Improving weight-sharing Neural Architecture Search with a marginal likelihood estimator

Candidate number: 1047586

University of Oxford

A thesis submitted for the degree of

MSc. Computer Science

Trinity 2021

Word count: 22 008

Abstract

Neural architecture search (NAS) aims to automatize the design of well-performing neural networks, which have recently proven to be a very influential class of models. However, the inability to efficiently predict the generalization performance of a neural network has long been a significant issue in NAS. We use Sum-over-Training-Losses (SoTL), a recently proposed theoretically-inspired generalization estimator based on Bayesian model selection, to get a computationally cheap yet accurate estimator of model performance for modern NAS weight-sharing methods. The standard generalization estimator is validation accuracy, which was previously shown to be unreliable for weight-sharing NAS.

We design ways to integrate SoTL into all major classes of weight-sharing NAS, including discretized one-shot and differentiable NAS. In discretized one-shot NAS, we show that using SoTL rather than validation accuracy to rank architectures leads to significantly better rankings with respect to the ground truth test accuracy as well as higher top-10 performances of selected architectures. For differentiable NAS, we likewise show that optimizing SoTL rather than validation loss leads to a more stable search resulting in better final architectures. Moreover, our experiments using SoTL challenge several stylized facts about weight-sharing NAS, and we advance the understanding of those algorithms by providing novel counter-examples to common NAS assumptions.

Contents

Chapter 1

Introduction

1.1 Motivation

Neural architecture search (NAS) is an emerging special case of the more general model selection problem applied to deep learning. Whereas NAS has only become popular in the last decade following the rapid rise of neural networks into mainstream machine learning practice, model selection has been an important topic in statistical inference at least since the early 20th century (Fisher, 1922; Fisher, 1938). However, the statistical tools developed for model selection in theoretically well-understood models such as linear regression are generally not applicable to selecting neural network architectures. Separate algorithms thus have to be developed in order to perform architecture search for deep learning. Furthermore, efficiency is paramount when it comes to neural networks because training can be very computationally expensive.

In this work, we use the recently proposed Sum-over-Training-Losses (SoTL) metric as an estimator of model generalization capability. This metric has a principled Bayesian interpretation for linear models and remains practically useful for neural networks, where it has previously been shown to significantly improve query-based neural architecture search (Ru et al., 2020). We show how it can be incorporated into other popular NAS algorithms based on weight-sharing and discuss ways of efficiently computing gradients of SoTL for use in gradient-based optimization. Our work shows that SoTL can significantly improve the performance of many state-of-the-art NAS algorithms such as DARTS (Liu et al., 2018) or SPOS (Guo et al., 2020) while being robust across datasets (CIFAR10, CIFAR100 and ImageNet) and search spaces (NASBench201, NASBench-1shot1 and the DARTS search space).

In summary, our main contributions are as follows:

- We show that using SoTL instead of validation-based estimators of generalization can significantly improve model selection in weight-sharing one-shot NAS, especially in the context of efficient NAS using early stopping, and alleviate many of its shortcomings observed in the literature
- By applying SoTL to differentiable NAS algorithms such as DARTS, we demonstrate that it simultaneously decreases the search cost while improving the final performance compared to baselines
- Our experiments reveal several interesting discoveries about the implicit biases of weight-sharing, and we deepen the empirical understanding of both one-shot and differentiable NAS

1.2 Thesis structure

The rest of the thesis is structured as follows. In Chapter 2, we briefly introduce the basics of deep learning and review the historical development of NAS to provide a high-level perspective of the most popular algorithms. A more thorough explanation of the theoretical background of SoTL is included in Chapter 3 along with a comprehensive exposition of the NAS benchmarks we study in this work. Afterward, we discuss the details of the most popular NAS algorithms to develop an understanding for the challenges of integrating SoTL into those algorithms, and to obtain an intuition for how using SoTL is expected to improve the results.

In Chapter 4, we show the results of integrating SoTL into several major NAS algorithms across a multitude of standard benchmarks. We describe the benefits of SoTL in several small-scale problems to sanity check our approach. Next, the improvements in performance coming from SoTL in discretized one-shot NAS algorithms are discussed. We also make several novel discoveries regarding the bias of weight-sharing in one-shot NAS. The last part of the experimental section concerns applying SoTL to differentiable NAS. We again show that our proposed changes lead to a better and more stable search across a variety of benchmarks. Moreover, several intriguing findings are made regarding the behavior of DARTS that notably extend the scope of understanding in differentiable NAS.

The thesis concludes with Chapter 5, in which we summarize the goals of this project and their fulfillment. We also reiterate the novel discoveries we have made and outline several promising research directions for future work.

Chapter 2

Background

2.1 Introduction to deep learning

Neural networks are a family of machine learning models which became very popular in the last decade, but their foundations have already been described in the 20th century (Lecun et al., 1998; Rumelhart et al., 1986). Neural networks and the associated deep learning paradigm have since become ubiquitous and the defacto default choice, especially when dealing with unstructured, high-dimensional data such as images or text, where their performance far surpassed that of previous models since early 2010s (Mikolov et al., 2013; Krizhevsky et al., 2012). We briefly introduce the main building blocks of neural network architectures that are most commonly used in neural architecture search, but our exposition will be brief out of necessity. Furthermore, most NAS algorithms treat the relevant building blocks as black-boxes, and a detailed understanding is therefore unnecessary. More detail on the basics of deep learning can be found in textbooks such as (Goodfellow et al., 2016).

In their most basic form, neural networks can be described as a series of *neurons* (also called *units*) that successively apply transformations in the form of

$$f(\mathbf{w^T x} + \mathbf{b}), \tag{2.1}$$

where $\mathbf{x} \in R^D$ is the D-dimensional input data, $\mathbf{w}$ are the trainable weights, $\mathbf{b}$ are constant terms called the bias and f is called the activation function. A neuron is equivalent to linear regression when f is the identity function and to logistic regression when f is the logistic function. In practice, having nonlinear activation functions such as ReLu (Nair et al., 2010) or various sigmoid functions is crucial for the success of neural networks.

Furthermore, while linear regression is exactly equivalent to a neural network with a single neuron and the identity activation function, it is common to stack several layers of neurons in a row, resulting in *deep* neural networks (DNNs). The prototypical example is the multi-layer perceptron (MLP) that simply stacks several neurons using fully connected layers, which apply the transformation in Eq. (2.1) with some activation function. An example of a multi-layer perceptron with one hidden layer is shown in Figure 2.1, highlighting the complex connectivity and multiple layers pattern typical for neural networks.

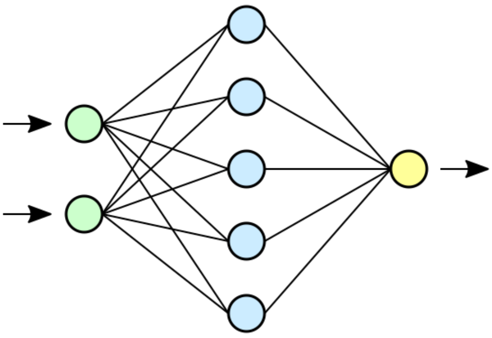


Figure 2.1: A simple multi-layer perceptron with two input neurons (green), one hidden layer (blue) and one output neuron (yellow). The edges between neurons represent input-output connections where each neuron is fully connected to all neurons in the preceding layer. Reprinted from Mysid (2006).

Various other neuron designs are in wide use. This work mainly concerns Convolutional Neural Networks (CNNs), which replace the fully connected pattern in Eq. (2.1) with only local connections by applying convolutional *kernels*. The output of a convolutional layer is sometimes referred to as a *feature map*. In convolutional layers, the output dimensions do not depend on the entire preceding feature map as in multi-layer perceptrons. The input dimensions that a feature map dimension depends on are referred to as the *receptive field*, whose size is known as the *kernel size*. Moreover, the weights in each kernel are consecutively applied over different regions of the whole input image, which means that convolutions introduce a form of parameter sharing. The kernels can be imagined as sliding over the input image, always applying the convolutional kernel to compute the value in one output dimension at a time. A visualization of the convolutional operation can be seen in Figure 2.2. The size of the receptive field is an important parameter of convolutional layers, and it is generally explicitly written out in the layer description such as *Conv* 3×3, meaning a 3×3 receptive field. It is typical to stack multiple convolutional kernels, the count of which determines the number of output channels.

Convolutional neural networks have shown to be particularly successful when working with image data. Similar to multi-layer perceptrons, an empirically proven

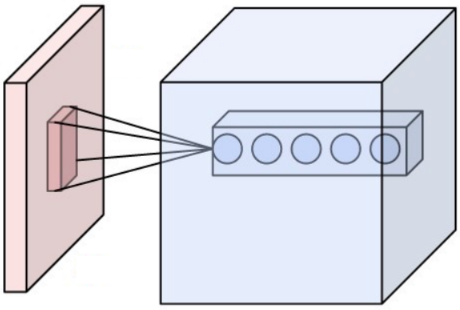


Figure 2.2: An illustration of the connectivity pattern in convolutional layers. Each dimension in the output feature map (blue) is connected to only a small square region in the input (red). In order to compose the whole output feature map, the receptive field would slide across the input and repeatedly apply the kernel. Reprinted from *Convolutional neural network* (2021).

network design is to simply stack a number of convolutional cells to form a deep network. Several of the most famous neural networks such as ResNet (He et al., 2016) or Transformers (Vaswani et al., 2017) are examples of this cell-stacking design. CNNs also frequently use other operations such as skip connections (He et al., 2016), which simply apply the identity operation to make it easier to propagate gradients over long distances in the network, and various kinds of pooling such as *max pool* or *avg pool*, which are similar to convolutions but they output either the maximum or the average of the receptive field.

Deep neural networks are popular because of their remarkably high generalization performance on unseen data. Even though they are typically heavily overparameterized and have up to billions of parameters (Brown et al., 2020), they generalize very well despite being trained on datasets many times smaller than the number of parameters. The reasons for this remain poorly understood despite being a very active area of research (Jiang et al., 2020; Arora et al., 2018; Neyshabur et al., 2015). Other machine learning models such as kernel-based methods (Schölkopf et al., 2002) or decision trees (Breiman, 2001) might also be able to perfectly fit the training data, but often fail to match the generalization performance of neural networks, especially on high-dimensional data such as those in computer vision or natural language processing (Mikolov et al., 2013; Krizhevsky et al., 2012). We further discuss the issue of deep learning generalization together with a Bayesian point of view relevant to SoTL in Section 3.1.

2.2 Neural architecture search

The objective of Neural Architecture Search is to automatically find a neural network design that achieves the best possible generalization performance. Architecture search was initially treated as a direct subset of hyperparameter optimization and optimized with the same approaches, using techniques such as random search (Bergstra et al., 2012), Bayesian optimization (Snoek et al., 2012; Snoek et al., 2015) or bandit-based algorithms (Li et al., 2018b). However, the architecture search spaces in these approaches were heavily constrained, and architectures were evaluated by training from scratch, which led to a very slow and inefficient search.

Earliest specialized NAS algorithms (Real et al., 2019; Zoph et al., 2018) still relied on training each architecture from scratch, which ended up costing thousands of GPU days for each search run. However, increased focus on engineering the search space meant that the best discovered architectures were able to achieve state-of-the-art performances on selected benchmarks. In particular, the focus on cell-based search spaces introduced in NASNet (Zoph et al., 2018) has been widely adopted by subsequent literature. The success of repeating identical cells to form a network was already discussed in Section 2.1. The cell-based design is particularly important for NAS since it is then only necessary to search for a single cell rather than the whole network, which significantly shrinks the search space.

Another major milestone for NAS was the introduction of weight-sharing, which tries to train all architectures at once by sharing weights in a large supernetwork, and it has allowed to significantly amortize the cost of searching in a large search space. In order to develop an intuition for the weight-sharing idea, we introduce several toy examples of what it might look like in practice. First, imagine a fully connected layer in which the architectural choice is whether to apply $tanh$ or $ReLu$ nonlinearity, meaning the layer output is either $tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$ or $ReLu(\mathbf{W}\mathbf{x} + \mathbf{b})$. When doing architecture search via weight-sharing, the $\mathbf{W}$ and $\mathbf{b}$ weight matrices would be exactly the same in both $tanh(..)$ and $ReLu(..)$, Only a single network is then trained (i.e. the supernetwork) that alternates between using $tanh$ and $ReLu$ so that the network accommodates both options at the same time. Non-weight-sharing NAS would train two separate networks using either $tanh$ or $ReLu$ before performing the model selection. As a more realistic example, imagine a large network with 20 layers with the first 19 layers fixed. The goal is to find the best generalizing option for the 20th layer among layers A and B. To do that, weight-sharing would again train only a single network with the first 19 layers shared and have the network alternate between using layer A and layer B as the final layer. This leads to the final supernetwork weights being an average of sorts between the weights that would be obtained from training only with layer A and only with layer B. The choice between layers A and B corresponds to two separate architectures within the architecture search. Both toy examples are equivalent to what is known as single-path one-shot NAS, which is one of the most popular weight-sharing NAS algorithms.

The weight-sharing training protocol implicitly assumes that it is possible for the supernetwork training to accommodate all the individual architecture subnetworks with a single set of weights, and the role of the search algorithm is to both efficiently train the supernetwork and then extract the final best architecture. Algorithms following this paradigm were able to bring the search cost down to single digit GPU days while often also improving the search performance (Bender et al., 2018; Liu et al., 2018; Pham et al., 2018). The two major classes of weight-sharing algorithms are one-shot and differentiable NAS algorithms, respectively. We briefly introduce them now before discussing them in detail in Chapter 3.

Training the supernetwork comes with its own challenges, and we first describe the issues faced by the main differentiable NAS algorithms. DARTS (Liu et al., 2018) trains all the supernetwork weights at once using a linear relaxation, which requires storing the computational graph of all architectures in memory at once. This severely limits the size of the supernetwork that can be searched. In the toy example with a 20 layer network above, DARTS would correspond to the network having two replicas of the 20th layer so that the network's forward/backward pass happens across both branches simultaneously rather than alternating between them. In practice, this also leads to a large *discretization gap* where the performance of architectures evaluated using the shared weights is not representative of their performance when trained separately using standard training protocols. Various modifications of the baseline DARTS algorithm have been proposed to alleviate the memory concerns (Chen et al., 2019), regularize training protocol to avoid large discretization gaps (Zela et al., 2019; Xu et al., 2019), reparameterize the architecture encoding (Chen et al., 2021; Chu et al., 2020) or use complex search algorithms to extract the final standalone architecture from the supernetwork (Wang et al., 2021). Nonetheless, the supernetwork optimization appears to be biased towards easy-to-train architectures (Shu et al., 2020) and the search performance can be unstable (Dong et al., 2020).

On the other hand, one-shot algorithms such as Single-Path One-Shot (Bender et al., 2018; Guo et al., 2020) only train one architecture (referred to as a *single-path* through the supernetwork) at a time, which reduces the overall memory costs of training the supernetwork to be equivalent to training a single architecture. However, those algorithms often fail to find top-performing architecture candidates. The final architecture selection usually proceeds by randomly sampling a number of architectures from the search space and computing their validation accuracy using weights inherited from the supernetwork. However, the ranking produced by reusing the supernetwork is usually poorly correlated with rankings based on the performance of architectures trained from scratch. This has been attributed to interference caused by sequential training of multiple architectures referred to as multi-model forgetting, alluding to its similarity to continual learning (Zhang et al., 2020a; Benyahia et al., 2019). Similar to the DARTS case, various inherent biases appear consistently present in single-path training, which biases the supernetwork towards certain operations and good-but-not-great architectures, causing rank disorder (Bender et al., 2018; Zhang et al., 2020b; Zhang et al., 2020b).

Even when the single-paths are sampled non-uniformly by using differentiable samplers based on the Gumbel-Softmax trick (Dong et al., 2019b; Xie et al., 2019) or other greedy sampling schemes (Dong et al., 2019a; You et al., 2020), the performance often falls short of differentiable NAS algorithms using the whole supernetwork. Interestingly, search spaces that are specially designed to minimize interference can achieve state-of-the-art performances in the search space without any additional training as all the subnetworks come out of the supernetwork fully trained (Yu et al., 2020a; Cai et al., 2019b).

For clarity, we note that one-shot simply means that only a single network is trained by the NAS search algorithm. As such, even DARTS (Liu et al., 2018) is sometimes referred to as one-shot, and similarly, algorithms using differentiable samplers are sometimes referred to as differentiable NAS rather than one-shot as we outlined above. For the purpose of this work, we will use the term *differentiable NAS* to refer to DARTS and other close variants that aim to jointly train all the architectures within the supernetwork, whereas *one-shot* will refer to NAS algorithms that only train single-paths at a time. The exact difference between DARTS and one-shot will be made clear in Section 3.3 and Section 3.4, respectively.

Despite the efforts to produce efficient NAS, random search has been shown to be a strong and consistent baseline (Li et al., 2020). Because the correlation between supernetwork and standalone performance of architectures is often weak or even negative, trying to exploit the supernetwork can effectively be equivalent to random search (Yang et al., 2020; Singh et al., 2019; Yu et al., 2020c). That said, NAS has proven to be particularly useful when searching for hardware-aware architectures with constraints on latency or parameter count where the current algorithms can produce strong results compared to purely human-designed architectures (Cai et al., 2019b; Cai et al., 2019a).

Our work on using SoTL tackles the problem of ranking single architectures in the one-shot scenario and of making the search in differentiable NAS properly favor the best generalizing architectures. There has been a recent surge of interest in improving the gradient computation in DARTS-like approaches (He et al., 2020; Zhang et al., 2021a) as well as the development of cheap yet effective estimators of architecture generalization performance (Abdelfattah et al., 2021; Mellor et al., 2021). In comparison to generalization estimators most commonly used in NAS practice such as early-stopped validation accuracy (Zoph et al., 2018; Pham et al., 2018; Li et al., 2018b), reduced model size (Real et al., 2019; Liu et al., 2018) or model-based architecture performance predictors (Domhan et al., 2015; Baker et al., 2017; Siems et al., 2020), the SoTL estimator typically offers higher fidelity of performance estimates at lower cost. Our work shows that SoTL can be incorporated into virtually all popular types of weight-sharing NAS algorithms, thus alleviating many of the long-standing issues in NAS.

Chapter 3

Methodology

In this section, we first provide more detail on the motivation of Sum-over-Training-Losses (SoTL) in Section 3.1 and discuss the NAS-specific benchmark datasets that we use in Section 3.2. We also explain in detail the two most influential families of NAS algorithms, namely DARTS (Liu et al., 2018) and variants in Section 3.3 and the discretized one-shot algorithms in Section 3.4. Most notably, we describe our approach for integrating SoTL into those algorithms.

3.1 SoTL

First, we define the Sum-over-Training-Losses (SoTL) metric and explain its importance as an estimator of optimization speed, and then show it has a principled Bayesian interpretation in specific settings as an estimate of the marginal likelihood used for Bayesian model selection.

In its basic form, Sum-over-Training-Losses simply sums over all the training losses during a model's optimization. Let $D = \{(x_1, y_1), (x_2, y_2), .., (x_n, y_n)\}$ be the training dataset, $f_{\theta_t}(x_{i(t)})$ represent a model's output for the training sample $x_{i(t)}$ in the t-th iteration with θ_t being the model weights at time t, L be a loss function and T be the total training iterations. We define SoTL as:

$$SoTL = \sum_{t=1}^{T} L(f_{\theta_t}(x_{i(t)}), y_{i(t)}). \tag{3.1}$$

Summing over a shorter period, typically one full epoch, has been shown to better correlate with generalization than summing over the entire history (Ru et al., 2020). Therefore, whenever we refer to computing SoTL in the rest of the manuscript, we will implicitly refer to the estimator computed over the last epoch unless otherwise specified. Furthermore, we note from Eq. (3.1) that SoTL can be interpreted as the area under the training loss curve. Because of this property, SoTL depends on model quality over the optimization trajectory, which contrasts with common model selection practices such as using the validation accuracy of only the final network weights. SoTL can subsequently be interpreted as a measure of training speed since a low value of SoTL implies that the training loss has been decreasing quickly during the whole training rather than only being low at the end.

Next, we discuss the connection between training speed and generalization. Generalization of deep neural networks has been an active research area in recent literature, and a large number of theoretically motivated estimators have been proposed (Arora et al., 2018; Long et al., 2020; Bartlett et al., 2017; Neyshabur et al., 2015). Those try to provide generalization bounds that would upper-bound the test error based on measures computed using only the training set. However, a large-scale study by Jiang et al. (2020) shows that the bounds are usually too loose to be empirically useful, or they might even be negatively correlated to generalization in practice. In fact, it is frequently observed that overparameterization of neural networks increases generalization rather than decreases it, which is in direct contrast to what many generalization bounds would predict.

Jiang et al. (2020) further show that a class of measures relying on the properties of optimization tend to be successful consistently. In particular, the number of iterations taken to reach 0.1 cross-entropy loss has a strong and robust correlation with generalization. Nakkiran et al. (2020) have shown that the observations about training speed determining generalization hold even for very large contemporary models such as Vision Transformers (Dosovitskiy et al., 2021) or Image-GPT (Chen et al., 2020a).

Next, we summarize the Bayesian interpretation of SoTL, which provides a theoretically principled connection between training speed and generalization (Lyle et al., 2020). For a dataset D, model M and its parameters θ, the marginal likelihood in Bayesian statistics is defined as:

$$P(D|M) = \int_{\theta} P(D|\theta, M)P(\theta|M)d\theta. \tag{3.2}$$

The marginal likelihood computation involves integrating over the model parameters θ, and it is specified both by defining the model form M and the weights prior $P(\theta|M)$. Using the marginal likelihood, one would typically choose the model M_i with the highest value of $P(D|M_i)$. It is possible to explicitly compare the posterior probabilities of two models by computing the Bayes factor (MacKay, 2002):

$$\frac{p(M_1|D)}{p(M_2|D)} = \frac{p(M_1)}{p(M_2)}\frac{p(D|M_1)}{p(D|M_2)} \tag{3.3}$$

From the form of the Bayes factor in Eq. (3.3), we see that it involves a ratio of prior model probabilities $p(M_i)$ and a ratio of marginal likelihoods $p(D|M_i)$. If we assume a uniform prior over the models $p(M_i) \propto 1$, which reflects no prior knowledge on which kind of model should be preferred, the Bayes factor is determined solely by the ratio of marginal likelihoods for the two models. This explains how we can pick the most probable model, i.e. maximize $p(M_i|D)$, by computing their marginal likelihoods, and it is sometimes referred to as Type-II maximum likelihood (MacKay, 2002).

Moreover, the formula for model evidence in Eq. (3.2) shows that computing the integral involves a trade-off between the prior $P(\theta|M)$ and the probability of the data $P(D|\theta, M)$. The trade-off arises because choosing a more complex model class M decreases $P(\theta|M)$ since the model now supports a larger class of hypotheses, and any single one is less likely as the overall probability still has to normalize to 1. In contrast, having a more complex model makes it possible to fit the data better, therefore increasing the $P(D|\theta, M)$ term. Thus in order to maximize the model evidence, we should try to have models as simple as possible which still fit the data as well as possible. This preference for simpler models is also known as Occam's factor (Rasmussen et al., 2000).

Building on the assumption that a high marginal likelihood is desirable, we now show that SoTL represents a lower bound on the marginal likelihood for linear models. If we abbreviate $P(D|M)$ by $P(D)$ and let $D_{<i} = (D_j)_{j=1}^{i-1}$ represent the first i samples of the dataset, we can use the probability chain rule to write the log marginal likelihood as:

$$\log P(D) = \log \prod_{i=1}^{n} P(D_i|D_{<i}) = \sum_{i=1}^{n} \log P(D_i|D_{<i}) = \sum_{i=1}^{n} \log(E_{P(\theta|D_{<i})}[P(D_i|\theta)]) \tag{3.4}$$

If we understand the negative log posterior predictive probability $-\log P(D_i|D_{<i})$ as a loss function, the log marginal likelihood is equal to a sum of losses that were incurred during training, where the training is represented by successive conditioning on the data points seen so far in $P(\cdot|D_{<i})$. Many common loss functions used in machine learning, such as the mean squared error or cross-entropy loss, have a probabilistic interpretation for a suitably chosen model, thus giving a direct correspondence between $-\log P(D_i|D_{<i})$ and training loss (Murphy, 2012).

Lyle et al. (2020) show that the interpretation of SoTL as model evidence lower bound is exact for Bayesian linear regression. Suppose we have data $D = (x_i, y_i)_{i=1}^N$ generated as $Y = \theta^T \Phi(X) + \epsilon \sim N(0, \sigma_N^2 I)$ for unknown θ, known σ_N^2 and design matrix Φ. A Gaussian prior is usually placed on θ. It can be shown that sampling θ from the prior and then optimizing via gradient descent on the data $D_{<i}$ gives samples from the exact posterior $P(\theta|D_{<i})$ (Matthews et al., 2017; Osband et al., 2018). Being able to sample from the posterior is required to compute Eq. (3.4) since it involves an expectation under $P(\theta|D_{<i})$.

In order to compute Eq. (3.4) in practice, we use Jensen's inequality and write

$$\log P(D) = \sum_{i=1}^{n} \log(E_{P(\theta|D_{<i})}[P(D_i|\theta)]) \geq \sum_{i=1}^{n} E[\log P(D_i|D_{<i})] \tag{3.5}$$

Therefore, we can approximate $L(D) = \sum_{i=1}^{n} E[\log P(D_i|D_{<i})]$ by drawing k samples of $\theta \sim P(\theta|D_{<i})$ and using the log likelihoods of our data samples for Monte Carlo estimation as

$$\hat{L}(D) = \sum_{i=1}^{n} \frac{1}{k} \sum_{j=1}^{k} \log P(D_i|\theta_i^j) \tag{3.6}$$

Appealing to our previous interpretation of $-\log P(D_i|D_{<i})$ as a loss function that we seek to minimize, we now get from Eq. (3.5) that a model with lower SoTL has higher log marginal likelihood lower bound. Note that SoTL corresponds to the negated RHS of Eq. (3.5) to interpret the relationship correctly. Following the discussion above, SoTL as an evidence lower bound holds exactly for Bayesian linear regression. For nonlinear models such as deep neural networks, we use SoTL as a theoretically inspired metric and show that it remains plausibly useful for model selection even in this setting.

