NEW PERSPECTIVES IN NEURAL ARCHITECTURE SEARCH: ARCHITECTURE EMBEDDINGS, EFFICIENT PERFORMANCE ESTIMATIONS, AND THEIR APPLICATIONS

By

Shen Yan

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science — Doctor of Philosophy

2022

ABSTRACT

Understanding the neural architecture representations and their associated learning curves through theoretical analysis and empirical evaluations is crucial for achieving stable and scalable neural architecture search (NAS). Despite recent advances in one-shot NAS, the stability of search performance remains an issue for users. Sampling-based NAS can eliminate the weights coupling problem by running multiple search trails separately, but it requires significant computational resources. In response, researchers have begun studying architecture representations as a potential solution to the search bias caused by joint optimization of network representations and search methods. These representations are learned to encourage neural architectures with similar structures or computations to cluster together, which helps to map architectures with similar performance to the same regions in the latent space and leads to smoother transitions in the latent space. This benefits downstream search and can be further accelerated by learning curve extrapolation, where the final validation accuracy of a neural network is estimated from the learning curve of a partially trained network.

This dissertation presents our contributions to the field of neural architecture search (NAS), which push the limits of NAS and achieve state-of-the-art performance. Our contributions include efficient one-shot NAS via hierarchical masking [1], addressing the joint optimization problem of architecture representations and search using unsupervised pre-training [2], improving the generalization ability of architecture representations with computation-aware embedding [3], developing a method for facilitating multi-fidelity NAS research and demonstrating the power of using partial learning curve extrapolation [4].

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest appreciation to my advisor, Professor Mi Zhang. Mi has been an inspiring and encouraging mentor, providing profound insights and guidance throughout my research. Not only has he taught me how to pursue top-quality research, but he has also shared invaluable lessons about life with me. I am incredibly grateful for his support and guidance.

During my PhD studies, I would like to express my deep gratitude to several individuals and organizations who have provided support and guidance. This includes Dr. Xuehan Xiong, Dr. Anurag Arnab, Zhichao Lu, Professor Chen Sun, and Professor Cordelia Schmid at Google Research's Perception Team, as well as Dr. Jiahui Yu, Dr. Tao Zhu, Dr. Zirui Wang, Dr. Yuan Cao and Dr. Yonghui Wu at Google Research's Brain Team. I also want to thank Dr. Colin White for hosting me at Abacus.AI, Ming Chen and Youlong Cheng for hosting me at Bytedance AML, Dr. Huan Song, Lincan Zou, and Dr. Liu Ren for hosting me at Bosch Research. Finally, I want to express my deep appreciation to Dr. David Ross and Dr. Rahul Sukthankar at Google Research for their support during the job search process. I am grateful for the valuable experiences and opportunities that these individuals and organizations have provided.

I am deeply grateful for the guidance and support of my master's advisor, Professor Hermann Ney, who introduced me to the world of machine learning and has been a role model for me ever since. I also want to thank Dr. Evgeny Matusov, Dr. Shahram Khadivi, Dr. Daniel Bug, Dr. Harald Hanselmann, and Dr. Abin Jose for their advice during my master's studies. These individuals have played a crucial role in shaping my education and career in machine learning.

I would like to express my deep gratitude to my colleagues and collaborators, who have

provided inspiration and support throughout my research. This includes Yu Zheng, Wei Ao, Dr. Biyi Fang, and Dr. Xiao Zeng. I am also thankful to other supportive individuals, including Dr. Kaiqiang Song, Dr. Yandong Li, Dr. Ji Hou, Dr. Yang Liu, Taojiannan Yang, Lemeng Wu, Dr. Li Erran Li, Professor Chen Chen, Professor Fei Liu, and Professor Frank Hutter. I am grateful for the contributions of these individuals, who have helped make this thesis possible.

I would like to express my appreciation to my committee and faculty members, including Professor Xiaoming Liu, Professor Jiliang Tang, Professor Arun Ross, Professor Sandeep Kulkarni, and Professor Wolfgang Banzhaf. I am grateful for their support and guidance throughout my academic career. These individuals have played a crucial role in shaping my education and research interests.

Lastly, I want to thank my parents, grandparents, and fiancé for their unwavering support and love. Their encouragement has been a constant source of motivation and inspiration throughout my life.

TABLE OF CONTENTS

Chapter 1

Introduction

NAS can be seen as subfield of AutoML, where architecture configuration itself can be viewed as a hyperparameter, and it can be optimized via meta-learning. In [5], NAS is categorized into three dimensions: search space, search strategy, and performance estimation strategy. Figure 1.1 shows an overview the the NAS process, where architecture representations are continuous encodings of the search space, and learning curve extrapolation can be viewed as an speedy performance estimation method.

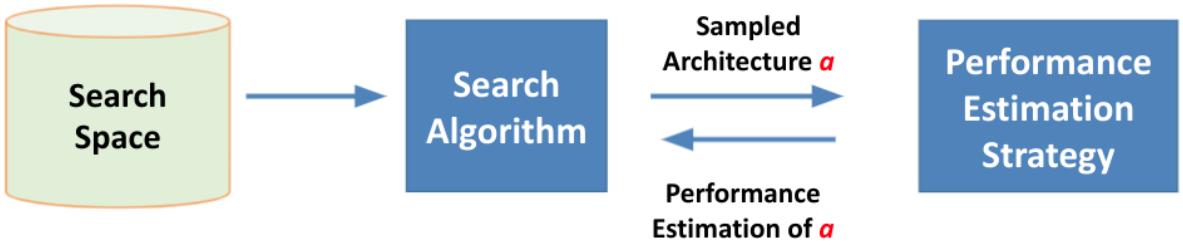


Figure 1.1: A search strategy selects an architecture A from a predefined search space A. The architecture is passed to a performance estimation strategy, which returns the estimated performance of A to the search strategy. Source: [5].

Search Space & Architecture Encoding

The search space defines which architectures and how they can be represented. Two most commonly used search spaces are Inception cell-based [7] and ResNet block-based search space [8]. The cell-based search space consists of two different kind of cells: a normal cell that learns the patterns of the input and a reduction cell which reduces the spatial dimension. The overall architecture is then built by stacking these cells in a repeated manner. The ResNet block-based search space is inspired by MobileNet [9] which is based on an inverted residual structure where the shortcut connections are between the bottleneck layers.

When designing a NAS algorithm, the goal is how should we encode the neural archi-

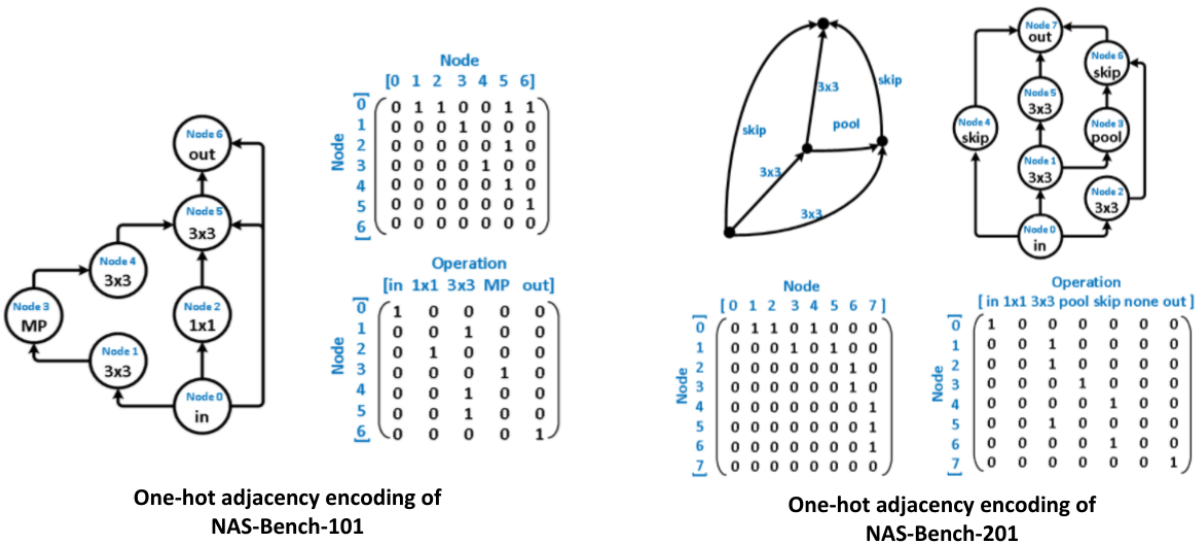


Figure 1.2: The one-hot adjacency matrix encoding is created by flattening the architecture adjacency matrix and concatenating it with a list of node operation labels. Each position in the operation list is a single integer-valued feature or a one-hot matrix.

tecture to maximize performance. Architectures sampled from the same search space share similar encoding properties. The representation of the DAG-based architectures may significantly change the outcome of NAS subroutines such as perturbing or manipulating architectures. In [10], a cell-based architecture is encoded using adjacency matrix-based encoding, categorical encoding, and continuous encoding. Figure 1.2 shows an architecture sampled in NAS-Bench-101 search space [10] and an architecture sampled in NAS-Bench-201 [11] using adjacency encoding, separately. Figure 1.3 an example of the cell encoding in DARTS search space. To convert the DARTS search space [12] into one with the same input format as NAS-Bench-101, we can add a summation node to make nodes represent operations and edges represent data flow. To this end, a 15 $\times$ 15 upper-triangular binary matrix is used to encode edges and a 15 $\times$ 11 one-hot matrix is used to encode operations. Similarly, the categorical adjacency encoding can be derived by first flattening the adjacency matrix, and is then defined as a list of the indices each of which specifies one of the possible edges in the adjacency matrix. Each architecture can be represented by the maximum number of possible

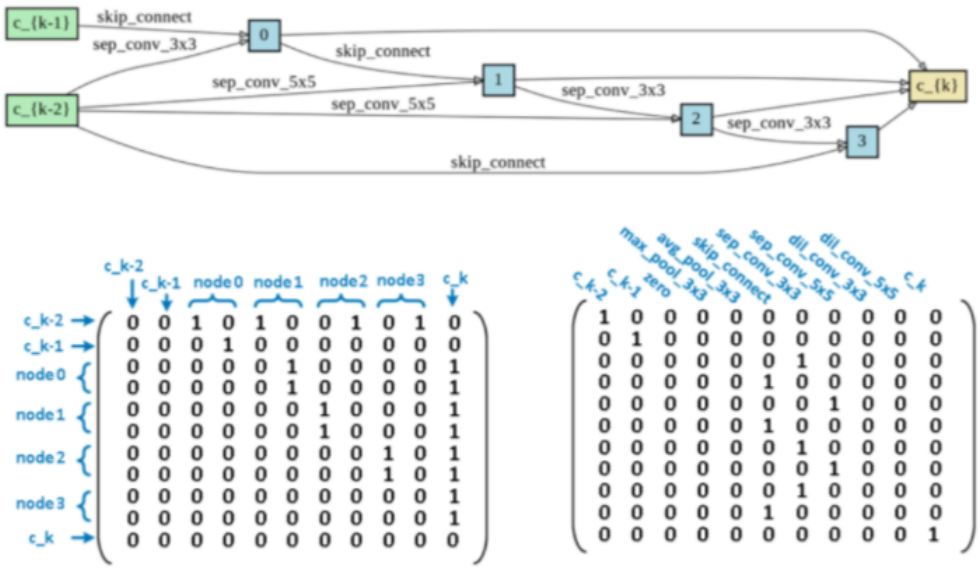


Figure 1.3: A 15 × 15 upper-triangular binary matrix is used to encode edges and a 15 × 11 one-hot matrix is used to encode operations.

edges to ensure a fixed length encoding. The benefit of doing it is that the encoding is not scaled quadratically with increasing nodes. Similarly, the continuous adjacency encoding can be created by taking the fixed number of edges with the highest continuous values, which is similar to encodings used in DARTS [12].

Search Algorithms

The search algorithm describes how to explore and exploit the search space. It is expected to find well-performing architectures quickly, while on the other hand, local convergence to a region of sub-optimal architectures should not be encouraged.

NAS using REINFORCE [13] is the first work that systematically investigated large-scale neural architecture search based on individual sampling process. It encodes neural networks as macro and micro architectures. The densely connected macro network has up to N layers, where each of the layer may have different predecessors. In additional to the direct connection with the previous layers, other connections can also be present or not, resulting in exponential compleixty of the search space. Within each layer, there are multiple

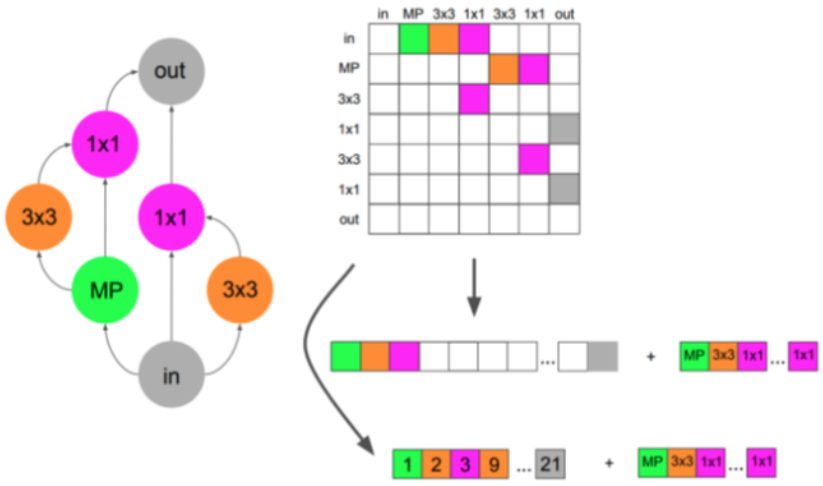


Figure 1.4: The categorical encoding is created to ensure fixed length encodings. Source: [6].

hyperparameter choices, *e.g.* strides, filter height/width and the number of filters. The model is trained using a recurrent neural network (RNN) to sequentially sample a string that in turn encodes the neural architecture, inspired by the success of neural machine translation. The network is optimized with REINFORCE [14] algorithm, but later by Proximal Policy Optimization (PPO) [10].

An alternative to using RL is to use evolutionary algorithms for optimizing the neural architecture. The first such approach for designing neural networks since 1990 [15–17] starting with genetic algorithms. Many neuro-evolutionary approaches [18, 19] since then use genetic algorithms to optimize both the neural architectures. Evolutionary methods differ in how they sample parents, update populations, and generate next generations. For example, [18, 20] use tournament selection to sample parents; [19] sample parents from a multi-objective Pareto front. The worst/oldest individual from a population is removed as a regularization, to escape from the local minimum.

Bayesian Optimization is also one of the popular methods for hyperparameter optimiza-

tion. [21] derive kernel functions for architecture search spaces in order to use classic Gaussian Process (GP) as the surrogate model, but it doesn't perform well in high dimensional space. [22] uses MLP as the surrogate model and is able to perform well on the high dimensional DARTS search space. Furthermore, [3] uses DNGO as the surrogates and shows well-performing results on NASBench101 and DARTS search space.

Performance Estimation

The objective of NAS is typically to find architectures that achieve high predictive performance on unseen data. Performance estimation refers to the process of estimating the performance from either a few training iterations. Techniques include fitting the partial curve to an ensemble of parametric functions [23], predicting the performance based on the partial trained neural network configurations [24], summing the training losses [25], using the basis functions as the output layer of a Bayesian neural network [26], using previous learning curves as basis function extrapolators [27], using the positive-definite covariance kernel to capture a variety of training curves [28], or using a Bayesian recurrent neural network [29]. While in this work we focus on multi-fidelity optimization utilizing learning curve-based extrapolation, another main category of methods lie in bandit-based algorithm selection [30–34], and the fidelities can be further adjusted according to the previous observations or a learning rate scheduler [35–37].

One-shot methods can also be viewed as a promising approach for speeding up NAS due to their computational efficiency [1,12,38–43]. Recent advances in performance prediction [2, 44–50] and other iterative techniques [31,51] have reduced the runtime gap between iterative and weight sharing techniques. For detailed surveys, it is suggested referring to [5, 52].

Thesis Outline

Despite all the aforementioned advantages and potentials, the current NAS methods still have many drawbacks. In this thesis, I will highlight two problems on architecture encodings and speedy performance estimation, namely *how unsupervised architecture representation learning helps downstream search methods*, and *how to efficiently estimate an architecture's performance built upon learning curve extrapolation*. Each chapter is devoted to solving one of these problems. In each chapter, I will describe how to make it more effective and efficient. In brief, this thesis is organized as follows. I start off by providing my first work on one-shot NAS [1] in Chapter 2. I will analysis the drawbacks of this line of research and detail my research on studying architecture representations [2, 3], learning curve extrapolation and its applications [4] in Chapters 3, 4 and 5 respectively. Chapter 6 wraps up and discusses remaining challenges in NAS research.

