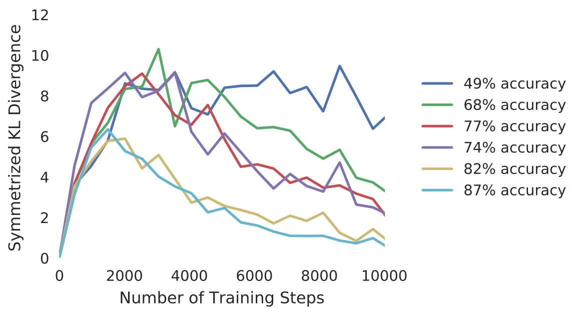

Figure 7. **Left:** symmetrized KL divergences from six architectures over the course of one-shot model training. **Right:** the one-shot validation accuracies of all six architectures at the end of training. The top-to-bottom order of the KL divergences in the plot mirrors the top-to-bottom order of the entries in the legend.

tures against the predictions of *candidate architectures* with fewer operations that were sampled from our actual search space. The comparison was performed on batches examples from the training set. If our hypothesis is correct then we should expect the predictions made by the best-performing models to be similar to the predictions made when all the operations in the network are enabled.

We use *symmetrized KL divergence* to quantify the extent to which candidate architectures' predictions differ from those of reference architectures on a given example. Our one-shot model is trained using a logistic loss function, so the reference output can be interpreted as a probability distribution $(p_1, p_2, \dots, p_n)$ over output classes. Compared against a candidate architecture that produces output distribution $(q_1, q_2, \dots, q_n)$, the KL divergence is estimated as $D_{\mathrm{KL}}(p \mid\mid q) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i}$. We use $D_{\mathrm{KL}}(p \mid\mid q) + D_{\mathrm{KL}}(q \mid\mid p)$, the *symmetrized KL divergence* between p and q, to quantify the similarity of the two distributions. If the distributions are nearly identical for the current training example then the symmetrized KL divergence will be close to 0. Conversely, the symmetrized KL divergence can grow quite large if the distributions are very different. We compute the KL divergence on 64 random examples from the training set and report the average.

The results of our experiment, shown in Figure 6, are striking. KL divergences measured on the *training set* are strongly correlated with prediction accuracies measured on the *validation set*. Furthermore, the KL divergences are computed without making use of any information about the examples' training labels. Combined with the results of Figure 5, this means that the closer a candidate architecture's predictions are to those of our reference architectures (where most of the operations in the one-shot model are turned on), the higher its quality typically is during stand-alone training. Weight sharing implicitly forces the one-shot model to identify and focus on the operations that are most useful for generating good predictions.

The same experiment suggests an explanation for the exaggerated difference in validation set accuracies that we observe when using the one-shot model. If certain operations are especially useful to the one-shot model, it will come to rely on those operations' outputs when generating predictions. Removing them from the network will result in catastrophic damage and extremely low validation set accuracies. On the other hand, less-useful operations can be removed from the network with only a modest impact on the one-shot model's predictions.

We next explore how the KL divergences evolve over time. We sampled six different architectures and tracked their symmetrized KL divergences over the course of training. Results are shown in Figure 7. Initially, all of the KL divergences are low because the model's predictions are initially low-confidence, and each output class is assigned a roughly equal probability. Our model gradually becomes more confident in its predictions, and the predictions of different architectures begin to separate. This accounts for the spike in KL divergences. Later in training, the most useful operations in the network have a strong influence over the model's predictions, and receive low KL divergences.

6. Conclusion

We analyzed a class of efficient architecture search methods based on weight sharing in a model containing the entire search space of architectures. We designed a training method and search space to address the fundamental challenges in making these methods work. Through this simplified lens, we explained how the fixed set of weights in a one-shot model can be used to predict the performance of stand-alone architectures, demonstrating that one shot architecture search only needs gradient descent, not reinforcement learning or hypernetworks, to work well.

Acknowledgements

We would like to thank David Dohan, Irwan Bello, Vishy Tirumalashetty, Samy Bengio, Jeff Dean, Geoff Hinton, and the rest of the Google Brain team for their helpful comments and discussions. We are grateful to Ryan Sepassi for his infrastructure work.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pp. 3981–3989, 2016.

Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

Bayer, J., Wierstra, D., Togelius, J., and Schmidhuber, J. Evolving memory cell structures for sequence learning. In *International Conference on Artificial Neural Networks*, pp. 755–764. Springer, 2009.

Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. Neural optimizer search with reinforcement learning. *arXiv preprint arXiv:1709.07417*, 2017.

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

Bergstra, J., Yamins, D., and Cox, D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning*, pp. 115–123, 2013.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pp. 2546–2554, 2011.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. SMASH: One-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Reinforcement learning for architecture search by network transformation. *arXiv preprint arXiv:1707.04873*, 2017.

Elsken, T., Metzen, J.-H., and Hutter, F. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

Gordon, A., Eban, E., Nachum, O., Chen, B., Yang, T., and Choi, E. Morphnet: Fast & simple resource-constrained structure learning of deep networks. *CoRR*, abs/1711.06798, 2017. URL `http://arxiv.org/abs/1711.06798`.

Hochreiter, S., Younger, A. S., and Conwell, P. R. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pp. 87–94. Springer, 2001.

Hoffer, E., Hubara, I., and Soudry, D. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pp. 1729–1739, 2017.

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

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Jozefowicz, R., Zaremba, W., and Sutskever, I. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pp. 2342–2350, 2015.

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

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017b.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Navruzyan, A., Duffy, N., and Hodjat, B. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Faster discovery of neural architectures by searching for paths in a large model. *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=ByQZjx-0-`.

Ramachandran, P., Zoph, B., and Le, Q. V. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q., and Kurakin, A. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

Schmidhuber, J. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., and Adams, R. Scalable bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pp. 2171–2180, 2015.

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Thrun, S. and Pratt, L. *Learning to learn*. Springer Science & Business Media, 2012.

Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., de Freitas, N., and Sohl-Dickstein, J. Learned optimizers that scale and generalize. *arXiv preprint arXiv:1703.04813*, 2017.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pp. 5–32. Springer, 1992.

Xie, L. and Yuille, A. Genetic cnn. *arXiv preprint arXiv:1703.01513*, 2017.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2016.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.