This explains the Bayesian viewpoint on the connection between training speed and model selection that we alluded to previously when discussing training speed purely as an empirically successful generalization estimator.

3.2 General NAS experimental setup

In this Section, we discuss the choice of search spaces for NAS followed by an explanation of the most commonly used evaluation metrics.

3.2.1 Search spaces

In Section 2.2, we mentioned that searching for cells rather than whole networks is ubiquitous in modern NAS research. In fact, all standard NAS benchmark search spaces involve searching only for the cell design to be used for stacking. Not only has the cell-stacking design been very successful in human-designed networks (He et al., 2016; Vaswani et al., 2017), it also greatly reduces the size of the search space in NAS because we only need to search for a typically small cell compared to searching for the entire network. Figure 3.1 shows an example network from Real et al. (2019), which is composed from multiple repeating blocks of normal and reduction cells. As noted earlier, the output of most NAS algorithms is only the architecture of the cells themselves rather than the connectivity of the whole network or its stacking pattern. The macro-skeleton of the final network is usually fixed by human experts independently of the search.

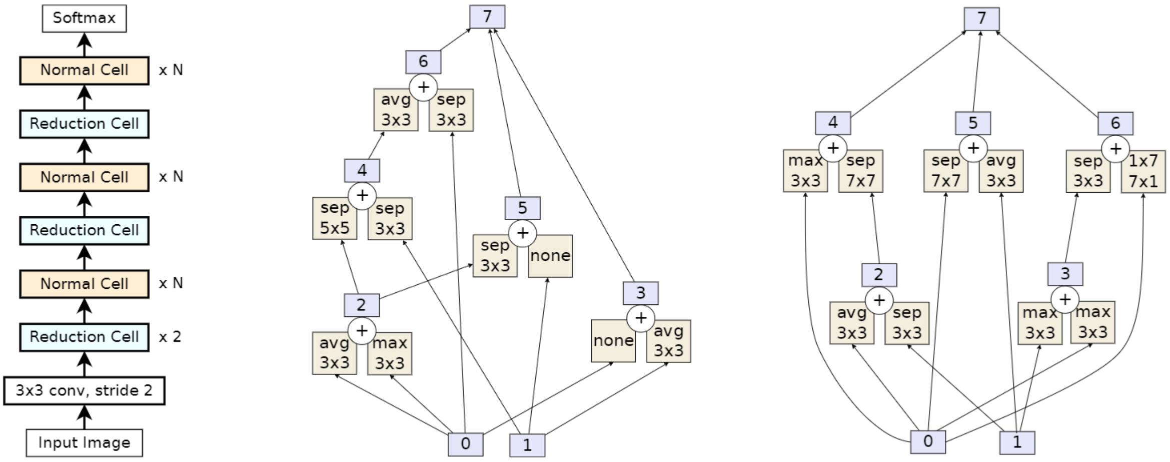


Figure 3.1: The output of a cell-based NAS algorithm are the normal (middle) and reduction (right) cells which are stacked into the final architecture (left) in a predetermined fashion. Reprinted from Real et al. (2019).

We now describe the precise form of a *cell* and each of our benchmarked search spaces in detail. A cell is simply a directed acyclic graph (DAG) consisting of several nodes, typically below 10. The graph is fully connected, meaning there always is a directed edge (i, j) between nodes x^i and x^j. Each edge is associated to an operation detailing the transformation it applies. In order to search for the topology of the cell, search spaces tend to explicitly include a *zero* operation which effectively makes an edge disappear. The input connectivity to a cell is usually fixed to involve connecting some of the first few nodes to the output of the previous cell. Likewise, all the intermediate nodes in a cell are combined through either concatenation or sum to form the output of a cell. Therefore, the general goal is to search for both the topology of the middle part of the cell and the operations for each edge. The best found cell is then stacked to form the whole network.

The most popular search space is the one introduced by DARTS (Liu et al., 2018), which was heavily inspired by the design of the NASNet search space (Zoph et al., 2018), from which we visualized an architecture in Figure 3.1. In the DARTS case specifically, each cell has two input nodes set equal to the cell outputs of the previous two layers, and the output of a cell is obtained by concatenating all the intermediate nodes. Each intermediate node output is a sum of the corresponding edge operations $o^{(i,j)}$:

$$x^j = \sum_{i<j} o^{(i,j)}(x^i) \tag{3.7}$$

The DARTS search space has eight candidate operations in total, namely *zero, skip connection,* 3 × 3 and 5 × 5 *separable convolutions,* 3 × 3 and 5 × 5 *dilated separable convolutions,* 3 × 3 *max pooling* and 3 × 3 *average pooling.* This search space is further unique in that it searches for two kinds of cells. It uses both a normal cell and a reduction cell, in which all operations adjacent to the input nodes are of stride two. This mimics human expert design of reducing the size of feature maps, which is standard in computer vision architectures (Simonyan et al., 2015; Krizhevsky et al., 2012).

The whole architecture encoding can be written as $(\alpha_{normal}, \alpha_{reduce})$ where the cell design α_{normal} is shared for all normal cells and likewise for the reduction cells. In total, each cell has 7 nodes. There are two reduction cells in each network positioned at 1/3 and 2/3 of the total depth. DARTS uses 8 layers for the search supernetwork and 20 layers for standalone architecture evaluation. While the search space specification might seem small, the combinatorial nature of constructing the cells means there are more than 10^{18} candidate architectures in total. An example normal cell found by DARTS is shown in Figure 3.2.

In order to examine the robustness of our work across search spaces, we further use smaller NAS benchmarks from the NASBench series, which we summarize in Table 3.1. Those benchmarks are multiple orders of magnitude smaller in terms of the total number of different architectures. While this might seem like a disadvantage, it means that it is possible to exhaustively evaluate each architecture in the search space, as is the case for NASBench-101 (Ying et al., 2019) and NASBench-201 (Dong et al., 2020). Those are referred to as tabular benchmarks because they tabulate the

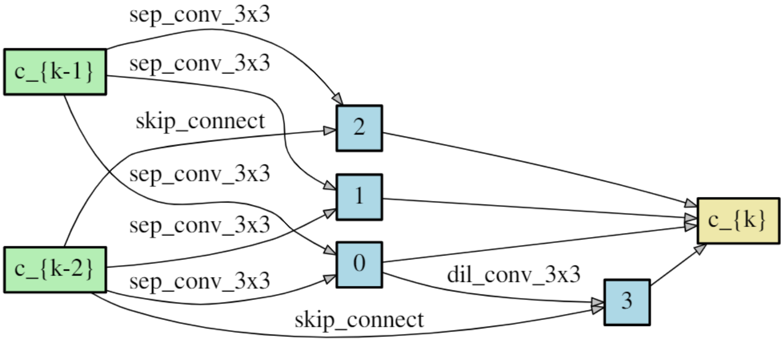


Figure 3.2: An example normal cell found by DARTS with two input nodes (green), four intermediate nodes (blue) and one output node (yellow). The labels above each edge signal the final chosen operation. c_{k-1} refers to the previous cell output. Reprinted from Liu et al. (2018).

ground truth performances of each architecture trained from scratch to convergence along with some training curve statistics, all of which can be queried via provided APIs.

We also use NASBench-301 (Siems et al., 2020), which uses a model-based predictor to predict the test set performance of architectures from the DARTS search space. Towards this purpose, the authors first trained 60k architectures to convergence on CIFAR10 and then trained models to predict the performances of all other architectures in the search space. Using NASBench-301 thus allows studying the search trajectories for the DARTS space. Note that explicitly training all architectures in the DARTS space would have been infeasible as it contains more than 10^{18} architectures.

The NASBench-201 search space is conceptually the same as the one we already described for DARTS. Each cell is again represented as a DAG, but each graph now has only 4 nodes. The operation set comprises just 5 operations, which are *zero, skip connection,* 1×1 *convolution,* 3×3 *convolution* and 3×3 *average pooling.* Only one type of cell is searched since the reduction cell architecture is fixed to be a residual block with stride 2. The schema of the whole NASBench-201 network is shown in Figure 3.3. In particular, we note that the network starts with a convolutional stem followed by a block of N searchable cells. Those are followed by the aforementioned residual block acting as a reduction cell, and the N cell blocks are stacked two more times in total. The network then finishes with a global average pooling layer.

NASBench-201 has only 15 625 cell candidates in total due to the small number of nodes and operations set. However, each architecture was trained to convergence from scratch on all of CIFAR10, CIFAR100 (Krizhevsky, 2012) and ImageNet16-120, a downsampled version of ImageNet (Deng et al., 2009). Each run was repeated for three random seeds. Both the final test set performance and training curves including training and validation losses are available to query at each epoch. Notably, this allows querying the training loss necessary for computing SoTL directly from the dataset without any further computation. It is therefore possible to evaluate the performance of SoTL on all architectures in the search space.

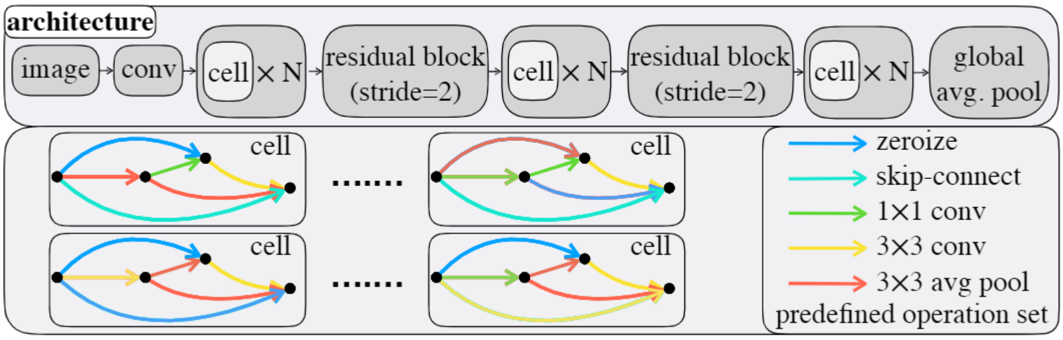


Figure 3.3: The macro skeleton (top) and cell structure (bottom) in NASBench-201. Reprinted from Dong et al. (2020).

The last benchmark we examine is the NASBench-101 (Ying et al., 2019). The search space is again conceptually similar to that of DARTS. Here, the operations set only includes three operations: 1×1 *convolution*, 3×3 *convolution* and 3×3 *max pooling*. The number of nodes V is limited to $V \leq 7$; in particular, the number of nodes is not constant as in the previous cases. Likewise, the maximum number of edges E is limited to $E \leq 9$. This gives a total of 423k unique architectures, all of which are trained from scratch on CIFAR10 using three random seeds and four different epoch budgets. The total compute time was roughly 120 TPU years. The reported training statistics include training, validation and test accuracy computed at four equally spaced out intervals during the training. Importantly, training loss is not available, and the remaining statistics are only recorded four times during the training. This makes it impossible to evaluate SoTL the same way as with NASBench-201 without retraining the architectures.

Another issue with NASBench-101 is that there is no immediate way to evaluate weight-sharing algorithms on the search space because of the changing topology caused by the variable number of nodes and edges. Zela et al. (2020) show that it is possible to restrict the original search space and construct reduced spaces, which are usable in weight-sharing algorithms. The NASBench-1shot1 (Zela et al., 2020) dataset includes three such sub-spaces of NASBench-101. The spaces referred to as Search

Search space	$N_{architectures}$	$N_{operations}$	Datasets	Ground truth
NASBench-201	15 625	5	CIFAR10/100, ImageNet16-120	Exact
DARTS (via NB301)	10^{18}	8	CIFAR10	Predicted
NB101-1	6 240	3	CIFAR10	Exact
NB101-2	29 160	3	CIFAR10	Exact
NB101-3	363 648	3	CIFAR10	Exact

Table 3.1: Overview of all our benchmarked NAS search spaces.

space 1, 2 and 3 contain 6 240, 29 160 and 363 648 architectures out of the original 423k. We will abbreviate those as NB101-1, NB101-2 and NB101-3, respectively.

While tabular benchmarks are very useful for precisely measuring the performance of NAS algorithms, they come with limitations. In particular, all architectures are trained using the same training protocol in terms of optimization parameters. Therefore, it is not obvious if the architectures that perform the best do so because they generalize the best or because the fixed training hyperparameters were the closest to optimal for those specific architectures. The same criticism can be applied to weight-sharing NAS itself because some hyperparameter settings might lead to unfair advantages for specific architectures within the supernetwork. Nonetheless, the fixed hyperparameters setup is common in NAS even outside of the NASBench series under the assumption that it leads to fair training without overfitting to some of the architectures.

3.2.2 Evaluation criteria

Several evaluation metrics are in common use for NAS. For the DARTS-family of models which only output a single candidate best architecture, the mean and standard deviation of the test set accuracy after standalone training of the selected architecture is usually reported. Unless otherwise noted, we query the final architecture performances from the API of a NASBench series benchmark as explained in Section 3.2.1. In particular, all search trajectories are constructed by querying the NASBench APIs. For the DARTS search space, we also retrain the most promising architectures from scratch following the standard DARTS evaluation protocol to make our results comparable to prior work. This is necessary because NASBench-301 offers only approximate test set accuracies for DARTS space architectures.

Furthermore, when using one-shot algorithms such as SPOS (Guo et al., 2020) or RandomNAS (Li et al., 2020), it is possible to produce a ranking for any subset of architectures in the search space rather than just outputting the best one. In those cases, we randomly sample a number of architectures from the search space and evaluate the ranking proposed by generalization estimators against the ground truth ranking based on test set accuracy queried from the APIs. As estimators, we compare SoTL against a minibatch training loss (TLMini), minibatch validation accuracy (ValAccMini) and whole validation set accuracy (ValAcc). We report Spearman correlation, which has been used in a number of prior NAS works (Dong et al., 2020; Ying et al., 2019), between the two rankings as the main quantity of interest. Other popular choice for evaluating the ranking quality is Kendall's τ (Chu et al., 2019; Yu et al., 2020c). We also report the average test set accuracy of the top-10 selected architectures. Other hyperparameters for one-shot NAS evaluation are described at the beginning of Section 4.2.

An issue when measuring correlations is that ranking distinct architectures can be difficult when their true performances are very close, which is necessarily going to be the case for large search spaces (e.g. the DARTS search space, which has 10^{18} candidate architectures). This can lead to poor correlations between the rankings proposed by search algorithms and the ground truth ranking. However, it bears no practical significance as all the misranked architectures are very close in their generalization performance. This is particularly relevant for evaluations based on NASBench-301, which uses a model-based predictor for estimating test set performance. The approximate predictions introduce another source of noise to the architecture ranking. One possible solution to those problems is to ignore rank disorders when the difference in performance is very small, which can be accomplished by using metrics such as sparse Kendall's τ (Yu et al., 2020b).

3.3 Differentiable NAS

3.3.1 DARTS

We now introduce DARTS (Liu et al., 2018) in detail as it is one of the most popular differentiable weight-sharing NAS algorithms. Next, we discuss the main challenges of incorporating SoTL into differentiable NAS and our proposed solutions. Finally, we briefly highlight the other state-of-the-art DARTS variants that we benchmark in our work to show the broad applicability of our method.

Earlier NAS algorithms such as NASNet (Zoph et al., 2018) or REA (Real et al., 2019) generally train single architectures separately, and the contribution of the NAS algorithm is in choosing which architectures to evaluate via mechanisms such as reinforcement learning or evolutionary algorithms. In contrast, DARTS trains all architectures jointly. It does so by relaxing the architectural constraints on having a single operation per edge to a softmax over the possible operations:

$$\overline{o}^{(i,j)} = \sum_{o \in O} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} \exp(\alpha_{o'}^{(i,j)})} o(x), \tag{3.8}$$

where O is the operations set and $\alpha^{(i,j)}$ are the learned softmax coefficients on an edge (i,j) with dimensionality $|O|$ to represent the architecture encoding. Intuitively, the cell DAG has multi-edges between each pair of nodes (i,j), in which each single edge represents one operation from the operations set.

At the end of training, the discrete architecture can be obtained via argmax on each edge, and then retrained separately from scratch. DARTS explicitly picks the top two strongest incoming edges at each node as the retained connections. The network with multi-edges in the cell DAG is referred to as the *supernetwork*, and a visualization of the continuous relaxation is shown in Figure 3.4. Because softmax is differentiable, we can optimize the continuous architecture parameter α by gradient descent. In order for the discretized architecture to have good performance, it is implicitly assumed that the most useful edge operation o will simultaneously have the highest weight $\alpha_o^{(i,j)}$ in the softmax at the end of optimization.

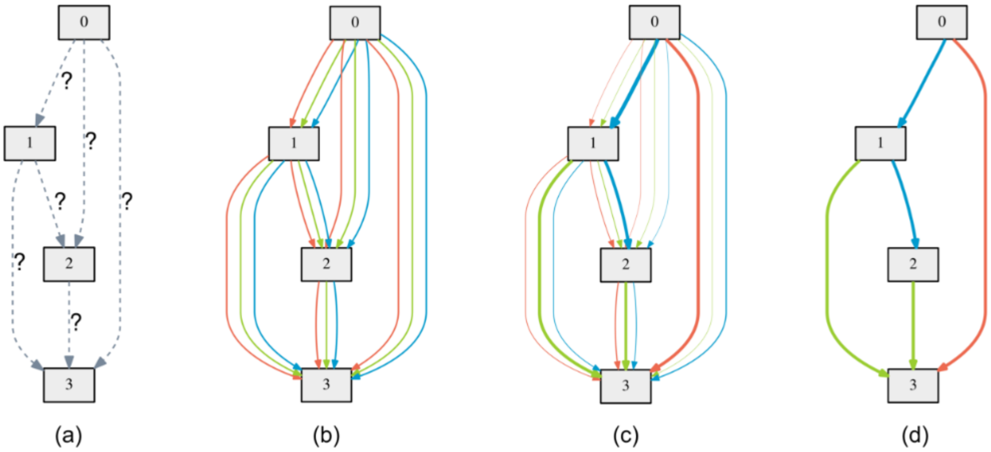


Figure 3.4: The evolution of a DARTS cell during optimization. In (a), we first initialize the network. In (b), all the edges are initialized with the same weights in the architecture softmax for fairness. (c) shows the state after training where the most useful operations on each edge are in bold. (d) displays the final discretized architecture after we drop all the insignificant edges. Reprinted from Liu et al. (2018).

DARTS formulates architecture search as a bi-level optimization problem similar similar to meta-learning (Finn et al., 2017) or gradient-based hyperparameter optimization (Luketina et al., 2016). Let L_{train} and L_{val} be losses on the train and validation set, respectively. The goal is to find the optimal architecture encoding α^* such that L_{val} is minimized at the optimal weights w^*, which are only trained on the training set. This leads to the following problem formulation:

$$\min_{\alpha} \quad L_{val}(w^*(\alpha), \alpha) \tag{3.9a}$$

$$\text{s. t.} \quad w^*(\alpha) = \text{argmin}_w L_{train}(w, \alpha) \tag{3.9b}$$

Eq. (3.9a) is referred to as the outer loop while Eq. (3.9b) is called the inner loop. The outer loop is usually optimized by gradient descent, which means descending $\nabla_\alpha L_{val}(w^*, \alpha)$ in this case. The inner loop is also typically optimized via gradient descent. However, computing the optimal weights w^* is equivalent to training the network to convergence after every outer loop architecture update, which is computationally prohibitive. A common solution (Finn et al., 2017; Luketina et al., 2016) is to approximate w^* by taking a single SGD step from the current weights w after each change in the outer loop variables. Therefore, the outer problem is equivalent to minimizing $L_{val}(w - \eta\nabla_w L_{train}(w, \alpha))$ for a given learning rate η. We shall denote $w - \eta\nabla_w L_{train}(w, \alpha)$ as $\hat{w}^*$. The iterative procedure is described in Algorithm 1.

```
Algorithm 1: DARTS - Differentiable Architecture Search
Create mixed operations $\overline{o}^{(i,j)}$ parameterized by $\alpha^{(i,j)}$ for each edge $(i, j)$
while not converged do
    1. Approximate the optimal weights $w^*$ with one step of SGD by
       computing $\hat{w}^* = w - \eta\nabla_w L_{train}(w, \alpha)$;
    2. Update architecture $\alpha$ by descending $\nabla_\alpha L_{val}(\hat{w}^*, \alpha)$;
    3. Update the original weights by descending $\nabla_w L_{train}(w, \alpha)$ using the
       new architecture encoding $\alpha$;
end
```

Notably, Step 2 in Algorithm 1 involves unrolled differentiation through the SGD update in Step 1. We can approximate the exact gradient by computing

$$\nabla_\alpha L_{val}(w^*(\alpha), \alpha) \approx \nabla_\alpha L_{val}(w - \eta\nabla_w L_{train}(w, \alpha), \alpha). \tag{3.10}$$

Applying the chain rule shows that the computing the architecture gradient necessitates second order derivatives:

$$\begin{aligned}\nabla_\alpha L_{val}(w - \eta\nabla_w L_{train}(w, \alpha), \alpha) &= \nabla_\alpha(w - \eta\nabla_w L_{train}(w, \alpha))\nabla_w L_{val}(\hat{w}^*, \alpha) + \nabla_\alpha L_{val}(\hat{w}^*, \alpha) \\ &= (-\eta\nabla^2_{\alpha,w} L_{train}(w, \alpha))\nabla_w L_{val}(\hat{w}^*, \alpha) + \nabla_\alpha L_{val}(\hat{w}^*, \alpha)\end{aligned} \tag{3.11}$$

Algorithm 2: First-order DARTS

```
Create mixed operations $\overline{o}^{(i,j)}$ parameterized by $\alpha^{(i,j)}$ for each edge $(i, j)$
while not converged do
    ~~1. Approximate the optimal weights $w^*$ with one step of SGD by computing $\hat{w}^* = w - \eta\nabla_w L_{train}(w, \alpha)$~~;
    2. Update architecture $\alpha$ by descending $\nabla_\alpha L_{val}(w, \alpha)$;
    3. Update the weights by descending $\nabla_w L_{train}(w, \alpha)$ using the new
       architecture encoding;
end
```

The main difficulty is computing the matrix-vector product $\nabla^2_{\alpha,w} L_{train}(w, \alpha))\nabla_w L_{val}(\hat{w}^*, \alpha)$. In common DARTS implementations, this is done by the method of finite differences. Let ϵ be a small scalar and $w^\pm = w \pm \epsilon\nabla_w L_{val}(\hat{w}^*, \alpha)$, then we can approximate the matrix-vector product by a central difference as:

$$\nabla^2_{\alpha,w} L_{train}(w, \alpha))\nabla_w L_{val}(\hat{w}^*, \alpha) \approx \frac{\nabla_\alpha L_{train}(w^+, \alpha) - \nabla_\alpha L_{train}(w^-, \alpha)}{2\epsilon} \tag{3.12}$$

We note that the computation above assumes that the optimal weights w^* can be obtained with a single SGD step from the current weights w, and the unrolled differentiation accordingly involves only a single Jacobian-vector product. For this reason, the DARTS training loop is sometimes referred to as one-step unrolled differentiation. Moreover, it is not necessary to use finite differences to approximate the Jacobian-vector product, and it can be computed exactly using automatic differentiation packages readily available in common deep-learning frameworks such as PyTorch (Paszke et al., 2019). Changing the gradient computation procedure will turn out to be crucial for integrating SoTL into DARTS as we discuss in Section 3.3.2.

Finally, Liu et al. (2018) have also proposed to sidestep the requirement to compute second-order derivatives by setting the unrolling learning rate $\eta = 0$, which implies that $\hat{w}^* = w$. In this setup, Step 1 in Algorithm 1 is thus unnecessary, and we get the so called first-order DARTS, in which we simply alternate between updating the architecture and the weights in a block coordinate-ascent optimization. First-order DARTS is significantly faster compute-wise due to no Jacobian-vector product computations as well as not making multiple passes over the same data as in Steps 1 and 3 of normal DARTS. However, it empirically achieves lower performance. We highlight the changes compared to normal DARTS in Algorithm 2, and we will sometimes explicitly refer to normal DARTS as the second-order DARTS to differentiate it from first-order DARTS.

3.3.2 Exact SoTL gradient

In order to integrate SoTL into DARTS, we would like the outer loop to optimize the architecture to minimize SoTL rather than L_{val}, and to do so with a fixed discretized architecture encoding that would serve as the final searched architecture. The problem formulation can be restated as:

$$\min_{\alpha} \qquad \text{SoTL} = \sum_{t=1}^{T} L_{train}(f(x_t, w_t, \alpha), y_t) \tag{3.13a}$$

where $f(x_t, w_t, \alpha)$ is the model's output using the weights w_t and architecture α. We will refer to the training trajectory over T steps in Eq. (3.13a) as the *unrolled* training history or simply the *unrolling* to directly appeal to the relationship with unrolled differentiation. Importantly, the architecture α stays fixed during the unrolling because our goal is to pick a single architecture that would perform well in the sense of having low SoTL when trained from scratch.