HM-NAS: Efficient Neural Architecture Search via Hierarchical Masking

The use of automatic methods, often referred to as Neural Architecture Search (NAS), in designing neural network architectures has recently drawn considerable attention. In this work, we present an efficient NAS approach, named HM-NAS, that generalizes existing weight sharing based NAS approaches. Existing weight sharing based NAS approaches still adopt hand designed heuristics to generate architecture candidates. As a consequence, the space of architecture candidates is constrained in a subset of all possible architectures, making the architecture search results sub-optimal. HM-NAS addresses this limitation via two innovations. First, it incorporates a multi-level architecture encoding scheme to enable searching for more flexible network architectures. Second, it discards the hand designed heuristics and incorporates a hierarchical masking scheme that automatically learns and determines

the optimal architecture. Compared to state-of-the-art weight sharing based approaches, HM-NAS is able to achieve better architecture search performance and competitive model evaluation accuracy. *This chapter is based on the following paper [1].*

Does Unsupervised Architecture Representation Learning Help Neural Architecture Search?

Existing Neural Architecture Search (NAS) methods either encode neural architectures using discrete encodings that do not scale well, or adopt supervised learning-based methods to jointly learn architecture representations and optimize architecture search on such representations which incurs search bias. Despite the widespread use, architecture representations learned in NAS are still poorly understood. We observe that the structural properties of neural architectures are hard to preserve in the latent space if architecture representation learning and search are coupled, resulting in less effective search performance. In this work, we find empirically that pre-training architecture representations using only neural architectures without their accuracies as labels improves the downstream architecture search efficiency. To explain this finding, we visualize how unsupervised architecture representation learning better encourages neural architectures with similar connections and operators to cluster together. This helps map neural architectures with similar performance to the same regions in the latent space and makes the transition of architectures in the latent space relatively smooth, which considerably benefits diverse downstream search strategies. *This chapter is based on the paper [2]*, which addresses the drawbacks of joint optimization of architecture representations and search algorithms such as [1].

Computation-aware Architecture Encodings with Transformers

Recent works [2, 6] demonstrate the importance of architecture encodings in Neural Architecture Search (NAS). These encodings encode either structure or computation infor-

mation of the neural architectures. Compared to structure-aware encodings, computation-aware encodings map architectures with similar accuracies to the same region, which improves the downstream architecture search performance [6, 53]. In this work, we introduce a **C**omputation-**A**ware **T**ransformer-based **E**ncoding method called CATE. Different from existing computation-aware encodings based on fixed transformation (e.g. path encoding), CATE employs a pairwise pre-training scheme to learn computation-aware encodings using Transformers with cross-attention. Such learned encodings contain dense and contextualized computation information of neural architectures. We compare CATE with eleven encodings under three major encoding-dependent NAS subroutines in both small and large search spaces. Our experiments show that CATE is beneficial to the downstream search, especially in the large search space. Moreover, the outside search space experiment shows its superior generalization ability beyond the search space on which it was trained. *This chapter is based on the following paper [3]*, which takes inspirations from my earlier work [2].

NAS-Bench-x11 and the Power of Learning Curves

While early research in neural architecture search (NAS) required extreme computational resources, the recent releases of tabular and surrogate benchmarks have greatly increased the speed and reproducibility of NAS research. However, two of the most popular benchmarks do not provide the full training information for each architecture. As a result, on these benchmarks it is not possible to run many types of multi-fidelity techniques, such as learning curve extrapolation, that require evaluating architectures at arbitrary epochs. In this work, we present a method using singular value decomposition and noise modeling to create surrogate benchmarks, NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11, that output the full training information for each architecture, rather than just the final validation accuracy. We demonstrate the power of using the full training information by

introducing a learning curve extrapolation framework to modify single-fidelity algorithms, showing that it leads to improvements over popular single-fidelity algorithms which claimed to be state-of-the-art upon release. *This chapter is based on the following paper [4]*, which builds the first multi-fidelity NAS surrogate benchmark and provides speedy performance estimation given different architecture representations such as [2] and [3].

Summary

In summary, this thesis has touched all aspects of NAS pipeline as illustrated in Figure 1.1, in which NAS can be significantly improved. We hope to convince the reader at the end of this thesis that neural architecture encoding is an important design decision in NAS and it is able to significantly improve the efficiency of NAS by leveraging the power of partial learning curves. Still, there are many challenging and rewarding problems to be explored which I will summarize in the conclusion chapter 6. *The material is based on the AutoML workshop tutorial at ICML'2021.*[1]

All code, data, and models used in this thesis can be found at https://github.com/MSU-MLSys-Lab.

[1]The tutorial website is at `https://sites.google.com/corp/view/automl2021`.

Chapter 2

Efficient One-Shot NAS via Hierarchical Masking

Neural architecture search (NAS) has recently attracted significant interests due to its capability of automating neural network architecture design and its success in outperforming hand-crafted architectures in many important tasks such as image classification [54], object detection [55], and semantic segmentation [56]. In early NAS approaches, architecture candidates are first sampled from the search space; the weights of each candidate are learned independently and are discarded if the performance of the architecture candidate is not competitive [18, 54, 57, 58]. Despite their remarkable performance, since each architecture candidate requires a full training, these approaches are computationally expensive, consuming hundreds or even thousands of GPU days in order to find high-quality architectures. To overcome this bottleneck, a majority of recent efforts focuses on improving the computation efficiency of NAS using the *weight sharing* strategy [8, 12, 57, 59, 60]. Specifically, rather than training each architecture candidate independently, the architecture search space is encoded within a single over-parameterized *supernet* which includes all the possible connections (i.e., wiring patterns) and operations (*e.g.*, convolution, pooling, identity). The supernet is trained *only once*. All the architecture candidates inherit their weights directly from the supernet without training from scratch. By doing this, the computation cost of NAS is significantly reduced. Unfortunately, although the supernet subsumes all the possible architecture candidates, existing weight sharing based NAS approaches still adopt *hand designed heuristics* to extract architecture candidates from the supernet. As an example, in many existing weight sharing based NAS approaches such as DARTS [12], the supernet is organized as stacked cells and each cell contains multiple nodes connected with edges. However, when extracting architecture candidates from the supernet, each candidate is hard coded to have *exactly* two

input edges for each node with *equal importance* and to associate each edge with *exactly* one operation. As such, the space of architecture candidates is constrained in a subset of all possible architectures, making the architecture search results sub-optimal. Given the constraint of existing weight sharing approaches, it is natural to ask the question: *will we be able to improve architecture search performance if we loosen this constraint?* To this end, we present HM-NAS, an efficient neural architecture search approach that effectively addresses such limitation of existing weight sharing based NAS approaches to achieve better architecture search performance and competitive model evaluation accuracy. As illustrated in Figure 2.1, to loosen the constraint, HM-NAS incorporates a *multi-level architecture encoding* scheme which enables an architecture candidate extracted from the supernet to have arbitrary numbers of edges and operations associated with each edge. Moreover, it allows each operation and edge to have different weights which reflect their relative importance across the entire network. Based on the multi-level encoded architecture, HM-NAS formulates neural architecture search as a model pruning problem: it discards the hand designed heuristics and employs a *hierarchical masking* scheme to automatically learn the optimal numbers of edges and operations and their corresponding importance as well as mask out unimportant network weights. Moreover, the addition of these learned hierarchical masks on top of the supernet also provides a mechanism to help correct the architecture search bias caused by bilevel optimization of architecture parameters and network weights during supernet training [38, 61, 62]. Because of such benefit, HM-NAS is able to use the unmasked network weights to speed up the training process. We evaluate HM-NAS on both CIFAR-10 and ImageNet and our results are promising: HM-NAS is able to achieve competitive accuracy on CIFAR-10 with 1.6× to 1.8× less parameters and 2.7× total training time speed-up compared with state-of-the-art weight sharing approaches. Similar results are also achieved

on ImageNet. Moreover, we have conducted a series of ablation studies that demonstrate the superiority of our multi-level architecture encoding and hierarchical masking schemes over randomly searched architectures, as well as single-level architecture encoding and hand designed heuristics used in existing weight sharing based NAS approaches. Finally, we have conducted an in-depth analysis on the best-performing network architectures found by HM-NAS. Our results show that without the constraint imposed by the hand designed heuristics, our searched networks contain more flexible and meaningful architectures that existing weight sharing based NAS approaches are not able to discover. In summary, our work makes the following contributions: 1). We present HM-NAS, an efficient neural architecture search approach that loosens the constraint of existing weight sharing based NAS approaches. 2). We introduce a multi-level architecture encoding scheme which enables an architecture candidate to have arbitrary numbers of edges and operations with different importance. We also introduce a hierarchical masking scheme which is able to not only automatically learn the optimal numbers of edges, operations and important network weights, but also help correct the architecture search bias caused by bilevel optimization during supernet training. 3). Extensive experiments show that compared to state-of-the-art weight sharing based NAS approaches, HM-NAS is able to achieve better architecture search efficiency and competitive model evaluation accuracy.

Related Work

Designing high-quality neural networks requires domain knowledge and extensive experiences. To cut the labor intensity, there has been a growing interest in developing automated neural network design approaches through NAS. Pioneer works on NAS employ reinforcement learning (RL) or evolutionary algorithms to find the best architecture based on nested optimization [18, 54, 57, 58]. However, these approaches are incredibly expensive in terms

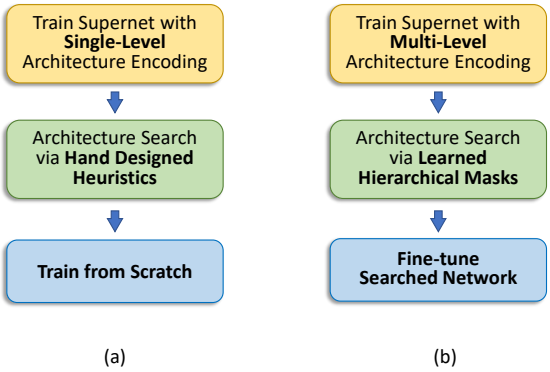


Figure 2.1: The pipelines of (a) existing weight sharing based NAS approaches; and (b) HM-NAS.

of computation cost. For example, in [54], it takes 450 GPUs for four days to search for the best network architecture. To reduce computation cost, many works adopt the weight sharing strategy where the weights of architecture candidates are inherited from a supernet that subsumes all the possible architecture candidates. To further reduce the computation cost, recent weight sharing based approaches such as DARTS [12] and SNAS [59] replace the discrete architecture search space with a continuous one and employ gradient descent to find the optimal architecture. However, these approaches restrict the continuous search space with hand designed heuristics, which could jeopardize the architecture search performance. Moreover, as discussed in [38,61,62], the bilevel optimization of architecture parameters and network weights used in existing weight sharing based approaches inevitably introduces bias to the architecture search process, making their architecture search results sub-optimal. Our approach is related to DARTS and SNAS in the sense that we both build upon the weight sharing strategy. However, our goal is to address the above limitations of existing approaches to achieve better architecture search performance.

Our approach is also related to ProxylessNAS [8]. ProxylessNAS formulates NAS as a model pruning problem. In our approach, the employed hierarchical masking scheme also prunes the redundant parts of the supernet to generate the optimal network architecture.

NAS Approach	Architecture Encoding	Retrain from Scratch	Use Proxy
ENAS [57]	Operations	Yes	Yes
NASNet [54]	Operations	Yes	Yes
AmoebaNet [18]	Operations	Yes	Yes
NAONet [63]	Operations	Yes	Yes
ProxylessNAS [8]	Operations	Yes	No
FBNet [64]	Operations	Yes	No
DARTS [12]	Operations	Yes	Yes
SNAS [59]	Operations	Yes	Yes
HM-NAS	**Operations & Edges**	**No**	**No**

Table 2.1: Comparison between HM-NAS and other NAS approaches.

The distinction is that ProxylessNAS focuses on pruning operations (referred to as path in [8]) of the supernet, while HM-NAS provides a more generalized model pruning mechanism which prunes the redundant operations, edges, and network weights of the supernet to derive the optimal architecture. Our approach is also similar to ProxylessNAS as being a proxyless approach. Rather than adopting a proxy strategy like [12, 59], which transfers the searched architecture to another larger network, both HM-NAS and ProxylessNAS directly search the architectures on target datasets without architecture transfer. However, unlike ProxylessNAS which involves retraining as the last step, HM-NAS eliminates the prolonged retraining process and replaces it with a fine-tuning process with the reuse of the unmasked pretrained supernet weights. Table 2.1 provides a comparison between HM-NAS and relevant approaches on a number of important dimensions. The combination of the proposed multi-level architecture encoding and hierarchical masking techniques makes HM-NAS superior over many existing approaches.

Approach

Supernet Design

Following [12, 18, 59], we use a cell structure with an ordered sequence of nodes as our search space. The network is then composed of several identical cells which are stacked on top of each other. Specifically, a cell is represented using a directed acyclic graph (DAG) where each node x in the DAG is a latent representation (*e.g.*, a feature map in a convolutional network). A cell is set to have two input nodes, one or more intermediate nodes, and one output node. Specifically, the two input nodes are connected to the outputs of cells from two previous cells; each intermediate node is connected by all its predecessors; and the output node is the concatenation of all the intermediate nodes within the cell. To build the supernet that subsumes all the possible architectures in the search space, existing works such as DARTS [12] and SNAS [59] associate each edge in the DAG with a mixture of candidate operations (*e.g.*, convolution, pooling, identity) instead of a definite one. Moreover, each candidate operation of the mixture is assigned with a learnable variable (*i.e.*, operation mixing weight) which encodes the importance of this candidate operation. As such, the mixture of candidate operations associated with a particular edge is represented as the softmax over all candidate operations:

$$\overline{o}(x) = \sum_{i=1}^{N} \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)} o_i(x) \tag{2.1}$$

where $\{o_i\}$ denote the set of N candidate operations, $\{\alpha_i\}$ denote the set of N real-valued operation mixing weights. Although this supernet encodes the importance of different candidate operations within each edge, *it does not provide a mechanism to encode the importance of different edges across the entire DAG*. Instead, all the edges across the DAG are constrained to have the same importance. However, as we observed in our experiments (Section

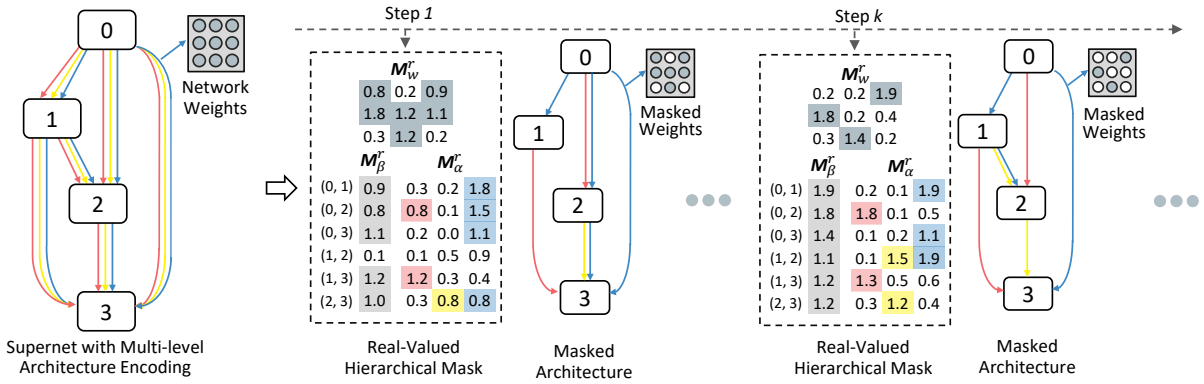


Figure 2.2: In this example, each edge has 3 candidate operations marked using red, yellow, and blue color respectively. In each iteration, the real-valued hierarchical masks $\{\boldsymbol{M}_\alpha^r, \boldsymbol{M}_\beta^r, \boldsymbol{M}_w^r\}$ are passed through a deterministic thresholding function to obtain the corresponding binary masks (highlighted grids represent '1', the rest represent '0') that mask out redundant operations, edges, and weights of the supernet.

2.5), loosening this constraint is able to help NAS find better architectures. Motivated by this observation, in HM-NAS's supernet, besides encoding the importance of each candidate operation within an edge, we introduce a separate set of learnable variables (*i.e.*, edge mixing weights) to *independently encode the importance of each edge across the DAG*. As such, each intermediate node $x^{(i)}$ in the DAG is computed based on all of its predecessors as:

$$x^{(i)} = \sum_{j<i} \frac{\exp(\beta^{(i,j)})}{\sum_{k<i} \exp(\beta^{(i,k)})} \overline{o}(x^{(j)}) \tag{2.2}$$

where $\beta^{(i,j)}$ denote the real-valued edge mixing weight for the directed edge (i, j). In summary, $\boldsymbol{\alpha} = \{\alpha_i\}$ encode the architecture at the *operation level* while $\boldsymbol{\beta} = \{\beta^{(i,j)}\}$ encode the architecture at the *edge level*. Therefore, we have constructed a supernet with *multi-level architecture encoding* where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ altogether encode the overall architecture of the network, and we refer to $\{\boldsymbol{\alpha}, \boldsymbol{\beta}\}$ as the *architecture parameters*.

