Thèse n° 8035

# From Human-Designed Convolutional Neural Networks Towards Robust Neural Architecture Search

Présentée le 6 août 2021

Faculté informatique et communications
Laboratoire de vision par ordinateur
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

**Kaicheng YU**

Acceptée sur proposition du jury

Prof. J. R. Larus, président du jury
Prof. P. Fua, Dr M. Salzmann, directeurs de thèse
Prof. N. Lane, rapporteur
Dr Y. Jia, rapporteur
Prof. M. Jaggi, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

It seems probable that once the machine thinking method had started,
it would not take long to outstrip our feeble powers.
— Alan Turing

To my grandfather

# Acknowledgements

I still remembered the day I arrived in Switzerland on August 27th, 2016, being excited about the coming Ph.D. life at Lausanne. Throughout these five years at EPFL, I have learned many things that will impact the rest of my life. None of these can happen if EDIC did not pick me, a fresh undergraduate with little research experience, as a fellowship student at EPFL. I would like to express my gratitude to the committee members for putting emphasis on the potential rather than prior publications.

This work would not have been possible without my thesis advisor, Dr. Mathieu Salzmann. I have learned so much from him, from the basic scientific writing principles to constructive and friendly discussion throughout the Ph.D. time. Most importantly, when I switched my research direction in 2018, Mathieu put his trust in me and gave me the necessary freedom, autonomy, and full support, which I could not have found elsewhere.

I deeply appreciate all my collaborators for their support for my thesis, Dr. Rene Ranftl, Prof. Martin Jaggi, Dr. Claudiu Musat, Dr. Wang Wei, Mr. Yassine Benyahia and Mr. Christian Sciuto. This thesis cannot be finished without your support.

I would like to thank the members of my thesis committee, Professors Nicholas Lane, James Larus, and Yangqing Jia, who kindly accepting to evaluate my thesis, and brought their internationally recognized machine learning experiences and insights.

I would like to thank Dr. Ronan Boulic for being my mentor and helped me to get familiar with life in Switzerland. I appreciate Prof. Pascal Fua for allowing me to start the project in the computer vision lab. Here, I met many fantastic colleagues and have become lifetime friends with many of them. In particular, to Jan, Sena, and Victor, for being such good friends that are willing to offer their generous help. To the Chinese folks, Weizhe, Zheng, Yinlin, for the fun lunchtime and coffee breaks. To Shuxuan, for supporting me at the darkest time, without your company, I would not finish my thesis in such a smooth manner. I would like to thank Andrii, Artem, Bugra, Isinsu, Helge, Krzysztof, Krishna, Joachim, Mat, Vidit, and many others, for making the CVLab an unforgettable place.

I would like to express my special thanks to my friends. To Xiaowei Wang and Tianbai Li, for the interesting discussions that enlighten me at domains outside computer vision and machine learning. To Zhaowen Dong and Fei Mi, for being good buddies at Lausanne and making my life outside campus colorful. To Hao Xue and Yuxiang Yang, for helping me improving my ski technique. My thanks go to many friends at Lausanne, Tao Lin, Yugang Guo, Lingqing Jiang, Yao Di, Hui Chen, Lingjing Kong, Ruofan Zhou, for enriching my stay here.

Last, I would like to thank my family, my father Mr. Wanhui Yu, my mother Prof. Xuerong Cui.

They have been my best teachers and role models throughout my life. Without their genuine advice, I might stay in Hong Kong to be a programmer rather than choosing to concur a Ph.D. degree, where I contributed, even by a baby step, to the research community and push the boundary of human knowledge. Specifically, I would express my whole-heart gratefulness to my grandfather, Mr. Xuebin Yu, who unfortunately passed away during my study. This work is dedicated to you.

*Lausanne, May 20, 2021* Kaicheng Yu

# Abstract

Artificial intelligence has been an ultimate design goal since the inception of computers decades ago. Among the many attempts towards general artificial intelligence, modern machine learning successfully tackles many complex problems thanks to the progress in deep learning, and in particular in convolutional neural networks (CNN). To design a CNN for a specific task, one common approach consists of adapting the heuristics from the pre-deep-learning era to the CNN domain. In the first part of this thesis, we introduce two methods that follow this approach: i) We build a covariance descriptor, i.e., a local descriptor that is suitable for texture recognition, to replace the first-order fully connected layers in an ordinary CNN, showing that such a descriptor yields state-of-the-art performance on many fine-grained image classification tasks with orders of magnitude fewer feature dimensions; ii) we develop a light-weight recurrent U-Net for image semantic segmentation, inspired by the biological eye saccadic movements, that yields real-time predictions on devices with limited computational resources.

As most methods pre-dating automatic machine learning (AutoML), the two above-mentioned CNNs were human-designed. In the past few years, however, neural architecture search (NAS), which aims to facilitate the design of deep networks for new tasks, has drawn an increasing attention. In this context, the weight-sharing approach, which consists of utilizing a super-net to encompass all possible architectures within a search space, has become a de facto standard in NAS because it enables the search to be done on commodity hardware. In the second part of this thesis, we then provide an in-depth study of recent weight-sharing NAS algorithms. First, we discover a phenomenon in the weight-sharing NAS training pipeline, which we dub multi-model forgetting, that negatively impacts the super-net quality, and propose a statistically motivated approach to address it. Subsequently, we find that (i) on average, many popular weight-sharing NAS algorithms perform similarly to a random architecture sampling policy; (ii) the widely-adopted weight sharing strategy degrades the ranking of the NAS candidates to the point of not reflecting their true performance, thus reducing the effectiveness of the search process. We then further decouple weight sharing from the NAS sampling policy, and isolate 14 factors that play a key role in the success of super-net training. Finally, to improve the super-net quality, we propose a regularization term that aims to maximize the correlation between the performance rankings of the super-net and of the stand-alone architectures using a small set of landmark architectures.

**Keywords**: Convolutional neural network, AutoML, Neural architecture search, Weight-sharing, Super-net, Image Classification, Semantic Segmentation.

# Résumé

L'intelligence artificielle a été un objectif ultime depuis la création des ordinateurs il y a des décennies. Parmi les nombreuses tentatives d'intelligence artificielle générale, le machine learning moderne s'attaque avec succès à de nombreux problèmes complexes grâce aux progrès de l'apprentissage profond, et en particulier dans les réseaux de neurones convolutifs (CNN). Pour concevoir un CNN pour une tâche spécifique, une approche courante consiste à adapter les heuristiques de l'ère pré-deep learning au domaine des CNN. Dans la première partie de cette thèse, nous introduisons deux méthodes qui suivent cette approche : i) Nous construisons un descripteur sour forme de covariance, c'est-à-dire un descripteur local adapté à la reconnaissance de texture, pour remplacer les couches linéaires de premier ordre dans un CNN ordinaire, montrant qu'un tel descripteur donne des performances de pointe sur de nombreuses tâches de classification "fine-grained" d'images malgré son utilisation de repésentations de bien plus petite dimension ; ii) nous développons un U-Net récurrent léger pour la segmentation sémantique d'image, inspiré des mouvements saccadiques de l'oeil, qui produit des prédictions en temps réel sur des appareils aux ressources de calcul limitées.

Comme la plupart des méthodes antérieures au machine learning automatique (AutoML), les deux CNN mentionnés ci-dessus ont été conçus manuellement. Cependant, ces dernières années, la recherche d'architecture neuronale (NAS), qui vise à faciliter la conception de réseaux profonds pour de nouvelles tâches, a attiré une attention croissante. Dans ce contexte, l'approche de partage de poids, qui consiste à utiliser un super-net pour englober toutes les architectures possibles dans un espace de recherche, est devenue un standard de facto dans le NAS car elle permet de faire la recherche sur du matériel de base. Dans la deuxième partie de cette thèse, nous proposons une étude approfondie des récents algorithmes de NAS avec partage de poids. Tout d'abord, nous découvrons un phénomène dans le pipeline d'entraînement de NAS avec partage de poids, que nous appelons l'oubli multimodèle, qui a un impact négatif sur la qualité du super-net, et proposons une approche statistiquement motivée pour y remédier. Par la suite, nous constatons que (i) en moyenne, de nombreux algorithmes de NAS populaires fonctionnent de manière similaire á un échantillonnage d'architecture aléatoire ; (ii) la stratégie de partage de poids largement adoptée dégrade le classement des candidats NAS au point de ne pas refléter leur véritable performance, réduisant ainsi l'efficacité du processus de recherche. Nous découplons ensuite davantage le partage de poids de la stratégie d'échantillonnage de NAS et isolons 14 facteurs qui jouent un rôle clé dans le succès de l'entraînement d'un super-net. Enfin, pour améliorer la qualité du super-net, nous proposons un terme de régularisation qui vise à maximiser la corrélation entre les classements de performance du super-net et des architectures autonomes en utilisant un petit

ensemble d'architectures de référence.
**Keywords** : Réseau de neurones convolutifs, AutoML, recherche d'architecture neuronale, partage de poids, Super-net, classification d'images, segmentation sémantique.

# Contents

# List of Figures

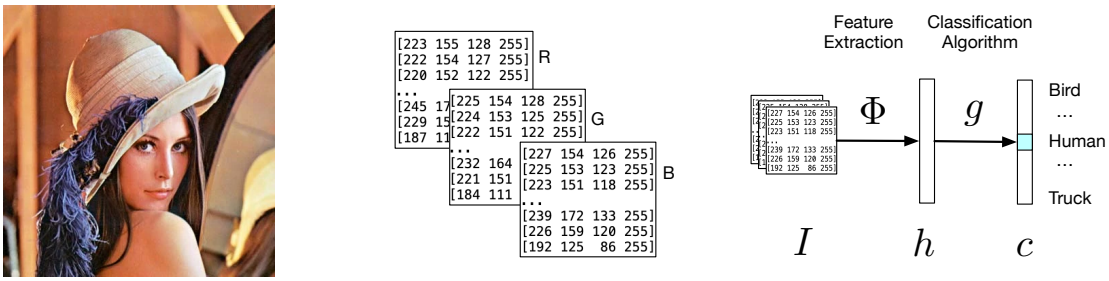
# List of Tables

# Introduction

Artificial intelligence has been one of the goals of the Computer Science community since the inception of computers. A popular approach consists of treating the computer like a digital brain, which handles input data, solves problems and draws conclusions as a human being. Similarly to us, computers should have the capability to understand the surrounding world captured in digital formats, such as images and audio signals. In particular, in this thesis, we focus on the process of tackling image-based problems, commonly referred to as visual recognition, which includes image classification, object detection, and semantic segmentation.

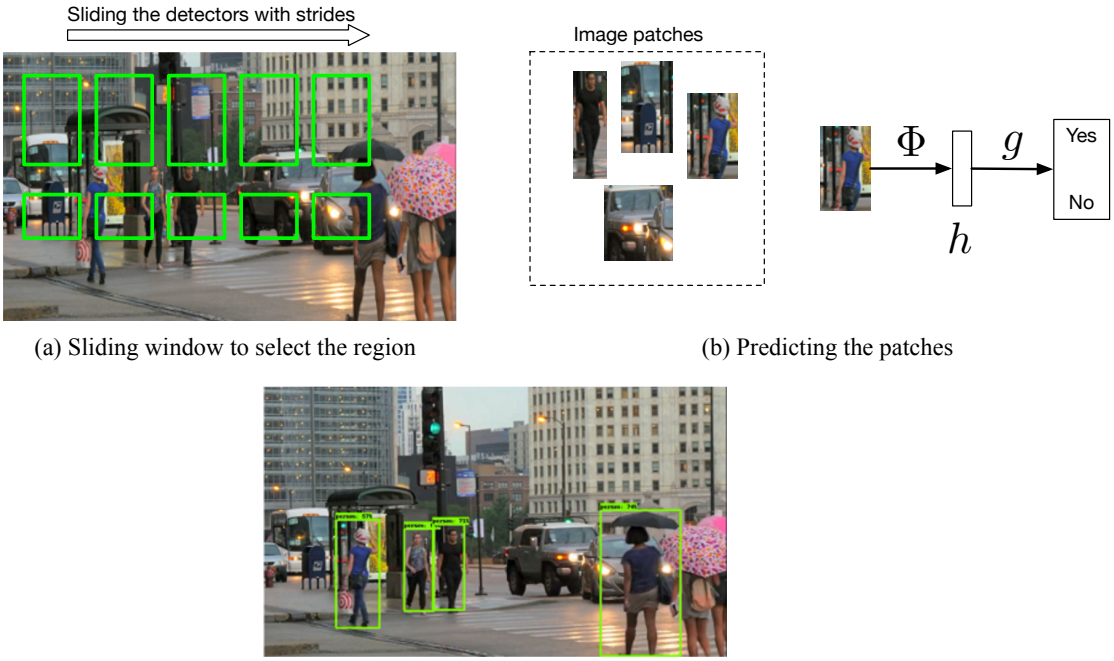
As illustrated in Figure 1, image classification aims to identify the object category depicted in one input image. In essence, this is achieved by constructing a *feature extractor* $\Phi$ that takes the image $I$ as input and outputs a *feature representation* $h$, which is usually a tensor or stacked matrices. These features are then passed to a classification algorithm $g$ that outputs a class label $c$. With digital cameras, the image is usually treated as a tensor in three dimension, one for the color channels, one for the height and one for the width. However, using the raw image as a feature is typically ineffective, because images differ widely from each other in terms of raw distance. As such, an important focus of visual recognition has been to finding an effective mapping $\Phi$ for the task of interest.

In this thesis, we study several approaches to extracting effective feature representations for different visual recognition problems via deep neural networks. Our work ranges from manually designing deep networks by drawing inspiration from traditional, handcrafted feature extraction




(a) Human perceptible image (b) Computer perceptible image (c) Image classification pipeline

Figure 1: **Illustration of image classification.**




(a) Sliding window to select the region

(b) Predicting the patches

(c) Output of the algorithm

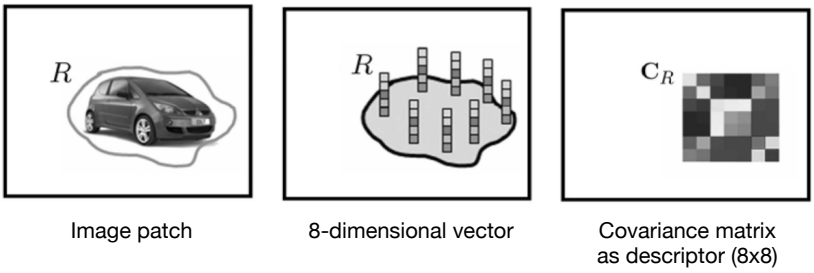
Figure 2: **Pedestrian detection pipeline.** (a) The sliding window will crop the original input image into various patches, and translate of detection into binary image classification problem. (b) As shown earlier, the algorithm need to first translate the image into a feature format, then classify the patches. (c) The output of the algorithm is a concatenation of positive patches.

that has been the common approach in computer vision for decades, to automatically learning the architecture of the deep network. To illustrate this, in this chapter, we first introduce a traditional approach as an example to demonstrate how to solve the task of pedestrian recognition. We then present the basic ideas behind neural networks, deep learning, and the design process of a neural network. Finally, we introduce the concept of automatic machine learning, which aims to automatically design an effective neural network for the task at hand.

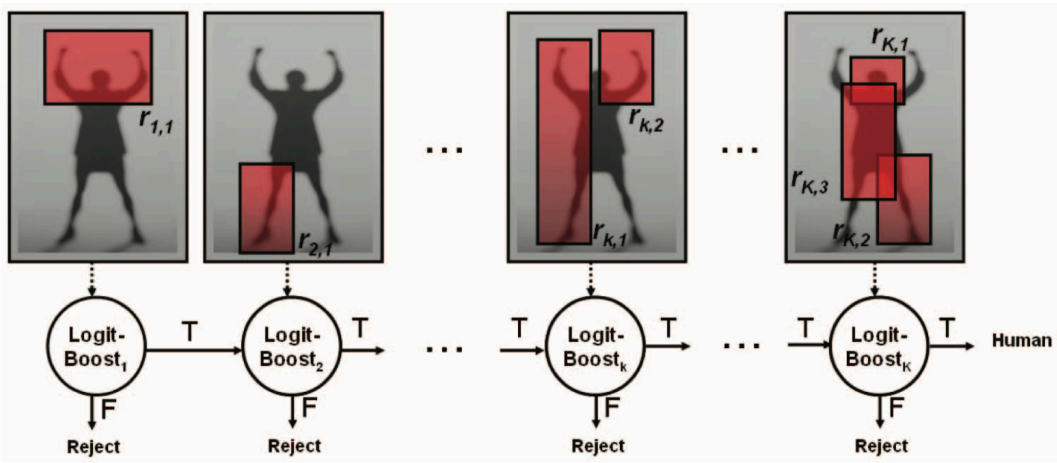
## Handcrafted Local Descriptors

Detecting pedestrians is a traditional computer vision task with two major challenges: i) pedestrians can be small and present at a high density in an image; ii) their appearance is highly diverse because of the articulated structure of the human body, the different poses taken and clothings worn by people, and the diversity of the environment they evolve in. As shown in Figure 2, a classical approach to pedestrian detection consists of sliding a window across the image, thus transforming the detection problem into a binary classification one.

To classify each patch of the original image, referred to as detection window, one needs to define appropriate $\Phi, g$ functions. A widely adopted approach consists of representing the image with




(a) Mapping the image patch into a covariance descriptor




(b) Cascaded classification algorithm of one detection window

Figure 3: **Pedestrian detection algorithm (Tuzel et al., 2007).** (a) It shows a sketch how to map a input image patch into a covariance matrix as feature. (b) A cascaded classification algorithm $g$ of one detection window. At each iteration, it randomly sample a few pre-defined patches of the input image crop (in red box) and compute the covariance descriptor. It then input the descriptors to a well-known Logit-Boost classifier (Friedman et al., 2000). If the majority of patches are predicted as pedestrian, it classify the window as pedestrian.

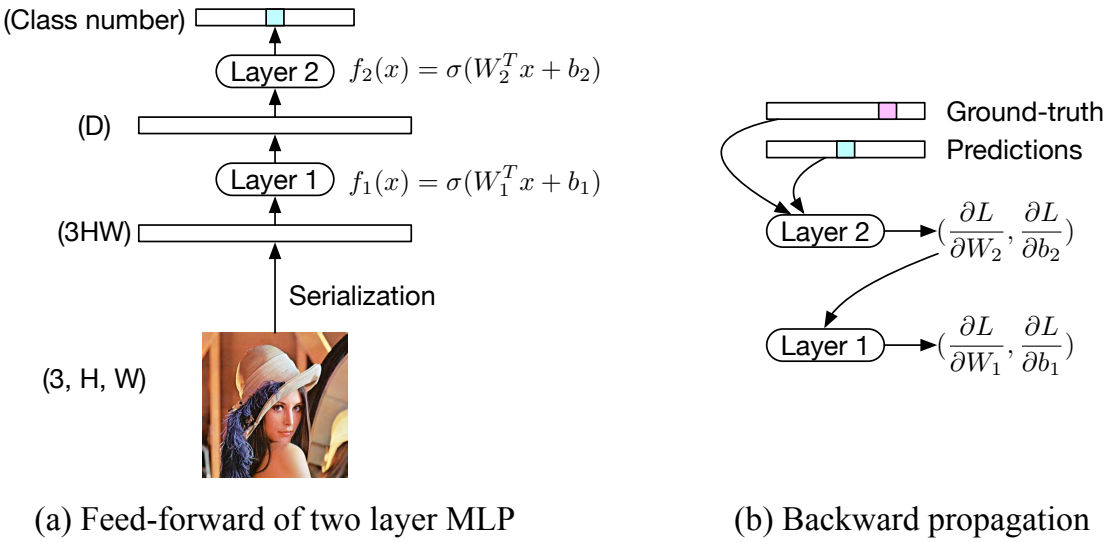
first order statistics, such as a histogram of features, typically related to image gradients along the x-y direction. One famous example is the Scale Invariant Feature Transform (Lowe, 2004), which computes the orientation histogram of keypoints across the entire image. Leveraging the observation that covariance-based representations have proven to be effective for texture recognition and that detecting body parts and texture bear similarities, Tuzel et al. (2007) introduced a second-order detector to replace the previous ones based on first-order representations. As shown in Figure 3, such a detector first extends the 1-dimensional greyscale image, $I \in \mathscr{R}^{(1,H,W)}$ to an 8-dimensional representation $R \in \mathscr{R}^{(8,H,W)}$, including the pixel location, first and second order derivatives of the intensity along the $x$ and $y$ axes. It then merges the height and width dimension to compute an $8 \times 8$ covariance matrix as representation of this image patch. By combining such representations with the well-known *LogitBoost* (Friedman et al., 2000) classification algorithm, Tuzel et al. (2007) formulate a cascaded weak classifier that identifies if a detection window contains a pedestrian. This method achieved the state-of-the-art pedestrian detection performance before the deep learning era.

Handcrafted descriptors such as the one discussed above have been the de-facto standard in computer vision until the emergence of deep learning. While they have many advantages, they suffer from one major drawback: the difficulty to generalize to a new task, even though it is highly similar to the one they were designed for. In other words, generalizing to a similar task is virtually as complex as designing a new descriptor. For example, the covariance-based method discussed above works well when the image is taken at human height, but may fail completely for images take from a drone. Deep learning has therefore emerged as an alternative, aiming to learn a general feature descriptor for visual recognition from training data.

## Deep Learning - Convolutional Neural Networks

Artificial neural networks have drawn drastic attention throughout the entire computer science history. One potential reason for this overwhelming hope is the elegant universal approximation theory. Pinkus (1999) proves that the multilayer feedforward perceptron (MLP), a common artificial neural network, has the capability to be a universal approximator for any continuous functions $f$.

In Figure 4, we show a simple example of MLP. It first serializes the image tensor into a 1-dimensional vector $x$. Each layer consists of two sets of parameters, $W$ and $b$, that can be updated by a gradient-based algorithm, and outputs $\sigma(W^T x + b)$, where $\sigma$ is the activation function. One can leverage the backward propagation algorithm to compute the gradients to update the parameters. To effectively handle an image as input, LeCun et al. (1989) proposed the first convolutional neural network, LeNet, to classify handwritten digits. As shown in Figure 5 (a), instead of applying a weight matrix to the entire image, the convolutional operation relies on a small kernel matrix that is convolved with the image in a sliding window manner, so that local in formation can be effectively preserved.




(a) Feed-forward of two layer MLP (b) Backward propagation

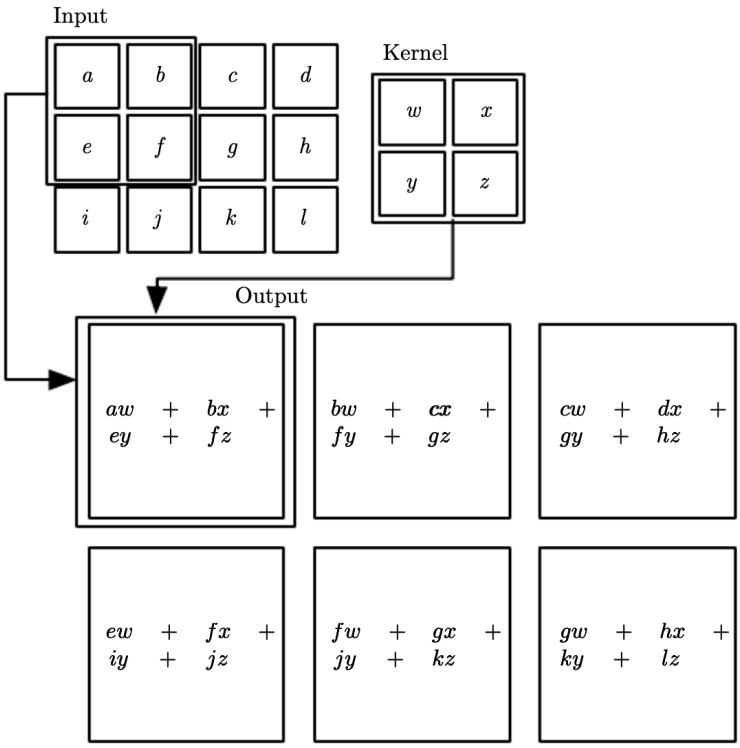
Figure 4: **Example of two layer multi-layer perception** (a) Each of the layer consists of mapping $f(x) = \sigma(W^T x + b)$, where $W, b$ are the parameters that can be updated, and $sigma$ is usually called the activation function to introduce non-linearity of MLP. (b) shows the *backprop* algorithm to compute gradients of each parameter, to perform gradient based algorithms. Note that each layer only need the output of previous layers. These two steps essentially capture the fundamental operation of the modern deep networks.

However, due to the limitation of computer resources, the power of neural networks was not fully revealed until AlexNet was proposed in 2012 (Krizhevsky et al., 2012). With the help of GPU for parallel computing, AlexNet achieves a top-5 classification error of 15.3% on the large scale ImageNet benchmark, containing 1000 object classes. This was more than 10.8 percentage points lower than its SVM competitor. Since then, CNNs have spread to virtually all computer vision problems.

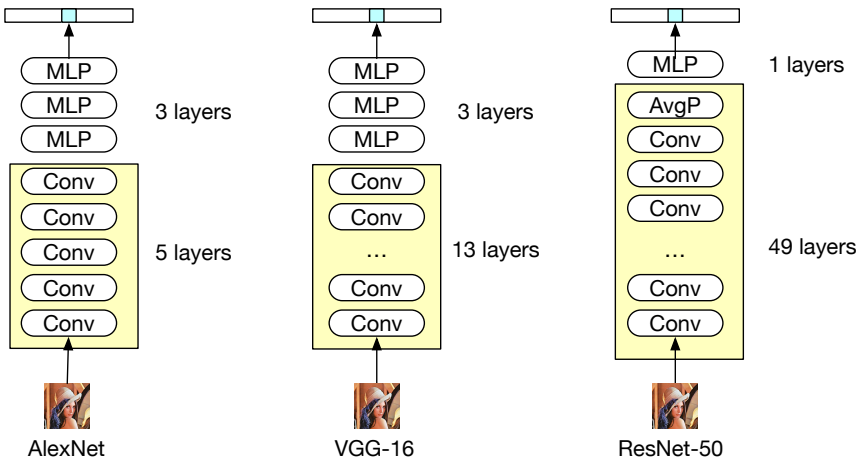
## Automatic Machine Learning - Neural Architecture Search

Considering that the development of deep learning led to improved generalization ability, the research community thus wondered if the network design task could further be automated. This resulted in the problem of neural architecture search, introduced by Zoph & Le (2017a), who proposed to construct a search space containing a variety of fundamental network elements, and to deploy a reinforcement learning agent to search for a promising architecture within this space. Since this pioneering work, NAS has received increasing attention, particularly because it has the potential to truly automate the feature extraction process, thus constituting a solid step towards real artificial intelligence.

However, the time complexity of the search remains impractical. Specifically, in the original paper, searching for a good model on CIFAR-10, which only contains 50,000 training images,




(a) 2D convolution to simplify the MLP for image-based inputs




(b) Modern human crafted convolutional neural networks

Figure 5: **Convolutional neural networks.** (a) (Goodfellow et al., 2016b) Unlike the previous multi-layer perceptron, which serializes the entire image into a vector and apply a weight matrix $W$, 2D convolution is an efficient replacement that consists a small kernel matrix (size usually smaller than 5 in most cases), that applies to the entire image in a sliding window manner. (b) It shows the sketch of modern handcrafted CNN which won the recent ImageNet challenge in the past few years, from the five layers AlexNet (Krizhevsky et al., 2012) to a deep 50 layers ResNet-50 (He et al., 2015). Nowadays, CNN based the ImageNet classifier achieves the top-1 accuracy over 78%, that surpass the human perception level by a significant margin.

took around 4000 GPU days. Pham et al. (2018b) then proposed a weight sharing regime that could effectively reduce the search cost by sharing the weights of all architectures, claiming similar accuracy as the original work while only requiring 4 GPU days. This marked a new era in automatic machine learning. However, as will be shown in the second part of this thesis, weight-sharing NAS suffers from many unsolved problems. In Chapters 4,5,6,7, we will present a systematic analysis of weight sharing NAS, and two solutions to improve its weaknesses.

## Thesis Organization

In this deep learning regime, the first part of this thesis tackles two such problems. In Chapter 1, we merged the previous covariance-based descriptors into the deep learning paradigm in an efficient manner, which drastically reduce the computational cost by order of magnitude comparing to the previous methods. In Chapter 2, we introduce a novel neural architecture targeting resource-constrained semantic segmentation. While, as will be discussed in Chapters 1, 2, the resulting methods have proven to be highly effective, designing neural networks for different tasks remains a challenging problem. In particular, achieving high performance and accuracy requires tedious architecture modifications and hyper-parameter tuning, both of which are tremendously time-consuming, and importantly, require previous experience.

In the second part of this thesis, we broadly focus on the topic of neural architecture search. In Chapter 4, we discover a novel phenomenon, multi-model forgetting, which negatively impact the training of shared networks under current NAS regime, and propose a statistically-justified solution to improve the performance. However, even we overcomes the multi-model forgetting in WS-NAS, the notorious reproducibility issue remains unsolved. In Chapter 5, we, for the first time with Li & Talwalkar (2019), points out the WS-NAS performance cannot surpass the basic random search algorithm. Furthermore, we reveal an hidden assumption of using weight sharing is violated in reality and causes such counter intuitive state. In Chapter 6, we further isolate fourteen factors of the super-net, that are agnostic to a specific search algorithm, and shows that tuning these can significantly improve the random search performance. In Chapter 7, we propose a novel landmark regularization that leverage the ground-truth ranking information of a small subset of landmark architectures, that can alleviate the ranking disorder issue of weight sharing NAS algorithms.

In Chapter 8, we will provide a conclusion of this thesis, and discuss the potential research directions in the future.

# Human Designed Convolutional Neural Networks

# Part I

In this part, we present two novel convolutional neural network architectures that tackle different computer vision problems.
First, in Chapter 1, we incorporate the covariance-based representation discussed in the introduction into a CNN. Such covariance-based representations, a.k.a. second-order pooling, or bilinear pooling, have proven effective for deep learning based visual recognition. However, the resulting second-order networks yield a final representation that is orders of magnitude larger than that of standard, first-order ones, making them memory-intensive and cumbersome to deploy. We introduce a general, parametric compression strategy that produces more compact representations than existing compression techniques, yet outperform both compressed and uncompressed second-order models. Our approach is motivated by a statistical analysis of the network's activations, relying on operations that lead to a Gaussian-distributed final representation, as inherently used by first-order deep networks.
Second, in Chapter 2, we observe that the state-of-the-art segmentation methods rely on very deep networks and are not always easy to train and tend to be relatively slow to run on standard GPUs. We therefore introduce a novel recurrent U-Net architecture that preserves the compactness of the original U-Net Ronneberger et al. (2015), while substantially increasing its performance to the point where it outperforms the baseline methods on several benchmarks.

# 1 Statistically-Motivated Second-Order Pooling

## 1.1 Introduction

As discussed in the Introduction, visual recognition is one of the fundamental goals of computer vision. Over the years, second-order representations, i.e., region covariance matrices as descriptors, have proven more effective than their first-order counterparts (Arandjelovic & Zisserman, 2013; Dalal & Triggs, 2005; Lazebnik et al., 2006; Perronnin et al., 2010) for many tasks, such as pedestrian detection (Tuzel et al., 2007), material recognition (Cimpoi et al., 2014) and semantic segmentation (Carreira et al., 2012). More recently, convolutional neural networks (CNNs) have achieved unprecedented performance in a wide range of image classification problems (Krizhevsky et al., 2012; He et al., 2016; Huang et al., 2017). Inspired by the past developments in handcrafted features, several works have proposed to replace the fully-connected layers with second-order pooling strategies, essentially utilizing covariance descriptors within CNNs (Lin et al., 2015; Ionescu et al., 2015; Li et al., 2017; Lin & Maji, 2017). This has led to second-order or bilinear CNNs whose representation power surpasses that of standard, first-order ones.

One drawback of these second-order pooling CNNs is that vectorizing the covariance descriptor to pass it to the classification layer, as done in (Lin et al., 2015; Ionescu et al., 2015; Li et al., 2017; Lin & Maji, 2017), yields a vector representation that is orders of magnitude larger than that of first-order CNNs, thus making these networks memory-intensive and subject to overfitting. While compression strategies have been proposed (Gao et al., 2016; Kong & Fowlkes, 2017), they are either nonparametric (Gao et al., 2016), thus limiting the representation power of the network, or designed for a specific classification formalism (Kong & Fowlkes, 2017), thus restricting their applicability.

In this chapter, we introduce a general, parametric compression strategy for second-order CNNs. As evidenced by our results, our strategy can produce more compact representations than (Gao et al., 2016; Kong & Fowlkes, 2017), with as little as 10% of their parameters, yet significantly outperforming these methods, as well as the state-of-the-art first-order (He et al., 2016; Simonyan

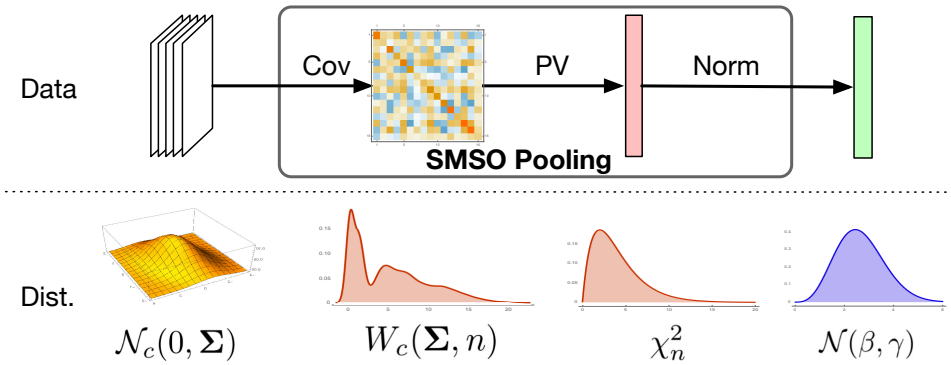



Figure 1.1: **Statistically-Motivated Second-Order (SMSO) pooling. Top:** Our parametric compression strategy vectorizes a covariance matrix and normalizes the resulting vector. **Bottom:** Each of these operations yields a well-defined distribution of the data, thus resulting in a consistent framework, whose final representation follows a Gaussian distribution, as state-of-the-art first-order deep networks.

& Zisserman, 2015) and uncompressed second-order pooling strategies (Lin et al., 2015; Ionescu et al., 2015; Li et al., 2017; Lin & Maji, 2017).

Unlike most deep learning architectures, our approach is motivated by a statistical analysis of the network's activations. In particular, we build upon the observation that first-order networks inherently exploit Gaussian distributions for their feature representations. This is due to the fact that, as discussed in (Goodfellow et al., 2016a; Ioffe & Szegedy, 2015) and explained by the Central Limit Theorem, the outputs of *linear* layers, and thus of operations such as global average pooling, follow a multivariate Gaussian distribution. The empirical success of such Gaussian distributions of feature representations in first-order deep networks motivated us to design a compression strategy such that the final representation also satisfies this property.

To this end, as illustrated by Figure 1.1, we exploit the fact that the covariance matrices resulting from second-order pooling follow a Wishart distribution (Johnson et al., 2014). We then introduce a parametric vectorization (PV) layer, which compresses the second-order information while increasing the model capacity by relying on trainable parameters. We show that our PV layer outputs a vector whose elements follow $\chi^2$ distributions, which motivates the use of a square-root normalization that makes the distribution of the resulting representation converge to a Gaussian, as verified empirically in Section 1.3.4. These operations rely on basic algebraic transformations, and can thus be easily integrated into any deep architecture and optimized with standard backpropagation.

We demonstrate the benefits of our statistically-motivated second-order (SMSO) pooling strategy on standard benchmark datasets for second-order models, including the Describing Texture Dataset (DTD) (Cimpoi et al., 2014), the Material in Context (MINC) dataset (Bell et al., 2015) and the scene recognition MIT-Indoor dataset (Quattoni & Torralba, 2009). Our approach consistently outperforms the state-of-the-art second-order pooling strategies, independently of

the base network used (i.e., VGG-D (Simonyan & Zisserman, 2015) or ResNet-50 (He et al., 2016)), as well as these base networks themselves.

## 1.2 Related Work

Visual recognition has a long history in computer vision. Here, we focus on the methods that, like us, make use of representations based on second-order information to tackle this task. In this context, the region covariance descriptors (RCDs) of (Tuzel et al., 2007) constitute the first attempt at leveraging second-order information. Similarly, Fisher Vectors (Arandjelovic & Zisserman, 2013) also effectively exploit second-order statistics. Following this success, several metrics have been proposed to compare RCDs (Arsigny et al., 2006; Pennec et al., 2006; Quang et al., 2014; Sra, 2012), and they have been used in various classification frameworks, such as boosting (Freund & Schapire, 1997), kernel Support Vector Machines (Vapnik, 1998), sparse coding (Cherian & Sra, 2014; Guo et al., 2010; Wang et al., 2016) and dictionary learning (Sra & Cherian, 2011; Harandi et al., 2012; Li et al., 2013; Harandi & Salzmann, 2015). In all these works, however, while the classifier was trained, no learning was involved in the computation of the RCDs.

To the best of our knowledge, (Harandi et al., 2014), and its extension to the log-Euclidean metric (Huang et al., 2015), can be thought of as the first attempts to learn RCDs. This, however, was achieved by reducing the dimensionality of input RCDs via a single transformation, which has limited learning capacity. In (Huang et al., 2017), the framework of (Harandi et al., 2014) was extended to learning multiple transformations of input RCDs. Nevertheless, this approach still relied on RCDs as input. The idea of incorporating second-order descriptors in a deep, end-to-end learning paradigm was introduced concurrently in (Ionescu et al., 2015) and (Lin et al., 2015). The former introduced the Deep$O_2$P operation, consisting of computing the covariance matrix of convolutional features. The latter proposed the slightly more general idea of bilinear pooling, which, in principle, can exploit inner products between the features of corresponding spatial locations from different layers in the network. In practice, however, the use of cross-layer bilinear features does not bring a significant boost in representation power (Gao et al., 2016; Lin & Maji, 2017), and bilinear pooling is therefore typically achieved by computing the inner products of the features within a single layer, thus becoming essentially equivalent to second-order pooling.

A key to the success of second-order pooling is the normalization, or transformation, of the second-order representation. In Ionescu et al. (2015), the matrix logarithm was employed, motivated by the fact that covariance matrices lie on a Riemannian manifold, and that this operation maps a matrix to its tangent space, thus producing a Euclidean representation. By contrast, Lin et al. (2015) was rather inspired by previous normalization strategies for handcrafted features (Arandjelovic & Zisserman, 2013; Perronnin et al., 2010), and proposed to perform an element-wise square-root and $\ell_2$ normalization after vectorization of the matrix representation. More recently, (Li et al., 2017; Lin & Maji, 2017) introduced a matrix square-root normalization strategy that was shown to outperform the other transformation techniques.

All the above-mentioned methods simply vectorize the second-order representation, i.e., covariance matrix. As such, they produce a final representation whose size scales quadratically with the number of channels in the last convolutional feature map, thus being typically orders of magnitude larger than the final representation of first-order CNNs. To reduce the resulting memory cost and parameter explosion, several approaches have been proposed to compress second-order representations while preserving their discriminative power. The first attempt at compression was achieved by (Gao et al., 2016), which introduced two strategies, based on the idea of random projection, to map the covariance matrices to vectors. These projections, however, were not learned, thus not increasing the capacity of the network and producing at best the same accuracy as the bilinear CNN of (Lin et al., 2015). In (Kong & Fowlkes, 2017), a parametric strategy was employed to reduce the dimensionality of the bilinear features. While effective, this strategy was specifically designed to be incorporated in a relatively complex bilinear Support Vector Machine formalism.

By contrast, here, we introduce a parametric compression approach that can be incorporated into any standard deep learning framework. Furthermore, our strategy is statistically motivated so as to yield a final representation whose distribution is of the same type as that inherently used by first-order deep networks. As evidenced by our experiments, our method can produce more compact representations than existing compression techniques, yet outperforms the state-of-the-art first-order and second-order models.

Note that higher-order information has also been exploited in the past (Cui et al., 2017; Koniusz et al., 2017). While promising, we believe that developing statistically-motivated pooling strategies for such higher-order information goes beyond the scope of this chapter.

## 1.3 Methodology

In this section, we first introduce our second-order pooling strategy while explaining the statistical motivation behind it. We then provide an alternative interpretation of our approach yielding a lower complexity, study and display the empirical distributions of our network's representations, and finally discuss the relation of our model to the recent second-order pooling techniques.

### 1.3.1 SMSO Pooling

Our goal is to design a general, parametric compression strategy for second-order deep networks. Furthermore, inspired by the fact that first-order deep networks inherently make use of Gaussian distributions for their feature representations, we reason about the statistical distribution of the network's intermediate representations so as to produce a final representation that is also Gaussian. Note that, while we introduce our SMSO pooling strategy within a CNN formalism, it applies to any method relying on second-order representations.

Formally, let $\mathbf{X} \in \mathbb{R}^{n \times c}$ be a data matrix, consisting of $n$ sample vectors of dimension $c$. For

example, in the context of CNNs, $\mathbf{X}$ contains the activations of the last convolutional layer, with $n = w \times h$ corresponding to the spatial resolution of the corresponding feature map. Here, we assume $\mathbf{x}_i \in \mathbb{R}^c$ to follow a multivariate Gaussian distribution $\mathcal{N}_c(\boldsymbol{\mu}, \Sigma)$. In practice, as discussed in (Goodfellow et al., 2016a; Ioffe & Szegedy, 2015) and explained by the Central Limit Theorem, this can be achieved by using a *linear* activation after the last convolutional layer, potentially followed by batch normalization (Ioffe & Szegedy, 2015).

**Covariance Computation.** Given the data matrix $\mathbf{X}$, traditional second-order pooling consists of computing a covariance matrix $\mathbf{Y} \in \mathbb{R}^{c \times c}$ as

$$\mathbf{Y} = \frac{1}{n-1} \sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T = \frac{1}{n-1} \tilde{\mathbf{X}}^T \tilde{\mathbf{X}}, \tag{1.1}$$

where $\tilde{\mathbf{X}}$ denotes the mean-subtracted data matrix.

The following definition, see, e.g.,Johnson et al. (2014), determines the distribution of $\mathbf{Y}$.

**Definition 1.** *If the elements $\mathbf{x}_i \in \mathbb{R}^c$ of a data matrix $\mathbf{X} \in \mathbb{R}^{n \times c}$ follow a zero mean multivariate Gaussian distribution $\mathbf{x}_i \sim \mathcal{N}_c(0, \Sigma)$, then the covariance matrix $\mathbf{Y}$ of $\mathbf{X}$ is said to follow a Wishart distribution, denoted by*

$$\mathbf{Y} = \mathbf{X}^T \mathbf{X} \sim W_c(\Sigma, n) . \tag{1.2}$$

Note that, in the bilinear CNN (Lin et al., 2015), the mean is typically not subtracted from the data. As such, the corresponding bilinear matrix follows a form of non-central Wishart distribution (James, 1955).

**Second-order Feature Compression.** The standard way to use a second-order representation is to simply vectorize it (Lin et al., 2015; Ionescu et al., 2015), potentially after some form of normalization (Lin & Maji, 2017; Li et al., 2017). This, however, can yield very high-dimensional vectors that are cumbersome to deal with in practice. To avoid this, motivated by Gao et al. (2016); Kong & Fowlkes (2017), we propose to compress the second-order representation during vectorization. Here, we introduce a simple, yet effective, compression technique that, in contrast with (Gao et al., 2016), is parametric, and, as opposed to Kong & Fowlkes (2017), amenable to general classifiers.

Specifically, we develop a parametric vectorization (PV) layer, which relies on trainable weights $\mathbf{W} \in \mathbb{R}^{c \times p}$, with $p$ the dimension of the resulting vector. Each dimension $j$ of the vector $\mathbf{z}$ output by this PV layer can be expressed as

$$z_j = \mathbf{w}_j^T \mathbf{Y} \mathbf{w}_j , \tag{1.3}$$

where $\mathbf{w}_j$ is a column of $\mathbf{W}$.

The distribution of each dimension $\mathbf{z}_j$ is defined by the following theorem.

**Theorem 1** (Theorem 5.6 in Johnson et al. (2014)). *If $\mathbf{Y} \in \mathbb{R}^{c \times c}$ follows a Wishart distribution $W_c(\Sigma, n)$, and $\mathbf{w} \in \mathbb{R}^c$ and $\mathbf{w} \neq \mathbf{0}$, then*

$$z = \frac{\mathbf{w}^T \mathbf{Y} \mathbf{w}}{\mathbf{w}^T \Sigma \mathbf{w}} \tag{1.4}$$

*follows a $\chi^2$ distribution with degree of freedom $n$, i.e., $z \sim \chi^2_n$.*

From this theorem, we can see that each output dimension of our PV layer follows a scaled $\chi^2$ distribution $\gamma \chi^2_n$, where $\gamma = \mathbf{w}_j^T \Sigma \mathbf{w}_j$, with $\Sigma$ the covariance matrix of the original multivariate Gaussian distribution.

**Transformation and normalization.** As shown above, each dimension of our current vector representation follows a $\chi^2$ distribution. However, as discussed above, first-order deep networks inherently exploit Gaussian distributions for their feature representations. To make our final representation also satisfy this property, we rely on the following theorem.

**Theorem 2** ((Wilson & Hilferty, 1931)). *If $z \sim \chi^2_n$ with degree freedom $n$, then*

$$z' = \sqrt{2z} \tag{1.5}$$

*converges to a Gaussian distribution with mean $\sqrt{2n-1}$ and standard deviation $\sigma = 1$ when $n$ is large, i.e., $z' \sim \mathcal{N}(\sqrt{2n-1}, 1)$.*

Following this theorem, we therefore define our normalization as the transformation

$$\mathbf{z}'_j = \sqrt{\alpha \mathbf{z}_j} - \sqrt{2n-1}\,, \tag{1.6}$$

for each dimension $j$, where we set $\alpha = 2/(\mathbf{w}_j^T \Sigma \mathbf{w}_j)$ to correspond to Theorem 2, while accounting for the factor $\gamma$ arising from our parametric vectorization above. Note that other transformations, such as $\log(z)$ and $(z/n)^{1/3}$, are known to also converge to Gaussian distributions as $n$ increases (Bartlett & Kendall, 1946; Wilson & Hilferty, 1931). We show that these operations yield similar results to the one above in Section 1.4.4.

Note that, according to Theorem 2, the mean and variance of the resulting Gaussian distribution are determined by the degree of freedom $n$, which, in our case, corresponds to the number of samples used to compute the covariance matrix in Eq. 1.1. Such pre-determined values, however, might limit the discriminative power of the resulting representation. To tackle this, we further rely on trainable scale and bias parameters, yielding a final representation

$$\mathbf{z}''_j = \beta_j + \gamma_j \mathbf{z}'_j\,, \tag{1.7}$$

where $\gamma_j > 0, \beta_j \in \mathbb{R}$. Note that this transformation is also exploited by batch normalization. However, here, we do not need to compute the batch statistics during training, since Theorem 2 tells us that the batches follow a consistent distribution.

Altogether, our SMSO pooling strategy, defined by the operations discussed above, yields a $p$-dimensional vector. This representation can then be passed to a classifier. It can easily be verified that the above-mentioned operations are differentiable, and the resulting deep network can thus be trained end-to-end.

### 1.3.2 Alternative Computation

Here, we derive an equivalent way to perform our SMSO pooling, with a lower complexity when $p$ is small, as shown in the Section 1.3.3. Note, however, that our statistical reasoning is much clearer for the derivation of Section 1.3.1 and was what motivated our approach.

To derive the alternative, we note that

$$\frac{1}{\sqrt{\alpha}}\mathbf{z}'_j = \sqrt{\mathbf{w}_j^T \mathbf{Y} \mathbf{w}_j} \tag{1.8}$$

$$= \sqrt{\mathbf{w}_j^T \left( \sum_{i=1}^{n} (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^T \right) \mathbf{w}_j} \tag{1.9}$$

$$= \sqrt{\sum_{i=1}^{n} \left( \mathbf{w}_j^T (\mathbf{x}_i - \mu) \right) \left( (\mathbf{x}_i - \mu)^T \mathbf{w}_j \right)} \tag{1.10}$$

$$= \sqrt{\sum_{i=1}^{n} (\mathbf{w}_j^T \tilde{\mathbf{x}}_i)^2}\,, \tag{1.11}$$

where $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \mu$.

So, in essence, given $\mathbf{X}$, $\mathbf{z}'$ can be computed by performing a $1 \times 1$ convolution, with weights shaped as $(1, 1, c, p)$ and without bias, followed by a global $\ell_2$ pooling operation, and a scaling by the constant $\sqrt{\alpha}$. Note that $\ell_2$ pooling was introduced several years ago (Sermanet et al., 2012), but has been mostly ignored in the recent deep learning advances. By contrast, feature reduction with $1 \times 1$ convolutions is widely utilized in first-order network designs (Szegedy et al., 2015; He et al., 2016). In essence, this mathematically equivalent formulation shows that our second-order compression strategy can be achieved without explicitly computing covariance matrices. Yet, our statistical analysis based on these covariance matrices remains valid.

### 1.3.3 Relation to Other Methods

In this section, we discuss the connections between our method and the other recent second-order pooling strategies in CNNs. Here, we also compare the computational complexity of different second-order methods with that of ours.

**Normalization.** Bilinear pooling (BP) (Lin et al., 2015) also proposed to make use of a square-root as normalization operation. An important difference with our approach, however, is that

BP directly vectorizes the matrix representation $\mathbf{Y}$. It is easy to see that the diagonal elements of $\mathbf{Y}$ follow a $\chi^2$ distribution, e.g., by taking $\mathbf{w}$ in Theorem 1 to be a vector with a single value 1 and the other values 0. Therefore, after normalization, some of the dimensions of the BP representation also follow a Gaussian distribution. However, the off-diagonal elements follow a variance-gamma distribution, and, after square-root normalization, will not be Gaussian, thus making the different dimensions of the final representation follow inconsistent distributions.

Ionescu et al. (2015) and Li et al. (2017) proposed that normalization was performed on the matrix $\mathbf{Y}$ directly, via a matrix logarithm and a matrix power normalization, respectively. As such, it is difficult to understand what distribution the elements of the final representation, obtained by standard vectorization, follow.

**Compression.** The compact bilinear pooling (CBP) of (Gao et al., 2016) exploits a compression scheme that has a form similar to ours in Eq. 1.3. However, the projection vectors $\mathbf{w}_j$ are random but fixed (Gao et al., 2016). Making them trainable, as in our PV layer, increases the capacity of our model, and, as shown in Section 1.4, allows us to significantly outperform CBP.

Kong & Fowlkes (2017) proposed a model developed specifically for a max-margin bilinear classifier. The parameter matrix of this classifier is approximated by a low-rank factorization, which translates to projecting the initial features to a lower-dimensional representation. As with our alternative formulation of Section 1.3.2, the resulting bilinear classifier can be obtained without having to explicitly compute $\mathbf{Y}$. This classifier is formulated in terms of quantities of the form $\|\mathbf{U}^T\mathbf{X}_i\|_F^2$, where $\mathbf{U}$ is a trainable low-rank weight matrix. In essence, this corresponds to removing the square-root operation from Eq. 1.11 and summing over all dimensions $j$. By contrast, our representation, ignoring the scale and bias of Eq. 1.7, is passed to a separate classification layer that computes a linear combination of the different dimensions with trainable weights, thus increasing the capacity of our model.

**Complexity analysis.** We compare the theoretical and empirical parameter numbers and computational complexity of the baselines and our SMSO pooling strategy. We show that we achieve a 90% reduction in parameter number and a speed-up of around 20% with our alternative operations.

**Parameter Analysis** We now compare the computational complexity of the different state-of-the-art second-order methods discussed in the main paper. For CBP, we focus on the Tensor Sketch variant (CBP-TS), which has proven more effective than the Random MacLaurin one. For LRBP, we focus on the second version (LRBP-II), which has typically a lower complexity than LRBP-I (Kong & Fowlkes, 2017), and was the version used the main paper. The comparison of different aspects of the methods is provided in Table 1.1.

In terms of feature dimension, BP and MPN, which vectorize the whole matrix $\mathbf{Y}$, lead to the highest values. They are followed by LRBP and CBP-TS, assuming the standard parameters $d = 8,192$ for CBP-TS and $m = 100$ for LRBP. Our model yields the smallest feature dimension,

| Model | Feature Dim. | Feature Param. | Classifier Param. | Total Param. |
|---|---|---|---|---|
| VGG-D | $d_{fc}$[4K] | $2d_{fc}^2 + hwcd_{fc}$ [478MB] | $Kd_{fc}$[$K\cdot$ 16KB] | [478MB] |
| BP | $c^2$ [262K] | 0 | $Kc^2$ [$K$MB] | $Kc^2$ [23MB] |
| MPN | $c^2$ [262K] | 0 | $Kc^2$ [$K$MB] | $Kc^2$ [23MB] |
| CBP-TS | $d$[10K] | $2c$ [4KB] | $Kd$ [$K\cdot$ 32KB] | $2c + Kd$ [0.74MB] |
| LRBP-II | $m^2$ [10K] | $cm$ [200KB] | $Krm$ [$K\cdot$ 3KB] | $cm + Krm$ [239KB] |
| SMSO (ours) | $p$ [0.06K] | $cp$ [131KB] | $Kp$ [$K\cdot$ 0.24KB] | $cp + Kp$ [136KB] |
| SMSO (ours) | $p$ [2K] | $cp$ [4MB] | $Kp$ [$K\cdot$ 6KB] | $cp + Kp$ [4.3MB] |

Table 1.1: **Parameters of the state-of-the-art first and second-order methods.** We compare original VGG-D (Simonyan & Zisserman, 2015), Bilinear Pooling (BP) (Lin et al., 2015), Matrix Power Normalization Covariance (MPN) (Li et al., 2017), Compact Bilinear Pooling with Tensor Sketch (CBP-TS) (Gao et al., 2016), Low Rank Bilinear Pooling (LRBP) (Kong & Fowlkes, 2017) and our SMSO strategy. The complexity is computed for initial features coming from the $conv5\_3$ layer of VGG-D, with spatial resolution $h = w = 14$ for an input image of size $224 \times 224$, and with $c = 512$ as feature dimension. For VGG-D, $d_{fc} = 4,096$ as in (Simonyan & Zisserman, 2015). For CBP-TS, we used $d = 8,192$ as in (Gao et al., 2016). For LRBP, we set $m = 100$ and $r = 8$ as the projection matrix dimension and rank, respectively. For our approach, we used either $p = 64$ or $p = 2,048$ as the projection dimension. The number of classes $K$ is set to 23, corresponding to the MINC-2500 dataset. Feature dim. refers to the dimension of the final feature representation. Feature param. and classifier param. refer to the total number of training parameters for the pooling and classification layers, respectively. Total param. is the sum of feature and classifier param.

whether using $p = 64$ or $2,048$. Despite this, as evidenced by our experiments, we outperform these baselines, which indicates the strong discriminative power of our SMSO strategy.