The principal issue with optimizing SoTL in unrolled differentiation is that calculating the exact gradients $\nabla_\alpha\text{SoTL}$ is computationally prohibitive in terms of both memory and compute time. To see this, we assume we optimize the weights by SGD updates, which means that the weights at a specified time step T optimized from initial w_0 can be written as:

$$w_T = w_0 - \eta \sum_{t=0}^{T} \nabla_w L_{train}(f(x_t, w_t, \alpha), y_t) \tag{3.14}$$

With this in mind, we will now compute the final training loss gradient after T iterations $\nabla_\alpha L_{train}(f(x_T, w_T, \alpha), y_T)$ as the first step to computing the SoTL gradients, where we abbreviate $L_{train}(f(x_T, w_T, \alpha), y_T)$ as L^T_{train}:

$$\nabla_\alpha L^T_{train} = \frac{\partial L^T_{train}}{\partial \alpha} + \frac{\partial L^T_{train}}{\partial w}\frac{\partial w_T}{\partial \alpha} \tag{3.15}$$

The $\frac{\partial L^T_{train}}{\partial \alpha}$ is sometimes referred to as the direct gradient, and the $\frac{\partial L^T_{train}}{\partial w}\frac{\partial w_T}{\partial \alpha}$ is also known as the indirect gradient or hypergradient. Now given the expression for w_T given in Eq. (3.14), we calculate

$$\begin{aligned}
\frac{\partial w_T}{\partial \alpha} &= \frac{\partial}{\partial \alpha}(w_{T-1} - \eta\frac{\partial L^{T-1}_{train}}{\partial w}) \\
&= \frac{\partial w_{T-1}}{\partial \alpha} - \eta(\frac{\partial^2 L^{T-1}_{train}}{\partial w \partial \alpha}\frac{\partial \alpha}{\partial \alpha} + \frac{\partial^2 L^{T-1}_{train}}{\partial w \partial w}\frac{\partial w_{T-1}}{\partial \alpha}) \\
&= -\eta\frac{\partial^2 L^{T-1}_{train}}{\partial w \partial \alpha} + (I - \eta\frac{\partial^2 L^{T-1}_{train}}{\partial w \partial w})\frac{\partial w_{T-1}}{\partial \alpha}
\end{aligned} \tag{3.16}$$

The $\frac{\partial w_{T-1}}{\partial \alpha}$ at the end of Eq. (3.16) gives a recurrent relation that can be further expanded to give

$$\begin{aligned}\frac{\partial w_T}{\partial \alpha} &= -\eta \frac{\partial^2 L_{train}^{T-1}}{\partial w \partial \alpha} + (I - \eta \frac{\partial^2 L_{train}^{T-1}}{\partial w \partial w})(-\eta \frac{\partial^2 L_{train}^{T-2}}{\partial w \partial \alpha} + (I - \eta \frac{\partial^2 L_{train}^{T-2}}{\partial w \partial w})\frac{\partial w_{T-2}}{\partial \alpha}) \\ &= -\eta \frac{\partial^2 L_{train}^{T-1}}{\partial w \partial \alpha} - \eta (I - \eta \frac{\partial^2 L_{train}^{T-1}}{\partial w \partial w}) \frac{\partial^2 L_{train}^{T-2}}{\partial w \partial \alpha} + \prod_{0 \leq k < 2} [I - \eta \frac{\partial^2 L_{train}^{T-k-1}}{\partial w \partial w}] \frac{\partial w_{T-2}}{\partial \alpha}\end{aligned} \tag{3.17}$$

If we unroll the whole history until w_0, for which it holds $\frac{\partial w_0}{\partial \alpha} = \mathbf{0}$, we get a string of summands that can be summarized as

$$\frac{\partial w_T}{\partial \alpha} = -\eta \sum_{0 \leq j \leq T} ([\prod_{0 \leq k < j} I - \eta \frac{\partial^2 L_{train}^{T-k-1}}{\partial w \partial w}] \frac{\partial^2 L_{train}^{T-j-1}}{\partial w \partial \alpha}) \tag{3.18}$$

In total, we have that Eq. (3.15) is equivalent to

$$\nabla_\alpha L_{train}^T = \frac{\partial L_{train}^T}{\partial \alpha} + \frac{\partial L_{train}^T}{\partial w}(-\eta \sum_{0 \leq j \leq T} ([\prod_{0 \leq k < j} I - \eta \frac{\partial^2 L_{train}^{T-k-1}}{\partial w \partial w}] \frac{\partial^2 L_{train}^{T-j-1}}{\partial w \partial \alpha})) \tag{3.19}$$

This is the exact unrolled differentiation hypergradient using the last training loss. The SoTL gradient computed over T steps of training is simply equal to a sum of the individual training loss gradients:

$$\nabla_\alpha SoTL = \sum_{t=0}^{T} (\frac{\partial L_{train}^t}{\partial \alpha} + \frac{\partial L_{train}^t}{\partial w}(-\eta \sum_{0 \leq j \leq t} ([\prod_{0 \leq k < j} I - \eta \frac{\partial^2 L_{train}^{t-k-1}}{\partial w \partial w}] \frac{\partial^2 L_{train}^{t-j-1}}{\partial w \partial \alpha}))) \tag{3.20}$$

Computing the gradient in this way is infeasible for DARTS primarily due to excessive memory requirements when using reverse-mode auto-differentiation as in PyTorch (Baydin et al., 2018a). The formula in Eq. (3.19) requires access to the weights w_t at all time steps, which means they must be stored with $O(T)$ space complexity. However, DARTS is already strongly memory constrained, and the search model size is artificially kept low to fit into single GPU memory. This is known to be a bottleneck since the end goal is to search for large models (Liu et al., 2018; Xu et al., 2019). Hence decreasing the model size in order to be able to do more steps of unrolling might even lead to worse results overall. Moreover, training with longer unrolling T is much slower since Eq. (3.19) also has a nested sum-product giving $O(T^2)$ Jacobian-vector products that must be calculated.

3.3.3 Approximate SoTL gradient

Due to the infeasibility of exact unrolled differentiation, we propose an approximation scheme for the SoTL gradient. Specifically, we use a first-order approximation equivalent to only considering the dependence of w_t on parameters at time step w_{t-1} for all the gradients in the sum. This is similar to how first-order DARTS only calculates the architecture gradients with respect to the final unrolled weights, which avoids the second-order derivatives that come in when computing gradients with respect to the initial weights. The difference in our approach is that we optimize SoTL rather than a single validation loss. Effectively, we only consider the direct gradient $\frac{\partial L^T_{train}}{\partial \alpha}$ out of the exact formula in Equation (3.19), giving:

$$\nabla_\alpha L^t_{train} \approx \frac{\partial L^t_{train}}{\partial \alpha} \tag{3.21}$$

The whole SoTL gradient can therefore be written as

$$\nabla_\alpha SoTL = \nabla_\alpha (L^0_{train} + L^1_{train} + .. + L^T_{train}) \approx \sum_{t=1}^{T} \frac{\partial L^t_{train}}{\partial \alpha} \tag{3.22}$$

This approximation is equivalent to summing first-order architecture gradients over an unrolled training history, which has several advantages. Most importantly, it does not require storing any extra weights parameters apart from the latest weights at each time step because every summand in the gradient can be computed alongside the weights gradients in backward passes. In fact, no additional computation on top of normal training is needed since the backward passes for training the weights can also compute the architecture gradients at virtually no extra cost. It is thus possible to compute the SoTL gradients over arbitrarily long training history with no concern for either space or time complexity.

Moreover, we optimize the SoTL objective in Eq. (3.13a) iteratively for computational reasons. Rather than retraining the model from scratch after every architecture update as would be necessary for computing the entire SoTL, we run training only over a short period to compute the SoTL gradient and keep the weights from the last iteration of the algorithm as the initialization for the next iteration together with updated architecture parameters. The rest of our algorithm proceeds the same as in second-order DARTS.

In this work, we usually compute the approximate SoTL gradients over $T = 100$ timesteps. Since the architecture gradients are always accumulated over 100 steps before being used to update the architecture, our proposed algorithm does 100x fewer architecture updates than the original DARTS for the same number of epochs. However, the SoTL gradient is a sum (rather than an average) of gradients over 100 hundred loss terms, so its magnitude is approximately 100 times larger than normal to naturally compensate for the less frequent updates compared to baseline DARTS. We summarize the new training procedure in Algorithm 3 and show an example PyTorch code in Appendix A to highlight the ease of implementation.

Algorithm 3: SoTL-DARTS

```
Create mixed operations $\overline{o}^{(i,j)}$ parameterized by $\alpha^{(i,j)}$ for each edge $(i, j)$
Set T=100 steps of unrolling for computing SoTL
while not converged do
    1. Approximate the optimal weights $w^*$ with T steps of SGD by
       computing $w_T = w_0 - \eta \sum_{t=0}^{T} \nabla_w L_{train}(f(x_t, w_t, \alpha), y_t)$;
    2. Update architecture $\alpha$ by descending SoTL gradient via Eq. (3.22);
    3. Update the original weights with T steps of SGD
       $w_T = w_0 - \eta \sum_{t=0}^{T} \nabla_w L_{train}(f(x_t, w_t, \alpha), y_t)$ using the new architecture
       encoding;
end
```

3.3.4 DARTS variants

In order to assess the robustness of SoTL gradients, we integrate our estimator into several more recent DARTS variants, namely DrNAS (Chen et al., 2021) and PDARTS (Chen et al., 2019). In general, the adaptation of DARTS-like algorithms is always virtually the same as outlined in SoTL-DARTS (Algorithm 3). Most DARTS-like algorithms only slightly modify the forward pass by using a different parameterization of the network. That is, they only modify the w_t and α while the loss L and datapoints (x_t, y_t) remain the same when considered in the SoTL formula

$$SoTL = \sum_{t=1}^{T} L_{train}(f(x_t, w_t, \alpha), y_t). \tag{3.23}$$

However, when using software with automatic differentiation support such as PyTorch (Paszke et al., 2019), the required gradients are computed automatically, and we simply aggregate them over T steps as in the normal SoTL-DARTS. The model is used as a black-box for the SoTL gradient calculation, and hence there is no change in the SoTL implementation regardless of how the DARTS search supernetwork is parameterized. This highlights that our approximate SoTL gradient is very simple to integrate on top of existing codebases.

We summarize the main idea behind each of the DARTS variants that we benchmark:

- **DrNAS** (Chen et al., 2021) reformulates the operation mixing weights α as a Dirichlet distribution learning problem rather than viewing them mechanically as softmax coefficients. The αs are treated as random variable samples from Dirichlet $q(\alpha|\beta)$, and we now optimize the distributional parameters β via pathwise derivatives.
- **PDARTS** (Chen et al., 2019) proposes to alleviate a problem known as the *depth gap*, which refers to the different number of layers used in DARTS for the search and evaluation phases. The lower search depth is necessary to prevent DARTS from taking too much memory. In PDARTS, the number of layers during the search becomes larger over time while the least important operation edges are simultaneously deleted from the search space. The memory consumption is thus kept constant even as the network grows.

3.4 One-shot NAS

While DARTS trains the whole supernetwork including all candidate architectures at once, other weight-sharing algorithms such as ENAS (Pham et al., 2018) or SPOS (Guo et al., 2020) sample single discrete architectures and only optimize the weights that belong to the chosen subnetwork in each iteration. This leads to $|O|$-times less memory usage since we only keep one edge from the DARTS multi-edge graph for each forward/backward pass. ENAS uses a reinforcement learning controller which tries to learn to sample the best performing architectures so that they are trained the most frequently. This assumes that the architecture subnetworks which have the best standalone performance also have the best performance with the shared weights from the supernetwork. Such an assumption is similar to how DARTS implicitly relies on the most useful edges also having the highest softmax weights.

Bender et al. (2018) show that the greedy sampling in algorithms such as ENAS is not necessary, and it suffices to train randomly sampled architectures at each step in order to attain good results. After the supernetwork finishes training, the best architectures are extracted by sampling N architectures and evaluating their validation accuracy using the shared supernetwork weights. The total cost of evaluating an architecture is therefore equal to one mini-batch forward pass. This kind of training protocol is often referred to as *one-shot* NAS. A randomly sampled architecture cell is shown in Figure 3.5, which is referred to as a *choice block* since it involves repeatedly choosing an operation edge in the cell DAG.

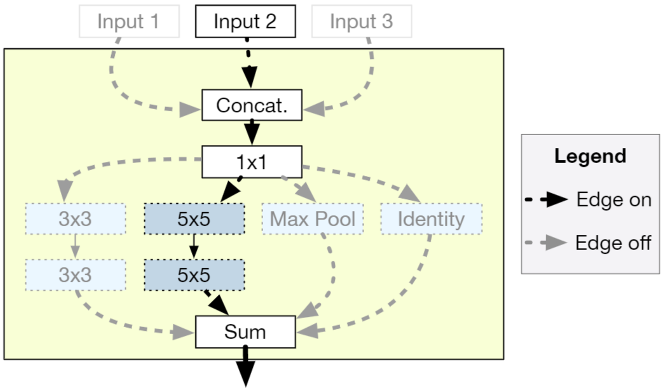


Figure 3.5: An example of a sampled architecture in one-shot NAS. Only the opaque edges were sampled to be kept while the grayed out edges are zeroed out, which effectively discretizes the supernetwork into a single architecture subnetwork. Reprinted from Bender et al., 2018.

Bender et al. (2018) also include a carefully tuned path Dropout (which we will refer to as *DropPath*), which drops out a percentage of incoming paths to a node. At the start of the training, the DropPath probability is kept low, which means that almost the whole supernetwork is kept active at each iteration. However, the dropout rate is increased over time to make the network ready for the eventual discretization, preventing excessive weight co-adaptation. The co-adaptation issue is caused by training multiple architectures at once, which then learn to rely on each other's weights that stop being available after the supernetwork is discretized into a single architecture. This issue is also present in DARTS. Furthermore, the memory requirements in this setup are still high as most of the supernetwork might be kept active in the early parts of training. Li et al. (2020) instead only ever sample individual architectures without any DropPath schedule. Even this setting is shown to have competitive results on the DARTS search space despite being conceptually very simple, and it is more stable in practice because it further reduces weight co-adaptation. It is also very efficient in terms of compute and memory since the forward/backward pass only updates a single architecture's weights at any point during the training.

3.4.1 Integrating SoTL into one-shot NAS

In this section, we discuss two ways in which we integrate SoTL into one-shot NAS, both by using it as a high fidelity generalization estimator and by highlighting a connection between SoTL and meta-learning.

First, we reiterate on architecture selection in one-shot NAS. The most common way to extract the best candidate architecture out of a one-shot supernetwork is to evaluate some moderately large subset (typically a few hundred or thousands) of architecture subnetworks by computing their validation accuracy on a minibatch. The architectures chosen for evaluation can be sampled randomly (Li et al., 2020), by evolutionary algorithms (Guo et al., 2020) or reinforcement learning (Pham et al., 2018). In order to use SoTL, we replace evaluating the selected architectures by their validation accuracy and instead shortly train (e.g. for 100 minibatches) the architectures separately while using the supernetwork weights as initialization. This makes it possible to calculate SoTL, which cannot be done without additional training. We then rank the sampled architectures by SoTL and pick the best. Using the supernetwork initialization makes each architecture immediately reach the early-to-mid phase of training, which leads to large compute savings compared to training architectures completely from scratch. In Section 4.2, we show that even this short amount of training is very effective in combination with SoTL. Moreover, we also show that using SoTL this way is often computationally faster than computing the validation accuracy using the whole validation set (which might be composed of a similar or higher amount of minibatches than 100) and significantly more accurate than using just one validation minibatch. Hence search algorithms usually take less time to reach a threshold accuracy using SoTL than if they relied on validation metrics even though the SoTL computation itself comes with additional training.

Second, we propose to reformulate one-shot NAS as a meta-learning problem and optimize the supernetwork with gradient-based meta-learning algorithms (Finn et al., 2017; Nichol et al., 2018; Zhou et al., 2019). We first introduce the general meta-learning setup, which can be understood as training a model that can quickly adapt to a new task using only a few training iterations. This is often referred to as few-shot learning (Finn et al., 2017). The meta-learned weights of the model in question are known as the meta-weights. An example task in few-shot image classification might be to achieve high accuracy on a new image class such as dogs after only training on class samples of other animals, such as cats. The meta-classifier has to learn features that generalize well for them to transfer across different tasks.

Next, we note that there are several similarities between meta-learning and one-shot NAS with SoTL. In line with the first integration of SoTL we proposed above, we would like to be able to train each sampled subnetwork architecture from the one-shot supernetwork for only a small amount of minibatches before it reaches its full performance, which would allow us to rank it accurately. We can understand the supernetwork as meta-weights since it provides initialization to the sampled architecture subnetworks the same way that the meta-weights are used as initialization for each task in meta-learning. The architecture subnetworks themselves can be understood as the tasks since we aim for rapid few-shot adaptation of each architecture. Therefore, the one-shot NAS setup is analogous to the more general meta-learning setup outlined above. Moreover, in large enough search spaces such as the DARTS search space, we are unlikely to ever randomly sample the same architecture twice, which shows that the supernetwork must learn features that generalize across architectures (i.e. tasks) for one-shot NAS to work at all.

Furthermore, first-order meta-learning algorithms such as Reptile (Nichol et al., 2018) or MetaMinibatchProx (Zhou et al., 2019) implicitly use the same approximate SoTL gradients as our work, which we now show. While the seminal work on MAML (Finn et al., 2017) used a similar bi-level optimization scheme as DARTS with a train-validation split, the first-order algorithms only use a training split. The Reptile gradient for updating the meta weights is set to be the direction from the meta-weights to the post-adaptation weights, ie. $\nabla_w^{Reptile} = \hat{\theta}^* - \theta$. The whole algorithm is shown in Algorithm 4.

Algorithm 4: Reptile in One-shot NAS

```
Initialize one-shot supernetwork;
Set T (e.g. 4) inner loop adaptation steps;
while not converged do
    1. Sample task (ie. subnetwork architecture) and a minibatch of training data;
    2. Update inherited supernetwork weights $\theta$ to $\hat{\theta}^*$ by T optimizer steps on the sampled minibatch as within-task adaptation;
    3. Update the original supernetwork weights $\theta \leftarrow \theta + \eta_{outer}(\hat{\theta}^* - \theta)$
end
```

Intuitively, Reptile first trains the weights on the target task and then moves the meta-weights in the direction of the final weights. This is equivalent to descending our proposed approximate SoTL gradient when training with SGD. To see this, consider the within-task SGD update $w_{t+1} = w_t + \eta_{inner} \nabla_w L^t_{train}$. The weights after T steps can be expressed as

$$w_T = w_0 + \eta_{inner} \sum_{t=0}^{T-1} \nabla_w L^t_{train} \tag{3.24}$$

The Reptile gradient in this setting is equivalent to $w_T - w_0 = \eta_{inner} \sum_{t=0}^{T-1} \nabla_w L_{train}^t$, which is the same as the approximate SoTL gradient from Equation (3.22) up to rescaling by the learning rate η_{inner}.

If we consider the supernetwork itself to be the meta-weights as discussed previously, then training the one-shot supernetwork using Reptile should also minimize the subsequent SoTL during finetuning of individual architectures. In fact, if we view one-shot NAS as meta-learning, it should be possible to use arbitrary meta-learning algorithms to effectively optimize the supernetwork. We demonstrate that this is possible and matches the performance of various NAS-specific state-of-the-art one-shot training protocols in Section 4.2.

3.4.2 One-shot NAS variants

Apart from the basic Single-Path One-Shot algorithm (Guo et al., 2020; Li et al., 2020), we also benchmark several proposed variants to demonstrate the wide applicability of SoTL. Specifically, we test the following algorithms:

- **FairNAS** (Chu et al., 2019) averages gradients after first sampling a subset of architectures that utilizes all the operations from the operations set on each edge. The total amount of architectures sampled at each step is thus equivalent to the number of operations in the search space since each sample reduces the number of remaining operations by one. For the DARTS search space, we first sample the topology of the cell before sampling the operations on each edge.
- **MultiPath** (Yu et al., 2019) averages gradients from several randomly sampled architectures before doing a weight update. We sample four architectures at a time.

Among meta-learning algorithms, we test Reptile, which we described above, and MetaMinibatchProx (which we will refer to as Metaprox for brevity):

- **MetaMinibatchProx** (Zhou et al., 2019) is very similar to Reptile except that the within-task optimization has the addition of proximal regularization. The loss is modified to also include an L2 penalty term $\frac{\lambda}{2}||\hat{\theta}^* - \theta||_2$ for some coefficient λ. Intuitively, this penalizes task solutions that move far away from the meta-weight initialization. We use $\lambda = 10$.

For both Reptile and Metaprox, we use four steps for the within-task adaptation. Furthermore, we note that if we understand sampling architectures as sampling tasks for meta-learning purposes, algorithms such as FairNAS or MultiPath can be interpreted as increasing the meta-batch size. The meta-batch size is simply the number of tasks sampled at each iteration of meta-learning training. Then because of the correspondence between architectures and tasks, sampling four architectures in MultiPath is equivalent to sampling four tasks at a time. Therefore, it is possible to combine Reptile with MultiPath to get Reptile with four within-task steps and meta-batch size equal to four. From this point of view, both FairNAS and MultiPath can also be understood as Reptile that only does one inner step with variable meta-batch size. However, combining MultiPath and Reptile as in the previous example would multiply the compute required 16× compared to normal SPOS, and we use a meta-batch size of one for our experiments with meta-learning algorithms due to computational reasons. We only try to combine MultiPath and Metaprox on NASBench-1shot1 and show that its performance exceeds other state-of-the-art algorithms. Investigating other ways to combine meta-learning and one-shot NAS is an exciting direction for future work.

Chapter 4

Results

4.1 Synthetic benchmarks

In order to check the basic applicability of SoTL in gradient-based optimization, we design several simple benchmarks to compare it against optimizing the validation loss. First, we reproduce experiments proposed by Baydin et al. (2018b) for online adjusting of the learning rate as a special case of gradient-based hyperparameter optimization. Next, we devise a feature selection experiment using a weight-sharing version of linear regression that serves as a toy NAS problem.

4.1.1 Gradient-based hyperparameter optimization

We showcase the broad utility of SoTL gradients by reproducing online learning rate finetuning experiments proposed by Baydin et al. (2018b). While the SoTL gradients in this setup lack the interpretation as model selection because training hyperparameters are generally not considered to be a part of the model, it is still possible to show that unrolled differentiation over training losses can exceed the performance of optimizing validation loss. Most gradient-based hyperparameter tuning literature focuses on one-step unrolled differentiation using the same bi-level formulation with train-validation split as in DARTS (Luketina et al., 2016; Baydin et al., 2018b). Some other prior work tries to optimize the hyperparameters over very long unrollings (Maclaurin et al., 2015; Fu et al., 2016). In fact, the challenges related to gradient-based hyperparameter optimization are exactly the same as in DARTS, where we can interpret the architecture parameters as a special kind of hyperparameters. Hence, improvements in the gradient computation in either problem are likely to transfer to the other.

Baydin et al. (2018b) propose to finetune the optimizer's learning rate online by treating it as a differentiable parameter rather than fixed or manually changed using a scheduler as in normal training. Assuming SGD update to weights θ_t with an objective function f and learning rate η, we have

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta f(\theta_{t-1}) \tag{4.1}$$

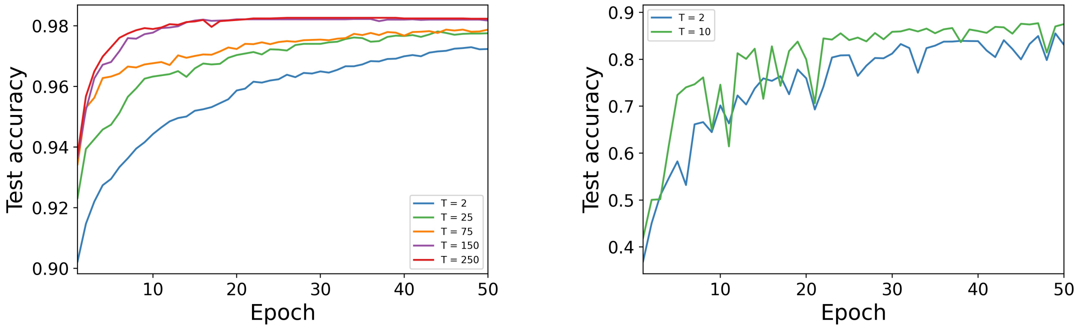
Because the update with respect to the learning rate is differentiable, we can compute $\frac{\partial \theta_t}{\partial \eta}$ and update the learning rate with SGD. The gradient can be expressed by applying the chain rule as

$$\frac{\partial f(\theta_{t-1})}{\partial \eta} = \nabla_\theta f(\theta_{t-1}) \cdot \frac{\partial(\theta_{t-2} - \eta \nabla_\theta f(\theta_{t-2}))}{\partial \eta} = \nabla_\theta f(\theta_{t-1}) \cdot (-\nabla f_\theta(\theta_{t-2})) \tag{4.2}$$

Using the gradient for the learning rate, we can apply SGD or any other gradient-based optimizer to update it. In practice, the gradient in Eq. (4.2) would not be computed by hand but instead obtained automatically using automatic differentiation packages the same way that the weights themselves are trained. Therefore, it is also trivially possible to finetune other optimization hyperparameters that are only used in a differentiable fashion during the training, such as momentum or L2 weight decay coefficient. However, we only finetune the learning rate in this Section as a proof of concept to demonstrate the applicability of SoTL.

We note that the gradient computation in Eq. (4.2) assumes one-step unrolled differentiation so that $\frac{\partial \theta_{t-2}}{\eta} = 0$. This means that we only differentiate through two steps of the training history. We relax this assumption in order to compute SoTL gradients over significantly longer unrollings. We again denote the length of unrolling as T, which corresponds to the number of training losses involved in SoTL. The rest of our algorithm proceeds as in second-order DARTS (Algorithm 1), where we can imagine the learning rate η to be the architecture α. Furthermore, Baydin et al. (2018b) already compute the learning rate hypergradients with respect to the training data rather than validation data in contrast to DARTS and other gradient-based hyperparameter tuning algorithms (Liu et al., 2018; Luketina et al., 2016; Maclaurin et al., 2015; Finn et al., 2017). Their setup is thus roughly equivalent to SoTL gradients with $T = 2$. Our contribution is in showing that having higher T leads to an even stronger performance.

Our main experiments involve training an MLP on MNIST with one hidden layer using 1000 units with 1.8M parameters total, and a VGGNet (Simonyan et al., 2015) on CIFAR10 with 138M parameters total. We vary T from 2 up to 250 for computing SoTL. Because the models involved here are relatively small, we compute the SoTL gradient exactly using automatic differentiation even over long unrollings.



(a) MLP on MNIST

(b) VGGNet on CIFAR10

Figure 4.1: a) Test set accuracy of an MLP with one hidden layer on MNIST during training while simultaneously finetuning the learning rate. Optimizing SoTL increases generalization performance. b) Training a VGG Net on CIFAR10. Because the model parameter count is large, we can only benchmark the unrolling up to $T = 10$, which still increases performance compared to $T = 2$.

For all experiments, we initialize the weights learning rate to 1e-3. The weights learning rate is intentionally set to a lower value than in human-expert derived training protocols to highlight the learning capability of benchmarked algorithms. We also fix the architecture learning rate to 1e-2. We clip the architecture gradients norm to 10 as we found the training to be unstable especially in the first few epochs, for which the large architecture steps led to divergence. Moreover, the architecture updates might make the learning rate negative. In those cases, we override the update and instead set $\eta_{t+1} = \eta_t/2$. This situation happens particularly often in later stages of the training when it is no longer possible to improve the performance by keeping learning rates high.