Training the Supernet

To train the multi-level encoded supernet, we follow [12] to jointly optimize the architecture parameters $\{\boldsymbol{\alpha}, \boldsymbol{\beta}\}$ and the network weights $\boldsymbol{w}$ in a bilevel way via stochastic gradient descent with first or second-order approximation. Let $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$ denote the training loss and validation loss respectively. The goal is to find $\{\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*\}$ that minimize $\mathcal{L}_{val}(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{w}^*)$, where $\boldsymbol{w}^*$ is obtained by minimizing the training loss $\boldsymbol{w}^* = \arg\min_w \mathcal{L}_{train}(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*, \boldsymbol{w})$. For the details of this bilevel optimization, please refer to [12] as we do not claim any new contribution on this part.

Here we want to emphasize two techniques that we find helpful in training our multi-level encoded supernet. First, due to insufficient training of network weights $\boldsymbol{w}$ at the beginning of the supernet training, the architecture parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ could be randomly selected. To avoid this, similar to [64], we adopt a warm start in training of $\boldsymbol{w}$ while freezing the training of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. Second, updating $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ too frequently could lead to underfitting of $\boldsymbol{w}$. We solve this by triggering the optimization of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ stochastically rather than doing it constantly, with a probability of $p = \sigma(iter)$, where *iter* is the number of iterations and $\sigma(\cdot)$ is a monotonically non-increasing function that satisfies $\sigma(0) = 1$. After a prolonged decrease, the probability p may even be set to zero, *i.e.*, no bilevel optimization is conducted any longer and only $\boldsymbol{w}$ is optimized.

Hierarchical Masking

Given the trained supernet, we formulate neural architecture search as a model pruning problem, and iteratively prune the redundant *operations*, *edges*, and *network weights* of the supernet in a hierarchical manner to derive the optimal architecture through a scheme which we refer to as *hierarchical masking*. Figure 2.2 illustrates the iterative hierarchical mask-

ing process on a single cell. Specifically, we begin with the trained supernet as our base network, and initiate three types of real-valued masks for operations, edges, and network weights, respectively. These masks are passed through a deterministic thresholding function to obtain the corresponding binary masks. These generated binary masks are then elementwisely multiplied with the architecture parameters $\{\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*\}$ and network weights $\boldsymbol{w}^*$ of the supernet to generate a searched network. By iteratively training the real-valued masks through backpropagation combined with network binarization techniques [65] in an end-to-end manner, the binary masks learned in the end are able to mask out redundant operations, edges, and network weights in the supernet to derive the optimal architecture. Formally, let $\boldsymbol{M}^r = \{\boldsymbol{M}^r_\alpha, \boldsymbol{M}^r_\beta, \boldsymbol{M}^r_w\}$ denote the real-valued hierarchical masks, where $\boldsymbol{M}^r_\alpha, \boldsymbol{M}^r_\beta, \boldsymbol{M}^r_w$ is the real-valued mask for operations, edges, and network weights, respectively. Architecture search is then reduced to finding $\boldsymbol{M}^{r*}$ which minimizes the training loss of the masked supernet:

$$\boldsymbol{M}^{r*} = \arg\min_{\boldsymbol{M}^r} \mathcal{L}(\mathcal{P}_{\boldsymbol{M}}(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*, \boldsymbol{w}^*)) \tag{2.3}$$

$$\boldsymbol{M} = H(\boldsymbol{M}^r - \tau) \tag{2.4}$$

where $\boldsymbol{M} = \{\boldsymbol{M}_\alpha, \boldsymbol{M}_\beta, \boldsymbol{M}_w\}$ are the corresponding binary masks, $H(\cdot)$ is the Heaviside step function as the deterministic thresholding function, τ is the pre-defined threshold, and $\mathcal{P}(\cdot)$ is the elementwise projection function. In this work, we use elementwise multiplication for $\mathcal{P}(\cdot)$.

Even though the Heaviside step function in (2.4) is non-differentiable, we adopt the approximation strategy used in BinaryConnect [65] to approximate the gradients of real-valued masks $\boldsymbol{M^r}$ using the gradients of the binary masks $\boldsymbol{M}$, and thus update the real-

valued masks $\boldsymbol{M^r}$ using the gradients of the binary masks $\boldsymbol{M}$. As shown in prior works [65–67], this strategy is effective because the gradients of $\boldsymbol{M}$ actually act as a regularizer or a noisy estimator of the gradients of $\boldsymbol{M^r}$. By doing this, the binary masks can be trained in an end-to-end differentiable manner.

Deriving the Final Model

The hierarchical masking process in §2 outputs not only the optimal network architecture but also a set of optimized network weights. As such, we can derive the final model via fine-tuning instead of retraining the searched architecture from scratch.

With the optimized network weights, the searched architecture is able to maintain comparable accuracy compared to the supernet (*e.g.*, $\sim$1% loss on CIFAR-10), and thus acts as a significantly better starting point for fine-tuning. This not only ensures higher accuracy, but also replaces the prolonged retraining process with a more efficient fine-tuning process.

Experiments

We evaluate the performance of HM-NAS and compare it with state-of-the-arts NAS approaches on two benchmark datasets: CIFAR-10 (Section 17) and ImageNet (Section 17).

Moreover, we have conducted a series of ablation studies that validate the importance and effectiveness of the proposed multi-level architecture encoding scheme and hierarchical masking scheme incorporated in the design of HM-NAS (Section 17).

Experimental Setup

We use 3 cells and 36 initial channels to build the supernet for CIFAR-10, and 5 cells and 24 initial channels for ImageNet. Following DARTS [12], our cell consists of 7 nodes in all the experiments. The input nodes, *i.e.*, the first and second nodes of cell k is the output of cell $k-1$ and $k-2$, respectively. The output node is the depthwise concatenation of all the intermediate nodes. We include the following operations: 3×3 and 5×5 separable

Algorithm 1: HM-NAS

Input: multi-level architecture encoded supernet $\boldsymbol{\Theta}(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{w})$, real-valued masks $\boldsymbol{M}_\alpha^r$, $\boldsymbol{M}_\beta^r$, $\boldsymbol{M}_w^r$, threshold τ, deterministic thresholding function $H(\cdot)$, elementwise projection function $\mathcal{P}(\cdot)$

Output: $\left\{\widehat{\boldsymbol{\Theta}}(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*, \boldsymbol{w}^*), \boldsymbol{M}_\alpha^*, \boldsymbol{M}_\beta^*, \boldsymbol{M}_w^*\right\}$: the optimized searched model and binary masks

`// supernet training`

Initialize $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha^0}$, $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta^0}$, $\boldsymbol{w} \leftarrow \boldsymbol{w^0}$, $t \leftarrow 0$.

while *not converge* **do**

  Update $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$ by descending $\nabla_{\boldsymbol{\beta}} \mathcal{L}_{val}(\boldsymbol{\alpha^*}, \boldsymbol{\beta}, \boldsymbol{w^*})$ and $\nabla_{\boldsymbol{\alpha}} \mathcal{L}_{val}(\boldsymbol{\alpha}, \boldsymbol{\beta^*}, \boldsymbol{w^*})$ with a probability of $\sigma(t)$

  Update $\boldsymbol{w}$ by descending $\nabla_{\boldsymbol{w}} \mathcal{L}_{train}(\boldsymbol{\alpha^*}, \boldsymbol{\beta^*}, \boldsymbol{w})$

  $t \leftarrow t + 1$

end

`// searching via hierarchical masking`

Initialize $\boldsymbol{M}_\alpha^r \leftarrow \boldsymbol{M}_\alpha^0$, $\boldsymbol{M}_\beta^r \leftarrow \boldsymbol{M}_\beta^0$, $\boldsymbol{M}_w^r \leftarrow \boldsymbol{M}_w^0$

while *not converge* **do**

  Feed forward and loss calculation with $\mathcal{P}_{H(\boldsymbol{M}_w^r - \tau)}(\boldsymbol{w^*})$, $\mathcal{P}_{H(\boldsymbol{M}_\alpha^r - \tau)}(\boldsymbol{\alpha}^*)$, $\mathcal{P}_{H(\boldsymbol{M}_\beta^r - \tau)}(\boldsymbol{\beta}^*)$

  Update $\boldsymbol{M}^r$ by descending $\nabla_{H(\boldsymbol{M}^r - \tau)} \mathcal{L}_{train}(\mathcal{P}_{\boldsymbol{M}}(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*, \boldsymbol{w^*}))$

end

`// fine-tuning the searched network`

Initialize $\boldsymbol{w} \leftarrow \boldsymbol{w^*}$. Construct searched network $\widehat{\boldsymbol{\Theta}}(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*, \boldsymbol{w})$ masked by $\boldsymbol{M}_\alpha^*, \boldsymbol{M}_\beta^*, \boldsymbol{M}_w^*$.

while *not converge* **do**

  Update unmasked $\boldsymbol{w}$ by descending $\nabla_{\boldsymbol{w}} \mathcal{L}_{train}(\mathcal{P}_{\boldsymbol{M}^*}(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*, \boldsymbol{w}))$

end

convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max pooling and average pooling, and identity. ReLU-Conv-BN triplet is adopted for convolutional operations except the first convolutional layer (Conv-BN), and each separable convolution is applied twice. The default stride is 1 for all operations unless the output size is changed. The experiments are conducted using a single NVIDIA Tesla V100 GPU.

Results on CIFAR-10

Training Details. We begin with training the supernet for 100 epochs with batch size 128. In each epoch, we first train weights $\boldsymbol{w}$ on 80% of the training set using SGD with momentum. The initial learning rate is 0.1 with decay following a cosine decaying schedule. The momentum is 0.9 and weight decay is 3e-4. Architecture parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ are randomly

initialized and scaled by 1e-3. Next, We train $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ on the rest 20% of the training set with Adam optimizer [68] with the learning rate of 3e-4 and weight decay of 1e-3. We empirically observe more stable training process when using Adam for optimizing the architecture parameters, which is also used in [12]. Following [64], we postpone the training of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ by 10 epochs to warm up $\boldsymbol{w}$ first. The supernet training takes 7.5 hours (or 30 hours for the second-order approximation). Once the supernet is trained, we perform 20 epochs of neural architecture search via hierarchical masking using the entire training set. Hierarchical masks $\boldsymbol{M}_\alpha^r, \boldsymbol{M}_\beta^r, \boldsymbol{M}_w^r$ are initialized as 1e-2. They are trained using the Adam optimizer with an initial learning rate of 1e-4 for $\boldsymbol{M}_w^r$ and 1e-5 for $\boldsymbol{M}_\alpha^r$ and $\boldsymbol{M}_\beta^r$, which is decayed by a factor of 10 after 10 epochs. The binarizer threshold τ (Equation 2.4) is 5e-3[1]. The hierarchical mask training takes 3.5 hours. Lastly, the masked network is fine-tuned for 200 epochs for 9.6 hours with cutout [69].

Architecture Evaluation. Table 2.2 shows our evaluation results on CIFAR-10 where 'c/o' denotes cutout adapted from [69]. The test error of HM-NAS is on par with state-of-the-art NAS methods. Notably, HM-NAS achieves this by using the *fewest* parameters among all methods. Specifically, HM-NAS only uses 1.8M parameters, which is 1.4× to 3.2× fewer compared to others.

Performance at Different Training Stages. Table 2.3 breaks down the complete architecture search process of HM-NAS and shows the performance of HM-NAS at different stages. Specifically, compared to the supernet, the searched network (derived after hierarchical masking) loses only ∼1% accuracy with 40% less parameters. Although directly using this searched network is not optimal (with test error 5.14%), it does provide a good initialization for fine-tuning, which leads to lower test error (from 5.14% to 2.41%).

[1] Our method is robust to thresholds in the range of [0, 1e-2].

Architecture	Test Error (%)	Params (M)	Search Cost (GPU days)	Train Cost (GPU days)	Total Cost (GPU days)	Search Method
DenseNet-BC [70]	3.46	25.6	-	-	-	manual
ENAS + c/o [57]	2.89	4.6	**0.45**	(630 epochs)†	-	RL
NASNet-A + c/o [54]	2.65	3.3	3150	-	-	RL
SNAS + c/o [59]	2.85	2.8	1.5	1.5 (600 epochs)	3	gradient-based
ProxyLess-G + c/o	**2.08**	5.7	4*	(600 epochs)	-	gradient-based
AmoebaNet-A + c/o [18]	3.34 ± 0.06	3.2	3150	-	-	evolution
AmoebaNet-B + c/o [18]	2.55 ± 0.05	2.8	3150	-	-	evolution
DARTS (2nd order) + c/o [12]	2.76 ± 0.09	3.3	4*	2† (600 epochs)	6	gradient-based
HM-NAS (1st order) + c/o	2.78 ± 0.07	**1.8**	**0.45**	**0.4** (200 epochs)	**0.85**	gradient-based
HM-NAS (2nd order) + c/o	2.41 ± 0.05	**1.8**	1.4	**0.4** (200 epochs)	**1.8**	gradient-based

* Results obtained from authors' official response in openreview.

† Results obtained using code publicly released by the authors.

Table 2.2: Comparison with state-of-the-arts on CIFAR-10.

Architecture	Test Error (%)	Params (M)	Params Reduction (%)
Supernet	4.2	3	-
Searched Network	5.14	1.8	40
Searched Network + Fine-Tuning	**2.41**	**1.8**	**40**

Table 2.3: Performance of HM-NAS at different architecture search stages.

Architecture Search Cost Analysis. To find the optimal architecture, HM-NAS only uses 0.85 or 1.8 GPU days, which is significantly faster compared to all other NAS methods. To understand why HM-NAS is efficient, we compare the complete architecture search process of HM-NAS to DARTS. Figure 2.3 illustrates the training curve of HM-NAS (in blue color) and DARTS[2] (in red color) during the complete architecture search process on CIFAR-10. Specifically, both HM-NAS and DARTS use the first 100 epochs to train the supernet with the same train/validation dataset split. Due to multi-level architecture encoding, HM-NAS is able to achieve better test results after 100 epochs. Then, DARTS transfers the learned cell to build a larger network and retrains it from scratch. This process

[2]Our implementation based on the code released by the authors.

takes approximately 600 epochs to converge. In contrast, from 100 epoch to 120 epoch and onward, HM-NAS performs architecture search via hierarchical masking and fine-tuning, respectively. This process only takes 220 epochs to converge, which is 2.7× faster compared to DARTS.

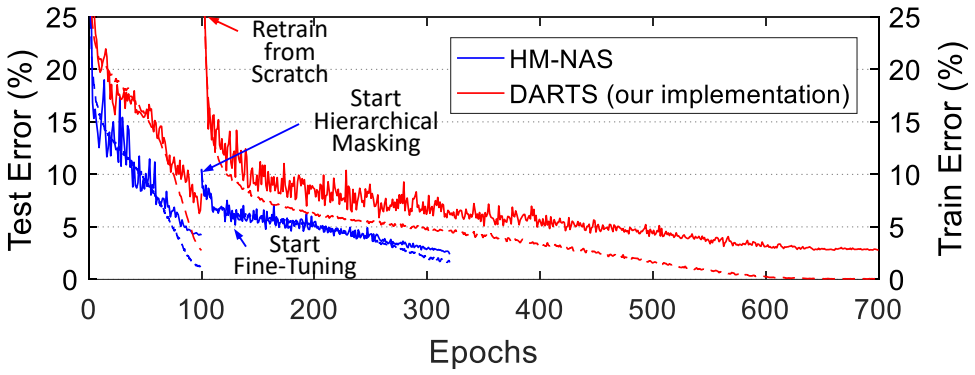


Figure 2.3: Training curves of HM-NAS (in blue color) and DARTS (in red color) on CIFAR-10. Solid lines denote test errors (y-axis on the left); dashed lines denote training errors (y-axis on the right).

Results on ImageNet

We conduct experiments on ImageNet 1000-class [71] classification task, where input image size is 224×224. The dataset has around 1.28M training images and we test on the 50k validation images. **Training Details.** We adopt the small computation regime (*e.g.*, MobileNet-V1 [72]) in the experiments. Following [64], 100 classes from the original 1,000 classes of ImageNet is randomly sampled to train the supernet for 100 epochs with batch size 128. It takes around one GPU day to finish the supernet training. Once the supernet is trained, the hierarchical masking is then performed with the same optimization settings mentioned in §17. The hierarchical masking process takes around one GPU day to finish. Lastly, the searched network is fine-tuned on the entire ImageNet training dataset (with 1,000 classes) for 60 epochs with initial learning rate 1e-2 then decreased to 1e-3 at the epoch 30. This phase takes around 3 GPU days to finish.