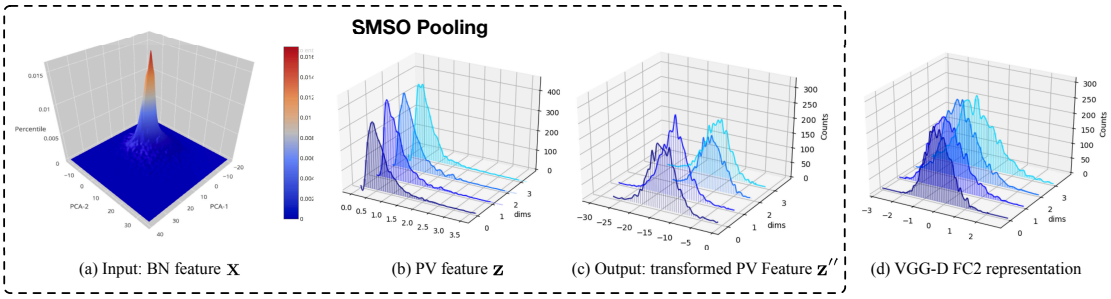
For the number of parameters, even though BP and MPN-COV have parameter-free normalization, their use of vanilla vectorization results in a large number of classifier parameters, thus making these methods parameter-intensive. By performing compression, but not training its projection, CPB-TS significantly reduces the total number of parameters. While this comes at no loss in accuracy over BP, it doesn't allow this model to reach the performance of LRBP or of our approach. By projecting the data to a much lower dimensionality, LRBP has the smallest total number of parameters among the baselines. Note however that our model is even more compact and significantly outperforms all the baselines.

**Runtime Analysis** In Table 1.2, we analyze the computational complexity and runtimes of the different models. In terms of computational complexity, MPN is the slowest because of its use of eigenvalue decomposition for normalization. Altogether, CBP-TS has the smallest computational complexity, with LRBP having smaller complexity in feature computation but larger in classifier computation. While our method has higher complexity than BP in feature computation, it has a much lower one in classifier computation, making it overall faster than this baseline. Note that, with our alternate formulation of Section 3.3, our feature computation reduces to $O(hwcp)$ because we avoid explicitly computing the covariance matrix.

We also profiled the training and inference times in milli-seconds (ms) per image of the baseline

| Model | Feature comp. | Classification comp. | $T_{train}$ | $T_{inf}$ |
|---|---|---|---|---|
| VGG-D | $O(hwcn+n^2)$ | $O(Kn)$ | 2.30 | 0.82 |
| BP | $O(hwc^2)$ | $O(Kc^2)$ | 1.0 | 0.91 |
| MPN | $O(hwc^2+c^3)$ | $O(Kc^2)$ | 47.9 | 42.7 |
| CBP-TS | $O(hw(c+d\log d))$ | $O(Kd)$ | 1.2 | 1.0 |
| LRBP-II | $O(hwm(c+m))$ | $O(Krm^2)$ | 1.3 | 0.91 |
| SMSO (ours) | $O(hwc(c+p))$ | $O(Kp)$ | 1.2 | 0.68 |
| SMSO-alt (ours) | $O(hwcp)$ | $O(Kp)$ | **0.87** | **0.58** |

Table 1.2: **Computational complexity of the state-of-the-art first- and second-order methods.** The symbols are defined as in Table 1.1. Feature and classifier comp. refer to the complexity of computing the features and the final class scores, respectively. $T_{train}$ and $T_{inf}$ refer to the training and inference time (ms) per image. Our alternative computation yields the fastest training and inference speed.




(a) Input: BN feature $\mathbf{x}$ (b) PV feature $\mathbf{z}$ (c) Output: transformed PV Feature $\mathbf{z}''$ (d) VGG-D FC2 representation

Figure 1.2: **Histograms of SMSO intermediate feature vectors.** We plot the distribution of (a) the initial features $\mathbf{X}$, (b) the features after our PV layer $\mathbf{z}$, (c) the final representation $\mathbf{z}''$ and, for comparison, (d) first-order features after the last fully-connected layer in VGG-D (Simonyan & Zisserman, 2015). Note that, as discussed in the text, these empirical distributions match the theoretical ones derived in Section 1.3.1, and our final representation does exploit the same type of distribution as first-order networks.

models and of our SMSO pooling strategy with $p = 2,048$. For the comparison to focus on the differences between the models, we do not include feature extraction up to, and including, the conv5_3 layer, but rather compute the runtimes from these features to the final $k$-way classification softmax layer. The results are shown in the last two columns of Table 1.2. Due to the dense fully-connected layers of VGG-D, processing one image requires 2.3ms during training, which is slower than most second-order models. However, the inference speed is faster than most of them, since no gradient computation is involved anymore. Nevertheless, inference in our model remains faster. For most second-order baselines, training time ranges from 1.0ms to 1.3ms, whereas inference time ranges from 0.6ms to 1.0ms. MPN requires significantly higher training (47.9ms) and inference (42.7ms) times, because it computes the matrix-logarithm via eigenvalue decomposition. Our original SMSO pooling takes 1.2ms for training and 0.68ms for inference on average. The alternative implementation of Section 3.3 reduces these timings to 0.87ms and 0.58ms, respectively, since it avoids explicitly computing the covariance matrix.

### 1.3.4 Empirical distributions of SMSO pooling

Our SMSO pooling strategy was motivated by considering the distribution of the representation at various stages in the network. Here, we study the empirical distributions of these features using the MINC dataset, discussed in Section 1.4, and with a model based on VGG-D. To this end, in Figure 1.2, we provide a visualization of the distributions after the initial batch normalization (i.e., before computing the covariance matrix, see Section 1.4.1 for details), after our PV layer, and after square-root transformation with trainable scaling and bias. Specifically, for the initial features, because visualizing a Gaussian distribution in hundreds of dimensions is not feasible, we plot the distribution along the first 2 principal components. For our representations, where each dimension follows an independent Gaussian, we randomly select four dimensions and plot stacked histograms. As expected from the theory, the initial features are close to Gaussian, and the features after our PV layer therefore follow a long-tailed $\chi^2$ distribution. The final features, after square-root normalization, scaling and bias, are much less skewed, and thus much closer to a Gaussian distribution, thus matching the type of distribution that the final representations of state-of-the-art deep networks follow, as shown in Figure 1.2(d). To further verify this, we conducted a Shapiro-Wilk test on the final representation. This resulted in a p-value of $0.19 > 0.05$, which means that the Gaussian assumption is not rejected, sustaining our claim.

## 1.4 Experiments

Here, we first provide implementation details and introduce the baseline models. We then compare our approach to these baselines on four standard benchmark datasets, and provide an ablation study of our framework.

### 1.4.1 Implementation Details

We evaluate our method on two popular network architectures: the VGG-D network of (Simonyan & Zisserman, 2015) (a.k.a. VGG-16) and the ResNet-50 of (He et al., 2016). For all second-order models discussed below, i.e., ours and the baselines, we remove all the fully-connected layers and the last max pooling layer from VGG-D, that is, we truncate the model after the ReLU activation following *conv5-3*. For ResNet-50, we remove the last global average pooling layer and take our initial features as those from the last residual block. As in Li et al. (2017), we add a $1 \times 1$ convolutional layer to project the initial features to $c = 256$ for all the experiments. Note that this is a linear layer, and thus makes the resulting features satisfy our Gaussian assumption.

Following common practice (Gao et al., 2016; Kong & Fowlkes, 2017; Lin et al., 2015; Li et al., 2017), we rely on weights pre-trained on ImageNet and use stochastic gradient descent with an initial learning rate 10 times smaller than the one used to learn from scratch, i.e., 0.001 for VGG-D and 0.01 for ResNet-50. We then divide this weight by 10 when the validation loss has stopped decreasing for 8 epochs. We initialize the weights of the new layers, i.e., the

$1 \times 1$ convolution, the PV layer and the classifier, with the strategy of (Glorot & Bengio, 2010), i.e., random values drawn from a Gaussian distribution. We implemented our approach using Keras (Chollet et al., 2015) with TensorFlow (Abadi et al., 2015) as backend.

### 1.4.2 Baseline Models

We now describe the different baseline models that we compare our approach with. Note that the classifier is defined as a $k$-way softmax layer for all these models, as for ours, except for low-rank bilinear pooling, which was specifically designed to make use of a low-rank hinge loss.

**Original model:** This refers to the original, first-order, models, i.e., either VGG-D or ResNet-50, pre-trained on ImageNet and fine-tuned on the new data. Other than replacing the 1000-way ImageNet classification layer with a $k$-way one, we keep the original model settings described in (Simonyan & Zisserman, 2015) and (He et al., 2016), respectively.

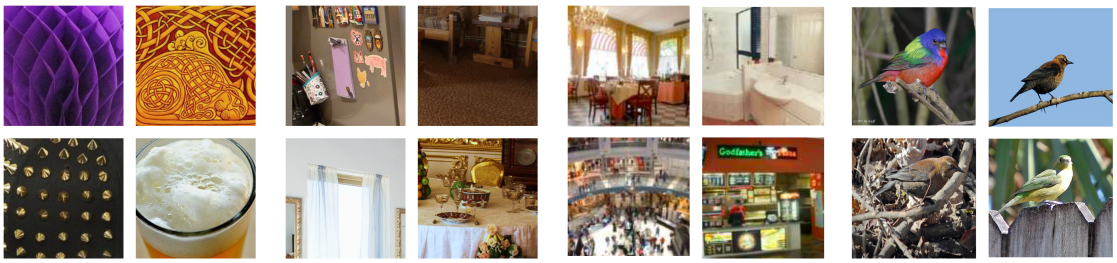
**Bilinear Pooling (BP) (Lin et al., 2015):** This corresponds to the original, uncompressed bilinear pooling strategy, with signed square-root and $\ell_2$ normalization after vanilla vectorization. In this case, we set $c = 512$, as in the original paper, as the feature dimension before computing the second-order representation. If the original feature dimension does not match this value, i.e., with ResNet-50, we make use of an additional $1 \times 1$ convolutional layer. Note that we observed that using either 512 or 256 as feature dimension made virtually no difference on the results. We therefore used $c = 512$, which matches the original paper.

**DeepO$_2$P (Ionescu et al., 2015):** This refers to the original, uncompressed covariance-based model, with matrix logarithm and vanilla vectorization. Again, as in the original paper, we set $c = 512$ as the feature dimension before computing the covariance matrix, by using an additional $1 \times 1$ convolutional layer when necessary.

**Matrix Power Normalization (MPN) (Li et al., 2017):** This model relies on a matrix square-root operation acting on the second-order representation. Following the original paper, we set $c = 256$ by making use of an additional $1 \times 1$ convolutional layer before second-order pooling. Note that the improved bilinear pooling of (Lin & Maji, 2017) has the same structure as MPN, and we do not report it as a separate baseline.

**Compact bilinear pooling (CBP) (Gao et al., 2016):** We report the results of both versions of CBP: the Random Maclaurin (RM) one and the Tensor Sketch (TS) one. For both versions, we set the projection dimension to $d = 8,192$, which was shown to achieve the same accuracy as BP, i.e., the best accuracy reported in (Gao et al., 2016). As in the original paper, we apply the same normalization as BP (Lin et al., 2015).

**Low rank bilinear pooling (LRBP) (Kong & Fowlkes, 2017):** This corresponds to the compression method dedicated to the bilinear SVM classifier. Following Kong & Fowlkes (2017),



(a) DTD (b) MINC-2500 (c) MIT-Indoor (d) CUB-2011

Figure 1.3: **Sample images from DTD, MINC-2500, MIT-Indoor and CUB.**

we set the projection dimension to $m = 100$ and its rank to $r = 8$, and initialize the dimensionality reduction layer using the SVD of the Gram matrix computed from the entire validation set. Following the authors' implementation, we apply a scaled square-root with factor $2 \times 10^5$ after the *conv5-3* ReLU, which seems to prevent the model from diverging. Furthermore, we found that training LRBP from the weights of BP fine-tuned on each dataset also helped convergence.

### 1.4.3 Comparison to the Baselines

Let us now compare the results of our model with those of the baselines described in Section 1.4.2. To this end, we make use of four diverse benchmark image classification datasets, thus showing the versatility of our approach. These datasets are the Describing Texture Dataset (DTD) (Cimpoi et al., 2014) for texture recognition, the challenging Material In Context (MINC-2500) dataset (Bell et al., 2015) for large-scale material recognition in the wild, the MIT-Indoor dataset (Quattoni & Torralba, 2009) for indoor scene understanding and the Caltech-UCSD Bird (CUB) dataset (Wah et al., 2011) for fine-grained classification. DTD contains 47 classes for a total of 5,640 images, mostly capturing the texture itself, with limited surrounding background. By contrast, MINC-2500, consisting of 57,500 images of 23 classes, depicts materials in their real-world environment, thus containing strong background information and making it more challenging. MIT-Indoor contains 15,620 images of 67 different indoor scenes, and, with DTD, has often been used to demonstrate the discriminative power of second-order representations. The CUB dataset contains 11,788 images of 200 different bird species. In Figure 1.3, we provide a few samples from each dataset. For our experiments, we make use of the standard train-test splits released with these datasets. For DTD, MIT-Indoor and CUB, we define the input size as $448 \times 448$ for all the experiments. For the large-scale MINC-2500 dataset, we use $224 \times 224$ images for all models to speed up training. Note that a larger input size could potentially result in higher accuracies (Bell et al., 2015). For all datasets and all models, we use the same data augmentation strategy as in (Lin et al., 2015; Lin & Maji, 2017).

**Experiments with VGG-D.** We first discuss the results obtained with the VGG-D architecture as base model. These results are reported in Table 1.3 for all models and all datasets. In short, our SMSO framework with PV dimension $p = 2,048$ outperforms all the baselines by a significant margin on all three datasets. In particular, our accuracy is 7% to 19% higher than the original VGG-

| Model name | Feature dim. | # param. | DTD | MIT | MINC | CUB |
|---|---|---|---|---|---|---|
| VGG-D (Simonyan & Zisserman, 2015) | 4,096 | 119.64M | 60.11 | 64.51 | 73.01 | 66.12 |
| BP (Lin et al., 2015) | $2.6 \times 10^5$ | 3.015M | 67.50 | 77.55 | 74.50 | 81.02 |
| MPN (Li et al., 2017) | 32,896 | 0.752M | 68.01 | 76.49 | 76.24 | 84.10 |
| DeepO$_2$P (Ionescu et al., 2015) | $2.6 \times 10^5$ | 3.015M | 66.07 | 72.35 | 69.29 | - |
| CBP-TS (Gao et al., 2016) | 8,192 | 0.189M | 67.71 | 76.83 | 73.28 | 84.00 |
| CBP-RM (Gao et al., 2016) | 8,192 | 0.189M | 63.24 | 73.89 | 73.54 | 83.86 |
| LRBP (Kong & Fowlkes, 2017) | 100 | 0.068M | 65.80 | 73.59 | 69.10 | 84.21 |
| SMSO (*Ours*) | 64 | 0.013M | 68.18 | 75.37 | 74.18 | 82.66 |
| SMSO (*Ours*) | 2,048 | 0.057M | **69.26** | **79.45** | **78.00** | **85.01** |

Table 1.3: **Comparison of VGG-D based models.** We report the top 1 classification accuracy (in %) of the original VGG-D model, uncompressed second-order models with different normalization strategies (BP, DeepO$_2$P, MPN), second-order compression methods (CBP-TS, CBP-RM, LRBP), and our approach (SMSO) with different PV dimensions. Note that our approach significantly outperforms all the baselines despite a more compact final representation (Feature dim.) and much fewer parameters (# param is the number of trainable parameters after the last convolutional layer).

D, with *much* fewer parameters, thus showing the benefits of exploiting second-order features. MPN is the best-performing baseline, but, besides the fact that we consistently outperform it, has a much higher computational complexity and run time, discussed in Section 1.3.3. The second-order compression methods (CBP and LRBP) underperform the uncompressed models on average. By contrast, even with $p = 64$, we outperform most baselines, with a model that corresponds to 10% of the parameters of the most compact baseline.

In Figure 1.4, we compare the training and validation loss curves of our approach with those of the best-performing baselines, BP and MPN, on DTD and MINC. Note that our model converges much faster than BP and tends to be more stable than MPN, particularly on DTD. This, we believe, is due to the fact that we rely on basic algebraic operations, instead of the eigenvalue decomposition involved in MPN whose gradient can be difficult to compute, particularly in the presence of small or equal eigenvalues (Ionescu et al., 2015).

During these VGG-D based experiments, we have observed that, in practice, LRBP was difficult to train, being very sensitive to the learning rate, which we had to manually adapt throughout training. Because of this, and the fact that LRBP yields lower accuracy than uncompressed models, we do not include this baseline in the remaining experiments. We also exclude DeepO$_2$P from the next experiments, because of its consistently lower accuracy.

**Experiments with ResNet-50.** To further show the generality of our approach, we make use of the more recent, very deep ResNet-50 (He et al., 2016) architecture as base network. Table 1.4 provides the results of our SMSO framework with $p = 64$ and $p = 2,048$, and of the baselines. In essence, the conclusions remain unchanged; we outperform all the baselines for $p = 2,048$. Note that, here, however, the second-order baselines typically do not even outperform the original

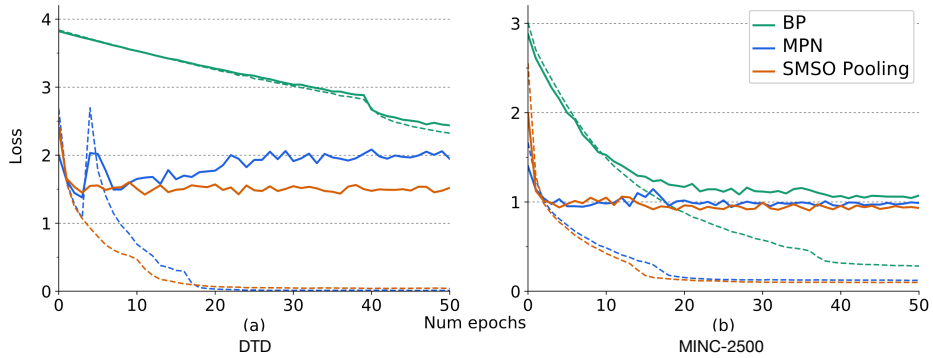



Figure 1.4: **Training and validation loss curves.** We plot the training (dashed) and validation (solid) loss values as a function of the number of training epochs for our SMSO pooling strategy (orange), BP (green) and MPN (blue) on DTD (a) and MINC-2500 (b). Our models clearly converge faster than BP, and tend to be more stable than MPN, particularly on the smaller-scale DTD dataset.

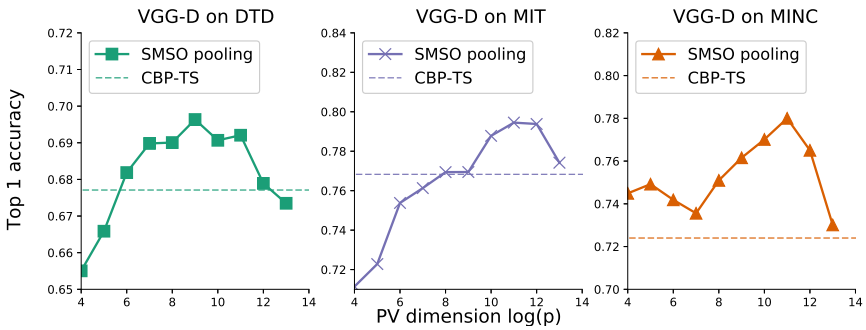



Figure 1.5: **Influence of the PV dimension $p$.** We plot the top 1 accuracy as a function of the value $p$ in logarithmic scale on MIT (left), MINC (middle) and DTD (right). Note that accuracy is quite stable over large ranges of $p$ values, yielding as good results as the best-performing compression baseline (CBP-TS) with as few as $p = 64$ dimensions, corresponding to only 10% of the parameters of CBP-TS.

ResNet-50, whose results are significantly higher than the VGG-D ones. By contrast, our model is able to leverage this improvement of the base model and to further increase its accuracy by appropriately exploiting second-order features.

### 1.4.4 Ablation Study

We evaluate the influence of different components of our model on our results.

**Influence of the PV dimension.** In our experiments, we proposed to set $p = 2,048$ or $p = 64$. We now investigate the influence of this parameter on our results. To this end, we vary $p$ in the range $[2^4, 2^{13}]$ by steps corresponding to a factor 2. The curves for this experiment on the validation data of the three datasets with VGG-D based models are provided in Figure 1.5. Note that our

| Model name | Feature dim. | # param. | DTD | MIT | MINC | CUB |
|---|---|---|---|---|---|---|
| ResNet-50 (He et al., 2016) | 2,048 | 4K | 71.45 | 76.45 | 79.12 | 74.51 |
| BP (Lin et al., 2015) | 32,896 | 752K | 69.37 | 68.35 | 79.05 | 82.70 |
| MPN (Li et al., 2017) | 32,896 | 752K | 71.10 | 72.12 | 79.83 | 85.43 |
| CBP-TS (Gao et al., 2016) | 8,192 | 189K | 65.30 | 72.60 | 75.91 | 77.35 |
| CBP-RM (Gao et al., 2016) | 8,192 | 189K | 62.35 | 67.81 | 74.15 | - |
| SMSO (*Ours*) | 64 | 13K | 71.03 | 76.31 | 79.17 | 81.98 |
| SMSO (*Ours*) | 2,048 | 57K | **72.51** | **79.68** | **81.33** | **85.77** |

Table 1.4: **Comparison of ResNet-50 based models.** We report the top 1 classification accuracy (in %) of the original ResNet-50 model, uncompressed second-order models with different normalization strategies (BP, MPN), second-order compression methods (CBP-TS, CBP-RM), and our approach (SMSO). Note that, as in the VGG-D case, our model outperforms all the baselines, including the original ResNet-50, which is not the case of most second-order baselines. It also yields much more compact models than the second-order baselines. (# params. refers to the same quantity as in Table 1.3.)

model is quite robust to the exact value of this parameter, with stable accuracies outperforming the best compression baseline for each dataset over large ranges. More importantly, even with $p = 64$, our model yields as good results as the best compression method, CBP-TS, with only 10% of its parameters.

**Comparison of different distributions and transformations.** We conduct experiments to compare different final feature distributions on MINC-2500 with a VGG-D based model. The results are provided in Table 1.5. Without our PV compression and without transformation or normalization, the resulting features follow a Wishart distribution, yielding an accuracy of 75.97%, which is comparable to BP (Lin et al., 2015). Adding our PV layer $p = 2,048$, but not using any transformation or normalization, yields $\chi^2$-distributed features and an accuracy similar to the previous one. This suggests that our parametric compression is effective, since we obtain similar accuracy with much fewer parameters. Including the square-root transformation, but without the additional scale and bias of Eq. 1.7, increases the accuracy to 76.32%. Additionally learning the scale and bias boosts the accuracy to 78.00%, thus showing empirically the benefits of Gaussian-distributed features over other distributions.

In the last two columns of Table 1.5, we report the results of different transformations that bring the $\chi^2$-distributed features to a Gaussian distribution, i.e., the cubic-root and the element-wise logarithm. Note that these two transformations yield accuracies similar to those obtained with the square-root. More importantly, all transformations yield higher accuracies than not using any (76.14%), which further evidences the benefits of Gaussian-distributed features.

**Effect of the $1 \times 1$ Convolutional Layer** For the baselines described in Section 1.4.1, we followed the setup described in the respective paper to define the dimensionality of the final

| Vec. | Flatten | PV | | | | | PV | | |
|---|---|---|---|---|---|---|---|---|---|
| Trans. | - | - | Sqrt | Sqrt | Sqrt | - | Sqrt | Log | $\sqrt[3]{\ }$ |
| Norm. | - | - | - | $\beta$ | $\gamma$ | $\beta,\gamma$ | $\beta,\gamma$ | | |
| Dist. | $W_n(\Sigma)$ | $\chi^2_n$ | $\mathcal{N}(\mu,1)$ | $\mathcal{N}(\beta,1)$ | $\mathcal{N}(\mu,\gamma^2)$ | $\gamma\chi^2_n+\beta$ | $\mathcal{N}(\beta,\gamma^2)$ | | |
| Acc. | 75.97 | 75.32 | 76.32 | 77.12 | 76.47 | 76.14 | 78.00 | 77.86 | 77.17 |

Table 1.5: **Comparison of different final feature distributions.** We report the results of different combinations of vectorization (vec.), transformation (trans.) and normalization (norm.) strategies, yielding different final feature distributions. Here, $\mu = \sqrt{2n-1}$ from Theorem 2. Ultimately, these results show that bringing the data back to a Gaussian distribution with a trainable scale and bias yields higher accuracies.

convolutional feature representation, before computing the second-order descriptor. For example, Bilinear Pooling (BP) (Lin et al., 2015) relies on 512D features, whereas MPN (Li et al., 2017) makes use of 256D features, obtained after an additional $1 \times 1$ convolutional layer. Here, we evaluate the influence of the dimensionality of these features and the importance of this additional $1 \times 1$ convolutional layer on our results and on those of the best-performing baselines, BP and MPN. In Table 1.6, we report recognition accuracies on DTD without (w/o 1x1) and with (w 1x1) $1 \times 1$ convolutions, and in the latter case, for a final dimension of either 512 or 256. Note that these different setups result in relatively small variations in accuracy. More importantly, our approach still yields the overall best result, with 256D features, which, as discussed above, corresponds to a much lower computational complexity than the best-performing baseline MPN with 512D features.

| Setup | BP | MPN | **SMSO** |
|---|---|---|---|
| w/o 1x1 | 67.50 | 63.50 | 69.15 |
| w 1x1(256) | 67.46 | 68.01 | **69.26** |
| w 1x1(512) | 67.65 | 68.72 | 68.34 |

Table 1.6: **Influence of the $1 \times 1$ convolutional layer before computing the second-order features.** We compare the results of our approach with the best-performing baselines BP and MPN without (w/o 1x1) and with (w 1x1) the use of an additional $1 \times 1$ convolutional layer. In the latter case, we also evaluate the effect of using either 256D features, as recommended for MPN in Li et al. (2017), or 512D ones. These experiments were conducted on DTD with VGG-D based models. Note that the performance of all models is quite stable, with ours achieving the overall best accuracy.

**Impact of Batch Normalization.**

We study the impact of using a batch normalization layer before computing the second-order representation, which was motivated by the intuition that it reduces the variance shift of the previous convolutional layer, and yields a more stable data distribution since our operations depend on its statistical properties. In particular, we also investigate the impact of such a layer on the best-performing baseline models. The results of this experiment on MINC-2500 using VGG-D based models are provided in Table 1.7. Note that our method strongly benefits from

| Setup | MPN | BP | **SMSO** |
|---|---|---|---|
| w/o BN | 76.24 | 74.50 | 69.76 |
| w BN | 70.04 | 72.63 | **78.00** |

Table 1.7: **Influence of batch normalization before computing the second-order features.** These experiments were conducted on MINC-2500 with VGG-D based models. Note that, as opposed to the baselines, our method benefits from an initial batch normalization, which makes the data better satisfy our initial assumption of stable Gaussian distribution.

this additional layer, which is reasonable, since it helps satisfy our initial assumption, i.e., stable Gaussian distribution across the mini-batches. By contrast, such an additional layer harms the baselines' accuracy. As such, our method still yields the highest accuracy on this dataset.

## 1.5 Conclusion

We have introduced a general and parametric compression strategy for second-order deep networks, motivated by a statistical analysis of the distribution of the network's intermediate representations. Our SMSO pooling strategy outperforms the state-of-the-art first-order and second-order models, with higher accuracies than other compression techniques for up to 90% parameter reduction. With a ResNet-50 base architecture, it is the only second-order model to consistently outperform the original one. While Gaussian distributions have proven effective here and for first-order models, there is no guarantee that they are truly optimal.

# 2 Recurrent U-Net

In the context of manually-designed deep networks, we now turn our focus to another critical computer vision task, semantic segmentation. The current state-of-the-art solutions to this problem tend to perform well on large datasets but can only process a few frames per second on expensive GPUs. To address this, we introduce a novel architecture, dubbed recurrent U-Net, that recurrently process a single input image to improve the prediction, targets application domains with limited amounts of data and achieves real-time performance on commodity hardware.

## 2.1 Introduction

Similarly to image classification, semantic segmentation is another critical computer vision task. The common approach to tackling it consists of formulating the segmentation problem as a pixel classification one. In this chapter, we continue on the topic of manually-designed deep networks and develop a CNN dedicated to the image segmentation task, drawing inspiration from human perception.

While recent semantic segmentation methods achieve impressive results (Chen et al., 2018a; Lin et al., 2017; Long et al., 2015; Zhao et al., 2017), they require very deep networks and their architectures tend to focus on high-resolution and large-scale datasets and to rely on pre-trained backbones. For instance, state-of-the-art models, such as Deeplab (Chen et al., 2017; 2018a), PSPnet (Zhao et al., 2017) and RefineNet (Lin et al., 2017), use a ResNet101 (He et al., 2016) as their backbone. This results in high GPU memory usage and inference time, and makes them less than ideal for operation in power-limited environments where real-time performance is nevertheless required, such as when segmenting hands using the onboard resources of an Augmented Reality headset. This has been addressed by architectures such as the ICNet (Zhao et al., 2018) at the cost of a substantial performance drop. Perhaps even more importantly, training very deep networks usually requires either massive amounts of training data or image statistics close to that of ImageNet (Deng et al., 2009), which may not be appropriate in fields such as biomedical image segmentation where the more compact U-Net architecture remains

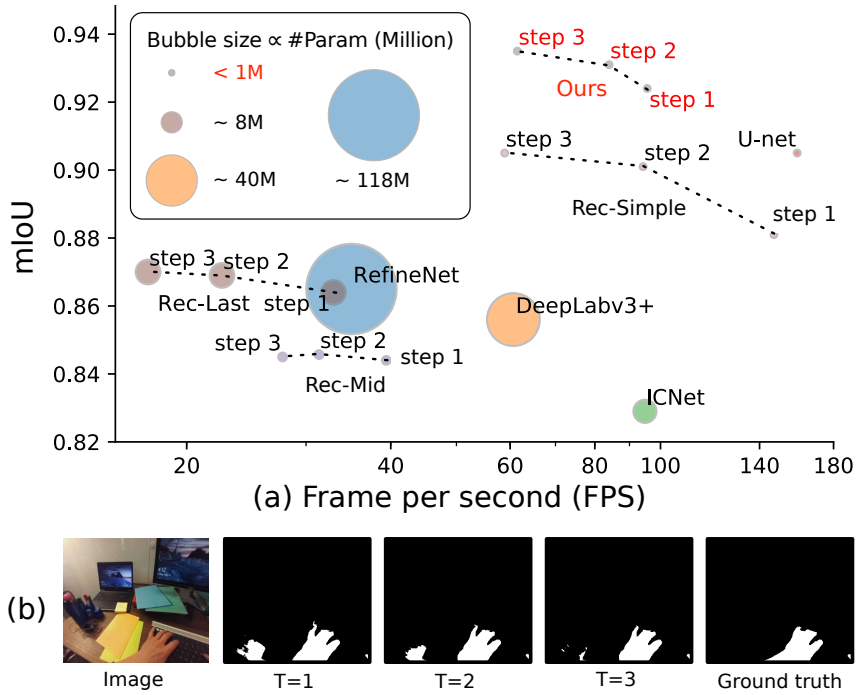



Figure 2.1: **Speed vs accuracy.** Each circle represents the performance of a model in terms frames-per-second and mIoU accuracy on our Keyboard Hand Dataset using a Titan X (Pascal) GPU. The radius of each circle denotes the models' number of parameters. For our recurrent approach, we plot these numbers after 1, 2, and 3 iterations, and we show the corresponding segmentations in the bottom row. The performance of our approach is plotted in red and the other acronyms are defined in Section 2.4.2. ICNet (Zhao et al., 2018) is slightly faster than us but at the cost of a significant accuracy drop, whereas RefineNet (Lin et al., 2017) and DeepLab (Chen et al., 2018a) are both slower and less accurate on this dataset, presumably because there are not enough training samples to learn their many parameters.

prevalent (Ronneberger et al., 2015).

In this chapter, we argue that these state-of-the-art methods do not naturally generalize to resource-constrained situations and introduce a novel recurrent U-Net architecture that preserves the compactness of the original U-Net (Ronneberger et al., 2015), while substantially increasing its performance to the point where it outperforms the current state of the art on 5 hand-segmentation datasets, one of which is showcased in Figure 2.1, and a retina vessel segmentation one. With only 0.3 million parameters, our model is much smaller than the ResNet101-based DeepLabv3+ (Chen et al., 2018a) and RefineNet (Lin et al., 2017), with 40 and 118 million weights. Respectively, our model are around 15% and 50% faster in terms of inference frame per seconds on commodity hardware. This helps explain why we can outperform state-of-the-art networks on specialized tasks: The pre-trained ImageNet features are not necessarily the best and training sets are not quite as large as CityScapes (Cordts et al., 2016). As a result, the large networks tend to overfit and do not perform as well as compact models trained from scratch.

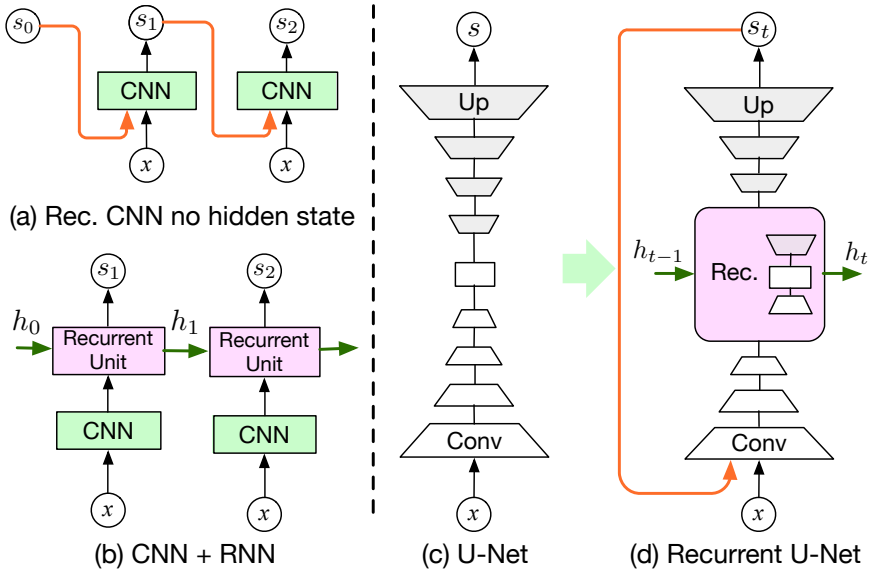



Figure 2.2: **Recurrent segmentation. (a)** The simple strategy (Mosinska et al., 2018; Pinheiro & Collobert, 2014) consists of concatenating the segmentation mask from the previous recurrence, $s_{t-1}$, to the image $x$, and recurrently feeding this to the network. **(b)** For sequence segmentation, to account for the network's internal state, one can instead combine the CNN with a standard recurrent unit (Valipour et al., 2017). Here, we build upon the U-Net architecture (Ronneberger et al., 2015) **(c)**, and propose to build a recurrent unit over several of its layers, as shown in **(d)**. This allows us to propagate higher-level information through the recurrence, and, in conjunction with a recurrence on the segmentation mask, outperforms the two simpler recurrent architectures **(a)** and **(b)**.

The standard U-Net takes the image as input, processes it, and directly returns an output. By contrast, our recurrent architecture iteratively refines both the segmentation mask and the network's internal state. This mimics human perception as in the influential AutoContext paper (Tu & Bai, 2009): When we observe a scene, our eyes undergo saccadic movements, and we accumulate knowledge about the scene and continuously refine our perception (Purves et al., 2011). To this end we retain the overall structure of the U-Net, but build a recurrent unit over some of its inner layers for internal state update. Note that our recurrence works on the same input image. By contrast with the simple CNN+RNN architecture of Figure 2.2(b), often used for video or volumetric segmentation (Valipour et al., 2017; Poudel et al., 2016; Ballas et al., 2016), this enables the network to keep track of and to iteratively update more than just a single-layer internal state. This gives us the flexibility to choose the portion of the internal state that we exploit for recursion purposes and to explore variations of our scheme.

We demonstrate the benefits of our recurrent U-Net on several tasks, including hand segmentation, retina vessel segmentation and road segmentation. Our approach consistently outperforms earlier and simpler approaches to recursive segmentation (Mosinska et al., 2018; Poudel et al., 2016; Valipour et al., 2017). For retina vessel segmentation, it also outperforms the state-of-the-art method of (Maninis et al., 2016) on the DRIVE (Staal et al., 2004) dataset, and for hand

segmentation, the state-of-the-art RefinetNet-based method of (Urooj & Borji, 2018) on several modern benchmarks (Fathi et al., 2011; Bambach et al., 2015; Urooj & Borji, 2018). As these publicly available hand segmentation datasets are relatively small, with at most 4.8K annotated images, we demonstrate the scalability of our approach, along with its applicability in a keyboard typing scenario, by introducing a larger dataset containing 12.5K annotated images. It is the one we used to produce the results shown in Figure 2.1.

Our contribution is therefore an effective recurrent approach to semantic segmentation that can operate in environments where the amount of training data and computational power are limited. It does not require more memory than the standard U-Net thanks to parameter sharing and does not require training datasets as large as other state-of-the-art networks do. It is practical for real-time application, reaching 55 frames-per-second (fps) to segment 230×306 images on an NVIDIA TITAN X with 12G memory. Furthermore, as shown in Figure 2.1, we can trade some accuracy for speed by reducing the number of iterations. Finally, while we focus on resource-constrained applications, our model can easily be made competitive on standard benchmarks such as Cityscapes by modifying its backbone architecture. We will show that replacing the U-Net encoder by a VGG16 backbone yields performance numbers comparable to the state of the art on this dataset.

## 2.2 Related Work

**Compact Semantic Segmentation Models.** State-of-the-art semantic segmentation techniques (Chen et al., 2018a; Lin et al., 2017; Long et al., 2015; Zhao et al., 2017) rely on very deep networks, which makes them ill-suited in resource-constrained scenarios, such as real-time applications and when there are only limited amounts of training data. In such cases, more compact networks are preferable. Such networks fall under two main categories.

The first group features encoder-decoder architectures (Ronneberger et al., 2015; Poudel et al., 2019; Badrinarayanan et al., 2015; Romera et al., 2018; Pohlen et al., 2017; Saxena & Verbeek, 2016; Fourure et al., 2017). Among those, U-Net (Ronneberger et al., 2015) has demonstrated its effectiveness and versatility on many tasks, in particular for biomedical image analysis where it remains a favorite. For example, a U-net like architecture was recently used to implement the flood-filling networks of Januszewski et al. (2018) and to segment densely interwoven neurons and neurites in teravoxel-scale 3D electron-microscopy image stacks. This work took advantage of the immense amount of computing power that Google can muster but, even then, it is unlikely that this could have been accomplished with much heavier architectures.

The second type involves multi-branch structures (Poudel et al., 2018; Yu et al., 2018; Zhao et al., 2018) to fuse low-level and high-level features at different resolutions. These require careful design to balance speed against performance. By contrast, the U-Net relies on simpler skip connections and, thus, does not require a specific design, which has greatly contributed to its popularity.

**Recurrent Networks for Segmentation.** The idea of recurrent segmentation predates the deep learning era and was first proposed in AutoContext (Tu & Bai, 2009), and recurrent random forest (Shotton et al., 2008). It has inspired many recent approaches, including several that rely on deep networks. For example, Mosinska et al. (2018) proposes the segmentation mask produced by a modified U-Net was passed back as input to it along with the original image, which resulted in a progressive refinement of the segmentation mask. Figure 2.2(a) illustrates this approach. A similar one was followed in the earlier work of Pinheiro & Collobert (2014), where the resolution of the input image patch varied across the iterations of the refinement process.

Instead of including the entire network in the recursive procedure, a standard recurrent unit can be added at the output of the segmentation network, as shown in Figure 2.2(b). This was done in Romera-Paredes & Torr (2016) to iteratively produce individual segmentation masks for scene objects. In principle, such a convolutional recurrent unit (Ballas et al., 2016; Poudel et al., 2016; Valipour et al., 2017) could also be applied for iterative segmentation of a single object and we will evaluate this approach in our experiments. We depart from this strategy by introducing gated recurrent units that encompass several U-Net layers. Furthermore, we leverage the previous segmentation results as input, not just the same image at every iteration.

Iterative refinement has also been used for pose estimation (Ramakrishna et al., 2014; Wei et al., 2016; Newell et al., 2016). The resulting methods all involve consecutive modules to refine the predictions with a loss function evaluated on the output of each module, which makes them similar in spirit to the model depicted by Figure 2.2(a). Unlike in our approach, these methods do not share the parameters across the consecutive modules, thus requiring more parameters and moving away from our aim to obtain a compact network. Furthermore, they do not involve RNN-inspired memory units to track the internal hidden state.

## 2.3 Method

We now introduce our novel recurrent semantic segmentation architecture. To this end, we first discuss the overall structure of our framework, and then provide the details of the recurrent unit it relies on. Finally, we briefly discuss the training strategy for our approach.

### 2.3.1 Recurrent U-Net

We rely on the U-Net architecture (Ronneberger et al., 2015) as backbone to our approach. As shown in Figure 2.3(a), the U-Net has an encoder-decoder structure, with skip connections between the corresponding encoding and decoding layers that allow the network to retain low-level features for the final prediction. Our goal being to operate in resource-constrained environments, we want to keep the model relatively simple. We therefore rely on a U-Net design where the first convolutional unit has 8 feature channels, and, following the original U-Net strategy, the channel number doubles after every pooling layer in the encoder. The decoder relies on transposed

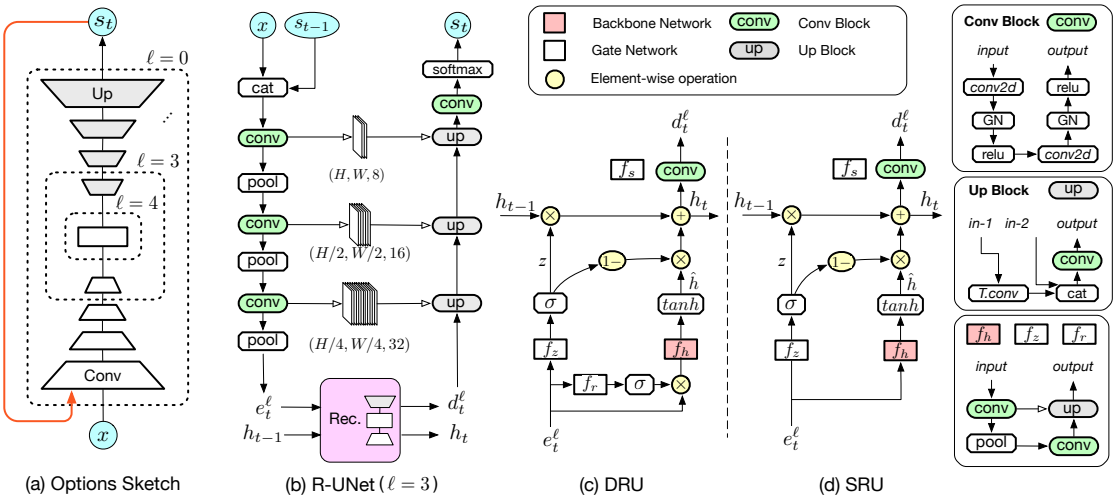



Figure 2.3: **Recurrent UNet (R-UNet). (a)** As illustrated in Fig. 2.2(d), our model incorporates several encoding and decoding layers in a recurrent unit. The choice of which layers to englobe is defined by the parameter $\ell$. **(b)** For $\ell = 3$, the recurrence occurs after the third pooling layer in the U-Net encoder. The output of the recurrent unit is then passed through three decoding up-convolution blocks. We design two different recurrent units, the Dual-gated Recurrent Unit (DRU) **(c)** and the Single-gated Recurrent Unit (SRU) **(d)**. They differ by the fact that the first one has an additional reset gate acting on its input. See the main text for more detail.

convolutions to increase the model's representation power compared to bilinear interpolation. We use group-normalization (Wu & He, 2018) in all convolutional layers since we usually rely on very small batch sizes.

Our contributions are to integrate recursions on 1) the predicted segmentation mask $s$ and 2) multiple internal states of the network. which is inspired from the hidden state of recurrent neural network. In short, our recurrent U-Net group the mask $s$ from previous time-step, with the input image as an input to the next time-step. The former can be achieved by simply concatenating, at each recurrent iteration $t$, the previous segmentation mask $s_{t-1}$ to the input image, and passing the resulting concatenated tensor through the network. For the latter, we propose to replace a subset of the encoding and decoding layers of the U-Net with a recurrent unit. Below, we first formalize this unit, and then discuss two variants of its internal mechanism.

To formalize our recurrent unit, let us consider the process at iteration $t$ of the recurrence. At this point, the network takes as input an image $x$ concatenated with the previously-predicted segmentation mask $s_{t-1}$. Let us then denote by $e_t^\ell$ the activations of the $\ell^{th}$ encoding layer, and by $d_t^\ell$ those of the corresponding decoding layer. Our recurrent unit takes as input $e_t^\ell$, together with its own previous hidden tensor $h_{t-1}$, and outputs the corresponding activations $d_t^\ell$, along with the new hidden tensor $h_t$. Note that, to mimic the computation of the U-Net, we use multiple encoding and decoding layers within the recurrent unit.

In practice, one can choose the specific level $\ell$ at which the recurrent unit kicks in. In Figure 2.3 (b), we illustrate the whole process for $\ell = 3$. When $\ell = 0$, the entire U-Net is included in

the recurrent unit, which then takes the concatenation of the segmentation mask and the image as input. Note that, for $\ell = 4$, the recurrent unit still contains several layers because the central portion of the U-Net in Figure 2.3(a) corresponds to a convolutional *block*. In our experiments, we evaluate two different structures for the recurrent units, which we discuss below.

### 2.3.2 Dual-gated Recurrent Unit

As a first recurrent architecture, we draw inspiration from the Gated Recurrent Unit (GRU) (Cho et al., 2014). As noted above, however, our recurrent unit replaces multiple encoding and decoding layers of the segmentation network. We therefore modify the equations accordingly, but preserve the underlying motivation of GRUs. Our architecture is shown in Figure 2.3(c).

Specifically, at iteration $t$, given the activations $e_t^\ell$ and the previous hidden state $h_{t-1}$, we aim to produce a candidate update $\hat{h}$ for the hidden state and combine it with the previous one according to how reliable the different elements of this previous hidden state tensor are. To determine this reliability, we use an update gate defined by a tensor

$$z = \sigma(f_z(e_t^\ell)) \,, \tag{2.1}$$

where $f_z(\cdot)$ denotes an encoder-decoder network with the same architecture as the portion of the U-Net that we replace with our recurrent unit.

Similarly, we obtain the candidate update as

$$\hat{h} = \texttt{tanh}(f_h(r \odot e_t^\ell)) \,, \tag{2.2}$$

where $f_h(\cdot)$ is a network with the same architecture as $f_z(\cdot)$, but a separate set of parameters, $\odot$ denotes the element-wise product, and $r$ is a reset tensor allowing us to mask parts of the input used to compute $\hat{h}$. It is computed as

$$r = \sigma(f_r(e_t^\ell)) \,, \tag{2.3}$$

where $f_r(\cdot)$ is again a network with the same encoder-decoder architecture as before.

Given these different tensors, the new hidden state is computed as

$$h_t = z \odot h_{t-1} + (1 - z) \odot \hat{h} \,. \tag{2.4}$$

Finally, we predict the output of the recurrent unit, which corresponds to the activations of the $\ell^{th}$ decoding layer as

$$d_t^\ell = f_s(h_t) \,, \tag{2.5}$$

where, as shown in Figure 2.3(c), $f_s(\cdot)$ is a simple convolutional block. Since it relies on two gates, $r$ and $z$, we dub this recurrent architecture Dual-gated Recurrent Unit (DRU). One main

difference with GRUs is the fact that we use multi-layer encoder-decoder networks in the inner operations instead of simple linear layers. Furthermore, in contrast to GRUs, we do not directly make use of the hidden state $h_{t-1}$ in these inner computations. This allows us not to have to increase the number of channels in the encoding and decoding layers compared to the original U-Net. Nevertheless, the hidden state is indirectly employed, since, via the recursion, $e_t^\ell$ depends on $d_{t-1}^\ell$, which is computed from $h_{t-1}$.

### 2.3.3 Single-Gated Recurrent Unit

As evidenced by our experiments, the DRU described above is effective at iteratively refining a segmentation. However, it suffers from the drawback that it incorporates three encoder-decoder networks, which may become memory-intensive depending on the choice of $\ell$. To decrease this cost, we therefore introduce a simplified recurrent unit, which relies on a single gate, thus dubbed Single-gated Recurrent Unit (SRU).

Specifically, as illustrated in Figure 2.3(d), our SRU has a structure similar to that of the DRU, but without the reset tensor $r$. As such, the equations remain mostly the same as above, with the exception of the candidate hidden state, which we now express as

$$\hat{h} = \texttt{tanh}(f_h(e_t^\ell)) \,. \tag{2.6}$$

This simple modification allows us to remove one of the encoder-decoder networks from the recurrent unit, which, as shown by our results, comes at very little loss in segmentation accuracy.

### 2.3.4 Training

To train our recurrent U-Net, we use the cross-entropy loss. More specifically, we introduce supervision at each iteration of the recurrence. To this end, we write our overall loss as

$$L = \sum_{t=1}^{N} w_t L_t, \tag{2.7}$$

where $N$ represents the number of recursions, set to 3 in this paper, and $L_t$ denotes the cross-entropy loss at iteration $t$, which is weighted by $w_t$.

$$w_t = \alpha^{N-t}. \tag{2.8}$$

The weight, by setting $\alpha \leq 1$, increases monotonically with the iterations. In our experiments, we either set $\alpha = 1$, so that all iterations have equal importance, or $\alpha = 0.4$, thus encoding the intuition that we seek to put more emphasis on the final prediction.

Figure 2.4: **Keyboard Hand (KBH) dataset.** Sample images featuring diverse environmental and lighting conditions, along with associated ground-truth segmentations.

## 2.4 Experiments

We compare the two versions of our Recurrent U-Net against the state of the art on several tasks including hand segmentation, retina vessel segmentation and road delineation. We further demonstrate that the core idea behind our idea also applies to non-resource-constrained scenarios, such as Cityscapes, by increasing the size of the U-Net encoder.

### 2.4.1 Datasets

**Hands.** We report the performance of our approach on standard hand-segmentation benchmarks, such as GTEA (Fathi et al., 2011), EYTH (Urooj & Borji, 2018), EgoHand (Bambach et al., 2015), and HOF (Urooj & Borji, 2018). These, however, are relatively small, with at most 4,800 images in total, as can be seen in Table 2.1. To evaluate our approach on a larger dataset, we therefore acquired our own. Because this work was initially motivated by an augmented virtuality project whose goal is to allow someone to type on a keyboard while wearing a head-mounted display, we asked 50 people to type on 9 keyboards while wearing an HTC Vive (HTC). To make this easier, we created a mixed-reality application to allow the users to see both the camera view and a virtual browser showing the text being typed. To ensure diversity, we varied the keyboard types, lighting conditions, desk colors, and objects lying on them, as can be seen in Figure 2.4. We provide additional details in Table 2.2.

We then recorded 161 hand sequences with the device's camera. We split them as 20/ 20/

| | Resolution | | # Images | | | |
|---|---|---|---|---|---|---|
| Dataset | Width | Height | Train | Val. | Test | Total |
| KBH (Ours) | 230 | 306 | 2300 | 2300 | 7936 | 12536 |
| EYTH (Urooj & Borji, 2018) | 216 | 384 | 774 | 258 | 258 | 1290 |
| HOF (Urooj & Borji, 2018) | 216 | 384 | 198 | 40 | 62 | 300 |
| EgoHand (Bambach et al., 2015) | 720 | 1280 | 3600 | 400 | 800 | 4800 |
| GTEA(Fathi et al., 2011) | 405 | 720 | 367 | 92 | 204 | 663 |

Table 2.1: **Hand-segmentation benchmark datasets.**

(a) Environment setup

| Parameters | Amount | Details |
|---|---|---|
| Desk | 3 | White, Brown, Black |
| Desk position | 3 | - |
| Keyboard | 9 | - |
| Lighting | 8 | 3 sources on/off |
| Objects on desk | 3 | 3 different objects |

(b) Attributes

| Attribute | #IDs |
|---|---|
| Bracelet | 10 |
| Watch | 14 |
| Brown-skin | 2 |
| Tatoo | 1 |
| Nail-polish | 1 |
| Ring(s) | 6 |

Table 2.2: **Properties of our new KBH dataset.**

60% for train/ validation/ test to set up a challenging scenario in which the training data is not overabundant and to test the scalability and generalizability of the trained models. We guaranteed that the same person never appears in more than one of these splits by using people's IDs during partitioning. In other words, our splits resulted in three groups of 30, 30, and 101 separate videos, respectively. We annotated about the same number of frames in each one of the videos, resulting in a total of 12,536 annotated frames.

**Retina Vessels.** We used the popular DRIVE dataset (Staal et al., 2004). It contains 40 retina images used for making clinical diagnoses, among which 33 do not show any sign of diabetic retinopathy and 7 show signs of mild early diabetic retinopathy. The images have been divided into a training and a test set with 20 images for each set.

**Roads.** We used the Massachusetts Roads dataset (Mnih, 2013). It is one of the largest publicly available collections of aerial road images, containing both urban and rural neighborhoods, with many different kinds of roads ranging from small paths to highways. The data is split into 1108 training and 49 test images, one of which is shown in Figure 2.6.

**Urban landscapes.** We employed the recent Cityscapes dataset. It is a very challenging dataset with high-resolution 1024×2048 images. It has 5,000 finely annotated images which are split into training/validation/test sets with 2975/500/1525 images. 30 classes are annotated, and 19 of them are used in training and testing.

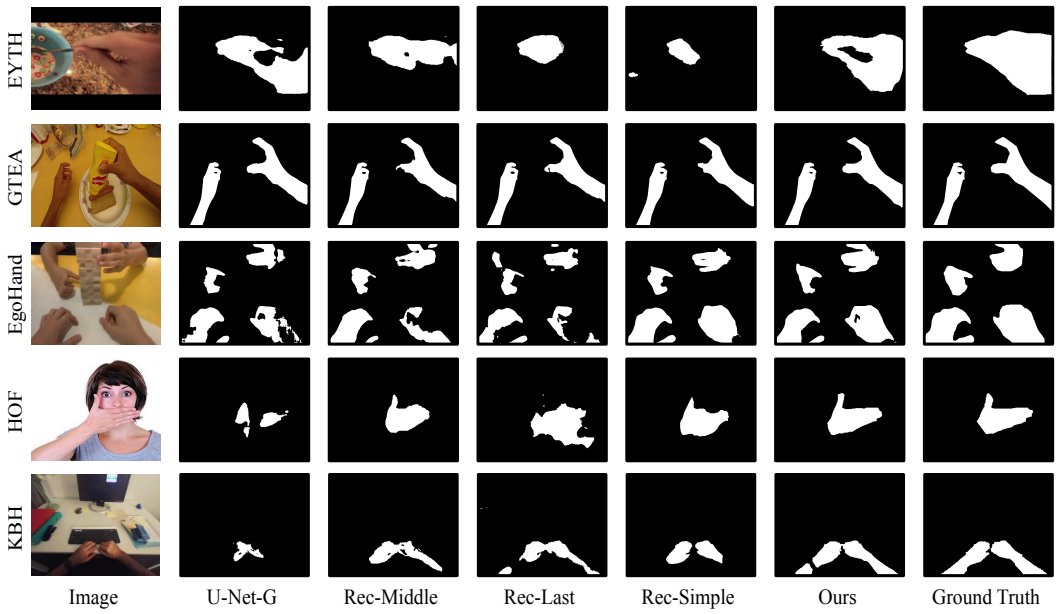



Figure 2.5: **Example predictions on hand segmentation datasets.** Note that our method yields accurate segmentations in diverse conditions, such as with hands close to the camera, multiple hands, hands over other skin regions, and low contrast images in our KBH dataset. By contrast, the baselines all fail in at least one of these scenarios. Interestingly, our method sometimes yields a seemingly a more accurate segmentation than the ground-truth ones. For example, in our EYTH result at the top, the gap between the thumb and index finger is correctly found whereas it is missing from the ground truth. Likewise, for KBH at the bottom, the watch band is correctly identified as not being part of the arm even though it is labeled as such in the ground truth.