Figure 4.1a shows the test accuracy of the MLP on MNIST during training for various values of T up to 250. Higher T markedly improves the convergence speed and final test accuracy, and the performance improvement keeps scaling even towards the highest T. Setting $T = 150$ or $T = 250$ allows the training to reach 98.3% test accuracy within 20 epochs, whereas using $T = 2$ only reaches 96.4% after 50 epochs. Figure 4.1b shows training of a VGGNet on CIFAR10. Because VGGNet is much larger in terms of parameters, we were only able to evaluate unrolling of length $T = 10$. Nonetheless, there is a clear advantage in doing so, and it leads to better test set accuracy again. It appears that higher T alleviates the short-horizon bias in gradient-based hyperparameter optimization (Wu et al., 2018), which makes the learning rate decrease too quickly for lower T and slows down training.

4.1.2 Weight-sharing linear model

The linear model feature selection follows the setup from Lyle et al. (2020) with the addition of weight-sharing to make it reminiscent of a small-scale NAS problem. The general problem definition is as follows. We construct a synthetic dataset $(\mathbf{X}, \mathbf{Y})$ with feature vectors $\mathbf{x}_i \in R^D$ where $y_i = w_1 x_1 + w_2 x_2 + .. + w_k x_k = \mathbf{w}^\mathbf{T}\mathbf{x}_i$ and $\mathbf{w} = \{w_1, w_2, .., w_k, 0, 0, .., 0\} \in R^D$. In other words, only the first k features from the feature vector $\mathbf{x}_i$ determine the values of y while the remaining $D - k$ features are noise since the corresponding weights are zeroed out. The goal is to find the true value of k. This can be seen as a form of architecture search if we consider each possible value of k as a different architecture. More generally, this problem can also be understood as feature or model selection.

Lyle et al. (2020) have already shown that by retraining the linear model separately for each value of k, it is possible to correctly recognize the ground truth model since it will have the lowest SoTL. This directly appeals to the theoretical motivation of SoTL from Section 3.1, in which it was shown that SoTL in the linear model case directly corresponds to an evidence lower bound. This lower bound can then be applied in Bayesian model selection. In order to make the model selection be gradient-based, we instead propose a continuous relaxation similar to DARTS.

First, we introduce the concrete linear model example used in this experiment. The data points $\mathbf{x}_i$ will be the first D Fourier series terms written out in the sine-cosine form with a separate dimension for each period length. Precisely, a data point is formed by performing feature expansion from scalar x_i to a high-dimensional vector $\mathbf{x}_i$ as

$$\mathbf{x}_i = \{a_0, cos(2\pi x_i) + sin(2\pi x_i), cos(2\pi 2x_i) + sin(2\pi 2x_i), .., cos(2\pi D x_i) + sin(2\pi D x_i)\} \in R^D \tag{4.3}$$

The targets y_i are set equal to

$$y_i = a_0 + \sum_{j=1}^{k}(a_j cos(2\pi j x_j) + b_j sin(2\pi j x_j)) = a_0 + \sum_{j=1}^{k}(cos(2\pi j x_j) + sin(2\pi j x_j)), \tag{4.4}$$

where we set all the ground truth Fourier series coefficients a_i and b_i to 1, and also $k < D$. The model $f(x_i) = \mathbf{w}^\mathbf{T}\mathbf{x}_i$ is just a linear regression with learnable weights $\mathbf{w}$. The goal is to properly identify the true Fourier series degree k in the presence of D features, where the $D - k$ highest level terms are noise since they are independent of the target values in Eq. (4.4).

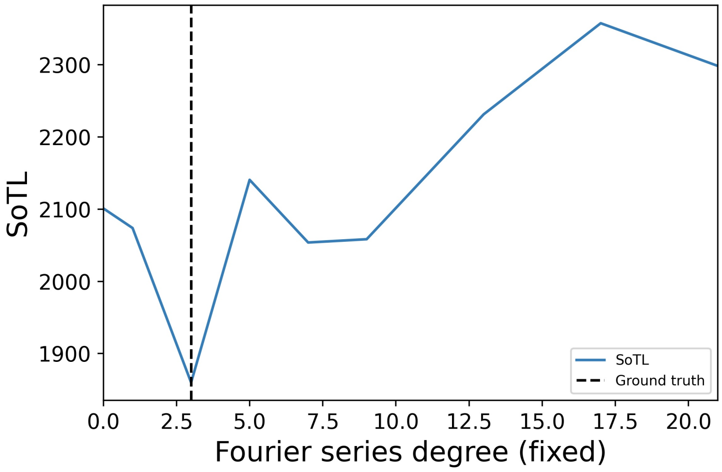


Figure 4.2: Training a linear model on the synthetic Fourier data with various fixed values of k, the maximum Fourier series degree. The model with $k = 3$, which matches the ground truth, trains the fastest and achieves the lowest SoTL.

We generate a synthetic regression dataset by randomly sampling 10 000 points in the range $[-10, 10]$. Those points play the role of scalar x_i in Eq. (4.3), which we then expand to the full feature vectors and compute the target values as in Eq. (4.4) using a ground truth Fourier series degree of $k = 3$. In order to sanity-check our approach, we first train linear models with fixed Fourier series degree similar to the example by Lyle et al. (2020). We set $k_{fixed} \in \{0, 1, 3, 5, .., 21\}$ as the maximum Fourier term degree to be used for training and zero out the weights for the remaining terms. Figure 4.2 shows that the linear model with $k_{fixed} = 3$ achieves the lowest SoTL while other models train slower because they either underfit or overfit the training data. Therefore, model selection via SoTL would succeed in picking out the ground truth model.

It is not immediately possible to search for the ground-truth Fourier series degree k with gradient descent because the sum in Eq. (4.4) is discrete. In order to use gradient-based methods, we reformulate the problem as a differentiable ordered feature selection. Our base model is still a linear regression, but we add a differentiable parameter $MaxDeg$ that will represent the architecture. $MaxDeg$ approximates the maximum selected degree of Fourier features, i.e. the upper range of the summation k. The new model output $f(x)$ is a variant of linear regression, and it is made equal to

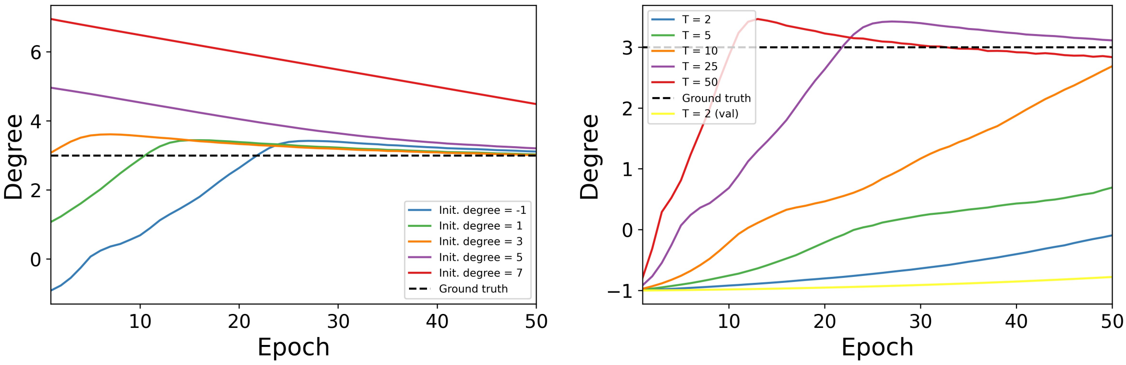
$$f(x_i) = w_0 \cdot f_\sigma(MaxDeg - 0) \cdot a_0 + \sum_{j=1}^{D} (w_j \cdot f_\sigma(MaxDeg - d_j) \cdot (cos(2\pi j x_j) + sin(2\pi j x_j))) \tag{4.5}$$

where f_σ is a sigmoid function, for which we specifically use $f_\sigma(x) = (tanh(x) + 1)/2$. The d_j are constants $\mathbf{d} = (0, 1, 2, 3, ..., D)$ representing the *degree* of a feature, which is determined by the corresponding summation index j in Eq. (4.5), hence $d_j = j$.

We first discuss the desiderata of the function f_σ, and then show how our choice of the function fulfills them. The goal is to make the feature selection be *ordered* so that $MaxDeg$ can actually be interpreted as the maximum degree of a Fourier series. If it was just regular feature selection, it would be possible to pick terms from the Fourier series of arbitrary degree. For example, the algorithm might select $sin(x), sin(2x), sin(4x)$ without picking $sin(3x)$, which does not respect the order of terms in the Fourier series sum.

Even in the ordered feature selection setting, the true model is still realizable because the ground truth data was generated in the ordered fashion as indicated by Eq. (4.4). Hence (i) we would like $f_\sigma(MaxDeg - d_j) \approx 1$ if $MaxDeg - d_j \geq 0$ and $f_\sigma(MaxDeg - d_j) \approx 0$ if $MaxDeg - d_j < 0$ so that all terms with $d_j > MaxDeg$ do not affect the model output in Eq. (4.5). Remember that d_j represents the Fourier degree of a feature. f_σ thus behaves like a step function centered around the value of $MaxDeg$, which can be continuously approximated with a sigmoid function. Next, (ii) it is desirable for the training of *MaxDeg* to be influenced only by the terms with the degree closest to *MaxDeg*. Otherwise, it would be viable to increase *MaxDeg* for the purpose of picking some very high-order term while actually not being interested in the preceding terms. This is undesirable for *ordered* feature selection.

Choosing $f_\sigma(x) = (tanh(x) + 1)/2$ fulfills both (i) and (ii). Because $tanh$ is a sigmoid function, it approximates a step function as required by (i). We also shift the range of $tanh$ from $(-1, 1)$ to $(0, 1)$ by using $(tanh + 1)/2$ rather than just $tanh$. For (ii), note that the $f_\sigma(MaxDeg - d_j)$ is a multiplicative term that multiplies weights w_j by a near-zero term for $d_j > MaxDeg$ since $f_\sigma(x) \approx 0$ when $x < 0$. Therefore, the weights w_j that correspond to Fourier series terms that are of higher degree than the parameter $MaxDeg$ are attenuated to near zero and have almost no influence on the predictions. In turn, the gradients corresponding to those weights are almost zero as well. For the low-order Fourier series terms, we instead have $f_\sigma(x) \approx 1$ when $x \geq 0$, so those weights are fully used. Even with the low-order terms, the parameter $MaxDeg$ is still trained almost exclusively in relation to the weights $w_{floor(MaxDeg)}$



(a) Fixed T = 25

(b) Degree initialization = -1

Figure 4.3: a) Training with SoTL using fixed T = 25. Regardless of whether the Fourier series degree initialization is too high or too low, it still converges to the ground truth value. b) SoTL has higher speed of convergence for higher T after fixing the initial degree to -1. This degree corresponds to including zero Fourier series terms (not even the constant).

and $w_{floor(MaxDeg)+1}$ because the gradients for all other weights are highly saturated as a result of using a sigmoid function. The degrees $d_{floor(MaxDeg)}$ and $d_{floor(MaxDeg)+1}$ are the only parameters close to the center of the sigmoid $f_\sigma(MaxDeg - d)$, and thus their corresponding weights are the only ones with high gradients. Therefore, the property (ii) is also fulfilled.

The experiment can be interpreted as ordered feature selection because $MaxDeg$ will only increase if the next closest unselected feature results in gradients that increase $MaxDeg$. The feature selection happens at discretization, where we choose the first n terms with degree $d_j \leq MaxDeg$. Ideally, $MaxDeg$ at the end of training should be close to the ground-truth Fourier series degree k so that $n \approx k$ and we only select the terms which were used for computing the target values in Eq. (4.4). The value of $MaxDeg$ is prevented from ever rising too high by setting L2 weight decay on the architecture parameter $MaxDeg$, which is equivalent to a Gaussian prior over the Fourier series degree (Murphy, 2012). This sets up the problem as Bayesian linear regression that we discussed in Section 3.1. It is necessary to penalize excessively complex models as it would otherwise be possible to set $MaxDeg$ very high and simply set all the high-order terms weights equal to zero.

Figure 4.3 shows the result of gradient-based architecture selection in the same experimental setup as before with ground truth degree $k = 3$. In Figure 4.3a, we test initializing $MaxDeg$ to different values both below and above the ground truth degree. We see that the model converges towards the true degree in all cases. The convergence is significantly faster when using SoTL gradients with higher lengths of the unrolling T. This is shown in Figure 4.3b, where we instead fix the $MaxDeg$ initialization at -1. SoTL also outperforms optimization using the one-step validation loss (denoted as $T = 2\ \ (val)$). We remark that the performance of lower T for SoTL and validation loss optimization can likely be improved by finetuning the hyperparameters, such as increasing the hypergradient learning rate. However, we found higher T with lower learning rate to be more stable than lower T with higher learning rate. Additionally, SoTL with $T = 2$ still strongly outperforms validation loss optimization, but $T = 2$ corresponds to essentially the same hyperparameter setup as in the validation loss baseline since both algorithms then compute the hypergradients over two iterations at a time.

Our results for the weight-sharing linear model help motivate applying SoTL to modern weight-sharing NAS. Even though SoTL has theoretical guarantees only for stand-alone linear models, we have shown that it retains strong performance in the weight-sharing linear regression case. In Sections 4.2 and 4.3, we show that it is useful in practice even for large-scale NAS.

4.2 Discretized one-shot NAS

In order to demonstrate the strong performance of SoTL when used as a generalization estimator in discretized one-shot NAS, we first compare it against a range of validation set based metrics in Section 4.2.1 on NASBench-201, and then on NASBench-1shot1 and NASBench-301 in Section 4.2.2. We observe several intriguing phenomena about the bias of weight-sharing, which we investigate in detail in Sections 4.2.3 and 4.2.4.

In order to evaluate performance improvements of SoTL, we first randomly sample a set of architectures from the relevant search space. The architectures are then kept fixed across all experiments for the specific search space. Doing so makes the resulting ranking correlations and top-k performances comparable across different algorithms. We sample 200 architectures for NASBench-201 and NASBench-1shot1, and 350 for the DARTS search space because it is by far the largest. Each supernetwork training algorithm is run for three seeds to produce three supernetworks used to obtain the mean and standard deviation for all metrics. After the supernetwork training finishes, we train each sampled architecture individually for 300 minibatches. The additional training will be referred to as the *finetuning*. We compare SoTL against a minibatch training loss (TLMini), minibatch validation accuracy (ValAccMini) and whole validation set accuracy (ValAcc), which is only computed once every 100 minibatches for compute reasons. For all of those, we compute the Spearman rank correlation between the ranking proposed by each metric and the ground truth ranking computed from the test set performances provided by the respective NASBench API. In tables, we report SoTL after 100 minibatches training and the remaining metrics without any finetuning (i.e. using the inherited supernetwork weights directly) to make them equivalent to the standard one-shot evaluation protocols (Dong et al., 2020; Guo et al., 2020). We also show the performance of the top-10 architectures selected by each metric.

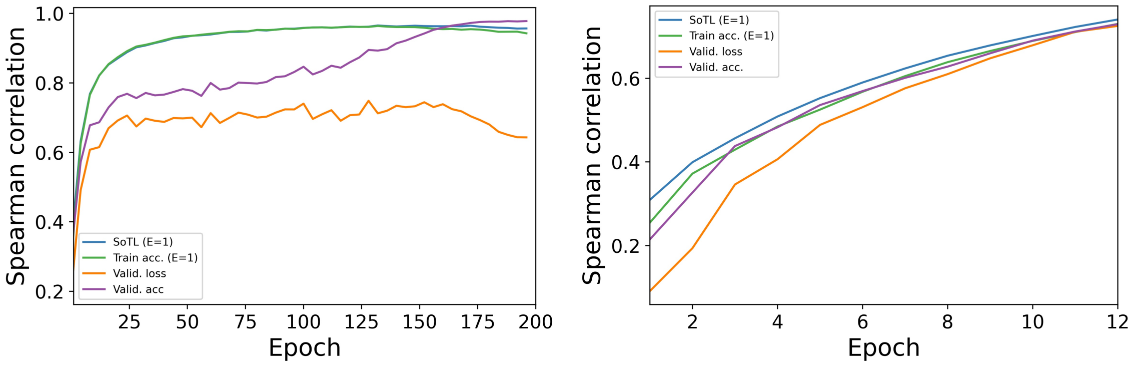
The main training hyperparameters are kept constant across algorithms for fairness of evaluation. The most important parameters are batch size 64 and learning rate of 0.01 for basic SPOS, which is decreased to 0.001 for SPOS variants. The lower learning rate reflects that the supernetwork is significantly better trained, and using too high learning rates causes catastrophic forgetting. We discuss hyperparameter choices in detail in Section 4.2.3. The whole finetuning procedure is significantly slower than the default evaluation proposed by Dong et al. (2020) and Li et al. (2020) who use a single validation minibatch accuracy to evaluate each architecture. However, the entire finetuning generally takes less time than the supernetwork training and at most a couple of hours on modern GPUs.

4.2.1 NASBench-201

NASBench-201 (Dong et al., 2020) is unique among NASBench series in that it provides the training loss for each architecture at every epoch, which makes it possible to evaluate SoTL by querying the API without any actual training. Apart from this, we also test the performance of SoTL when used to select the best architectures out of a supernetwork after additional short training of each candidate architecture.

While the original NASBench-201 (Dong et al., 2020) only included networks trained on CIFAR10 along with evaluation of transfer to CIFAR100 and ImageNet16-120, the later version called NATS-Bench (Dong et al., 2021) has training curves for all datasets separately. Furthermore, it includes a 200-epoch training schedule alongside a shorter 12-epoch one. We first evaluate the correlations on each dataset while training each architecture from scratch by randomly sampling 200 architectures for 10 random seeds and querying their training statistics at each epoch from the provided API.

Figure 4.4a shows the Spearman correlation curves between rankings proposed by various metrics and the ground truth test set performance for CIFAR10 training in the 200 epochs schedule. SoTL is significantly better correlated with the test



(a) 200 epochs training schedule

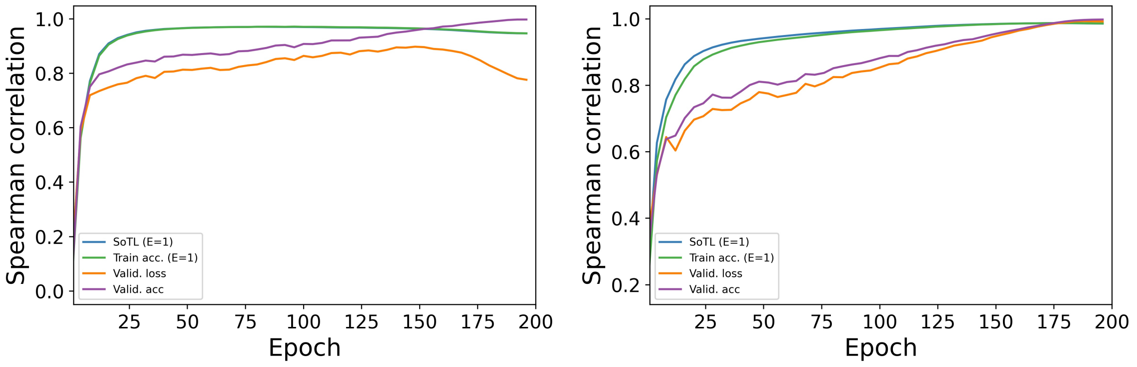
(b) 12 epochs training schedule

Figure 4.4: We notice a significant difference between training for 200 and 12 epochs schedules on NASBench-201 when using cosine annealing in both cases. For a) with the 200 epochs training, SoTL has a significantly higher correlation than validation statistics in the first 150 epochs before validation accuracy overtakes it. For b) with 12 epochs total, SoTL is still the best but only by a narrow margin which further closes as the training progresses.

set performance than validation metrics until around 150 epochs into the training, after which the performance of SoTL decreases slightly. Ru et al. (2020) attributed this to many architectures having near-zero training loss towards the end of training and effectively overfitting CIFAR10. For efficient NAS, it is desirable to stop the training of individual architectures as early as possible to save compute, which makes it possible to evaluate more architectures in total. Using SoTL is clearly superior for this purpose as it significantly outperforms validation accuracy early on in the training, and the regime in which SoTL is outperformed by validation accuracy would never be reached in early-stopped NAS. In particular, SoTL reaches 90% correlation within 50 epochs, while validation accuracy needs around 120 epochs to do so.

We also investigate the 12-epoch training schedule on CIFAR10 in Figure 4.4b, which shows several discrepancies from the 200-epoch schedule. All the training parameters, including cosine learning rate schedule, are the same apart from the total number of epochs. Interestingly, the gap between SoTL and validation accuracy is now much smaller and gets increasingly tighter. Nonetheless, we again see that SoTL has a better correlation early on and maintains this advantage until the end of training. Because the networks do not overfit CIFAR10 in only 12 epochs, we never see validation accuracy overtaking SoTL.

Ultimately, the final correlations in the 12-epoch schedule are around 75%, which is equivalent to the values that the longer training schedule reaches at epoch $E \approx 10$.



(a) CIFAR100

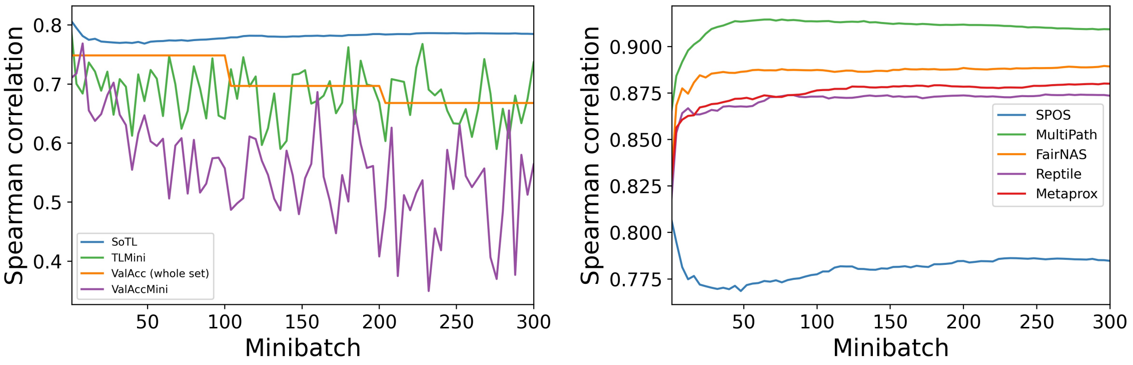
(b) ImageNet16-120

Figure 4.5: Training NASBench-201 architectures from scratch on CIFAR100 and ImageNet16-120 is mostly similar to training on CIFAR10. Notably, SoTL rank correlation with the ground truth performance does not decrease at the end of ImageNet16-120 training.

However, SoTL has over 15% better correlation than validation accuracy in the 200-epochs schedule at that point, whereas the difference in the final epoch of the 12-epoch schedule is only about 1%. It appears that the faster learning rate annealing in the 12-epoch schedule decreases the performance differential between SoTL and validation metrics. Based on those results, we hypothesize that the size of learning rates heavily influences the efficacy of SoTL, and we provide additional supporting evidence to this in Section 4.2.3.

We also investigated the 200-epoch schedule for CIFAR100 and ImageNet16-120 in Figure 4.5a and Figure 4.5b, respectively. For the most part, the results are the same as when training on CIFAR10. Notably, the correlations for SoTL rise even faster now, and it is possible to reach 90% Spearman correlation between proposed rankings and ground truth test set performance within only about 20 epochs of training. Even though the CIFAR100 SoTL correlation curve still shows a slight decrease towards the end of training, there appears to be no decrease at all for ImageNet16-120. This further suggests that this phenomenon is driven by overfitting on easier datasets when using long training schedules, and is not a practical concern for efficient NAS that uses early stopping.

Next, we introduce the results obtained by finetuning individual architectures that inherit weights from the supernetwork trained by the one-shot algorithms we described in Section 3.4. Using the supernetworks as initialization amortizes the cost of training each architecture from scratch thanks to the weight-sharing as each architecture will have already received a baseline level of training by the time we start collecting SoTL,



(a) SPOS and multiple metrics

(b) Multiple algorithms with SoTL

Figure 4.6: a) shows Spearman correlations of different metrics to the ground truth test set performance on NASBench-201 during finetuning. SoTL has both the highest correlation and lowest variance. The whole set validation accuracy (ValAcc) is only computed once every 100 minibatches to save compute. In b), we compare the correlations of SoTL across different one-shot training algorithms.

and it is not necessary to train for as long before good rank correlations are achieved. The success of SoTL even in this setup helps motivate applying it to DARTS, which we do in Section 4.3.

First, we show the correlation curves of all the metrics of interest using basic SPOS training in Figure 4.6a. Similar to the training-from-scratch case shown previously, the ranking based on SoTL has a much stronger correlation with the ground truth test set performance than those of other metrics, including the whole validation set accuracy. Even a single minibatch training loss tends to have a correlation slightly above that of minibatch validation metrics. In fact, the additional training decreases the correlation between validation metrics and the ground truth while SoTL and minibatch training losses slightly improve over time. SoTL maintains a correlation level of around 78% whereas whole validation set accuracy gradually drops below 70% percent. The minibatch statistics have high variance correlation coefficients that average at 70% and 60% for training loss and validation accuracy, respectively.

Next, we compare various one-shot training algorithms by measuring the correlations of SoTL during finetuning of their respective supernetworks in Figure 4.6b. Namely, we compare the basic SPOS (Guo et al., 2020; Li et al., 2020), FairNAS (Chu et al., 2019) and MultiPath (Yu et al., 2019) as one-shot NAS algorithms from prior work. We additionally benchmark Reptile (Nichol et al., 2018) and MetaMinibatch-Prox (Zhou et al., 2019) (which we refer to as Metaprox for brevity) as meta-learning algorithms based on the discussion in Section 3.4.2. The exact tabular results for all algorithms and all metrics are shown in Table 4.1.