Architecture	Top-1 Acc. (%)	Params (M)	FLOPs (M)
MobileNet-V1 [72]	70.6	4.2	569
MobileNet-V2 [9]	**74.7**	6.9	585
NASNet-A [54]	74.0	5.3	564
Amoeba-A [18]	74.5	5.1	555
DARTS [12]	73.3	4.7	574
SNAS [59]	72.7	4.3	522
HM-NAS	73.4	**3.6**	**482**

Table 2.4: Comparison with state-of-the-arts on ImageNet.

Architecture Evaluation. Table 2.4 shows our evaluation results on ImageNet. The result is comparable to DARTS, considering that we adopt the exact same search space in DARTS [12], which uses the operations incorporated in MobileNet-V1 [72]. Notably, we achieve comparable results to the state-of-the-art gradient-based NAS approaches [12, 59] with 1.2× to 1.3× less parameters and 1.08× to 1.2× less FLOPs. With a larger supernet and better candidate operations such as the ones used in MobileNet-V2 [9], We believe that the results could be further improved.

Analysis

In this section, we conduct a series of ablation studies to demonstrate the superiority of the design of HM-NAS. The ablation studies are conducted on CIFAR-10 with second-order derivative introduced in Table 2.2. **Comparison to Single-Level Architecture Encoding.** To demonstrate the superiority of the proposed multi-level architecture encoding scheme over single-level architecture encoding, we compare the single-level encoded network against the multi-level encoded network, both with hand designed heuristics (by replacing each mixed operation with the most likely operation and taking the top-2 confident edges from distinct nodes). As shown in Table 2.5, the multi-level architecture encoding achieves

Architecture Encoding	Derived Rule	Test Error (%)	Params (M)
Single-Level ($\boldsymbol{\alpha}$)	Hand Designed Heuristics	3.1	2.5
Multi-Level ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)	Hand Designed Heuristics	2.7	2.1
Multi-Level ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)	Learned Hierarchical Masks	**2.41**	**1.8**

Table 2.5: Comparison to hand designed heuristics.

2.7% test error, giving 0.4% accuracy improvement over the single-level one.

Comparison to Hand Designed Heuristics. To demonstrate the superiority of learned hierarchical masks over hand designed heuristics, we compare the multi-level encoded network with learned hierarchical masks against the one with hand designed heuristics. As shown in Table 2.5, the hierarchical masks achieve 2.41% test error, providing about 0.3% accuracy improvement over hand designed heuristics. Similar findings are also observed in [61]. In contrast, HM-NAS outperforms the random architecture by 1% in test error with 3× less training epochs. This is because with multi-level architecture encoding and hierarchical masking, the search space is significantly enlarged, making it challenging for random search to find a competitive network.

Comparison to Random Initialization. As our final ablation study, to demonstrate the superiority of unmasked network weights obtained from hierarchical masking over random weights, we *randomly initialize* the weights of the searched network (same architecture as HM-NAS) and train it for 600 epochs on par with the training setup in DARTS. We run 5 times of random initialization, each running the same number of epochs. As shown in Table 2.6, the average test error of random initialization is 2.95%, which is comparable to DARTS but considerably higher than HM-NAS.

Searched Architecture Analysis. Finally, we provide an in-depth analysis on the net-

Architecture	Test Error (%)	Params (M)	Train Cost (epochs)
Random Architecture	3.41 ± 0.15	2.1	600
Random Initialization ‡	2.95 ± 0.08	1.8	600
HM-NAS	**2.41** ± 0.05	**1.8**	**200**

‡ Same architecture as HM-NAS + c/o with random initialized weights.

Table 2.6: Comparison to random architectures and random initialization.

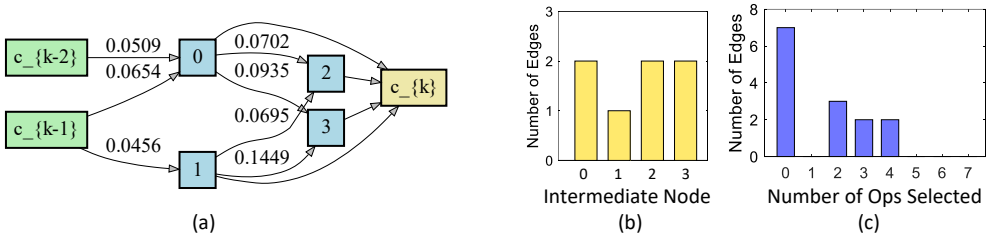


Figure 2.4: (a) cell structure. (b) number of input edges of four intermediate nodes. (c) histogram of the number of edges w.r.t the number of operations selected.

work architecture found by HM-NAS. We have the following three important observations.

Different Learned Importance for Different Edges. Figure 2.4(a) and Figure 2.5(a) illustrate the details of the learned cell for CIFAR-10 and ImageNet respectively, where the importance of edge, *i.e.*, edge mixing weight $\beta^{(i,j)}$, is marked above every edge. Unlike DARTS in which each edge has the *same hard-coded* importance, due to multi-level architecture encoding, the best-performing cell found by HM-NAS has *different learned* importance for different edges across the cell. Moreover, we find that edges connecting to later intermediate nodes have higher importance than early intermediate nodes. One possible explanation is that during cell construction, each intermediate node is ordered and is derived from its predecessors by accumulating information passed from its predecessors. Hence, it has more influence on the output of the cell, which is reflected by the higher importance learned through our approach. *Once the importance of the edge is no longer heuristically*

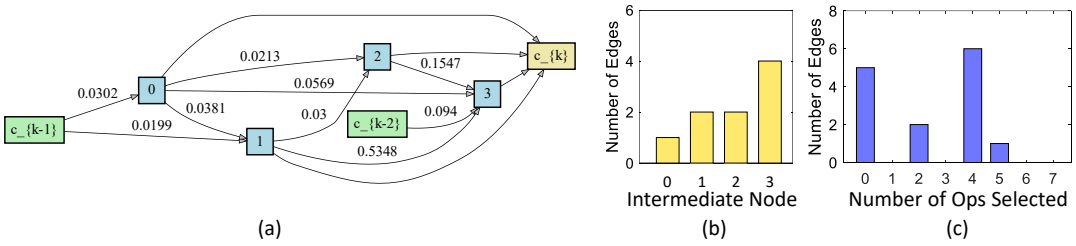


Figure 2.5: (a) cell structure. (b) number of input edges of four intermediate nodes. (c) histogram of the number of edges w.r.t the number of operations selected.

determined but automatically learned, the multi-level architecture encoding provides a more flexible way to encode the entire supernet architecture and thus provides us with a better superset for architecture search.

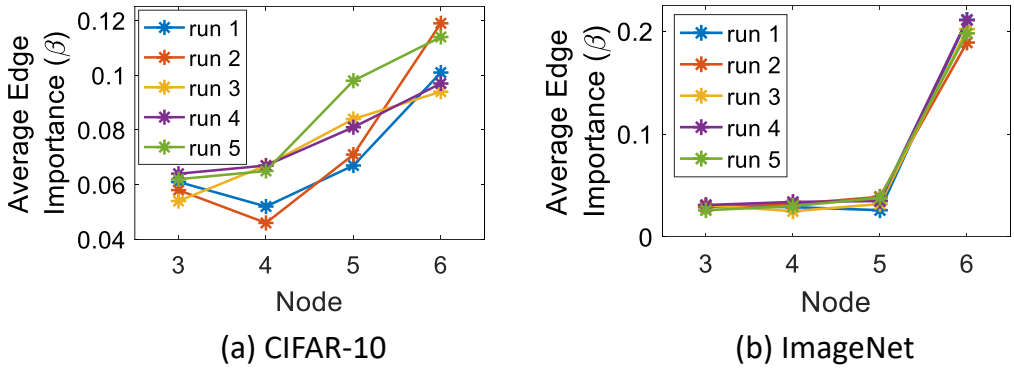


Figure 2.6: Robustness of learned edge importance of (a) CIFAR-10 and (b) ImageNet. The learned importance of different edges does not strongly depend on initialization.

Robustness of Learned Edge Importance. We repeat the experiments 5 times with random seeds on both CIFAR-10 and ImageNet datasets, and report the (per run) averaged incoming edge importance in each immediate node with the best validation performance of the architecture over epochs (we keep track of the most recent architectures). As shown in Figure 2.6, we observe that the learned importance of different edges does not strongly depend on initialization: *even if the initial weights are randomly initialized, after the search process completes, the later intermediate nodes always have higher importance than earlier*

nodes.

More Flexible Architectures. Figure 2.4(b) and Figure 2.5(b) show the number of input edges connecting to each intermediate node, while Figure 2.4(c) and Figure 2.5(c) show the histogram of the number of edges w.r.t the number of operations selected (*e.g.* the third bar from the left shows that three edges have two associated operations). Unlike DARTS in which each intermediate node is hard coded to have *exactly* two input edges and each edge is hard coded to have *exactly* one operation, the best-performing cell found by HM-NAS has intermediate nodes which have more (≥ 2) incoming edges, and edges are associated with zero (the edge is removed) or multiple (≥ 1) operations. This observation suggests that *HM-NAS is able to find more flexible architectures that existing weight sharing based NAS approaches are not able to discover*. In principle, other constraints such as *the number of cells*, *the number of channels*, *the number of nodes in a cell*, and *the combination operation* (*e.g.* sum, concatenation) can all be further relaxed by the proposed multi-level architecture encoding and hierarchical masking schemes.

Conclusion

In this chapter, We propose an efficient NAS approach named HM-NAS that generalizes existing weight sharing based NAS approaches. HM-NAS incorporates a multi-level architecture encoding scheme to enable an architecture candidate to have arbitrary numbers of edges and operations with different importance. The learned hierarchical masks not only select the optimal numbers of edges, operations and important network weights, but also help correct the architecture search bias caused by bilevel optimization in supernet training. Experiment results show that, compared to state-of-the-arts, HM-NAS is able to achieve competitive accuracy on CIFAR-10 and ImageNet with improved architecture search efficiency.

Chapter 3

Arch2Vec

Unsupervised representation learning has been successfully used in a wide range of domains including natural language processing [73–75], computer vision [76, 77], robotic learning [78, 79], and network analysis [80, 81]. Although differing in specific data type, the root cause of such success shared across domains is learning good data representations that are independent of the specific downstream task. In this work, we investigate unsupervised representation learning in the domain of neural architecture search (NAS), and demonstrate how NAS search spaces encoded through unsupervised representation learning could benefit the downstream search strategies. Standard NAS methods encode the search space with the adjacency matrix and focus on designing different downstream search strategies based on reinforcement learning [82], evolutionary algorithm [83], and Bayesian optimization [31] to perform architecture search in discrete search spaces. Such discrete encoding scheme is a natural choice since neural architectures are discrete. However, the size of the adjacency matrix grows quadratically as search space scales up, making downstream architecture search less efficient in large search spaces [5]. To reduce the search cost, recent NAS methods employ dedicated networks to learn continuous representations of neural architectures and perform architecture search in continuous search spaces [12, 59, 63, 84]. In these methods, architecture representations and downstream search strategies are jointly optimized in a supervised manner, guided by the accuracies of architectures selected by the search strategies. However, these supervised architecture representation learning-based methods are biased towards weight-free operations (e.g., skip connections, max-pooling) which are often preferred in the early stage of the search process, resulting in lower final accuracies [40, 62, 85, 86]. In this work, we propose *arch2vec*, a simple yet effective neural architecture search method based

on unsupervised architecture representation learning. As illustrated in Figure 3.1, compared to supervised architecture representation learning-based methods, *arch2vec* circumvents the bias caused by joint optimization through decoupling architecture representation learning and architecture search into two separate processes. To achieve this, *arch2vec* uses a variational graph isomorphism autoencoder to learn architecture representations using only neural architectures without their accuracies. As such, it injectively captures the local structural information of neural architectures and makes architectures with similar structures (measured by edit distance) cluster better and distribute more smoothly in the latent space, which facilitates the downstream architecture search. We visualize the learned architecture representations in Section 3. It shows that architecture representations learned by *arch2vec* can better preserve structural similarity of local neighborhoods than its supervised architecture representation learning counterpart. In particular, it is able to capture topology (e.g. skip connections or straight networks) and operation similarity, which helps cluster architectures with similar accuracy.

We follow the NAS best practices checklist [87] to conduct our experiments. We validate the performance of *arch2vec* on three commonly used NAS search spaces NAS-Bench-101 [10], NAS-Bench-201 [11] and DARTS [12] and two search strategies based on reinforcement learning (RL) and Bayesian optimization (BO). Our results show that, with the same downstream search strategy, *arch2vec* consistently outperforms its discrete encoding and supervised architecture representation learning counterparts across all three search spaces.

Our contributions are summarized as follows: 1). We propose a neural architecture search method based on unsupervised representation learning that decouples architecture representation learning and architecture search. 2). We show that compared to supervised architecture representation learning, pre-training architecture representations without using

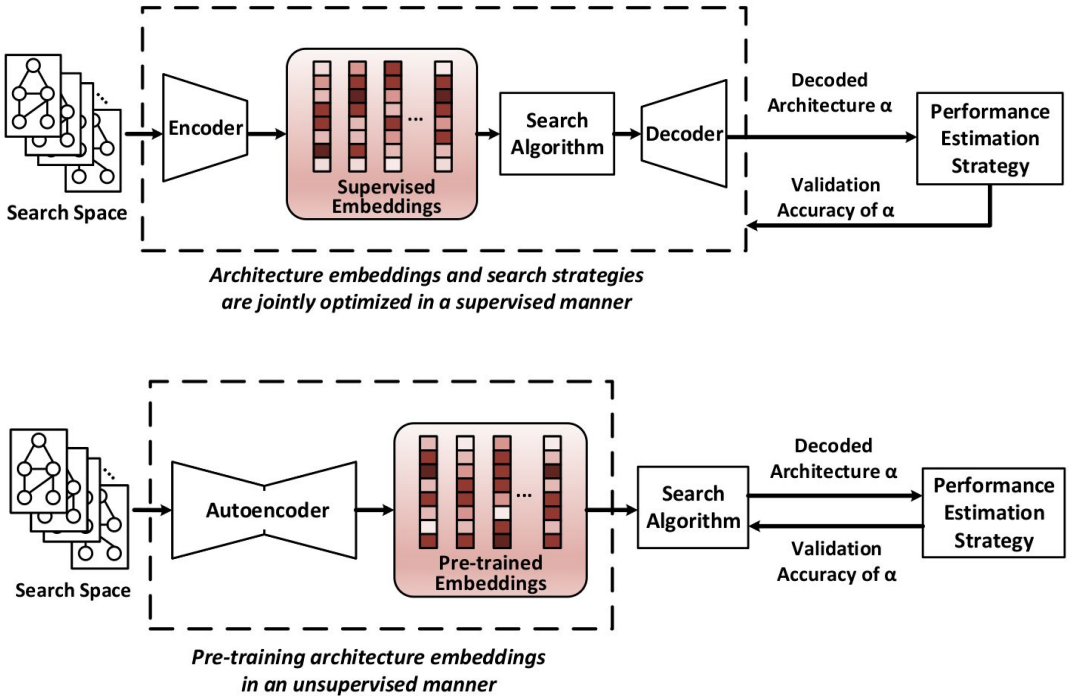


Figure 3.1: Supervised architecture representation learning (top): the supervision signal for representation learning comes from the accuracies of architectures selected by the search strategies. *arch2vec* (bottom): disentangling architecture representation learning and architecture search through unsupervised pre-training.

their accuracies is able to better preserve the local structure relationship of neural architectures and helps construct a smoother latent space. 3). The pre-trained architecture embeddings considerably benefit the downstream architecture search in terms of efficiency and robustness. This finding is consistent across three search spaces, two search strategies and two datasets, demonstrating the importance of unsupervised architecture representation learning for neural architecture search.

Related Work

Unsupervised Representation Learning of Graphs. Our work is closely related to unsupervised representation learning of graphs. In this domain, some methods have been proposed to learn representations using local random walk statistics and matrix factorization-based learning objectives [80,81,88,89]; some methods either reconstruct a graph's adjacency

matrix by predicting edge existence [90, 91] or maximize the mutual information between local node representations and a pooled graph representation [92]. The expressiveness of Graph Neural Networks (GNNs) is studied in [93] in terms of their ability to distinguish any two graphs. It also introduces Graph Isomorphism Networks (GINs), which is proved to be as powerful as the Weisfeiler-Lehman test [94] for graph isomorphism. [53] proposes an asynchronous message passing scheme to encode DAG computations using RNNs. In contrast, we injectively encode architecture structures using GINs, and we show a strong pre-training performance based on its highly expressive aggregation scheme. [95] focuses on network generators that output relational graphs, and the predictive performance highly depends on the structure measures of the relational graphs. In contrast, we encode structural information of neural networks into compact continuous embeddings, and the predictive performance depends on how well the structure is injected into the embeddings.