### 2.4.2 Experimental Setup

#### Baselines

We refer to the versions of our approach that rely on the dual gated unit of Section 2.3.2 and the single gated unit of Section 2.3.3 as *Ours-SRU* and *Ours-DRU*, respectively, with, e.g., *Ours-SRU*(3) denoting the case where $\ell = 3$ in Figure 2.3. We compare them against the state-of-the-art model for each task, i.e., *RefineNet* (Urooj & Borji, 2018) for hand segmentation, (Maninis et al., 2016) for retina vessel segmentation and (Mosinska et al., 2018) for road delineation, the general purpose DeepLab V3+ (Chen et al., 2018a), the real-time ICNet (Zhao et al., 2018), and the following baselines.

- ***U-Net*-B** and ***U-Net*-G** (Ronneberger et al., 2015). We treat our U-Net backbone by itself as a baseline. *U-Net*-B uses batch-normalization and *U-Net*-G group-normalization. For a fair comparison, they, *Ours-SRU*, *Ours-DRU*, and the recurrent baselines introduced below all use the same parameter settings.

- ***Rec-Last***. It has been proposed to add a recurrent unit after a convolutional segmentation network to process sequential data, such as video (Poudel et al., 2016). The corresponding *U-Net*-based architecture can be directly applied to segmentation by inputing the same image at all time steps, as shown in Figure 2.2(b). The output then evolves as the hidden state is updated.

- ***Rec-Middle***. Similarly, the recurrent unit can replace the bottleneck between the U-Net encoder and decoder, instead of being added at the end of the network. This has been demonstrated to handle volumetric data (Valipour et al., 2017). Here we test it for segmentation. The hidden state then is of the same size as the inner feature backbone, that is, 128 in our experimental setup.

- ***Rec-Simple*** (Mosinska et al., 2018). We perform a recursive refinement process, that is, we concatenate the segmentation mask with the input image and feed it into the network. Note that the original method of (Mosinska et al., 2018) relies on a VGG-19 pre-trained on ImageNet (Simonyan & Zisserman, 2015), which is far larger than our *U-Net*. To make the comparison fair, we therefore implement this baseline with the same U-Net backbone as in our approach.

#### Scaling Up using Pretrained Deep Networks as Encoder

While our goal is resource-constrained segmentation, our method extends to the general setting. In this case, to further boost its performance, we replace the U-Net encoder with a pretrained VGG-16 backbone. We refer to the corresponding models as U-Net-VGG16 and DRU-VGG16.

**Metrics.** We report the mean intersection over union (mIoU), mean recall (mRec) and mean precision (mPrec).

| | Model | EYTH | | | GTEA | | | EgoHand | | | HOF | | | KBH | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | mIOU | mRec | mPrec | mIOU | mRec | mPrec | mIOU | mRec | mPrec | mIOU | mRec | mPrec | mIOU | mRec | mPrec |
| Light | *No pre-train* | | | | | | | | | | | | | | | |
| | ICNet | 0.731 | 0.915 | 0.764 | 0.898 | 0.971 | 0.922 | 0.872 | **0.925** | 0.931 | 0.580 | 0.801 | 0.628 | 0.829 | 0.925 | 0.876 |
| | U-Net-B | 0.803 | 0.912 | 0.830 | 0.950 | 0.973 | 0.975 | 0.815 | 0.869 | 0.876 | 0.694 | **0.867** | 0.778 | 0.870 | 0.943 | 0.911 |
| | U-Net-G | 0.837 | 0.928 | 0.883 | 0.952 | 0.977 | 0.980 | 0.837 | 0.895 | 0.899 | 0.621 | 0.741 | 0.712 | 0.905 | 0.949 | 0.948 |
| | Rec-Middle | 0.827 | 0.920 | 0.877 | 0.924 | **0.979** | 0.976 | 0.828 | 0.894 | 0.905 | 0.654 | 0.733 | **0.796** | 0.845 | 0.924 | 0.898 |
| | Rec-Last | 0.838 | 0.920 | 0.894 | 0.957 | 0.975 | 0.980 | 0.831 | 0.906 | 0.897 | 0.674 | 0.807 | 0.752 | 0.870 | 0.930 | 0.924 |
| | Rec-Simple | 0.827 | 0.918 | 0.864 | 0.952 | 0.975 | 0.976 | 0.858 | 0.909 | 0.931 | 0.693 | 0.833 | 0.704 | 0.905 | 0.951 | 0.944 |
| | *Ours at layer ($\ell$)* | | | | | | | | | | | | | | | |
| | Ours-SRU(0) | 0.844 | 0.924 | 0.890 | **0.960** | 0.976 | 0.981 | 0.862 | 0.913 | 0.932 | **0.712** | 0.844 | 0.764 | 0.930 | 0.968 | 0.957 |
| | Ours-SRU(3) | 0.845 | **0.931** | 0.891 | 0.956 | 0.977 | **0.982** | 0.864 | 0.913 | 0.933 | 0.699 | 0.864 | 0.773 | 0.921 | 0.964 | 0.951 |
| | Ours-DRU(4) | **0.849** | 0.926 | **0.900** | 0.958 | 0.978 | 0.977 | **0.873** | 0.924 | **0.935** | 0.709 | 0.866 | 0.774 | **0.935** | **0.980** | **0.970** |
| Heavy | *With pretrain* | | | | | | | | | | | | | | | |
| | RefineNet | 0.688 | 0.776 | 0.853 | 0.821 | 0.869 | 0.928 | 0.814 | 0.919 | 0.879 | 0.766 | 0.882 | 0.859 | 0.865 | 0.954 | 0.921 |
| | Deeplab V3+ | 0.757 | 0.819 | 0.875 | 0.907 | 0.928 | 0.976 | 0.870 | 0.909 | **0.958** | 0.722 | 0.822 | 0.816 | 0.856 | 0.901 | 0.935 |
| | U-Net-VGG16 | 0.879 | 0.945 | 0.921 | 0.961 | 0.978 | 0.981 | 0.879 | 0.916 | 0.951 | 0.849 | 0.937 | 0.893 | 0.946 | 0.971 | 0.972 |
| | DRU-VGG16 | **0.897** | **0.946** | **0.940** | **0.964** | **0.981** | **0.982** | **0.892** | **0.925** | **0.958** | **0.863** | **0.948** | **0.901** | **0.954** | **0.973** | **0.979** |

Table 2.3: **Comparing against the state of the art.** According to the mIOU, *Ours-DRU*(4) performs best on average, with *Ours-SRU*(0) a close second. Generally speaking all recurrent methods do better than *RefineNet*, which represents the state of the art, on all datasets except HOF. We attribute this to HOF being too small for optimal performance without pre-training, as in *RefineNet*. This is confirmed by looking at DRU-VGG16, which yields the overall best results by relying on a pretrained deep backbone.

### 2.4.3 Comparison to the State of the Art

We now compare the two versions of our approach to the state of the art and to the baselines introduced above on the tasks of hand segmentation, retina vessel segmentation and road delineation. We split the methods into the light ones and the heavy ones. The light models contain fewer parameters and are trained from scratch, whereas the heavy ones use a pretrained deep model as backbone. Furthermore, we provide a qualitative assessment of our model for recurrent refinement in Figure 2.6.

**Hands.** As discussed in Section 2.4.1, we tested our approach using 4 publicly available datasets and our own large-scale one. We compare it against the baselines in Table 2.3 quantitatively and in Figure 2.5 qualitatively.

Overall, among the light models, the recurrent methods usually outperform the one-shot ones, *i.e*, ICNet (Zhao et al., 2018) and *U-Net*. Besides, among the recurrent ones, *Ours-DRU*(4) and *Ours-SRU*(0) clearly dominate with *Ours-DRU*(4) usually outperforming *Ours-SRU*(0) by a small margin. Note that, even though *Ours-DRU*(4) as depicted by Figure 2.3(a) looks superficially similar to *Rec-Middle*, they are quite different because *Ours-DRU* takes the segmentation mask as input and relies on our new DRU gate, as discussed at the end of Section 2.3.1 and in Section 2.3.2. To confirm this, we evaluated a simplified version of *Ours-DRU*(4) in which we removed the segmentation mask from the input. The validation mIOU on EYTH decreased from 0.836 to 0.826 but remained better than that of *Rec-Middle* which is 0.814.

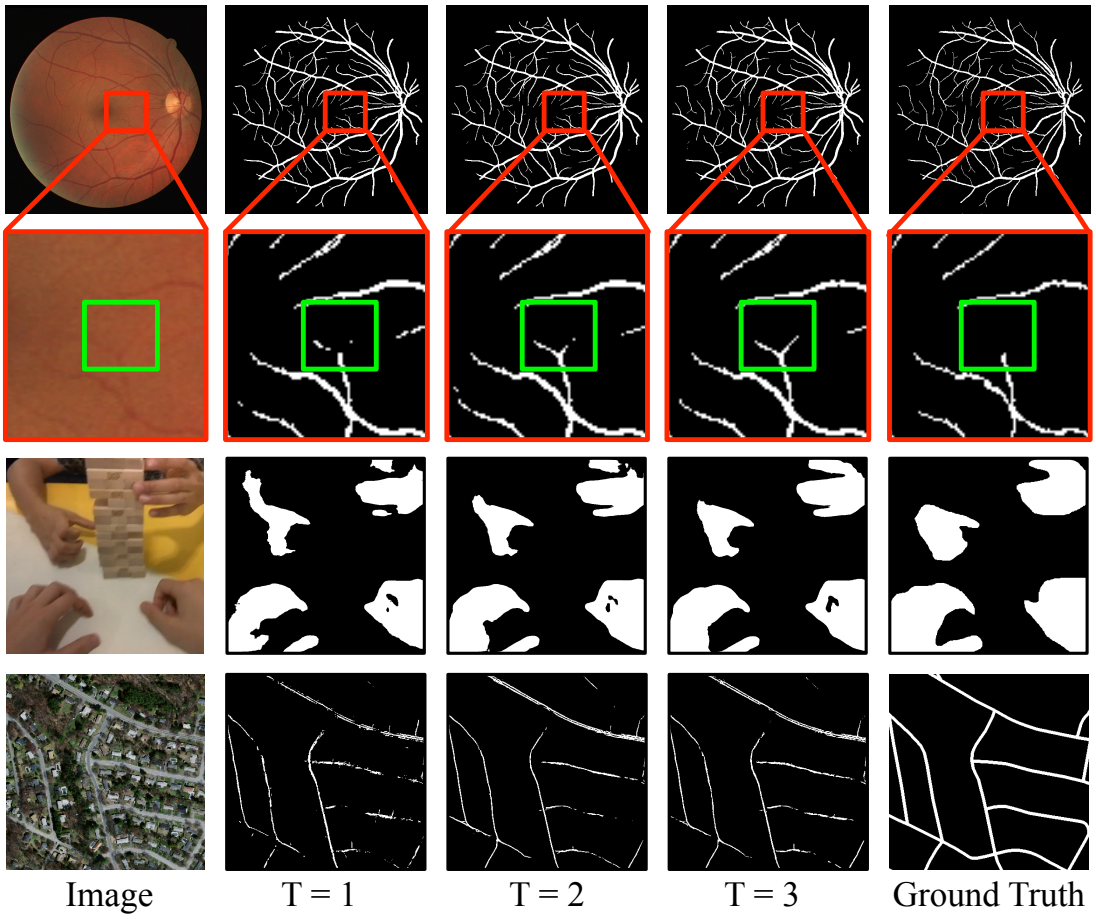



Figure 2.6: **Recursive refinement.** Retina, hand and road images; segmentation results after 1, 2, and 3 iterations; ground truth. Note the progressive refinement and the holes of the vessels, hands and roads being filled recursively. It is worth pointing out that even the tiny vessel branches in the retina which are ignored by the human annotators could be correctly segmented by our algorithm. Better viewed in color and zoom in.

Note that *Ours-DRU*(4) is better than the heavy *RefineNet* model on 4 out of the 5 datasets, despite *RefineNet* representing the current state of the art. The exception is HOF, and we believe that this can be attributed to HOF being the smallest dataset, with only 198 training images. Under such conditions, *RefineNet* strongly benefits from exploiting a ResNet-101 backbone that was pre-trained on PASCAL person parts (Chen et al., 2014), instead of training from scratch as we do. This intuition is confirmed by looking at the results of our DRU-VGG16 model, which, by using a pretrained deep backbone, yields the overall best performance.

**Model Performance, Size and Speed.** Table 2.3 shows that DRU-VGG16 outperforms *Ours-DRU*, e.g., by 0.02 mIoU points on KBH. This, however, comes at a cost. To be precise, DRU-VGG16 has 41.38M parameters. This is 100 times larger than *Ours-DRU*(4), which has only 0.36M parameters. Moreover, DRU-VGG16 runs only at 18 fps, while *Ours-DRU*(4) reaches

| | Models | mIOU | mRec | mPrec | mF1 |
|---|---|---|---|---|---|
| Light | ICNet (Zhao et al., 2018) | 0.618 | 0.796 | 0.690 | 0.739 |
| | U-Net-G (Ronneberger et al., 2015) | 0.800 | 0.897 | 0.868 | 0.882 |
| | Rec-Middle (Poudel et al., 2016) | 0.818 | **0.903** | 0.886 | 0.894 |
| | Rec-Simple (Mosinska et al., 2018) | 0.814 | 0.898 | 0.885 | 0.892 |
| | Rec-Last (Valipour et al., 2017) | 0.819 | 0.900 | 0.890 | 0.895 |
| | Ours-DRU(4) | **0.821** | 0.902 | **0.891** | **0.896** |
| Heavy | DeepLab V3+ (Chen et al., 2018a) | 0.756 | 0.875 | 0.828 | 0.851 |
| | U-Net-VGG16 | 0.804 | **0.910** | 0.862 | 0.886 |
| | DRU-VGG16 | **0.817** | 0.905 | **0.883** | **0.894** |

Table 2.4: **Retina vessel segmentation results.**

61 fps. This makes DRU-VGG16, and the other heavy models, ill-suited to embedded systems, such as a VR camera, while *Ours-DRU* can more easily be exploited in resource-constrained environments.

**Retina Vessels.**

We report our results in Table 2.4. Our DRU yields the best mIOU, mPrec and mF1 scores. Interestingly, on this dataset, it even outperforms the larger DRU-VGG16 and DeepLab V3+, which performs comparatively poorly on this task. This, we believe, is due to the availability of only limited data, which leads to overfitting for such a very deep network. Note also that retina images significantly differ from the ImageNet ones, thus reducing the impact of relying on pretrained backbones. On this dataset, (Maninis et al., 2016) constitutes the state of the art, reporting an F1 score on the vessel class only of 0.822. According to this metric, *Ours-DRU*(4) achieves 0.92, thus significantly outperforming the state of the art.

**Roads.** Our results on road segmentation are provided in Table 2.5. We also outperform all the baselines by a clear margin on this task, with or without ImageNet pre-training. In particular, Ours-DRU(4) yields an mIoU 8 percentage point (pp) higher than U-Net-G, and DRU-VGG16 5pp higher than U-Net-VGG16. This verifies that our recurrent strategy helps. Furthermore, Ours-DRU(4) also achieves a better performance than DeepLab V3+ and U-Net-VGG16. Note that, here, we also report two additional metrics: Precision-recall breaking point (P/R) and F1-score. The cutting threshold for all metrics is set to 0.5 except for P/R. For this experiment, we did not report the results of *U-Net*-B because *U-Net*-G is consistently better.

Note that a P/R value of 0.778 has been reported on this dataset in (Mosinska et al., 2018). However, this required using an additional topology-aware loss and a *U-Net* much larger than ours, that is, based on 3 layers of a VGG19 pre-trained on ImageNet. Rec-Simple duplicates the approach of (Mosinska et al., 2018) without the topology-aware loss and with the same *U-Net* as Ours-DRU. Their mIoU of 0.723, inferior to ours of 0.757, shows our approach to recursion to be beneficial.

| | Models | mIOU | mRec | mPrec | P/R | mF1 |
|---|---|---|---|---|---|---|
| Light | ICNet (Zhao et al., 2018) | 0.476 | 0.626 | 0.500 | 0.513 | 0.656 |
| | U-Net-G (Ronneberger et al., 2015) | 0.479 | 0.639 | 0.502 | 0.642 | 0.563 |
| | Rec-Middle (Poudel et al., 2016) | 0.494 | 0.767 | 0.518 | 0.660 | 0.574 |
| | Rec-Simple (Mosinska et al., 2018) | 0.534 | 0.802 | 0.559 | 0.723 | 0.659 |
| | Rec-Last (Valipour et al., 2017) | 0.526 | 0.786 | 0.551 | 0.730 | 0.648 |
| | Ours-DRU(4) | **0.560** | **0.865** | **0.583** | **0.757** | **0.691** |
| Heavy | Deeplab V3+ (Chen et al., 2018a) | 0.529 | 0.763 | 0.555 | 0.710 | 0.643 |
| | U-Net-VGG16 | 0.521 | 0.836 | 0.544 | 0.745 | 0.659 |
| | DRU-VGG16 | **0.571** | **0.862** | **0.595** | **0.761** | **0.704** |

Table 2.5: **Road segmentation results.**

| Model | mIoU | Model | mIoU |
|---|---|---|---|
| ICNet(Zhao et al., 2018) | 0.695 | DeepLab V3 (Chen et al., 2017) | 0.778 |
| U-Net-G | 0.429 | U-Net-G ×2 | 0.476 |
| Rec-Last | 0.502 | Rec-Last ×2 | 0.521 |
| DRU(4) | 0.532 | DRU(4) ×2 | 0.627 |
| | | DRU-VGG16 | 0.761 |

Table 2.6: **Cityscapes Validation Set with Resolution** 1024×2048**.** ×2 indicates that we doubled the number of channels in the U-Net backbone. Note that, for our method, we do not use multi-scaling or horizontal flips during inference.

**Urban landscapes.**

The segmentation results on the Cityscapes validation set are shown in Table 2.6. Note that *Ours-DRU* is consistently better than U-Net-G and than the best recurrent baseline, *i.e.*, Rec-Last. Furthermore, doubling the number of channels of the U-Net backbone increases accuracy, and so does using a pretrained VGG-16 as encoder. Ultimately, our DRU-VGG16 model yields comparable accuracy with the state-of-the-art DeepLab V3 one, despite its use of a ResNet101 backbone.

## 2.5 Conclusion

We have introduced a novel recurrent *U-Net* architecture that preserves the compactness of the original one, while substantially increasing its performance. At its heart is the fact that the recurrent units encompass several encoding and decoding layers of the segmentation network. We also introduced a new hand segmentation dataset that is larger than existing ones.

In the first part of this thesis, we have presented two approaches focusing on different tasks but relying on the manual design of convolutional neural network architectures. While our methods have shown promising results on those two tasks, we have discovered during our research that such design process can be separated into two parts: i) Designing the network operations or overall architecture based on some task-dependent knowledge, such as covariance descriptors for fine-grained image classification; ii) finetuning the architecture little by little based on empirical

results. While the first part is not easily automated, the recent advances in automatic machine learning (AutoML) provide a revolutionary solution for the second part of this process. In the next part of this thesis, we introduce our contributions in neural architecture search, a promising AutoML direction.

# Part II

# Towards Robust Neural Architecture Search with Parameter Sharing

Neural architecture search (NAS), aims to facilitate the design of deep networks for new tasks, and has drawn an increasing attention recently. Its essential idea is to pick a candidate neural architecture from a pre-defined search space, train the architecture and obtain the metrics of interest. The weight sharing approach, which utilizes a super-net to encompass all possible architectures within the search space, has become a de facto standard in NAS because it enables the search to be done on commodity hardware. In general, the super-net is defined as a composition of all weights within the search space. In this part, in Chapter 3, we will introduce the concept of NAS and weight sharing in detail. We then identify several weaknesses of weight-sharing NAS. In Chapter 5, we observe that, (i) on average, many popular NAS algorithms perform similarly to a random architecture sampling policy; (ii) this widely-adopted weight sharing strategy degrades the ranking of the NAS candidates to the point of not reflecting their true performance, thus reducing the effectiveness of the search process. In Chapter 6, we further decouple weight sharing from the NAS sampling policy, and isolate 14 factors of super-net training. To further improve the super-net quality, in Chapter 7, we propose a regularization term that aims to maximize the correlation between the performance rankings of the super-net of the stand-alone architectures using a small set of landmark architectures.

# 3 Neural Architecture Search – Preliminaries

## 3.1 Problem Definition

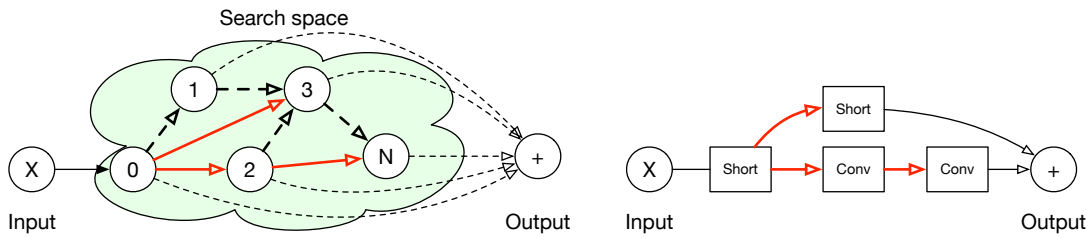
As discussed in the previous chapters, when we design a novel architecture for a specific computer vision task, we typically rely on our previous experience and try to translate successful designs to the task of interest, further exploiting some domain knowledge or requirements. For example, our recurrent model was motivated by the saccadic movements of the human eye and constructed manually. Nevertheless, the final architecture was obtained via multiple rounds of fine-tuning the design and its hyper-parameters, each one consisting of training the architecture from scratch, checking the results and modifying the network accordingly.

By contrast, neural architecture search aims to automate this process. It first encodes design choices and heuristics into a search space, such that many human-designed architectures constitute sub-models within the space. It then defines an architecture sampling policy that can be trained to select the best architecture in the search space based on certain metrics of interest, e.g. top-1 accuracy for image classification task. Below, we discuss each component of NAS in more detail.

### 3.1.1 Search Space

In general, the search space is a set of neural architectures, and plays a critical role in the success of neural architecture search. One typical example is the 'cell-based' graph structure search space shown in Figure 3.1. This is the most widely used approach in the current NAS field because of its flexibility, e.g., one can search for a novel architecture with arbitrary connections by treating one cell as an entire network.

Since the cell-based space will only yield a construction block but not a complete architecture, we summarize three common approaches to assemble cells into a network. These approaches are depicted in Figure 3.2. The first is the most restrictive one; we search for only one cell and stack replicates of this cell to form the final architecture. This is widely adopted in RNN search spaces (Pham et al., 2018a; Zoph & Le, 2017a). However, for CNNs that take an image as input,




(a) Illustration of the search space

(b) A ResNet cell extracted from the space

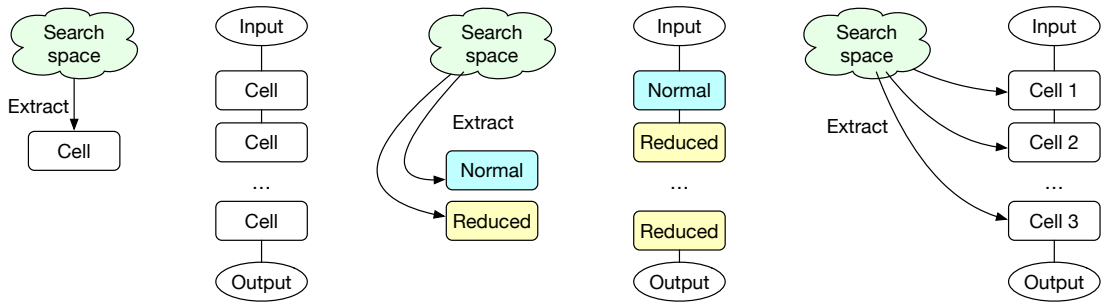
Figure 3.1: **Search space.** We present an example of cell-based graphical search space that is widely used in many previous works (Zoph & Le, 2017a; Pham et al., 2018b) when NAS is initially proposed. Here, we search for a 'cell' that composes of multiple basic operations, e.g. convolutional operations or pooling operations. We then formulate the network by repeating the searched cell. In **(a)** , each node represents the searchable operation, and each edge represents the data flow. **(b)** If we select the red edge in (a), we can remove the unused nodes and extract one cell architecture, which mimics the ResNet cell in He et al. (2015).

the feature dimensions (width and height) in the beginning is usually relatively large, and this first approach requires to trade off between the number of cells and the dimension reduction ratio; if the final searched cell contains reduction operations, such as pooling, we can only stack a few cells before the height and width drops to one, and stacking more will make no sense. On the other hand, if the cell does not reduce the feature dimension, the architecture may contain too many cells, which reduces its convergence ability. Zoph & Le (2017b) thus proposed an alternative approach, shown in Figure 3.2 (b), where the space contains two types of cells, the normal one that does not modify the image dimension, and the reduced one that always decreases the dimension by 2. This allows one to repeat as many normal cells as necessary, and use reduced cells according to the original image resolution. For example, we use four reduced cells for ImageNet-like input dimension. Recently, however, researchers have argued that restricting the normal cell might further limit the performance, and have proposed to search individual cells for the entire network (Cai et al., 2018b), as depicted in Figure 3.2 (c).

A variation of this last approach is now emerging as a highly successful one; it consists of simplifying the cell-space to the minimal level, where each cell only contains one node (Yu et al., 2020a; Cai et al., 2018a; 2020). This effectively limits the topology difference while maintaining a large search space by allowing for a large number of nodes. Furthermore, multiple recent works have modified this search space to support multi-input and multi-output, allowing one to perform NAS for more complicated tasks than image classification, such as semantic segmentation (Liu et al., 2019a), action recognition (Ryoo et al., 2020) and object detection (Chen et al., 2019b).

### 3.1.2 Search policy

Once the search space is designed, one should develop an algorithm to select effective architectures from it. This algorithm is usually referred to as a search policy. Since its introduction




(a) Stacking the same cell (b) Normal cell and reduced cell (c) All cells can be different

Figure 3.2: **Search space illustration.** Here, we show three examples to map a cell-based structure to a final neural architecture. **(a)** The simplest architecture is search for one cell architecture, and then repeatedly stacking the same cell to construct the architecture. This approach is commonly seen in RNN construction. **(b)** Zoph & Le (2017a) proposed a simple variation that searches for two different cells, normal cell that do not reduce the feature dimension, while the reduced cell will always reduce the size by half. **(c)** represents the most general approach that each cell can be different than others.

in (Zoph & Le, 2017a), NAS has demonstrated great potential to surpass the human design of deep networks for both visual recognition (Liu et al., 2018c; Ahmed & Torresani, 2018; Chen et al., 2018b; Pérez-Rúa et al., 2018; Liu et al., 2019a) and natural language processing (Zoph & Le, 2017a; Pham et al., 2018b; Luo et al., 2018a; Zoph et al., 2018; Liu et al., 2018c; Cai et al., 2018a). Existing search strategies include reinforcement learning (RL) samplers (Zoph & Le, 2017a; Zoph et al., 2018; Pham et al., 2018b), evolutionary algorithms (Xie & Yuille, 2017; Real et al., 2017a; Miikkulainen et al., 2019; Liu et al., 2018c; Lu et al., 2018), gradient-descent (Liu et al., 2019b), bayesian optimization (Kandasamy et al., 2018; Jin et al., 2019; Zhou et al., 2019) and performance predictors (Liu et al., 2018a; Luo et al., 2018a).

Here, we categorize the search policies into two groups: i) Full NAS policies that need to train a set of sampled architecture until convergence; ii) weight sharing NAS methods that share the parameters of all the architectures within the search space to accelerate training.

**Full NAS policy.** Considering the search space formulation, one important challenge is to translate the discrete architecture encoding so as to exploit it within the modern differentiable machine learning paradigm. One natural approach consists of relying on reinforcement learning (RL). As shown in Figure 3.3, Zoph & Le (2017a) first formulated the search space as a traditional action space in the RL domain, and encoded architecture sampling as one action within the space. This effectively translates the NAS problem to a reinforcement learning one, which can be solved using the policy gradient.

While other approaches, such as Bayesian optimization (Zhou et al., 2019), or evolutionary algorithm (So et al., 2019), have been proposed to replace the RL policy in Figure 3.3, they all rely on one common component: One needs to train the sampled architecture until convergence to

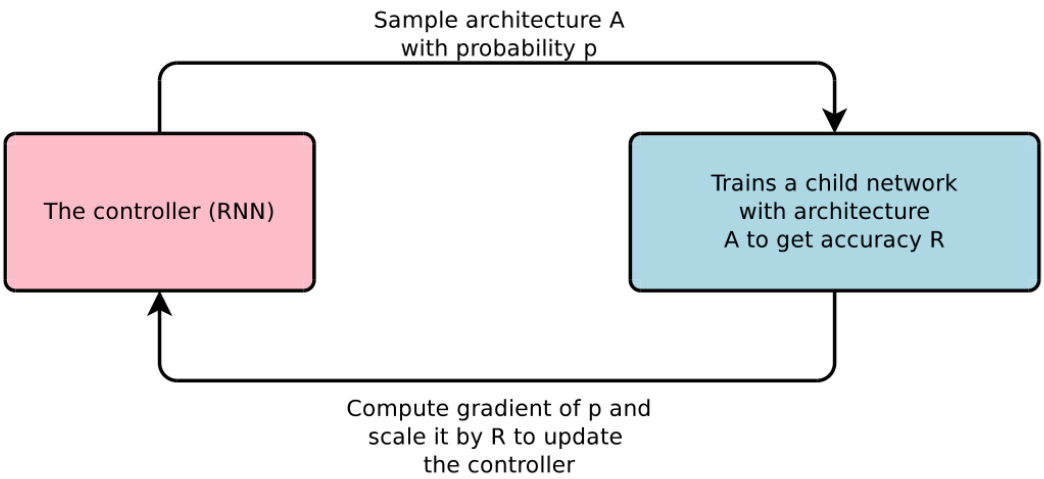



Figure 3.3: **Reinforce learning based NAS** (taken from Zoph & Le (2017a)). Child network is one stand-alone neural architecture within the search space.

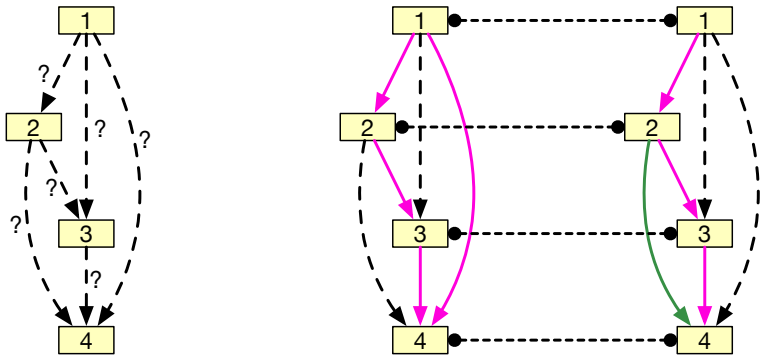
obtain the classification accuracy, or other metrics for different tasks, as depicted in Figure 3.5(a). This typically entails a tremendous computational burden of the NAS algorithm, e.g., taking over four thousand GPU days to search for a good model for the CIFAR-10 dataset. In the remainder of this thesis, we refer to this approach, that requires to train each sampled architecture individually, as the full NAS policy.

The potential of full NAS comes with the drawback of requiring thousands of GPU hours even for small datasets, such as PTB and CIFAR-10. Furthermore, even when using such heavy computational resources, vanilla NAS has to restrict the number of trained architectures from a total of $10^9$ to $10^4$, and increasing the sampler accuracy can only be achieved by increasing the resources.

**Weight sharing NAS.** To accelerate the NAS training process, Pham et al. (2018b) proposed to share the parameters of children networks within the search space, such that one can reuse the previously trained models, as depicted in Figure 3.5(b). Specifically, the weights are initialized randomly in the beginning, and are trained while the search progresses.

One illustrative example of this is shown in Figure 3.4. With the help of such a simple technique, the NAS training process decreases from thousands of GPU days to a single, while maintaining a similar final performance. As such, weight sharing has quickly become the de facto standard in NAS training. We discuss the related literature in detail in Section 3.2.1.

## 3.2 Related Work




(a) Search Space / Super-net (a) Sharing the parameters of the same node

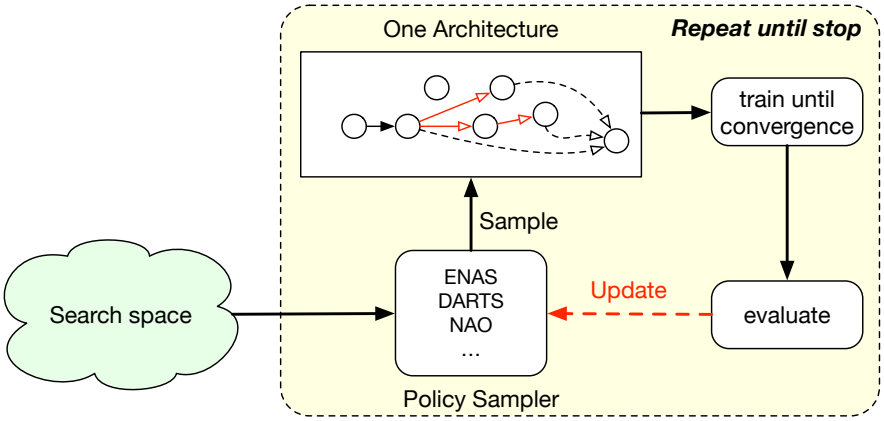
Figure 3.4: **Weight sharing example. (a)** The search space contains four nodes, where each node contains the parametric operations, and edges represents the data flow. The super-net can be easily constructed as is. **(b)** Two architectures can easily inherit the weights from the super-net, and thus shares the parameters during training and evaluation.

### 3.2.1 NAS with Parameter Sharing

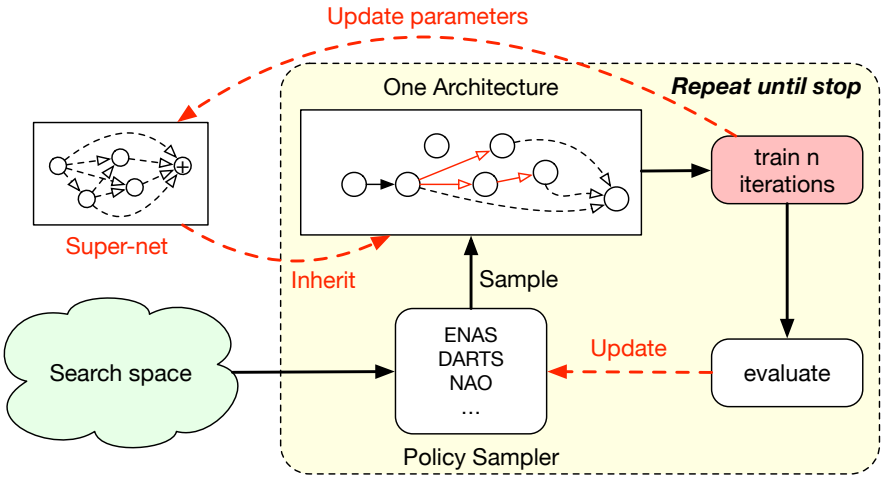
Initial search solutions required hundreds of GPUs due to the huge search space, but recent efforts have made the search more tractable, for example via the use of neural blocks (Negrinho & Gordon, 2017; Bennani-Smires et al., 2018). Similarly, and directly related to this work, *weight sharing* between the candidates has allowed researchers to greatly decrease the computational cost of neural architecture search. For neuro-evolution methods, sharing is implicit. For example, Real et al. (2017b) define weight inheritance as allowing the children to inherit their parents' weights whenever possible. For RL-base techniques, weight sharing is modeled explicitly and has been shown to lead to significant gains. In particular, ENAS (Pham et al., 2018b) was the first to propose a training scheme with shared parameters, reducing the resources from thousands of GPU days to one. Instead of being trained from scratch each sampled model inherits the parameters from previously-trained ones. Since then, NAS research has mainly focused on two directions: 1) Replacing the RL sampler with a better search algorithm, such as gradient descent (Liu et al., 2019b), bayesian optimiziation (Zhou et al., 2019) and performance predictors (Luo et al., 2018a); 2) Exploiting NAS for other applications, e.g., object detection (Ghiasi et al., 2019; Chen et al., 2019b), semantic segmentation (Liu et al., 2019a), and finding compact networks (Cai et al., 2018b; Wu et al., 2018; Chu et al., 2019; Guo et al., 2019). While weight sharing has proven effective, its downsides have never truly been studied. In Chapter 4, we will evidence one such drawback, which we dub multi-model forgetting, and introduce a solution to overcome it.

### 3.2.2 Understanding NAS algorithms

While successfully surpassing the state-of-the-art human designed architectures, NAS on the other hand is notoriously hard to reproduce (Yang et al., 2020). Understanding NAS algorithms has therefore recently emerged as another important research direction.




(a) Full NAS approach




(b) Weight Sharing NAS approach

Figure 3.5: **Comparison of full NAS approach and weight sharing one. (a)** we presents the full NAS approach, where each sampled architecture needs to be trained until convergence to obtain the metrics in order to update the policy. **(b)** depicts the weight sharing approach, where it contains a super-net that encompass all the parameters to initialize any child network within the search space. Each time we sample a new architecture, we inherit the parameters from the super-net and only trains for $n$ iterations. The parameters update is regarding the super-net after the training is done.

(Ying et al., 2019; Dong & Yang, 2020) introduced a dataset that contains the ground-truth performance of CNN cells, and (Wang et al., 2019) evaluated some traditional search algorithms on it. Similarly, Radosavovic et al. (2019) characterizes many CNN search spaces by computing the statistics of a set of sampled architectures, revealing that, for datasets such as CIFAR-10 or ImageNet, these statistics are similar. While these works support our claim that evaluation of NAS algorithms is crucial, they do not directly evaluate the state-of-the-arts NAS algorithms as

we will do in this thesis.

**Evaluation of NAS algorithms.** Typically, the quality of NAS algorithms is judged based on the results of the final architecture they produce on the downstream task. In other words, the search and robustness of these algorithms are generally not studied, with (Liu et al., 2019b; So et al., 2019) the only exception for robustness, where results obtained with different random seeds were reported.

In Chapter 5, we will aim to improve the understanding of the mechanisms behind the search phase of NAS algorithms. Specifically, we propose doing so by comparing them with a simple random search policy, which uniformly randomly samples one architecture per run in the same search space as the NAS techniques.

While some works have provided partial comparisons to random search, these comparisons unfortunately did not give a fair chance to the random policy. Specifically, (Pham et al., 2018b) reports the results of only a single random architecture, and (Liu et al., 2018c) those of an architecture selected among 8 randomly sampled ones as the most promising one after training for 300 epochs only. Here, we show that a fair comparison to the random policy, obtained by training all architectures, i.e., random and NAS ones, for 1000 epochs and averaging over multiple random seeds for robustness, yields a different picture; the state-of-the-art search policies are no better than the random one.

The motivation behind this comparison was our observation of only a weak correlation between the performance of the searched architectures and the ones trained from scratch during the evaluation phase. This phenomenon was already noticed by (Zela et al., 2018), and concurrently to our work by (Li & Talwalkar, 2019; Xie et al., 2019; Ying et al., 2019), but the analysis of its impact or its causes went no further. In this thesis, by contrast, we link this difference in performance between the search and evaluation phases to the use of weight sharing.

While this may seem to contradict the findings of (Bender et al., 2018b), which, on CIFAR-10, observed a strong correlation between architectures trained with and without weight sharing when searching a CNN cell, our analysis differs from (Bender et al., 2018b) in two fundamental ways: 1) The training scheme in (Bender et al., 2018b), in which the *entire model* with shared parameters is trained via random path dropping, is fundamentally different from those used by state-of-the-arts weight sharing NAS strategies (Pham et al., 2018b; Liu et al., 2019b; Luo et al., 2018a); 2) While the correlation in (Bender et al., 2018b) was approximated using a small subset of sampled architectures, we make use of a reduced search space where we can perform a complete evaluation of *all* architectures, thus providing an exact correlation measure in this space.

**Holistic review of super-net design.** In Chapter 6, we decouple the weight sharing NAS into four components, a mapping function $f_{proxy}$ construction of a stand-alone model from the search space and its training protocol $P_{proxy}$, and its counterpart for the weight sharing super-net $f_{ws}$

$P_{ws}$[1].

Many strategies have been proposed to implement the search algorithm, such as reinforcement learning (Zoph & Le, 2017a; Zoph et al., 2018), evolutionary algorithms (Real et al., 2017a; Miikkulainen et al., 2019; So et al., 2019; Liu et al., 2018a; Lu et al., 2018), gradient-based optimization (Liu et al., 2019b; Xu et al., 2020; Li et al., 2020a), Bayesian optimization (Kandasamy et al., 2018; Jin et al., 2019; Zhou et al., 2019; Wang et al., 2020), and separate performance predictors (Liu et al., 2018a; Luo et al., 2018a). Until very recently, the common trend to evaluate NAS consisted of reporting the searched architecture's performance on the proxy task (Xie et al., 2018; Real et al., 2018; Ryoo et al., 2020). This, however, hardly provides real insights about the NAS algorithms themselves, because of the many components involved in them. Many factors that differ from one algorithm to another can influence the performance. In practice, the literature even commonly compares NAS methods that employ different protocols to train the final model.

Concurrently, Li & Talwalkar (2019) and us were the first to systematically compare different algorithms with the same settings for the proxy task and using several random initializations. Their surprising results revealed that many NAS algorithms produce architectures that do not significantly outperform a randomly-sampled architecture. In Chapter 5, we will highlight the importance of the training protocol $P_{proxy}$. They showed that optimizing the training protocol can improve the final architecture performance on the proxy task by three percent on CIFAR-10. This non-trivial improvement can be achieved regardless of the chosen sampler, which provides clear evidence for the importance of unifying the protocol to build a solid foundation for comparing NAS algorithms.

While recent advances for systematic evaluation are promising, no work has yet thoroughly studied the influence of the super-net training protocol $P_{ws}$ and the mapping function $f_{ws}$. Previous works (Zela et al., 2020a) performed hyper-parameter tuning to evaluate their own algorithms, and focused only on a few parameters. In Chapter 6, we fill this gap by benchmarking different choices of $P_{ws}$ and $f_{ws}$ and by proposing novel variations to improve the super-net quality.

Recent works have shown that sub-nets of super-net training can surpass some human designed models without retraining (Yu et al., 2020a; Cai et al., 2020) and that reinforcement learning can surpass the performance of random search (Bender et al., 2020). However, these findings are still only shown on MobileNet-like search spaces where we only search for the size of convolution kernels and the channel ratio for each layer. This is an effective approach to discover a compact network, but it does not change the fact that on cell-based search space super-net quality remains low.

[1] Details in Section 6.2

### 3.2.3 Improving the ranking correlation of WS-NAS

Neural architecture search (NAS) methods can be categorized into conventional approaches, that obtain architecture performance via stand-alone training (Zoph & Le, 2017a; Zoph et al., 2018; Tan et al., 2018; Wang et al., 2019; Real et al., 2018; 2017a; Wang et al., 2020), and weight sharing NAS, where the performance is obtained from one or a few super-nets that encompass all architectures within the search space (Pham et al., 2018b; Luo et al., 2018a; Cai et al., 2018b; Zhao et al., 2020; Peng et al., 2020; You et al., 2020). Motivated by the success of early NAS works, the literature has now branched into several research directions, such as using multi-objective optimization to discover architectures under resource constraints for mobile devices (Tan & Le, 2019; Tan et al., 2018; Wu et al., 2018; Cai et al., 2018b; Guo et al., 2019; Bender et al., 2020), applying NAS to other computer vision tasks than image recognition (Liu et al., 2019a; Chen et al., 2019b; Li et al., 2020b; Ryoo et al., 2020), and using knowledge distillation to eliminate the performance gap between super-net and stand-alone training for linear search spaces based on MobileNet (Cai et al., 2020; Yu et al., 2020a).

In contrast to the diversity of these research directions, super-net training in weight-sharing NAS has remained virtually unchanged since its first appearance in (Li & Talwalkar, 2019; Guo et al., 2019; Bender et al., 2018b). At its core, it consists of sampling one or few architectures at each training step, and updating the parameters encompassed by these architectures with a small batch of data. This approach has been challenged in many ways (Li & Talwalkar, 2019; Yang et al., 2020), particular thanks to the introduction of the NASBench series (Radosavovic et al., 2019; Ying et al., 2019; Dong & Yang, 2019b; Siems et al., 2020; Zela et al., 2020b) of NAS benchmarks, which provide stand-alone performance of a substantial number or architectures and thus facilitate the analysis of the behavior of NAS algorithms. A critical issue that has been identified recently is the inability of most modern NAS algorithms to surpass simple random search under a fair comparison. In Chapter 5, this was traced back to to the low ranking correlation between stand-alone performance and the corresponding super-net estimates. While recent works (Guo et al., 2019; Chu et al., 2019) have shown that the ranking correlation is high on a MobileNet-based search space, where one only searches for the convolutional operations and the number of channels, others (Yu et al., 2020b; Zela et al., 2020b) have revealed that the correlation remains low on cell-based NASNet-like search spaces (Liu et al., 2019b; Luo et al., 2018a; Zoph & Le, 2017a), even when carefully tuning the design of the super-net.

In Chapter 7, we will introduce a simple, differentiable regularization term to improve the ranking correlation in weight-sharing NAS algorithms. We will show that this regularization term can be used in a variety of weight-sharing NAS algorithms, and that it leads to a consistent improvement in terms of ranking correlation and final search performance. Our regularization leverages the stand-alone performance of a few architectures. While some contemporary works also use ground-truth architecture performance, our approach differs fundamentally from theirs. Specifically, these methods aim to train a performance predictor, based on an auto-encoder in (Luo et al., 2020) or on a graphical neural network in (Tang et al., 2020), and are thus only applicable to weight-sharing NAS strategies that exploit such a performance predictor. By contrast, we add a regularizer to

the super-net training loss, which allows our method to be applied to most weight-sharing NAS search strategies. Furthermore, our approach requires an order of magnitude fewer architectures with associated stand-alone performance; in our experiments, we use 30 instead of 300 in (Luo et al., 2020) and 1000 in (Tang et al., 2020).

# 4 Overcoming Multi-model Forgetting

In this chapter, we identify a phenomenon that commonly appear with the weight sharing NAS, which we refer to as *multi-model forgetting*, that occurs when sequentially training multiple deep networks with partially-shared parameters; the performance of previously-trained models degrades as one optimizes a subsequent one, due to the overwriting of shared parameters. To overcome this, we introduce a statistically-justified weight plasticity loss that regularizes the learning of a model's shared parameters according to their importance for the previous models, and demonstrate its effectiveness when training two models sequentially and for neural architecture search. Adding weight plasticity in neural architecture search preserves the best models to the end of the search and yields improved results in both natural language processing and computer vision tasks.

## 4.1 Introduction

Deep neural networks have been very successful for tasks such as visual recognition (Xie & Yuille, 2017) and natural language processing (Young et al., 2017), and much recent work has addressed the training of models that can generalize across multiple tasks (Caruana, 1997). In this context, when the tasks become available sequentially, a major challenge is *catastrophic forgetting*: when a model initially trained on task A is later trained on task B, its performance on task A can decline calamitously. Several recent articles have addressed this problem (Kirkpatrick et al., 2017; Rusu et al., 2016; He & Jaeger, 2017; Li & Hoiem, 2016). In particular, Kirkpatrick et al. (2017) show how to overcome catastrophic forgetting by approximating the posterior probability, $p(\boldsymbol{\theta} \mid \mathscr{D}_1, \mathscr{D}_2)$, with $\boldsymbol{\theta}$ the network parameters and $\mathscr{D}_1, \mathscr{D}_2$ datasets representing the tasks.

In many situations one does not train *a single model for multiple tasks* but *multiple models for a single task*. This is the scenario we tackle in this paper. When dealing with many large models, a common strategy to keep training tractable is to share a subset of the weights across the multiple models and to train them sequentially (Pham et al., 2018a; Xie & Yuille, 2017; Liu et al., 2018b). This strategy has a major drawback. Figure 4.1 shows that for two models, A and B, the larger the

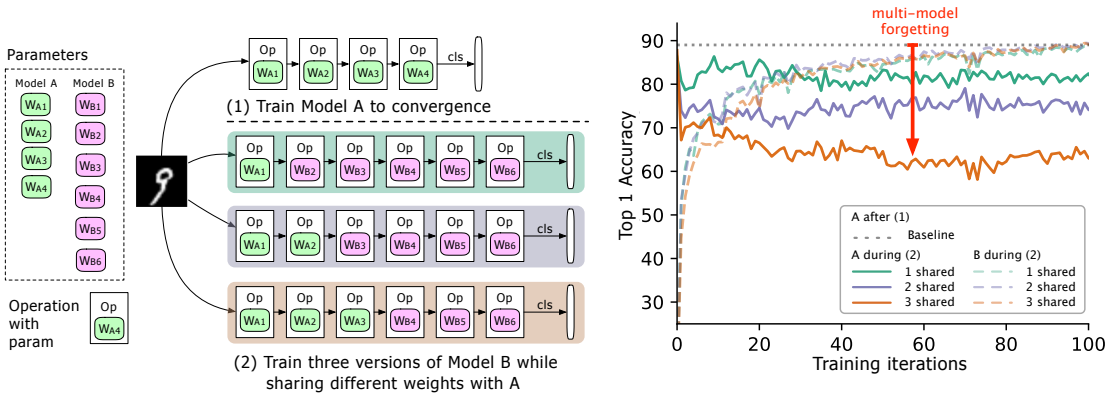



Figure 4.1: **Multimodel forgetting.** *(Left)* Two models to be trained (A, B), where A's parameters are in green and B's in purple, and B shares some parameters with A (indicated in green during phase 2). We first train A to convergence and then train B. *(Right)* Accuracy of model A as the training of B progresses. The different colors correspond to different numbers of shared layers. The accuracy of A decreases dramatically, especially when more layers are shared, and we refer to the drop (the red arrow) as multi-model forgetting. This experiment was performed on MNIST (LeCun & Cortes, 2010).

number of shared weights, the more the accuracy of A drops when training B; B overwrites some of the weights of A and this damages the performance of A. We call this *multi-model forgetting*. The benefits of weight-sharing have been emphasized in tasks like neural architecture search, where the associated speed gains have been key in making the process practical (Pham et al., 2018a; Liu et al., 2018d), but its downsides remain unexplored.

In this paper we introduce an approach to overcoming multi-model forgetting. Given a dataset $\mathscr{D}$, we first consider two models $f_1(\mathscr{D};\boldsymbol{\theta}_1,\boldsymbol{\theta}_s)$ and $f_2(\mathscr{D};\boldsymbol{\theta}_2,\boldsymbol{\theta}_s)$ with shared weights $\boldsymbol{\theta}_s$ and private weights $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$. We formulate learning as the maximization of the posterior $p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\boldsymbol{\theta}_s|\mathscr{D})$. Under mild assumptions we show that this posterior can be approximated and expressed using a loss, dubbed Weight Plasticity Loss (WPL), that minimizes multi-model forgetting. Our framework evaluates the importance of each weight, conditioned on the previously-trained model, and encourages the update of each shared weight to be inversely proportional to its importance. We then show that our approach extends to more than two models by exploiting it for neural architecture search.

Our work is the first to propose a solution to multi-model forgetting. We establish the merits of our approach when training two models with partially shared weights and in the context of neural architecture search. For the former, we establish the effectiveness of WPL in the strict convergence case, where each model is trained until convergence, and in the more realistic loose convergence setting, where training is stopped early. WPL can reduce the forgetting effect by 99% when model A converges fully, and by 52% in the loose convergence case.

For neural architecture search, we implement WPL within the efficient ENAS method of Pham

et al. (2018a), a state-of-the-art technique that relies on parameter sharing and corresponds to the loose convergence setting. We show that, at each iteration, the use of WPL reduces the forgetting effect by 51% on the most affected model and by 95% on average over all sampled models. Our final results on the best architecture found by the search confirm that limiting multi-model forgetting yields better results and better convergence for both language modeling (on the PTB dataset (Marcus et al., 1994b)) and image classification (on the CIFAR10 dataset (Krizhevsky et al., 2009a)). For language modeling the perplexity decreases from 65.01 for ENAS without WPL to 61.9 with WPL. For image classification WPL yields a drop of top-1 error from 4.87% to 3.81%. We also adapt our method to NAO (Luo et al., 2018c) and show that it also significantly reduces multi-model forgetting. Our code is public available at https://github.com/kcyu2014/multi-model-forgetting.

## 4.2 Related work

**Single-model Forgetting**. The goal of training a single model to tackle multiple problems is to leverage the structures learned for one task for other tasks. This has been employed in transfer learning (Pan & Yang, 2010), multi-task learning (Caruana, 1997) and lifelong learning (Silver et al., 2013). However, sequential learning of later tasks has visible negative consequences for the initial one. Kirkpatrick et al. (2017) selectively slow down the learning of the weights that are comparatively important for the first task by defining the importance of an individual weight using its Fisher information (Rissanen, 1996). He & Jaeger (2017) project the gradient so that directions relevant to the previous task are unaffected. Other families of methods save the older models separately to create progressive networks (Rusu et al., 2016) or use regularization to force the parameters to remain close to the values obtained by previous tasks while learning new ones (Li & Hoiem, 2016). In (Xu & Zhu, 2018), forgetting is avoided altogether by fixing the parameters of the first model while complementing the second one with additional operations found by an architecture search procedure. This work, however, does not address the multi-model forgetting that occurs during the architecture search. An extreme case of sequential learning is lifelong learning, for which the solution to catastrophic forgetting developed by Aljundi et al. (2018) is also to prioritize the weight updates, with smaller updates for weights that are important for previously-learned tasks. Teh et al. (2017) proposes a reinforcement learning approach for multi-task and multi-model scenario, but it relies on knowledge distillation which works under the assumption of two models. Applying it to train every two consecutive models, the knowledge of model not in the current pair will again be forgotten.

**Analysis of Parameter Sharing in Neural Architecture Search.** In both sequential learning on multiple tasks and lifelong learning, the forgetfulness concerns an individual model. Here we tackle scenarios where one seeks to optimize a population of multiple models that share parts of their internal structure. The use of multiple models to solve a single task dates back to model ensembles (Dietterich, 2000). Recently, the weight sharing in neural architecture search arises as a mainstream NAS method, as discussed in Section 3.2.1. Bender et al. (2018a) realized that

training was unstable and proposed to circumvent this issue by randomly dropping network paths. However, they did not analyze the reasons for the instability. Here, by contrast, we highlight the underlying multi-model forgetting problem and introduce a statistically-justified solution that further improves on path dropout.

## 4.3 Methodology

In this section we study the training of multiple models that share certain parameters. As discussed above, training the multiple models sequentially as in (Pham et al., 2018a), for example, is suboptimal, since multi-model forgetting arises. Below we derive a method to overcome this for two models, and then show how our formalism extends to multiple models in the context of neural architecture search, and in particular within ENAS (Pham et al., 2018a).

### 4.3.1 Weight Plasticity Loss: Preventing Multi-model Forgetting

Given a dataset $\mathscr{D}$, we seek to train two architectures $f_1(\mathscr{D};\boldsymbol{\theta}_1,\boldsymbol{\theta}_s)$ and $f_2(\mathscr{D};\boldsymbol{\theta}_2,\boldsymbol{\theta}_s)$ with shared parameters $\boldsymbol{\theta}_s$ and private parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$. We suppose that the models are trained sequentially, which reflects common large-model, large-dataset scenarios and will facilitate generalization. Below, we derive a statistically-motivated framework that prevents multi-model forgetting; it stops the training of the second model from degrading the performance of the first model.