All the SPOS variants significantly improve on the SoTL correlations by up to 10% compared to SPOS, with MultiPath having the best result overall at around 90.5%. Interestingly, SPOS has the best top-10 performances out of all algorithms across all datasets despite having weaker overall correlations. This suggests that ranking all architectures in the search space and ranking only the best architectures is not necessarily the same task. Algorithms such as FairNAS are explicitly motivated by making the training fairer for all architectures in the search space, but it appears the supernetwork gets naturally biased towards the best performing architectures otherwise, which makes the top-10 easier to select. Hence even though prior work identifies poor ranking correlation as a major drawback of SPOS (Zhang et al., 2021b; Bender et al., 2018), it appears possible to achieve good top-k performance while simultaneously having weak correlations across the entire search space. This also occurs on our other benchmarked search spaces, and it agrees with the recent results of Zhang et al. (2021b), who also show that all the SPOS variants hardly improve top-k performance. We note that the strong performance of SPOS in top-10 selection is strongly related to the chosen learning rate, and we discuss this in detail in Section 4.2.3.

From Figure 4.6, we can also see that finetuning for longer than 100 minibatches has almost no effect on the correlation levels while linearly increasing the training time. Training a NASBench-201 architecture for 100 minibatches takes around 4.5s on average, whereas running evaluation on the whole validation set takes roughly 6.5s on a Tesla P100. This shows that even though SoTL delivers better results than validation metrics computed on the whole validation set, computing SoTL is faster as well.

The results suggest that all the sophisticated one-shot variants are able to improve on the basic SPOS formulation in terms of ranking correlations despite having different motivations. Interestingly, MultiPath randomly samples multiple architectures at each iteration and averages their training gradients before each update. This actually turns out to be better than FairNAS, which uses the same setup but with more complex structured sampling. The meta-learning algorithms Reptile and Metaprox also significantly improve on SPOS and almost match the performance of specialized NAS algorithms. Both Reptile and Metaprox do not sample any more architectures than SPOS but instead focus on more expensive iterations that include second-order information to aid generalization across tasks (Nichol et al., 2018). This contrasts

		Spearman correlation					Top 10				
Dataset	Metric	SPOS	FairNAS	MultiPath	Reptile	Metaprox	SPOS	FairNAS	MultiPath	Reptile	Metaprox
CIFAR10	SoTL	0.78 (0.06)	**0.89 (0.01)**	**0.91 (0.01)**	**0.87 (0.01)**	**0.88 (0.01)**	**92.87 (0.25)**	92.23 (0.15)	92.73 (0.12)	91.97 (0.15)	92.00 (0.17)
	TLMini	**0.81 (0.04)**	0.83 (0.03)	0.86 (0.02)	0.81 (0.03)	0.82 (0.01)	92.13 (0.47)	92.20 (0.27)	92.30 (0.70)	92.40 (0.20)	**92.23 (0.31)**
	ValAcc	0.75 (0.03)	0.76 (0.12)	0.76 (0.06)	0.76 (0.16)	0.64 (0.02)	92.33 (0.74)	92.57 (0.25)	**93.03 (0.06)**	**92.63 (0.31)**	92.07 (0.23)
	ValAccMini	0.71 (0.05)	0.69 (0.11)	0.66 (0.19)	0.74 (0.21)	0.76 (0.08)	91.83 (0.70)	**92.93 (0.06)**	91.69 (0.97)	92.37 (0.25)	91.60 (0.87)
CIFAR100	SoTL	0.63 (0.04)	**0.87 (0.01)**	**0.84 (0.01)**	**0.83 (0.01)**	**0.84 (0.01)**	**69.55 (0.10)**	**68.35 (0.22)**	67.88 (0.09)	67.73 (0.18)	67.81 (0.26)
	TLMini	**0.82 (0.01)**	0.83 (0.03)	0.76 (0.01)	0.82 (0.02)	0.84 (0.03)	68.49 (0.36)	68.14 (0.08)	**67.98 (0.25)**	**68.58 (0.43)**	68.71 (0.65)
	ValAcc	0.78 (0.05)	0.50 (0.26)	0.28 (0.16)	0.58 (0.28)	0.78 (0.06)	68.48 (0.34)	67.37 (1.37)	63.58 (4.91)	66.44 (5.31)	**69.25 (1.11)**
	ValAccMini	0.75 (0.04)	0.24 (0.17)	0.32 (0.15)	0.57 (0.28)	0.71 (0.11)	68.69 (0.77)	64.66 (2.99)	62.19 (4.60)	67.14 (3.72)	68.69 (0.21)
ImageNet	SoTL	0.61 (0.01)	**0.86 (0.01)**	**0.84 (0.01)**	**0.82 (0.00)**	**0.84 (0.01)**	**41.67 (0.15)**	**41.53 (0.15)**	**41.23 (1.06)**	39.85 (0.49)	40.37 (0.06)
	TLMini	0.59 (0.06)	0.78 (0.03)	0.79 (0.03)	0.76 (0.03)	0.79 (0.03)	39.97 (1.39)	40.67 (0.98)	40.60 (0.85)	40.15 (1.20)	40.57 (0.50)
	ValAcc	**0.62 (0.06)**	0.60 (0.09)	0.59 (0.09)	0.79 (0.02)	0.73 (0.06)	40.17 (1.01)	41.00 (1.05)	41.07 (1.36)	**41.15 (0.21)**	**41.80 (0.61)**
	ValAccMini	0.53 (0.09)	0.42 (0.13)	0.41 (0.04)	0.64 (0.12)	0.41 (0.13)	39.40 (0.61)	37.47 (4.10)	35.56 (2.26)	39.75 (1.20)	37.97 (1.09)

Table 4.1: Summary of results on NASBench-201 across various SPOS-like algorithms. SoTL is usually the best in terms of rank correlation to the ground-truth ranking and often also on the top-10 performance. The best result across metrics for an algorithm is in bold. SoTL is reported after 100 minibatches finetuning and the remaining metrics are using the supernetwork weights directly.

with MultiPath and FairNAS, where the multiple architecture sampling can be understood as simple variance reduction due to averaging over multiple architectures. Furthermore, we note that all the one-shot variants have almost the same compute cost for training the supernetwork when measured in terms of SGD iterations and compute overall.

4.2.2 NASBench-1shot1 and DARTS search space

We first show the results in the NASBench-1shot1 search spaces as they have the true test set performances available, and then discuss the performance in the DARTS search space based on predicted accuracy using NASBench-301. We will abbreviate the three search spaces in NASBench-1shot1 as NB101-1, NB101-2 and NB101-3, respectively.

The performance of discretized single-path one-shot algorithms such as SPOS or ENAS has often been questioned using NASBench-1shot1 as an example because it was observed that such algorithms have particularly poor results in this benchmark (Zela et al., 2020; Zhang et al., 2021b). We show that the weak performance is primarily caused by the standard one-shot protocol consisting of random sampling followed by a single validation mini-batch evaluation, and using SoTL can make SPOS match or even exceed DARTS performance.

The full tabular results are available in Table 4.2. We again computed metrics based on 200 randomly sampled architectures from each search space, which were sampled ahead of time and kept the same for all the search algorithms. Overall, the rank correlations on NASBench-1shot1 are lower compared to NASBench-201. For example, the SoTL Spearman correlations in NB101-3, which is by far the largest

NASBench-1shot1 search space, are only around 15% for SPOS. All the specialized one-shot algorithms only improve the SoTL correlation to around 20%. However, SoTL still performs well compared to validation metrics which often have single-digit or negative correlations. The smaller search spaces NB101-1 and NB101-2 have higher SoTL correlations of around 50% across algorithms, consistently beating validation accuracy. The top-10 selected architecture performances are strongly improved by using SoTL as well. While the default evaluation protocol using validation accuracy often struggles to exceed average 92% test set accuracy and has very high standard deviations, SoTL consistently results in 93% or above accuracies with up to 10x lower standard deviations on all search spaces. The overall improvement is significant enough to make discretized one-shot NAS competitive performance-wise rather than consistently $1-2\%$ behind DARTS, which achieves around 93.3% on all search spaces. SPOS with SoTL actually performs better than DARTS in NB101-1, likely because it is the smallest search space.

The special one-shot variants including the meta-learning algorithms Reptile and Metaprox have roughly identical performance, and there is no clear winner. We could not evaluate FairNAS in this space because the architecture encoding in NASBench-1shot1 specifies operations per-node rather than per-edge as would be required to execute the FairNAS structured sampling of architectures. Nonetheless, we already saw in Section 4.2.1 that the performance of FairNAS is in practice quite similar to sampling multiple architectures randomly as in MultiPath. Overall, the SPOS variants consistently increase correlations on all three search spaces while also being strong on the top-10 performance, often outperforming basic SPOS on both.

Next, we show the performance of SoTL in the DARTS search space in Table 4.2, where the correlations and top-10 performances are estimated using NASBench-301. The results are qualitatively almost the same as for NASBench-201 and NASBench-1shot1. For this experiment, we sampled 350 architectures from the DARTS space because it is very large compared to the other search spaces. We again see a higher average correlation with lower standard deviation when we rank architectures using SoTL rather than validation accuracy, which translates into stronger top-10 performance. FairNAS performs particularly well with correlations up to 43% in comparison to the 18% of SoTL in SPOS. The SoTL correlations are roughly twice as high as those based on validation set metrics, which tend to be slightly below 20%. Combining MultiPath and Metaprox significantly increases correlations and top-10 performance for all NASBench-1shot1 search spaces, suggesting that the meta-learning algorithms can be stacked on top of NAS-specific algorithms to bring additional gains in performance

		Spearman correlation						Top 10					
Search space	Metric	SPOS	FairNAS	MultiPath	Reptile	Metaprox	MultiPath + Metaprox	SPOS	FairNAS	MultiPath	Reptile	Metaprox	MultiPath + Metaprox
NB101-1	SoTL	**0.43 (0.02)**	-	**0.54 (0.03)**	**0.51 (0.02)**	**0.55 (0.01)**	**0.61 (0.01)**	**93.44 (0.04)**	-	**93.41 (0.01)**	**93.48 (0.05)**	**93.46 (0.03)**	**93.46 (0.03)**
	TLMini	0.34 (0.08)	-	0.48 (0.03)	0.43 (0.05)	0.50 (0.05)	0.52 (0.07)	92.23 (0.71)	-	92.36 (0.69)	91.91 (0.69)	92.88 (0.67)	92.65 (0.28)
	ValAcc	0.23 (0.05)	-	0.27 (0.19)	-0.04 (0.13)	0.01 (0.10)	0.11 (0.31)	90.64 (2.44)	-	91.37 (1.97)	91.12 (2.00)	90.78 (1.46)	90.27 (0.84)
	ValAccMini	0.35 (0.17)	-	0.42 (0.03)	0.37 (0.06)	0.55 (0.03)	0.60 (0.01)	92.10 (0.83)	-	92.71 (0.46)	93.11 (0.26)	93.33 (0.18)	93.05 (0.36)
NB101-2	SoTL	**0.58 (0.01)**	-	**0.56 (0.01)**	**0.50 (0.02)**	**0.55 (0.01)**	**0.61 (0.01)**	**93.23 (0.18)**	-	**93.23 (0.06)**	92.79 (0.14)	**93.17 (0.08)**	**93.24 (0.03)**
	TLMini	0.43 (0.05)	-	0.51 (0.06)	0.41 (0.03)	0.48 (0.03)	0.57 (0.02)	92.08 (0.87)	-	92.33 (0.90)	91.91 (0.37)	92.43 (0.22)	92.85 (0.04)
	ValAcc	0.06 (0.19)	-	0.43 (0.08)	0.46 (0.07)	0.04 (0.31)	0.38 (0.12)	91.74 (0.20)	-	93.06 (0.21)	**93.21 (0.03)**	90.76 (2.10)	92.15 (1.47)
	ValAccMini	0.43 (0.04)	-	0.54 (0.05)	0.41 (0.08)	0.47 (0.05)	0.54 (0.03)	92.79 (0.76)	-	92.47 (0.68)	92.30 (0.61)	92.88 (0.46)	92.60 (0.05)
NB101-3	SoTL	0.15 (0.02)	-	**0.20 (0.01)**	**0.19 (0.01)**	**0.20 (0.01)**	**0.27 (0.04)**	**92.93 (0.14)**	-	**93.16 (0.05)**	**93.05 (0.26)**	**92.92 (0.28)**	**93.20 (0.05)**
	TLMini	**0.17 (0.04)**	-	0.18 (0.02)	0.15 (0.08)	0.15 (0.02)	0.27 (0.08)	91.14 (0.59)	-	91.62 (1.19)	91.71 (0.61)	90.93 (1.28)	92.33 (0.71)
	ValAcc	0.02 (0.14)	-	0.07 (0.01)	0.03 (0.19)	-0.08 (0.02)	0.01 (0.02)	91.05 (1.37)	-	91.36 (0.94)	91.23 (1.51)	91.31 (0.57)	91.40 (0.17)
	ValAccMini	0.05 (0.04)	-	0.15 (0.06)	0.03 (0.06)	0.16 (0.02)	0.19 (0.04)	89.76 (0.85)	-	90.72 (0.39)	91.62 (0.32)	92.20 (0.01)	92.86 (0.10)
DARTS	SoTL	**0.18 (0.01)**	**0.43 (0.02)**	**0.38 (0.02)**	**0.32 (0.02)**	**0.37 (0.01)**	-	93.41 (0.09)	**93.65 (0.07)**	**93.67 (0.10)**	93.51 (0.11)	**93.57 (0.02)**	-
	TLMini	0.02 (0.07)	0.23 (0.01)	0.23 (0.06)	0.20 (0.07)	0.26 (0.04)	-	93.04 (0.33)	93.30 (0.00)	93.28 (0.04)	93.41 (0.11)	93.29 (0.15)	-
	ValAcc	0.17 (0.03)	0.19 (0.05)	0.19 (0.02)	0.16 (0.03)	0.08 (0.05)	-	**93.45 (0.13)**	93.50 (0.14)	93.54 (0.08)	**93.56 (0.06)**	93.50 (0.10)	-
	ValAccMini	0.15 (0.07)	0.21 (0.13)	0.12 (0.05)	0.12 (0.14)	0.06 (0.04)	-	93.25 (0.15)	93.35 (0.21)	93.42 (0.03)	93.44 (0.15)	93.46 (0.04)	-

Table 4.2: Summary of results on NASBench-1shot1 and DARTS search spaces across various SPOS-like algorithms. SoTL consistently achieves the best correlation and top-10 performance out of all the benchmarked metrics. The best result across metrics for an algorithm is in bold. NB101-1/2/3 abbreviate NASBench-1shot1 search spaces 1, 2 and 3. We did not evaluate FairNAS for NASBench-1shot1 because its formulation is incompatible with those search spaces, and we could not evaluate MultiPath + Metaprox for DARTS due to compute reasons.

at the cost of extra compute. We also extract the top-1 architectures from three FairNAS supernetworks and retrain them using the original DARTS 600 epochs training protocol. This is shown in Table 4.3. The best architecture selected by ValAcc only achieved 2.81% test accuracy. Meanwhile, the best one selected by SoTL had a 2.73% test accuracy, which is better than the 2.76% of DARTS itself (Liu et al., 2018).

In summary, we have shown that the additional training to collect SoTL significantly improves results across the whole NASBench series even though the cost of extra training is typically lower than evaluating on the whole validation set. Furthermore, SoTL is essentially hyperparameter-free and can be applied to all one-shot algorithms that require choosing architectures from a candidate pool as the final step of the search. It appears that finetuning for as little as 50 minibatches might be sufficient to reap most of the benefits, and the performance of SoTL remains stable until at least one epoch worth of training, making it easy to find a suitable time to stop the finetuning. SoTL can easily be used for non-random architecture sampling as well. For instance, when using Regularized Evolution for the final architecture selection (Guo et al., 2020), it suffices to replace ranking via validation accuracy to ranking via SoTL for guiding the evolution. We leave such extensions for future work. In Section 4.3, we show that we can optimize SoTL directly as a part of the training procedure in gradient-based NAS algorithms to guide the search better than when optimizing validation loss.

	Test acc.	
	Average	Best
FairNAS (SoTL)	2.88 (0.19)	2.73
FairNAS (ValAcc)	3.02 (0.19)	2.81

Table 4.3: The test accuracy of architectures extracted from FairNAS supernetworks in the DARTS search space using either SoTL or validation accuracy on the whole validation set. The evaluation was done using the standard DARTS 600 epochs protocol. Using SoTL leads to better average and top-1 architecture selections across three random seeds.

4.2.3 The effect of learning rates

Our one-shot experiments generally reuse the default parameters from the corresponding NASBench setup (Dong et al., 2020; Ying et al., 2019; Li et al., 2020). For the finetuning phase, we set parameters similar to those in the search phase as a natural default. This avoids excessive overfitting to the benchmark because with the ground-truth test set performances being available, it would be possible to tune hyperparameters to maximize the resulting rank correlations. However, doing so would have been impossible outside of tabular benchmark settings. Nonetheless, we show that there appear to be several guidelines for setting the hyperparameters that follow a certain intuition and can be used to explain some of the interesting phenomena we observed when discussing the one-shot results on the NASBench series.

We study the effect of different batch sizes and learning rates on the finetuning phase, always using the same pre-trained supernetworks. We finetune each architecture for one epoch or 300 minibatches, whichever is sooner. This means that larger batch sizes do a lower amount of training iterations than smaller batch sizes. First, we discuss the possible prior motivations for the choice of finetuning learning rate. In regular standalone training of deep learning architectures, it has generally been observed that it is necessary to use large learning rates at least at the start of training in order for the final generalization performance to be good after decaying the learning rate to near zero. Likewise, higher batch sizes seem to require larger learning rates to work well (Lewkowycz et al., 2020; Li et al., 2019; Keskar et al., 2017; Smith et al., 2018). By using low learning rates, we thus risk not realizing an architecture's true generalization potential. On the other hand, large learning rates generally only lead to strong performance at the end of training rather than in the early phases since the larger step sizes seem to function by encouraging exploration.

Similarly, there is no obvious choice of learning rate when finetuning architectures that inherit weights from the supernetwork. With low learning rates, we risk never significantly modifying the supernetwork initialization, but high learning rates might lead to catastrophic forgetting. Moreover, because we finetune for one epoch or less, it is not obvious whether the advantages of high learning rates even have time to materialize. We might also expect higher batch size to work better than lower batch sizes because it provides more precise estimates of the training gradients, which should be advantageous over a short training period.

We study the problem by doing a small grid search with SPOS checkpoint initialization on NASBench-201 CIFAR10. We pick the batch size from $\{64, 128, 256\}$ and the learning rate from $\{0.001, 0.01\}$, computing a Cartesian product of the parameters where each setting is evaluated over three supernetwork seeds while the evaluated set of 200 randomly sampled architectures is held constant as usual. Figure 4.7 shows the resulting correlation and top-10 performance curves. We hold the batch size constant at 64 while varying the learning rate, and the learning rate at 0.01 while varying the batch size. Both smaller learning rate and higher batch size are generally understood to increase the stability of training. In our case, this appears to increase the rank correlations, but it simultaneously impedes the top-10 ranking. Having a high learning rate with low batch size achieves the best performance when the objective is to select the top architectures. This further adds to the evidence that being able to rank both the median and top-k architectures is difficult, and improvements in ranking median architectures do not necessarily transfer to ranking top architectures as observed by Zhang et al. (2021b).

Next, we study the positive correlation between learning rate and top-10 performance in more detail. Specifically, Figure 4.8 shows the top-10 curve and the average training loss across all architectures when training on ImageNet16-120 in the NASBench-201 search space. We see that using 0.01 learning rate achieves over 2% higher top-10 performance than using 0.001 learning rate. Moreover, the average training loss with 0.01 learning rate actually increases over the finetuning. This is a symptom of the learning rate being too high, and the average loss decreases properly when using only 0.001 learning rate. We also investigate the training loss for two architecture subgroups - the top 10% and median architectures as ranked by the true test set performances within our pre-sampled 200 architectures subset. When finetuning with 0.001 learning rate, the average training losses per group are extremely close to the point of being indistinguishable. Meanwhile, when using 0.01 learning

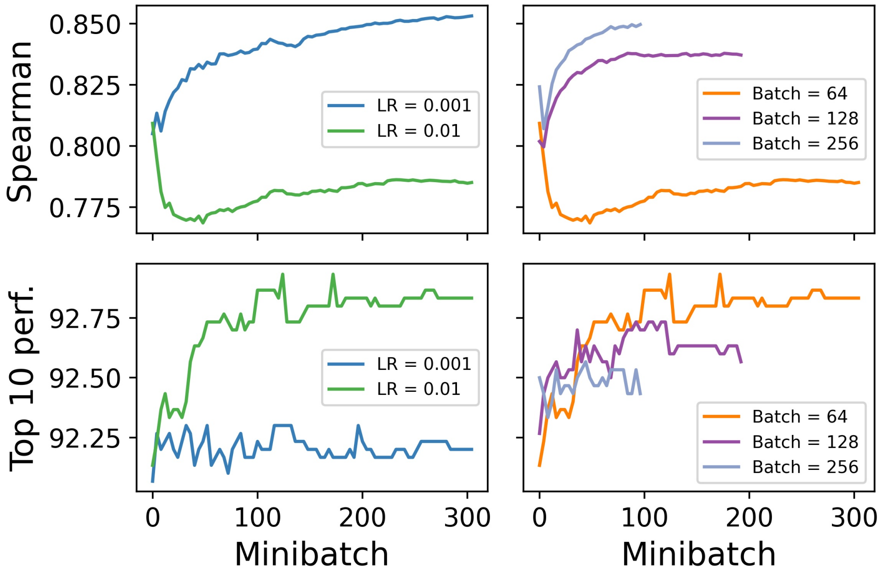


Figure 4.7: Grid-search of batch size and learning rate on NASBench-201 CIFAR10 for the finetuning of SPOS checkpoints. We show that while both lower learning rates and higher batch sizes improve overall correlations, they actually decrease the performance of top-10 ranked architectures.

rate, the gap between top 10% and median architectures widens during the training. This makes it easier to select the best architectures and results in higher top 10 performance of the selected architectures. We found this effect to occur on all NASBench-201 supported datasets, but it is the strongest on ImageNet16-120.

Table 4.4 shows the results of finetuning SPOS checkpoints with both low and high learning rates on all our benchmark datasets. Higher learning rate improves the top-10 performance across all datasets for SoTL, often despite lower correlations. The improvement in the top-10 is particularly significant on NASBench-201 and NB101-3. For both the DARTS search space and NASBench-201, there is a marked decrease in rank correlations after increasing the learning rate, but the same or higher top-10 performance than in the low learning rate case.

Moreover, Table 4.4 also reports the results for the whole validation set accuracy (ValAcc). However, it is now computed after 100 minibatches finetuning (same as SoTL) rather than using the supernetwork weights directly as before. The finetuning significantly improves the ValAcc correlations and top-10 performance on NASBench-1shot1. However, the performance on NASBench-201 stays the same or decreases. In the DARTS search space, ValAcc strongly improves when finetuning with 0.001 learning rate, and gets much worse with 0.01 learning rate to the point that the

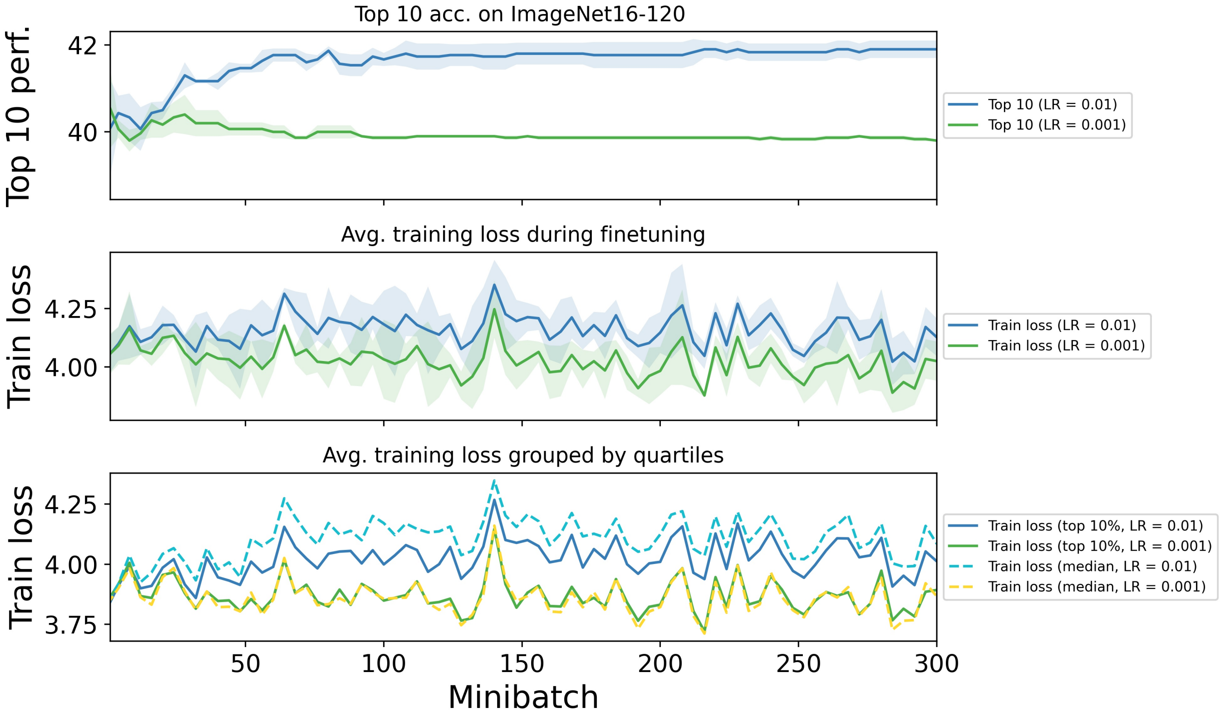


Figure 4.8: Training statistics from finetuning SPOS checkpoints on NASBench-201 ImageNet16-120. Using high learning rates gives better top 10 performance of selected architectures (top) even though the average loss goes up over finetuning (middle). We explain this by showing that high learning rates cause the best architectures to go up in loss the least (bottom).

correlation becomes zero.

It appears that ValAcc performs better when finetuned with a lower learning rate overall. This outcome is similar to the training-from-scratch experiments on NASBench-201 shown in Figure 4.4, where we saw that validation accuracy improves as an estimator of generalization when either the learning rate has been sufficiently annealed after long training in the 200-epochs case or when the learning rate simply became low quickly in the 12-epochs case. As a potential explanation for this effect, we conjecture that validation accuracy performance is improved with low learning rates because they encourage overfitting to the training data in a way that is not possible with high learning rates, which limits the performance of SoTL. The validation accuracy then accurately shows which architectures have overfitted the least, which is synonymous with generalization. However, more closely studying the interaction between learning rate regimes and training/validation data splits remains an exciting avenue for future work.