Regularized Autoencoders. Autoencoders can be seen as energy-based models trained with reconstruction energy [96]. Our goal is to encode neural architectures with similar performance into the same regions of the latent space, and to make the transition of architectures in the latent space relatively smooth. To prevent degenerated mapping where latent space is free of any structure, there is a rich literature on restricting the low-energy area for data points on the manifold [97–101]. Here we adopt the popular variational autoencoder framework [90,99] to optimize the variational lower bound w.r.t. the variational parameters, which as we show in our experiments acts as an effective regularization. While [102, 103] use graph VAE for the generative problems, we focus on mapping the finite discrete neural architectures into the continuous latent space regularized by KL-divergence such that each architecture is encoded into a unique area in the latent space.

Neural Architecture Search (NAS). As mentioned in §1, early NAS methods are

built upon discrete encodings [7, 18, 21, 24, 31], which face the scalability challenge [5, 104] in large search spaces. To address this challenge, recent NAS methods shift from conducting architecture search in discrete spaces to continuous spaces using different architecture encoders such as SRM [24], MLP [50], LSTM [63] or GCN [44, 46]. However, what lies in common under these methods is that the architecture representation and search direction are jointly optimized by the supervision signal (e.g., accuracies of the selected architectures), which could bias the architecture representation learning and search direction. [6] emphasizes the importance of studying architecture encodings, and we focus on encoding adjacency matrix-based architectures into low-dimensional embeddings in the continuous space. [105] shows that architectures searched without using labels are competitive to their counterparts searched with labels. Different from their approach which performs pretext tasks using image statistics, we use architecture reconstruction objective to preserve the local structure relationship in the latent space.

Approach

In this section, we describe the details of *arch2vec*, followed by two downstream architecture search strategies we use in this work.

Variational Graph Isomorphism Autoencoder

Preliminaries. We restrict our search space to the cell-based architectures. Following the configuration in NAS-Bench-101 [10], each cell is a labeled DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ as a set of N nodes and $\mathcal{E}$ as a set of edges. Each node is associated with a label chosen from a set of K predefined operations. A natural encoding scheme of cell-based neural architectures is an upper triangular adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and an one-hot operation matrix $\mathbf{X} \in \mathbb{R}^{N \times K}$. This discrete encoding is not unique, as permuting the adjacency matrix $\mathbf{A}$ and the operation matrix $\mathbf{X}$ would lead to the same graph, which is known as graph isomorphism [94].

Encoder. To learn a continuous representation that is invariant to isomorphic graphs, we leverage Graph Isomorphism Networks (GINs) [93] to encode the graph-structured architectures given its better expressiveness. We augment the adjacency matrix as $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{A}^T$ to transfer original directed graphs into undirected graphs, allowing bi-directional information flow. Similar to [90], the inference model, *i.e.* the encoding part of the model, is defined as:

$$q(\mathbf{Z}|\mathbf{X}, \tilde{\mathbf{A}}) = \prod_{i=1}^{N} q(\mathbf{z}_i|\mathbf{X}, \tilde{\mathbf{A}}), \text{with } q(\mathbf{z}_i|\mathbf{X}, \tilde{\mathbf{A}}) = \mathcal{N}(\mathbf{z}_i|\boldsymbol{\mu}_i, diag(\boldsymbol{\sigma}_i^2)). \tag{3.1}$$

We use the L-layer GIN to get the node embedding matrix $\mathbf{H}$:

$$\mathbf{H}^{(k)} = \text{MLP}^{(k)}\left(\left(1 + \epsilon^{(k)}\right) \cdot \mathbf{H}^{(k-1)} + \tilde{\mathbf{A}}\mathbf{H}^{(k-1)}\right), k = 1, 2, \ldots, L, \tag{3.2}$$

where $\mathbf{H}^{(0)} = \mathbf{X}$, ϵ is a trainable bias, and MLP is a multi-layer perception where each layer is a linear-batchnorm-ReLU triplet. The node embedding matrix $\mathbf{H}^{(L)}$ is then fed into two fully-connected layers to obtain the mean $\boldsymbol{\mu}$ and the variance $\boldsymbol{\sigma}$ of the posterior approximation $q(\mathbf{Z}|\mathbf{X}, \tilde{\mathbf{A}})$ in Eq. (3.1). During the inference, the architecture representation is derived by summing the representation vectors of all the nodes.

Decoder. Our decoder is a generative model aiming at reconstructing $\hat{\mathbf{A}}$ and $\hat{\mathbf{X}}$ from the latent variables $\mathbf{Z}$:

$$p(\hat{\mathbf{A}}|\mathbf{Z}) = \prod_{i=1}^{N}\prod_{j=1}^{N} P(\hat{A}_{ij}|\mathbf{z}_i, \mathbf{z}_j), \text{with } p(\hat{A}_{ij} = 1|\mathbf{z}_i, \mathbf{z}_j) = \sigma(\mathbf{z}_i^T\mathbf{z}_j), \tag{3.3}$$

$$p(\hat{\mathbf{X}} = [k_1, ..., k_N]^T|\mathbf{Z}) = \prod_{i=1}^{N} P(\hat{\mathbf{X}}_i = k_i|\mathbf{z}_i) = \prod_{i=1}^{N} \text{softmax}(\mathbf{W}_o\mathbf{Z} + \mathbf{b}_o)_{i,k_i}, \tag{3.4}$$

where $\sigma(\cdot)$ is the sigmoid activation, softmax$(\cdot)$ is the softmax activation applied row-wise, and $k_n \in \{1, 2, ..., K\}$ indicates the operation selected from the predifined set of K opreations at the nth node. $\mathbf{W}_o$ and $\mathbf{b}_o$ are learnable weights and biases of the decoder.

Training Objective

In practice, our variational graph isomorphism autoencoder consists of a five-layer GIN and a one-layer MLP. The details of the model architecture are described in Section 3. The dimensionality of the embedding is set to 16. During training, model weights are learned by iteratively maximizing a tractable variational lower bound:

$$\mathcal{L} = \mathbb{E}_{q(\mathbf{Z}|\mathbf{X},\tilde{\mathbf{A}})}[\log p(\hat{\mathbf{X}}, \hat{\mathbf{A}}|\mathbf{Z})] - \mathcal{D}_{KL}(q(\mathbf{Z}|\mathbf{X}, \tilde{\mathbf{A}})||p(\mathbf{Z})), \tag{3.5}$$

where $p(\hat{\mathbf{X}}, \hat{\mathbf{A}}|\mathbf{Z}) = p(\hat{\mathbf{A}}|\mathbf{Z})p(\hat{\mathbf{X}}|\mathbf{Z})$ as we assume that the adjacency matrix $\mathbf{A}$ and the operation matrix $\mathbf{X}$ are conditionally independent given the latent variable $\mathbf{Z}$. The second term $\mathcal{D}_{KL}$ on the right hand side of Eq. (3.5) denotes the Kullback-Leibler divergence [106] which is used to measure the difference between the posterior distribution $q(\cdot)$ and the prior distribution $p(\cdot)$. Here we choose a Gaussian prior $p(\mathbf{Z}) = \prod_i \mathcal{N}(\mathbf{z}_i|0, \mathbf{I})$ due to its simplicity. We use reparameterization trick [99] for training since it can be thought of as injecting noise to the code layer. The random noise injection mechanism has been proved to be effective on the regularization of neural networks [99, 107, 108]. The loss is optimized using mini-batch gradient descent over neural architectures.

Architecture Search Strategies

We use reinforcement learning (RL) and Bayesian optimization (BO) as two representative search algorithms to evaluate *arch2vec* on the downstream architecture search.

Reinforcement Learning (RL). We use REINFORCE [82] as our RL-based search

strategy as it has been shown to converge better than more advanced RL methods such as PPO [109] for neural architecture search. For RL, the pre-trained embeddings are passed to the Policy LSTM to sample the action and obtain the next state (valid architecture embedding) using nearest-neighborhood retrieval based on L2 distance to maximize accuracy as reward. We use a single-layer LSTM as the controller and output a 16-dimensional output as the mean vector to the Gaussian policy with a fixed identity covariance matrix. The controller is optimized using Adam optimizer [68] with a learning rate of 1×10^{-2}. The number of sampled architectures in each episode is set to 16 and the discount factor is set to 0.8. The baseline value is set to 0.95. The search is stopped when it reaches the time budget 1.2×10^4, 5×10^5, 1.4×10^6 seconds for CIFAR-10, CIFAR-100, and ImageNet-16-200, respectively. For CIFAR-10, we follow the same implementation established in NAS-Bench-201 by searching based on the validation accuracy obtained after 12 training epochs with converged learning rate scheduling. The discount factor and the baseline value is set to 0.4. All the other hyperparameters are the same as described in Section 3.

Bayesian Optimization (BO). We use DNGO [110] as our BO-based search strategy. We use a one-layer adaptive basis regression network with hidden dimension 128 to model distributions over functions. It serves as an alternative to Gaussian process in order to avoid cubic scaling [111]. We use expected improvement (EI) [112] as the acquisition function which is widely used in NAS [21, 46, 50]. The best function value of EI is set to 0.95. During the search process, the pre-trained embeddings are passed to DNGO to select the top-5 architectures in each round of search, which are then added to the pool. The network is retrained for 100 epochs in the next round using the selected architectures in the updated pool. This process is iterated until the maximum estimated wall-clock time is reached. We use the same time budget used in RL-based search. All the other hyperparameters are the

same.

Experiments

Encoding of NAS-Bench-101. We validate *arch2vec* on three commonly used NAS search spaces. We followed the encoding scheme in NAS-Bench-101 [10]. Specifically, a cell in NAS-Bench-101 is represented as a directed acyclic graph (DAG) where nodes represent operations and edges represent data flow. A 7×7 upper-triangular binary matrix is used to encode edges. A 7×5 operation matrix is used to encode operations, input, and output, with the order as {input, 1×1 conv, 3×3 conv, 3×3 max-pool (MP), output}. For cells with less than 7 nodes, their adjacency and operator matrices are padded with trailing zeros. Figure 1.2 (left) shows an example of a 7-node cell in NAS-Bench-101 search space and its corresponding adjacency and operation matrices.

Encoding of NAS-Bench-201. Different from NAS-Bench-101, NAS-Bench-201 [11] employs a fixed cell-based DAG representation of neural architectures, where nodes represent the sum of feature maps and edges are associated with operations that transform the feature maps from the source node to the destination node. To represent the architectures in NAS-Bench-201 with discrete encoding that is compatible with our neural architecture encoder, we first transform the original DAG in NAS-Bench-201 into a DAG with nodes representing operations and edges representing data flow as the ones in NAS-Bench-101. We then use the same discrete encoding scheme in NAS-Bench-101 to encode each cell into an adjacency matrix and operation matrix. An example is shown in Figure 1.2 (right). The hyperparameters we used for pre-training on NAS-Bench-201 are the same as NAS-Bench-101.

DARTS search space. The DARTS search space [12] is a popular search space for large-scale NAS experiments. The search space consists of two cells: a convolutional cell and a reduction cell, each with six nodes. For each cell, the first two nodes are the outputs

from the previous two cells. The next four nodes contain two edges as input, creating a DAG. The network is then constructed by stacking the cells. Following [113], we use the same cell for both normal and reduction cell, allowing roughly 10^9 DAGs without considering graph isomorphism. We randomly sample 600,000 unique architectures in this search space following the mobile setting [12]. We use the same data split as used in NAS-Bench-101.

For pre-training, we use a five-layer Graph Isomorphism Network (GIN) with hidden sizes of {128, 128, 128, 128, 16} as the encoder and a one-layer MLP with a hidden dimension of 16 as the decoder. The adjacency matrix is preprocessed as an undirected graph to allow bi-directional information flow. After forwarding the inputs to the model, the reconstruction error is minimized using Adam optimizer [68] with a learning rate of 1×10^{-3}. We train the model with batch size 32 and the training loss is able to converge well after 8 epochs on NAS-Bench-101, and 10 epochs on NAS-Bench-201 and DARTS. After training, we extract the architecture embeddings from the encoder for the downstream architecture search.

In the following, we first evaluate the pre-training performance of *arch2vec* and then the neural architecture search performance based on its pre-trained representations.

Pre-training Performance

Observation (1): We compare *arch2vec* with two popular baselines GAE [90] and VGAE [90] using three metrics suggested by [53]: 1) Reconstruction Accuracy (reconstruction accuracy of the held-out test set), 2) Validity (how often a random sample from the prior distribution can generate a valid architecture), and 3) Uniqueness (unique architectures out of valid generations). As shown in Table 3.1, *arch2vec* outperforms both GAE and VGAE, and achieves the highest reconstruction accuracy, validity, and uniqueness across all the three search spaces. This is because encoding with GINs outperforms GCNs in reconstruction accuracy due to its better neighbor aggregation scheme; the KL term effectively regularizes the

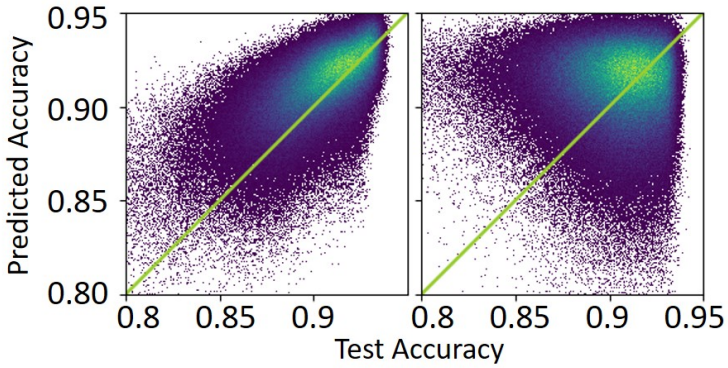


Figure 3.2: predictive performance comparison between *arch2vec* (left) and supervised architecture representation learning (right) on NAS-Bench-101.

mapping from the discrete space to the continuous latent space, leading to better generative performance measured by validity and uniqueness. Given its superior performance, we stick to *arch2vec* for the remainder of our evaluation.

Method	NAS-Bench-101			NAS-Bench-201			DARTS		
	Accuracy	Validity	Uniqueness	Accuracy	Validity	Uniqueness	Accuracy	Validity	Uniqueness
GAE [90]	98.75	29.88	99.25	99.52	79.28	78.42	97.80	15.25	99.65
VGAE [90]	97.45	41.18	99.34	98.32	79.30	88.42	96.80	25.25	99.27
arch2vec (w.o. KL)	**100**	30.31	99.20	**100**	77.09	96.57	99.46	16.01	99.51
arch2vec	**100**	**44.97**	**99.69**	**100**	**79.41**	**98.72**	**99.79**	**33.36**	**100**

Table 3.1: Reconstruction accuracy, validity, and uniqueness of different GNNs.

We compare *arch2vec* with its supervised architecture representation learning counterpart on the predictive performance of the latent representations. This metric measures how well the latent representations can predict the performance of the corresponding architectures. Being able to accurately predict the performance of the architectures based on the latent representations makes it easier to search for the high-performance points in the latent space. Specifically, we train a Gaussian Process model with 250 sampled architectures to predict the performance of the other architectures, and report the predictive performance across 10 different seeds. We use RMSE and the Pearson correlation coefficient (Pearson's r) to evaluate

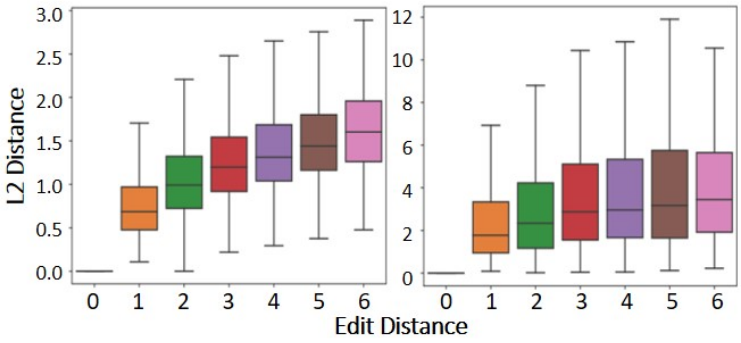


Figure 3.3: Comparing distribution of L2 distance between architecture pairs by edit distance on NAS-Bench-101, measured by 1,000 architectures sampled in a long random walk with 1 edit distance apart from consecutive samples. left: *arch2vec*. right: supervised architecture representation learning.