We formulate training as finding the parameters $\boldsymbol{\theta} = (\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\boldsymbol{\theta}_s)$ that maximize the posterior probability $p(\boldsymbol{\theta} \mid \mathscr{D})$, which we approximate to derive our new loss function. Below we discuss the different steps of this approximation, first expressing $p(\boldsymbol{\theta} \mid \mathscr{D})$ more conveniently.

**Lemma 1.** *Given a dataset $\mathscr{D}$ and two architectures with shared parameters $\boldsymbol{\theta}_s$ and private parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, and if $p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2 \mid \boldsymbol{\theta}_s,\mathscr{D}) = p(\boldsymbol{\theta}_1 \mid \boldsymbol{\theta}_s,\mathscr{D})p(\boldsymbol{\theta}_2 \mid \boldsymbol{\theta}_s,\mathscr{D})$, we have*

$$p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\boldsymbol{\theta}_s \mid \mathscr{D}) \propto \frac{p(\mathscr{D} \mid \boldsymbol{\theta}_2,\boldsymbol{\theta}_s)p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_s \mid \mathscr{D})p(\boldsymbol{\theta}_2,\boldsymbol{\theta}_s)}{\int p(\mathscr{D} \mid \boldsymbol{\theta}_1,\boldsymbol{\theta}_s)p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_s)d\boldsymbol{\theta}_1}. \tag{4.1}$$

*Proof.* Provided in the Appendix A.1. □

Lemma 1 presupposes that $p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2 \mid \boldsymbol{\theta}_s,\mathscr{D}) = p(\boldsymbol{\theta}_1 \mid \boldsymbol{\theta}_s,\mathscr{D})p(\boldsymbol{\theta}_2 \mid \boldsymbol{\theta}_s,\mathscr{D})$, i.e., $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ are conditionally independent given $\boldsymbol{\theta}_s$ and the dataset $\mathscr{D}$. While this must be checked in applications, it is suitable for our setting, since we want both networks, $f_1(\mathscr{D};\boldsymbol{\theta}_1,\boldsymbol{\theta}_s)$ and $f_2(\mathscr{D};\boldsymbol{\theta}_2,\boldsymbol{\theta}_s)$, to train independently well.

To derive our loss we study the components on the right of equation (4.1). We start with the integral in the denominator, for which we seek a closed form. Suppose we have trained the first model and seek to update the parameters of the second one while avoiding forgetting. The following lemma provides an expression for the denominator of equation (4.1).

**Lemma 2.** *Suppose we have the maximum likelihood estimate* $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ *for the first model, write* $\mathrm{Card}(\boldsymbol{\theta}_1) + \mathrm{Card}(\boldsymbol{\theta}_s) = p_1 + p_s = p$*, and let the negative Hessian* $\boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ *of the log posterior probability distribution* $\log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D})$ *evaluated at* $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ *be partitioned into four blocks corresponding to* $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$ *as*

$$\boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s) = \left[\begin{array}{c|c} \boldsymbol{H}_{11} & \boldsymbol{H}_{1s} \\ \hline \boldsymbol{H}_{s1} & \boldsymbol{H}_{ss} \end{array}\right].$$

*If the parameters of each model follow Normal distributions, i.e.,* $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) \sim \mathscr{N}_p(\boldsymbol{0}, \sigma^2 \boldsymbol{I}_p)$*, with* $\boldsymbol{I}_p$ *the p-dimensional identity matrix, then the denominator of equation (4.1),* $A = \int p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_s, \boldsymbol{\theta}_1) d\boldsymbol{\theta}_1$ *can be written as*

$$A = \exp\{l_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s) - \frac{1}{2}\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v}\} \times (2\pi)^{p_1/2} |\det(\boldsymbol{H}_{11}^{-1})|^{1/2}, \tag{4.2}$$

*where* $\boldsymbol{v} = \boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s$*,* $l_p(\boldsymbol{\theta}) = l(\boldsymbol{\theta}) - \boldsymbol{\theta}^T \boldsymbol{\theta}/2\sigma^2$*, and* $\boldsymbol{\Omega} = \boldsymbol{H}_{ss} - \boldsymbol{H}_{1s}^\top \boldsymbol{H}_{11}^{-1} \boldsymbol{H}_{1s}$ *.*

*Proof.* Provided in the Appendix A.1. □

Lemma 2 requires the maximum likelihood estimate $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$, which can be hard to obtain with deep networks, since they have non-convex objective functions. In practice, one can train the network to convergence and treat the resulting parameters as maximum likelihood estimates. Our experiments show that the parameters obtained without optimizing to convergence can be used effectively. Moreover Haeffele & Vidal (2017) showed that networks relying on *positively homogeneous functions* have critical points that are either global minimizers or saddle points, and that training to convergence yields near-optimal solutions, which correspond to true maximum likelihood estimates.

Following Lemmas 1 and 2, as shown in the appendix,

$$\begin{aligned} \log p(\boldsymbol{\theta} \mid \mathscr{D}) \propto \log p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) + \log p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s) \\ + \log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D}) + \frac{1}{2}\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v}, \end{aligned} \tag{4.3}$$

apart from an additive constant. To derive a loss function that prevents multi-model forgetting, consider equation (4.3). The first term on its right-hand side corresponds to the log likelihood of the second model and can be replaced by the cross-entropy $\mathscr{L}_2(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s)$, and if we use a Gaussian prior on the parameters, the second term encodes an $L^2$ regularization. Since equation (4.3) depends only on the log likelihood of the second model $f_2(\mathscr{D}; \boldsymbol{\theta}_2, \boldsymbol{\theta}_s)$, the information learned from the first model $f_1(\mathscr{D}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$ must reside in the conditional posterior probability $\log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D})$, and the final term, $\frac{1}{2}\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v}$, must represent the interactions between the models $f_1(\mathscr{D}; \boldsymbol{\theta}_2, \boldsymbol{\theta}_s)$ and $f_2(\mathscr{D}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$. This term will not appear in a standard single-model forgetting scenario. Let us examine these terms more closely.

The posterior probability $p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D})$ is intractable, so we apply a Laplace approximation (MacKay, 1992); we approximate the log posterior using a second-order Taylor expansion around

the maximum likelihood estimate $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$. This yields

$$\begin{aligned}\log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D}) = &\log p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s \mid \mathscr{D}) \\ &- \frac{1}{2}(\boldsymbol{\theta}_1', \boldsymbol{\theta}_s')^\top \boldsymbol{H}_p(\boldsymbol{\theta}_1', \boldsymbol{\theta}_s'),\end{aligned} \tag{4.4}$$

where $(\boldsymbol{\theta}_1', \boldsymbol{\theta}_s') = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) - (\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$, and $\boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ is the negative Hessian of the log posterior evaluated at the maximum likelihood estimate (MLE). As the first derivative is evaluated at the MLE, it equals zero.

Equation (4.4) yields a Gaussian approximation to the posterior with mean $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ and covariance matrix $\boldsymbol{H}_p^{-1}$, i.e.,

$$p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D}) \propto \exp\big\{-\frac{1}{2}(\boldsymbol{\theta}_1', \boldsymbol{\theta}_s')^\top \boldsymbol{H}_p(\boldsymbol{\theta}_1', \boldsymbol{\theta}_s')\big\}. \tag{4.5}$$

Our parameter space is too large to compute the inverse of the negative Hessian $\boldsymbol{H}_p$, so we replace it with the diagonal of the Fisher information, $\mathrm{diag}(\boldsymbol{F})$. This approximation falsely presupposes that the parameters $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$ are independent, but it has already proven effective (Kirkpatrick et al., 2017; Pascanu & Bengio, 2014). One of its main advantages is that we can compute the Fisher information from the squared gradients, thereby avoiding any need for second derivatives.

Using equation (4.5) and the Fisher approximation we can express the log posterior as

$$\log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D}) \propto \frac{\alpha}{2} \sum_{\theta_{s_i} \in \boldsymbol{\theta}_s} F_{\theta_{s_i}} (\theta_{s_i} - \hat{\theta}_{s_i})^2 \,, \tag{4.6}$$

where $F_{\theta_{s_i}}$ is the diagonal element corresponding to parameter $\theta_{s_i}$ in the diagonal approximation of the Fisher information matrix and $\alpha$ is a hyper-parameter, which can be obtained from the trained model $f_1(\mathscr{D}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$.

Now consider the last term in equation (4.3), noting that $\boldsymbol{\Omega} = \boldsymbol{H}_{ss} - \boldsymbol{H}_{1s}^\top \boldsymbol{H}_{11}^{-1} \boldsymbol{H}_{1s}$, as defined in Lemma 2. As our previous approximation relies on the assumption of a diagonal Fisher information matrix, we have $\boldsymbol{H}_{1s} = \boldsymbol{0}$, leading to $\boldsymbol{\Omega} = \boldsymbol{H}_{ss}$, so

$$\frac{1}{2}\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v} = \frac{1}{2} \sum_{\theta_{s_i} \in \boldsymbol{\theta}_s} F_{\theta_{s_i}} (\theta_{s_i} - \hat{\theta}_{s_i})^2 \,. \tag{4.7}$$

The last two terms on the right-hand side of equation (4.3), as expressed in equation (4.6) and equation (4.7), can then be grouped. Combining the result with the first two terms, discussed below equation (4.3), yields our Weight Plasticity Loss,

$$\begin{aligned}\mathscr{L}_{\mathrm{WPL}}(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s) = &\mathscr{L}_2(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s) + \frac{\lambda}{2}(\|\boldsymbol{\theta}_s\|^2 + \|\boldsymbol{\theta}_2\|^2) \\ &+ \frac{\alpha}{2} \sum_{\theta_{s_i} \in \boldsymbol{\theta}_s} F_{\theta_{s_i}} (\theta_{s_i} - \hat{\theta}_{s_i})^2,\end{aligned} \tag{4.8}$$

where $F_{\theta_{s_i}}$ is the diagonal element corresponding to parameter $\theta_{s_i}$ in the Fisher information

matrix obtained from the trained first model $f_1(\mathscr{D};\boldsymbol{\theta}_1,\boldsymbol{\theta}_s)$. We omit the terms depending on $\boldsymbol{\theta}_1$ in equation (4.6) because we are optimizing with respect to $(\boldsymbol{\theta}_2,\boldsymbol{\theta}_s)$ at this stage. The Fisher information in the last term encodes the importance of each shared weight for the first model's performance, so WPL encourages preserving shared parameters that were important for the first model, while allowing others to undergo larger changes and thus to improve the accuracy of the second model.

**Relation to Elastic weight consolidation.** The final loss function obtained in equation (4.8) may appear similar to that obtained by Kirkpatrick et al. (2017) when formulating their Elastic Weight Consolidation (EWC) to address catastrophic forgetting. However, the problem we address here is fundamentally different. Kirkpatrick et al. (2017) tackle sequential learning on different tasks, where a *single model* is sequentially trained using *two datasets*, and their goal is to maximize the posterior $p(\boldsymbol{\theta} \mid \mathscr{D}) = p(\boldsymbol{\theta} \mid \mathscr{D}_1,\mathscr{D}_2)$. By relying on Laplace approximations in neural networks (MacKay, 1992) and the connection between the Fisher information matrix and second-order derivatives (Pascanu & Bengio, 2014), EWC is then formulated as the loss $\mathscr{L}(\boldsymbol{\theta}) = \mathscr{L}_{\mathrm{B}}(\boldsymbol{\theta}) + \sum_i \frac{\lambda}{2} F_i(\theta_i - \theta^{\star}_{\mathrm{A},i})^2$, where A and B refer to *two different tasks*, $\boldsymbol{\theta}$ encodes the network parameters and $F_i$ is the Fisher information of $\theta_i$.

Here we consider scenarios with a *single dataset* but *two models* with shared parameters as shown in Figure 4.2, and aim to maximize the posterior $p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2,\boldsymbol{\theta}_s \mid \mathscr{D})$. The resulting WPL combines the original loss of the second model, a Fisher-weighted MSE term on the shared parameters and an $L^2$ regularizer on the parameters of the second model. More importantly, the last term in equation (4.3), $\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v}$, is specific to the multi-model case, since it encodes the interaction between the two models; it never appears in the EWC derivation. Because we adopt a Laplace approximation based on the diagonal Fisher information matrix, as shown in equation (4.7), this term can be grouped with that of equation (4.6). In principle, however, other approximations of $\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v}$ could be used, such as a Laplace one with a full covariance matrix, which would yield a final loss that differs fundamentally from the EWC one.

### 4.3.2 WPL for Neural Architecture Search

In the previous section, we considered only two models being trained sequentially, but in practice one often seeks to train three or more models. Our approach is then unchanged, but each model shares parameters with several other models, which entails using diagonal approximations to Fisher information matrices for all previously-trained models from equation (4.3). In the remainder of this section, we discuss how our approach can be used for neural architecture search.

Consider using our WPL within the ENAS strategy of Pham et al. (2018a). ENAS is a reinforcement-learning-based method that consists of two training processes: 1) sequentially train sampled models with shared parameters; and 2) train a controller RNN that generates model candidates. Incorporating our WPL within ENAS only affects 1).

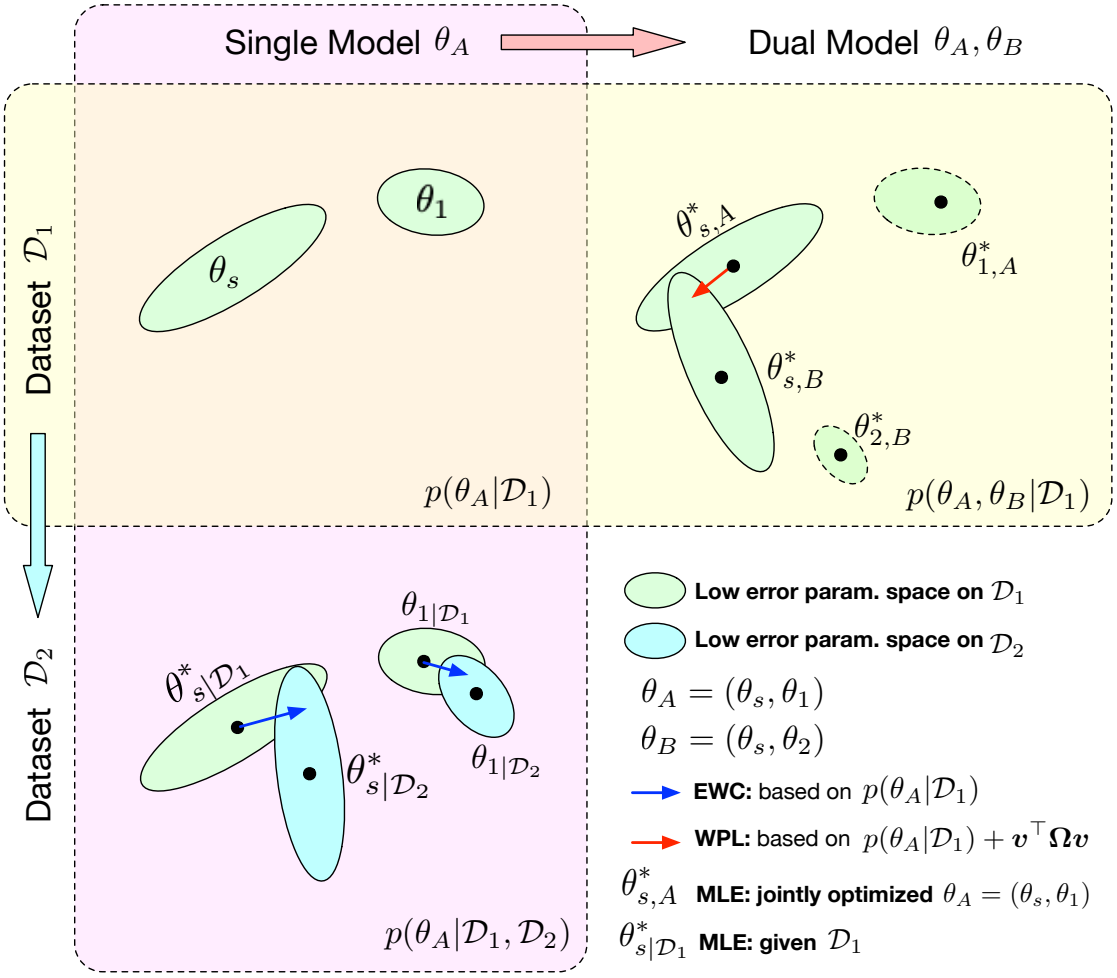



Figure 4.2: **Comparison between EWC and WPL.** The ellipses in each subplot represent parameter regions corresponding to low error. (**Top left**) Both methods start with a single model, with parameters $\theta_A = \{\theta_s, \theta_1\}$, trained on a single dataset $\mathscr{D}_1$. (**Bottom left**) EWC regularizes *all* parameters based on $p(\theta_A|\mathscr{D}_1)$ to train the same initial model on a new dataset $\mathscr{D}_2$. (**Top right**) By contrast, WPL makes use of the initial dataset $\mathscr{D}_1$ and regularizes only the shared parameters $\theta_s$ based on both $p(\theta_A|\mathscr{D}_1)$ and $\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v}$, while the parameters $\theta_2$ can vary freely.

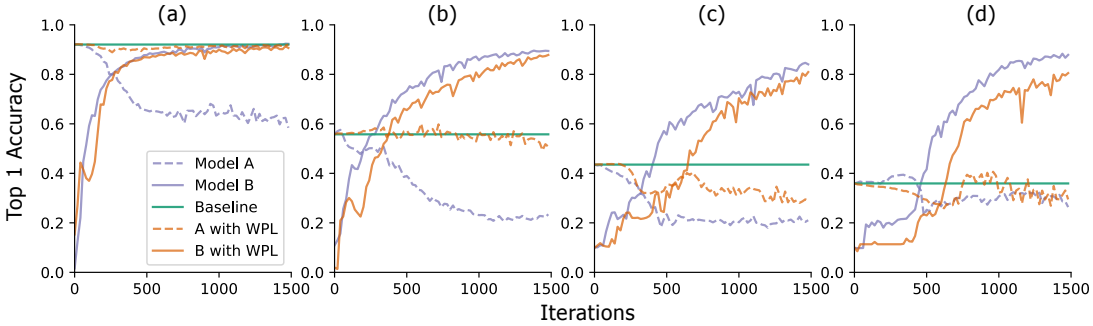



Figure 4.3: **From strict to loose convergence.** We conduct experiments on MNIST with models A and B with shared parameters, and report the accuracy of Model A before training Model B (baseline, green) and the accuracy of Models A and B while training Model B with (orange) or without (blue) WPL. In *(a)* we show the results for strict convergence: A is initially trained to convergence. We then relax this assumption and train A to around 55% *(b)*, 43% *(c)*, and 38% *(d)* of its optimal accuracy. We see that WPL is highly effective when A is trained to at least 40% of optimality; below, the Fisher information becomes too inaccurate to provide reliable importance weights. Thus WPL helps to reduce multi-model forgetting, even when the weights are not optimal. WPL reduced forgetting by up to 99.99% for *(a)* and *(b)*, and by up to 2% for *(c)*.

The first step of ENAS consists of sampling a fixed number of architectures from the RNN controller, and training each architecture on $B$ batches. This implies that our requirement for access to the maximum likelihood estimate of the previously-trained models is not satisfied, but we verify that in practice our WPL remains effective in this scenario. After sufficiently many epochs it is likely that all the parameters of a newly-sampled architecture are shared with previously-trained ones, and then we can consider that all parameters of new models are shared.

At the beginning of the search, the parameters of all models are randomly initialized. Adopting WPL directly from the start would therefore make it hard for the process to learn anything, as it would encourage some parameters to remain random. To better satisfy our assumption that the parameters of previously-trained models should be optimal, we follow the original ENAS training strategy for $n$ epochs, with $n = 5$ for RNN search and $n = 3$ for CNN search in our experiments. We then incorporate our WPL and store the optimal parameters after each architecture is trained. We also update the Fisher information, which adds virtually no computational overhead, because $F_{\theta i} = (\partial \mathscr{L} / \partial \theta_i)^2$, where $\mathscr{L} = \sum_i \mathscr{L}_i$, with $i$ indexing the previously-sampled architectures, and the derivatives are already computed for back-propagation. To ensure that these updates use the contributions from all previously-sampled architectures, we use a momentum-based update expressed as $F_{\theta}{}_i^t = (1-\eta) F_{\theta}{}_i^{t-1} + \eta (\partial \mathscr{L} / \partial \theta_i)^2$, with $\eta = 0.9$. Since this is not computed at the MLE of the parameters, we flush the global Fisher buffer to zero every three epochs, yielding an increasingly accurate estimate of the Fisher information as optimization proceeds. We also use a scheduled decay for $\alpha$ in equation (4.8).

## 4.4 Experiments

We first evaluate our weight plasticity loss (WPL) in the general scenario of training two models sequentially, both in the strict convergence case and when the weights of the first model are sub-optimal. We then evaluate the performance of our approach within the ENAS framework.

### 4.4.1 General Scenario: Training Two Models

To test WPL in the general scenario, we used the MNIST handwritten digit recognition dataset (LeCun & Cortes, 2010). We designed two feed-forward networks with 4 (Model A) and 6 (Model B) layers, respectively. All the layers of A are shared by B.

Let us first evaluate our approach in the strict convergence case. To this end, we trained A until convergence, thus obtaining a solution close to the MLE $\hat{\boldsymbol{\theta}}_{\mathrm{A}} = (\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$, since all our operations are positively homogeneous (Haeffele & Vidal, 2017). To compute the Fisher information, we used the backward gradients of $\boldsymbol{\theta}_s$ calculated on 200 images in the validation set. We then initialized $\boldsymbol{\theta}_s$ of Model B, $f_{\mathrm{B}}(\mathscr{D}; (\boldsymbol{\theta}_2, \boldsymbol{\theta}_s))$, as $\hat{\boldsymbol{\theta}}_s$ and trained B by standard SGD with respect to all its parameters. Figure 4.3*(a)* compares the performance of training Model B with and without WPL. Without WPL the performance of A degrades as training B progresses, but using WPL allows us to maintain the initial performance of A, indicated as Baseline in the plot. This entails no loss of performance for B, whose final accuracy is virtually the same both with and without WPL.

The assumption of optimal weights is usually hard to enforce. We therefore now turn to the more realistic loose convergence scenario. To evaluate the influence of sub-optimal weights for Model A on our approach, we trained Model A to different, increasingly lower, top 1 accuracies. As shown in Figure 4.3*(b)* and *(c)*, even in this setting our approach still significantly reduces multi-model forgetting. We can quantify the relative reduction rate of such forgetting as $d_{\mathrm{A}} - d_{\mathrm{A+WPL}}/d_{\mathrm{A}}$, where $d = acc^*_{\mathrm{A}} - acc$ is A's accuracy decay after training B. WPL can reduce multi-model forgetting by up to 99% for a converged model, and by 52% even for the loose convergence case. This suggests that the Fisher information remains a reasonable empirical approximation to the weights' importance even when our optimality assumption is not satisfied.

### 4.4.2 WPL for Neural Architecture Search

We demonstrate the effectiveness of WPL in a real-world application, neural architecture search. We incorporate WPL in the ENAS framework (Pham et al., 2018a), which relies on weight-sharing across model candidates to speed up the search and thus, while effective, will suffer from multi-model forgetting even with random dropping of weights and output dropout. To show this, we examine how the previously-trained architectures are affected by the training of new ones by evaluating the prediction error of each sampled architecture on a fraction of the validation dataset immediately after it is trained, denoted by $err_1$, and at the end of the epoch, denoted by $err_2$.

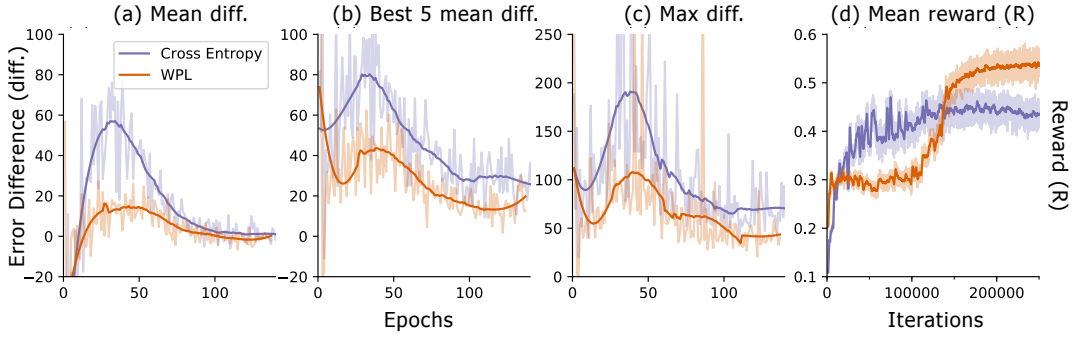



Figure 4.4: **Error difference during neural architecture search.** For each architecture, we compute the RNN error differences $err_2 - err_1$, where $err_1$ is the error right after training this architecture and $err_2$ the error after all architectures are trained in the current epoch. We plot (a) the mean difference over all sampled models, (b) the mean difference over the 5 models with lowest $err_1$, and (c) the max difference over all models. The plots show that WPL reduces multi-model forgetting; the error differences are much closer to 0. Quantitatively, the forgetting reduction can be up to 95% for *(a)*, 59% for *(b)* and 51% for *(c)*. In (d), we plot the average reward of the sampled architectures as a function of training iterations. Although WPL initially leads to lower rewards, due to a large weight $\alpha$ in equation (4.8), by reducing the forgetting it later allows the controller to sample better architectures, as indicated by the higher reward in the second half.

A positive difference $err_2 - err_1$ for a specific architecture indicates that it has been forced to forget by others.

We performed two experiments: RNN cell search on the PTB dataset and CNN micro-cell search on the CIFAR10 dataset. We report the mean error difference for all sampled architectures, the mean error difference for the 5 architectures with the lowest $err_1$, and the maximum error difference over all sampled architectures. Figure 4.4*(a)*, *(b)* and *(c)* plot these as functions of the training epochs for the RNN case, and similar plots for CNN search are in the appendix. The plots show that without WPL the error differences are much larger than 0, clearly displaying the multi-model forgetting effect. This is particularly pronounced in the first half of training, which can have a dramatic effect on the final results, as it corresponds to the phase where the algorithm searches for promising architectures. WPL significantly reduces the forgetting, as shown by much lower error differences. With WPL, these differences tend to decrease over time, emphasizing that the observed Fisher information encodes an increasingly reliable notion of weight importance as training progresses. Owing to limited computational resources we estimate the Fisher information using only small validation batches, but use of larger batches could further improve our results.

In Figure 4.4*(d)*, we plot the average reward of all sampled architectures as a function of the training iterations. In the first half of training, the models trained with WPL tend to have lower rewards. This can be explained by the use of a large value for $\alpha$ in equation (4.8) during this phase; while such a large value may prevent the best models from achieving as high a reward

Table 4.1: **Results of the best models found.** We take the best model obtained during the search and train it from scratch. ENAS* corresponds to the results of Pham et al. (2018a) obtained after extensive hyper-parameter search, while ENAS and ENAS+WPL were trained in comparable conditions. For both RNN and CNN search, our WPL gives a significant boost to ENAS, thus showing the importance of overcoming multi-model forgetting. In the RNN case, our approach outperforms ENAS* without requiring extensive hyper-parameter tuning.

| Datasets | Metric | ENAS* | ENAS | **ENAS + WPL** |
|---|---|---|---|---|
| PTB | perplexity | 63.26 | 65.01 | **61.9** |
| CIFAR10 | top-1 error | **3.54** | 4.87 | 3.81 |

as possible, it has the advantage of preventing the forgetting of good models, and thus avoiding their being discarded early. This is shown by the fact that, in the second half of training, when we reduce $\alpha$, the mean reward of the architectures trained with WPL is higher than without using it. In other words, our approach allows us to maintain better models until the end of training.

When the search is over, we train the best architecture from scratch and evaluate its final accuracy. Table 4.1 compares the results obtained without (ENAS) and with WPL (ENAS+WPL) with those from the original ENAS paper (ENAS*), which were obtained after conducting an extensive hyper-parameter search. For both datasets, using WPL improves final model accuracy, thus showing the importance of overcoming multi-model forgetting. In the case of PTB, our approach even outperforms ENAS*, without extensive hyper-parameter tuning. Based on the gap between ENAS and ENAS*, we anticipate that such a tuning procedure could further boost our results. In any event, we believe that these results already clearly show the benefits of reducing multi-model forgetting.

### 4.4.3 Neural Architecture Optimization

Our approach is general, and its use in the context of neural architecture search is not limited to ENAS. To demonstrate this, we applied it to the neural architecture optimization (NAO) method of Luo et al. (2018c), which also exploits weight-sharing in the search phase. In this context, we therefore investigate (i) whether multi-model forgetting occurs, and if so, (ii) the effectiveness of our approach in the NAO framework. Due to resource and time constraints, we focus our experiments mainly on the search phase, as training the best model that was found from scratch takes around 4 GPU days. To evaluate the influence of the dropout strategy of Bender et al. (2018a), we test NAO with or without random path-dropping and with four output dropout rates from 0 to 0.75 by steps of 0.25. As in Section 4.4.2, in Figure 4.5, we plot the mean validation perplexity and the best five model's error differences for all models that are sampled during a single training epoch. For random path-dropping, since Luo et al. (2018c) exploit a more aggressive dropping policy than that used in Bender et al. (2018a), validation perplexity quickly plateaus. Hence we do not add WPL to the path dropout strategy, but use it in conjunction with output dropout.

At all four different dropout rates, WPL clearly reduces multi-model forgetting and accelerates

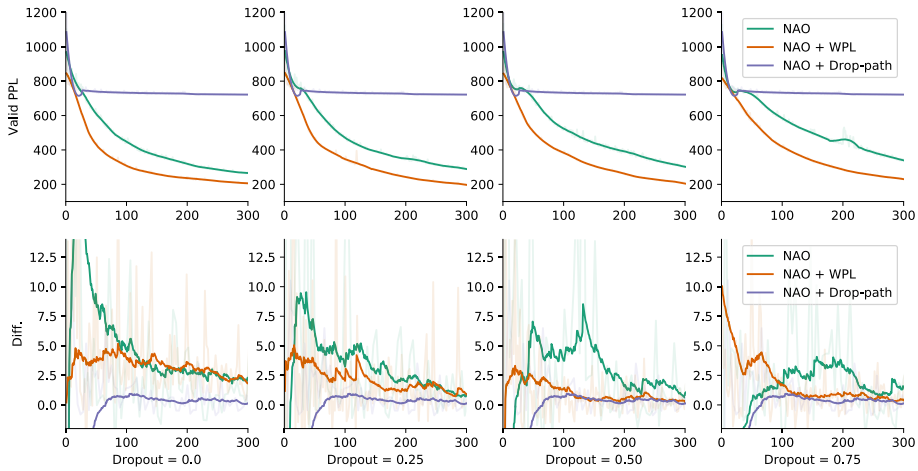



Figure 4.5: **Comparison of different output dropout rates for NAO.** We plot the mean validation perplexity while searching for the best architecture (top) and the best 5 model's error differences (bottom) for four different dropout rates. Note that path dropping in NAO prevents learning shortly after model initialization with all different dropout rates. At all the dropout rates, our WPL achieves lower error differences, i.e., it reduces multi-model forgetting, as well as speeds up training.

training. The level of forgetting decreases with the dropout rate, but our loss always further reduces it. Among the three methods, NAO with path dropping suffers the least from forgetting, but only because it does not learn properly. By contrast, WPL reduces multi-model forgetting while still allowing the models to learn. This shows that our approach generalizes beyond ENAS for neural architecture search.

## 4.5 Conclusion

We identified the problem of multi-model forgetting in the context of sequentially training multiple models: the shared weights of previously-trained models are overwritten during training of subsequent models, leading to performance degradation. We show that the degree of degradation is linked to the proportion of shared weights, and introduce a statistically-motivated weight plasticity loss (WPL) to overcome this. Our experiments on multi-model training and on neural architecture search clearly show the effectiveness of WPL in reducing multi-model forgetting and yielding better architectures, leading to improved results in both natural language processing and computer vision tasks. We believe that the impact of WPL goes beyond the tasks studied in this paper. In terms of the future work, one direction is to integrate WPL within other neural architecture search strategies in which weight sharing occurs and to study its use in other multi-model contexts, such as for ensemble learning.

# 5 Evaluating the search phase of neural architecture search

While experimenting with NAS, we have discovered that the performance of NAS samplers is often unstable; in some cases, simply changing the random seed will cause a drastic performance difference. This motivated us to revisit the current NAS evaluation scheme. Existing NAS approaches rely on two stages: searching over the architecture space and validating the best architecture. NAS algorithms are currently compared solely based on their results on the downstream task. While intuitive, this fails to explicitly evaluate the effectiveness of their search strategies. In this chapter, we propose to evaluate the NAS search phase. To this end, we compare the quality of the solutions obtained by NAS search policies with that of random architecture selection. We find that: (i) On average, the state-of-the-art NAS algorithms perform similarly to the random policy; (ii) the widely-used weight sharing strategy degrades the ranking of the NAS candidates to the point of not reflecting their true performance, thus reducing the effectiveness of the search process. We believe that our evaluation framework will be key to designing NAS strategies that consistently discover architectures superior to random ones.

## 5.1 Introduction

By automating the design of a neural network for the task at hand, Neural Architecture Search (NAS) has tremendous potential to impact the practicality of deep learning (Zoph & Le, 2017a; Liu et al., 2018c;a; Tan et al., 2018; Baker et al., 2016), and has already obtained state-of-the-art performance on many tasks. A typical NAS technique (Zoph & Le, 2017a; Pham et al., 2018b; Liu et al., 2018a) has two stages: the search phase, which aims to find a good architecture, and the evaluation one, where the best architecture is trained from scratch and validated on the test data.

In the literature, NAS algorithms are typically compared based on their results in the evaluation phase. While this may seem intuitive, the search phase of these algorithms often differ in several ways, such as their architecture sampling strategy and the search space they use, and the impact of these individual factors cannot be identified by looking at the downstream task results only.

Table 5.1: **Comparison of NAS algorithms with random sampling.** We report results on PTB using mean validation perplexity (the lower, the better) and on CIFAR-10 using mean top 1 accuracy. We also provide the *p-value* of Student's t-tests against random sampling.

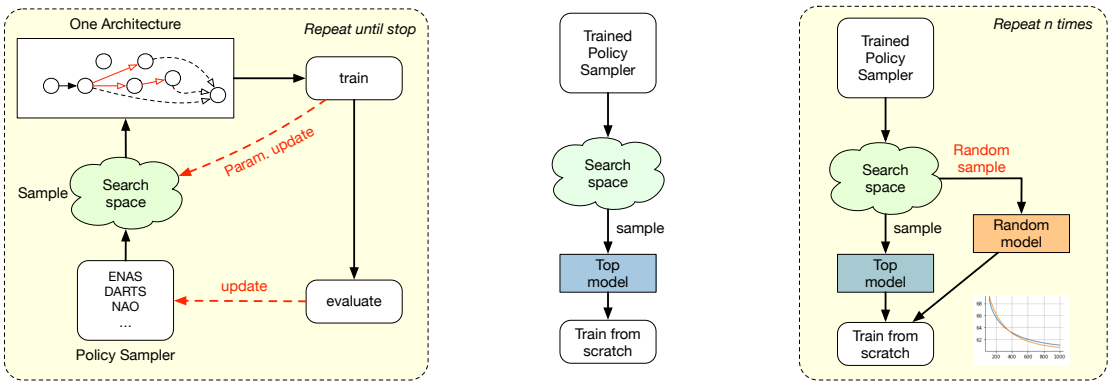
| | PTB (PPL) | t-test | CIFAR-10 (acc.) | t-test |
|---|---|---|---|---|
| ENAS | 59.88 ± 1.92 | 0.73 | 96.79 ± 0.11 | 0.01 |
| DARTS | 60.61 ± 2.54 | 0.62 | 96.62 ± 0.23 | 0.20 |
| NAO | 61.99 ± 1.95 | 0.02 | 96.86 ± 0.17 | 0.00 |
| Random | 60.13 ± 0.65 | - | 96.44 ± 0.19 | - |

Furthermore, the downstream task results are often reported for a single random seed, which leaves unanswered the question of robustness of the search strategies.

In this chapter, we therefore propose to investigate the search phase of existing NAS algorithms in a controlled manner. To this end, we compare the quality of the NAS solutions with a random search policy, which uniformly randomly samples an architecture from the same search space as the NAS algorithms, and then trains it using the same hyper-parameters as the NAS solutions. To reduce randomness, the search using each policy, i.e., random and NAS ones, is repeated several times, with different random seeds.

We perform a series of experiments on the Penn Tree Bank (PTB) (Marcus et al., 1994a) and CIFAR-10 (Krizhevsky et al., 2009a) datasets, in which we compared the state-of-the-art NAS algorithms whose code is publicly available—DARTS (Liu et al., 2019b), NAO (Luo et al., 2018a) and ENAS (Pham et al., 2018b)—to our random policy. We reached the surprising conclusions that, as shown in Table 5.1, none of them significantly outperforms random sampling. Since the mean performance for randomly-sampled architectures converges to the mean performance over the entire search space, we further conducted *Welch Student's t-tests (Welch, 1947)*, which reveal that, in RNN space, ENAS and DARTS cannot be differentiated from the mean of entire search space, while NAO yields worse performance than random sampling. While the situation is slightly better in CNN space, all three algorithms still perform similarly to random sampling. Note that this does not necessarily mean that these algorithms perform poorly, but rather that the search space has been sufficiently constrained so that even a random architecture in this space provides good results. To verify this, we experiment with search spaces where we can exhaustively evaluate all architectures, and observe that these algorithms truly cannot discover top-performing architectures.

In addition to this, we observed that the ranking by quality of candidate architectures produced by the NAS algorithms during the search does not reflect the true performance of these architectures in the evaluation phase. Investigating this further allowed us to identify that weight sharing (Pham et al., 2018b), widely adopted to reduce the amount of required resources from thousands of GPU days to a single one, harms the individual networks' performance. More precisely, using




(a) Search Phase (b) Standard Evaluation Phase (c) Our Evaluation Phase

Figure 5.1: **Evaluating NAS.** Existing frameworks consist of two phases: **(a)** The search phase, where a sampler is trained to convergence or a pre-defined stopping criterion; **(b)** The evaluation phase that trains the best model from scratch and evaluates it on the test data. Here, we argue that one should evaluate the search itself. To this end, as shown in **(c)**, we compare the best architecture found by the NAS policy with *a single* uniformly randomly sampled architecture. For this comparison to be meaningful, we repeat it with different random seeds for both training the NAS sampler and our random search policy. We then report the mean and standard deviations over the different seeds.

reduced search spaces, we make use of the Kendall Tau $\tau$ metric[1] to show that the architecture rankings obtained with and without weight sharing are entirely uncorrelated in RNN space ($\tau$ = -0.004 over 10 runs); and have little correlation in the CNN space ($\tau$ = 0.195 over 10 runs). Since such a ranking is usually treated as training data for the NAS sampler in the search phase, this further explains the small margin between random search and the NAS algorithms. We also show that training samplers without weight sharing in CNN space surpasses random sampling by a significant margin.

In other words, we disprove the common belief that the quality of architectures trained with and without weight sharing is similar. We show that the difference in ranking negatively impacts the search phase of NAS algorithms, thus seriously impeding their robustness and performance.

In short, evaluating the search phase of NAS, which is typically ignored, allowed us to identify two key characteristics of state-of-the-art NAS algorithms: The importance of the search space and the negative impact of weight sharing. We believe that our evaluation framework will be instrumental in designing NAS search strategies that are superior to the random one.

## 5.2 Evaluating the NAS Search

In this section, we detail our evaluation framework for the NAS search phase. As depicted in Figure 5.1(a,b), typical NAS algorithms consist of two phases:

[1] The Kendall Tau (Kendall, 1938) metric measures the correlation of two ranking. Details in Section 5.2.5.

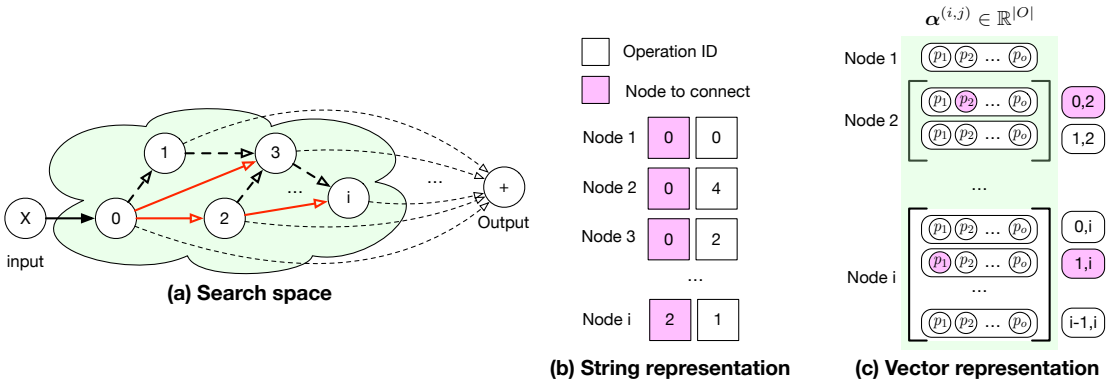



Figure 5.2: **Search space of NAS algorithms.** Typically, the search space is encoded as **(a)** a directed acyclic graph, and an architecture can be represented as **(b)** a string listing the node ID that each node is connected to, or the operation ID employed by each node. **(c)** An alternatively representation is a list of vectors $\alpha$ of size $\frac{n(n+1)}{2}|\mathcal{O}|$, where $n$ is the number of nodes and $\mathcal{O}$ is the set of all operations. Each vector, $\alpha^{(i,j)}$, captures, via a softmax, the probability $p_o$ that operation $o$ is employed between node $i$ and $j$. Note that any node only takes one incoming edge, thus (b) and (c) represent the same search space and only differs in its formality.

- **Search:** The goal of this phase is to find the best candidate architecture from the search space. This is where existing algorithms, such as ENAS, DARTS and NAO, differ. Nevertheless, for all the algorithms, the search depends heavily on initialization. In all the studied policies, initialization is random and the outcome thus depends on the chosen random seed.
- **Evaluation:** In this phase, all the studied algorithms retrain the best model found in the search phase. The retrained model is then evaluated on the test data.

The standard evaluation of NAS techniques focuses solely on the final results on the test data. Here, by contrast, we aim to evaluate the search phase itself, which truly differentiates existing algorithms.

To do this, as illustrated in Figure 5.1(c), we establish a baseline; we compare the search phase of existing algorithms with a random search policy. An effective search algorithm should yield a solution that clearly outperforms the random policy. Below, we introduce our framework to compare NAS search algorithms with random search. The three NAS algorithms that we evaluated, DARTS (Liu et al., 2019b), NAO (Luo et al., 2018a) and ENAS (Pham et al., 2018b), are representative of the state of the art for different search algorithms: reinforcement learning, gradient-descent and performance prediction.

### 5.2.1 NAS Search Space Representation

Our starting point is a neural search space for a neural architecture, as illustrated in Figure 5.2. A convolutional cell can be represented with a similar topological structures. Following common practice in NAS (Zoph & Le, 2017a), a candidate architecture sampled from this space connects

the input and the output nodes through a sequence of intermediary ones. Each node is connected to others and has an operation attached to it.

A way of representing this search space (Pham et al., 2018b; Luo et al., 2018a), depicted in Figure 5.2(b), is by using strings. Each character in the string indicates either the node ID that the current node is connected to, or the operation selected for the current node. Operations include the identity, sigmoid, tanh and ReLU (Nair & Hinton, 2010).

Following the alternative way introduced in (Liu et al., 2019b), we make use of a vectorized representation of these strings. More specifically, as illustrated by Figure 5.2(c), a node ID, resp. an operation, is encoded as a vector of probabilities over all node IDs, resp. all operations. For instance, the connection between nodes $i$ and $j$ is represented as $y^{(i,j)}(x) = \sum_{o \in \mathcal{O}} p_o o(x)$, with $\mathcal{O}$ the set of all operations, and $p_o = \texttt{softmax}(\alpha_o) = \exp(\alpha_o) / \sum_{o' \in \mathcal{O}} \exp(\alpha_{o'})$ the probability of each operation.

### 5.2.2 NAS algorithms

Here, we discuss the three state-of-the-art NAS algorithms used in our experiments in detail, including their hyper-parameters during the search phase. The current state-of-the-arts NAS on CIFAR-10 is ProxylessNAS (Cai et al., 2018b) with a top-1 accuracy of 97.92. However, this algorithm inherits the sampler from ENAS and DARTS, but with a different objective function, backbone model, and search space. In addition, the code is not publicly available, which precludes us from directly evaluating it.

**ENAS** adopts a reinforcement learning sampling strategy that is updated with the REINFORCE algorithm. The sampler is implemented as a two-layer LSTM Hochreiter & Schmidhuber (1997) and generates a sequence of strings. In the training process, each candidate sampled by the ENAS controller is trained on an individual mini-batch. At the end of each epoch, the controller samples new architectures that are evaluated on a single batch of the validation dataset. After this, the controller is updated accordingly using these validation metrics. We refer the reader to (Pham et al., 2018b) for details about the hyper-parameter settings.

**DARTS** It vectorizes the aforementioned strings as discussed in Section 5.2.1 and shown in Figure 5.2(c). The sampling process is then parameterized by the vector $\alpha$, which is optimized via gradient-descent in a dual optimization scheme: The architecture is first trained while fixing $\alpha$, and $\alpha$ is then updated while the network is fixed. This process is repeated in an alternating manner. In the evaluation phase, DARTS samples the top-performing architecture by using the trained $\alpha$ vector as probability prior, i.e., the final model is not a soft average of all paths but one path in the DAG, which makes its evaluation identical to that of the other NAS algorithms. Note that we use the same hyper-parameters as in the released code of Liu et al. (2019b).

**NAO** It implements a gradient-descent algorithm, but instead of vectorizing the strings as in

DARTS, it makes use of a variational auto-encoder (VAE) to learn a latent representation of the candidate architectures. Furthermore, it uses a performance predictor, which takes a latent vector as input to predict the corresponding architecture performance. In short, the search phase of NAO consists of first randomly sampling an initial pool of architectures and training them so as to obtain a ranking. This ranking is then used to train the encoder-predictor-decoder network, from which new candidates are sampled, and the process is repeated in an iterative manner. The best architecture is then taken as the top-1 in the NAO ranking. We directly use the code released by Luo et al. (2018a).

**BayesNAS** Bayesian optimization was first introduced to the neural architecture search field by Kandasamy et al. (2018) and Jin et al. (2019). We chose to evaluate BayesNAS (Zhou et al., 2019) because it is more recent than Auto-Keras (Jin et al., 2019) and than the work of Kandasamy et al. (2018), and because these two works use different search spaces than DARTS, resulting in models with significantly worse performance than DARTS. BayesNAS adopts Bayesian optimization to prune the fully-connected DAG graph using the shared weights to obtain accuracy metrics. The search space follows that of DARTS (Liu et al., 2019b) with minor modifications in connections, but exactly the same operations. Please see (Zhou et al., 2019) for more details. Note that BayesNAS was only implemented in CNN space. We use the search and model code released by Zhou et al. (2019) with our training pipeline, since the authors did not release the training code.

### 5.2.3 Comparing to Random Search

We implement our random search policy by simply assigning uniform probabilities to all operations. Then, for each node in the Directed Acyclic Graph (DAG) that is typically used to represent an architecture, we randomly sample a connection to *one* previous node from the resulting distributions.

An effective search policy should outperform the random one. To evaluate this, we compute the validation results of the best architecture found by the NAS algorithm trained from scratch, as well as those of *a single* randomly sampled architecture. Comparing these values for a single random seed would of course not provide a reliable measure. Therefore, we repeat this process for multiple random seeds used both during the search phase of the NAS algorithm and to sample one random architecture as described above. We then report the means and standard deviations of these results over the different seeds. Note that while we use different seeds for the search and random sampling, we always use the same seed when training the models from scratch during the evaluation phase.

Our use of multiple random seeds and of the same number of epochs for the NAS algorithms and for our random search policy makes the comparison fair. This contrasts with the comparisons performed in (Pham et al., 2018b), where the results of only *a single random* architecture were reported, and in (Liu et al., 2019b), which selected a single best random architecture among

an initial set of 8 after training for 300 epochs only. As shown in Section 5.3.5, some models that perform well in the early training stages may yield worse performance than others after convergence. Therefore, choosing the best random architecture after only 300 epochs for PTB and 100 for CIFAR-10, and doing so for a single random seed, might not be representative of the general behavior.

### 5.2.4 Search in a reduced space

Because of the size of standard search spaces, one cannot understand the quality of the search by fully evaluating all possible solutions. Hence, we propose to make use of reduced search spaces with ground-truth architecture performances available to evaluate the search quality. For RNNs, we simply reduce the number of nodes in the search space from 12 to 2. Given that each node is identified by two values, the ID of the incoming node and the activation function, the space has a cardinality $|S| = n! * |\mathcal{O}|^n$, where $n = 2$ nodes and $|\mathcal{O}| = 4$ operations, thus yielding 32 possible solutions. To obtain ground truth, we train all of these architectures individually. Each architecture is trained 10 times with a different seed, which therefore yields a mean and standard deviation of its performance. The mean value is used as ground truth—the actual potential of the given architecture. These experiments took around 5000 GPU hours.

For CNNs, we make use of NASBench-101 (Ying et al., 2019), a CNN graph-based search space with 3 possible operations, conv3x3, conv1x1 and max3x3. This framework defines search spaces with between 3 and 7 nodes, with 423,624 architectures in 7-node case. To the best of our knowledge, we are the first to evaluate the NAS methods used in this chapter on NASBench.

### 5.2.5 Metrics to evaluate NAS algorithms

**Kendall Tau metric.** As a correlation measure, we make use of the *Kendall Tau ($\tau$)* metric (Kendall, 1938): a number in the range [-1, 1] with the following properties:

- $\tau = -1$: Maximum disagreement. One ranking is the opposite of the other.
- $\tau = 1$: Maximum agreement. The two rankings are identical.
- $\tau$ close to 0: A value close to zero indicates the absence of correlation.

**Probability to surpass random search** As discussed in Section 5.2.4, the goal of NASBench is to search for a CNN cell with up to 7 nodes and 3 operations, resulting in total 423,624 architectures. Each architecture is trained 3 times with different random initialization up to 108 epochs on the CIFAR-10 training set, and evaluated on the test split. Hence, the average test accuracy of these runs can be seen as the ground-truth performances. In our experiments, we use this to rank the architectures, from 1 (highest accuracy) to 423,624. Given the best architecture's rank $r$ after $n$ runs, and maximum rank $r_{max}$ equals to the total number of architectures, the probability that the best architecture discovered is better than a randomly searched one given the

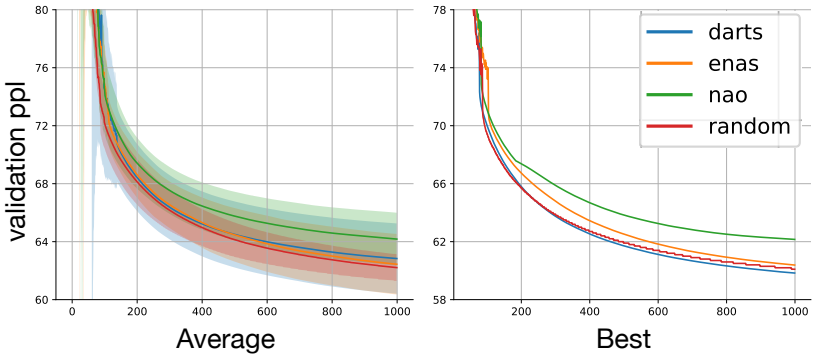



Figure 5.3: **Validation perplexity evolution in the 12-node RNN search space.** (Best viewed in color)

same budget is given by

$$p = 1 - (1 - (r/r_{max}))^n. \tag{5.1}$$

We use this as a new metric to evaluate the search phase.

## 5.3 Experimental Results

To analyze the search phase of the three state-of-the-art NAS algorithms mentioned above, we first compare these algorithms to our random policy when using standard search spaces for RNNs on (PTB) and CNNs on CIFAR-10. The surprising findings in this typical NAS use case prompted us to study the behavior of the search strategies in reduced search spaces. This allowed us to identify a factor that has a significant impact on the observed results: Weight sharing. We then quantify this impact on the ranking of the NAS candidates, evidencing that it dramatically affects the effectiveness of the search.

### 5.3.1 NAS Comparison in a Standard Search Space

Below, we compare DARTS (Liu et al., 2019b), NAO (Luo et al., 2018a), ENAS (Pham et al., 2018b) and BayesNAS (Zhou et al., 2019) with our random search policy, as discussed in Section 5.2.3. We follow (Liu et al., 2019b) to define an RNN search space of 12 nodes and a CNN ones of 7 nodes. For each of the four search policies, we run 10 experiments with a different initialization of the sampling policy. During the search phase, we used the authors-provided hyper-parameters and code for each policy. Once a best architecture is identified by the search phase, it is used for evaluation, i.e., we train the chosen architecture from scratch for 1000 epochs for RNN and 600 for CNN.

**RNN Results.**

In Figure 5.3, we plot, on the left, the mean perplexity evolution over the 1000 epochs, obtained

Table 5.2: **Top 1 accuracy in the original DARTS Search space.** We report the mean and best top-1 accuracy on the test sets of architectures found by DARTS, NAO, ENAS, BayesNAS, and our random policy. As sanity check, we also train from scratch the architectures reported in original papers, as well as their reported performance.

| | Our seed | | Best reported result | |
|---|---|---|---|---|
| Type | Mean test | Best test | Original | Reproduced |
| DARTS | 96.62 ± 0.23 | 96.80 | **97.24** | **97.15** |
| NAO | **96.86 ± 0.17** | **97.10** | 96.47 | 96.92 |
| ENAS | 96.76 ± 0.10 | 96.95 | 96.46 | 96.87 |
| BayesNAS | 95.99 ± 0.25 | 96.41 | 97.19 | 97.13 |
| Random | 96.48 ± 0.18 | 96.74 | 97.15 † | |

†Result took from Li & Talwalkar (2019)

by averaging the results of the best architectures found using the 10 consecutive seeds.[2] On the right, we show the perplexity evolution for the best cell of each strategy among the 10 different runs. Random sampling is robust and consistently competitive. As shown in Table 5.1, it outperforms on average the DARTS and NAO policies, and yields the overall best cell for these experiments with perplexity of 57.60. Further training this cell for 4000 epochs, as in (Liu et al., 2019b), yields a perplexity of 55.93. The excellent performance of the random policy evidences the high expressiveness of the manually-constructed search space; even arbitrary policies in this space perform well, as evidenced by the relatively low standard deviation over the 10 seeds of the random architectures, shown in Table 5.1 and Figure 5.3(left).

**CNN Results.** In Table 5.2, we compare the NAS methods with our random policy in the search space of Liu et al. (2019b). We provide the accuracy reported in the original papers as well as the accuracy we reproduced using our implementation. Note that the NAS algorithms only marginally outperform random search, by less than 0.5% in top-1 accuracy. The best architecture was discovered by NAO, with an accuracy of 97.10%, again less than 0.5% higher than the randomly discovered one. Note that, our random sampling comes at no search cost. By contrast, Li & Talwalkar (2019) obtained an accuracy of 97.15% with a different random search policy having the same cost as DARTS.

**Observations:**

- The evaluated state-of-the-art NAS algorithms do not surpass random search by a significant margin, and even perform worse in the RNN search space.
- The ENAS policy sampler has the lowest variance among the three tested ones. This shows that ENAS is more robust to the variance caused by the random seed of the search phase.
- The NAO policy is more sensitive to the search space; while it yields the best performance in CNN space, it performs the worst in RNN one.

[2]Starting from 1268, which is right after 1267, the seed released by Liu et al. (2019b). Note that, using this seed, we can reproduce the DARTS RNN search and obtain a validation PPL of 55.7 as in (Liu et al., 2019b).