To conclude, we describe several limitations of our work in this Section. For

		Spearman correlation		Top 10	
Search space	Metric	SPOS (0.01)	SPOS (0.001)	SPOS (0.01)	SPOS (0.001)
NB201 (CIFAR10)	SoTL	0.78 (0.06)	0.84 (0.06)	92.87 (0.25)	92.23 (0.21)
	ValAcc	0.70 (0.11)	0.81 (0.06)	91.53 (2.10)	92.30 (0.20)
NB101-1	SoTL	0.43 (0.02)	0.41 (0.03)	93.44 (0.04)	93.32 (0.12)
	ValAcc	0.43 (0.02)	0.44 (0.01)	93.16 (0.11)	93.12 (0.04)
NB101-2	SoTL	0.58 (0.01)	0.55 (0.01)	93.23 (0.18)	93.00 (0.19)
	ValAcc	0.57 (0.02)	0.56 (0.01)	92.94 (0.06)	92.94 (0.32)
NB101-3	SoTL	0.15 (0.02)	0.16 (0.03)	92.93 (0.14)	90.69 (0.41)
	ValAcc	0.13 (0.01)	0.17 (0.06)	92.84 (0.21)	93.08 (0.07)
DARTS	SoTL	0.18 (0.01)	0.26 (0.03)	93.41 (0.09)	93.40 (0.06)
	ValAcc	-0.02 (0.09)	0.29 (0.04)	93.31 (0.19)	93.48 (0.17)

Table 4.4: A comparison of high and low learning rates for the finetuning phase across the whole NASBench series. High learning rate increases top-10 performance but simultaneously decreases the overall rank correlations. The parentheses after SPOS indicate the learning rate used. Note that 0.01 learning rate is the same as we used in our other experiments with SPOS.

compute reasons, we were not able to explicitly search for hyperparameters that would make something the phenomenon on NB101-1 and NB101-2, for which the results of finetuning with both 0.01 and 0.001 learning rates are almost the same. Likewise, we could not run similar experiments for SPOS variants such as FairNAS, although we found in isolated cases that using 0.01 learning rate is generally too high for them and leads to catastrophic forgetting that harms performance overall. We did find all the benchmarks to share the property that actually decreasing the training loss over finetuning is not necessary for SoTL to work. Even when the training loss increases over time, it increases the least for the best architectures, even if only by a slight margin, and this in turn still allows SoTL to outperform validation set metrics. In fact, using 0.01 rather than 0.001 learning rate universally leads to higher average training loss on all benchmarks similar to the observations in Figure 4.8.

4.2.4 Investigating the bias of high learning rates

Based on the previous observations about learning rates in Section 4.2.3, we propose an intuitive explanation. First, architectures that generalize better also tend to have smoother loss landscapes. In practice, this makes them easier to optimize, and in particular, it makes it possible to use higher learning rates to train them using SGD. This means that the high-performing architectures train quickly compared to weaker architectures, leading to the idea of training speed determining generalization that underlies SoTL. Weakly performing architectures have significantly less smooth loss landscapes, making the training fail for high learning rates or be very slow. The seeming paradox of high learning rate SPOS having low rank correlations but high top-10 performance can then be explained by the best architectures being able to train significantly better with 0.01 learning rate than the weaker architectures. Excessively high learning rates cause the weak architectures to suffer catastrophic forgetting, which makes their performance more random-like. This in turn means that the weaker architectures can no longer be properly distinguished, which hurts the overall rank correlations. It also effectively prunes the weaker architectures out of consideration for top-10, which makes the true top-10 easier to select. Meanwhile, all the architectures train at approximately the same rate with 0.001 learning rate, and it becomes difficult to distinguish the best ones. However, the correlations are improved because no architectures devolve into random-like behavior.

Smoothness of loss landscapes has previously been explored in the context of DARTS by Shu et al. (2020), who argue that DARTS prefers shallow architectures because they have smoother loss landscapes and are easier to optimize while still having good generalization performance. Furthermore, flat minima are commonly identified with good generalization properties (Keskar et al., 2017). Other recent work (Fort et al., 2020; Smith et al., 2020) connects gradient variance over training data to generalization, and gradient variance during training can also be understood in terms of the smoothness of the training loss landscape. Flat loss landscapes have low gradient variance, whereas sharp loss landscapes lead to frequent zigzag patterns when training with SGD, which results in high gradient variance. Based on this insight, we design the last experiment of this Section to act as a preliminary investigation of the bias behind high learning rates.

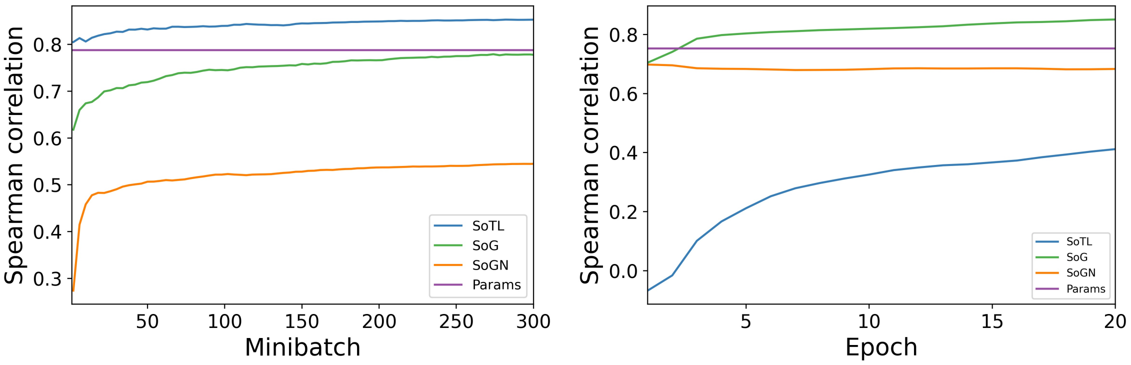
In order to test whether well-performing architectures in our benchmarks also tend to have smoother loss landscapes, we design two more metrics. First, Sum-of-Gradients (SoG) accumulates the gradients over the training trajectory dimension-wise, and then all the dimensions are summed up into a scalar at prediction time. When training with SGD, the gradient accumulation also corresponds to the distance from initialization. Second, we also track the Sum-of-Gradient-Norm (SoGN), which is the accumulated L1 norm of gradients over the training trajectory. In contrast to SoG, SoGN is computed by taking the norm at each training iteration and keeping a running scalar sum of the norms. In order to have high SoGN, it is sufficient to zigzag in sharp regions of the loss landscape without making any real progress. To have high

SoG, it is necessary for the training to have gradients that are consistently biased in a certain direction (i.e. the final optimum) since the summation is now sensitive to signs of the gradients, and it is also necessary to have high magnitude gradients at the same time. Gradients with high norms that have a consistent direction then naturally lead to fast training, resulting in high SoG and low SoTL. Hence if it was true that better generalizing architectures have smoother loss landscapes than their weaker counterparts, it should be the case that they have low SoTL and high SoG.

Figure 4.9 shows the resulting Spearman correlation curves during training on NASBench-201 CIFAR10 for both finetuning from SPOS checkpoints and for training the same architecture subset from scratch over 20 epochs. In both cases, SoG achieves a significantly better correlation than SoGN. For example, SoG has up to 85% correlation in the training from scratch case compared to 68% of SoGN. We also report the correlation between parameter count and the ground truth test set performance (denoted as Params). Even the parameter count achieves correlations of around 80% despite it being a very simple baseline. The parameter count is an important metric in this case because both SoG and SoGN are a proxy to the number of parameters since more parameters naturally lead to higher gradient norms. However, we see that SoGN tends to have poorer performance than SoG, which suggests that correlation with parameter count is insufficient to explain the difference between those two metrics. Instead, we attribute this to the weaker architectures having zigzag training trajectories, which induces high SoGN but low SoG.

We evaluate SoG and SoGN on all our benchmarks in Table 4.5 using both 0.01 and 0.001 learning rates. Apart from rank correlations with the true test set performance, we also show rank correlations between the proposed rankings and parameter count for SoTL and SoG (denoted as SoTL-P and SoG-P, respectively). In terms of the ground truth correlation, SoG generally outperforms SoTL, especially on NASBench-1shot1 and the DARTS search space. The parameter count maintains a correlation of roughly 80%, which is enough to make it a top predictor across most search spaces. We further see that SoG can be up to 85% correlated with the parameter count, which is higher than its correlation with the ground truth performance. However, even SoTL can better correlate with the parameter count than the ground truth, particularly on NASBench-1shot1.

We also note that the rank correlations of SoG are lower with 0.001 learning rate, and the same happens but is even more severe for SoGN. If the intrinsic correlation between parameter count and gradient norm was the only source of high performance for SoG and SoGN, there should be no major change when going from 0.01 to 0.001



(a) NB201 - Supernetwork initialization (b) NB201 - Training from scratch

Figure 4.9: a) Spearman correlations of SoTL and SoG/SoGN during finetuning of SPOS checkpoints on NASBench-201 CIFAR10 with 0.001 learning rate. SoG attains significantly better correlations than SoGN. We note that the difference between them is smaller with high learning rates as shown in Table 4.5. b) Similar as before, except that the architectures are now trained from scratch rather than with supernetwork initialization. The standalone training uses NASBench-201 defaults for training-from-scratch, including 0.1 learning rate.

learning rate. Instead, if we appeal to the intuition outlined at the beginning of this Section, a possible explanation might be that all architectures tend to zigzag around sharp regions for a low enough learning rate. Having higher learning rates instead makes the good architectures stay in flat regions, where the weights solution moves quickly towards the optimum with SGD, while poor architectures are still stuck in sharp regions.

Our results here are similar to those of Zhang et al. (2021b), who also show that SPOS-like algorithms are usually better correlated with parameter count than the ground truth. The excessive predictiveness of simple parameter count can be seen as a weakness of all the most common vision search spaces for NAS. Zhang et al. (2021b) also show that for DARTS-PTB, which is a natural language processing benchmark from the original DARTS paper (Liu et al., 2018), there is a negative correlation between parameter count and performance, and also a negative correlation between the supernetwork validation accuracy and ground truth performance. We were not able to evaluate SoG on DARTS-PTB because it would require excessive compute for training all examined architectures to convergence.

Nonetheless, it appears that the success of one-shot NAS might be at least partially driven by the correlation between parameter count and ground truth test set performance on mainstream NAS benchmarks. While it is not a surprise that such a correlation would increase the performance of SoG, the fact that even SoTL can

Search space	Spearman correlation						
	SoG (0.01)	SoG (0.001)	SoGN (0.01)	SoGN (0.001)	Params	SoTL-P	SoG-P
NB201 (CIFAR10)	0.86 (0.01)	0.78 (0.01)	0.81 (0.04)	0.54 (0.02)	0.79 (0.00)	0.56 (0.07)	0.78 (0.03)
NB201 (CIFAR100)	0.84 (0.01)	0.78 (0.01)	0.82 (0.01)	0.53 (0.03)	0.77 (0.00)	0.55 (0.03)	0.80 (0.01)
NB201 (ImageNet)	0.72 (0.00)	0.58 (0.00)	0.71 (0.00)	0.42 (0.01)	0.74 (0.00)	0.53 (0.08)	0.70 (0.06)
NB101-1	0.67 (0.00)	0.61 (0.01)	0.12 (0.01)	-0.05 (0.03)	0.79 (0.00)	0.58 (0.02)	0.84 (0.01)
NB101-2	0.62 (0.01)	0.47 (0.02)	0.14 (0.03)	-0.13 (0.02)	0.81 (0.00)	0.79 (0.01)	0.73 (0.01)
NB101-3	0.43 (0.01)	0.40 (0.01)	0.32 (0.01)	0.08 (0.01)	0.78 (0.00)	0.27 (0.02)	0.68 (0.01)
DARTS	0.45 (0.01)	0.38 (0.01)	0.50 (0.01)	0.38 (0.02)	0.47 (0.00)	0.27 (0.03)	0.71 (0.05)

Table 4.5: Spearman correlations for SoG and SoGN across all our benchmark datasets. While the performance of SoG is generally very high, it is usually better correlated to the parameter count than the ground truth. However, the same can be true for SoTL. The "-P" suffix signifies that the rank correlation is with respect to the parameter count rather than the test set accuracy. Parentheses after metric indicate the learning rate used to obtain it.

be significantly better correlated with parameter count rather than the ground truth test set accuracy is concerning. However, we note that it is impossible to have both a good correlation with test set accuracy and a bad correlation with parameter count because the correlation between parameter count and test set accuracy is so strong in the first place.

Ultimately, it is not clear whether SoG outperforms SoTL simply due to better exploiting the bias towards high parameter count architectures also having high generalization performance. The experiments we have presented here are quite preliminary and have mixed results at times. While the intuitive reasoning behind SoG is appealing, more work is needed to verify its practical usefulness and its relationship to the interesting phenomena we observed when finetuning one-shot supernetworks. We leave such extensions for future work.

4.3 Differentiable NAS

In this section, we first discuss the performance of SoTL-DARTS in comparison to other weight-sharing NAS algorithms on NASBench-201 and NASBench-1shot1 in Section 4.3.1. Next, we do the same in the DARTS search space using both the model-based predictors from NASBench-301 as well as training discovered architectures from scratch using the original DARTS evaluation protocol in Section 4.3.2. Finally, we apply a suite of DARTS diagnostics previously described in the differentiable NAS literature to identify the source of performance improvement in SoTL-DARTS in Sections 4.3.3, 4.3.4 and 4.3.5.

We run all our experiments with three random seeds to account for the search instability in DARTS. We always use 100 minibatches for computing the approximate SoTL gradients. Experiments on CIFAR10 generally use only 50% of the CIFAR10 training data for SoTL-DARTS, which amounts to using only the training set of the train-validation 50-50 split in original DARTS. This makes it possible to show that the SoTL performance improvement is not caused by increasing the volume of training data that would be caused by merging train-validation into a single training set. Appendix B summarizes all the relevant hyperparameter defaults that are adopted from the relevant NASBench.

4.3.1 NASBench-201 and NASBench-1shot1

NASBench-201 is a notorious example of DARTS architecture overfitting since both first and second-order DARTS tend to select an architecture with all skip connections. This leads to DARTS catastrophically underperforming both other weight-sharing algorithms and query-based NAS algorithms, such as random search (Bergstra et al., 2012) or Regularized Evolution (Real et al., 2019). We summarize the NASBench-201 results in Table 4.6. When training SoTL-DARTS, we use the train portion of the NASBench-201 data splits, which results in using only 50% of the training data for CIFAR10 due to the 50-50 train-validation split, and 100% for both CIFAR100 and ImageNet16-120 since the original splits use a small portion of the true test set as validation sets. All training hyperparameters are kept at the default values of their respective NASBench implementations.

In comparison to the baseline DARTS which achieves only 59.84% CIFAR10 test accuracy on NASBench-201, SoTL-DARTS is able to prevent the skip connection collapse, and its selected architectures reach 92.66% average test accuracy. While this is still not enough to outperform other baselines such as Regularized Evolution

(Real et al., 2019), which are especially performant because the total number of architectures in NASBench-201 is small, SoTL-DARTS is highly competitive with other weight-sharing algorithms such as GDAS (Dong et al., 2019b) or SPOS (Li et al., 2020). The performance improvements of SoTL-DARTS also extend to CIFAR100 and ImageNet16-120, on which the accuracies changed from 60.49% and 36.79% to 68.13% and 36.6%, respectively.

We also show several ablations to ascertain the strong performance of our method. First, we report the results of searching via SoTL-DARTS for 150 epochs instead of the default 50 epochs. The performance does not degrade even with more training epochs unlike baseline DARTS, for which longer training is known to exacerbate the skip connection overfitting problem (Zela et al., 2019). While the 50 epochs training default in DARTS is necessary to act as early stopping to prevent excessive architecture overfitting, SoTL-DARTS is less sensitive to the number of training epochs. We also test SoVL-DARTS which is similar to first-order DARTS, except that the architecture gradients are accumulated over 100 validation minibatches before each update. SoVL-DARTS has even worse performance than normal DARTS, which shows that the performance gains in SoTL-DARTS come from computing gradients with respect to the training data rather than the summation over longer periods. The performance of SoTL-DrNAS is slightly better than that of the baseline DrNAS (Chen et al., 2021) as well. Figure 4.10c shows the search trajectory of DARTS and SoTL-DARTS during CIFAR10 search. It is clear that while DARTS rapidly overfits to all skip connections, SoTL-DARTS keeps improving with more training time.

For NASBench-1shot1, we observe that SoTL-DARTS performs poorly on all three search spaces. We were able to fix this by significantly altering the hyperparameters, decreasing the SoTL summation period to 10 from 100 and decreasing the batch size to 32 from 64. Remarkably, the weak performance is not due to a bad choice of operations; in fact, both SoTL-DARTS and normal DARTS choose 3×3 *conv* for almost every operation. Instead, we identify that the problem is in the topology of the cell since SoTL-DARTS chooses deep cells while DARTS chooses shallow cells. This issue is discussed in great detail in Section 4.3.4. We suspect that changing the hyperparameters as we did destabilizes the search, which in turn implicitly regularizes the depth of discovered architectures because deeper architectures are more unstable to train. Having the SoTL summation period set to 100 over-stabilizes the training, allowing the deep architectures to dominate the search. However, we could not assess whether this hypothesis holds in greater generality over other search spaces due to compute reasons. Nonetheless, it highlights the fact that adopting the DARTS default

	CIFAR10	CIFAR100	ImageNet16-120
REA (Real et al., 2019)	94.02 (0.31)	72.23 (0.95)	45.77 (0.80)
Random Search (Bergstra et al., 2012)	93.90 (0.26)	71.86 (0.89)	45.28 (0.97)
SPOS (Li et al., 2020)	91.05 (0.66)	68.26 (0.96)	40.69 (0.36)
GDAS (Dong et al., 2019b)	93.23 (0.58)	68.17 (2.50)	39.40 (0.00)
ENAS (Pham et al., 2018)	93.76 (0.00)	70.67 (0.62)	41.44 (0.00)
DrNAS (Chen et al., 2021)	93.76 (0.00)	68.82 (2.06)	41.44 (0.00)
DARTS (first-order) (Liu et al., 2018)	59.84 (7.84)	61.26 (4.43)	37.88 (2.91)
DARTS (second-order) (Liu et al., 2018)	65.38 (7.84)	60.49 (4.95)	36.79 (7.59)
SoTL-DARTS	92.66 (0.00)	68.13 (2.18)	36.60 (0.00)
SoTL-DARTS (150 epochs)	89.89 (2.52)	71.03 (0.66)	33.75 (0.00)
SoVL-DARTS	54.30 (0.00)	15.61 (0.00)	17.37 (1.48)
SoTL-DrNAS	93.76 (0.00)	71.11 (0.00)	41.44 (0.00)

Table 4.6: Baseline DARTS overfits to all skip connections on NASBench-201, which leads to very poor performance. SoTL-DARTS prevents this and achieves 92.66% test set accuracy. Most baselines were reprinted from Dong et al. (2021) except the DrNAS results, which are our own. Notably, we were unable to reproduce the original results from Chen et al. (2021) with near-perfect performance using the public code.

	NB101-1	NB101-2	NB101-3
DARTS	93.33 (0.01)	93.37 (0.00)	93.35 (0.01)
SoTL-DARTS	91.86 (0.68)	90.15 (0.00)	87.87 (0.00)
SoTL-DARTS (diff. hparams)	93.28 (0.00)	93.33 (0.00)	92.87 (0.53)

Table 4.7: On NASBench-1shot1, the performance of SoTL-DARTS is quite weak overall and significantly trails baseline DARTS on all the three search spaces. NB101-1, NB101-2 and NB101-3 abbreviate the NASBench-1shot1 search spaces 1, 2 and 3, respectively.

parameters might be quite suboptimal, especially since we show in later Sections that the inherent biases of SoTL-DARTS are fundamentally different from normal DARTS.

For NASBench-1shot1, we also tried several other SoTL-DARTS variants including computing exact rather than approximate SoTL gradients over 7 time steps, which is feasible in this case because the NASBench-1shot1 supernetwork is small. However, none were able to fix the low performance without altering other hyperparameters. It also did not make a difference whether we used 50% or 100% of CIFAR10 training data for the search.

4.3.2 DARTS search space

In this section, we run the search using SoTL-DARTS on the original DARTS space and several of its variants. We show the search trajectories predicted by NASBench-301, and also run the full DARTS evaluation protocol for the most promising algorithms. Most prior work reuses the DARTS evaluation protocol, which involves running the search four times, retraining all architectures from scratch, picking the best out of the four and retraining it ten times from scratch using 600 epochs training; the average of those ten runs is then reported as the mean performance of the best architecture. We instead report the average and best performance of the search ran three times and evaluating each of the found architectures only once due to compute constraints. We were unable to evaluate the searched architectures more thoroughly because the 600 epochs training takes around 2 days on a GTX 1080Ti for a single seed.

First, we show the NASBench-301 predicted performances of SoTL-DARTS and SoTL-DrNAS in Figure 4.10. The SoTL versions of the algorithms maintain stronger anytime and final performances throughout the whole search compared to the baselines. Simultaneously, the compute cost of SoTL-DARTS is only as high as first-order DARTS because there is no need to compute second-order gradients. The performance of SoTL variants is the highest at the end of the search whereas normal DARTS overfits towards the end of the search, which suggests that SoTL-DARTS is more robust to the length of training. This mirrors the observed robustness to the number of epochs in the NASBench-201 results from Section 4.3.1, in which we showed it was possible to train for 150 epochs without overfitting in the search. Moreover, we again ran the SoTL-DARTS search using only 50% of CIFAR10. Hence SoTL-DARTS outperforms DARTS while using only half the data in total.

Table 4.8 shows the full performance results including several other evaluation protocols. First, we tabulate the prediction from NASBench-301 and also retrain using the original DARTS evaluation protocol with 600 epochs training (denoted as 20 layers eval). The best architecture found by SoTL-DARTS has 2.73% error rate, which is slightly better than the baseline DARTS at 2.76%, but significantly worse than the other baselines such as PDARTS (Chen et al., 2019) or PC-DARTS (Xu et al., 2019). The remainder of this Section will be spent on diagnosing the problems that prevent achieving better performance.

In particular, we will now argue that the problem is not necessarily that the search works poorly, but rather that the optimum during the search does not transfer well to the full evaluation architecture due to an issue known as the depth gap (Chen et al., 2019; Chen et al., 2021). The network used for DARTS search only uses 8 layers of the searched cell because of memory requirements, and the evaluation network depth is increased to 20 layers after architecture selection since the end goal is to find an architectural cell that would work for very large networks. SoTL-DARTS finds a better architecture than DARTS in the 8 layers regime, which we verify by retraining the architectures from scratch using only 8 layers (denoted as 8 layers eval). All the SoTL variants have significantly better performance than the baselines when measured with 8 layers evaluation. This is particularly relevant for SoTL-DrNAS, which reports a poor performance of only 2.90% in the original 20 layers eval, but it is the best in 8 layers eval.

Moreover, we also run SoTL-DARTS using 20 layers during the search itself, which we refer to as SoTL-DARTS-20. This is particularly tractable for SoTL-DARTS because no second-order gradients are needed unlike in normal DARTS. As a result, the total compute time is roughly equivalent to second-order DARTS ($\approx$ 1 day) when using the default 64 batch size, which induces a memory requirement of only 23GB VRAM using our implementation. For SoTL-DARTS-20, we also used 100% of the CIFAR10 training set to compare it more fairly against other state-of-the-art algorithms. Table 4.8 shows that the search results are improved, and the average search performance of SoTL-DARTS-20 exceeds the best architecture from DARTS. However, even the top-1 performance of SoTL-DARTS-20 at 2.68% is not competitive with the state-of-the-art performances of algorithms such as DrNAS (Chen et al., 2021) or iDARTS (Zhang et al., 2021a), which can achieve CIFAR10 test set accuracy of less than 2.5%. We also tested first-order DARTS with 20 layers in the search (denoted as DARTS-20) and found it to improve the performance to 2.89% from the original 3.00%.

That said, we also struggled to reproduce the published performance of state-of-the-art algorithms, which we account to not having run enough search seeds. Because NAS literature usually only reports the top-1 discovered architecture performance, rerunning the search multiple times is trivially expected to yield better results. In fact, some related work such as GAEA (Li et al., 2021) mention they ran the search 15 times, whereas ours was only run three times. Not only we found it difficult to reproduce the search results of other algorithms, even retraining the best published architectures did not yield the same performance as reported in the papers. We suspect that changes in library versions slightly perturb the final performance. Because the gap between baselines such as DARTS and state-of-the-art algorithms on CIFAR10 is only about 0.25% extra accuracy, even a small performance regression is

	NB301	8 layers eval	20 layers eval	
	Average	Average	Average	Best
DARTS (1st) (Liu et al., 2018)	93.89 (0.17)	95.13 (0.18)	-	3.00
DARTS (2nd) (Liu et al., 2018)	93.79 (0.57)	95.09 (0.22)	-	2.76
DrNAS (Chen et al., 2021)	93.75 (0.58)	95.53 (0.28)	-	2.46
PC-DARTS (Xu et al., 2019)	-	-	-	2.57
PDARTS (Chen et al., 2019)	-	-	-	2.50
iDARTS (Zhang et al., 2021a)	-	-	-	2.37
SoTL-DARTS	94.11 (0.24)	95.36 (0.22)	2.82 (0.12)	2.73
SoTL-DrNAS	94.50 (0.13)	95.65 (0.19)	3.00 (0.14)	2.90
SoTL-PDARTS	93.51 (0.15)	95.39 (0.22)	3.27 (0.12)	3.19
SoTL-DARTS-20	94.21 (0.18)	95.53 (0.18)	2.72 (0.04)	2.68
DARTS-20 (first-order)	93.31 (0.31)	94.94 (0.39)	2.97 (0.11)	2.89

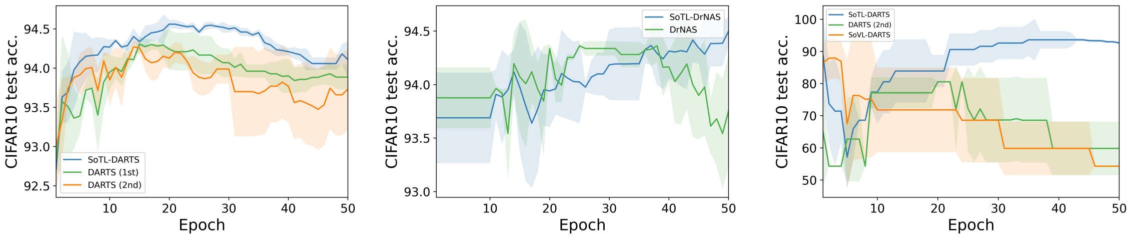
Table 4.8: Summary of performances of SoTL variants against baselines in the DARTS search space. SoTL-DARTS can match baseline DARTS and SoTL-DARTS-20 achieves better average performance that DARTS's best. "-" means that the original papers did not provide those results and we did not have the resources to re-run all the baselines. DARTS (1st) and (2nd) refer to first and second-order DARTS, respectively.

enough to invalidate the results. Moreover, the reproducibility of NAS research has previously been a subject of criticism (Li et al., 2020).