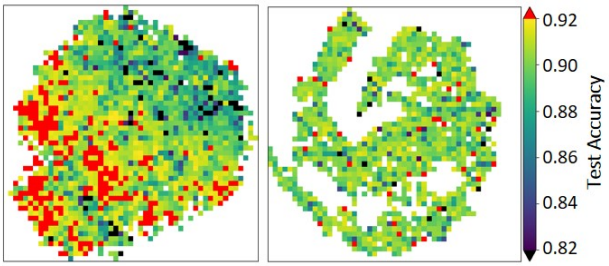


Figure 3.4: Latent space 2D visualization [114] comparison between *arch2vec* (left) and supervised architecture representation learning (right) on NAS-Bench-101. Color encodes test accuracy. We randomly sample $10,000$ points and average the accuracy in each small area.

points with test accuracy higher than 0.8. Figure 3.2 compares the predictive performance between *arch2vec* and its supervised counterpart on NAS-Bench-101. As shown, *arch2vec* outperforms its supervised counterpart[1], indicating *arch2vec* is able to better capture the local structure relationship of the input space and hence is more informative on guiding the downstream search process.

Observation (3): In Figure 3.3, we plot the relationship between the L2 distance in the latent space and the edit distance of the corresponding DAGs between two architectures. As

[1]The RMSE and Pearson's r are: 0.038±0.025 / 0.53±0.09 for the supervised architecture representation learning, and 0.018±0.001 / 0.67±0.02 for *arch2vec*. A smaller RMSE and a larger Pearson's r indicates a better predictive performance.

shown, for *arch2vec*, the L2 distance grows monotonically with increasing edit distance. This result indicates that *arch2vec* is able to preserve the closeness between two architectures measured by edit distance, which potentially benefits the effectiveness of the downstream search. In contrast, such closeness is not well captured by supervised architecture representation learning.

Observation (4): In Figure 3.4, we visualize the latent spaces of NAS-Bench-101 learned by *arch2vec* (left) and its supervised counterpart (right) in the 2-dimensional space generated using t-SNE. We overlaid the original colorscale with red (>92% accuracy) and black (<82% accuracy) for highlighting purpose. As shown, for *arch2vec*, the architecture embeddings span the whole latent space, and architectures with similar accuracies are clustered together. In particular, the left-lower region has higher accuracy while the right-upper region has lower accuracy. Conducting architecture search on such smooth performance surface is much easier and is hence more efficient. In contrast, for the supervised counterpart, the embeddings are discontinuous in the latent space, and the transition of accuracy is non-smooth. This indicates that joint optimization guided by accuracy cannot injectively encode architecture structures. As a result, architecture does not have its unique embedding in the latent space, which makes the task of architecture search more challenging.

Observation (5): To provide a closer look at the learned latent space, Figure 3.5 visualizes the architecture cells decoded from the latent space of *arch2vec* (upper) and supervised architecture representation learning (lower). For *arch2vec*, the adjacent architectures change smoothly and embrace similar connections and operations. This indicates that unsupervised architecture representation learning helps model a smoothly-changing structure surface. As we show in the next section, such smoothness greatly helps the downstream search since architectures with similar performance tend to locate near each other in the latent space

Figure 3.5: Visualization of a sequence of architecture cells decoded from the learned latent space of *arch2vec* (upper) and supervised architecture representation learning (lower) on NAS-Bench-101. The two sequences start from the same architecture. For both sequences, each architecture is the closest point of the previous one in the latent space excluding previously visited ones. Edit distances between adjacent architectures of the upper sequence are 4, 6, 1, 5, 1, 1, 1, 5, 2, 3, 2, 4, 2, 5, 2, and the average is 2.9. Edit distances between adjacent architectures of the lower sequence are 8, 6, 7, 7, 9, 8, 11, 11, 6, 10, 10, 11, 10, 11, 9, and the average is 8.9.

instead of locating randomly. In contrast, the supervised counterpart does not group similar connections and operations well and has much higher edit distances between adjacent architectures. This biases the search direction since dependencies between architecture structures are not well captured.

NAS Performance

NAS results on NAS-Bench-101. For fair comparison, we reproduced the NAS methods which use the adjacency matrix-based encoding in [10][2], including Random Search (RS) [115], Regularized Evolution (RE) [18], REINFORCE [82] and BOHB [31]. For supervised architecture representation learning-based methods, the hyperparameters are the same as *arch2vec*, except that the architecture representation learning and search are jointly optimized. Figure

[2] `https://github.com/automl/nas_benchmarks`

NAS Methods	#Queries	Test Accuracy (%)	Encoding	Search Method
Random Search [10]	1000	93.54	Discrete	Random
RL [10]	1000	93.58	Discrete	REINFORCE
BO [10]	1000	93.72	Discrete	Bayesian Optimization
RE [10]	1000	93.72	Discrete	Evolution
NAO [63]	1000	93.74	Supervised	Gradient Decent
BANANAS [50]	500	94.08	Supervised	Bayesian Optimization
RL (ours)	400	93.74	Supervised	REINFORCE
BO (ours)	400	93.79	Supervised	Bayesian Optimization
arch2vec-RL	**400**	**94.10**	Unsupervised	REINFORCE
arch2vec-BO	400	94.05	Unsupervised	Bayesian Optimization

Table 3.2: Comparison of NAS performance between *arch2vec* and SOTA methods on NAS-Bench-101. It reports the mean performance of 500 independent runs given the number of queried architectures.

3.6 and Table 3.2 summarize our results.

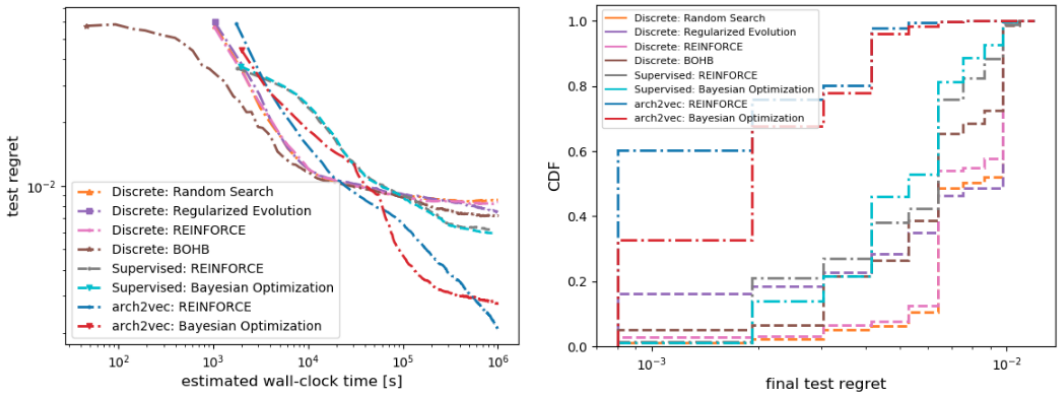


Figure 3.6: Comparison of NAS performance between discrete encoding, supervised architecture representation learning, and *arch2vec* on NAS-Bench-101. The plot shows the mean test regret (left) and the empirical cumulative distribution of the final test regret (right) of 500 independent runs given a wall-clock time budget of 1×10^6 seconds.

As shown in Figure 3.6, BOHB and RE are the two best-performing methods using the adjacency matrix-based encoding. However, they perform slightly worse than supervised architecture representation learning because the high-dimensional input may require more observations for the optimization. In contrast, supervised architecture representation learning focuses on low-dimensional continuous optimization and thus makes the search more

NAS Methods	CIFAR-10		CIFAR-100		ImageNet-16-120	
	validation	test	validation	test	validation	test
RE [18]	91.08±0.43	93.84±0.43	73.02±0.46	72.86±0.55	45.78±0.56	45.63±0.64
RS [115]	90.94±0.38	93.75±0.37	72.17±0.64	72.05±0.77	45.47±0.65	45.33±0.79
REINFORCE [82]	91.03±0.33	93.82±0.31	72.35±0.63	72.13±0.79	45.58±0.62	45.30±0.86
BOHB [31]	90.82±0.53	93.61±0.52	72.59±0.82	72.37±0.90	45.44±0.70	45.26±0.83
arch2vec-RL	91.32±0.42	94.12±0.42	73.13±0.72	73.15±0.78	46.22±0.30	46.16±0.38
arch2vec-BO	**91.41±0.22**	**94.18±0.24**	**73.35±0.32**	**73.37±0.30**	**46.34±0.18**	**46.27±0.37**

Table 3.3: The mean and standard deviation of the validation and test accuracy of different algorithms under three datasets in NAS-Bench-201. The results are calculated over 500 independent runs.

efficient. As shown in Figure 3.6 (left), *arch2vec* considerably outperforms its supervised counterpart and the adjacency matrix-based encoding after 5×10^4 wall clock seconds. Figure 3.6 (right) further shows that *arch2vec* is able to robustly achieve the lowest final test regret after 1×10^6 seconds across 500 independent runs.

NAS results on NAS-Bench-201. For CIFAR-10, we follow the same implementation established in NAS-Bench-201 by searching based on the validation accuracy obtained after 12 training epochs with converged learning rate scheduling. The search budget is set to 1.2×10^4 seconds. The NAS experiments on CIFAR-100 and ImageNet-16-120 are conducted with a budget that corresponds to the same number of queries used in CIFAR-10. As listed in Table 3.3, searching with *arch2vec* leads to better validation and test accuracy as well as reduced variability among different runs on all datasets.

NAS results on DARTS search space. Similar to [50], we set the budget to 100 queries in this search space. In each query, a sampled architecture is trained for 50 epochs and the average validation error of the last 5 epochs is computed. To ensure fair comparison with the same hyparameters setup, we re-trained the architectures from works that *exactly*[3] use DARTS search space and report the final architecture. As shown in Table 3.4, *arch2vec*

[3] `https://github.com/quark0/darts/blob/master/cnn/train.py`

	Test Error		Params (M)	Search Cost				
NAS Methods	Avg	Best		Stage 1	Stage 2	Total	**Encoding**	**Search Method**
Random Search [12]	3.29±0.15	-	3.2	-	-	4	-	Random
ENAS [57]	-	2.89	4.6	0.5	-	-	Supervised	REINFORCE
ASHA [116]	3.03±0.13	2.85	2.2	-	-	9	-	Random
RS WS [116]	2.85±0.08	2.71	4.3	2.7	6	8.7	-	Random
SNAS [59]	2.85±0.02	-	2.8	1.5	-	-	Supervised	GD
DARTS [12]	2.76±0.09	-	3.3	4	1	5	Supervised	GD
BANANAS [50]	2.64	2.57	3.6	100 (queries)	-	11.8	Supervised	BO
Random Search (ours)	3.1±0.18	2.71	3.2	-	-	4	-	Random
DARTS (ours)	2.71±0.08	2.63	3.3	4	1.2	5.2	Supervised	GD
BANANAS (ours)	2.67±0.07	2.61	3.6	100 (queries)	1.3	11.5	Supervised	BO
arch2vec-RL	2.65±0.05	2.60	3.3	100 (queries)	1.2	9.5	Unsupervised	REINFORCE
arch2vec-BO	**2.56±0.05**	**2.48**	3.6	100 (queries)	1.3	10.5	Unsupervised	BO

Table 3.4: Comparison with state-of-the-art cell-based NAS methods on DARTS search space using CIFAR-10. The test error is averaged over 5 seeds. Stage 1 shows the GPU days (or number of queries) for model search and Stage 2 shows the GPU days for model evaluation.

generally leads to competitive search performance among different cell-based NAS methods with comparable model parameters. The best performed cells and transfer learning results on ImageNet.

Ablation Study

Pretraining Metrics

We split the the dataset into 90% training and 10% held-out test sets for *arch2vec* pre-training on each search space. In Section 3, we evaluate the pre-training performance of *arch2vec* using three metrics suggested by [53]: 1) Reconstruction Accuracy (reconstruction accuracy of the held-out test set) which measures how well the embeddings can errorlessly remap to the original structures; 2) Validity (how often a random sample from the prior distribution can generate a valid architecture) which measures the generative ability the model; and 3) Uniqueness (unique architectures out of valid generations) which measures the smoothness and diversity of the generated samples. To compute Reconstruction Accuracy, we report the proportion of the decoded neural architectures of the held-out test set that are identical

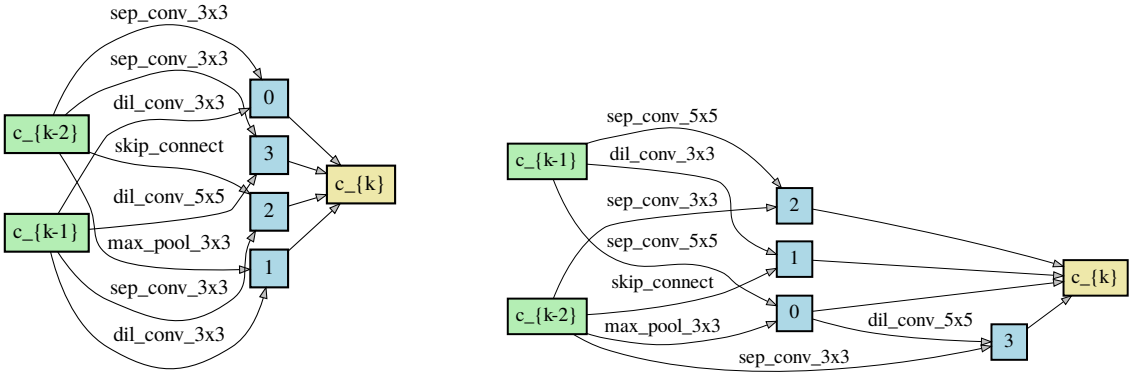
to the inputs. To compute Validity, we randomly pick 10,000 points $\mathbf{z}$ generated by the Gaussian prior $p(\mathbf{Z}) = \prod_i \mathcal{N}(\mathbf{z}_i|0, \mathbf{I})$ and then apply $\mathbf{z} = \mathbf{z} \odot \text{std}(\mathbf{Z}_{train}) + \text{mean}(\mathbf{Z}_{train})$, where $\mathbf{Z}_{train}$ are the encoded means of the training data. It scales the sampled points and shifts them to the center of the embeddings of the training set. We report the proportion of the decoded architectures that are valid in the search space. To compute Uniqueness, we report the proportion of the unique architectures out of valid decoded architectures. The validity check criteria varies across different search spaces. For NAS-Bench-101 and NAS-Bench-201, we use the NAS-Bench-101[4] and NAS-Bench-201[5] official APIs to verify whether a decoded architecture is valid or not in the search space. For DARTS search space, a decoded architecture has to pass the following validity checks: 1) the first two nodes must be the input nodes c_{k-2} and c_{k-1}; 2) the last node must be the output node c_k; 3) except the two input nodes, there are no nodes which do not have any predecessor; 4) except the output node, there are no nodes which do not have any successor; 5) each intermediate node must contain two edges from the previous nodes; and 6) it has to be an upper-triangular binary matrix (representing a DAG).

Transfer Learning Results

Figure 3.7 shows the best cell found by *arch2vec* using RL-based and BO-based search strategy. As observed in [117], the shapes of normalized empirical distribution functions (EDFs) for NAS design spaces on ImagetNet [71] match their CIFAR-10 counterparts. This suggests that NAS design spaces developed on CIFAR-10 are transferable to ImageNet [117]. Therefore, we evaluate the performance of the best cell found on CIFAR-10 using *arch2vec* for ImageNet. In order to compare in a fair manner, we consider the mobile setting [12, 18, 54]

[4] `https://github.com/google-research/nasbench/blob/master/nasbench/api.py`

[5] `https://github.com/D-X-Y/NAS-Bench-201/blob/v1.1/nas_201_api/api.py`



(a) *arch2vec*-RL (b) *arch2vec*-BO

Figure 3.7: Best cell found by *arch2vec* using (a) RL-based and (b) BO-based search strategy.

NAS Methods	Params (M)	Mult-Adds (M)	Top-1 Test Error (%)	Comparable Search Space
NASNet-A [54]	5.3	564	26.0	Y
AmoebaNet-A [18]	5.1	555	25.5	Y
PNAS [18]	5.1	588	25.8	Y
SNAS [59]	4.3	522	27.3	Y
DARTS [12]	4.7	574	26.7	Y
arch2vec-RL	4.8	533	25.8	Y
arch2vec-BO	5.2	580	25.5	Y

Table 3.5: Transfer learning results on ImageNet.

where the number of multiply-add operations of the model is restricted to be less than 600M. We follow [116] to use the exactly same training hyperparameters used in the DARTS paper [12]. Table 3.5 shows the transfer learning results on ImageNet. With comparable computational complexity, *arch2vec*-RL and *arch2vec*-BO outperform DARTS [12] and SNAS [59] methods in the DARTS search space, and is competitive among all cell-based NAS methods under this setting.