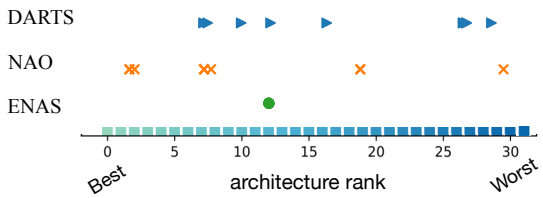



Figure 5.4: **Architectures discovered by NAS algorithms.** We rank all 32 architectures in the reduced search space based on their performance of individual training, from left (best) to right (worst), and plot the best cell found by three NAS algorithms across the 10 random seeds.

- The DARTS policy is very sensitive to random initialization, and yields the largest standard deviation across the 10 runs (2.54 in RNN and 0.23 in CNN space).

Such a comparison of search policies would not have been possible without our framework. Nevertheless, the above analysis does not suffice to identify the reason behind these surprising observations. As mentioned before, one reason could be that the search space has been sufficiently constrained so that all architectures perform similarly well. By contrast, if we assume that the search space does contain significantly better architectures, then we can conclude that these search algorithms truly fail to find a good one. To answer this question, we evaluate these methods in a reduced search space, where we can obtain the true performance of all possible architectures.

### 5.3.2 Searching a Reduced Space

The results in the previous section highlight the inability of the studied methods to surpass random search. Encouraged by these surprising results, we then dig deeper into their causes. Below, we make use of search spaces with fewer nodes, which we can explore exhaustively.

**Reduced RNN space.** We use the same search space as in Section 5.2.4 but reduce the number of intermediate nodes to 2. In Table 5.3 (A), we provide the results of searching the RNN 2-node space. Its smaller size allows us to exhaustively compute the results of all possible solutions, thus determining the upper bound for this case. In Figure 5.4, we plot the rank of the top 1 architecture discovered by the three NAS algorithms for each of the 10 different runs.

We observe that: (i) All policies failed to find the architecture that actually performs best; (ii) The ENAS policy always converged to the same architecture. This further evidences the robustness of ENAS to the random seed; (iii) NAO performs better than random sampling on average because it keeps a ranking of architectures; (iv) DARTS never discovered a top-5 architecture.

**Reduced CNN space.** In Table 5.3 (B), we report the mean and best test top-1 accuracy over 10 different runs on the NASBench-101 7-node space. To assess the search performance, we also show the best architecture rank in the entire space. The best test accuracy found by these methods is 93.33, by NAO, which remains much lower than the ground-truth best of 95.06. In terms of ranking, the best rank of these methods across 10 runs is 19522, which is among the

Table 5.3: **Results in reduced search spaces.** For RNNs (A), We report the mean and best perplexity on the validation and test sets at the end of training the architectures found using DARTS, NAO, ENAS. For CNNs (B), we show the mean and best top-1 accuracy on the test set. Instead of running random sampling in the reduced space, we compute the probability of the best model found by each method to surpass the random one (details in Section 5.2.5). The mean and best statistics of the entire search space are / reported as Space.

| | (A) **RNN** n=2(32) in PPL. | | | | (B) **NASBench** n=7(423K) | | | |
|---|---|---|---|---|---|---|---|---|
| Type | Mean Valid | Mean Test | Best Valid | Best Test | Mean Acc. | Best Acc. | Best Rank | p(>random) |
| DARTS | 71.29 ± 2.45 | 68.74 ± 2.42 | 68.05 | 65.55 | 92.21 ± 0.61 | 93.02 | 57079 | 0.24 |
| NAO | **68.66 ± 2.50** | **66.03 ± 2.40** | **66.22** | **63.59** | **92.59 ± 0.59** | **93.33** | **19552** | **0.62** |
| ENAS | 69.99 ± 0.0 | 66.61 ± 0.0 | 69.99 | 66.61 | 91.83 ± 0.42 | 92.54 | 96939 | 0.07 |
| Space | 69.69 ± 2.44 | 67.21 ± 2.52 | 65.38 | 62.63 | 90.93 ± 5.84 | 95.06 | - | - |

top 4% architectures and yields a probability of 0.62 to surpass a randomly-sampled one given the same search budget. Note that ENAS and DARTS only have 7% and 24% chance to surpass the random policy. Note that we provide a more comprehensive comparison on this space in Section 5.3.6.

NAO seems to constantly outperform random search in the reduced space. Nevertheless, the final architecture chosen by NAO is *always* one of the architectures from the initial pool, which were sampled uniformly randomly. This indicates that the ranking of NAO is not correctly updated throughout the search and that, in practice, in a reduced space, NAO is similar to random search.

### 5.3.3 Impact of Weight Sharing

Our previous experiments in reduced search spaces highlight that the ranking of the searched architectures does not reflect the ground-truth one. As we will show below, this can be traced back to weight sharing, which all the tested algorithms, and the vast majority of existing ones, rely on. To evidence this, we perform the following experiments:

**Without WS:** We make use of the reduced space, where we have the architecture's real performance.

**With WS:** We train the architectures in parallel, using the weight sharing strategy employed in NAO and ENAS. As DARTS does not have discrete representations of the solutions during the search, the idea of solution ranking does not apply. During training, each mini-batch is given to an architecture uniformly sampled from the search space. We repeat the process 10 times, with 10 random seeds and train the shared weights for 1000 epochs for the RNN experiments and 200 epochs for the CNN ones. Note that, this approach is equivalent to Single Path One Shot (SPOS) (Guo et al., 2019). It guarantees equal expectations of the number of times each architecture is sampled, thus overcoming the bias due to unbalanced training resulting from ineffective sampling policies.

We then compute the correlation between the architecture rankings found with WS and the ground


(a) Architecture performance with and without WS b) Rank difference due to weight sharing c) Example

Figure 5.5: **Rank disorder due to weight sharing in RNN reduced space. (a)** We report the average and std over 10 different runs. Note that the rankings significantly differ between using (line plot) and not using WS (bar plot), showing the negative impact of this strategy. **(b)** We visualize from left to right, the best, worst and average cases, and show the corresponding Kendall Tau value. A change in ranking, indicated by the colors and numbers, is measured as the absolute position change between the WS ranking and the true one. For conciseness, we only show the top 10 architectures. **(c)** For example, in the average scenario, the 6-th best architecture is wrongly placed as the best one, as indicated by the red arrow.

Table 5.4: **Search results w/o weight sharing.** We report results from ENAS ans NAO on NASBench with 7 nodes over 10 runs.

| Type | Mean Acc. | Best Acc. | Best Rank | P(>random) |
|---|---|---|---|---|
| NAO | 93.08 ± 0.71 | 94.11 | 3543 | 0.92 |
| ENAS | 93.54 ± 0.45 | 94.04 | 4610 | 0.90 |

truth (i.e., the architectures trained independently). For each of the 10 runs of the weight sharing strategy, we evaluate the Kendall Tau metric (defined in Section 5.2.5) of the final rankings with respect to the real averaged ranking.

**RNN Results.** In Figure 5.5(a), we depict the architecture performance obtained without WS (sorted in ascending order of average validation perplexity), and the corresponding performance with WS. In Figure 5.5(b), we show the rank difference, where the best and worst were found using the Kendall Tau metric, and show a concrete rank change example in Figure 5.5(c).

**CNN Results.** We report the average Kendall tau across 10 different runs. Note that we sampled up to 200 architectures for each experiment and fully evaluated on the entire test set to use the test accuracy for ranking. The Kendall tau for search spaces from 3 to 7 nodes is, respectively, 0.441, 0.314, 0.214, 0.195. We also provide other statistics in Table 5.6 of Appendix 5.3.6. Since NAO and ENAS intrinsically disentangle the training of shared weights and sampler, to further confirm the negative effect of weight sharing, we adapt these algorithms to use the architecture's performance in the NASBench dataset to train their sampler. Table 5.4 evidences that, after removing weight sharing, both ENAS and NAO consistently discover a good architecture, as

indicated by a small difference between the best over 10 runs and the mean performance. More interestingly, for the 7-node case, the best cell discovered (94.11% by NAO and 94.04% by ENAS) are more than 1% higher than the best cells found with weight sharing (93.33 and 92.54, respectively, in Table 5.3).

**Observations:**

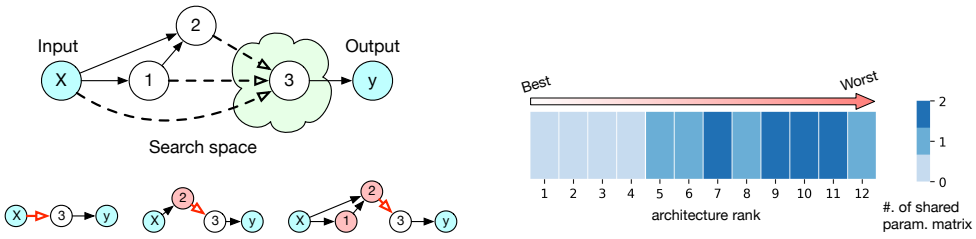
- The difference of architecture performance is *not* related to the use of different random seeds, as indicated by the error bars in Figure 5.5(a).
- WS *never* produces the true ranking, as evidenced by the Best case in Figure 5.5(b).
- The behavior of the WS rankings is greatly affected by changing the seed. In particular, the Kendall Tau for the plots in Figure 5.5(b) are $0.282, -0.004, -0.116$ for Best, Average and Worst.
- For RNNs, the Kendall Tau are close to 0, which suggests a lack of correlation between the WS rankings and the true one. By contrast, for CNNs, the correlation is on average higher than for RNNs. This matches the observation in Section 5.3.1 that CNN results are generally better than RNN ones.
- If we train NAO and ENAS without weight sharing in NASBench, on average the performance is 1% higher than them with it. This further evidences that weight sharing negatively impacts the sampler, and with a good ranking, the sampler can be trained better. Furthermore, the probability to surpass random search increases from 0.62 to 0.92 for NAO and from 0.07 to 0.90 for ENAS.

Together with previous results, we believe that these results evidence the negative impact of weight sharing; it dramatically affects the performance of the sampled architectures, thus complicating the overall search process and leading to search policies that are no better than the random one.

### 5.3.4 Influence of the Amount of Sharing

Depending on the active connections in the DAG, different architectures are subject to different amounts of weight sharing. In Figure 5.6 (a), let us consider the 3-node case, with node 1 and node 2 fixed and node 3 having node 1 as incoming node. In this scenario, the input to node 3 can be either directly node 0 (i.e., the input), or node 1, or node 2. In the first case, the only network parameters that the output of node 3 depends on are the weights of its own operation. In the second and third cases, however, the output further depends on the parameters of node 1, and of nodes 1 and 2, respectively.

To study the influence of the amount of sharing on the architecture ranking, we performed an experiment where we fixed the first two nodes and only searched for the third one. This represents a space of 12 architectures (3 possible connections to node 3 $\times$ 4 operations). We train them using the same setting in Section 5.3.3. The ranking of the 12 architectures is shown in Figure 5.6 (b), where color indicates the number of shared weight matrices, that is, matrices of nodes 1 and 2




(a) Search Space (b) Impact of weight sharing

Figure 5.6: **A toy search space to assess the influence of weight sharing in RNN space.** (a) Top: the reduced space. (a) Bottom: The amount of sharing depends on the activated path. (b) Ranking obtained from weight sharing training, that best ranked architectures has weight matrix to share.

also used in the search for node 3. Note that the top-performing architectures do not share any weights and that the more weights are shared, the worse the architecture performs.

In CNN space, we conduct a similar experiment in NAS-Bench. With total node equals to 6, we only permute the last node operation and connection to one of the previous nodes. In short, we will have a total 4 connection possibility and 3 operation choices, in total 12 architectures. We compute the Kendall Tau among the architectures with the same connection but different operations, and the results are reported in Table 5.5. Clearly, the correlation of architectures decrease while the weight sharing matrices increase.

Table 5.5: **Ranking disorder of weight sharing in CNN.**

| # of shared matrix | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Kendall Tau $\tau$ | 0.67. | 0.33 | -0.33 | 0.0 |

### 5.3.5 Random Sampling Comparison

As discussed before, the random policy in (Liu et al., 2019b) samples 8 architectures, and picks the best after training them for 300 epochs independently. It might seem contradictory that DARTS outperforms this random policy, but cannot surpass the much simpler one designed in this chapter, which only randomly samples 10 architectures (1 per random seed), trains them to convergence and picks the best. However, the random policy in DARTS relies on the assumption that a model that performs well in the early training stage will remain effective until the end of training. While this may sound intuitive, we observed a different picture with our reduced search space.

Since we obtained the ground-truth performance ranking, as discussed in Section 5.3.2, in Figure 5.7, we plot the evolution of models' rank while training proceeds, based on the average validation perplexity over 10 runs. Clearly, there are significant variations during training: Good models in early stages drop lower in the ranking towards the end. As such, there is a non-negligible chance that the random policy in DARTS picks a model whose performance will be sub-optimal. We therefore believe that our policy that simply samples one model and trains it until convergence yields a more fair baseline. Furthermore, the fact that we perform our comparison

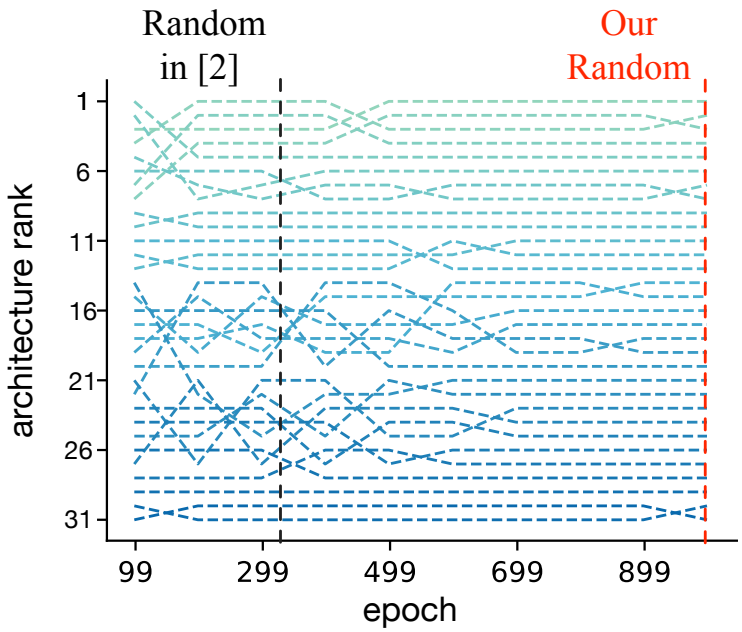



Figure 5.7: **Rank changes while training**. Each line represents the evolution of the rank of a single architecture. The models are sorted based on their test performance after 1000 epochs, with the best-performing one at the top. The curves were averaged over 10 runs. They correspond to the experiment in Section 4.2. The vertical dashed lines indicate the epoch number where random sampling was performed, either by the random policy in Liu et al. (2019b), or by ours.

Table 5.6: **Comparison of state-of-the-art methods on NASBench-101 search space with CIFAR-10.** n: number of nodes, (x): total architecture choices, mean and best: top 1 accuracy (in %)

| Search Space | n = 4 (91) | | | n = 5 (2.5K) | | |
|---|---|---|---|---|---|---|
| Method | Mean | Best | K-T | Mean | Best | K-T |
| *Sampling methods, train sampler during training super-net* | | | | | | |
| ENAS | 89.41 ± 3.54 | 92.95 | - | 89.03 ± 2.76 | 91.84 | - |
| NAO | **92.87** ± 0.69 | 93.88 | - | 92.07 ± 1.14 | **93.97** | - |
| DARTS | 91.54 ± 1.93 | 93.71 | - | 91.82 ± 1.10 | 93.63 | - |
| FBNET | 91.56 ± 1.89 | 93.71 | - | **92.51** ± 1.51 | 93.90 | - |
| *One-shot methods, train sampler after optimizing super-net* | | | | | | |
| SPOS | 91.14 ± 3.47 | **94.24** | 0.441 | 91.53 ± 1.76 | 93.72 | 0.314 |
| FAIRNAS | 89.08 ± 4.35 | 94.13 | -0.043 | 91.38 ± 1.44 | 93.55 | -0.028 |

| Search Space | n = 6 (64K) | | | n = 7 (423K) | | | Best of |
|---|---|---|---|---|---|---|---|
| Method | Mean | Best | K-T | Mean | Best | K-T | all n |
| *Sampling methods, train sampler during training super-net* | | | | | | | |
| ENAS | 91.41 ± 1.42 | 92.75 | - | 91.83 ± **0.42** | 92.54 | - | 93.69 |
| NAO | 92.83 ± **0.78** | 93.62 | - | **92.59** ± 0.59 | 93.33 | - | 93.97 |
| DARTS | 91.12 ± 1.86 | 93.92 | - | 92.21 ± 0.61 | 93.02 | - | 93.92 |
| FBNET | 91.76 ± 1.26 | 92.98 | - | 92.29 ± 1.25 | **93.98** | - | 93.98 |
| *One-shot methods, train sampler after optimizing super-net* | | | | | | | |
| SPOS | 90.56 ± 1.03 | 92.29 | 0.214 | 89.85 ± 3.80 | 93.84 | 0.195 | 94.24 |
| FAIRNAS | 91.75 ± 2.20 | **94.47** | -0.221 | 91.10 ± 1.84 | 93.55 | -0.232 | **94.47** |

using 10 random seeds, for both our approach and the NAS algorithms, vs a single one in (Liu et al., 2019b) makes our conclusions more reliable.

### 5.3.6 NASBench detailed results.

We provide additional evaluations on the NASBench dataset to benchmark the performance of the state-of-the-art NAS algorithms. In addition to the aforementioned three methods, we re-implemented some recent algorithms, such as FBNet (Wu et al., 2018), Single Path One Shot (SPOS) (Guo et al., 2019), and FairNAS (Chu et al., 2019). Note that we removed the FBNet device look-up table and model latency from the objective function since the search for a mobile model is not our primary goal. This also makes it comparable with the other baselines.

To ensure fairness, after the search phase is completed, each method trains the top 1 architectures found by its policy from scratch to obtain ground-truth performance; we repeated all the experiments with 10 random seeds. We report the mean and best top 1 accuracy in Table 5.6 for a number of nodes $n \in [4, 7]$, and the Kendall Tau (K-T) values for one-shot methods following

Section 5.3.2 in the chapter.

From the results, we observe that: 1) Sampling-based NAS strategies always have better mean accuracy with lower standard deviation, meaning that they converge to a local minimum more easily but do not exploit the entire search space. 2) By contrast, one-shot methods explore more diverse solutions, thus having larger standard deviations but lower means, but are able to pick a better architecture than sampling-based strategies (94.47 for FairNAS and 94.24 for SPOS, vs best of sampler based FBNet 93.98). 3) ENAS constantly improves as the number of nodes increases. 4) FBNet constantly outperforms DARTS, considering the similarity, using Gumbel Softmax seems a better choice. 5) The variance of these algorithms is large and sensitive to initialization. 6) Even one-shot algorithms cannot find the overall best architecture with accuracy 95.06.

## 5.4 Conclusion

In this chapter, we have analyzed the effectiveness of the search phase of NAS algorithms via fair comparisons to random search. We have observed that, surprisingly, the search policies of state-of-the-art NAS techniques are no better than random, and have traced the reason for this to the use of (i) a constrained search space and (ii) weight sharing, which shuffles the architecture ranking during the search, thus negatively impacting it.

In essence, our gained insights highlight two key properties of state-of-the-art NAS strategies, which had been overlooked in the past due to the single-minded focus of NAS evaluation on the results on the target tasks. We believe that this will be key to the development of novel NAS algorithms.

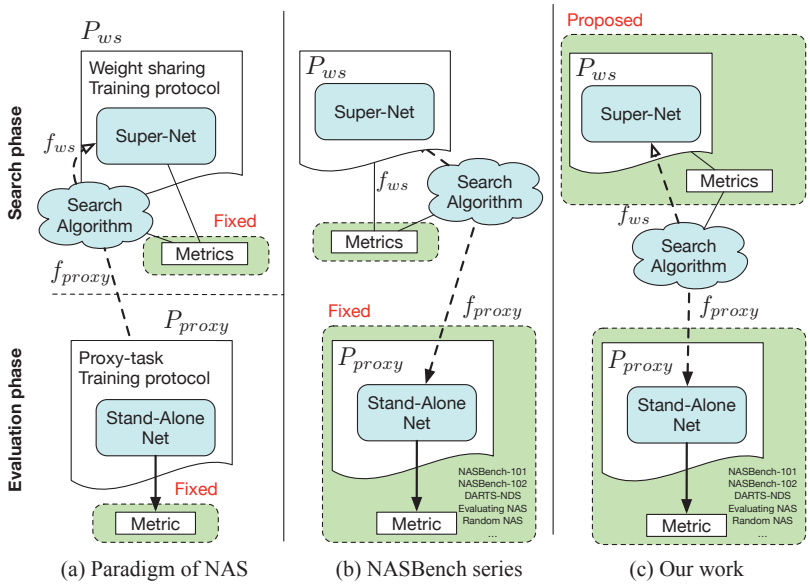
# 6 How to Train Your Super-net

Although weight sharing promises to make neural architecture search (NAS) tractable even on commodity hardware, in the previous chapter, we have shown that weight sharing negatively impacts the search performance. Existing weight sharing NAS methods rely on a diverse set of heuristics to design and train the shared-weight backbone network, a.k.a. the super-net. Since heuristics substantially vary across different methods and have not been carefully studied, it is unclear to which extent they impact super-net training and hence the weight-sharing NAS algorithms. In this chapter, we disentangle super-net training from the search algorithm, isolate 14 frequently-used training heuristics, and evaluate them over three benchmark search spaces. Our analysis uncovers that several commonly-used heuristics negatively impact the performance correlation between using the parameters from the stand-alone training and the super-net training, whereas simple, but often overlooked factors, such as proper hyper-parameter settings, are key to achieve strong performance. Equipped with this knowledge, we show that simple random search achieves competitive performance to complex state-of-the-art NAS algorithms when the super-net is properly trained.

## 6.1 Introduction

Neural architecture search (NAS) has received growing attention in the past few years, yielding state-of-the-art performance on several machine learning tasks (Liu et al., 2019a; Wu et al., 2018; Chen et al., 2019b; Ryoo et al., 2020). One of the milestones that led to the popularity of NAS is weight sharing (Pham et al., 2018b; Liu et al., 2019b), which, by allowing all possible network architectures to share the same parameters, has reduced the computational requirements from thousands of GPU hours to just a few. Figure 6.1 shows the two phases that are common to weight-sharing NAS (WS-NAS) algorithms: the search phase, including the design of the search space and the search algorithm; and the evaluation phase, which encompasses the final training protocol on the target task [1].

[1] Target task refers to the tasks that neural architecture search aims to optimize on.




(a) Paradigm of NAS (b) NASBench series (c) Our work

Figure 6.1: **WS-NAS benchmarking.** Green blocks indicate which aspects of NAS are benchmarked in different works. A search algorithm usually consists of a search space that encompass many architectures, and a policy to select the best one. $P$ indicates a training protocol, and $f$ a mapping function from the search space to a neural network. **(a)** Early works fixed and compared the metrics on the proxy task, which doesn't allow for a holistic comparison between algorithms. **(b)** The NASBench benchmark series partially alleviates the problem by sharing the stand-alone training protocol and search space across algorithms. However, the design of the weight-sharing search space and training protocol is still not controlled. **(c)** We fill this gap by benchmarking existing techniques to construct and train the shared-weight backbone. We provide a controlled evaluation across three benchmark spaces.

While most works focus on developing a good sampling algorithm (Cai et al., 2018b; Xie et al., 2018) or improving existing ones (Zela et al., 2020a; Nayman et al., 2019; Li et al., 2020a), they tend to overlook or gloss over important factors related to the design and training of the shared-weight backbone network, i.e. the super-net. For example, the literature encompasses significant variations of learning hyper-parameter settings, batch normalization and dropout usage, capacities for the initial layers of the network, and depth of the super-net. Furthermore, some of these heuristics are directly transferred from standalone network training to super-net training without carefully studying their impact in this drastically different scenario. For example, the fundamental assumption of batch normalization that the input data follows a slowly changing distribution whose statistics can be tracked during training is violated in WS-NAS, but nonetheless typically assumed to hold.

In this paper, we revisit and systematically evaluate commonly-used super-net design and training heuristics and uncover the strong influence of certain factors on the success of super-net training. To this end, we leverage three benchmark search spaces, NASBench-101 (Ying et al., 2019), NASBench-201 (Dong & Yang, 2020), and DARTS-NDS (Radosavovic et al., 2019), for which the ground-truth stand-alone performance of a large number of architectures is available. We report the results of our experiments according to two sets of metrics: i) metrics that directly

measure the quality of the super-net, such as the widely-adopted super-net accuracy [2] and a modified Kendall-Tau correlation between the searched architectures and their ground-truth performance, which we refer to as *sparse Kendall-Tau*; ii) proxy metrics such as the ability to surpass random search and the stand-alone accuracy of the model found by the WS-NAS algorithm.

Via our extensive experiments (over 700 GPU days), we uncover that (i) the training behavior of a super-net drastically differs from that of a standalone network, e.g., in terms of feature statistics and loss landscape, thus allowing us to define training settings, e.g., for batch-normalization (BN) and learning rate, that are better suited for super-nets; (ii) while some neglected factors, such as the number of training epochs, have a strong impact on the final performance, others, believed to be important, such as path sampling, only have a marginal effect, and some commonly-used heuristics, such as the use of low-fidelity estimates, negatively impact it; (iii) the commonly-adopted super-net accuracy is unreliable to evaluate the super-net quality.

Altogether, our work is the first to systematically analyze the impact of the diverse factors of super-net design and training, and we uncover the factors that are crucial to design a super-net, as well as the non-important ones. Aggregating these findings allows us to boost the performance of simple weight-sharing random search to the point where it reaches that of complex state-of-the-art NAS algorithms across all tested search spaces. Our code is available at https://github.com/kcyu2014/nas-supernet, and we will release our trained models so as to establish a solid baseline to facilitate further research.

## 6.2 Preliminaries - A holistic overview of weight sharing NAS

We first introduce the necessary concepts that will be used throughout the paper. As shown in Figure 6.1*(a)*, weight-sharing NAS algorithms consist of three key components: a search algorithm that samples an architecture from the search space in the form of an encoding, a mapping function $f_{proxy}$ that maps the encoding into its corresponding neural network, and a training protocol for a proxy task $P_{proxy}$ for which the network is optimized.

To train the search algorithm, one needs to additionally define the mapping function $f_{ws}$ that generates the shared-weight network. Note that the mapping $f_{proxy}$ frequently differs from $f_{ws}$, since in practice the final model contains many more layers and parameters so as to yield competitive results on the proxy task. After fixing $f_{ws}$, a training protocol $P_{ws}$ is required to learn the super-net. In practice, $P_{ws}$ often hides factors that are critical for the final performance of an approach, such as hyper-parameter settings or the use of data augmentation strategies to achieve state-of-the-art performance (Liu et al., 2019b; Chu et al., 2019; Zela et al., 2020a). Again, $P_{ws}$ may differ from $P_{proxy}$, which is used to train the architecture that has been found by the search. For example, our experiments reveal that the learning rate and the total number of epochs frequently differ due to the different training behavior of the super-net and stand-alone

[2] The mean accuracy over a small set of randomly sampled architectures during super-net training.

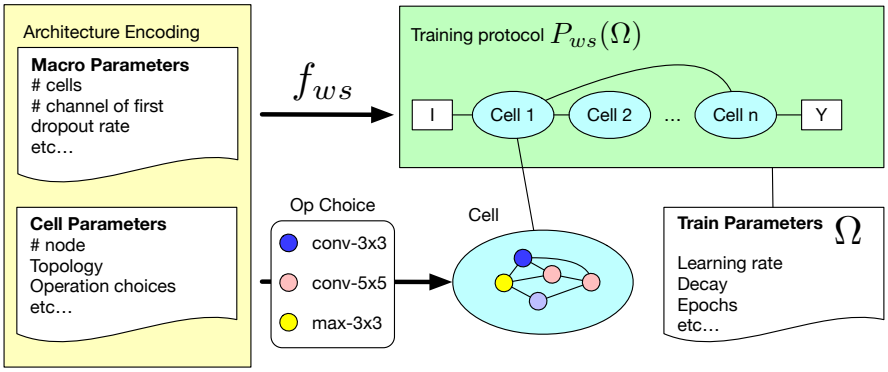



Figure 6.2: **Constructing a super-net.**

architectures.

## 6.3 Evaluation Methodology

We first isolate 14 factors that need to be considered during the design and training of a super-net, and then introduce the metrics to evaluate the quality of the trained super-net. Note that these factors are agnostic to the search policy that is used after training the super-net.

### 6.3.1 Disentangling the super-net from the search algorithm

Our goal is to evaluate the influence of the super-net mapping $f_{ws}$ and weight-sharing training protocol $P_{ws}$. As shown in Figure 6.2, $f_{ws}$ translates an architecture encoding, which typically consists of a discrete number of choices or parameters, into a neural network. Based on a well-defined mapping, the super-net is a network in which every sub-path has a one-to-one mapping with an architecture encoding (Pham et al., 2018b). Recent works (Xu et al., 2020; Li et al., 2020a; Ying et al., 2019) separate the encoding into *cell parameters*, which define the basic building blocks of a network, and *macro parameters*, which define how cells are assembled into a complete architecture.

**Weight-sharing mapping $f_{ws}$.** To make the search space manageable, all cell and macro parameters are fixed during the search, except for the topology of the cell and its possible operations. However, the exact choices for each of these fixed factors differ between algorithms and search spaces. We report the common factors in the left part of Table 6.1. They include various implementation choices, e.g., the use of convolutions with a dynamic number of channels (Dynamic Channeling), super-convolutional layers that support dynamic kernel sizes (OFA Kernel) (Cai et al., 2020), weight-sharing batch-normalization (WSBN) that tracks independent running statistics and affine parameters for different incoming edges (Luo et al., 2018a), and path and global dropout (Pham et al., 2018b; Luo et al., 2018a; Liu et al., 2019b). They also include the use of low-fidelity estimates (Elsken et al., 2019) to reduce the complexity of super-net training, e.g., by

Table 6.1: **Summary of factors**

| WS Mapping $f_{ws}$ | | WS Protocol $P_{ws}$ | |
|---|---|---|---|
| *implementation* | *low fidelity* | *hyperparam.* | *sampling* |
| Dynamic Channeling | # layer | batch-norm | FairNAS |
| OFA Conv | train portion | learning rate | Random-NAS |
| WSBN | batch size | epochs | Random-A |
| Dropout | # channels | weight decay | |
| Op on Node/Edge | | | |

reducing the number of layers (Liu et al., 2019b) and channels (Yang et al., 2020; Chen et al., 2019a), the portion of the training set used for super-net training (Liu et al., 2019b), or the batch size (Liu et al., 2019b; Pham et al., 2018b; Yang et al., 2020).

**Weight-sharing protocol** $P_{ws}$ Given a mapping $f_{ws}$, different training protocols $P_{ws}$ can be employed to train the super-net. Protocols can differ in the training hyper-parameters and the sampling strategies they rely on. We will evaluate the different hyper-parameter choices listed in the right part of Table 6.1. This includes the initial learning rate, the hyper-parameters of batch normalization, the total number of training epochs, and the amount of weight decay.

We randomly sample one path to train the super-net (Guo et al., 2019), which is also known as single-path one-shot (SPOS) or Random-NAS (Li & Talwalkar, 2019). The reason for this choice is that Random-NAS is equivalent to the initial state of many search algorithms (Liu et al., 2019b; Pham et al., 2018b; Luo et al., 2018a), some of which even freeze the sampler training so as to use random sampling to warm-up the super-net (Xu et al., 2020; Dong & Yang, 2019b). Note that we also evaluated two variants of Random-NAS, but found their improvement to be only marginal.

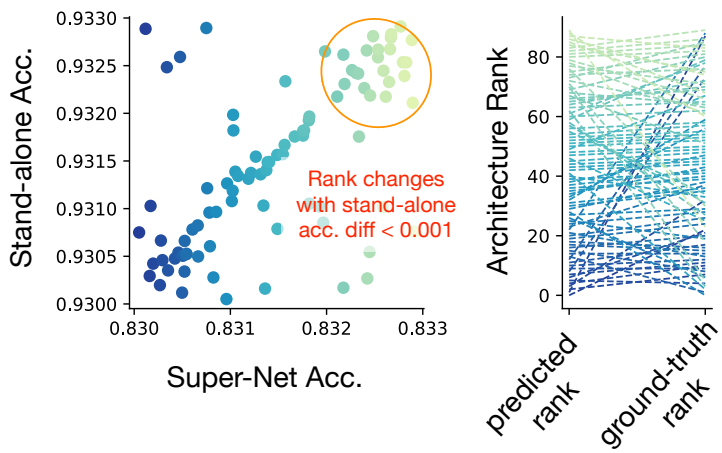
In our experiments, for the sake of reproducibility, we ensure that $P_{ws}$ and $P_{proxy}$, as well as $f_{ws}$ and $f_{proxy}$, are as close to each other as possible. For the hyper-parameters of $P_{ws}$, we cross-validate each factor following the order in Table 6.1, and after each validation, use the value that yields the best performance in $P_{proxy}$. For all other factors, we change one factor at a time.

### 6.3.2 Search spaces

We use three commonly-used search spaces, for which a large number of stand-alone architectures have been trained and evaluated on CIFAR-10 (Krizhevsky et al., 2009b) to obtain their ground-truth performance. In particular, we use NASBench-101 (Ying et al., 2019), which consists of $423,624$ architectures and is compatible with weight-sharing NAS (Yu et al., 2020c; Zela et al., 2020b); NASBench-201 (Dong & Yang, 2020), which contains more operations than NASBench-101 but fewer nodes; and DARTS-NDS (Radosavovic et al., 2019) that contains over $10^{13}$ architectures, of which a subset of 5000 models was sampled and trained in a stand-alone fashion. A summary of these search spaces and their properties is shown in Table 6.2. The

Table 6.2: **Search Spaces.**

| | NASBench-101 | NASBench-201 | DARTS-NDS |
|---|---|---|---|
| # Arch. | 423,624 | 15,625 | >$10^12$ |
| # Op. | 3 | 5 | 8 |
| Channel | Dynamic | Fix | Fix |
| Optimal | Global | Global | Sample |
| Nodes=($n$) | 5 | 4 | 4 |
| Param. | $O(n)$ | $O(n)$ - $O(n^2)$ | $O(n)$ - $O(n^2)$ |
| Edges | $O(n^2)$ | $O(n^2)$ | $O(n)$ |
| Merge | Concat. | Sum | Sum |




| | Kendall Tau |
|---|---|
| original | 0.6444 |
| sparse | 0.8140 |

Figure 6.3: **Kendall-Tau vs Sparse Kendall-Tau.** Kendall-Tau is not robust when many architectures have similar performance. Minor performance differences can lead to large perturbations in the ranking. Our sparse Kendall-Tau alleviates this by dismissing minor differences in performance.

search spaces differ in the number of architectures that have known stand-alone accuracy (# Arch.), the number of possible operations (# Op.), how the channels are handled in the convolution operations (Channel), where dynamic means that the number of super-net channels might change based on the sampled architecture, and the type of optimum that is known for the search space (Optimal). We further provide the maximum number of nodes ($n$), excluding the input and output nodes, in each cell, as well as a bound on the number of shared weights (Param.) and edge connections (Edges). Finally, the search spaces differ in how the nodes aggregate their inputs if they have multiple incoming edges (Merge).

### 6.3.3 Sparse Kendall-Tau - A novel super-net evaluation metric

We define a novel super-net metric, which we name *sparse Kendall-Tau*. It is inspired by the Kendall-Tau metric used in Chapter 5 to measure the discrepancy between the ordering of stand-

alone architectures and the ordering that is implied by the trained super-net. An ideal super-net should yield the same ordering of architectures as the stand-alone one and thus would lead to a high Kendall-Tau. However, Kendall-Tau is not robust to negligible performance differences between architectures (c.f. Figure 6.3). To robustify this metric, we share the rank between two architectures if their stand-alone accuracies differ by less than a threshold (0.1% here). Since the resulting ranks are sparse, we call this metric *sparse Kendall-Tau* (s-KdT).

**Implementation Details.** To compute the sparse Kendall-Tau we need access to two quantities: 1) the performance of the sampled architectures based on the trained super-net; and 2) the associated ground-truth performances. For each architecture in 1), we compute the average top-1 accuracy over $n = 3$ super-nets (that where trained with different random initialization) to improve the stability of the evaluation. We round the ground-truth top-1 accuracy to a precision of 0.1% for each sampled architecture to obtain the ground-truth performance 2). We then rank the architectures in 1) and 2) and compute the Kendall-Tau rank coefficient (Kendall, 1938) between the two ranked lists.

**Sparse Kendall-Tau threshold.** This value should be chosen according to what is considered a significant improvement for a given task. For CIFAR-10, where accuracy is larger than 90%, we consider a 0.1% performance gap to be sufficient. For tasks with smaller state-of-the-art performance, larger values might be better suited.

**Number of architectures.** In practice, we observed that the sparse Kendall-Tau metric became stable and reliable when using at least $n = 150$ architectures. We used $n = 200$ in our experiments to guarantee stability and fairness of the comparison of the different factors.

**Limitation of Sparse Kendall-Tau.** We nonetheless acknowledge that our sparse Kendall-Tau has some limitations. For example, a failure case of using sparse Kendall-Tau for super-net evaluation may occur when the top 10% architectures are perfectly ordered, while the bottom 90% architectures are purely randomly distributed. In this case, the Kendall Tau will be close to 0. However, the search algorithm will always return the best model, as desired.

Nevertheless, while this corner case would indeed be problematic for the standard Kendall Tau, it can be circumvented by tuning the threshold of our sKdT. A large threshold value will lead to a small number of groups, whose ranking might be more meaningful. For instance in some randomly-picked NASBench-101 search processes, setting the threshold to 0.1% merges the top 3000 models into 9 ranks, but still yields an sKdT of only 0.2. Increasing the threshold to 10% clusters the 423K models into 3 ranks, but still yields an sKdT of only 0.3. This indicates the stability of our metric.

In Figure 6.4, we randomly picked 12 settings and show the corresponding bipartite graphs relating the super-net and ground-truth rankings to investigate where disorder occurs. In practice, the corner case discussed above virtually never occurs; the ranking disorder is typically spread uniformly across the architectures.

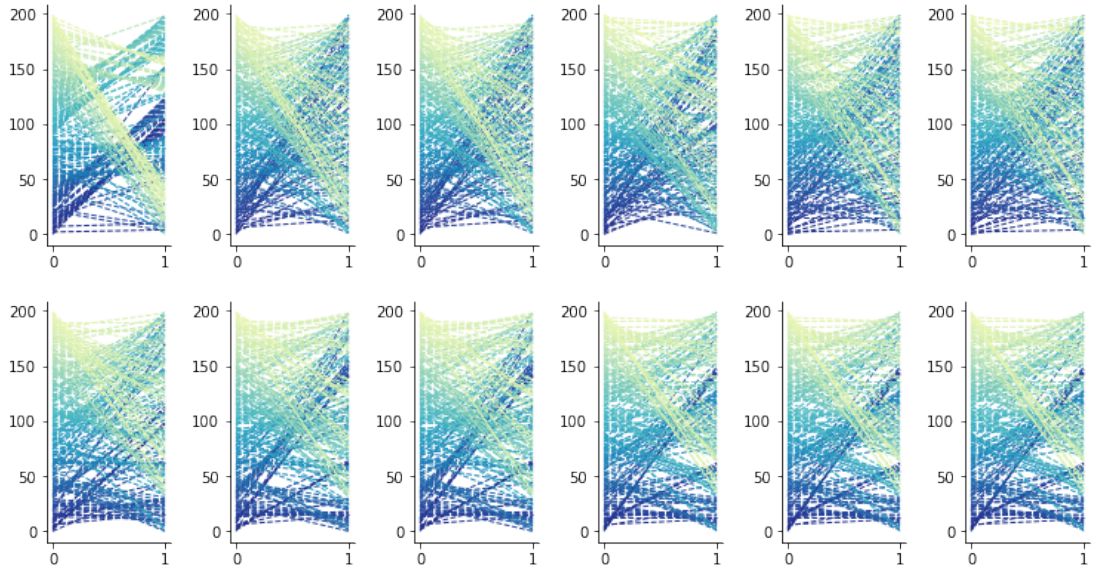



Figure 6.4: **Ranking disorder examples.** We randomly select 12 runs from our experiments. For each sub-plot, 0 indicates the architecture ground-truth rank, and 1 indicates the ranking according to their super-net accuracy. We can clearly see that the ranking disorder happens uniformly across the search space and does not follow a particular pattern.

### 6.3.4 Other metrics

Although, sparse Kendall-Tau captures the super-net quality well, it may fail in extreme cases, such as when the top-performing architectures are ranked perfectly while poor ones are ordered randomly. To account for such rare situations and ensure the soundness of our analysis, we also report additional metrics. We define two groups of metrics to holistically evaluate different aspects of a trained super-net.

The first group of metrics directly evaluates the quality of the super-net, including sparse Kendall-Tau and the widely-adopted super-net accuracy. For the super-net accuracy, we report the average accuracy of 200 architectures on the validation set of the dataset of interest. We will refer to this metric simply as *accuracy*. It is frequently used (Guo et al., 2019; Chu et al., 2019) to assess the quality of the trained super-net, but we will show later that it is in fact a poor predictor of the final stand-alone performance. The metrics in the second group evaluate the search performance of a trained super-net. The first metric is the probability to surpass random search: Given the ground-truth rank $r$ of the best architecture found after $n$ runs and the maximum rank $r_{max}$, equal to the total number of architectures, the probability that the best architecture found is better than a randomly searched one is given by $p = 1 - (1 - (r/r_{max}))^n$.

Finally, where appropriate, we report the stand-alone accuracy of the model that was found by the complete WS-NAS algorithm. Concretely, we randomly sample 200 architectures, select the 3 best models based on the super-net accuracy and query the ground-truth performance. We then take the mean of these architectures as stand-alone accuracy. Note that the same architectures are

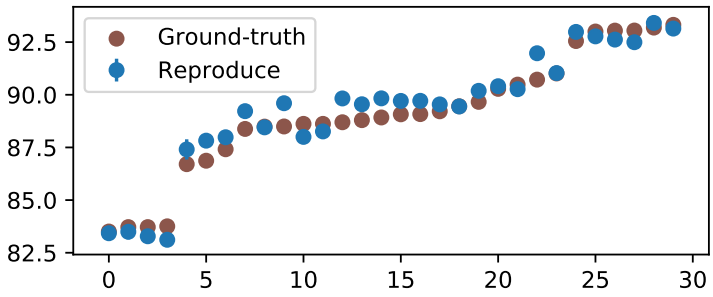



Figure 6.5: **Reproducing NASBench-101.**

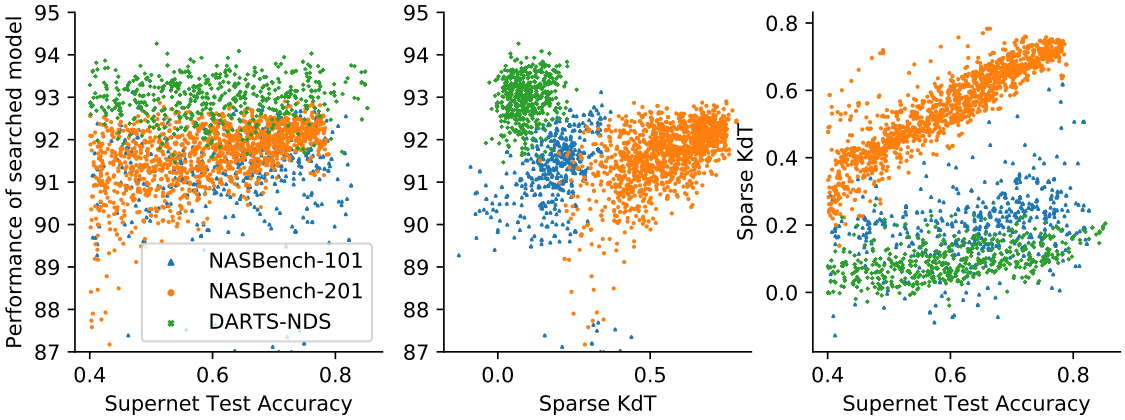
used to compute the sparse Kendall-Tau.

## 6.4 Analysis

We provide an analysis on the impact of the factors that are shown in Table 6.1 across three different search spaces. In addition, we report the complete numerical results of all metrics in Section 6.4.6.

**Training Details.** We use PyTorch (Paszke et al., 2019) for our experiments. Since NASBench-101 was constructed in TensorFlow we implement a mapper that translates TensorFlow parameters into our PyTorch model. We exploit two large-scale experiment management tools, SLURM (Slurm, 2020) and Kubernetes (Kubernetes, 2020), to deploy our experiments. We use various GPUs throughout our project, including NVIDIA Tesla V100, RTX 2080 Ti, GTX 1080 Ti and Quadro 6000 with CUDA 10.1. Depending on the number of training epochs, parameter sizes and batch-size, most of the super-net training finishes within 12 to 24 hours, with the exception of FairNAS, whose training time is longer, as discussed earlier. We split the data into training/validation using a 90/10 ratio for all experiments, except those involving validation on the training portion. Please consult our submitted code for more details.

**Reproducing the Ground Truth from Tensorflow.** As the ground-truth performance used by NASBench-101 are obtained on Tensorflow with TPU computation structure. We firstly reproduce these results in Pytorch with our implementation to make sure the re-implementation is trustworthy. We uniformly random sampled 10 architectures, and repeat 3 times. It results 30 architectures and covers the spectrum of performance from 82% to 93%. We adopted the optimizer and hyper-parameter setting according to the code release of NASBench-101, repeated with 3 random initializations and take the mean performance. We plot the performance comparison in Figure 6.5. The Kendall Tau metric is 0.81, and should be considered as the upper-bound of super-net training. It clearly indicates that even the reproducing results are not perfectly aligned with the Tensorflow original, it cannot explain why the significant drop to 0.2 after using weight sharing (Yu et al., 2020c).



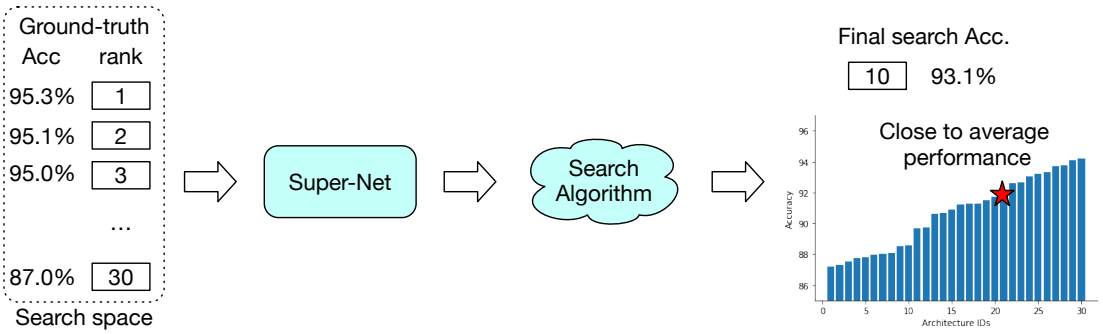

| Spearman Corr. | Acc. - Perf. | S-KdT - Perf. | Acc. - S-KdT |
|---|---|---|---|
| NASBench-101 | 0.09 | 0.45 | 0.23 |
| NASBench-201 | 0.52 | 0.55 | 0.94 |
| DARTS-NDS | 0.07 | 0.19 | 0.47 |

Figure 6.6: **Super-net evaluation**. We collect all experiments across 3 benchmark spaces. **(Top)** Pairwise plots of super-net accuracy, final performance, and the sparse Kendall-Tau. Each point corresponds to statistics computed over a trained super-net. Note that for super-net accuracy, we filtered out the accuracies below 40% to remove the statistics of ill-trained super-nets. **(Bottom)** Spearman correlation coefficients between the metrics.

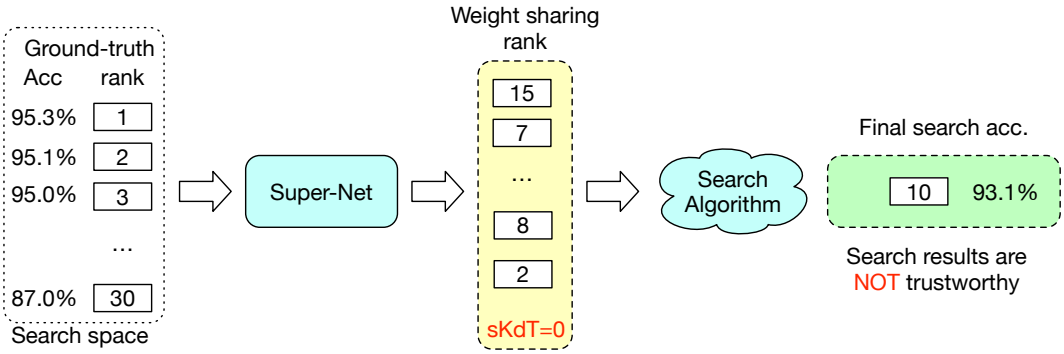
### 6.4.1 Evaluation of a super-net

The standalone performance of the architecture that is found by a NAS algorithm is clearly the most important metric to judge its merits. However, in practice, one cannot access this metric—we wouldn't need NAS if standalone performance was easy to query. Furthermore, stand-alone performance inevitably depends the sampling policy, and does not directly evaluate the quality of the super-net. Consequently, it is important to rely on metrics that are well correlated with the final performance but can be queried efficiently. To this end, we collect all our experiments and plot the pairwise correlation between final performance, sparse Kendall-Tau, and super-net accuracy. As shown in Figure 6.6, the super-net accuracy has a low correlation with the final performance on NASBench-101 and DARTS-NDS. Only on NASBench-201 does it reach a correlation of 0.52. The sparse Kendall-Tau yields a consistently higher correlation with the final performance. This is evidence that one should not focus too strongly on improving the super-net accuracy. While this metric remains computationally heavy, it serves as a middle ground that is feasible to evaluate in real-world applications.

In the following experiments, we thus mainly rely on sparse Kendall-Tau, and use final search performance as a reference only.




(a) Relying on final search accuracy only




(b) Using sparse Kendall Tau to diagnose super-net

Figure 6.7: **Comparing sparse Kendall-Tau and final search accuracy.** Here, we provide a toy example to illustrate why one cannot rely on the final search accuracy to evaluate the quality of the super-net. Let us consider a search space with only 30 architectures, whose accuracy ranges from 95.3% to 87% on the CIFAR-10 dataset, and we run a search algorithm on top. **(a)** describes a common scenario: we run the search for multiple times, yielding a best architecture with 93.1% accuracy. While this may seem good, it does not give any information about the quality of the search or the super-net. If we had full knowledge about the performance of every architecture in this space, we would see that this architecture is close to the average performance and hence no better than random. In **(b)**, the sparse Kendall-Tau allows us to diagnose this pathological case. A small sparse Kendall-Tau implies that there is a problem with super-net training.

Table 6.3: **Comparison of Kendall Tau (KdT) and Spearman ranking (SpR) with their sparse variants.**

| Corr. of Perf. | KdT. | S-KdT | SpR | S-SpR |
|---|---|---|---|---|
| NASBench-101 | 0.29 | 0.45 | 0.23 | 0.41 |
| NASBench-201 | 0.42 | 0.55 | 0.38 | 0.57 |
| DARTS-NDS | 0.08 | 0.19 | 0.09 | 0.20 |

#### Kendall Tau v.s. Spearman ranking correlation

Kendall-tau is not the only metric to evaluate the ranking correlation. Spearman ranking correlation is also widely adopted in this field (Guo et al., 2019; Dong & Yang, 2020). Note that our idea of sparsity also applies to SpR. In Table 6.3, we compare the performance of Kendall Tau(KdT), Spearman ranking correlation (SpR) and their sparse variants, in the same setting as Figure 6.6. Note that SpR and KdT performs similarly but that their sparse variants effectively improve the correlation on all search spaces.

#### Stand-alone Accuracy v.s. Sparse Kendall-Tau

A common misconception is that the super-net quality is well reflected by stand-alone accuracy of the final selected architecture. Neither sparse Kendall-Tau (sKdT) nor stand-alone accuracy are perfect. Both are tools to measure different aspects of a super-net.

Let us consider a completely new search space in which we have no prior knowledge about performance. As depicted by Figure 6.7, if we only rely on the stand-alone accuracy, the following situation might happen: Due to the lack of knowledge, the ranking of the super-net is purely random, and the search space accuracy is uniformly distributed. When trying different settings, there will be 1 configuration that 'outperforms' the others in terms of stand-alone accuracy. However, this configuration will be selected by pure chance. By only measuring stand-alone accuracy, it is technically impossible to realize that the ranking is random. By contrast, if one measures the sKdT (which is close to 0 in this example), an ill-conditioned super-net can easily be identified. In other words, purely relying on stand-alone accuracy could lead to pathological outcomes that can be avoided using sparse Kendall-Tau.

Additionally, stand-alone accuracy is related to both the super-net and the search algorithm. sparse Kendall-Tau allows us to judge super-net accuracy independently from the search algorithm. As an example, consider the use of a reinforcement learning algorithm, instead of random sampling, on top of the super-net. When observing a poor stand-alone accuracy, one cannot conclude if the problem is due to a poor super-net or to a poor performance of the RL algorithm. Prior to our work, people relied on the super-net accuracy to analyze the super-net quality. This is not a reliable metric, as shown in Figure 6.6. We believe that sparse Kendall-Tau is a better alternative.

### 6.4.2 Weight-sharing Protocol $P_{ws}$ – Hyperparameters

**Batch normalization in the super-net**

Batch normalization (BN) is commonly used in standalone networks to allow for faster and more stable training. It is thus also employed in most CNN search spaces. However, BN behaves differently in the context of WS-NAS, and special care has to be taken when using it. In a standalone network (c.f. Figure 6.8 *(Top)*), a BN layer during training computes the batch statistics $\mu_B$ and $\sigma_B$, normalizes the activations $f_A(x)$ as $(f_A(x) - \mu_B)/\sigma_B$, and finally updates the population statistics using a moving average. For instance, the mean statistics is updated as $\hat{\mu} \leftarrow \gamma\hat{\mu} + (1-\gamma)\mu_B$. At test time, the stored population statistics are used to normalize the feature map. In the standalone setting, both batch and population statistics are unbiased estimators of the population distribution $\mathcal{N}(\mu, \sigma)$.

By contrast, when training a super-net (Figure 6.8 *(Bottom)*) the population statistics that are computed based on the running average are not unbiased estimators of the population distribution, because the effective architecture before the BN layer varies in each epoch. More formally, let $f_{A_i}$ denote the $i$-th architecture. During training, the batch statistics are computed as $\mu_B^i = \sum_j f_{A_i}(x_j)/m$, and the output feature follows the distribution $\mathcal{N}(\mu_B^i, \sigma_B^i)$, where the superscript $i$ indicates that the current batch statistics depends on $A_i$ only. The population mean statistics is then updated as $\hat{\mu} \leftarrow \gamma\hat{\mu} + (1-\gamma)\mu_B^i$. However, during training, different architecture from the super-net are sampled. Therefore, the population mean statistics essentially becomes a weighted combination of means from different architectures, i.e., $\hat{\mu} \leftarrow \sum \alpha_i \mu_B^i = \sum \alpha_i f_{A_i}(x)$, where $\alpha_i$ is the sampling frequency of the $i$-th architecture. When evaluating a specific architecture $A_i$ at test time, the estimated population statistics thus depend on the other architectures in the super-net. This leads to a train-test discrepancy. One solution to mitigate this problem is to re-calibrate the batch statistics by recomputing the statistics on the entire training set before the the final evaluation (Yu & Huang, 2019). While the cost of doing so is negligible for a standalone network, NAS algorithms typically sample $\sim 10^5$ architectures for evaluation, which makes this approach intractable.