4.3.3 Qualitative analysis of the discovered architectures

Our experiments with altering the number of layers in the search supernetwork yielded additional interesting insights on the depth gap in DARTS. First, we found that the architectures discovered by SoTL-DARTS are very distinct from baseline DARTS even in the 8 layers case. Instead of overfitting to skip connections, the architectures become very heavy on separable convolutions and often contain no parameter-free operations whatsoever. The only parameter-free operation that tends to get selected out of the present *skip connection*, *max pool* and *avg pool* tends to be *max pool*, but it is not rare for the final architecture to have no parameter-free operations whatsoever.

In order to investigate the different operation bias between DARTS and SoTL-DARTS further, we first evaluated the search space S2 from Zela et al. (2019). S2 is a subset of the original DARTS search space containing only $\{\textit{skip connection}, 3 \times 3\ \textit{separable convolution}\}$. We compare the number of skip connections alongside the predicted NASBench-301 performance in Figure 4.15. Normal DARTS selects up to



(a) DARTS NB301 perf. (b) DrNAS NB301 perf. (c) DARTS NB201 perf.

Figure 4.10: a) and b) compare the SoTL versions of DARTS and DrNAS against the baselines with search trajectories provided by NASBench-301. The SoTL variants typically maintain higher performance throughout the whole search.

12 out of 16 operations as skip connections, which is excessive and leads to poor performance. Meanwhile, SoTL-DARTS selects zero skip connections during most of the search, which means its predicted accuracy does not degrade towards the end of the search as in normal DARTS.

To prevent ambiguity, we will also refer to normal SoTL-DARTS with 8 layers during the search as SoTL-DARTS-8. We argue that the relatively weak performance of SoTL-DARTS-8 architectures in the 20 layer evaluation is caused by architectural biases that work better with only 8 layers of depth rather than 20 layers. We conjecture that SoTL-DARTS-8 does not pick skip connections because they are not required with only 8 layers, but SoTL-DARTS-20 starts selecting them as they are more useful in deeper architectures (He et al., 2016; Vaswani et al., 2017). Figure 4.12 shows that this happens in practice in the search on S2 using SoTL-DARTS-20 and DARTS-20. The overfitting to skip connections in DARTS-20 gets even worse, and up to 15 out of 16 selected operations tend to be skip connections. Importantly, SoTL-DARTS-20 now also starts picking skip connections throughout the whole search including at the end. Meanwhile, SoTL-DARTS-8 picked zero skip connections at all times on S2.

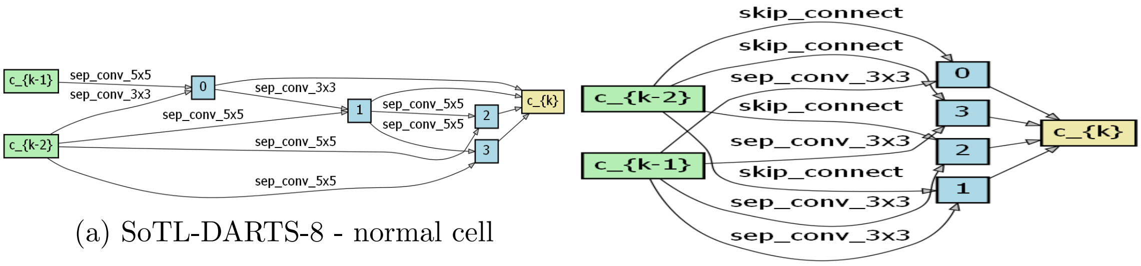
Figure 4.12 also shows that SoTL-DARTS-20 picks skip connections and other parameter-free operations both during and at the end of the search in the full DARTS space. This is the likely cause of the stronger performance of SoTL-DARTS-20 when using the DARTS evaluation protocol compared to SoTL-DARTS-8. The just described experiments yield a concrete example of the depth gap that aligns well with the intuition that skip connections should be picked more as the search network gets deeper. Our best SoTL-DARTS-8 and SoTL-DARTS-20 cells are visualized in Figure 4.11.

Some prior work such as PDARTS (Chen et al., 2019) or DrNAS (Chen et al., 2021) also tries to solve the depth gap issue by using progressive deepening schedules, in which some parts of the operations set are discarded while the network is made deeper during multiple phases of the search. This keeps the memory usage low overall. We adapted this approach and evaluated SoTL-PDARTS, but found it unable to amend the issue. The skip connections in particular already get discarded during the early phases of the search while the search network is still shallow, and as a result, they can never get selected at the end. Therefore, the final architecture still has zero skip connections. From Table 4.8, we also see that the performance of SoTL-PDARTS is especially poor with only 3.19% best test accuracy. While PDARTS (Chen et al., 2019) has a strong performance of 2.50%, the authors noted that it was necessary to manually restrict the number of skip connections in the final architecture since it would give poor performances otherwise. Meanwhile, the deepened DARTS-20 (first-order) reached 2.89% without any special treatment of skip connections, which is significantly better than the baseline at 3.00%. Using the full-depth search thus appears to yield better performances than the progressive deepening without any other adjustments for both SoTL and normal DARTS.

As noted, the architectures found by progressive deepening are quite different from those that were discovered using full depth search from the start. The full depth search therefore appears necessary to properly alleviate the depth gap, and even though the algorithmic changes in PDARTS can provide performance gains, those might not necessarily be due to reducing the depth gap. While most DARTS-related literature implicitly assumes that the architectural optimum in the search with 8 layers network is the same as when evaluating with a 20 layers network, our observations regarding skip connections in SoTL-DARTS show that this is not the case. Instead, we suggest that a promising avenue for improvement in differentiable NAS is to focus on searching without any depth gap. This can be particularly feasible from a memory standpoint when using techniques such as gradient accumulation, which might lead to higher running times of algorithms, or partial channels connection similar to PC-DARTS (Xu et al., 2019).

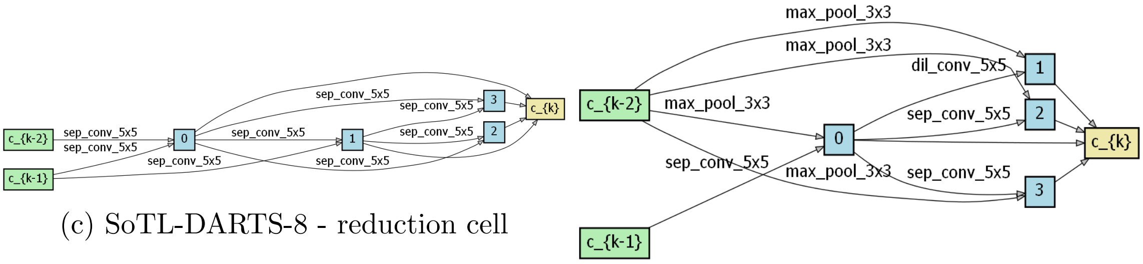
4.3.4 Bias towards shallow architectures

In this Section and Section 4.3.5, we conclude our investigation of SoTL-DARTS by applying several heuristics designed to explain the behavior of DARTS. We show that the observations from prior work do not apply to SoTL-DARTS, and the predictions made therein might even be the opposite of what happens in practice. First, we previously discussed the effect of network depth measured on the macro level by considering the number of stacked layers, which is determined as a hyperparameter



(a) SoTL-DARTS-8 - normal cell

(b) SoTL-DARTS-20 - normal cell



(c) SoTL-DARTS-8 - reduction cell

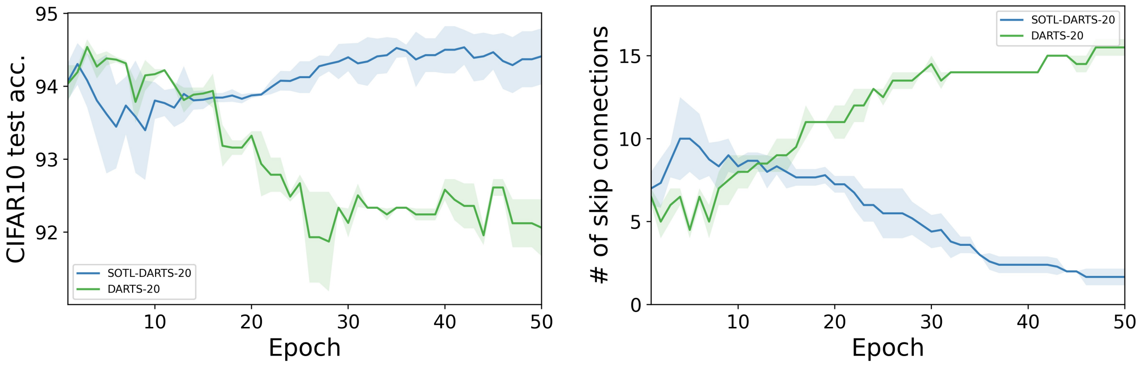
(d) SoTL-DARTS-20 - reduction cell

Figure 4.11: Overview of the best normal and reduction cells found by SoTL-DARTS-8 and SoTL-DARTS-20 on the DARTS search space.

outside of the search. We now also demonstrate that SoTL-DARTS has surprisingly different biases from DARTS even within the topology of an individual cell, which is determined by the search itself.

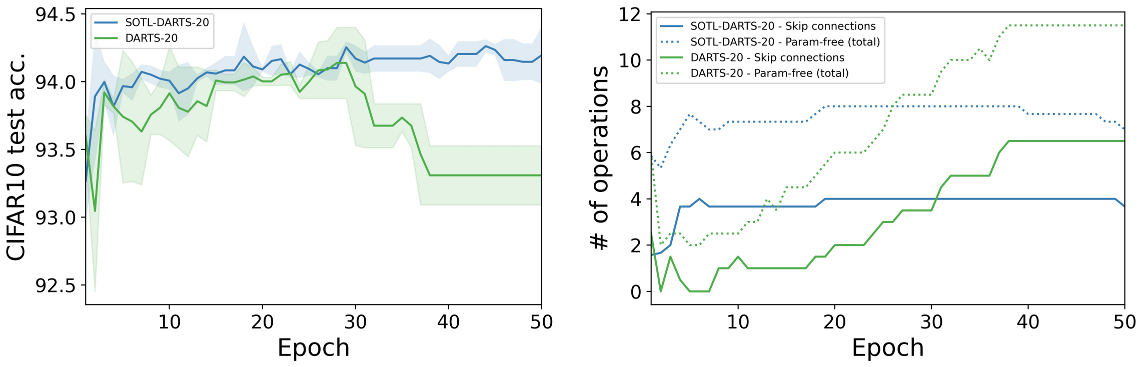
Shu et al. (2020) show that weight-sharing NAS algorithms including DARTS tend to prefer shallow rather than deep architectures. This behavior is explained by shallow architectures having smoother loss landscapes, which makes them easier to train and thus easier to find with DARTS because it tries to optimize loss one step ahead. However, deeper architectures might have stronger final generalization performance despite being harder to train (He et al., 2016; Li et al., 2018a). Likewise, Zhou et al. (2020) have shown both theoretically and empirically that architectures with more skip connections tend to be easier to train initially but plateau at higher loss than their more parameter-heavy counterparts. Shallow architectures with lots of skip connections should therefore train especially fast, which can potentially explain why the greedy gradient-based search in DARTS overfits to them.

Following the same intuition, optimizing the training rather than validation loss in the architecture updates of DARTS should further exacerbate the bias towards skip connections and shallow models provided that the reason behind the bias is that such architectures train faster. However, Figure 4.13 shows that SoTL-DARTS selects significantly deeper architectures than baseline DARTS. Our methodology for



(a) S2 - test acc.

(b) S2 - # of skip



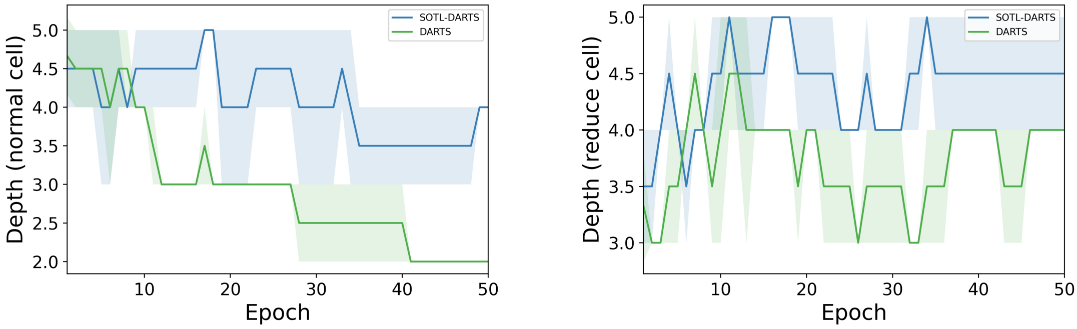
(c) DARTS space - test acc.

(d) DARTS space - # of skip and param-free ops

Figure 4.12: Comparison of the predicted performances on the S2 and DARTS search spaces for DARTS-20 and SoTL-DARTS-20. While increasing the number of layers makes the skip connection collapse worse for DARTS, we find that it improves the search for SoTL-DARTS and results in a moderate amount of skip connections.

investigating the cell topology is the same as proposed by Shu et al. (2020), who define the cell depth as the length of the longest path from the input to output nodes in the cell. Moreover, we noted in Section 4.3.2 that SoTL-DARTS only rarely picks any skip connections throughout the whole search, and it generally finishes the search with very few or zero parameter-free operations overall. This contradicts the notion that skip connections are so prominent because they increase the training speed. As an additional point of interest, we note from Figure 4.11 that the 20 layers search tends to pick shallower cells compared to the 8 layers one. The low 8 layers depth, which is set as a hyperparameter, appears to get compensated by the search preferring deeper individual cells.

In fact, the poor performance of SoTL-DARTS on NASBench-1shot1 is also caused by finding architectures with excessive depth. Figure 4.14 compares the depth, ac-



(a) DARTS space - Normal cell (b) DARTS space - Reduction cell

Figure 4.13: Comparison of the depth trajectory between DARTS and SoTL-DARTS search on the DARTS search space. While baseline DARTS is biased towards shallow architectures, architectures found by SoTL-DARTS are significantly deeper throughout the whole search for both normal and reduction cells shown in a) and b), respectively. The range of depth in the DARTS space is [2, 5], hence DARTS attains the minimum depth possible in the normal cell.

curacy of selected architectures and operation choices for DARTS and SoTL-DARTS on NB101-3. While both algorithms choose architectures with conv 3x3 as the most frequent operation, DARTS ultimately converges towards a topology with minimum depth while SoTL-DARTS instead has maximum depth. The high depth causes the CIFAR10 accuracy of the final architecture selected by SoTL-DARTS to be below 89% in NB101-3, but simply making the architecture shallower would have been enough to get the accuracy above 93%. This is the case for DARTS at around epoch 25 in Figure 4.14 when all the seeds have chosen 5x conv 3x3 out of 5 total operation choices, and the average performance is 92.9% because the architecture is shallow. SoTL-DARTS also chooses 5x conv 3x3, but because the architecture gets deeper over time, the performance keeps decreasing.

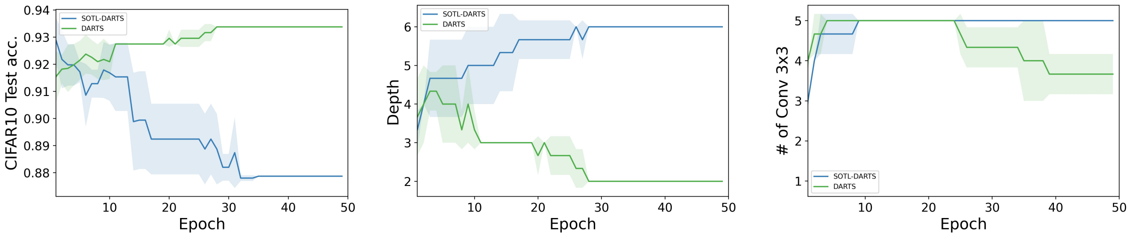
However, we note that the NASBench-101 test accuracies might be potentially misleading because all the architectures were trained for 108 epochs only. Deeper architectures are known to require longer training schedules (He et al., 2016; Zhou et al., 2020), and the performance of the deep architectures found by SoTL-DARTS might thus be understated by the benchmark. Verifying whether the NASBench-101 parameter settings consistently favor shallow cells would be an interesting question for future work.

Nonetheless, it appears that optimizing SoTL rather than validation loss consistently makes the cell topology deeper rather than wider, which is in contrast to the predictions made by Shu et al. (2020). Interestingly, it also shows that the overfitting problem in the DARTS-family of algorithms is not necessarily limited to the operations set but also the topology. However, mainstream NAS research generally ignores concerns about cell topology as the focus tends to be on preventing the skip connection collapse that is so prominent in normal DARTS. In particular, we suspect that the topology issue is caused by the edge selection in DARTS being optimized only very indirectly. To recapitulate the process, each node in the cell DAG retains the top-2 incoming edge operations. However, the incoming multi-edges from different nodes of the graph do not directly compete with each other in any way. Meanwhile, the within-multi-edge operations are all engaged in the architecture softmax from Eq. (3.8). Therefore, it is not obvious whether the highest weight incoming edges are globally the most important because there is no mechanism to enforce that behavior. We refer to Figure 3.4 for an intuitive visualization of how the multi-edges have local within-multi-edge competition but no global across-multi-edge competition. Hence even if SoTL-DARTS can consistently improve the within-multi-edge operation selection by preventing skip connection collapse, this might lead to unpredictable and potentially suboptimal changes in the cell topology.

Several recently proposed DARTS variants can be interpreted in the context of implicitly improving cell topology. Doing DARTS architecture selection via iterative pruning (Wang et al., 2021) can be understood as a way of performing topology-aware architecture selection since the supernetwork jointly adapts to the changes in operation set and topology during the pruning. This is unlike in normal DARTS, in which the argmax selection magnitude prunes the whole network at once. Other recently proposed NAS optimization methods that encourage edge sparsity such as GAEA (Li et al., 2021) or use different architecture encoding and selection procedure, such as FairDARTS (Chu et al., 2020), can also be seen as influencing the topology selection. Whether it would be possible to directly treat the within-cell depth bias of SoTL-DARTS by applying some of those methods is an exciting direction for future work.

4.3.5 Eigenvalues of the architecture Hessian

Zela et al. (2019) argue that the skip connection overfitting and general deterioration of the DARTS selected architectures is related to the growing dominant eigenvalues of the architecture Hessian computed on the validation set. Intuitively, the increasing architecture eigenvalues mean that the found solutions are in progressively sharper regions. Higher sharpness should increase the disparity between the supernetwork performance and the performance of discretized architectures. This is because the



(a) CIFAR10 test accuracy

(b) Depth

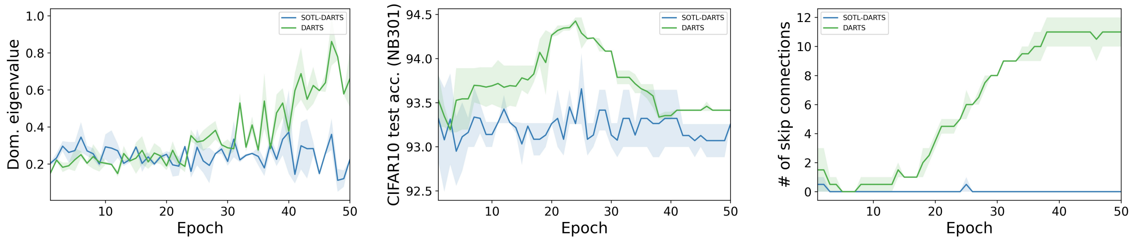
(c) # of 3 × 3 convolutions

Figure 4.14: a) SoTL-DARTS only gets worse over time on NB101-3 whereas baseline DARTS achieves top tier performance. In b) and c), we show that the root cause is the excessively deep cell topology in SoTL-DARTS since both algorithms select 3×3 conv for majority of the 5 operation choices.

network accuracy with respect to perturbing the architecture parameters changes more quickly in sharper regions, and the discretization can be understood simply as a large perturbation, in which we set most of the architecture coefficients to zero.

However, this intuition holds primarily for the case where we evaluate discretized architectures using the weights inherited from the supernetwork. The usual DARTS evaluation protocol instead retrains the selected architecture from scratch. The architecture selection via argmax is by itself so non-smooth that local curvature might yield no substantial information on the ground truth performance of selected architectures. Nonetheless, the efficiency of several methods such as SDARTS (Chen et al., 2020b) or DrNAS (Chen et al., 2021) has been justified by them regularizing the growth of architecture eigenvalues.

Specifically, Zela et al. (2019) were the first to propose the reduced search space $S2 = \{\textit{skip connection}, 3 \times 3 \textit{ sep. conv.}\}$. Using this search space, they showed that the architecture overfitting coincides with a rise in eigenvalues. First, we reproduce this result and then extend it to the full DARTS search space. For SoTL-DARTS, we measure the architecture eigenvalues on the training set as no validation set is used for the training. Figure 4.15 shows the evolution of the dominant eigenvalues and predicted performance via NASBench-301 for both DARTS and SoTL-DARTS on the S2 space. We previously observed that SoTL-DARTS chooses no skip connections whatsoever. Now, we also see that its architecture eigenvalues do not increase, and the performance of selected architectures shows no sign of architecture overfitting even at the end of the search. Meanwhile, normal DARTS picks 11 out of 16 total operation choices as skip connections, and its performance deteriorates along with exploding eigenvalues. However, the predicted performance of SoTL-DARTS is actually quite low because having at least some skip connections would be helpful. That said, we



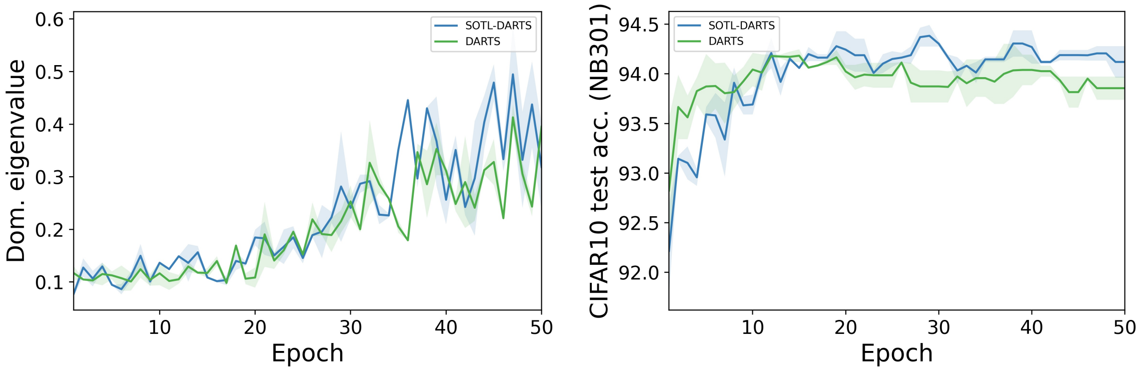
(a) S2 - dom. eigenvalue (b) S2 - CIFAR10 test acc. (c) S2 - # of skip

Figure 4.15: a) The dominant eigenvalue of the architecture Hessian $\nabla^2_{\alpha\alpha} L_{val}$ rises during search on the S2 search space for DARTS, but not for SoTL-DARTS. In b) and c), we see that SoTL-DARTS does not overfit to skip connections and its selected architecture performance does not deteriorate.

have already shown that SoTL-DARTS picks more skip connections when the search architecture is deeper in Section 4.3.3.

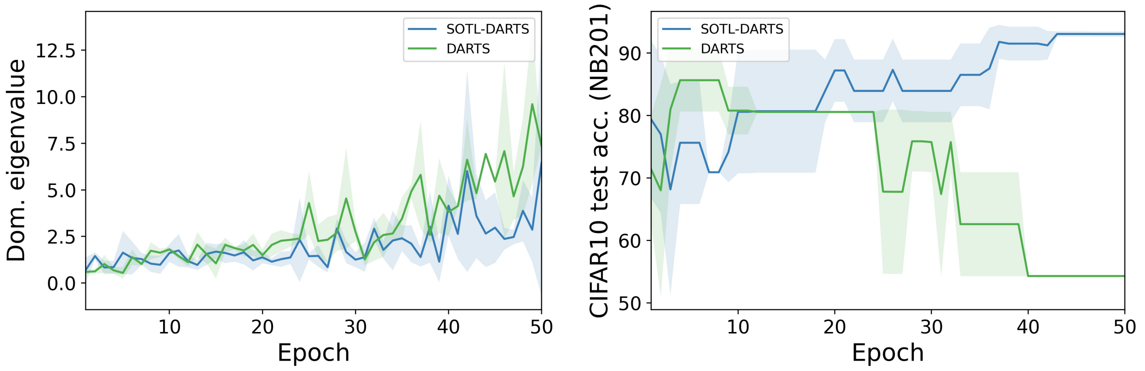
Figure 4.16 shows the same experiment on the full DARTS search space. This time, the growth of the dominant architecture eigenvalue is even stronger for SoTL-DARTS than for normal DARTS. Despite that, the SoTL-DARTS search achieves a strong performance that does not decrease over training. We were not able to verify by retraining that the performance is not getting worse for SoTL-DARTS, and we only show the search trajectory provided by NASBench-301. However, we already saw in Table 4.8 that the test set performance of the final SoTL-DARTS architecture is better than baseline DARTS. Furthermore, Figure 4.16 also shows the same experiment on NASBench-201. The eigenvalues again rise for both algorithms during training, but the performance of SoTL-DARTS peaks at the end of training, whereas the performance of DARTS keeps declining until it overfits to all skip connections. Finally, Figures 4.16e and 4.16f show that on NASBench-1shot1, the eigenvalues go down for SoTL-DARTS and up for DARTS, and the final architecture performance has exactly the same trend.