Visualizations

NAS-Bench-101. In Figure 3.8, we visualize three randomly selected pairs of sequences of architecture cells decoded from the learned latent space of *arch2vec* (upper) and supervised

architecture representation learning (lower) on NAS-Bench-101. Each pair starts from the same point, and each architecture is the closest point of the previous one in the latent space excluding previously visited ones. As shown, architecture representations learned by *arch2vec* can better capture topology and operation similarity than its supervised architecture representation learning counterpart. In particular, Figure 3.8 (a) and (b) show that *arch2vec* is able to better cluster straight networks, while supervised learning encodes straight networks and networks with skip connections together in the latent space.

NAS-Bench-201. Similarly, Figure 3.9 shows the visualization of five randomly selected pairs of sequences of decoded architecture cells using *arch2vec* (upper) and supervised architecture representation learning (lower) on NAS-Bench-201. The red mark denotes the change of operations between consecutive samples. Note that the edge flow in NAS-Bench-201 is fixed; only the operator associated with each edge can be changed. As shown, *arch2vec* leads to a smoother local change of operations than its supervised architecture representation learning counterpart.

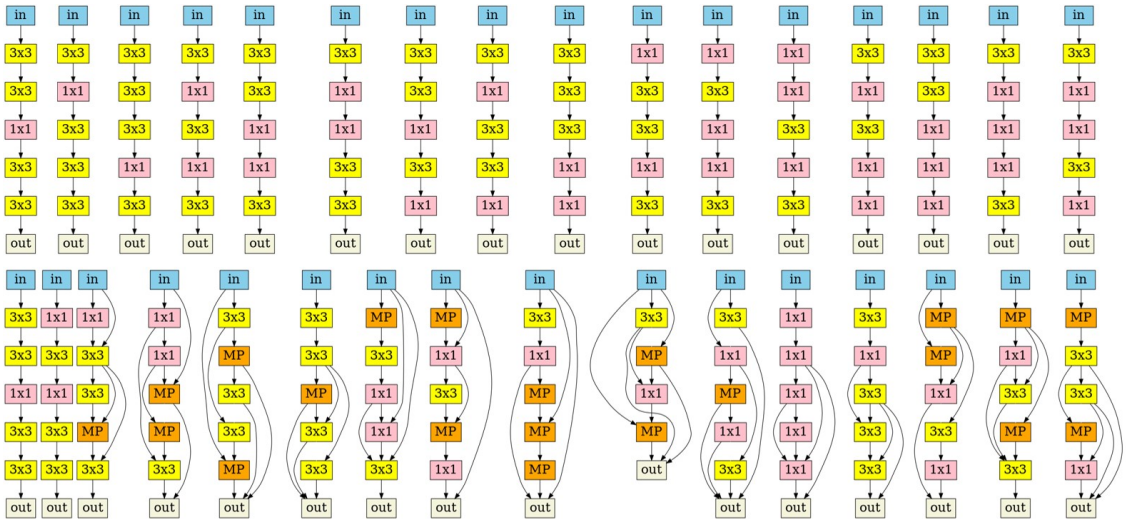
DARTS Search Space. For the DARTS search space, we can only visualize the decoded architecture cells using *arch2vec* since there is no architecture accuracy recorded in this large-scale search space. Figure 3.10 shows an example of the sequence of decoded neural architecture cells using *arch2vec*. As shown, the edge connections of each cell remain unchanged in the decoded sequence, and the operation associated with each edge is gradually changed. This indicates that *arch2vec* preserves the local structural similarity of neighborhoods in the latent space.

Conclusion

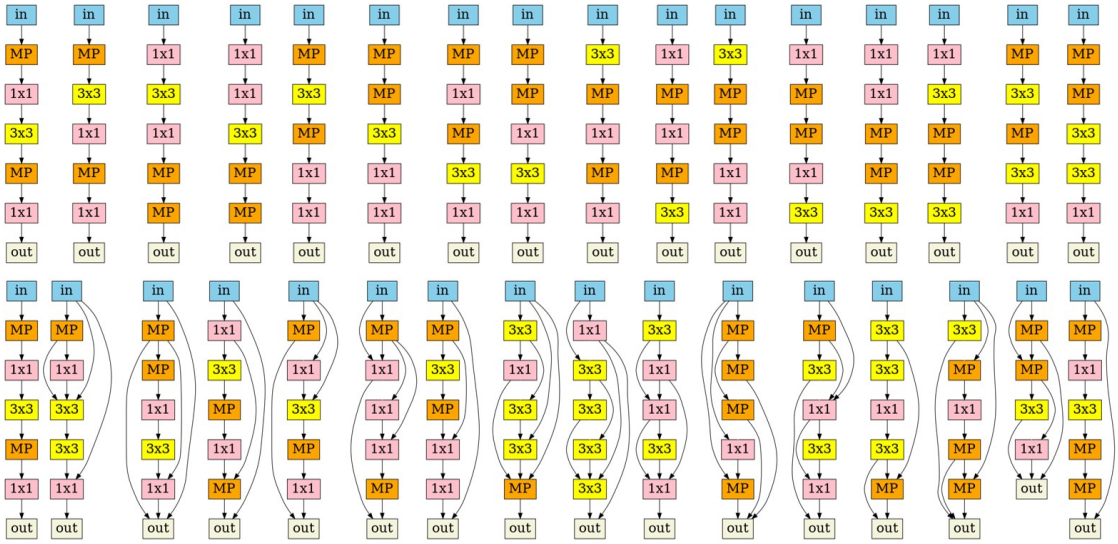
We have shown that *arch2vec* is a simple yet effective neural architecture search method based on unsupervised architecture representation learning. By learning architecture representa-

tions without using their accuracies, it constructs a more smoothly-changing architecture performance surface in the latent space compared to its supervised architecture representation learning counterpart. We have demonstrated its effectiveness on benefiting different downstream search strategies in three NAS search spaces. We suggest that it is desirable to take a closer look at architecture representation learning for neural architecture search. It is also possible that designing neural architecture search method using *arch2vec* with a better search strategy in the continuous space will produce better results.

We will now show a more effective self-supervised method to learn computation-aware architecture encodings.

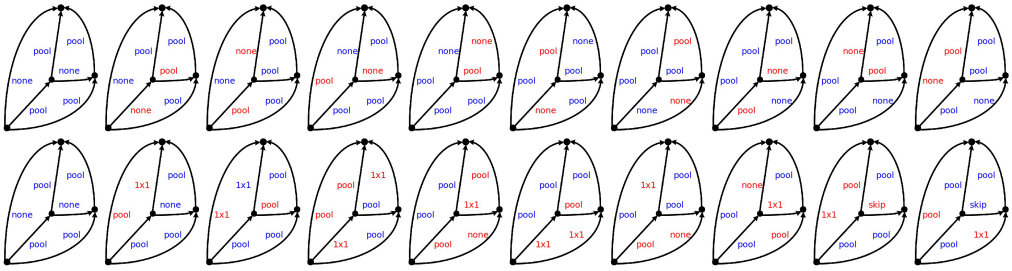


(a) *arch2vec* (upper) and supervised architecture representation learning (lower).

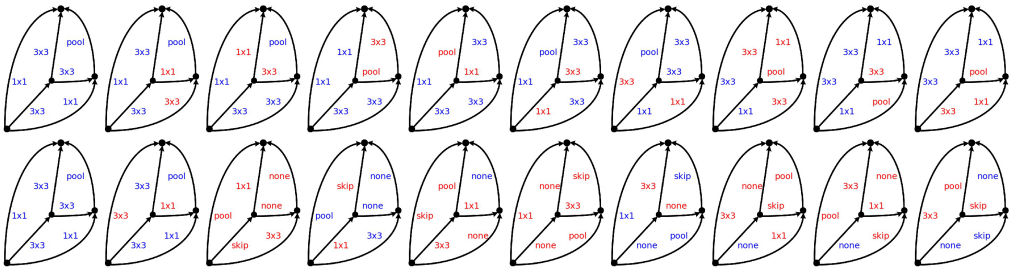


(b) *arch2vec* (upper) and supervised architecture representation learning (lower).

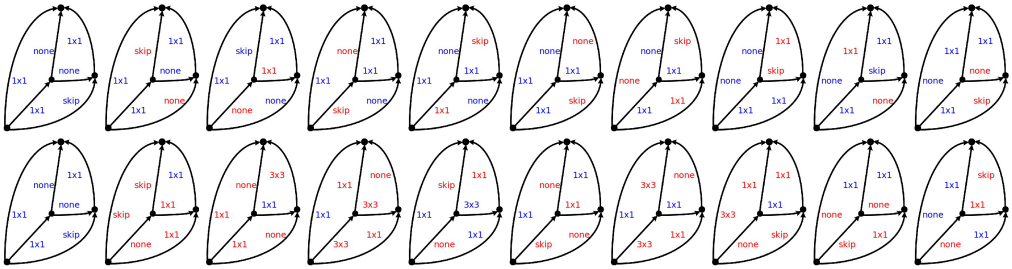
Figure 3.8: Visualization of decoded cells on NAS-Bench-101.



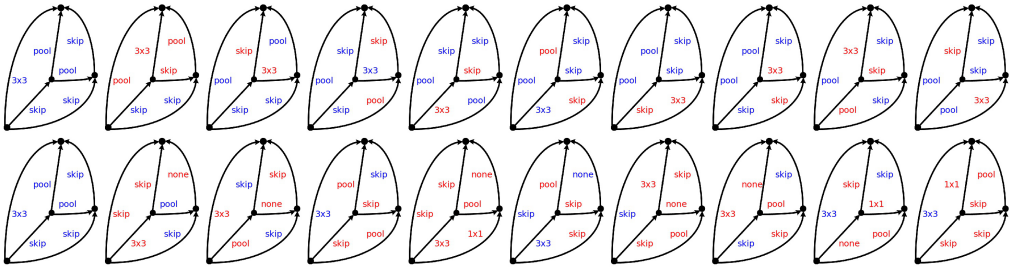
(a) *arch2vec* (upper) and supervised architecture representation learning (lower).



(b) *arch2vec* (upper) and supervised architecture representation learning (lower).



(c) *arch2vec* (upper) and supervised architecture representation learning (lower).



(d) *arch2vec* (upper) and supervised architecture representation learning (lower).

Figure 3.9: Visualization of decoded cells on NAS-Bench-201.

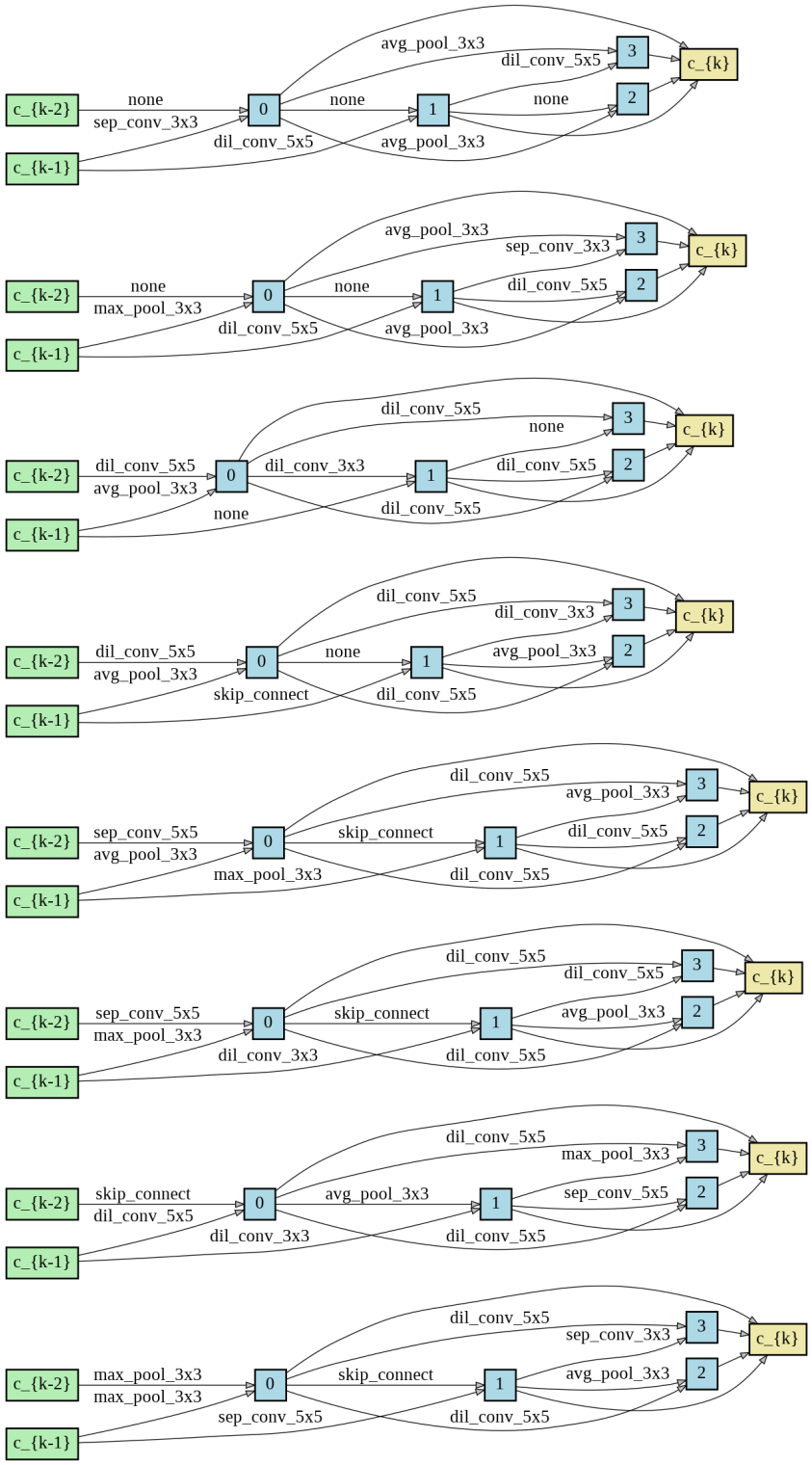


Figure 3.10: Visualization of decoded neural architecture cells using *arch2vec* on DARTS search space. It starts from a randomly sampled point. Each architecture in the sequence is the closest point of the previous one in the latent space excluding previously visited ones.

Chapter 4

Computation-aware Neural Architecture Encoding

As demonstrated in Chapter 3, while majority of the prior work focuses on either constructing new search spaces [20, 118, 119] or designing efficient architecture search and evaluation methods [46, 50, 63], some of the most recent work [2, 6] sheds light on the importance of *architecture encoding* on the subroutines in the NAS pipeline as well as on the overall performance of NAS.

While existing NAS methods use diverse architecture encoders such as LSTM [54, 63], SRM [24], MLP [113, 120], GNN [2, 44, 46] or adjacency matrix itself [18, 21, 121], these encoders encode either *structures* [2, 10, 44, 46, 63, 120] or *computations* [50, 53, 122] of the neural architectures. Compared to structure-aware encodings, computation-aware encodings are able to map architectures with different structures but similar accuracies to the same region. This advantage contributes to a smooth encoding space with respect to the *actual* architecture performance instead of structures, which improves the efficiency of the downstream architecture search [6, 53, 123].