In contrast to Dong & Yang (2020) and Bender et al. (2020) that use the training mode also during testing, we formalize a simple, yet effective, approach to tackle the train-test discrepancy of BN in super-net training: we leave the normalization based on batch statistics during training unchanged, but use batch statistics also during testing. Since super-net evaluation is always conducted over a complete dataset, we are free to perform inference in mini-batches of the same size as the ones used during training. This allows us to compute the batch statistics on the fly in the exact same way as during training.

Figure 6.9 compares standard BN to our proposed modification. Using the tracked population statistics leads to many architectures with an accuracy around 10%, i.e., performing no better than random guessing. Our proposed modification allows us to significantly increase the fraction of high-performing architectures. Our results also show that the choice of fixing vs. learning an

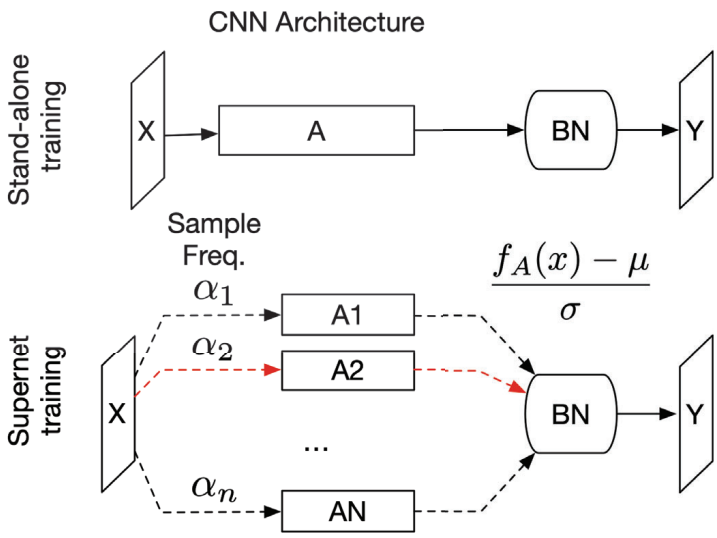



Figure 6.8: **Batch normalization in standalone and super-net training.**

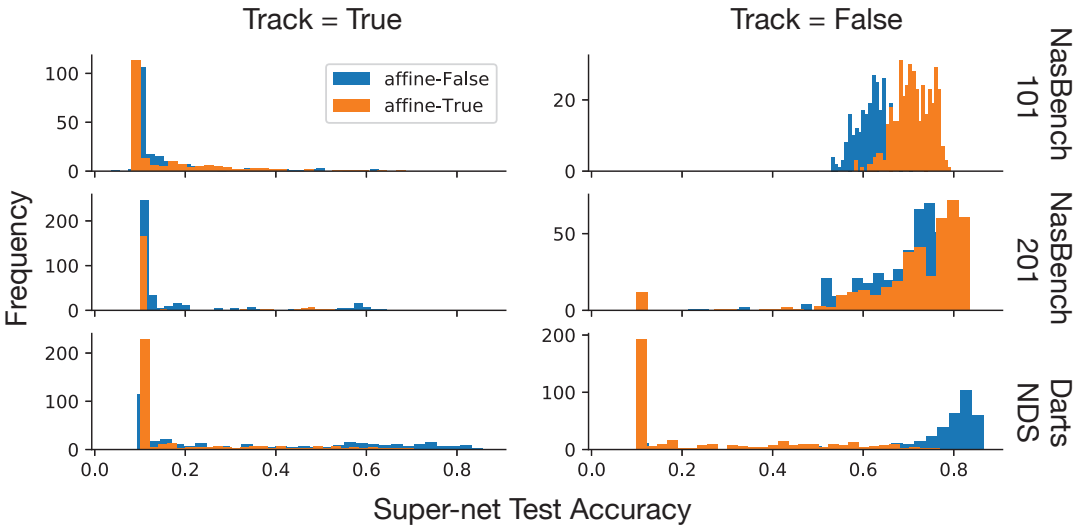



Figure 6.9: **Validation of BN.** We plot histograms of the super-net accuracy for different hyper-parameter settings. Tracking statistics **(left)** leads to many architectures with random performance. Without tracking **(right)**, learning the affine parameters (*affine-true*) increases accuracy on NASBench-101 and NASBench-201, but strongly decreases it for DARTS-NDS.

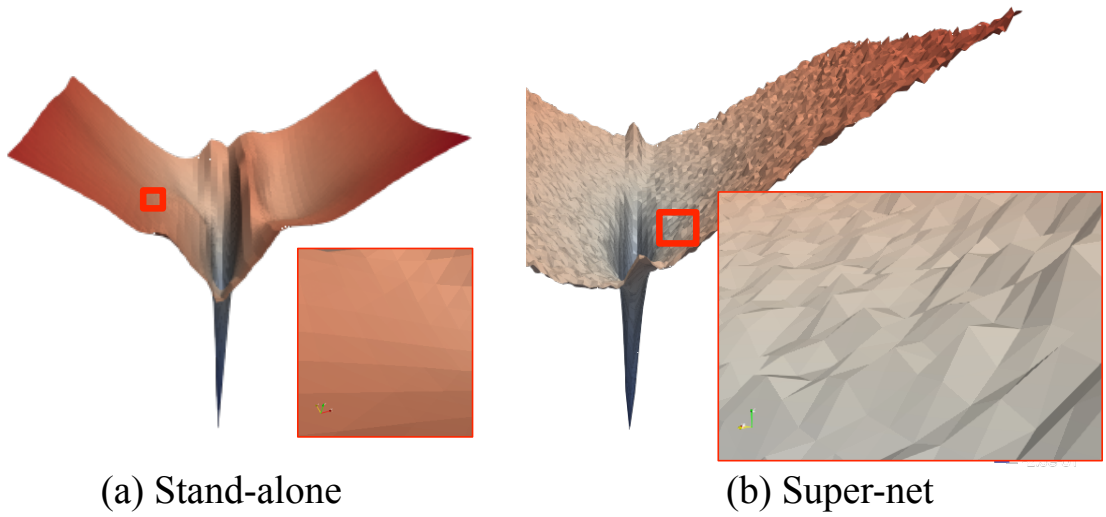



Figure 6.10: **Loss landscapes of a standalone network vs the super-net** (Sampling $n = 300$ architectures, better see in color).

affine transformation in batch normalization should match the standalone protocol $P_{proxy}$.

### Learning rate

The training loss of the super-net encompasses the task losses of all possible architectures. We suspect that the training difficulty increases with the number of architectures represented by the super-net. To better study this, we visualize the loss landscape (Li et al., 2018) of the standalone network and a super-net with $n = 300$ architectures. Concretely, the landscape is computed over the super-net training loss under the single-path one-shot sampling method, i.e.,

$$\mathscr{L}_s(x, \theta_s) = \sum_i \mathscr{L}_s(x, \theta_i), \quad \text{where } \forall i, \cup_i \theta_i = \theta_s. \tag{6.1}$$

Figure 6.10 shows that the loss landscape of the super-net is less smooth than that of a standalone architecture, which confirms our intuition. A smoother landscape indicates that optimization will converge more easily to a good local optimum. With a smooth landscape, one can thus use a relatively large learning rate. By contrast, a less smooth landscape requires using a smaller one.

Our experiments further confirm this observation. In the standalone protocol $P_{proxy}$, the learning rate is set to 0.2 for NASBench-101, and to 0.1 for NASBench-201 and DARTS-NDS, respectively. All protocols use a cosine learning rate decay. Figure 6.11 shows that super-net training requires lower learning rates than standalone training. The same trend is shown for other search spaces in Section 6.4.6 Table 6.10. We set the learning rate to 0.025 to be consistent across the three search spaces.

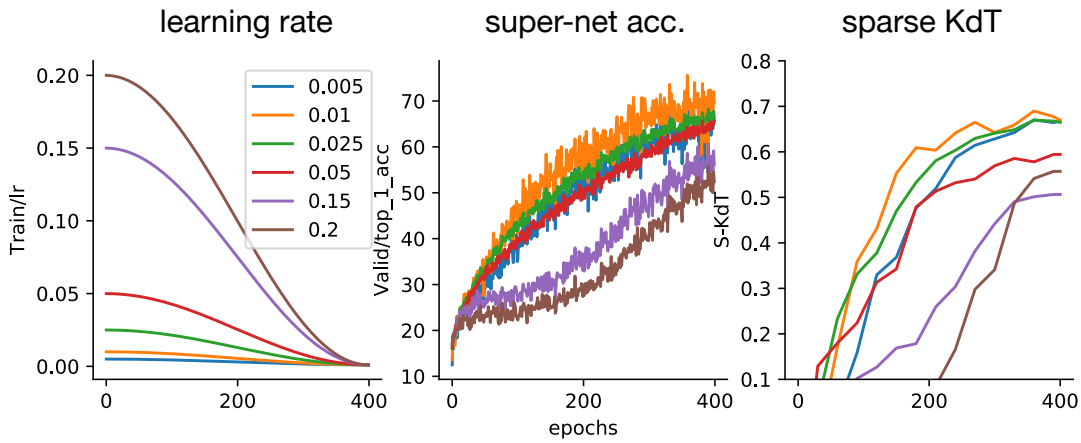



Figure 6.11: **Learning rate on NASBench-201.**

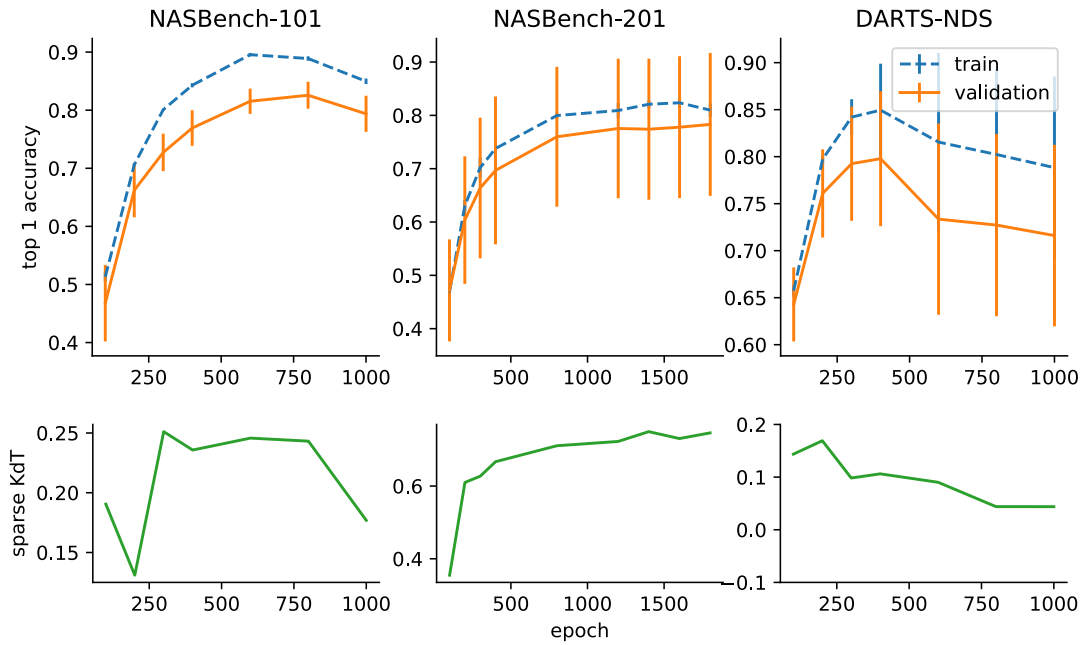



Figure 6.12: **Validating the number of epochs.** Each data point summarizes 3 individual runs.

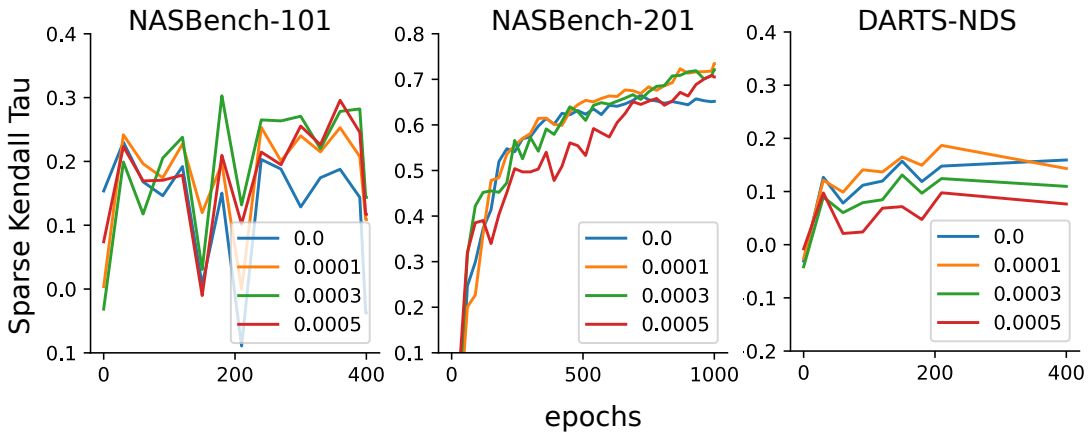



Figure 6.13: **Weight decay validation.**

#### Number of epochs

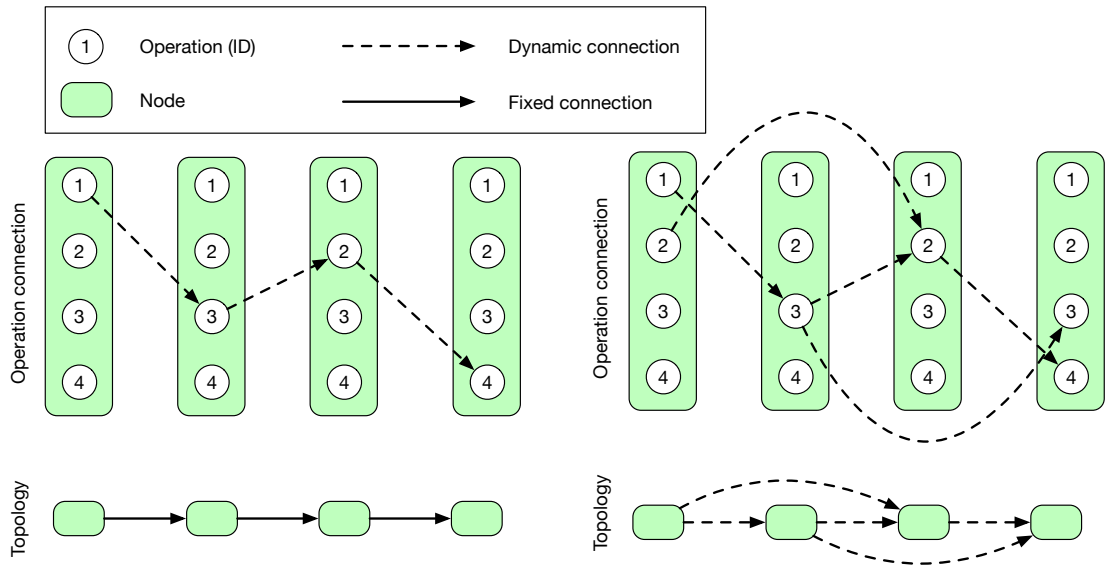
Since the cosine learning rate schedule decays the learning rate to zero towards the end of training, we evaluate the impact of the number of training epochs. In stand-alone training, the number of epochs was set to 108 for NASBench-101, 200 for NASBench-201, and 100 for DARTS-NDS. Figure 6.12 shows that increasing the number of epochs significantly improves the accuracy in the beginning, but eventually decreases the accuracy for NASBench-101 and DARTS-NDS. Interestingly, the number of epochs impacts neither the correlation of the ranking nor the final selected model performance after 400 epochs. We thus use 400 epochs for the remaining experiments.

#### Weight decay

Weight decay is used to reduce overfitting. For WS-NAS, however, overfitting does not occur because there are billions of architectures sharing the same set of parameters, which in fact rather causes underfitting. Based on this observation, (Nayman et al., 2019) propose to disable weight decay during super-net training. Figure 6.13, however, shows that the behavior of weight decay varies across datasets. While on DARTS-NDS weight decay is indeed harmful, it improves the results on NASBench 101 and 201. We conjecture that this is due to the much larger number of architectures in DARTS-NDS (243 billion) than in the NASBench series (less than 500,000).

### 6.4.3 Weight-sharing Protocol $P_{ws}$ – Sampling

Aside from the Random-NAS described in Section 6.3.1, we additionally include two variants of Random-NAS: 1) As pointed out by (Ying et al., 2019), two super-net architectures might be topologically equivalent in the stand-alone network by simply swapping operations. We thus include architecture-aware random sampling that ensures equal probability for unique architectures in Chapter 5. We name this variant Random-A; 2) We evaluate a variant called FairNAS (Chu et al., 2019), which ensures that each operation is selected with equal probability




(a) Search space with fixed topology

(b) Search space with dynamic topology

Figure 6.14: **Comparison between fixed and dynamic topology search spaces.**

during super-net training. Although FairNAS was designed for a search space where only operations are searched but not the topology, we adapt it to our setting.

**Adaptation of FairNAS.** Originally, FairNAS (Chu et al., 2019) was proposed in a search space with a fixed sequential topology, as depicted by Figure 6.14 *(a)*, where every node is sequentially connected to the previous one, and only the operations on the edges are subject to change. However, our benchmark search spaces exploit a more complex dynamic topology, as illustrated in Figure 6.14 *(b)*, where one node can connect to one or more previous nodes. Before generalizing to a dynamic topology search space, we simplify the original approach into a 2-node scenario: for each input batch, FairNAS will first randomly generate a sequence of all $o$ possible operations. It then samples one operation at a time, computes gradients for the fixed input batch, and accumulates the gradients across the operations. Once all operations have been sampled, the super-net parameters are updated with the average gradients. This ensures that all possible paths are equally exploited . With this simplification, FairNAS can be applied regardless of the topology. For a sequential-topology search space, we repeat the 2-node policy for every consecutive node pair. Naturally, for a dynamic topology space, FairNAS can be adopted in a similar manner, i.e., one first samples a topology, then applies the 2-node strategy for all connected node pairs. Note that adapting FairNAS increases the training time by a factor $o$.

**Results.** With the hyper-parameters fixed, we now compare three path-sampling techniques. Since DARTS-NDS does not contain enough samples trained in a stand-alone manner, we only report results on NASBench-101 and 201. In Figure 6.15, we show the sampling distributions of different approaches and the impact on the super-net in terms of sparse Kendall-Tau. These experiments reveal that, on NASBench-101, uniformly randomly sampling one architecture, as

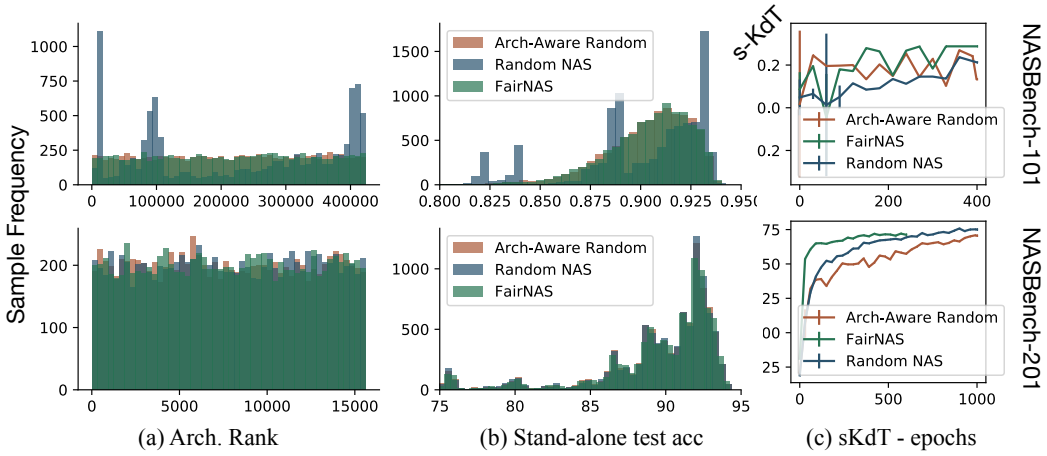



Figure 6.15: **Path sampling comparison on NASBench-101 (a) and NASBench-201 (b).** We sampled 10,000 architectures using different samplers and plot histograms of the architecture rank and the stand-alone test accuracy. We plot the s-KdT across the epochs. Results averaged across 3 runs.

in Chapter 5, is strongly biased in terms of accuracy and ranking. This can be observed from the peaks around rank 0, 100,000, and 400,000. The reason is that a single architecture can have multiple encodings, and uniform sampling thus oversamples such architectures with equivalent encodings. FairNAS samples architectures more evenly and yields consistently better sparse Kendall-Tau values, albeit by a small margin.

On NASBench-201, the three sampling policies have a similar coverage. This is because, in NASBench-201, topologically-equivalent encodings were not pruned. In this case, Random-NAS performs better than in NASBench-101, and FairNAS yields good early performance but quickly saturates. In short, using different sampling strategies might in general be beneficial, but we advocate for FairNAS in the presence of a limited training budget.

### 6.4.4 Weight-sharing Mapping $f_{ws}$ – Lower Fidelity Estimates lower the ranking correlation

Reducing memory foot-print and training time by proposing smaller super-nets has been an active research direction, and the resulting super-nets are referred to as *lower fidelity estimates* Elsken et al. (2019). The impact of this approach on the super-net quality, however, has never been studied systematically over multiple search spaces. We compare four popular strategies in Table 6.4. We deliberately prolong the training epochs inversely proportionally to the computational budget that would be saved by the low-fidelity estimates. For example, if the number of channels is reduced by half, we train the model for two times more epochs. Note that this provides an upper bound to the performance of low-fidelity estimates.

A commonly-used approach to reduce memory requirements is to decrease the batch size (Yang

Table 6.4: **Low fidelity estimates under same computational budget**, reporting final search model accuracy (FSA) and sparse Kendall-Tau (S-KdT) on NASBench-201.

| Metrics | Settings | | |
|---|---|---|---|
| Repeated cells | 3 | 2 | 1 |
| S-KdT | 0.751 ± 0.09 | 0.692 ± 0.18 | 0.502 ± 0.21 |
| FSA | 91.91 ± 0.09 | 91.95 ± 0.10 | 90.30 ± 0.71 |
| Init Channel | 16 | 8 | 4 |
| S-KdT | 0.740 ± 0.07 | 0.677 ± 0.10 | 0.691 ± 0.15 |
| FSA | 92.92 ± 0.48 | 92.32 ± 0.37 | 92.79 ± 0.85 |
| Batch-size | 256 | 128 | 64 |
| S-KdT | 0.740 ± 0.07 | 0.728 ± 0.16 | 0.703 ± 0.16 |
| FSA | 92.92 ± 0.48 | 92.37 ± 0.61 | 92.35 ± 0.34 |
| Train portion | 0.75 | 0.5 | 0.25 |
| S-KdT | 0.751 ± 0.11 | 0.742 ± 0.12 | 0.693 ± 0.13 |
| FSA | 92.13 ± 0.51 | 92.74 ± 0.43 | 91.47 ± 0.81 |

et al., 2020). Surprisingly, lowering the batch size from 256 to 64 has limited impact on the accuracy, but decreases sparse Kendall-Tau and the final searched model's performance, the most important metric in practice.

Another approach is to decrease the number of channels in the first layer (Liu et al., 2019b). This reduces the total number of parameters proportionally, since the number of channels in consecutive layers depends on the first one. Table 6.4 shows that this decreases the sparse Kendall-Tau from 0.7 to 0.5. By contrast, reducing the number of repeated cells (Pham et al., 2018b; Chu et al., 2019) by one has little impact. Hence, to train a good super-net, one should avoid changes between $f_{ws}$ and $f_{proxy}$, but one can reduce the batch size by a factor > 0.5 and use only one repeated cell.

The last lower-fidelity factor is the portion of training data that is used (Liu et al., 2019b; Xu et al., 2020). Surprisingly, reducing the training portion only marginally decreases the sparse Kendall-Tau for all three search spaces. On NASBench-201, keeping only 25% of the CIFAR-10 dataset results in a 0.1 drop in sparse Kendall-Tau. This explains why DARTS-based methods typically use only 50% of the data to train the super-net but can still produce reasonable results.

Table 6.5: **Dynamic channels on NASBench-101.**

| Type | Accuracy | S-KdT | P > R | Stand-alone Acc. |
|---|---|---|---|---|
| Fixed | 71.52 ± 6.94 | 0.22 | 0.546 | 91.79 ± 1.72 |
| Shuffle | 31.79 ± 10.90 | 0.17 | 0.391 | 90.58 ± 1.58 |
| Interpolate | 57.53 ± 10.05 | 0.37 | 0.865 | 93.35 ± 3.27 |
| Baseline | 76.91 ± 10.05 | 0.22 | 0.865 | 89.43 ± 4.30 |
| Baseline-v2 | 75.18 ± 9.28 | 0.33 | 0.891 | 91.27 ± 1.18 |
| Ours | **76.95** ± 8.29 | **0.46** | **0.949** | **93.65** ± 0.73 |

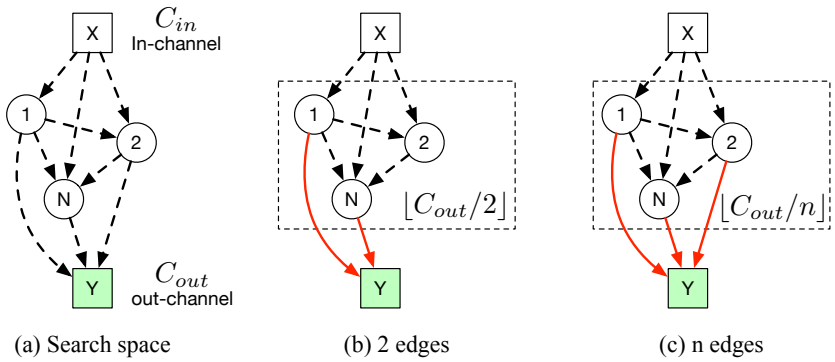
Table 6.6: A fair comparison between the baseline dynamic channeling with randomly sampling sub-spaces and our disable dynamic channeling approach.

| Edges | Accuracy | S-KdT | P > R | Stand-alone Acc. |
|---|---|---|---|---|
| *Baseline: random sampling sub-spaces with dynamic channeling.* | | | | |
| 1 | 70.04± 8.15 | 0.173 | 0.797 | 91.19±2.01 |
| 2 | 78.29±10.51 | 0.206 | 0.734 | 82.03±1.50 |
| 3 | 79.92± 9.42 | 0.242 | 0.576 | 92.20±1.19 |
| 4 | 79.37± 17.34 | 0.270 | 0.793 | 92.32±1.10 |
| Average | 76.905 ± 10.05 | 0.223 | 0.865 | 89.435 ± 4.30 |
| *Disable dynamic channels by fixing the edges to the output node.* | | | | |
| 1 | 76.92± 7.87 | 0.435 | 0.991 | 93.94±0.22 |
| 2 | 74.32± 8.21 | 0.426 | 0.925 | 93.34±0.01 |
| 3 | 77.24± 9.18 | 0.487 | 0.901 | 93.66±0.07 |
| 4 | 79.31± 7.04 | 0.493 | 0.978 | 93.65±0.07 |
| Average | 76.95 ± 8.29 | 0.460 | 0.949 | 93.65 ± 0.73 |

### 6.4.5 Weight-sharing Mapping $f_{ws}$ - Implementation

**Dynamic channeling hurts super-net quality**

Dynamic channeling is an implicit factor in many search spaces (Ying et al., 2019; Cai et al., 2018b; Guo et al., 2019; Dong & Yang, 2019b). It refers to the fact that the number of channels of the intermediate layers depends on the number of incoming edges to the output node. This is depicted by Figure 6.16 *(a)*: for a search cell with $n$ intermediate nodes, where $X$ and $Y$ are the input and output node with $C_{in}$ and $C_{out}$ channels, respectively. When there are $n = 2$ edges (c.f. Figure 6.16 *(b)*), the associated channel numbers decrease so that their sum equals $C_{out}$. That is, the intermediate nodes have $\lfloor C_{out}/2 \rfloor$ channels. In the general case, shown in Figure 6.16 *(c)*, the number of channels in intermediate nodes is thus $\lfloor C_{out}/n \rfloor$ for $n$ incoming edges. A weight sharing approach has to cope with this architecture-dependent fluctuation of the number of channels during training.




(a) Search space (b) 2 edges (c) n edges

Figure 6.16: **NASBench-101 dynamic channel.**

Let $C$ denote the number of channels of a given architecture, and $C_{max}$ the maximum number of channels for a node across the entire search space. All existing approaches allocate $C_{max}$ channels and, during training, extract a subset of these channels. The existing methods then differ in how they extract the channels: (Guo et al., 2019) use a fixed chunk of channels, e.g., $[0:C]$; (Zhang et al., 2018) randomly shuffle the channels before extracting a fixed chunk; and (Dong & Yang, 2019a) linearly interpolate the $C_{max}$ channels into $C$ channels using a moving average across neighboring channels.

Instead of sharing the channels between architectures, we propose to disable dynamic channelling completely. As the channel number only depends on the incoming edges, we separate the search space into a discrete number of sub-spaces, each with a fixed number of incoming edges. As shown in Table 6.5, disabling dynamic channeling improves the sparse Kendall-Tau and the final search performance by a large margin and yields a new state of the art on NASBench101.

**Random subspace baseline.** Since each sub-space now encompasses fewer architectures, it is not fair to perform a comparison with the full NASBench 101 search space. Therefore, for each sub-space, we construct a baseline space where we drop architectures uniformly at random until the number of remaining architectures matches the size of the sub-space. We repeat this process with 3 different initializations, while keeping all other factors unchanged when training the super-net. We refer to this as 'Baseline' in Table 6.5. We also provide additional results in Table 6.6 for each individual sub-space and show that the sparse Kendall-Tau remains similar to that of the baseline using the full search space, which clearly evidences the effectiveness of our approach to disable the dynamic channeling.

Furthermore, we compose another baseline, where we enable dynamic channeling during super-net training. During validation, we compute the average sparse Kendall-Tau of each sub-space, where we sample 200 architectures that share the same number of channels. We call this baseline-v2. In Table 6.5, we can see that this surpasses the original baseline by a significant margin. It further evidences the importance of disabling dynamic channels. Nonetheless, the best is to disable dynamic channeling during both the training and the validation phase.

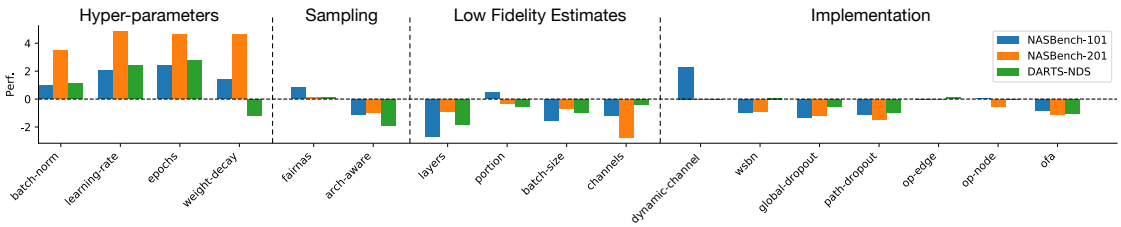



Figure 6.17: **Influence of factors on the final model.** We plot the difference in percent between the searched model's performance with and without applying the corresponding factor. For the hyper-parameters of $P_{ws}$, the baseline is Random NAS, as reported in Table 6.9. For the other factors, the baseline of each search space uses the best setting of the hyper-parameters. Each experiment was run at least 3 times.

### WS on Edges or Nodes?

Most existing works build $f_{ws}$ to define the shared operations on the graph nodes rather than on the edges. This is because, if $f_{ws}$ maps to the edges, the parameter size increases from $O(n)$ to $O(n^2)$, where $n$ is the number of intermediate nodes. We provide a concrete example in Figure 6.18. However, the high sparse Kendall-Tau on NASBench-201 in the top part of Table 6.7, which is obtained by mapping to the edges, may suggest that sharing on the edges is beneficial. Here we investigate if this is truly the case.

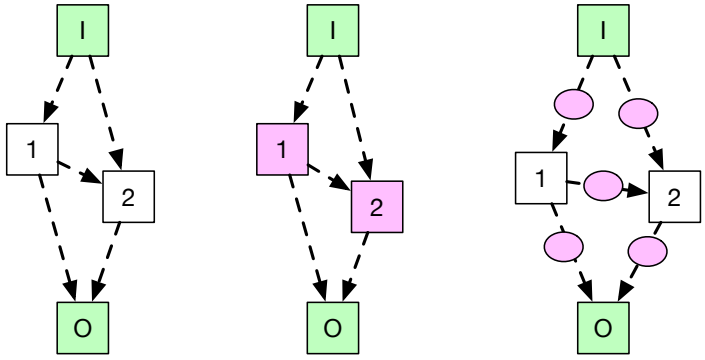
On NASBench-101, by design, each node merges the previous nodes' outputs and then applies parametric operations. This makes it impossible to build an equivalent sharing on the edges. We therefore construct sharing on the edges for DARTS-NDS and sharing on the nodes for NASBench-201. As shown in Table 6.8, for both spaces, sharing on the edges yields a marginally better super-net than sharing on the nodes. Such small differences might be due to the fact that, in both spaces, the number of nodes is 4, while the number of edges is 6, thus mapping to edges will not drastically affect the number of parameters. Nevertheless, this indicates that one should consider having a larger number of shared weights when the resources are not a bottleneck.

### Other mapping factors

We evaluate the weight-sharing batch normalization (WSBN) of (Luo et al., 2018b) , which keeps an independent set of parameters for each incoming edge. Furthermore, we test the two commonly-used dropout strategies: right before global pooling (global dropout); and at all edge connections between the nodes (path dropout). Note that path dropout has been widely used in WS-NAS (Luo et al., 2018a; Liu et al., 2019b; Pham et al., 2018b). For both dropout strategies, we set the dropout rate to 0.2. Finally, we evaluate the super convolution layer of (Cai et al., 2020), referred to as OFA kernel, which accounts for the fact that, in CNN search spaces, convolution operations appear as groups, and thus merges the convolutions within the same group, keeping only the largest kernel parameters and performing a parametric projection to obtain the other kernels. The results in Table 6.8 show that all these factors negatively impact the search performances and the super-net quality.

Table 6.7: **Comparison of operations on the nodes or on the edges.** We report sKT / final search performance.

| | NASBench-101 | NASBench-201 | DARTS-NDS |
|---|---|---|---|
| Baseline | 0.236 / 92.32 | 0.740 / 92.92 | 0.159 / 93.59 |
| Op-Edge | N/A | as Baseline | 0.189 / 93.97 |
| Op-Node | as Baseline | 0.738 / 92.36 | as Baseline |




(a) Search space (b) Op on the Node (c) Op on the Edge

Figure 6.18: **Operation on the node or edge? (a)** Consider a search space with 2 intermediate nodes, 1, 2, with one input (I) and output (O) node. This yields 5 edges. Let us assume that we have 4 possible operations to choose from, as indicated as the purple color code. **(b)** When the operations are on the nodes, there are 2 × 4 ops to share, i.e., I→2 and 1→2 share weights on node 2. **(c)** If the operations are on the edges, then we have 5 × 4 ops to share.

Table 6.8: **Comparison of different mappings** $f_{ws}$. We report s-KdT / final search performance.

| | NASBench-101 | NASBench-201 | DARTS-NDS |
|---|---|---|---|
| Baseline | 0.236 / 92.32 | 0.740 / 92.92 | 0.159 / 93.59 |
| WSBN | 0.056 / 91.33 | 0.675 / 92.04 | 0.331 / 92.95 |
| Global-Dropout | 0.179 / 90.95 | 0.676 / 91.76 | 0.102 / 92.30 |
| Path-Dropout | 0.128 / 91.19 | 0.431 / 91.42 | 0.090 / 91.90 |
| OFA Kernel | 0.132 / 92.01 | 0.574 / 91.83 | 0.112 / 92.83 |

Table 6.9: **Final results.** Results on NASBench-101 and 201 are from Chapter 5, and Dong & Yang (2020). We report the mean over 3 runs. Note that NASBench-101 ($n = 7$) in Chapter 5 is identical to our setting. Our new strategy significantly surpasses the random search baseline.

| Method | NASBench 101 (n=7) | NASBench 201 | DARTS NDS | DARTS NDS$^\star$ |
|---|---|---|---|---|
| ENAS Pham et al. (2018b) | 91.83 ± 0.42 | 54.30 ± 0.00 | 94.45 ± 0.09 | 97.11 |
| DARTS-V2 Liu et al. (2019b) | 92.21 ± 0.61 | 54.30 ± 0.00 | 94.79 ± 0.11 | 97.37 |
| NAO Luo et al. (2018a) | 92.59 ± 0.59 | - | - | 97.10 |
| GDAS Dong & Yang (2019b) | - | 93.51 ± 0.13 | - | 96.23 |
| Random NAS Li & Talwalkar (2019) | 89.89 ± 3.89 | 87.66 ± 1.69 | 91.33 ± 0.12 | 96.74$^\dagger$ |
| Random NAS (Ours) | 93.12 ± 0.06 | 92.71 ± 0.15 | 94.26 ± 0.05 | 97.08 |

$^\dagger$Results from Li & Talwalkar (2019)
$^\star$Trained according to Liu et al. (2019b) for 600 epochs.
On NASBench-201, both random NAS and our approach samples 100 final architectures to follow the setting of Dong & Yang (2020)

### 6.4.6 Results for All Factors

We report the numerical results for all hyper-parameter factors in Table 6.10, low-fidelity factors in Table 6.11 and implementation factors in Table 6.12. These results were computed from the last epochs of 3 different runs.

## 6.5 How should you train your super-net?

Figure 6.17 summarizes the influence of all tested factors on the final performance. It stands out that properly tuned hyper-parameters lead to the biggest improvements by far. Surprisingly, most other factors and techniques either have a hardly measurable effect or in some cases even lead to worse performance. Based on these findings, here is how you should train your super-net:

1. Do not use super-net accuracy to judge the quality of your super-net. The sparse Kendall-Tau has much higher correlation with the final search performance.
2. When batch normalization is used, do not use the moving average statistics during evaluation.

Table 6.10: **Results for all WS Protocol $P_{ws}$ factors on the three search spaces.**

| Factor and settings | NASBench-101 Super-net Accuracy | NASBench-101 S-KdT | NASBench-101 P > R | NASBench-101 Final Performance | NASBench-201 Super-net Accuracy | NASBench-201 S-KdT | NASBench-201 P > R | NASBench-201 Final Performance | DARTS-NDS Super-net Accuracy | DARTS-NDS S-KdT | DARTS-NDS P > R | DARTS-NDS Final Performance |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Batch-norm.** | | | | | | | | | | | | |
| affine F track F | 0.651±0.05 | 0.161 | 0.996 | 0.916±0.13 | 0.660±0.13 | 0.783 | 0.997 | 92.67±1.21 | 0.735±0.18 | 0.056 | 0.224 | 93.14±0.28 |
| affine T track F | 0.710±0.04 | 0.240 | 0.996 | 0.924±0.01 | 0.713±0.14 | 0.718 | 0.707 | 91.71±1.05 | 0.265±0.21 | -0.071 | 0.213 | 91.89±2.01 |
| affine F track T | 0.144±0.09 | 0.084 | 0.112 | 0.882±0.02 | 0.182±0.15 | -0.171 | 0.583 | 86.41±4.84 | 0.359±0.25 | -0.078 | 0.023 | 90.33±0.76 |
| affine T track T | 0.153±0.10 | -0.008 | 0.229 | 0.905±0.01 | 0.134±0.09 | -0.417 | 0.274 | 90.77±0.40 | 0.216±0.18 | -0.050 | 0.109 | 90.49±0.32 |
| **Learning rate.** | | | | | | | | | | | | |
| 0.005 | 0.627±0.07 | 0.091 | 0.326 | 0.908±0.01 | 0.658±0.11 | 0.668 | 0.141 | 90.14±0.55 | 0.792±0.08 | 0.130 | 0.033 | 91.81±0.68 |
| 0.01 | 0.668±0.06 | 0.095 | 0.546 | 0.919±0.00 | 0.713±0.12 | 0.670 | 0.711 | 91.21±1.18 | 0.727±0.05 | 0.131 | 0.258 | 92.86±0.64 |
| 0.025 | 0.715±0.05 | 0.220 | 0.910 | 0.917±0.01 | 0.659±0.13 | 0.665 | 0.844 | 92.42±0.58 | 0.656±0.14 | 0.218 | 0.299 | 93.42±0.20 |
| 0.05 | 0.727±0.05 | 0.143 | 0.905 | 0.911±0.02 | 0.631±0.14 | 0.594 | 0.730 | 92.02±0.70 | 0.623±0.04 | 0.147 | 0.489 | 91.70±0.33 |
| 0.1 | 0.690±0.07 | 0.005 | 0.905 | 0.909±0.02 | 0.609±0.28 | 0.571 | 0.618 | 91.82±0.81 | 0.735±0.06 | 0.096 | 0.099 | 92.73±0.24 |
| 0.15 | 0.000±0.00 | -0.274 | N/A | N/A | 0.551±0.14 | 0.506 | 0.553 | 91.22±1.20 | 0.371±0.27 | 0.027 | 0.218 | 91.20±0.72 |
| 0.2 | - | - | - | - | 0.519±0.12 | 0.557 | 0.035 | 88.74±0.11 | 0.102±0.48 | -0.366 | N/A | N/A |
| **Epochs.** | | | | | | | | | | | | |
| 100 | 0.468±0.07 | 0.190 | 0.759 | 0.920±0.01 | 0.472±0.09 | 0.355 | 0.997 | 92.11±1.67 | 0.643±0.04 | 0.144 | 0.901 | 93.90±0.49 |
| 200 | 0.662±0.05 | 0.131 | 0.685 | 0.914±0.01 | 0.604±0.12 | 0.610 | 0.881 | 91.88±2.01 | 0.761±0.05 | 0.169 | 0.778 | 94.08±0.21 |
| 300 | 0.727±0.03 | 0.251 | 0.739 | 0.920±0.01 | 0.664±0.13 | 0.627 | 0.840 | 91.42±1.91 | 0.793±0.06 | 0.098 | 0.870 | 93.22±0.95 |
| 400 | 0.769±0.03 | 0.236 | 0.932 | 0.921±0.01 | 0.697±0.14 | 0.667 | 0.158 | 89.83±0.97 | 0.798±0.07 | 0.106 | 0.036 | 92.34±0.22 |
| 600 | 0.815±0.02 | 0.246 | 0.556 | 0.911±0.01 | 0.720±0.13 | 0.682 | 0.285 | 90.28±0.82 | 0.734±0.10 | 0.090 | 0.209 | 93.23±0.19 |
| 800 | 0.826±0.02 | 0.243 | 0.177 | 0.907±0.00 | 0.760±0.13 | 0.711 | 0.378 | 91.53±0.53 | 0.728±0.10 | 0.044 | 0.853 | 93.29±0.81 |
| 1000 | 0.794±0.03 | 0.177 | 0.831 | 0.920±0.01 | 0.782±0.13 | 0.740 | 0.589 | 92.92±0.48 | 0.717±0.09 | 0.044 | 0.997 | 93.92±0.90 |
| 1200 | - | - | - | - | 0.775±0.13 | 0.723 | 0.198 | 90.81±0.56 | - | - | - | - |
| 1400 | - | - | - | - | 0.774±0.13 | 0.750 | 0.604 | 92.26±0.33 | - | - | - | - |
| 1600 | - | - | - | - | 0.778±0.13 | 0.731 | 0.882 | 91.85±1.20 | - | - | - | - |
| 1800 | - | - | - | - | 0.783±0.13 | 0.746 | 0.266 | 90.64±0.82 | - | - | - | - |
| **Weight decay.** | | | | | | | | | | | | |
| 0.0 | 0.645±0.05 | -0.037 | 0.179 | 0.899±0.01 | 0.713±0.13 | 0.652 | 0.266 | 90.58±0.99 | 0.670±0.03 | 0.159 | 0.629 | 93.09±0.73 |
| 0.0001 | 0.719±0.03 | 0.109 | 0.659 | 0.912±0.01 | 0.756±0.13 | 0.734 | 0.612 | 91.88±0.59 | 0.751±0.05 | 0.143 | 0.396 | 93.37±0.44 |
| 0.0003 | 0.771±0.03 | 0.144 | 0.648 | 0.915±0.01 | 0.772±0.13 | 0.721 | 0.726 | 92.34±0.57 | 0.759±0.06 | 0.110 | 0.890 | 93.82±0.51 |
| 0.0005 | 0.782±0.03 | 0.117 | 0.910 | 0.911±0.02 | 0.764±0.13 | 0.705 | 0.882 | 92.61±0.59 | 0.739±0.07 | 0.077 | 0.051 | 91.61±1.01 |
| **Sampling.** | | | | | | | | | | | | |
| Random-A | 0.717±0.04 | 0.133 | 0.862 | 0.919±0.02 | 0.764±0.13 | 0.705 | 0.882 | 92.61±0.59 | - | - | - | - |
| Random-NAS | 0.638±0.20 | 0.167 | 0.949 | 0.913±0.02 | 0.765±0.14 | 0.750 | 0.897 | 92.17±1.01 | - | - | - | - |
| FairNAS | 0.789±0.03 | 0.288 | 0.382 | 0.908±0.01 | 0.774±0.14 | 0.713 | 0.917 | 93.06±0.31 | - | - | - | - |

Table 6.11: **Results for all low-fidelity factors on the three search spaces.**

| Factor and settings | NASBench-101 Super-net Accuracy | NASBench-101 S-KdT | NASBench-101 P > R | NASBench-101 Final Performance | NASBench-201 Super-net Accuracy | NASBench-201 S-KdT | NASBench-201 P > R | NASBench-201 Final Performance | DARTS-NDS Super-net Accuracy | DARTS-NDS S-KdT | DARTS-NDS P > R | DARTS-NDS Final Performance |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Number of Layer** (-X indicates the baseline minus X) | | | | | | | | | | | | |
| Baseline | 0.769±0.03 | 0.236 | 0.932 | 0.921±0.01 | 0.782±0.13 | 0.740 | 0.589 | 92.92±0.48 | 0.670±0.03 | 0.159 | 0.629 | 93.09±0.73 |
| -1 | 0.759±0.03 | 0.214 | 0.222 | 0.901±0.01 | 0.749±0.13 | 0.710 | 0.796 | 91.85±0.92 | 0.843±0.04 | 0.178 | 0.299 | 92.35±1.25 |
| -2 | 0.817±0.03 | 0.228 | 0.713 | 0.910±0.02 | 0.777±0.13 | 0.700 | 0.822 | 92.68±0.37 | 0.852±0.03 | 0.205 | 0.609 | 92.65±1.89 |
| **Train portion** | | | | | | | | | | | | |
| 0.25 | 0.433±0.07 | 0.216 | 0.281 | 0.901±0.01 | 0.660±0.11 | 0.668 | 0.979 | 92.30±1.14 | 0.597±0.14 | 0.132 | 0.359 | 92.27±1.84 |
| 0.5 | 0.612±0.06 | 0.251 | 0.424 | 0.896±0.02 | 0.740±0.12 | 0.669 | 0.979 | 93.17±0.47 | 0.666±0.17 | 0.083 | 0.551 | 92.22±1.36 |
| 0.75 | 0.688±0.05 | 0.222 | 0.857 | 0.920±0.01 | 0.758±0.13 | 0.725 | 0.618 | 92.46±0.19 | 0.715±0.18 | 0.096 | 0.081 | 92.29±0.47 |
| 0.9 | 0.722±0.05 | 0.186 | 0.996 | 0.931±0.01 | 0.772±0.13 | 0.721 | 0.726 | 92.34±0.57 | 0.703±0.18 | 0.042 | 0.065 | 92.78±0.10 |
| **Batch size** (/ X indicates the baseline divide by X) | | | | | | | | | | | | |
| Baseline | 0.769±0.03 | 0.236 | 0.932 | 0.921±0.01 | 0.782±0.13 | 0.740 | 0.589 | 92.92±0.48 | 0.670±0.03 | 0.159 | 0.629 | 93.09±0.73 |
| / 2 | 0.670±0.05 | 0.246 | 0.807 | 0.920±0.01 | 0.728±0.16 | 0.719 | 0.842 | 92.37±0.61 | 0.698±0.20 | 0.037 | 0.209 | 93.24±0.13 |
| / 4 | 0.686±0.07 | 0.155 | 0.913 | 0.921±0.01 | 0.703±0.16 | 0.679 | 0.672 | 92.35±0.34 | 0.633±0.20 | 0.033 | 0.690 | 93.68±0.62 |
| **# channel** (/ X indicates the baseline divide by X) | | | | | | | | | | | | |
| Baseline | 0.769±0.03 | 0.236 | 0.932 | 0.921±0.01 | 0.782±0.13 | 0.740 | 0.589 | 92.92±0.48 | 0.670±0.03 | 0.159 | 0.629 | 93.09±0.73 |
| / 2 | 0.658±0.05 | 0.156 | 0.704 | 0.898±0.02 | 0.697±0.14 | 0.667 | 0.158 | 89.83±0.97 | 0.776±0.05 | 0.190 | 0.993 | 93.90±0.71 |
| / 4 | 0.604±0.06 | 0.093 | 0.907 | 0.922±0.01 | 0.606±0.13 | 0.616 | 0.878 | 92.86±0.34 | 0.707±0.05 | 0.202 | 0.359 | 92.93±0.58 |

Instead, compute the statistics on the fly over a batch of the same size as used during training.

3. The loss landscape of super-nets is less smooth than that of standalone networks. Start from a

Table 6.12: **Results for all implementation factors on the three search spaces.**

| Factor and settings | NASBench-101 Super-net Accuracy | NASBench-101 S-KdT | NASBench-101 P > R | NASBench-101 Final Performance | NASBench-201 Super-net Accuracy | NASBench-201 S-KdT | NASBench-201 P > R | NASBench-201 Final Performance | DARTS-NDS Super-net Accuracy | DARTS-NDS S-KdT | DARTS-NDS P > R | DARTS-NDS Final Performance |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Other factors** | | | | | | | | | | | | |
| Baseline | 0.769±0.03 | 0.236 | 0.932 | 0.921±0.01 | 0.782±0.13 | 0.740 | 0.589 | 92.92±0.48 | 0.670±0.03 | 0.159 | 0.629 | 93.09±0.73 |
| OFA Kernel | 0.708±0.08 | 0.132 | 0.203 | 92.01±0.19 | 0.672±0.18 | 0.574 | 0.605 | 91.83 ± 0.86 | 0.782±0.05 | 0.112 | 0.399 | 93.22±0.43 |
| WSBN | 0.155±0.07 | 0.085 | 0.504 | 0.809±0.13 | 0.703±0.14 | 0.676 | 0.585 | 92.06±0.48 | 0.744±0.16 | 0.033 | 0.682 | 92.88±1.22 |
| **Path dropout rate** | | | | | | | | | | | | |
| Baseline | 0.769±0.03 | 0.236 | 0.932 | 0.921±0.01 | 0.782±0.13 | 0.740 | 0.589 | 92.92±0.48 | 0.670±0.03 | 0.159 | 0.629 | 93.09±0.73 |
| 0.05 | 0.750±0.02 | 0.206 | 0.819 | 0.915±0.07 | 0.490±0.09 | 0.712 | 0.881 | 92.25±0.89 | 0.184±0.06 | 0.006 | 0.359 | 92.93±0.60 |
| 0.15 | 0.726±0.02 | 0.186 | 0.482 | 0.910±0.01 | 0.250±0.03 | 0.640 | 0.526 | 91.44±1.25 | 0.366±0.05 | 0.059 | 0.570 | 92.61±1.28 |
| 0.2 | 0.669±0.01 | 0.110 | 0.282 | 0.901±0.01 | 0.185±0.02 | 0.431 | 0.809 | 92.15±0.85 | 0.518±0.06 | 0.090 | 0.009 | 91.45±0.58 |
| **Global dropout** | | | | | | | | | | | | |
| Baseline | 0.769±0.03 | 0.236 | 0.932 | 0.921±0.01 | 0.782±0.13 | 0.740 | 0.589 | 92.92±0.48 | 0.670±0.03 | 0.159 | 0.629 | 93.09±0.73 |
| 0.2 | 0.739±0.05 | 0.233 | 0.221 | 0.910±0.00 | 0.712±0.13 | 0.702 | 0.950 | 91.76±1.36 | 0.557±0.19 | 0.018 | 0.451 | 93.51±0.27 |

Please refer to Section 6.4.5 for mapping on the node or edge and Section 6.4.5 for dynamic channel factor results.

Table 6.13: **Parameter settings that obtained the best searched results.**

| Search Space | *implementation* | | | | | *low fidelity* | | | | *hyperparam.* | | | | *sampling* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Dynamic Conv | OFA Conv | WSBN | Dropout | Op map | # layer | portion | batch-size | # channels | batch-norm | learning rate | epochs | weight decay | |
| NASBench-101 | Interpolation | N | N | 0. | Node | 9 | 0.75 | 256 | 128 | Tr=F A=T | 0.025 | 400 | 1e-3 | FairNAS |
| NASBench-201 | Fix | N | N | 0. | Edge | 5 | 0.9 | 128 | 16 | Tr=F A=T | 0.025 | 1000 | 3e-3 | FairNAS |
| DARTS-NDS | Fix | N | Y | 0. | Edge | 12 | 0.9 | 256 | 36 | Tr=F A=F | 0.025 | 400 | 0 | FairNAS |

For batch-norm, we report Track statistics (Tr) and Affine (A) setting with True (T) or False (F).
For other notation, Y = Yes, N = No.

smaller learning rate than standalone training.

4. Do not use other low-fidelity estimates than moderately reducing the training set size to decrease the search time.
5. Do not use dynamic channeling in search spaces that have a varying number of channels in the intermediate nodes. Break the search space into multiple sub-spaces such that dynamic channeling is not required.

**Comparison to the state of the art.** Table 6.9 shows that carefully controlling the relevant factors and adopting the techniques proposed in Section 6.4 allow us to considerably improve the performance of Random-NAS. Thanks to our evaluation, we were able to show that simple Random-NAS together with an appropriate training protocol $P_{ws}$ and mapping function $f_{ws}$ yields results that are competitive to and sometimes even surpass state-of-the-art algorithms. Our results provide a strong baseline upon which future work can build.

We also report the best settings in Table 6.13.

# 7 Landmark Regularization

As empirically shown in the previous chapters, the ranking disorder between the performance of stand-alone architectures and that of the corresponding shared-weight networks violates the main assumption of weight-sharing NAS algorithms, thus limiting their effectiveness. In this chapter, we tackle this issue by proposing a regularization term that aims to maximize the correlation between the performance rankings of the shared-weight network and that of the standalone architectures using a small set of landmark architectures. We incorporate our regularization term into three different NAS algorithms and show that it consistently improves performance across algorithms, search-spaces, and tasks.

## 7.1 Introduction

Modern algorithms for neural architecture search (NAS) can now find architectures that outperform the human-designed ones for many computer vision tasks (Liu et al., 2019a; Wu et al., 2018; Chen et al., 2019b; Ryoo et al., 2020). A driving factor behind this progress was the introduction of parameter sharing (Pham et al., 2018b), which reduces the search time from thousands of GPU hours to just a few and has thus become the backbone of most state-of-the-art NAS frameworks (Bender et al., 2020; Yu et al., 2020a; Cai et al., 2020; Luo et al., 2020). At the heart of all these methods is a shared network, a.k.a. super-net, that encompasses all architectures within the search space.