Considering the results of both DARTS and SoTL-DARTS in tandem, it appears that the negative correlation between architecture eigenvalues and performance of selected architecture might be spurious. We saw that the correlation can occur independently of whether the search overfits to skip connections, and it does not seem to necessarily indicate a drop in architecture performance. However, based on only the results of normal DARTS, it might indeed appear that the eigenvalues are negatively correlated with the final performance, and using only the results of SoTL-DARTS might potentially suggest a positive correlation. Therefore, it is likely that the eigenvalues are independent of the architecture overfitting.



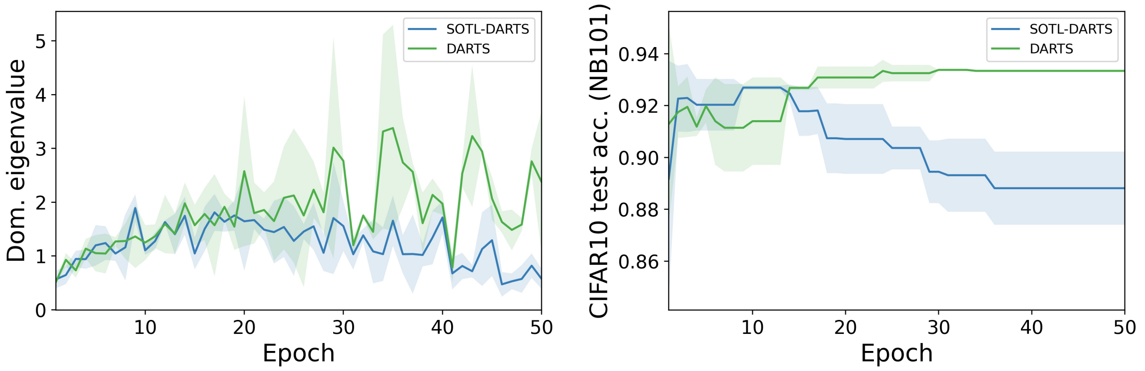
(a) DARTS space - dom. eigenvalue

(b) DARTS space - CIFAR10 test acc.



(c) NB201 - dom. eigenvalue

(d) NB201 - CIFAR10 test acc.



(e) NB101-3 - dom. eigenvalue

(f) NB101-3 - CIFAR10 test acc.

Figure 4.16: a) and c) show that the dominant eigenvalue of the architecture Hessian $\nabla^2_{\alpha\alpha} L_{val}$ rises during training for both DARTS and SoTL-DARTS on the DARTS space and NASBench-201. However, b) and d) demonstrate that SoTL-DARTS performance keeps increasing whereas DARTS starts overfitting to skip connections. e) and f) show the results on NASBench-1shot1, where the performance of SoTL-DARTS is weak and its eigenvalues simultaneously decrease. Meanwhile, DARTS has good performance with increasing eigenvalues.

Chapter 5

Conclusion

This thesis investigated the possibility of applying Sum-over-Training-Losses (SoTL), a new generalization estimator with a principled Bayesian model selection interpretation, to weight-sharing neural architecture search (NAS). While the theoretical background of the estimator only strictly applies to linear models, it has previously been demonstrated to be useful for NAS in the non-weight-sharing setting. We extended those results by showing that the estimator can be applied as a drop-in replacement for validation set metrics in the most popular weight-sharing NAS algorithms including SPOS (Guo et al., 2020) and DARTS (Liu et al., 2018).

We performed several experiments to prove the efficacy of our proposed methods. First, we designed synthetic benchmarks that either mirror the theoretical guarantees of SoTL or replicate small-scale experiments from gradient-based hyperparameter optimization. In both, we have shown that SoTL significantly outperforms validation loss as a metric to optimize for.

Next, we reached similar conclusions when applying SoTL to architecture selection in the final phase of SPOS, in which architectures are ranked by their validation accuracy to find the top-1 best architecture. We replaced evaluation via validation accuracy by a short training phase to collect SoTL, which was then used to rank the candidate architectures. This was enough to significantly improve the performance across multiple benchmarks from the NASBench series. Similarly, SoTL-DARTS was shown to have significantly stronger performance than baseline DARTS across a suite of benchmarks, and optimizing SoTL alleviated many long-standing problems known to occur in DARTS, such as the skip connection collapse. For DARTS specifically, we developed an approximation to the exact SoTL gradient that performs well despite being computationally cheap.

Moreover, our results uncovered several intriguing observations about the biases in weight-sharing NAS. For the SPOS family of algorithms, we have shown that ranking disorder is not necessarily the primary obstacle in one-shot NAS as most related work assumes. Instead, our results suggest there is a distinct difference between ranking median and top-k architectures. This idea has seen a surge in interest recently, and we contributed a novel perspective because other work generally does not do any finetuning of the supernetwork weights. We have also described a novel interpretation of one-shot NAS with SoTL as meta-learning and shown that simply reusing the existing meta-learning algorithms is competitive with the performance of specialized NAS algorithms. Furthermore, we have shown that the meta-learning algorithms can be integrated into state-of-the-art one-shot NAS algorithms to increase their performance further.

We also found that SoTL-DARTS behaves differently compared to baseline DARTS. Prior work has tried to explain the behavior of DARTS by appealing to the properties of its Hessian spectrum or identifying its bias towards shallow architectures. However, the observations in prior literature do not hold for SoTL-DARTS, and the predictions made therein sometimes even contradict the results obtained with SoTL-DARTS in practice. Interestingly, while overfitting to skip connections is a major pain point of DARTS, SoTL-DARTS is completely free of this problem. We also investigated the role of depth and topology in the DARTS search and demonstrated specific examples of topology overfitting. Topology is often ignored in differentiable NAS, perhaps because the skip connection collapse in normal DARTS overshadows it.

In summary, we have shown that SoTL is practically useful for weight-sharing NAS, trivial to implement and often even cheaper compute-wise than existing approaches. It can be applied to most popular NAS algorithms as a drop-in replacement for validation set metrics. It has no sensitive hyperparameters, and using it tends to stabilize the search significantly. We also identified several promising research directions including the usage of supernetwork finetuning to diagnose weight-sharing biases, usage of general meta-learning algorithms for one-shot NAS training, and explicitly accounting for the role of depth and architecture topology in DARTS. Those remain as exciting extensions for future work.

Appendices

Appendix A

SoTL-DARTS implementation

We showcase an example PyTorch implementation of Algorithm 3 (SoTL-DARTS) to demonstrate that SoTL is very easy to implement on top of standard training loops. The most important part is on lines 12-17, which show that it is trivial to accumulate the approximate SoTL gradient by not zeroing out the architecture gradients after each weight update. All implementations of SoTL-DARTS variants proceed analogously to this example.

```
 1 def train_sotl(train_queue, network, criterion, w_optimizer, a_optimizer, T=100):
 2     train_iter = iter(train_queue)
 3     network.train()
 4 
 5     for unrolling_step in range(math.ceil(len(train_queue)/T)):
 6       # format_input_data outputs a list of T (input, output) pairs
 7       all_base_inputs, all_base_targets = format_input_data(train_iter, T=T)
 8       network.zero_grad()
 9       model_init = deepcopy(network.state_dict()) # Save weights before unrolling so they can be restored later
10 
11       # Step 1 of Algorithm 3 - do the unrolling over 100 steps to collect SoTL gradient
12       for (base_inputs, base_targets) in zip(all_base_inputs, all_base_targets):
13           logits = network(base_inputs)
14           base_loss = criterion(logits, base_targets)
15           base_loss.backward()
16           w_optimizer.step() # Train the weights during unrolling as normal,
17           w_optimizer.zero_grad() # but the architecture gradients are not zeroed during the unrolling
18 
19 
20       # Step 2 of Algorithm 3 - update the architecture encoding using accumulated gradients
21       a_optimizer.step()
22 
23       a_optimizer.zero_grad() # Reset to get ready for new unrolling
24       w_optimizer.zero_grad()
25 
26       new_arch_params = deepcopy(network.arch_params) # Temporary backup for new architecture encoding
27 
28       network.load_state_dict(model_init) # Old weights are loaded, which also reverts the architecture encoding
29       for p1, p2 in zip (network.arch_params, new_arch_params):
30         p1.data = p2.data
31 
32       # Step 3 of Algorithm 3 - training weights after updating the architecture encoding
33       for (base_inputs, base_targets) in zip(all_base_inputs, all_base_targets):
34         logits = network(base_inputs)
35         base_loss = criterion(logits, base_targets)
36         base_loss.backward()
37         w_optimizer.step()
```

```

        w_optimizer.zero_grad()
        a_optimizer.zero_grad()
```

Appendix B

NASBench default hyperparameters

We briefly summarize the most important hyperparameter settings relevant to both one-shot NAS search and DARTS in our experiments. Note that all the search spaces use very similar defaults, which were originally designed by Liu et al. (2018). Furthermore, one-shot and DARTS share most of the parameters. The parameters for architecture optimizers are applicable to DARTS only.

	NASBench-201	NASBench-1shot1	DARTS space
Batch size	64	64	64
LR scheduler	Cosine	Cosine	Cosine
LR max	0.025	0.025	0.025
LR min	0.001	0.001	0.001
Optimizer	SGD	SGD	SGD
Weight decay	0.0005	0.0003	0.0003
Momentum	0.9	0.9	0.9
Epochs	100 (one-shot) 50 (DARTS)	50	100 (one-shot) 50 (DARTS)
Arch. LR	0.0003	0.0003	0.0003
Arch. weight decay	0.001	0.001	0.001
Arch. optimizer	Adam	Adam	Adam

Table B.1: Summary of hyperparameters for both one-shot NAS search and DARTS across all the benchmark search spaces we used in this work.

Bibliography

Abdelfattah, Mohamed S. et al. (2021). “Zero-Cost Proxies for Lightweight NAS”. In: *International Conference on Learning Representations (ICLR)*.

Arora, Sanjeev et al. (July 3, 2018). “Stronger Generalization Bounds for Deep Nets via a Compression Approach”. In: *International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 2640-3498. PMLR, pp. 254–263.

Baker, Bowen et al. (Nov. 8, 2017). “Accelerating Neural Architecture Search using Performance Prediction”. In: *arXiv:1705.10823 [cs]*. arXiv: `1705.10823`.

Bartlett, Peter L., Dylan J. Foster, and Matus J. Telgarsky (2017). “Spectrally-normalized margin bounds for neural networks”. In: *Advances in Neural Information Processing Systems* 30.

Baydin, Atilim Gunes et al. (2018a). “Automatic Differentiation in Machine Learning: a Survey”. In: *Journal of Machine Learning Research* 18.153, pp. 1–43. ISSN: 1533-7928.

Baydin, Atilim Gunes et al. (2018b). “Online Learning Rate Adaptation with Hypergradient Descent”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Bender, Gabriel et al. (2018). “Understanding and simplifying one-shot architecture search”. In: *International Conference on Machine Learning*. PMLR, pp. 550–559.

Benyahia, Yassine et al. (2019). “Overcoming multi-model forgetting”. In: *International Conference on Machine Learning*. PMLR, pp. 594–603.

Bergstra, James and Yoshua Bengio (2012). “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13.10, pp. 281–305. ISSN: 1533-7928.

Breiman, Leo (Oct. 1, 2001). “Random Forests”. In: *Machine Learning* 45.1, pp. 5–32. ISSN: 1573-0565. DOI: `10.1023/A:1010933404324`.

Brown, Tom et al. (2020). “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems* 33, pp. 1877–1901.

Cai, Han, Ligeng Zhu, and Song Han (2019a). “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware”. In: *International Conference on Learning Representations*.

Cai, Han et al. (2019b). “Once-for-All: Train One Network and Specialize it for Efficient Deployment”. In: *International Conference on Learning Representations*.

Chen, Mark et al. (2020a). “Generative pretraining from pixels”. In: *International Conference on Machine Learning*. PMLR, pp. 1691–1703.

Chen, Xiangning and Cho-Jui Hsieh (2020b). “Stabilizing differentiable architecture search via perturbation-based regularization”. In: *International Conference on Machine Learning*. PMLR, pp. 1554–1565.

Chen, Xiangning et al. (2021). “Dr{NAS}: Dirichlet Neural Architecture Search”. In: *International Conference on Learning Representations*.

Chen, Xin et al. (2019). “Progressive differentiable architecture search: Bridging the depth gap between search and evaluation”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 1294–1303.

Chu, Xiangxiang et al. (2019). “Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search”. In: *arXiv preprint arXiv:1907.01845*.

Chu, Xiangxiang et al. (2020). “Fair darts: Eliminating unfair advantages in differentiable architecture search”. In: *European Conference on Computer Vision*. Springer, pp. 465–480.

Convolutional neural network (Aug. 4, 2021). In: *Wikipedia*. Page Version ID: 1037144532.

Deng, Jia et al. (June 2009). “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009 IEEE Conference on Computer Vision and Pattern Recognition. ISSN: 1063-6919, pp. 248–255. DOI: `10.1109/CVPR.2009.5206848`.

Domhan, Tobias, Jost Tobias Springenberg, and Frank Hutter (2015). “Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves”. In: *Twenty-fourth international joint conference on artificial intelligence*.

Dong, Xuanyi and Yi Yang (2019a). “One-shot neural architecture search via self-evaluated template network”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3681–3690.

— (2019b). “Searching for a robust neural architecture in four gpu hours”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1761–1770.

— (2020). “NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search”. In: *International Conference on Learning Representations (ICLR)*.

Dong, Xuanyi et al. (2021). “NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*. DOI: `10.1109/TPAMI.2021.3054824`.

Dosovitskiy, Alexey et al. (June 3, 2021). “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *arXiv:2010.11929 [cs]*. arXiv: `2010.11929`.

Finn, Chelsea, Pieter Abbeel, and Sergey Levine (July 17, 2017). “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks”. In: *International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 2640-3498. PMLR, pp. 1126–1135.

Fisher, R. A. (1922). “The Goodness of Fit of Regression Formulae, and the Distribution of Regression Coefficients”. In: *Journal of the Royal Statistical Society* 85.4.

Publisher: [Wiley, Royal Statistical Society], pp. 597–612. ISSN: 0952-8385. DOI: `10.2307/2341124`.

Fisher, Ronald Aylmer (1938). *Statistical methods for research workers.* In collab. with Internet Archive. Edinburgh, Oliver and Boyd. 386 pp. ISBN: 978-0-05-002170-5.

Fort, Stanislav et al. (Mar. 13, 2020). “Stiffness: A New Perspective on Generalization in Neural Networks”. In: *arXiv:1901.09491 [cs, stat]*. arXiv: `1901.09491`.

Fu, Jie et al. (Apr. 6, 2016). “DrMAD: Distilling Reverse-Mode Automatic Differentiation for Optimizing Hyperparameters of Deep Neural Networks”. In: *arXiv:1601.00917 [cs]*. arXiv: `1601.00917`.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (Nov. 18, 2016). *Deep Learning.* Red. by Francis Bach. Adaptive Computation and Machine Learning series. Cambridge, MA, USA: MIT Press. 800 pp. ISBN: 978-0-262-03561-3.

Guo, Zichao et al. (2020). “Single path one-shot neural architecture search with uniform sampling”. In: *European Conference on Computer Vision.* Springer, pp. 544–560.

He, Chaoyang et al. (2020). “MiLeNAS: Efficient Neural Architecture Search via Mixed-Level Reformulation”. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 11993–12002.

He, Kaiming et al. (June 2016). “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). ISSN: 1063-6919, pp. 770–778. DOI: `10.1109/CVPR.2016.90`.

Jiang, Yiding et al. (Apr. 2020). “Fantastic Generalization Measures and Where to Find Them”. In: Eighth International Conference on Learning Representations.

Keskar, Nitish Shirish et al. (2017). “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* OpenReview.net.

Krizhevsky, Alex (May 8, 2012). “Learning Multiple Layers of Features from Tiny Images”. In: *University of Toronto.*

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1.* NIPS’12. event-place: Lake Tahoe, Nevada. Red Hook, NY, USA: Curran Associates Inc., pp. 1097–1105.

Lecun, Y. et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. DOI: `10.1109/5.726791`.

Lewkowycz, Aitor et al. (Mar. 4, 2020). “The large learning rate phase of deep learning: the catapult mechanism”. In: *arXiv:2003.02218 [cs, stat]*. arXiv: `2003.02218`.

Li, Hao et al. (2018a). “Visualizing the Loss Landscape of Neural Nets”. In: *Advances in Neural Information Processing Systems* 31.

Li, Liam and Ameet Talwalkar (2020). “Random search and reproducibility for neural architecture search”. In: *Uncertainty in Artificial Intelligence.* PMLR, pp. 367–377.

Li, Liam et al. (2021). “Geometry-Aware Gradient Algorithms for Neural Architecture Search”. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Li, Lisha et al. (June 18, 2018b). “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”. In: *arXiv:1603.06560 [cs, stat]*. arXiv: `1603.06560`.

Li, Yuanzhi, Colin Wei, and Tengyu Ma (2019). “Towards Explaining the Regularization Effect of Initial Large Learning Rate in Training Neural Networks”. In: *Advances in Neural Information Processing Systems* 32.

Liu, Hanxiao, Karen Simonyan, and Yiming Yang (2018). “DARTS: Differentiable Architecture Search”. In: *International Conference on Learning Representations*.

Long, Philip M. and Hanie Sedghi (Apr. 2020). “Generalization bounds for deep convolutional neural networks”. In: Eighth International Conference on Learning Representations.

Luketina, Jelena et al. (June 11, 2016). “Scalable Gradient-Based Tuning of Continuous Regularization Hyperparameters”. In: *International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 1938-7228. PMLR, pp. 2952–2960.

Lyle, Clare et al. (2020). “A Bayesian Perspective on Training Speed and Model Selection”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., pp. 10396–10408.

MacKay, David J. C. (2002). *Information Theory, Inference & Learning Algorithms*. USA: Cambridge University Press. ISBN: 0-521-64298-1.

Maclaurin, Dougal, David Duvenaud, and Ryan Adams (June 1, 2015). “Gradient-based Hyperparameter Optimization through Reversible Learning”. In: *International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 1938-7228. PMLR, pp. 2113–2122.

Matthews, Alexander G de G et al. (2017). “Sample-then-optimize posterior sampling for bayesian linear models”. In: *NIPS Workshop on Advances in Approximate Bayesian Inference*.

Mellor, Joseph et al. (2021). “Neural Architecture Search without Training”. In: *International Conference on Machine Learning*.

Mikolov, Tomas et al. (2013). “Distributed Representations of Words and Phrases and their Compositionality”. In: *Advances in Neural Information Processing Systems* 26.

Murphy, Kevin P. (Aug. 24, 2012). *Machine Learning: A Probabilistic Perspective*. Red. by Francis Bach. Adaptive Computation and Machine Learning series. Cambridge, MA, USA: MIT Press. 1104 pp. ISBN: 978-0-262-01802-9.

Mysid, Dake (Nov. 28, 2006). *A simplified view of an artifical neural network*.

Nair, Vinod and Geoffrey E. Hinton (June 21, 2010). “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Madison, WI, USA: Omnipress, pp. 807–814. ISBN: 978-1-60558-907-7.

Nakkiran, Preetum, Behnam Neyshabur, and Hanie Sedghi (Sept. 28, 2020). “The Deep Bootstrap Framework: Good Online Learners are Good Offline Generalizers”. In: International Conference on Learning Representations.

Neyshabur, Behnam, Ryota Tomioka, and Nathan Srebro (June 26, 2015). "Norm-Based Capacity Control in Neural Networks". In: *Conference on Learning Theory*. Conference on Learning Theory. ISSN: 1938-7228. PMLR, pp. 1376–1401.

Nichol, Alex, Joshua Achiam, and John Schulman (Oct. 22, 2018). "On First-Order Meta-Learning Algorithms". In: *arXiv:1803.02999 [cs]*. arXiv: `1803.02999`.

Osband, Ian, John Aslanides, and Albin Cassirer (2018). "Randomized Prior Functions for Deep Reinforcement Learning". In: *Advances in Neural Information Processing Systems* 31.

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems* 32.

Pham, Hieu et al. (July 3, 2018). "Efficient Neural Architecture Search via Parameters Sharing". In: *International Conference on Machine Learning*. International Conference on Machine Learning. ISSN: 2640-3498. PMLR, pp. 4095–4104.

Rasmussen, Carl Edward and Zoubin Ghahramani (Jan. 1, 2000). "Occam's Razor". In: *Proceedings of the 13th International Conference on Neural Information Processing Systems*. NIPS'00. Cambridge, MA, USA: MIT Press, pp. 276–282.

Real, Esteban et al. (2019). "Regularized evolution for image classifier architecture search". In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. Issue: 01, pp. 4780–4789.

Ru, Binxin et al. (June 8, 2020). "Revisiting the Train Loss: an Efficient Performance Estimator for Neural Architecture Search". In: *arXiv:2006.04492 [cs, stat]*. arXiv: `2006.04492`.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (Oct. 1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088. Bandiera_abtest: a Cg_type: Nature Research Journals Number: 6088 Primary_atype: Research Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: `10.1038/323533a0`.

Schölkopf, Bernhard et al. (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Google-Books-ID: y8ORL3DWt4sC. MIT Press. 658 pp. ISBN: 978-0-262-19475-4.

Shu, Yao, Wei Wang, and Shaofeng Cai (Apr. 2020). "Understanding Architectures Learnt by Cell-based Neural Architecture Search". In: Eighth International Conference on Learning Representations.

Siems, Julien et al. (Nov. 5, 2020). "NAS-Bench-301 and the Case for Surrogate Benchmarks for Neural Architecture Search". In: *arXiv:2008.09777 [cs]*. arXiv: `2008.09777`.

Simonyan, Karen and Andrew Zisserman (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*.

Singh, Prabhant et al. (2019). "A Study of the Learning Progress in Neural Architecture Search Techniques". In: *ArXiv* abs/1906.07590.

Smith, Samuel L. and Quoc V. Le (2018). "A Bayesian Perspective on Generalization and Stochastic Gradient Descent". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Smith, Samuel L. et al. (Sept. 28, 2020). “On the Origin of Implicit Regularization in Stochastic Gradient Descent”. In: International Conference on Learning Representations.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams (Aug. 29, 2012). “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *arXiv:1206.2944 [cs, stat]*. arXiv: `1206.2944`.

Snoek, Jasper et al. (July 13, 2015). “Scalable Bayesian Optimization Using Deep Neural Networks”. In: *arXiv:1502.05700 [stat]*. arXiv: `1502.05700`.

Vaswani, Ashish et al. (2017). “Attention is All you Need”. In: *Advances in Neural Information Processing Systems* 30.

Wang, Ruochen et al. (2021). “RETHINKING ARCHITECTURE SELECTION IN DIFFER-ENTIABLE NAS”. In: *International Conference on Learning Representations*.

Wu, Yuhuai et al. (2018). “Understanding Short-Horizon Bias in Stochastic Meta-Optimization”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Xie, Sirui et al. (2019). “SNAS: stochastic neural architecture search”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Xu, Yuhui et al. (2019). “PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search”. In: *International Conference on Learning Representations*.

Yang, Antoine, Pedro M. Esperança, and Fabio M. Carlucci (Apr. 2020). “NAS evaluation is frustratingly hard”. In: Eighth International Conference on Learning Representations.

Ying, Chris et al. (2019). “Nas-bench-101: Towards reproducible neural architecture search”. In: *International Conference on Machine Learning*. PMLR, pp. 7105–7114.

You, Shan et al. (2020). “Greedynas: Towards fast one-shot nas with greedy supernet”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1999–2008.

Yu, Jiahui and Thomas S. Huang (2019). “Universally Slimmable Networks and Improved Training Techniques”. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 1803–1811.

Yu, Jiahui et al. (2020a). “Bignas: Scaling up neural architecture search with big single-stage models”. In: *European Conference on Computer Vision*. Springer, pp. 702–717.

Yu, Kaicheng, Rene Ranftl, and Mathieu Salzmann (Sept. 28, 2020b). “How to Train Your Super-Net: An Analysis of Training Heuristics in Weight-Sharing NAS”. In:

Yu, Kaicheng et al. (Apr. 2020c). “Evaluating The Search Phase of Neural Architecture Search”. In: Eighth International Conference on Learning Representations.

Zela, Arber, Julien Siems, and Frank Hutter (Apr. 2020). “NAS-Bench-1Shot1: Benchmarking and Dissecting One-shot Neural Architecture Search”. In: Eighth International Conference on Learning Representations.

Zela, Arber et al. (2019). "Understanding and Robustifying Differentiable Architecture Search". In: *International Conference on Learning Representations.*

Zhang, Miao et al. (2020a). "Overcoming multi-model forgetting in one-shot nas with diversity maximization". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 7809–7818.

Zhang, Miao et al. (June 20, 2021a). "iDARTS: Differentiable Architecture Search with Stochastic Implicit Gradients". In: *arXiv:2106.10784 [cs].* arXiv: `2106.10784`.

Zhang, Yuge, Quanlu Zhang, and Yaming Yang (May 5, 2021b). "How Does Supernet Help in Neural Architecture Search?" In: *arXiv:2010.08219 [cs].* arXiv: `2010.08219`.

Zhang, Yuge et al. (2020b). "Deeper insights into weight sharing in neural architecture search". In: *arXiv preprint arXiv:2001.01431.*

Zhou, Pan et al. (2019). "Efficient Meta Learning via Minibatch Proximal Update". In: *Advances in Neural Information Processing Systems.* Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc.

Zhou, Pan et al. (2020). "Theory-Inspired Path-Regularized Differential Network Architecture Search". In: *Advances in Neural Information Processing Systems.* Vol. 33. Curran Associates, Inc., pp. 8296–8307.

Zoph, Barret et al. (2018). "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710.