To train the super-net, NAS algorithms essentially sample individual architectures from the super-net and train them for one or a few steps. The sampling can be done explicitly, with strategies such as reinforcement learning (Pham et al., 2018b; Cai et al., 2018b), evolutionary algorithms (Guo et al., 2019; Wang et al., 2020), or random sampling (Li & Talwalkar, 2019) and Chapter 5, or implicitly, by relying on a differentiable parameterization of the architecture space (Li et al., 2020a; Liu et al., 2019b; Cai et al., 2020; Wu et al., 2018; Zela et al., 2020a). Whether explicit or implicit, the underlying assumption of these methods is that the relative performance of the individual architectures in the super-net is highly correlated with the performance of the same

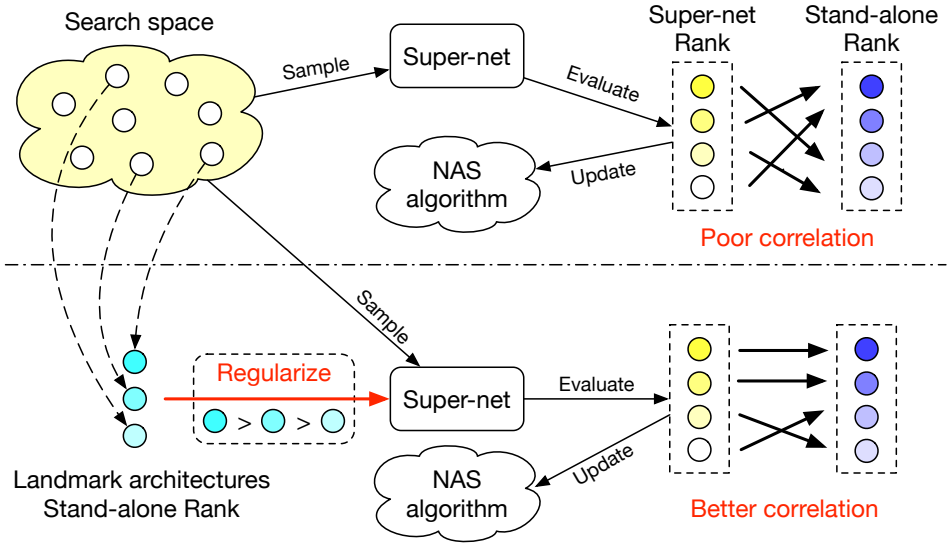



Figure 7.1: **Landmark regularization.** Traditional super-net training leads to poor correlation between relative stand-alone performance and super-net performance (top). We sample landmark architectures and use their relative performance to guide training towards an improved ranking and show that this improves the search performance (bottom).

architectures when they are trained in a stand-alone fashion. If this were the case, one could then safely choose the best individual architecture from the super-net after the search and use it for evaluation. However, this assumption was disproved in Zela et al. (2020b) and Chapter 5, who showed a correlation close to zero between the two rankings on complex search spaces (Ying et al., 2019; Liu et al., 2019b). The major reason behind this is fairly intuitive: To be optimal, different individual architectures should have different parameter values, which they cannot because the parameters are shared. Super-net training will thus *not* produce the same results as stand-alone training. More importantly, there is no guarantee that even the relative ranking of the architectures will be maintained. While for simple, linear search spaces the ranking can be improved by using a carefully crafted sampling strategy (Chu et al., 2019; 2020), addressing the ranking disorder for more realistic, complex search spaces remains an open problem (Yu et al., 2020b).

In this chapter, we propose to explicitly encourage architectures represented by the super-net to have a similar ranking to their counterparts trained in a stand-alone fashion. As illustrated by Figure 7.1, we leverage a set of landmark architectures, that is, architectures with known stand-alone performance, to define a regularization term that guides super-net training towards this goal. We show that a small set of landmark architectures suffices to significantly improve the global ranking correlation, so that the overall search procedure, including the independent training of the landmark architectures, remains tractable.

Our regularization term is general and does not make assumptions about the specific sampling algorithm used for super-net training. As such, it can easily be combined with many popular

weight-sharing NAS algorithms. We demonstrate this by integrating it into three different algorithms (Guo et al., 2019; Luo et al., 2018a; Dong & Yang, 2019b) that are representative of three different categories of weight-sharing NAS algorithms: i) Algorithms that sample architectures from the super-net in an unbiased manner throughout the super-net training (Li & Talwalkar, 2019; Bender et al., 2018b; Guo et al., 2019; Chu et al., 2019); ii) approaches that employ learning-based samplers, which are updated during the training based on the performance of the partially-trained super-net (Pham et al., 2018b; Li et al., 2020a; Luo et al., 2018a; Wang et al., 2019; Zhao et al., 2020); and iii) algorithms that rely on differentiable architecture search (Liu et al., 2019b; Cai et al., 2018b; Xie et al., 2018; Nayman et al., 2019; Xu et al., 2020).

Our extensive experiments on CIFAR-10 and ImageNet show that landmark regularization significantly reduces the ranking disorder that occurs in these algorithms and that they are consequently able to consistently find better-performing architectures. To further showcase the effectiveness and generality of our approach, we study its use in the context of architecture search for monocular depth estimation. To the best of our knowledge, this is the first attempt at performing NAS for this task. We, therefore, construct a dedicated search space and show that a landmark-regularized NAS algorithm can find novel architectures that improve upon the state of the art in this field.

## 7.2 Preliminaries

We first revisit the basics of super-net training and highlight the ranking disorder problem.

Let $\Omega$ be a search space, defined as a set of $N$ neural network architectures $a_i, i \in [1, N]$. stand-alone training optimizes the parameters $\theta_{a_j}$ of architecture $a_j$ independently from the other architectures by minimizing a loss function $\mathscr{L}(x, \theta_{a_j})$, thus yielding the optimal parameters $\theta^*_{a_j}$ for the given training data $x_{train}$. Without weight sharing, NAS then aims to train a search algorithm $S$ to sample architectures $S(\Omega) = \{a_k\}$ whose stand-alone performance outperforms that of other architectures, that is, $\mathscr{L}(x, \theta^*_{a_k}) < \mathscr{L}(x, \theta^*_{a_j})$, $\forall a_j \notin S(\Omega)$. In its simplest form, i.e., random search, there is no search algorithm to train per se, and one just samples a set of architectures, trains them in a stand-alone fashion, and ranks them to choose the ones with lowest loss.

With a proper search algorithm, however, NAS without weight sharing requires training and evaluating an impractically large amount of stand-alone architectures Zoph & Le (2017a); Zoph et al. (2018); Wang et al. (2020); Tan et al. (2018); Tan & Le (2019) . To circumvent this, weight-sharing NAS strategies construct a super-net $\theta^s$ that encompasses all architectures in the search space. The relative performance of individual architectures sampled from the super-net then acts as an estimate of their relative stand-alone performance. Training is typically formulated

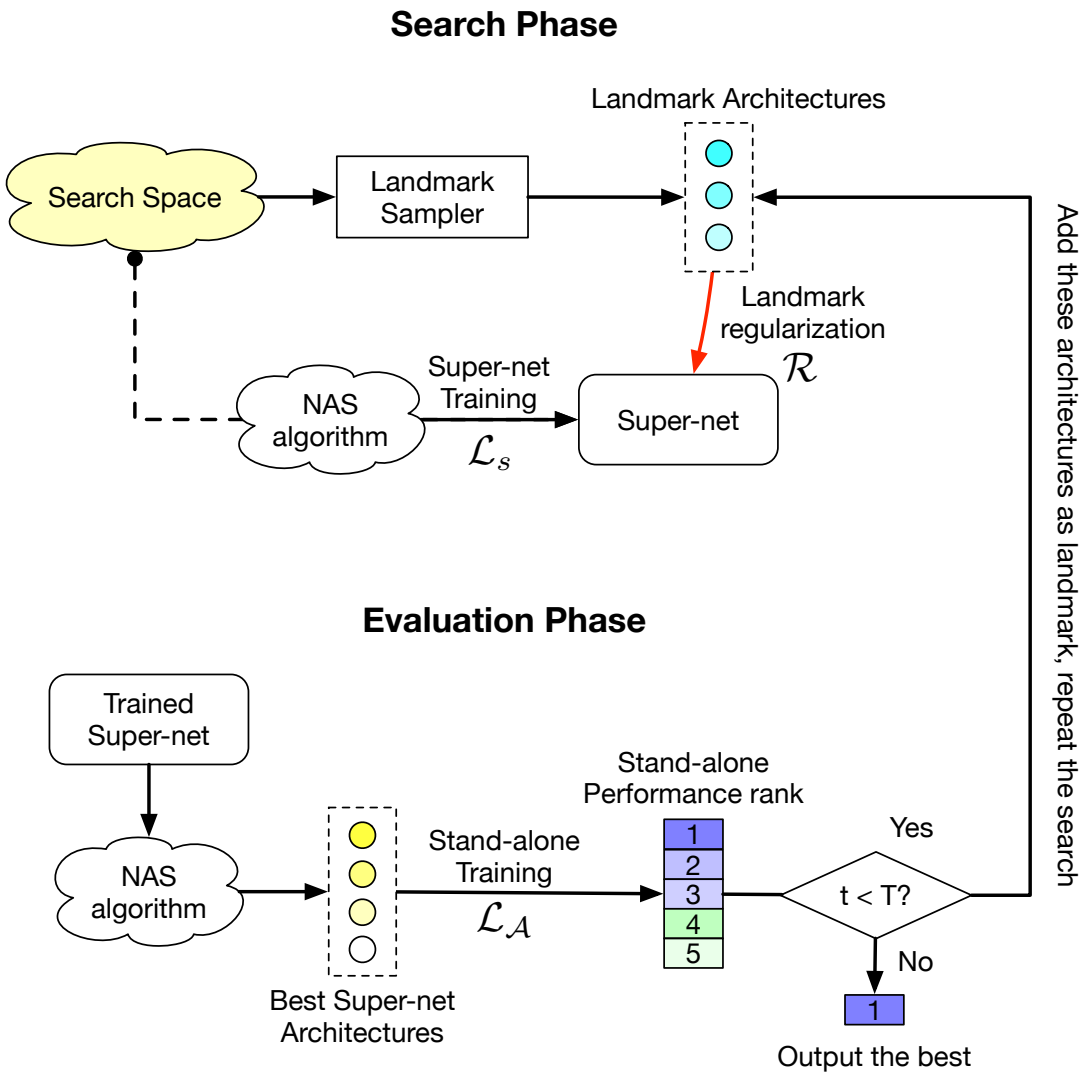



Figure 7.2: **Overview of our approach.** Left: During the search phase, we first sample a set of landmark architectures and obtain their stand-alone performance. We train the super-net with our regularization term such that the landmark ranking is preserved. Right: After a round of training, we sample the best architectures given the current super-net performance and evaluate their stand-alone performance. We add these architectures to the set of landmarks and repeat the process for a few iterations.

as minimizing a joint loss over all architectures that are represented by the super-net:

$$\mathscr{L}_s(\theta^s) = \sum_{i=1}^{N} \mathscr{L}(x, \theta^s_{a_i}), \tag{7.1}$$

where the optimization is made tractable by randomly sampling terms from this sum for each update.

In contrast to stand-alone training, parameters overlap between different architectures and in general we have that $\theta^s_{a_i} \cap \theta^s_{a_j} \neq \varnothing$. Since the parameters shared by $a_i$ and $a_j$ would typically not have the same optimal values in the stand-alone training of Chapter 4, the optimal solution of super-net training is *not* the same as that of stand-alone training, and neither is the ranking of the architectures.

## 7.3 Landmark regularization

We address the issue of low ranking correlation by introducing a simple yet effective approach to regularize super-net training with prior knowledge about the relative performance of individual architectures. To this end, we sample $M << N$ architectures to form a set of landmark architectures, $\Omega_L = \{a_i\}_{i=1}^{M}$, and obtain their stand-alone performance $\mathscr{L}_{valid}(x_{valid}, \theta^*_{a_i})$ on validation data, where $\theta^*_{a_i} = \arg\min \mathscr{L}(x_{train}, \theta_{a_i})$. We shorten this to $\mathscr{L}_{\mathscr{A}}(x, \theta^*_{a_i})$ to simplify the notation. To ensure that the trained super-net is predictive of the performance of the stand-alone architectures, we aim to preserve the relative performance of these landmark architectures in the super-net. Formally, we want to enforce that

$$\mathscr{L}_{\mathscr{A}}(x, \theta^*_{a_i}) \leq \mathscr{L}_{\mathscr{A}}(x, \theta^*_{a_j}) \Rightarrow \mathscr{L}(x, \theta^s_{a_i}) \leq \mathscr{L}(x, \theta^s_{a_j}), \tag{7.2}$$

for all pairs of architectures $i, j \in [1, M]$.

To achieve this, we propose a differentiable regularization term that can readily be integrated into standard super-net training procedures. We first sort the landmark architectures in ascending order based on their ground-truth loss:

$$\forall i, j \in [1, M], \quad i < j \Leftrightarrow \mathscr{L}_{\mathscr{A}}(x, \theta^*_{a_i}) \leq \mathscr{L}_{\mathscr{A}}(x, \theta^*_{a_j}), \tag{7.3}$$

and use this ordering to define a regularization term

$$\mathscr{R}(\theta^s) = \sum_{i=1}^{M} \sum_{j=i+1}^{M} \max(0,\ \mathscr{L}(x, \theta^s_{a_i}) - \mathscr{L}(x, \theta^s_{a_j}))\,, \tag{7.4}$$

which penalizes deviations from the ordering induced by equation (7.2). Since all operations involved in the proposed regularizer are differentiable, it is straight-forward to implement in existing deep learning frameworks. Note that the several alternative formulations of this loss are possible. We will discuss them in

The landmark-regularized training loss is then given by

$$\mathscr{L}(\theta^s) = \mathscr{L}_s(\theta^s) + \lambda \mathscr{R}(\theta^s)\,, \tag{7.5}$$

where $\lambda > 0$ is a hyper-parameter. To avoid leakage of validation data into the super-net training, we follow Liu et al. (2019b); Luo et al. (2018a); Cai et al. (2018b) and split the training set into two parts. We use one part to evaluate the super-net loss $\mathscr{L}_s$ and the other part to evaluate the regularization $\mathscr{R}$ during training.

**Computational cost.** The complexity of evaluating the regularizer discussed above is $O(M^2)$. This factor can have a significant impact on the training time, as the regularizer has to be evaluated at every training iteration. To reduce this computational burden while still encouraging the super-net to encode a correct architecture ranking, we propose to randomly sample $m$ pairs of landmark architectures $i, j$ at each iteration and evaluate their ranking:

$$\mathscr{R}(\theta^s) = \sum_{i,j}^{m} \max(0, \mathscr{L}(x, \theta^s_{a_i}) - \mathscr{L}(x, \theta^s_{a_j}))\,. \tag{7.6}$$

This reduces the time complexity from $O(M^2)$ to $O(m)$. We will show empirically that even evaluation with a single pair introduces virtually no degradation of the resulting architecture ranking.

**Landmark selection.** The choice of the landmark architectures has an impact on the effectiveness of our regularizer. In particular, we would like to use landmarks that cover the complete search space. To promote this, we introduce the diverse landmark sampling strategy described by Algorithm 1. We start by randomly sampling a root architecture from the search space. We then generate $M-1$ diverse architectures by mutating the root architecture such that the Hamming distance is larger than a threshold $\tau$. For example, in the DARTS search space (Liu et al., 2019b), one architecture is encoded as a sequence, where each element represents selecting either a previous node or an operation. Mutating an architecture is then achieved by randomly altering one element, and the Hamming distance between two architectures is computed as the number of unequal elements.

**Regularization schedule.** For all practical applications, the number of landmark architectures will be several orders of magnitude smaller than the total number of architectures in the search space. The regularization term thus needs to have high weight to be effective and have a noticeable effect on the training. However, too much regularization can negatively impact the training dynamics, especially in the early stages. To alleviate this issue, we propose to enable regularization after a warm-up phase and to gradually increase its influence using a cosine schedule. Specifically, we set the regularization weight at epoch $t$ to

$$\lambda_t = \mathbf{1}_{t>t_w} \cdot \frac{1}{2}\left(1 + \cos\frac{\pi(t-t_w)}{t_{total}}\right)\lambda_{\max}, \tag{7.7}$$

**Algorithm 1:** Landmark-regularized training.

**Input :** Search space $\Omega$, NAS algorithm $\mathscr{S}$, super-net and stand-alone losses $\mathscr{L}_s$, $\mathscr{L}_{\mathscr{A}}$, distance threshold $\tau$.

*initialize super-net parameters* $\theta^s$
*initialize an empty landmark set* $\Omega_L$
**while** *step* $t < T$ **do**
  *Obtain landmark architectures*
  $a_0 \leftarrow$ `RandomSample`$(\Omega, 1)$
  **while** $|\Omega_L| < M \times T$ **do**
    $a_t \leftarrow$ `Mutate`$(a_0)$
    **if** $d_{Hamming}(a_t, a_0) > \tau$ **then**
      add $a_t$ to $\Omega_L$
    **end**
  **end**
  **foreach** *training step* **do**
    *Train super-net* $\mathscr{L}$ while sampling $m$ pairs
    $\{(a_i, a_j)\} \leftarrow$ `RandomSample` $(\Omega_L \times \Omega_L, m)$
    $\mathscr{L} = \mathscr{L}_s(\theta^s) + \lambda \mathscr{R}(\theta^s)$
    *Train sampler* $\mathscr{S}$ *if necessary*
  **end**
  *Sample architectures to get stand-alone performance*
  $\forall a_j \in \Omega_t \leftarrow \mathscr{S}(\Omega)$, obtain $\mathscr{L}_{\mathscr{A}}(x, \theta_{a_j})$
  $\Omega_L \leftarrow \Omega_L \cup \Omega_t$
**end**

**Output :** Model $a_t \leftarrow \arg\min_{a \in \Omega_t} \mathscr{L}_{GT}(x, \theta_a)$

where $t_w$ denotes the number of warm-up epochs, $t_{total}$ denotes the total number of epochs, and $\lambda_{max}$ denotes the final value for the regularization parameter.

**Application to existing NAS methods.** The proposed regularization term is independent of the search algorithm, and thus widely applicable to many different weight-sharing NAS algorithms. We discuss its use in three different classes of NAS strategies, specific instances of which will act as baselines in our experiments.

We categorize weight sharing NAS algorithms into three broad categories according to their interaction with the super-net: i) unbiased architecture sampling algorithms (Li & Talwalkar, 2019; Bender et al., 2018b; Guo et al., 2019; Chu et al., 2019) that sample one or a few paths uniformly at random, ii) learning based sampling that favors the most promising architectures given the performance of the current, partially trained super-net (Pham et al., 2018b; Li et al., 2020a; Luo et al., 2018a; Wang et al., 2019; Zhao et al., 2020), and iii) differentiable architecture search that parametrizes the architecture sampling probability as part of the super-net (Liu et al., 2019b; Cai et al., 2018b; Xie et al., 2018; Nayman et al., 2019; Xu et al., 2020).

For the first two categories, our method can be directly incorporated into the super-net training to improve its quality, and hence to improve the final search results. For algorithms in the last category, the algorithm is usually composed of two distinct phases that are executed alternatingly. In the the first phase the parameters that define the architecture are fixed and only the weights

are updated, whereas in the second phase the weights are fixed and the architecture parameters are updated. Our regularization term can directly be integrated into the first phase when using discrete architectures in the forward pass as in works that employ the Gumbel-Softmax (Dong & Yang, 2019b; Wu et al., 2018) or binary gates (Cai et al., 2018b; Xie et al., 2018). When using continuous architecture specifications in the forward pass (Liu et al., 2019b; Xu et al., 2020), we do not have a discrete sub-path to sample and evaluate the regularization term, so additional care has to be taken to incorporate it.

**Multi-iteration pipeline.** Figure 7.2 depicts the complete training pipeline. We first sample landmark architectures using our landmark selection strategy and obtain their stand-alone performance. We then train the NAS algorithm with landmark regularization. After a round of training, we sample the top $M$ architectures using the trained NAS algorithm, obtain their stand-alone performances, and add these architectures to the set of landmarks. We proceed training of the super-net with the expanded set of landmarks and iterate this process.

In our experiments on different tasks and algorithms, we observed a stable improvement after sampling 3 sets of $M = 10$ architectures for a total of 30 landmarks, which is computationally feasible. Additionally, our algorithm can leverage previously trained models to improve the search by simply adding them to the landmark set. Considering that search spaces usually encompass billions of architectures, the number of landmarks is negligible.

## 7.4 Experiments

To validate the landmark regularization we incorporate it into three popular weight-sharing algorithms and evaluate them on the task of image classification using the CIFAR-10 (Krizhevsky et al., 2009b) and ImageNet (Russakovsky et al., 2015) datasets. We then discuss a new search space for monocular depth estimation architectures and show that our approach also applies to this new task. Finally, we ablate the key components and hyperparameter choices of our landmark regularization.

We used PyTorch for all our experiments and follow the evaluation framework defined in Chapter 5 to ensure a fair comparison with the baseline methods. Following (Radosavovic et al., 2019; 2020), we shorten the training time from 600 to 100 epochs on CIFAR-10 and from 250 to 50 on ImageNet, which still yields a good prediction quality. We will make our code publicly available upon acceptance.

**Baselines.** We select single-path one-shot (SPOS) as a representative unbiased architecture sampling algorithm. We use SPOS to train the super-net, followed by an evolutionary search to select the best models based on the super-net performance (Guo et al., 2019). Among the learning-based architecture sampling methods, we select neural architecture optimization (NAO) (Luo et al., 2018a), which trains an explicit auto-encoder-based performance predictor. Finally, for differentiable architecture search, we select the gradient-based search using a differentiable

architecture sampler (GDAS) (Dong & Yang, 2019b), which has been widely used in other works (Xie et al., 2018; Wu et al., 2018; Cai et al., 2018b; Li et al., 2020a).

**Hyperparameters.** We sample $M = 10$ landmark architectures at each iteration, and perform $T = 3$ iterations. We sample $m = 1$ pairs for each training step and set $\lambda_{max} = 10$ in all of our experiments unless otherwise specified. The Hamming distance threshold $\tau$ is set according to the configuration of each search space. We train the baselines for the same total number of epochs, to ensure that any performance improvement cannot be attributed to our approach sampling more architectures.

**Metrics.** We follow (Guo et al., 2019; Yu et al., 2020b) and report the ranking correlation in terms of the sparse Kendall-Tau (S-KdT). We sample 200 architectures randomly to compute this metric for the CIFAR-10 experiments, 90 for the ImageNet experiments, and 20 for the monocular depth estimation experiments. Note that we exclude the landmark architectures from this set to avoid reporting overly optimistic numbers for our approach. Furthermore, following (Dong & Yang, 2020), we report the mean and best stand-alone performance of the best architectures found over three independent runs.

### 7.4.1 Image classification on CIFAR-10

Since the inception of NAS, CIFAR-10 has acted as one of the main datasets to benchmark NAS performance (Zoph et al., 2018; Pham et al., 2018b; Luo et al., 2018a; Guo et al., 2019; Xu et al., 2020; You et al., 2020). We utilize two search spaces, NASBench-101 and NASBench-201, for which the stand-alone performance of many architectures is known.

**NASBench-101.** NASBench-101 (Ying et al., 2019) is a cell search space that contains 423,624 architectures with known stand-alone accuracy on CIFAR-10. It is the largest tabular benchmark search space to date. We use the implementation of Chapter 5 to benchmark the performance of SPOS and NAO. We do not report the results of GDAS on this search space, as the matrix-based configuration used in NASBench-101 is not amenable to differentiable approaches (Ying et al., 2019; Zela et al., 2020b). We set the Hamming distance threshold to $\tau = 5$.

As shown in Table 7.1 (top), landmark regularization improves the ranking correlation (sparse Kendall-Tau) from 0.267 to 0.347 for SPOS, and from 0.329 to 0.457 for NAO. This translates to a 1-2% improvement in terms of mean stand-alone accuracy over three runs. The best architecture discovered on this search space, thanks to our regularizer, ranks 9313-th which corresponds to the top 2% of architectures. Note that NAO without landmark regularization consistently got trapped in local minima, leading to its best architecture being only in the top 30%.

**NASBench-201.** We report additional results on the NASBench-201 cell search space (Dong & Yang, 2020), where each cell is a fully connected graph with 4 nodes. Each edge contains 5 searchable operations including a *zero* operation that effectively removes the edge from the

| | NASBench-101 | | | |
|---|---|---|---|---|
| Model | S-KdT | Mean Acc. | Best Rank | Best Acc. |
| SPOS | 0.267 ± 0.02 | 91.02 ± 0.52 | 38953 | 92.82 |
| SPOS+Ours | 0.347 ± 0.03 | 92.48 ± 0.51 | 27697 | 92.99 |
| NAO | 0.329 ± 0.11 | 90.56 ± 0.88 | 131969 | 91.60 |
| NAO+Ours | 0.457 ± 0.03 | 92.23 ± 1.32 | 9313 | 93.34 |
| | NASBench-201 | | | |
| Model | S-KdT | Mean Acc. | Best Rank | Best Acc. |
| SPOS | 0.771 ± 0.04 | 87.66 ± 4.95 | 3383 | 92.30 |
| SPOS+Ours | 0.802 ± 0.02 | 92.08 ± 0.37 | 2557 | 92.53 |
| GDAS | 0.691 ± 0.01 | 93.58 ± 0.12 | 463 | 93.48 |
| GDAS+Ours | 0.755 ± 0.01 | 93.98 ± 0.09 | 109 | 93.84 |
| NAO | 0.653 ± 0.05 | 91.75 ± 1.52 | 649 | 93.35 |
| NAO+Ours | 0.758 ± 0.05 | 92.84 ± 0.71 | 179 | 93.75 |

Table 7.1: **Results on NASBench-101 and NASBench-201.** We report the S-KdT at the end of training, the mean stand-alone accuracy of the searched architectures, the best rank, and the best accuracy. Each method was run 3 times.

computation graph. This yields a total of 15,625 architectures. We set $\tau = 5$ and benchmark all three baseline algorithms on this space. As shown in Table 7.1 (bottom), landmark regularization consistently improves the ranking correlation (sparse Kendall-Tau) across all three methods. We also observe an improvement in mean accuracy of more than 4% with SPOS. The best architecture is obtained by GDAS with landmark regularization and ranks 109-th. This corresponds to the top 0.7% architectures across the search space. In Figure 7.3 (top), we plot the mean sparse Kendall-Tau as super-net training progresses. We can see that the regularization improves the ranking correlation by a significant margin, especially towards the end of training.

**DARTS search space.** The NASBench search spaces are relatively small. To evaluate our

| Model | S-KdT | Mean Acc. | Params (in M) | Best Acc. |
|---|---|---|---|---|
| SPOS | 0.058 ± 0.010 | 92.80 ± 0.03 | 5.082 | 92.88 |
| SPOS+Ours | 0.206 ± 0.018 | 93.41 ± 0.43 | 2.181 | 93.84 |
| GDAS | 0.176 ± 0.014 | 90.48 ± 2.95 | 3.418 | 93.43 |
| GDAS+Ours | 0.209 ± 0.001 | 94.32 ± 0.28 | 2.540 | 94.60 |
| NAO | 0.102 ± 0.018 | 92.93 ± 0.87 | 5.080 | 93.03 |
| NAO+Ours | 0.231 ± 0.012 | 93.53 ± 0.43 | 2.184 | 93.78 |

Table 7.2: **Results on the DARTS search space on CIFAR-10.** Our best model (GDAS+Ours) surpasses the state-of-the-art model of Xu et al. (2020) (94.02% accuracy with 3.62M parameters) with 30% fewer parameters.

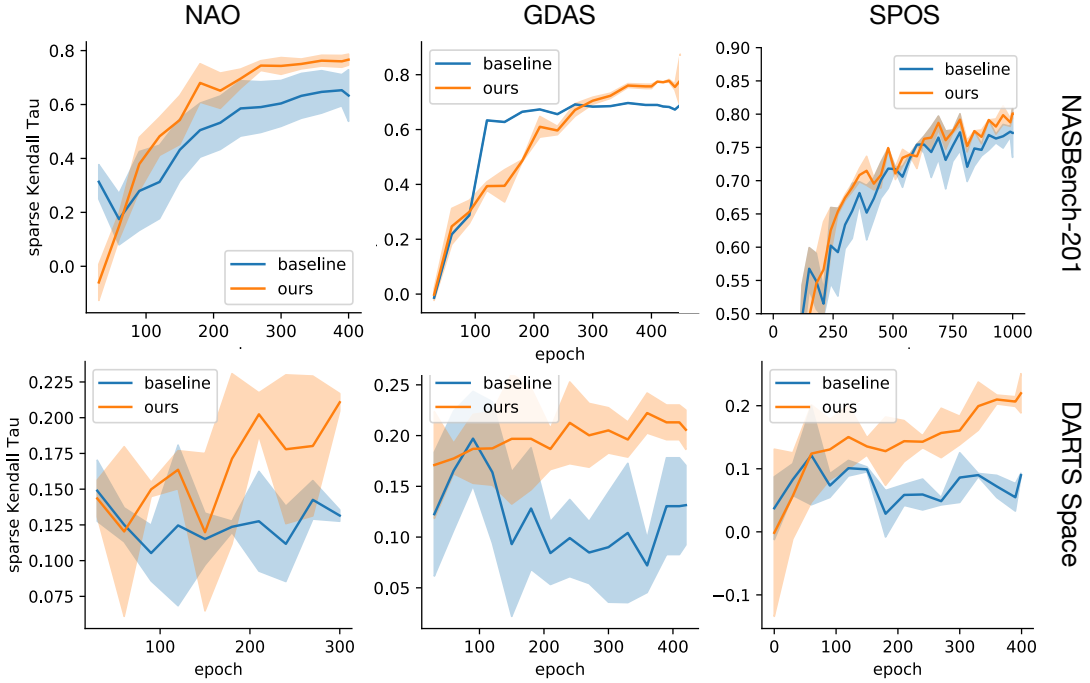



Figure 7.3: **Evolution of the S-KdT of three NAS algorithms on two search spaces.** Landmark regularization significantly improves the ranking correlation of the super-net in all cases.

approach in a more realistic scenario, we make use of the DARTS search space (Liu et al., 2019b; Xie et al., 2018; Xu et al., 2020; Nayman et al., 2019) which spans $3.3 \times 10^{13}$ architectures and is commonly used to evaluate real-world NAS performance. In contrast to NASBench, for which we could query the existing stand-alone performances, here, we need to train the discovered architecture from scratch. To compute the ranking correlation, we rely on the 5,000 pre-trained models of (Radosavovic et al., 2019) from which we randomly sample 200 as before.

We report our results in Table 7.2. We observe a clear improvement in terms of both sparse Kendall-Tau and mean accuracy over three independent searches across all three algorithms. Interestingly, the best model obtained using our ranking loss can surpass the baseline models by almost 1% with only around 50% of the parameters. The best model from GDAS with our regularizer surpasses the state-of-the-art model on this space (Xu et al., 2020) by 0.58% with 30% fewer parameters.

### 7.4.2 Image classification on ImageNet

To further evidence the effectiveness of our method, we move to ImageNet classification. For evaluation, we pick the best model of three independent runs predicted by each NAS algorithm and train them from scratch on the entire ImageNet training set for 50 epochs. We follow the setup of Xu et al. (2020) and use stochastic gradient descent with a linear learning rate scheduler,

| Model | S-KdT | Mean Top-1 | Params (in M) | Best Top-1 (50/250) |
|---|---|---|---|---|
| SPOS | 0.210 ± 0.010 | 64.57 ± 3.30 | 4.579 | 67.88 / 73.69 |
| SPOS+Ours | 0.267 ± 0.018 | 67.38 ± 0.92 | 4.766 | 68.61 / 74.58 |
| GDAS | 0.247 ± 0.012 | 67.50 ± 0.26 | 5.076 | 67.26 / 74.03 |
| GDAS+Ours | 0.272 ± 0.023 | 68.82 ± 0.24 | 5.073 | 68.36 / 74.82 |
| NAO | 0.253 ± 0.006 | 67.70 ± 0.43 | 4.675 | 68.21 / 73.71 |
| NAO+Ours | 0.279 ± 0.003 | 68.89 ± 0.58 | 4.488 | 69.58 / 74.92 |

Table 7.3: **Results on ImageNet.** We report mean top-1 accuracy over 3 runs after 50 epochs and best top-1 accuracy after 50 and 250 epochs, respectively.

which linearly increases from 0.1 to 0.5 in the first five epochs, and decreases to 0 over the remaining 45 epochs. We use a weight decay of 3e-4 and a label smoothing coefficient of 0.1 for all models. For this dataset, we use the popular DARTS search space, which provides 120 architectures trained for 50 epochs (Radosavovic et al., 2019). We split these architectures into 90 to report the sparse Kendall-Tau evaluation metric, and 30 for landmark sampling. We train the super-net with only 15% of the training dataset as in Xu et al. (2020). As the test data is not public, we report the top-1 validation accuracy as a metric for stand-alone training.

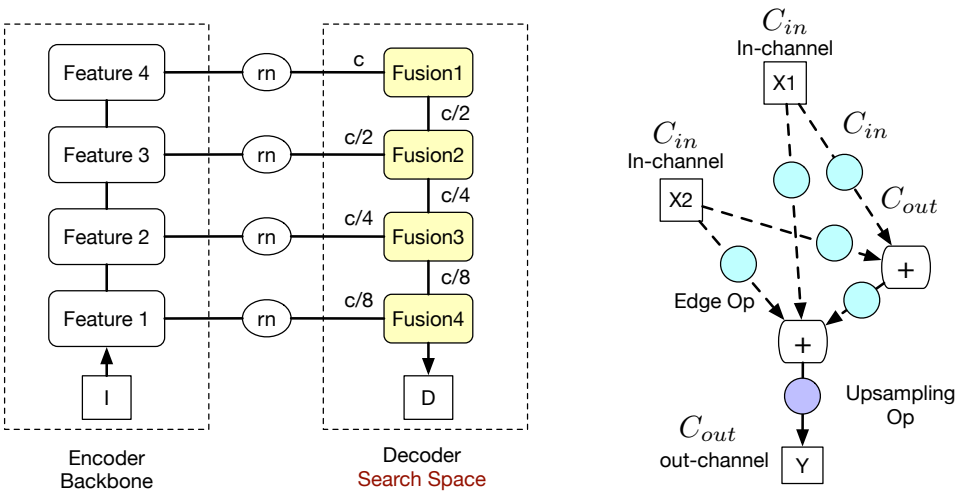
**Results.** Table 7.3 shows that landmark regularization consistently improves the three baselines. Overall, the best model is found by NAO with landmark regularization and achieves 74.92% top-1 accuracy, which outperforms the best baseline model by more than 1%. This further evidences the effectiveness of our regularization and its robustness across datasets.

### 7.4.3 Monocular depth estimation

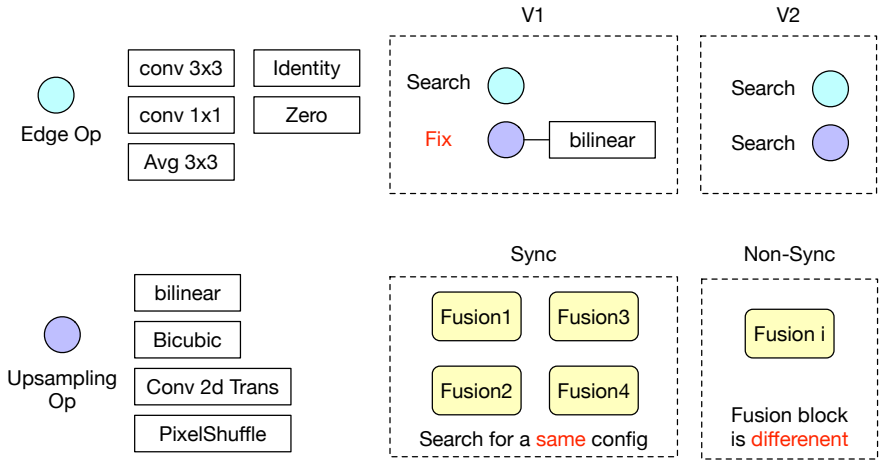
To showcase the generality of our approach, we apply our landmark-regularized NAS to the task of monocular depth estimation. Monocular depth estimation aims to predict pixel-wise depth from a single RGB image. Different paradigms have emerged on how to train single-image depth predictors, ranging from fully supervised training (Alhashim & Wonka, 2018; Xian et al., 2018; Ranftl et al., 2020), to self-supervised approaches (Godard et al., 2019). We follow the supervised paradigm and use the loss function proposed by Ranftl et al. (2020) to search for an architecture on the ReDWeb (Xian et al., 2018) dataset.

**Search space.** Figure 7.4 gives a detailed overview of the structure of our search space. Figure 7.4 (a) shows the structure of a traditional monocular depth estimation network. It is composed of a backbone network that acts as a feature extractor, typically pre-trained on ImageNet, followed by decoder fusion blocks that aggregate multi-scale information into a final prediction. Since using a pre-trained high-capacity network has been shown to be of high importance for final performance (Alhashim & Wonka, 2018; Ranftl et al., 2020), its architecture is fixed. We thus propose to search for the fusion blocks that define the decoder.

As depicted in Figure 7.4 (b), each fusion block is a search cell and takes the output of its




(a) Overview of MiDAS Net with searchable decoder (b) Decoder Fusion block as a search cell




(c) Operation choices (d) Search space configuration

Figure 7.4: **Monocular depth estimation search space.** (a) We modify MiDaS Ranftl et al. (2020) to construct the search space. We keep the backbone unchanged and search for fusion blocks in the decoder branches. (b) To define a fusion block, we model the input from the backbone and from the preceding fusion block as two nodes. We add two feature nodes, which sum up all previous inputs. The nodes are connected by edges, which represent the searchable operations. The final output node $Y$ takes the output of the last feature node and applies a potentially searchable upsampling operation. (c) Each edge, except the output edge, represents an *Edge Op (blue)* that contains five operations to choose from, while an upsampling edge *(purple)* contains four. (d) We propose four sub-space configurations: 'V1' indicates that we only search edge operations and fix the upsampling operator to bilinear upsampling. 'V2' includes a search over the upsampling operators. 'Sync' indicates that all fusion blocks share the same configuration, while 'Non-sync' allows them to differ.

preceding fusion block and the features from the backbone network as input. As the first fusion block does not have a predecessor, we simply duplicate the features from the encoder.

| Search space | Method | Sync | | Non-sync | |
|---|---|---|---|---|---|
| | | s-KdT | Best Val. loss | S-KdT | Best Val. loss |
| | | $\lvert\Omega\rvert = 3,125$ | | $\lvert\Omega\rvert = 9.5 \times 10^{13}$ | |
| V1 | SPOS | 0.751 ± 0.003 | 0.0960 ± 0.001 | 0.732 ± 0.008 | 0.0973 ± 0.002 |
| | SPOS+*Ours* | 0.781 ± 0.002 | 0.0958 ± 0.001 | 0.867 ± 0.044 | 0.0974 ± 0.001 |
| | | $\lvert\Omega\rvert = 12,500$ | | $\lvert\Omega\rvert = 2.4 \times 10^{16}$ | |
| V2 | SPOS | 0.401 ± 0.010 | 0.0957 ± 0.001 | 0.611 ± 0.004 | 0.0973 ± 0.001 |
| | SPOS+*Ours* | 0.555 ± 0.026 | **0.0936 ± 0.000** | 0.681 ± 0.002 | 0.0964 ± 0.001 |

Table 7.4: **Results on the RedWeb validation set.** The performance achieved by Ranftl et al. (2020) is 0.0942 (lower is better).

Figure 7.4 (c) shows the possible searchable operations, whereas (d) illustrates four possible configurations of our search space. We report the total number of architectures $|\Omega|$ for each search space in Table 7.4. While the search space is relatively simple, it is large enough to exhibit the problem of ranking disorder.

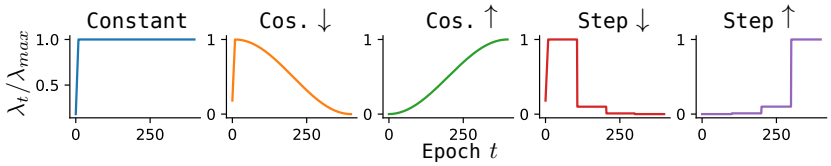
**Results.** We use the single-path one-shot algorithm to benchmark the influence of our regularization term on this task. We run $T = 2$ iterations, and sample 10 architectures in the first iteration. After the first round of training, we pick the three top models, obtain their stand-alone performance, and add them to the set of landmarks before we perform training for a second iteration. In addition to sparse Kendall-Tau, we report the scale- and shift-invariant loss (Ranftl et al., 2020) on the validation set as the performance metric.

Our results in Table 7.4 indicate that our method consistently yields an improvement in terms of sparse Kendall-Tau for all four configurations of the search space. Our best model improves upon the state-of-the-art handcrafted architecture of (Ranftl et al., 2020) in terms of the final performance.

### 7.4.4 Ablation studies

We finally provide an analysis of different aspects of our approach. We first evaluate the influence of the hyper-parameter $\lambda$ and different regularization schedules. We further study the robustness of our method to the landmark sampling distance $\tau$ and the number of iterations $T$. We conduct the ablations under the experimental setting described in Section 7.4.1 and use SPOS as baseline.

**Loss coefficient $\lambda$ and scheduler.** In Table 7.5, we ablate five different coefficient schedulers: constant regularization throughout training, two schedulers that gradually decrease the regularization, and two schedulers that gradually increase regularization (c.f.Table 7.5 (top)). For the constant and decreasing schedulers, we gradually increase the loss from 0 to $\lambda_{max}$ linearly in the first 10 epochs to avoid an abrupt change of the loss. As shown in Table 7.5, the cosine increasing scheduler, with $\lambda_{max} = 10$, yields the best results. We used this strategy in all our experiments.




| | Schedule | | | | | $\lambda_{max}$ | | |
|---|---|---|---|---|---|---|---|---|
| | Const | Cos ↓ | Cos ↑ | Step ↓ | Step ↑ | 1 | 10 | 100 |
| S-KdT | 0.753 | 0.712 | 0.803 | 0.709 | 0.785 | 0.789 | 0.803 | 0.793 |
| Mean Acc. | 91.13 | 91.38 | 92.08 | 91.02 | 91.31 | 91.89 | 92.08 | 91.97 |

Table 7.5: **Influence of the regularization parameter $\lambda$.** Left: Different schedules (c.f.top plots) to modify the strength of regularization throughout training. Right: Influence of $\lambda_{max}$ with the increasing cosine schedule.

| | SPOS | | | SPOS+*Ours* | | |
|---|---|---|---|---|---|---|
| Iteration | T=1 | T=3 | T=10 | T=1 | T=3 | T=10 |
| S-KdT | 0.763 | 0.771 | 0.758 | 0.760 | 0.802 | 0.811 |
| Mean Acc. | 91.13 | 91.48 | 91.78 | 91.29 | 92.08 | 92.17 |

Table 7.6: **Influence of the number of iterations $T$ on NASBench-201.**

| $m$ Pair(s) | 1 | 2 | 10 | 20 | 50 | 100 |
|---|---|---|---|---|---|---|
| S-KdT | 0.803 | 0.801 | 0.805 | 0.812 | 0.807 | 0.791 |
| Mean Acc. | 92.08 | 92.12 | 92.13 | 92.10 | 92.11 | 92.00 |

Table 7.7: **Influence of the stochastic approximation of Eq. 7.4.** We randomly sample $m$ pairs from 30 landmarks during each training step .

**Iterations.** We investigate the impact of the number of iterations $T$ in Table 7.6. We first pre-train the super-net for 250 epochs and then continue training for another 150 epochs per iteration $T$ both with and without landmark regularization. Table 7.6 indicates that our method improves the results when increasing the number of iterations, while the performance of the baseline does not increase. We selected $T = 3$ for our experiments as it strikes a balance between efficiency and accuracy.

**Sampling in loss computation.** In Table 7.7, we show that sampling a single pair of architectures per iteration to compute the regularization term is sufficient. Using more pairs does not improve the ranking correlation. We hypothesize this to be due to the fact that, overall, super-net training undergoes thousands of steps, thus providing a good coverage of all possible combinations of landmark architectures when the landmark set is small.

**Influence of the sampling distance.**

We finally evaluate the importance of the distance threshold $\tau$ in our landmark sampler. We first pre-train the super-net with SPOS on NASBench-201 for 150 epochs, then train for 50 epochs

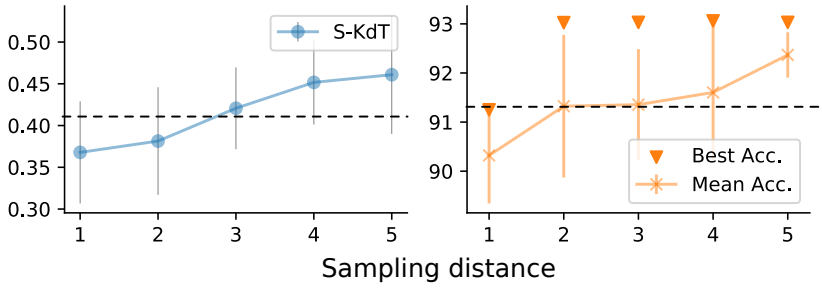



Figure 7.5: **Comparison of different sampling distances $\tau$.** The black, dashed line indicates the baseline performance.

Table 7.8: **Validating different loss function on NASBench-201.**

| Loss Name | ReLU | Softplus | ReLU-Norm | Sign | Tanh |
|---|---|---|---|---|---|
| S-KdT | 0.802 | 0.613 | 0.741 | 0.709 | 0.756 |
| Mean Acc. | 92.08 | 92.00 | 91.91 | 91.02 | 91.48 |

with landmark regularization, where we sample landmarks with varying distances $\tau$. We repeated this experiment 3 times and report the average sparse Kendall-Tau as well as the mean accuracy of the discovered architectures. Figure 7.5 shows that the performance degrades if $\tau$ is chosen too small. Performance gradually improves as $\tau$ increases. This highlights the importance of a diverse set of landmarks. Note that our sampling strategy does not require knowledge about the stand-alone performance and thus is applicable to new NAS search spaces.

**Loss formulation.**

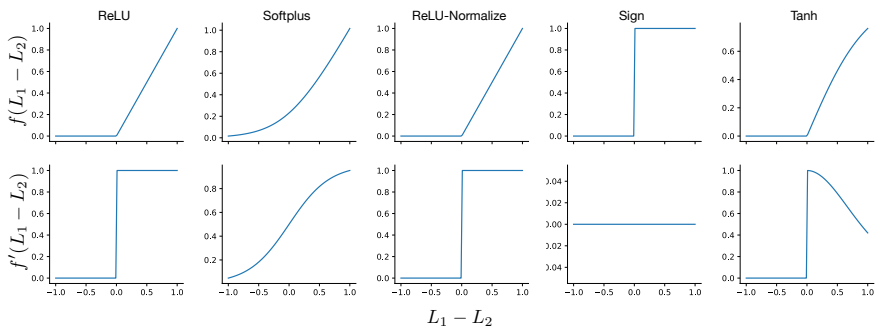



Figure 7.6: **Visualization of different loss formulation** $f(L_1 - L_2)$ and its first-order gradient. Note that the loss shape of ReLU and ReLU-Normalize are the same, because we normalize $L_i$ but not its associated function.

We validate other possible formulations of the ranking regularization. Specifically, to penalize the architectures that disobey the ideal ranking order, we evaluate various instances of loss functions $f$ that obey $\mathscr{R} = f(\mathscr{L}(x, \theta^s_{a_i}), \mathscr{L}(x, \theta^s_{a_j})) > 0$ when $i < j$ but $\mathscr{L}(x, \theta^s_{a_i}) > \mathscr{L}(x, \theta^s_{a_j})$. We shorten

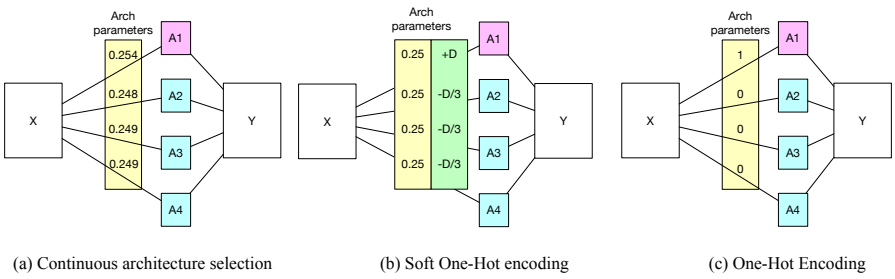
$\mathscr{L}(x, \theta^s_{a_i})$ to $L_i$ during this discussion, and rewrite the formulation as $\mathscr{R} = f(L_i - L_j)$. We evaluate the following functions (c.f.Figure 7.6):

- *ReLU*: $f(x) = \max(0, x)$
- *Softplus*: $f(x) = \ln 1 + e^{kx}/k$, where $k = 3$
- *ReLU-normalized*: $f(x) = \max(0, x)$, however, we normalize $L_i$ s.t. $L_i \in [0, 1]$.
- *Sign*: $f(x) = \max(0, \text{sign}(x))$
- *Tanh*: $f(x) = \max(0, \tanh(x))$

Table 7.8 reports the results of the different loss functions on NASBench-201. The best performance is achieved by the ReLU formalism, which corresponds to the formulation shown in Section 7.3.

### 7.4.5 Case study: Application on PC-DARTS

We show how to apply our method to DARTS-based methods with a continuous sampling of the super-net. We pick the state-of-the-arts method PC-DARTS (Xu et al., 2020) to experimentally validate our approach. We conduct our experiments on NASBench-201 search space.




(a) Continuous architecture selection (b) Soft One-Hot encoding (c) One-Hot Encoding

Figure 7.7: **Different encodings of the continuous super-net.** (a) Traditional continuous encoding used in DARTS-based method. The architecture is encoded in architecture parameters that sum to one. (b) We propose to select one architecture during continuous training by adding a fixed amount perturbation $D$ to the selected branch, and subtracting a perturbation of $D/(n-1)$ from the other branches. (c) Naive approach to select one architecture by one-hot encoding. Note that, we refer (b) as soft one-hot encoding because it is in between of (a) and (c).

**Naive one-hot encoding in continuous space.** We introduce a novel soft one-hot encoding to apply our landmark regularization in a continuous super-net. A fundamental difference between discrete super-net with a continuous one is how to select a single architecture from the continuous architecture specification (also known as rounding), which is a requirement to be able to evaluate our regularization term. As shown in Figure 7.7(c), a single architecture in a discrete space can be picked directly from its associated weights. In continuous space, however, there is no real architecture selection because the architecture parameters are part of the super-net as shown in Figure 7.7(a). To the best of our knowledge, there is no existing method that can effectively

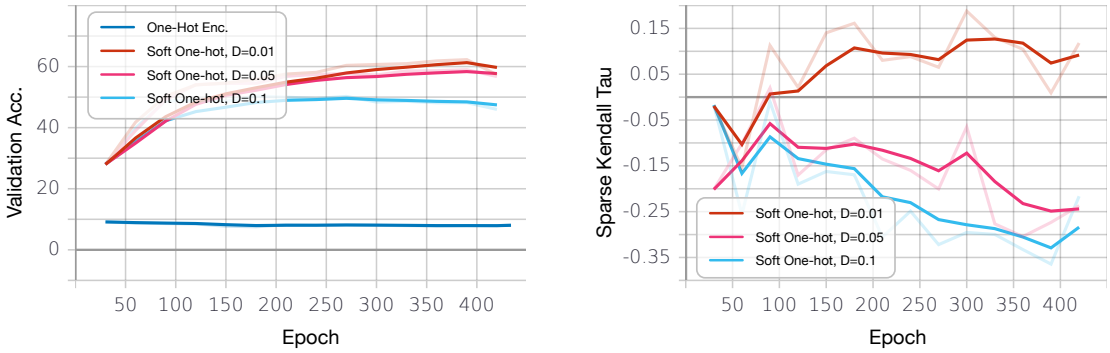



Figure 7.8: **Soft One-hot encoding results.** (Left) Validation accuracy during super-net training for one-hot encoding and our proposed soft one-hot with different $D$. We can observe that the naive one-hot encoding always yields accuracies around 0.1 on the validation set, whereas our proposed soft one-hot ranges from 50% to 60%. (Right) We further compare sparse Kendall-Tau. We choose $D = 0.01$ for further experiments with our landmark regularization.

sample one architecture from a continuous search space to conduct ranking analysis.

**Soft one-hot encoding.** To this end, we propose a simple but effective encoding to sample one architecture from a continuous super-net, dubbed as soft one-hot encoding. The idea is to relax the traditional one-hot encoding to simulate the continuous one. As shown in Figure 7.7(b), we start with a uniform distribution and add a perturbation $D$ to the selected branch, while substracting a perturbation by $D/(n-1)$ from the other branches such that the sum of weights remains equal to one. In Figure 7.8 (left), we can see this drastically increases validation accuracy from 10% to around 50%. In addition, we ablate different values of $D \in \{0.01, 0.05, 0.1\}$ in Figure 7.8 (right). Setting $D = 0.01$ reaches the highest sparse Kendall-Tau.

**Applying landmark regularization with the soft one-hot encoding.** We further deploy this method with our landmark regularization. We keep all configuration the same as in our previous NASBench-201 GDAS experiment. Our regularization improves the sparse Kendall-Taufrom 0.12 to 0.269, and improves the best model from rank 906 to 245. This evidences that our method can generalize to continuous differentiable architecture search method.

### 7.4.6 Searched model

For DARTS search space, we report the configuration of the searched models on CIFAR-10 in Figure 7.9 and on ImageNet in Figure 7.10.

## 7.5 Conclusion

We have presented a simple yet effective approach to leverage a few landmark architectures to guide the super-net training of weight-sharing NAS algorithms towards a better correlation with

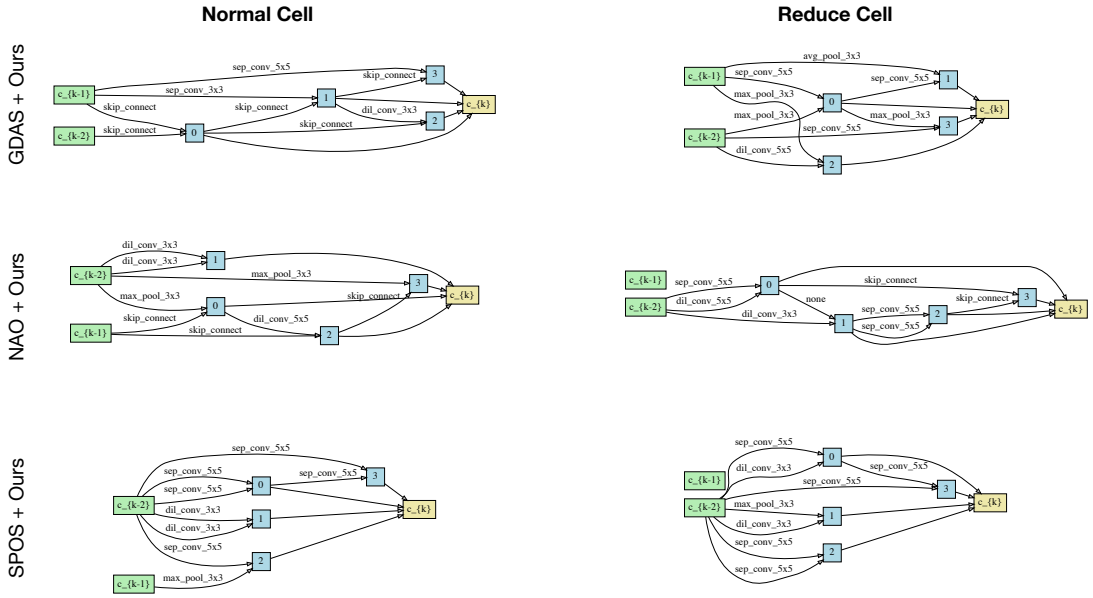



Figure 7.9: **Best architectures discovered by our algorithms on CIFAR-10.**

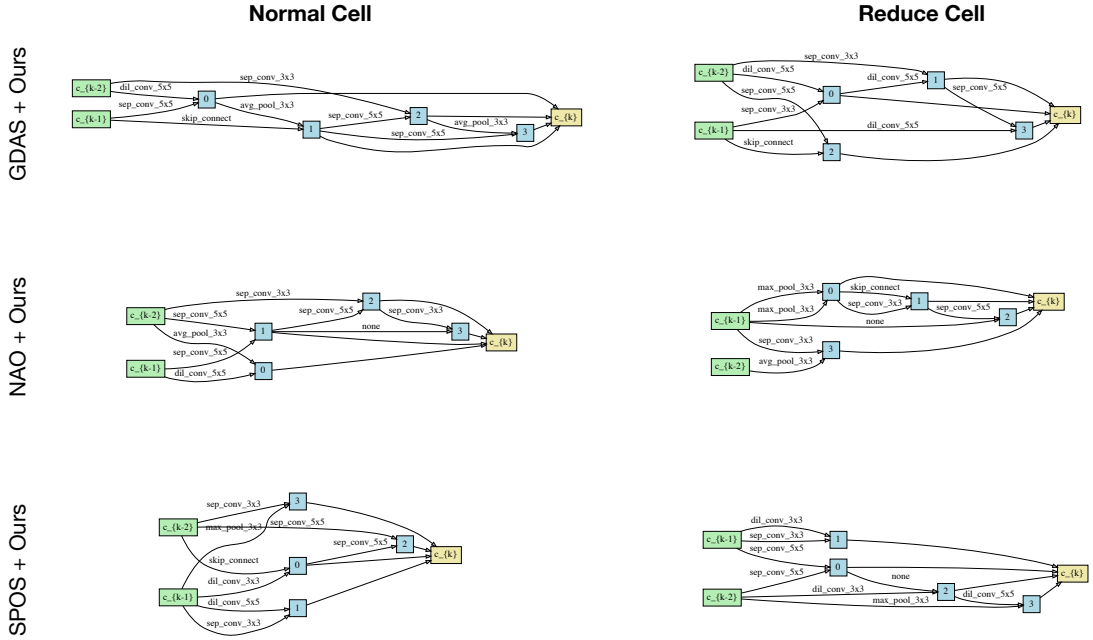



Figure 7.10: **Best architectures discovered by our algorithms on ImageNet.**

stand-alone performance. Our strategy is applicable to most NAS algorithms and our experiments have shown that it consistently improves both the ranking correlation between the super-net and stand-alone performance as well as the final performance across three different search algorithms and three different tasks. Additionally, our approach can leverage the information from previously trained stand-alone models to improve NAS performance.

# 8 Conclusion and future directions

In summary, this thesis has covered a variety of current machine learning trends, ranging from human-designed networks to automatically searched ones. We have started by manually building convolutional neural networks for two different tasks, exploiting heuristics from earlier experience in a human-designed fashion. We have then moved to an automatic machine learning approach, neural architecture search with parameter sharing, discovered several drawbacks in this field, and proposed effective solutions to address them. Below, we first summarize the contributions of the individual chapters, and then discuss the remaining limitations in this field and identify some potential directions for future research.

## 8.1 Thesis Summary

In Chapter 1, we have presented a statistically-motivated second-order pooling approach motivated by the earlier success of covariance descriptors. Compared to the previous second-order baselines, which yield final representations that are orders of magnitude larger than the standard first-order ones, our approach effectively maps the representation to a first-order one, and significantly reduces the memory consumption and implementation difficulty. Furthermore, our statistically-motivated approach leads to a Gaussian-distributed final representation also similar to the first-order ones inherently used by standard first-order networks, thus reducing the training difficulty. Our approach out-performed the previous baselines by a significant margin in terms of computational cost on many fine-grained classification tasks and for two popular network architectures, VGG-16 and ResNet-50.

In Chapter 2, we have observed that most current segmentation CNNs rely on very deep networks as backbone, which makes them hard to train on limited data and translates to long inference times on standard GPUs. We have proposed to transform the traditional light-weight U-Net into a recurrent formalism, which significantly reduces the number of model parameters while retaining its performance. This approach substantially reduces the GPU cost and achieves real-time performance on mobile devices, while out-performing earlier baselines over multiple benchmark

segmentation tasks.

From Chapter 3 on, we have then turned to neural architecture search, starting by briefly introducing the core concepts of neural architecture search and parameter sharing. In Chapter 4, we have identified the phenomenon of multi-model forgetting, where the shared weights of previously trained models are overwritten during training subsequent ones. This negatively impacts the super-net training in one-shot neural architecture search. We have shown that the degree of forgetting is positively correlated with the proportion of weight sharing, and derived a statistically sound approach to addressing this.

In Chapter 5, we have revisited the current evaluation of NAS algorithms, which surprisingly showed that the search policy of many state-of-the-art algorithms cannot surpass a random baseline. We have then traced the reason for this phenomenon to two factors, the constrained nature of the search space and parameter sharing, which shuffles the architecture ranking of stand-alone training, i.e., makes the search policy see a randomly shuffled architectures ranking during the search.

In Chapter 6, we have focused on the core part of weight-sharing NAS, the super-net. We have isolated 14 factors that are agnostic to the NAS algorithms, and provided a detailed analysis of these factors over three benchmark datasets. We have discovered that many factors that are commonly adopted in weight-sharing NAS in fact negatively impact the prediction ability of the super-net. After carefully tuning these factors, a simple random algorithm can again surpass many state-of-the-art algorithms. However, the ranking disorder cannot be fundamentally improved by simply tuning these factors.

In Chapter 7, we have observed that the standard training objective of a super-net, that is, the summation of individual architecture losses, differs fundamentally from the goal of NAS, i.e., discovering the best architectures in a given search space. We have then incorporated a small set of carefully sampled landmark architectures to regularize the super-net training. We have shown this approach to effectively improve the super-net prediction ability of individual architecture performance and reduce the ranking disorder for three popular NAS algorithms and across multiple search spaces. Furthermore, we have extended our method to monocular depth estimation, constituting the first NAS approach to this task.

## 8.2 Limitations and Future Directions

In this section, we discuss some remaining limitations and provide potential future research directions to address them.

**Combining second-order pooling with NAS.** During the experiments we performed for Chapter 1, we observed that second-order pooling is not always better than first-order ones. While second-order pooling outperforms first-order networks on fine-grained classification tasks, the

first-order ones tend to yield better results when the visual differences between the different classes are more significant, e.g., recognize cars from humans. In the future, one natural research direction would therefore be to combine the NAS approach with second-order pooling, letting the network automatically learn to select either first-order or second-order operations based on the input data.

**Multi-model forgetting in one-shot NAS.** In Chapter 4, we have introduced a novel weight plasticity loss (WPL) that addresses the forgetting issue. However, computing WPL relies on intermediate gradients during back-propagation, which increases the computational cost by factor around two compared to the baseline. As a consequence, we did not leverage this loss in the subsequent chapters. One potential direction would therefore be to study the effectiveness of WPL with our landmark regularization, and in particular evaluate if, together, these two loss terms can further boost the super-net training speed.

**Sampling landmark architectures.** In Chapter 7, we have introduced a landmark regularization scheme. While effective, our current landmark sampling approach merely aims to maximize the topological distance between the landmark architectures. While we have shown this strategy to outperform random sampling, there is no guarantee for this approach to be optimal. In the future, one potential direction to improve this could consist of exploiting meta-learning techniques to select better landmark architectures, i.e., search for the landmark architectures that yield the highest super-net Kendall Tau

**Going beyond a single super-net.** The ranking disorder still impedes progress in the current weight sharing NAS field. Based on our analysis in Chapter 6, the upper limit of the super-net Kendall Tau is around 0.8, while our landmark regularization of Chapter 7 usually increases this metric from 0.2 to 0.3 on harder search spaces. This raises the question of whether using a single super-net to represent the entire search space truly is reasonable, or if one should not rather reconsider the current one-shot neural architecture search approach in general. One possible direction would consist of again leveraging landmark architectures, but letting each landmark represent a group of architectures. One could then formulate a new training pipeline where poor landmark architecture groups are pruned whereas good ones are kept. Iteratively repeating this pruning process would then translate to finding a superior super-net, whose ranking correlation is high, and thus has higher potential to improve the NAS performance.

# A Proofs

## A.1 Overcome multi-model forgetting

**Lemma 1.** *Given a dataset $\mathscr{D}$ and two architectures with shared parameters $\boldsymbol{\theta}_s$ and private parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, and if $p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \mid \boldsymbol{\theta}_s, \mathscr{D}) = p(\boldsymbol{\theta}_1 \mid \boldsymbol{\theta}_s, \mathscr{D}) p(\boldsymbol{\theta}_2 \mid \boldsymbol{\theta}_s, \mathscr{D})$, we have*

$$p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_s \mid \mathscr{D}) \propto \frac{p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D}) p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s)}{\int p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) d\boldsymbol{\theta}_1}. \tag{4.1}$$

*Proof.* Using Bayes' theorem and ignoring constants, we have

$$\begin{aligned}
p(\boldsymbol{\theta} \mid \mathscr{D}) &= \frac{p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_s, \mathscr{D})}{p(\mathscr{D})} \\
&\propto p(\boldsymbol{\theta}_1 \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s, \mathscr{D}) p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s, \mathscr{D}) \\
&= p(\boldsymbol{\theta}_1 \mid \boldsymbol{\theta}_s, \mathscr{D}) p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s) \\
&\propto \frac{p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s, \mathscr{D}) p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s)}{p(\mathscr{D}, \boldsymbol{\theta}_s)} \\
&\propto \frac{p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s, \mathscr{D}) p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s)}{\int p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_s, \boldsymbol{\theta}_1) d\boldsymbol{\theta}_1} \\
&\propto \frac{p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D}) p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s)}{\int p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_s, \boldsymbol{\theta}_1) d\boldsymbol{\theta}_1},
\end{aligned}$$

where we used the conditional independence assumption $p(\boldsymbol{\theta}_1 \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s, \mathscr{D}) = p(\boldsymbol{\theta}_1 \mid \boldsymbol{\theta}_s, \mathscr{D})$ in the third line. □

We now derive a closed-form expression for the denominator of equation (4.1).

**Lemma 2.** *Suppose we have the maximum likelihood estimate $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ for the first model, write $\mathrm{Card}(\boldsymbol{\theta}_1) + \mathrm{Card}(\boldsymbol{\theta}_s) = p_1 + p_s = p$, and let the negative Hessian $\boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ of the log posterior probability distribution $\log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D})$ evaluated at $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$ be partitioned into four blocks corresponding to $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$ as*

$$\boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s) = \left[\begin{array}{c|c} \boldsymbol{H}_{11} & \boldsymbol{H}_{1s} \\ \hline \boldsymbol{H}_{s1} & \boldsymbol{H}_{ss} \end{array}\right].$$

*If the parameters of each model follow Normal distributions, i.e.,* $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) \sim \mathcal{N}_p(\mathbf{0}, \sigma^2 \boldsymbol{I}_p)$*, with* $\boldsymbol{I}_p$ *the p-dimensional identity matrix, then the denominator of equation (4.1),* $A = \int p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_s, \boldsymbol{\theta}_1) d\boldsymbol{\theta}_1$ *can be written as*

$$A = \exp\{l_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s) - \frac{1}{2}\boldsymbol{v}^\top \boldsymbol{\Omega} \boldsymbol{v}\} \times (2\pi)^{p_1/2} |\det(\boldsymbol{H}_{11}^{-1})|^{1/2}, \tag{4.2}$$

*where* $\boldsymbol{v} = \boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s$*,* $l_p(\boldsymbol{\theta}) = l(\boldsymbol{\theta}) - \boldsymbol{\theta}^T \boldsymbol{\theta}/2\sigma^2$*, and* $\boldsymbol{\Omega} = \boldsymbol{H}_{ss} - \boldsymbol{H}_{1s}^\top \boldsymbol{H}_{11}^{-1} \boldsymbol{H}_{1s}$ *.*

*Proof.* We have

$$p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s) p(\boldsymbol{\theta}_s, \boldsymbol{\theta}_1) \propto e^{l(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) - (\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)^T (\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)/2\sigma^2} = e^{l_p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)},$$

where $l(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) = \log p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$, and $l_p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) = l(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) - (\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)^T (\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)/2\sigma^2$.

Let $\boldsymbol{H}_p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) = \boldsymbol{H}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) + \sigma^{-2} \boldsymbol{I}_p$ be the negative Hessian of $l_p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$, with $\boldsymbol{I}_p$ the $p$-dimensional identity matrix and $\boldsymbol{H}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$ the negative Hessian of $l(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$.

Using the second-order Taylor expansion of $l_p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)$ around its maximum likelihood estimate $(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)$, we have

$$l_p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) = l_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s) - \frac{1}{2}[(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) - (\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)]^T \boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)[(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) - (\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)]; \tag{A.1}$$

the first derivative is zero since it is evaluated at the maximum likelihood estimate. We now partition our negative Hessian matrix as

$$\boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s) = \left[\begin{array}{c|c} \boldsymbol{H}_{11} & \boldsymbol{H}_{1s} \\ \hline \boldsymbol{H}_{s1} & \boldsymbol{H}_{ss} \end{array}\right],$$

which gives

$$\begin{aligned}
\boldsymbol{A} &= [(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) - (\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)]^T \boldsymbol{H}_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)[(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) - (\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s)] \\
&= (\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1)^T \boldsymbol{H}_{11}(\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1) + (\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s)^T \boldsymbol{H}_{ss}(\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s) + (\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s)^T \boldsymbol{H}_{s1}(\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1) + (\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1)^T \boldsymbol{H}_{1s}(\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s) \\
&= (\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1)^T \boldsymbol{H}_{11}(\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1) + (\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s)^T \boldsymbol{H}_{ss}(\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s) + (\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1)^T (\boldsymbol{H}_{1s} + \boldsymbol{H}_{s1}^T)(\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s).
\end{aligned}$$

Let us define $\boldsymbol{u} = \boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1$, $\boldsymbol{v} = \boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s$ and $\boldsymbol{w} = \boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}\boldsymbol{v}$. We then have

$$\begin{aligned}
(\boldsymbol{u}+\boldsymbol{w})^T\boldsymbol{H}_{11}(\boldsymbol{u}+\boldsymbol{w}) &= \boldsymbol{u}^T\boldsymbol{H}_{11}\boldsymbol{u} + \boldsymbol{u}^T\boldsymbol{H}_{11}\boldsymbol{w} + \boldsymbol{w}^T\boldsymbol{H}_{11}\boldsymbol{w} + \boldsymbol{w}^T\boldsymbol{H}_{11}\boldsymbol{u} \\
&= (\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1)^T\boldsymbol{H}_{11}(\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1) + (\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1)^T\boldsymbol{H}_{11}\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}(\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s) \\
&\qquad + \boldsymbol{v}^T\boldsymbol{H}_{1s}^T\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{11}\boldsymbol{H}_{1s}\boldsymbol{v} + \boldsymbol{v}^T\boldsymbol{H}_{1s}^T\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{11}(\boldsymbol{\theta}_1 - \hat{\boldsymbol{\theta}}_1) \\
&= \boldsymbol{A} - \boldsymbol{v}^T\boldsymbol{H}_{ss}\boldsymbol{v} + \boldsymbol{v}^T\boldsymbol{H}_{1s}^T H_{11}^{-1}\boldsymbol{H}_{1s}\boldsymbol{v} \\
&= \boldsymbol{A} - \boldsymbol{v}^T(\boldsymbol{H}_{ss} - \boldsymbol{H}_{1s}^T\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s})v \\
&= \boldsymbol{A} - \boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v},
\end{aligned}$$

with $\boldsymbol{\Omega} = \boldsymbol{H}_{ss} - \boldsymbol{H}_{1s}^T\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}$.

Thus

$$\boldsymbol{A} = (\boldsymbol{u} + \boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}v)^T\boldsymbol{H}_{11}(\boldsymbol{u} + \boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}\boldsymbol{v}) + \boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v}. \tag{A.2}$$

Given equation (A.2), we are now able to prove Lemma 2, as

$$\begin{aligned}
\int e^{l_p(\boldsymbol{\theta}_1,\boldsymbol{\theta}_s)}d\boldsymbol{\theta}_1 &= \int e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s) - \frac{1}{2}\boldsymbol{A}}d\boldsymbol{\theta}_1 \\
&= \int e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s)}e^{-\frac{1}{2}\boldsymbol{A}}d\boldsymbol{\theta}_1 \\
&= e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s)}\int e^{-\frac{1}{2}\boldsymbol{A}}d\boldsymbol{\theta}_1 \\
&= e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s)}\int e^{-\frac{1}{2}((\boldsymbol{u}+\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}\boldsymbol{v})^T\boldsymbol{H}_{11}(\boldsymbol{u}+\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}\boldsymbol{v}) + \boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v})}d\boldsymbol{\theta}_1 \\
&= e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s)}\int e^{-\frac{1}{2}((\boldsymbol{u}+\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}\boldsymbol{v})^T\boldsymbol{H}_{11}(\boldsymbol{u}+\boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}\boldsymbol{v}))}e^{-\frac{1}{2}\boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v}}d\boldsymbol{\theta}_1 \\
&= e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s) - \frac{1}{2}\boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v}}\int e^{-\frac{1}{2}(\boldsymbol{\theta}_1-\boldsymbol{z})^T\boldsymbol{H}_{11}(\boldsymbol{\theta}_1-\boldsymbol{z})}d\boldsymbol{\theta}_1 \\
&= e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s) - \frac{1}{2}\boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v}}(2\pi)^{\frac{p_1}{2}}|\det(\boldsymbol{H}_{11}^{-1})|^{\frac{1}{2}}(2\pi)^{-\frac{p_1}{2}}|\det(\boldsymbol{H}_{11}^{-1})|^{-\frac{1}{2}}\int e^{-\frac{1}{2}(\boldsymbol{\theta}_1-\boldsymbol{z})^T\boldsymbol{H}_{11}(\boldsymbol{\theta}_1-\boldsymbol{z})}d\boldsymbol{\theta}_1 \\
&= e^{l_p(\hat{\boldsymbol{\theta}}_1,\hat{\boldsymbol{\theta}}_s) - \frac{1}{2}\boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v}}(2\pi)^{\frac{p_1}{2}}|\det(\boldsymbol{H}_{11}^{-1})|^{\frac{1}{2}},
\end{aligned}$$

where we re-arranged the terms so that the integral is over a normal distribution with mean $\boldsymbol{z} = \hat{\boldsymbol{\theta}}_1 - \boldsymbol{H}_{11}^{-1}\boldsymbol{H}_{1s}(\boldsymbol{\theta}_s - \hat{\boldsymbol{\theta}}_s)$ and covariance matrix $\boldsymbol{H}_{11}^{-1}$, which can be computed in closed form. □

From Lemma 1 and Lemma 2, we can obtain equation (4.3) by replacing the denominator with the closed form above and taking the log on both size of equation (4.1). This yields

$$\begin{aligned}
\log p(\boldsymbol{\theta}|\mathscr{D}) &\propto \log p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) + \log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) + \log p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s) - \log\{\int p(\mathscr{D} \mid \boldsymbol{\theta}_1, \boldsymbol{\theta}_s)p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s)d\boldsymbol{\theta}_1\} \\
&= \log p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) + \log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s) + \log p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s) - l_p(\hat{\boldsymbol{\theta}}_1, \hat{\boldsymbol{\theta}}_s) + \frac{1}{2}\boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v} \\
&\propto \log p(\mathscr{D} \mid \boldsymbol{\theta}_2, \boldsymbol{\theta}_s) + \log p(\boldsymbol{\theta}_2, \boldsymbol{\theta}_s) + \log p(\boldsymbol{\theta}_1, \boldsymbol{\theta}_s \mid \mathscr{D}) + \frac{1}{2}\boldsymbol{v}^T\boldsymbol{\Omega}\boldsymbol{v}\,.
\end{aligned}$$

# Bibliography


HTC Vive Virtual Reality Toolkit. https://www.vive.com/.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Ahmed, K. and Torresani, L. Maskconnect: Connectivity learning by gradient descent. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 349–365, 2018.

Alhashim, I. and Wonka, P. High quality monocular depth estimation via transfer learning. *arXiv e-prints*, abs/1812.11941:arXiv:1812.11941, 2018. URL https://arxiv.org/abs/1812.11941.

Aljundi, R., Babiloni, F., Elhoseiny, M., Rohrbach, M., and Tuytelaars, T. Memory aware synapses: Learning what (not) to forget. *The European Conference on Computer Vision (ECCV)*, 2018.

Arandjelovic, R. and Zisserman, A. All About VLAD. In *CVPR*, pp. 1578–1585, 2013.

Arsigny, V., Fillard, P., Pennec, X., and Ayache, N. Log-Euclidean metrics for fast and simple calculus on diffusion tensors. *Magnetic Resonance in Medicine*, 2006.

Badrinarayanan, V., Kendall, A., and Cipolla, R. Segnet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *arXiv Preprint*, 2015.

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

Ballas, N., Yao, L., Pal, C., and Courville, A. Delving Deeper into Convolutional Networks for Learning Video Representations. *International Conference on Learning Representations*, 2016.

Bambach, S., Lee, S., Crandall, D., and Yu, C. Lending a Hand: Detecting Hands and Recognizing Activities in Complex Egocentric Interactions. In *International Conference on Computer Vision*, pp. 1949–1957, 2015.

Bartlett, M. S. and Kendall, D. G. The statistical analysis of variance-heterogeneity and the logarithmic transformation. *Supplement to the Journal of the Royal Statistical Society*, 8(1): 128–138, 1946. ISSN 14666162. URL http://www.jstor.org/stable/2983618.

Bell, S., Upchurch, P., Snavely, N., and Bala, K. Material Recognition in the Wild with the Materials in Context Database. In *CVPR*, 2015.

Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pp. 549–558, 2018a.

Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pp. 549–558, 2018b.

Bender, G., Liu, H., Chen, B., Chu, G., Cheng, S., Kindermans, P.-J., and Le, Q. V. Can weight sharing outperform random architecture search? an investigation with tunas. In *CVPR*, 2020.

Bennani-Smires, K., Musat, C., Hossmann, A., and Baeriswyl, M. Gitgraph - from computational subgraphs to smaller architecture search spaces. *International Conference on Learning Representations (ICLR), Workshop track*, 2018.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Efficient architecture search by network transformation. *AAAI*, 2018a.

Cai, H., Zhu, L., and Han, S. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *arXiv:1812.00332 [cs, stat]*, December 2018b. URL http://arxiv.org/abs/1812.00332. arXiv: 1812.00332.

Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. Once for all: Train one network and specialize it for efficient deployment. In *ICLR*, 2020. URL https://openreview.net/forum?id=HylxE1HKwS.

Carreira, J., Caseiro, R., Batista, J., and Sminchisescu, C. Semantic Segmentation with Second-Order Pooling. *ECCV*, 2012.

Caruana, R. Multitask learning. *Machine Learning*, 28(1):41–75, 1997.

Chen, L., Papandreou, G., Schroff, F., and Adam, H. Rethinking Atrous Convolution for Semantic Image Segmentation. *arXiv Preprint*, abs/1706.05587, 2017.

Chen, L., Zhu, Y., Papandreou, G., Schroff, F., and Adam, H. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. *arXiv Preprint*, abs/1802.02611, 2018a.

Chen, L.-C., Collins, M., Zhu, Y., Papandreou, G., Zoph, B., Schroff, F., Adam, H., and Shlens, J. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in Neural Information Processing Systems*, pp. 8713–8724, 2018b.

Chen, X., Mottaghi, R., Liu, X., Fidler, S., Urtasun, R., and Yuille, A. Detect What You Can: Detecting and Representing Objects Using Holistic Models and Body Parts. In *Conference on Computer Vision and Pattern Recognition*, 2014.

Chen, X., Xie, L., Wu, J., and Tian, Q. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *ICCV*, 2019a.

Chen, Y., Yang, T., Zhang, X., Meng, G., Pan, C., and Sun, J. DetNAS: Neural Architecture Search on Object Detection. *arXiv:1903.10979 [cs]*, March 2019b. URL http://arxiv.org/abs/1903.10979. arXiv: 1903.10979.

Cherian, A. and Sra, S. Riemannian Sparse Coding for Positive Definite Matrices. In *ECCV*, 2014.

Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv Preprint*, 2014.

Chollet, F. et al. Keras. https://github.com/fchollet/keras, 2015.

Chu, X., Zhang, B., Xu, R., and Li, J. FairNAS: Rethinking Evaluation Fairness of Weight Sharing Neural Architecture Search. *arXiv:1907.01845 [cs, stat]*, July 2019. URL http://arxiv.org/abs/1907.01845. arXiv: 1907.01845.

Chu, X., Zhou, T., Zhang, B., and Li, J. Fair DARTS: Eliminating unfair advantages in differentiable architecture search. *ECCV*, 2020.

Cimpoi, M., Maji, S., Kokkinos, I., Mohamed, S., and Vedaldi, A. Describing Textures in the Wild. In *CVPR*, 2014.

Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., and Schiele, B. The cityscapes dataset for semantic urban scene understanding. In *Conference on Computer Vision and Pattern Recognition*, 2016.

Cui, Y., Zhou, F., Wang, J., Liu, X., Lin, Y., and Belongie, S. Kernel Pooling for Convolutional Neural Networks. In *CVPR*, 2017.

Dalal, N. and Triggs, B. Histograms of Oriented Gradients for Human Detection. In *CVPR*, pp. 886–893, 2005.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A Large-Scale Hierarchical Image Database. In *Conference on Computer Vision and Pattern Recognition*, 2009.

Dietterich, T. G. Ensemble methods in machine learning. *Multiple Classifier Systems*, pp. 1–15, 2000.

Dong, X. and Yang, Y. Network pruning via transformable architecture search. In *Advances in Neural Information Processing Systems*, pp. 760–771, 2019a.

Dong, X. and Yang, Y. Searching for a robust neural architecture in four gpu hours. In *CVPR*, 2019b.

Dong, X. and Yang, Y. Nas-bench-102: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HJxyZkBKDr.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Fathi, A., Farhadi, A., and Rehg, J. Understanding Egocentric Activities. In *International Conference on Computer Vision*, pp. 407–414, 2011.

Fourure, D., Emonet, R., Fromont, E., Muselet, D., Tremeau, A., and Wolf, C. Residual conv-deconv grid network for semantic segmentation. In *British Machine Vision Conference*, 2017.

Freund, Y. and Schapire, R. E. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 1997.

Friedman, J., Hastie, T., and Tibshirani, R. Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2):337 – 407, 2000. doi: 10.1214/aos/1016218223. URL https://doi.org/10.1214/aos/1016218223.

Gao, Y., Beijbom, O., Zhang, N., and Darrell, T. Compact Bilinear Pooling. In *CVPR*, pp. 317–326, 2016.

Ghiasi, G., Lin, T.-Y., Pang, R., and Le, Q. V. NAS-FPN: Learning Scalable Feature Pyramid Architecture for Object Detection. *arXiv:1904.07392 [cs]*, April 2019. URL http://arxiv.org/abs/1904.07392. arXiv: 1904.07392.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

Godard, C., Aodha, O. M., Firman, M., and Brostow, G. J. Digging into self-supervised monocular depth estimation. In *ICCV*, 2019.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016a. http://www.deeplearningbook.org.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016b. http://www.deeplearningbook.org.

Guo, K., Ishwar, P., and Konrad, J. Action Recognition Using Sparse Representation on Covariance Manifolds of Optical Flow. In *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2010.

Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., and Sun, J. Single Path One-Shot Neural Architecture Search with Uniform Sampling. *arXiv:1904.00420 [cs]*, March 2019. URL http://arxiv.org/abs/1904.00420. arXiv: 1904.00420.

Haeffele, B. D. and Vidal, R. Global optimality in neural network training. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4390–4398, 2017.

Harandi, M. and Salzmann, M. Riemannian coding and dictionary learning: Kernels to the rescue. In *CVPR*, 2015.

Harandi, M. T., Sanderson, C., Hartley, R. I., and Lovell, B. C. Sparse Coding and Dictionary Learning for Symmetric Positive Definite Matrices - A Kernel Approach. In *ECCV*, 2012.

Harandi, M. T., Salzmann, M., and Hartley, R. From manifold to manifold: Geometry-aware dimensionality reduction for SPD matrices. In *ECCV*, 2014.

He, K., Zhang, X., Ren, R., and Sun, J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification. In *International Conference on Computer Vision*, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. In *CVPR*, pp. 770–778, 2016.

He, X. and Jaeger, H. Overcoming catastrophic interference by conceptors. *arXiv preprint arXiv:1707.04853*, 2017.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Huang, C. H., Boyer, E., Angonese, B. D. C., Navab, N., and Ilic, S. Toward User-Specific Tracking by Detection of Human Shapes in Multi-Cameras. In *CVPR*, 2015.

Huang, G., Liu, Z., Weinberger, K., and van der Maaten, L. Densely Connected Convolutional Networks. In *CVPR*, 2017.

Ioffe, S. and Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, 2015.

Ionescu, C., Vantzos, O., and Sminchisescu, C. Matrix backpropagation for Deep Networks with Structured Layers. 2015.

James, A. T. The non-central wishart distribution. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 229(1178):364–366, 1955. ISSN 00804630. URL http://www.jstor.org/stable/99771.

Januszewski, M., Kornfeld, J., Li, P. H., Pope, A., Blakely, T., Lindsey, L., Maitin-Shepard, J., Tyka, M., Denk, W., and Jain, V. High-precision automated reconstruction of neurons with flood-filling networks. *Nature methods*, 15(8):605, 2018.

Jin, H., Song, Q., and Hu, X. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956. ACM, 2019.

Johnson, R. A., Wichern, D. W., et al. *Applied multivariate statistical analysis*, volume 4. Prentice-Hall New Jersey, 2014.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. P. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pp. 2016–2025, 2018.

Kendall, M. G. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 2017.

Kong, S. and Fowlkes, C. Low-Rank Bilinear Pooling for Fine-Grained Classification. In *CVPR*, 2017.

Koniusz, P., Tas, Y., and Porikli, F. Domain Adaptation by Mixture of Alignments of Second- or Higher-Order Scatter Tensors. In *CVPR*, 2017.

Krizhevsky, A., Nair, V., and Hinton, G. Cifar-10 (canadian institute for advanced research). 2009a.

Krizhevsky, A., Nair, V., and Hinton, G. CIFAR-10 (canadian institute for advanced research). 2009b.

Krizhevsky, A., Sutskever, I., and Hinton, G. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, pp. 1106–1114, 2012.

Kubernetes. kubernetes.io, 2020. URL https://kubernetes.io/docs/reference/.

Lazebnik, S., Schmid, C., and Ponce, J. Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. In *CVPR*, 2006.

LeCun, Y. and Cortes, C. MNIST handwritten digit database. 2010.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1 (4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. Visualizing the loss landscape of neural nets. In *NeurIPS*, 2018.

Li, L. and Talwalkar, A. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

Li, P., Wang, Q., Zuo, W., and Zhang, L. Log-Euclidean Kernels for Sparse Representation and Dictionary Learning. In *ICCV*, 2013.

Li, P., Xie, J., Wang, Q., and Zuo, W. Is Second-Order Information Helpful for Large-Scale Visual Recognition? In *ICCV*, 2017.

Li, X., Lin, C., Li, C., Sun, M., Wu, W., Yan, J., and Ouyang, W. Improving one-shot NAS by suppressing the posterior fading. *CVPR*, 2020a.

Li, Y., Yang, Z., Wang, Y., and Xu, C. Adapting neural architectures between domains. *CVPR*, 2020b.

Li, Z. and Hoiem, D. Learning without forgetting. In *European Conference on Computer Vision*, pp. 614–629. Springer, 2016.

Lin, G., Milan, A., Shen, C., and Reid, I. Refinenet: Multi-Path Refinement Networks for High-Resolution Semantic Segmentation. In *Conference on Computer Vision and Pattern Recognition*, 2017.

Lin, T., RoyChowdhury, A., and Maji, S. Bilinear Cnn Models for Fine-Grained Visual Recognition. In *ICCV*, pp. 1449–1457, 2015.

Lin, T. Y. and Maji, S. Improved Bilinear Pooling with CNNs. In *BMVC*, 2017.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 19–34, 2018a.

Liu, C., Chen, L.-C., Schroff, F., Adam, H., Hua, W., Yuille, A., and Fei-Fei, L. Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation. *arXiv:1901.02985 [cs]*, 2019a. URL http://arxiv.org/abs/1901.02985.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. *International Conference on Learning Representations (ICLR), Conference track*, 2018b.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. *International Conference on Learning Representations (ICLR), Conference track*, 2018c.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018d.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *ICLR*, 2019b.

Long, J., Shelhamer, E., and Darrell, T. Fully Convolutional Networks for Semantic Segmentation. In *Conference on Computer Vision and Pattern Recognition*, 2015.

Lowe, D. G. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60 (2):91–110, November 2004. ISSN 0920-5691. doi: 10.1023/B:VISI.0000029664.99615.94. URL http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., and Banzhaf, W. Nsga-net: A multi-objective genetic algorithm for neural architecture search. *arXiv preprint arXiv:1810.03522*, 2018.

Luo, R., Tian, F., Qin, T., Chen, E.-H., and Liu, T.-Y. Neural architecture optimization. In *Advances in neural information processing systems*, 2018a.

Luo, R., Tian, F., Qin, T., Chen, E.-H., and Liu, T.-Y. Weight sharing batch normalization code, 2018b. URL https://www.github.com/renqianluo/NAO_pytorch/NAO_V2/operations.py#L144.

Luo, R., Tian, F., Qin, T., and Liu, T.-Y. Neural architecture optimization. *arXiv preprint arXiv:1808.07233*, 2018c.

Luo, R., Tan, X., Wang, R., Qin, T., Chen, E., and Liu, T.-Y. Semi-Supervised Neural Architecture Search. *NeurIPS*, 2020.

MacKay, D. J. C. A Practical Bayesian Framework for Backpropagation Networks. *Neural Computation*, 4(3):448–472, 1992.

Maninis, K.-K., Pont-Tuset, J., Arbeláez, P., and Van Gool, L. Deep retinal image understanding. In *Conference on Medical Image Computing and Computer Assisted Intervention*, 2016.

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pp. 114–119, Stroudsburg, PA, USA, 1994a. Association for Computational Linguistics. ISBN 1-55860-357-3. doi: 10.3115/1075812.1075835. URL https://doi.org/10.3115/1075812.1075835.

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, pp. 114–119. Association for Computational Linguistics, 1994b.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pp. 293–312. Elsevier, 2019.

Mnih, V. *Machine Learning for Aerial Image Labeling*. PhD thesis, University of Toronto, 2013.

Mosinska, A., Marquez-neila, P., Kozinski, M., and Fua, P. Beyond the Pixel-Wise Loss for Topology-Aware Delineation. In *Conference on Computer Vision and Pattern Recognition*, 2018.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Nayman, N., Noy, A., Ridnik, T., Friedman, I., Jin, R., and Zelnik, L. Xnas: Neural architecture search with expert advice. In *NeurIPS*, 2019.

Negrinho, R. and Gordon, G. DeepArchitect: Automatically Designing and Training Deep Architectures. *arXiv preprint arXiv:1704.08792*, 2017.

Newell, A., Yang, K., and Deng, J. Stacked hourglass networks for human pose estimation. In *European Conference on Computer Vision*, 2016.

Pan, S. J. and Yang, Q. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, 22 (10):1345–1359, 2010.

Pascanu, R. and Bengio, Y. Revisiting natural gradient for deep networks. *International Conference on Learning Representations (ICLR), Conference track*, 2014.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pp. 8026–8037. Curran Associates, Inc., 2019.

Peng, H., Du, H., Yu, H., Li, Q., Liao, J., and Fu, J. Cream of the crop: Distilling prioritized paths for one-shot neural architecture search. *CVPR*, 2020.

Pennec, X., Fillard, P., and Ayache, N. A Riemannian Framework for Tensor Computing. *IJCV*, 2006.

Pérez-Rúa, J.-M., Baccouche, M., and Pateux, S. Efficient progressive neural architecture search. *arXiv preprint arXiv:1808.00391*, 2018.

Perronnin, F., Sánchez, J., and Mensink, T. Improving the Fisher Kernel for Large-Scale Image Classification. *ECCV*, 2010.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient Neural Architecture Search via Parameter Sharing. *International Conference on Machine Learning (ICML)*, 2018a.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. *ICML*, 2018b.

Pinheiro, P. and Collobert, R. Recurrent Neural Networks for Scene Labelling. In *International Conference on Machine Learning*, 2014.

Pinkus, A. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8: 143–195, 1999. doi: 10.1017/S0962492900002919.

Pohlen, T., Hermans, A., Mathias, M., and Leibe, B. Full-resolution residual networks for semantic segmentation in street scenes. In *Conference on Computer Vision and Pattern Recognition*, 2017.

Poudel, R., Lamata, P., and Montana, G. Recurrent Fully Convolutional Neural Networks for Multi-Slice MRI Cardiac Segmentation. In *Reconstruction, Segmentation, and Analysis of Medical Images*, pp. 83–94. Springer, 2016.

Poudel, R. P., Bonde, U., Liwicki, S., and Zach, C. Contextnet: Exploring context and detail for semantic segmentation in real-time. *British Machine Vision Conference*, 2018.

Poudel, R. P., Liwicki, S., and Cipolla, R. Fast-scnn: Fast semantic segmentation network. *arXiv preprint arXiv:1902.04502*, 2019.

Purves, D., Augustine, G. J., Fitzpatrick, D., Katz, L. C., LaMantia, A.-S., McNamara, J. O., and Williams, S. M. Types of eye movements and their functions. In *Neuroscience*, 2011.

Quang, M. H., San-Biagio, M., and Murino, V. Log-Hilbert-Schmidt metric between positive definite operators on Hilbert spaces. *NIPS*, 2014.

Quattoni, A. and Torralba, A. Recognizing Indoor Scenes. In *CVPR*, pp. 413–420, 2009.

Radosavovic, I., Johnson, J., Xie, S., Lo, W.-Y., and Dollár, P. On Network Design Spaces for Visual Recognition. In *International Conference on Computer Vision*, May 2019.

Radosavovic, I., Kosaraju, R. P., Girshick, R., He, K., and Dollár, P. Designing network design spaces. *CVPR*, 2020.

Ramakrishna, V., Munoz, D., Hebert, M., Bagnell, J. A., and Sheikh, Y. Pose machines: Articulated pose estimation via inference machines. In *European Conference on Computer Vision*, 2014.

Ranftl, R., Lasinger, K., Hafner, D., Schindler, K., and Koltun, V. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020.

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

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., and Kurakin, A. Large-scale evolution of image classifiers. *International Conference on Machine Learning (ICML)*, 2017b.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

Rissanen, J. Fisher information and stochastic complexity. *IEEE Trans. Information Theory*, 42, 1996.

Romera, E., Alvarez, J. M., Bergasa, L. M., and Arroyo, R. Erfnet: Efficient residual factorized convnet for real-time semantic segmentation. *IEEE Transactions on Intelligent Transportation Systems*, 19(1):263–272, 2018.

Romera-Paredes, B. and Torr, P. Recurrent Instance Segmentation. In *European Conference on Computer Vision*, pp. 312–329, 2016.

Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *Conference on Medical Image Computing and Computer Assisted Intervention*, 2015.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Ryoo, M. S., Piergiovanni, A., Tan, M., and Angelova, A. Assemblenet: Searching for multi-stream neural connectivity in video architectures. In *ICLR*, 2020. URL https://openreview.net/forum?id=SJgMK64Ywr.

Saxena, S. and Verbeek, J. Convolutional neural fabrics. In *Advances in Neural Information Processing Systems*, 2016.

Sermanet, P., Chintala, S., and LeCun, Y. Convolutional Neural Networks Applied to House Numbers Digit Classification. In *ICPR*, 2012.

Shotton, J., Johnson, M., and Cipolla, R. Semantic texton forests for image categorization and segmentation. In *Conference on Computer Vision and Pattern Recognition*, 2008.

Siems, J., Zimmer, L., Zela, A., Lukasik, J., Keuper, M., and Hutter, F. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. *arXiv preprint arXiv:2008.09777*, 2020.

Silver, D., Yang, Q., and Li, L. Lifelong machine learning systems: Beyond learning algorithms. In *AAAI Spring Symposium Series*, 2013.

Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.

Slurm. Slurm workload manager, 2020. URL https://slurm.schedmd.com/documentation.html.

So, D. R., Liang, C., and Le, Q. V. The evolved transformer. In *ICML*, 2019.

Sra, S. A new metric on the manifold of kernel matrices with application to matrix geometric means. In *NIPS*, 2012.

Sra, S. and Cherian, A. Generalized Dictionary Learning for Symmetric Positive Definite Matrices with Application to Nearest Neighbor Retrieval. In *Machine Learning and Knowledge Discovery in Databases*. Springer, Berlin, Heidelberg, 2011.

Staal, J., Abramoff, M., Niemeijer, M., Viergever, M., and van Ginneken, B. Ridge based vessel segmentation in color images of the retina. *IEEE Transactions on Medical Imaging*, 23(4): 501–509, 2004.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going Deeper with Convolutions. In *CVPR*, pp. 1–9, June 2015.

Tan, M. and Le, Q. V. EfficientNet: Rethinking model scaling for convolutional neural networks. *ICML*, 2019.

Tan, M., Chen, B., Pang, R., Vasudevan, V., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018.

Tang, Y., Wang, Y., Xu, Y., Chen, H., Shi, B., Xu, C., Xu, C., Tian, Q., and Xu, C. A Semi-Supervised Assessor of Neural Architectures. In *CVPR*, 2020.

Teh, Y., Bapst, V., Czarnecki, W. M., Quan, J., Kirkpatrick, J., Hadsell, R., Heess, N., and Pascanu, R. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 4496–4506, 2017.

Tu, Z. and Bai, X. Auto-Context and Its Applications to High-Level Vision Tasks and 3D Brain Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2009.

Tuzel, O., Porikli, F., and Meer, P. Human Detection via Classification on Riemannian Manifolds. In *CVPR*, pp. 1–8, 2007.

Urooj, A. and Borji, A. Analysis of Hand Segmentation in the Wild. In *Conference on Computer Vision and Pattern Recognition*, 2018.

Valipour, S., Siam, M., Jagersand, M., and Ray, N. Recurrent Fully Convolutional Networks for Video Segmentation. In *IEEE Winter Conference on Applications of Computer Vision*, 2017.

Vapnik, V. *Statistical Learning Theory*. Wiley-Interscience, 1998.

Wah, C., Branson, S., Welinder, P., Perona, P., and Belongie, S. The Caltech-UCSD Birds-200-2011 Dataset. Technical report, 2011.

Wang, L., Zhao, Y., Jinnai, Y., Tian, Y., and Fonseca, R. AlphaX: eXploring Neural Architectures with Deep Neural Networks and Monte Carlo Tree Search. *arXiv preprint arXiv:1903.11059*, 2019.

Wang, L., Xie, S., Li, T., Fonseca, R., and Tian, Y. Neural architecture search by learning action space for monte carlo tree search. *AAAI*, 2020. URL https://openreview.net/forum?id=SklR6aEtwH.

Wang, Q., Li, P., Zuo, W., and Zhang, L. RAID-G - Robust Estimation of Approximate Infinite Dimensional Gaussian with Application to Material Recognition. In *CVPR*, 2016.

Wei, S.-E., Ramakrishna, V., Kanade, T., and Sheikh, Y. Convolutional pose machines. In *Conference on Computer Vision and Pattern Recognition*, 2016.

Welch, B. L. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

Wilson, E. B. and Hilferty, M. M. The distribution of chi-square. *Proceedings of the National Academy of Sciences*, 17(12):684–688, 1931.

Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. *arXiv:1812.03443 [cs]*, December 2018. URL http://arxiv.org/abs/1812.03443. arXiv: 1812.03443.

Wu, Y. and He, K. Group Normalization. In *European Conference on Computer Vision*, 2018.

Xian, K., Shen, C., Cao, Z., Lu, H., Xiao, Y., Li, R., and Luo, Z. Monocular relative depth perception with web stereo data supervision. In *CVPR*, 2018.

Xie, L. and Yuille, A. Genetic cnn. *IEEE International Conference on Computer Vision (ICCV)*, 2017.

Xie, S., Zheng, H., Liu, C., and Lin, L. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

Xie, S., Kirillov, A., Girshick, R., and He, K. Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*, 2019.

Xu, J. and Zhu, Z. Reinforced continual learning. In *NIPS*, 2018.

Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G.-J., Tian, Q., and Xiong, H. PC-DARTS: Partial channel connections for memory-efficient architecture search. In *ICLR*, 2020. URL https://openreview.net/forum?id=BJlS634tPr.

Yang, A., Esperança, P. M., and Carlucci, F. M. NAS evaluation is frustratingly hard. In *ICLR*, 2020. URL https://openreview.net/forum?id=HygrdpVKvr.

Ying, C., Klein, A., Real, E., Christiansen, E., Murphy, K., and Hutter, F. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

You, S., Huang, T., Yang, M., Wang, F., Qian, C., and Zhang, C. GreedyNAS: Towards fast one-shot NAS with greedy supernet. *CVPR*, 2020.

Young, T., Hazarika, D., Poria, S., and Cambria, E. Recent trends in deep learning based natural language processing. *arXiv preprint arXiv:1708.02709*, 2017.

Yu, C., Wang, J., Peng, C., Gao, C., Yu, G., and Sang, N. Bisenet: Bilateral segmentation network for real-time semantic segmentation. In *European Conference on Computer Vision*, 2018.

Yu, J. and Huang, T. S. Network slimming by slimmable networks: Towards one-shot architecture search for channel numbers. *ICLR*, 2019.

Yu, J., Jin, P., Liu, H., Bender, G., Kindermans, P.-J., Tan, M., Huang, T., Song, X., Pang, R., and Le, Q. BigNAS: Scaling Up Neural Architecture Search with Big Single-Stage Models. *ECCV*, 2020a.

Yu, K., Ranftl, R., and Salzmann, M. How to Train Your Super-Net: An Analysis of Training Heuristics in Weight-Sharing NAS. *arXiv*, 2020b.

Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. Evaluating the search phase of neural architecture search. In *ICLR*, 2020c. URL https://openreview.net/forum?id=H1loF2NFwr.

Zela, A., Klein, A., Falkner, S., and Hutter, F. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *ICML AutoML Workshop*, 2018.

Zela, A., Elsken, T., Saikia, T., Marrakchi, Y., Brox, T., and Hutter, F. Understanding and robustifying differentiable architecture search. In *ICLR*, 2020a. URL https://openreview.net/forum?id=H1gDNyrKDS.

Zela, A., Siems, J., and Hutter, F. NAS-Bench-1Shot1: Benchmarking and dissecting one-shot neural architecture search. In *ICLR*, 2020b. URL https://openreview.net/forum?id=SJx9ngStPH.

Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

Zhao, H., Shi, J., Qi, X., Wang, X., and Jia, J. Pyramid Scene Parsing Network. In *Conference on Computer Vision and Pattern Recognition*, 2017.

Zhao, H., Qi, X., Shen, X., Shi, J., and Jia, J. Icnet for real-time semantic segmentation on high-resolution images. In *European Conference on Computer Vision*, pp. 405–420, 2018.

Zhao, Y., Wang, L., Tian, Y., Fonseca, R., and Guo, T. Few-shot neural architecture search. *arXiv*, 2020.

Zhou, H., Yang, M., Wang, J., and Pan, W. Bayesnas: A bayesian approach for neural architecture search. In *ICML*, 2019.

Zoph, B. and Le, Q. V. Neural Architecture Search with Reinforcement Learning. *International Conference on Learning Representations (ICLR), Conference track*, 2017a.

Zoph, B. and Le, Q. V. Neural Architecture Search with Reinforcement Learning. *International Conference on Learning Representations (ICLR), Conference track*, 2017b.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

# Kaicheng YU

BC368, CVLab, Station 14 EPFL, Switzerland, kaicheng.yu@epfl.ch • +41 076 206 53 20

**EDUCATION**

**École Polytechnique Fédérale de Lausanne (EPFL)** — Lausanne, Switzerland
PhD in computer vision and machine learning — Sep 2016 – Present
- Cumulative GPA: 5.5/6.0, fellowship recipient

**The University of Hong Kong (HKU)** — Pok Fo Lam, Hong Kong
Bachelor of Engineering (B.Eng.) in Computer Science — Sep 2012 – Jun 2016
- Cumulative GPA: 3.79/4.3, Major GPA: 4.15/4.3;
- Minor in Statistics, GPA: 4.0/4.3

**SELECTED PUBLICATIONS**

* Indicates equal contribution
- **K. Yu**, R. Ranftl, M. Salzmann, *Landmark Regularization: Ranking Guided Super-Net Training in Neural Architecture Search*, CVPR2021
- **K. Yu**, R. Ranftl, M. Salzmann, *How to train your super-net: An analysis of training heuristics of weight sharing neural architecture search*, in submission of TPAMI
- **K. Yu***, C. Sciuto*, M. Jaggi, C. Musat, M. Salzmann, *Evaluating the Search Phase of Neural Architecture Search*, ICLR2020
- Y. Benyahia*, **K. Yu***, K.B. Smires, M. Jaggi, M. Salzmann, C. Musat, *Overcoming multi-model forgetting*, ICML2019
- W. Wang*, **K. Yu***, J. Hugunot, P. Fua, M. Salzmann, *Recurrent U-Net for Resource-Constrained Segmentation*, ICCV2019
- **K. Yu**, M. Salzmann, *Statistically-motivated second-order pooling*, ECCV2018

**RESEARCH EXPERIENCE**

**Intel Labs** — Munich, Germany
*Research Intern, Intelligent System Lab* — May 2019 – Feb 2020
**Understanding the parameter sharing of neural architecture search (NAS)**
- Supervisor: Dr. Rene Ranftl and Dr. Mathieu Salzmann
- Decoupled the search algorithm with super-net design, isolated 14 independent factors that overlooked in previous research, and conducted a comprehensive analysis
- Improved the random search baseline by a significant margin and achieve state-of-the-art performance on three benchmark spaces
- Leading to two research papers on CVPR'21 and one TPAMI submission

**Swisscom AI Lab** — Lausanne, Switzerland
*Research Assistant* — Sep 2017 – Sep 2018
**One-year joint research project between CVLab-EPFL and Swisscom AI Lab**
- Supervisor: Dr. Claudiu Musat and Dr. Mathieu Salzmann
- Discovered a multi-model forgetting phenomenon in NAS with weight sharing developed a Bayesian-based solution to resolve this effect and improve NAS search results
- Identified many state-of-the-art NAS algorithms are difficult to reproduce, and reveals this is due to the ranking correlation between shared networks and the stand-alone ones are almost zero. It points out a future research direction, and receive attention from the community
- Leading to two research papers on ICML'19 and ICLR'20

**The University of Hong Kong** — Hong Kong
*Summer Research Internship, Computer Science Department* — Jun 2015 – Aug 2015
**Computer-Aided Diagnostic tool: Segmentation and reconstruction of 4D mitral valve echocardiogram.**
- Supervisor: Dr. Kenneth Wong
- In charge of GUI design and implementation of the software, delivered to a local hospital
- Observed the sparse characteristic of mitral valve tissue and proposed a method to reduce data dimension by the Region of Interest, it reduces the classification time of a single volume from 9 hours to 1.8 hours
- Implemented a 3D ortho-slice and animation visualization module via OpenGL and Java3D to facilitate cardiologists interactive review of detailed structure and motion of Mitral Valve

**ACADEMIC HONORS & AWARDS**

Qualcomm Innovation Fellowship Europe — May 2019
- 40,000 USD research funding, 4 out of 56 PhD applicants among top European universities, e.g. Cambridge, ETH, EPFL.

Undergraduate Research Fellowship (Best poster award), HKU — Apr 2015
Dean's Honor Lists, HKU — 2013 – 2016
Soong Ching Ling Hong Kong Scholarship, China Soong Ching Ling Foundation — Sep 2012

**PROGRAMMING LANGUAGE**

Frequently used: Java, Python. Familiar: Bash, C/C++, Haskell, Objective-C, Swift, MATLAB
Packages: Keras, Kubernetes, MatConvNet, TensorFlow, Pytorch, SLURM

**ACADEMIC SERVICE**

Review service: ICCV, CPVR, ECCV, ICLR, ICML, NeurIPS, TPAMI, TIP.
Outstanding reviewer for ECCV2020

**SELECTED PROJECTS**

**Satellite Road Delineation with Fully-Convolutional Network**
- Implemented a fully-convolutional network from scratch in TensorFlow within one week
- Designed a data augmentation technique which rotates data by 45 ° that improves 10% accuracy
- Ranked 3 out of 60 teams in the final competition score board

**4D Mitral Valve Echocardiogram Visualization on iOS Devices with Augmented Reality**
- A mobile solution for cardiologists to facilitate interactive review and discussion about diagnostic result
- Dynamic camera calibration, object detection and pattern recognition with SIFT, motion prediction, OpenGL rendering, Objective-C and Swift programming

**Comparison between Different Models on GIS Forest Cover Types Prediction**
- Data mining project, using Decision Tree, Logistic Regression, Artificial Neural Network and Support Vector Machine to conduct prediction based on 500,000 data set with SAS and R

**Satellite Container Terminal Simulator**
- Software engineering team project, a complex Java simulator of harbor management system produced under Unified Process, with whole set of documentations composed

**TEACHING EXPERIENCE**

**Department of Information and Communication** — EPFL
*Teaching Assistant, Introduction to Machine Learning* — Sep 2020- Jan 2021
*Teaching Assistant, Computer Vision* — Mar 2020- Aug 2020
*Teaching Assistant, Information, Calculation and Communication* — Feb 2017- Jan 2019
*Teaching Assistant, System of data science* — Feb 2018- Jun 2019

**Faculty of Engineering** — The University of Hong Kong
*Teaching Assistant, Object-oriented programming and Java* — Sep 2015 – Dec 2015
*Lab Assistant, Introduction to Electric and Electronics Engineering* — Jan 2014 – Apr 2014

**EXTRA-CURRICULARS**

**BoAo Asia Forum** — Hainan, China
*On-site Event Coordinator* — Apr 2014
- Coordinated hotel service, social media team, local volunteers and security crew to prepare for each conference and maintain order
- One of 21 selected top student volunteers to represent Hong Kong

**Union of Student External Exploration (USEE)** — Hong Kong
*President & Co-founder* — Mar 2013 – Mar 2014
- Co-founded this student society to increase bonding and understanding between mainland and local students
- Organized 2013 Orientation Camp for 80 mainland students new to the university with 2 local student helpers per team, and delivered opening speech
- Dealt with social affairs about funding and negotiated with China Mobile to obtain sponsorship over 10,000 HKD

**Ignite The Dream** — Hong Kong & Shandong, China
*Volunteer teacher* — Dec 2013 – Feb 2014
- Collaborated with volunteers from 4 countries to design oral English materials teaching topics aimed at increasing students' interest and extending their horizon in other culture
- Contributed 2 weeks to teaching activities in 4 classes among three schools of rural area in Shandong

**OTHER WORK EXPERIENCE**

**Division of Information and Technology Studies** — The University of Hong Kong
*Developer, Faculty of Education* — Aug 2014
- Built a Python script to automatically access, retrieve and analyze over 200 students' posts based on authorization then generate a holistic report, based on Facebook Developer API

**Shenda Group** — Shanghai, China
*Android developer, GEAK Electronics* — Jun 2014 – Aug 2014
- Developed a Python test framework and test cases for Bluetooth and Wi-Fi module, which could collect and compare testing results automatically
- Designed and implemented functional smoke tests on Android Wear device and emulator to expose loopholes of core software and hardware functions after each daily update, then report to the developing team and supervisor
- Translated Android documents for the developer and assisted in the composition of company training documents

**LANGUAGES**

English: Proficient (TOEFL 105, GRE 323), Chinese: Native

**INTERESTS**

Tennis, ski, piano.