We argue that current architecture encoders limit the power of computation-aware architecture encoding for NAS. The major limitations lie in their representation power and the effectiveness of their pre-training objectives. Specifically, [53] uses shallow GRUs to encode computation, which is not sufficient to capture deep contextualized computation information. Moreover, their decoder is trained with the reconstruction loss via asynchronous message passing. This is very challenging in practice because directly learning the generative model based on a single architecture is not trivial. As a result, its pre-training is less effective and the downstream NAS performance is not as competitive as state-of-the-art structure-aware encoding methods. [6] proposes a computation-aware encoding method based on a

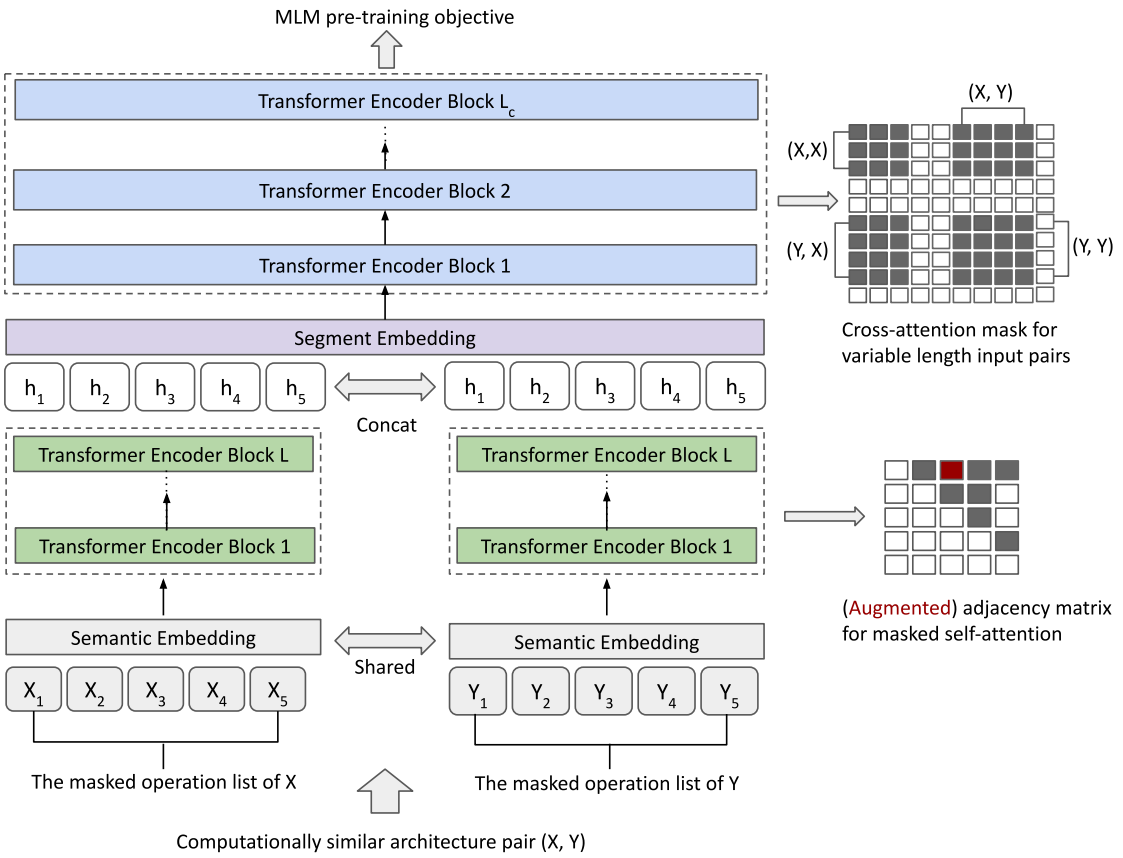


Figure 4.1: Overview of CATE. CATE takes computationally similar architecture pairs as the input. The model is trained to predict masked operators given the pairwise computational information. Apart from the cross-attention blocks, the pretrained Transformer encoder is used to extract architecture encodings for the downstream encoding-dependent NAS subroutines.

fixed transformation called path encoding, which shows outstanding performance under the predictor-based NAS subroutine. However, path encoding scales exponentially without truncation and it inevitably causes information loss with truncation. Moreover, path encoding exhibits worse generalization performance in outside search space compared to the adjacency matrix encoding since it could not generalize to unseen paths that are not included in the training search space.

In this work, we propose a new computation-aware neural architecture encoding method named CATE (**C**omputation-**A**ware **T**ransformer-based **E**ncoding) that alleviates the limita-

tions of existing computation-aware encoding methods. As shown in Figure 4.1, CATE takes paired computationally similar architectures as its input. Similar to BERT, CATE trains the Transformer-based model [124] using the masked language modeling (MLM) objective [74]. Each input architecture pair is corrupted by replacing a fraction of their operators with a special mask token. The model is trained to predict those masked operators from the corrupted architecture pair.

CATE differs from BERT [74] in two aspects. First, each prediction in LMs has its inductive bias given the contextual information from different positions. This, however, is not the case in architecture representation learning since the prediction distribution is uniform for any valid graph, making it difficult to directly learn the generative model from a single architecture. Therefore, we propose a pairwise pre-training scheme that encodes computationally similar architecture pairs through two Transformers with shared parameters. The two individual encodings are then concatenated, and the concatenated encoding is fed into another Transformer with a cross-attention encoder to encode the joint information of the architecture pair. Second, the fully-visible attention mask [125] could not be used for architecture representation learning because it does not reflect the single-directional flow (*e.g.* directed, acyclic, single-in-single-out) of neural architectures [95, 126]. Therefore, instead of using a bidirectional Transformer encoder as in BERT, we directly use the adjacency matrix to compute the causal mask [125]. The adjacency matrix is further augmented with the Floyd algorithm [127] to encode the long-range dependency of different operations. Together with the MLM objective, CATE is able to encode the computation of architectures and learn dense and deep contextualized architecture representations that contain both local and global computation information in neural architectures. This is important for architecture encodings to be generalized to outside search space beyond the training search space.

We compare CATE with eleven structure-aware and computation-aware architecture encoding methods under three major encoding-dependent subroutines as well as eight NAS algorithms on NAS-Bench-101 [10] (small), NAS-Bench-301 [128] (large), and an outside search space [6] to evaluate the effectiveness, scalability, and generalization ability of CATE.

Our results show that CATE is beneficial to the downstream architecture search, especially in the large search space. Specifically, we found the strongest NAS performance in all search spaces using CATE with a Bayesian optimization-based predictor subroutine together with a novel computation-aware search. Moreover, the outside search space experiment shows its superior generalization capability beyond the search space on which it was trained. Finally, our ablation studies show that the quality of CATE encodings and downstream NAS performance are non-decreasingly improved with more training architecture pairs, more cross-attention Transformer blocks and larger dimension of the feed-forward layer.

Related Work

Neural Architecture Search (NAS). NAS has been started with genetic algorithms [15–17] and recently becomes popular when [7, 24] gain significant attention. Since then, various NAS methods have been explored including sampling-based and gradient-based methods. Representative sampling-based methods include random search [116], evolutionary algorithms [18,19], local search [121,129], reinforcement learning [54,130], Bayesian optimization [21,131], Monte Carlo tree search [120,132] and Neural predictor [2,24,44–47,49,50,113,133]. Weight-sharing methods [38, 57] have become popular due to their computation efficiency. Based on weight-sharing, gradient-based methods are proposed to optimize the architecture selection with gradient decent [1,12,39–42,59,63,134]. For comprehensive surveys, we suggest referring to [5, 52].

Neural Architecture Encoding. Majority of existing NAS work use one-hot adjacency matrix to encode the structures of neural architectures. However, adjacency matrix-based encoding grows quadratically as the search space scales up. [10] proposes categorical adjacency matrix-based encoding to ensure fixed length encodings. They also propose continuous adjacency matrix-based encoding that is similar to DARTS [12], where the architecture is created by taking fixed number of edges with the highest continuous values. However, this approach is not easily applicable to some NAS algorithms such as regularized evolution [83] without major changes. Tabular encoding in the form of ConfigSpace [135] is often used for hyperparameter optimization [30, 31] and recently adopted by NAS-Bench-301 [128] to represent architectures by introducing categorical hyperparameters for each operation along each potential edge. Recent NAS methods [44, 46, 63, 120] use adjacency matrix as the input to LSTM/MLP/GNN to encode the structures of neural architectures in the latent space. [2] validates that pre-training architecture representations without using accuracies can better preserve the local structural relationship of neural architectures in the latent space. [136] proposes to learn architecture representations using contrastive learning to find low-dimensional embeddings. [137] studies various locality-based self-supervised objectives on the effect of architecture representations. One disadvantage of these methods is that they rely on a prior where the edit distance closeness between different architectures is a good indicator of the relative performance; however, structure-aware encodings may not be computationally unique unless some certain graph hashing is applied [10, 122]. [50, 138] use path encoding and its categorical and continuous variants, which encode computation of architectures so that isomorphic cells are mapped to the same encoding. [53] uses GRU-based asynchronous message passing to encode computation of architectures and the model is trained with the VAE loss. [139] proposes a two-sided variational encoder-decoder GNN to

learn smooth embeddings in various NAS search spaces. CATE is inspired by the advantage of computation encoding and addresses the drawbacks of [50, 53]. Another line of work is based on the intrinsic properties of the architectures. [140] generates architecture representations by using contrastive learning over data Jacobian matrix values computed based on different initializations, and the generated embeddings are independent of the parameterization of the search space.

Context Dependency. Our work is close to self-supervised learning in language models (LMs) [141]. In particular, ELMo [142] uses two shallow unidirectional LSTMs [143] to encode bidirectional text information, which is not sufficient for modeling deep interactions between the two directions. GPT-2 [144] proposes an autoregressive language modeling method with Transformer [124] to cover the left-to-right dependency and is further generalized by XLNet [145] which encodes bidirectional context. (Ro)BERT/BART/T5 [74, 125, 146, 147] use bidirectional Transformer encoder to encode both left and right context. In architecture representation learning, however, the attention mask in the encoder cannot be used to attend to all the operators because it does not reflect the single-directional flow of the computational graphs [95, 126].

Approach

Search Space

We restrict our search space to the cell-based architectures. Following the configuration in [10], each cell is a labeled directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ as a set of N nodes and $\mathcal{E}$ as a set of edges that connect the nodes. Each node $v_i \in \mathcal{V}$, $i \in [1, N]$ is associated with an operation selected from a predefined set of V operations, and the edges between different nodes are represented as an upper triangular binary adjacency matrix $\mathbf{A} \in \{0, 1\}^{N \times N}$.

Computation-aware Neural Architecture Encoder

Our proposed computation-aware neural architecture encoder is built upon the Transformer encoder architecture which consists of a semantic embedding layer and L Transformer blocks stacked on top. Given $\mathcal{G}$, each operation v_i is first fed into a semantic embedding layer of size d_e:

$$\mathbf{Emb}_i = \mathbf{Embedding}(v_i) \tag{4.1}$$

The embedded vectors are then contextualized at different levels of abstract. We denote the hidden state after l-th layer as $\mathbf{H}^l = [\mathbf{H}^l_1, ..., \mathbf{H}^l_N]$ of size d_h, where $\mathbf{H}^l = T(\mathbf{H}^{l-1})$ and T is a transformer block containing n_{head} heads. The l-th Transformer block is calculated as:

$$\mathbf{Q}_k = \mathbf{H}^{l-1}\mathbf{W}^l_{qk}, \mathbf{K}_k = \mathbf{H}^{l-1}\mathbf{W}^l_{kk}, \mathbf{V}_k = \mathbf{H}^{l-1}\mathbf{W}^l_{vk} \tag{4.2}$$

$$\hat{\mathbf{H}}^l_k = softmax(\frac{\mathbf{Q}_k\mathbf{K}^T_k}{\sqrt{d_h}} + \mathbf{M})\mathbf{V}_k \tag{4.3}$$

$$\hat{\mathbf{H}}^l = concatenate(\hat{\mathbf{H}}^l_1, \hat{\mathbf{H}}^l_2, \ldots, \hat{\mathbf{H}}^l_{n_{head}}) \tag{4.4}$$

$$\mathbf{H}^l = \mathbf{ReLU}(\hat{\mathbf{H}}^l\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{4.5}$$

where the initial hidden state $\mathbf{H}^0_i$ is $\mathbf{Emb}_i$, thus $d_e = d_h$. $\mathbf{Q}_k$, $\mathbf{K}_k$, $\mathbf{V}_k$ stand for "Query", "Key" and "Value" in the attention operation of the k-th head respectively. $\mathbf{M}$ is the attention mask in the Transformer, where $\mathbf{M}_{i,j} \in \{0, -\infty\}$ indicates whether operation j is a dependent operation of operation i. $\mathbf{W}_1 \in \mathbb{R}^{d_c \times d_{ff}}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d_c}$ denote the weights in the feed-forward layer.

Direct/Indirect Dependency Mask. A pair of nodes (operations) within an architecture

Algorithm 2: Floyd Algorithm

Input: the node set $\mathcal{V}$, the adjacent matrix $\mathbf{A}$
$\tilde{\mathbf{A}} \leftarrow \mathbf{A}$
for $k \in \mathcal{V}$ **do**
  for $i \in \mathcal{V}$ **do**
    for $j \in \mathcal{V}$ **do**
      $\tilde{\mathbf{A}}_{i,j} \quad | = \tilde{\mathbf{A}}_{i,k} \quad \& \quad \tilde{\mathbf{A}}_{k,j}$
    end for
  end for
end for
Output: $\tilde{\mathbf{A}}$

are dependent if there is either a directed edge that directly connects them (*local dependency*) or a path made of a series of such edges that indirectly connects them (*long-range dependency*). We create dependency masks for such pairs of nodes for both direct and indirect cases and use these dependency masks as the attention masks in the Transformer. Specifically, the direct dependency mask $\mathbf{M}^{Direct}$ and the indirect dependency mask $\mathbf{M}^{Indirect}$ can be created as follows:

$$\mathbf{M}_{i,j}^{Direct} = \begin{cases} 0, & if \quad A_{i,j} = 1 \\ -\infty, & if \quad A_{i,j} = 0 \end{cases}$$

$$\mathbf{M}_{i,j}^{Indirect} = \begin{cases} 0, & if \quad \tilde{A}_{i,j} = 1 \\ -\infty, & if \quad \tilde{A}_{i,j} = 0 \end{cases}$$

where $\mathbf{A}$ is the adjacency matrix and $\tilde{\mathbf{A}} = \mathit{Floyed}(\mathbf{A})$ is derived using Floyd algorithm in Algorithm 2.

Uni/Bidirectional Encoding. Finally, the final hidden vector $\mathbf{H}_N^l$ is used as the unidi-

rectional encoding for the architecture. We also considered encoding the architecture in a bidirectional manner, where both the output node hidden vector from the original DAG and the input node hidden vector from the reversed one are extracted and then concatenated together. However, our experiments show that bidirectional encoding performs worse than unidirectional encoding.

Pre-training CATE

Architecture Pair Sampling. We split the dataset into 95% training and 5% held-out test sets for our pairwise pre-training. To ensure that it does not scale with quadratic time complexity, we first sort the architectures based on their computational attributes $\mathbf{P}$ (*e.g.* number of parameters, FLOPs). We then employ a sliding window for each architecture x^i and its neighborhood $r(x^i) = \{y : |\mathbf{P}(x^i) - \mathbf{P}(y)| < \delta\}$, where δ is a hyperparameter for the pairwise computation constraint. Finally, we randomly select K distinct architectures $Y = \{y^1, \ldots, y^K\}, x^i \notin Y, Y \subset r(x^i)$ within the neighborhood to compose K architecture pairs $\{(x^i, y^1), \ldots, (x^i, y^K)\}$ for architecture x^i.

Pairwise Pre-training with Cross-Attention. Once the computationally similar architecture pair is composed, we randomly select 20% operations from each architecture within the pair for masking, where 80% of them are replaced with a $[MASK]$ token and the remaining 20% are replaced with a random token chosen from the predefined operation set. We apply padding to architectures that have nodes less than the maximum number of nodes N in one batch to handle variable length inputs. The joint representation $\mathbf{H}^L_{XY}$ is derived by concatenating $\mathbf{H}^L_X$ and $\mathbf{H}^L_Y$ followed by the summation of the corresponding segment embedding. Segment embedding acts as an identifier of different architectures during pre-training. We set it to be trainable and randomly initialized. The joint representation $\mathbf{H}^L_{XY}$ is then contextualized with another L_c-layer Transformer with the cross-attention mask $\mathbf{M}_c$

such that segments from the two architectures can attend to each other given the pairwise information. For example, given two architectures X with three nodes and Y with four nodes in Figure 4.1, X has access to the non-padded nodes of Y and itself, and same for Y. The cross-attention dimension of the encoder is denoted as d_c. The joint representation of the last layer is used for prediction. The model is trained by minimizing the cross-entropy loss computed using the predicted operations and the original operations.

Encoding-dependent NAS Subroutines

[6] identifies three major encoding-dependent subroutines included in existing NAS algorithms: *sample random architecture*, *perturb architecture*, and *train predictor model*. The *sample random architecture* subroutine includes random search [116]. The *perturb architecture* subroutine includes regularized evolution (REA) [18] and local search (LS) [121]. The *train predictor model* subroutine includes neural predictor [44, 46, 50], Bayesian optimization with Gaussian process (GP) [148], and Bayesian optimization with neural networks (DNGO) [110] which is much faster to fit compared to GP and scales linearly with large datasets rather than cubically.

Inspired by [121, 129], we found that LS (*perturb architecture*) can be combined with DNGO (*train predictor model*). We thus propose a DNGO-based computation-aware search using CATE called CATE-DNGO-LS. Specifically, we maintain a pool of sampled architectures and take iterations to add new ones. In each iteration, we pass all architecture encodings to the predictor trained 30 epochs with samples in the current pool. We select new architectures with top-5 predicted accuracy and add them to the pool. Assume there are M new architectures which become the new top-5 in the updated pool. We then select the nearest neighbors of the other (5-M) top-5 architectures in L2 distance in latent space and add them to the pool. Hence, there will be 5 to 10 new architectures added to the

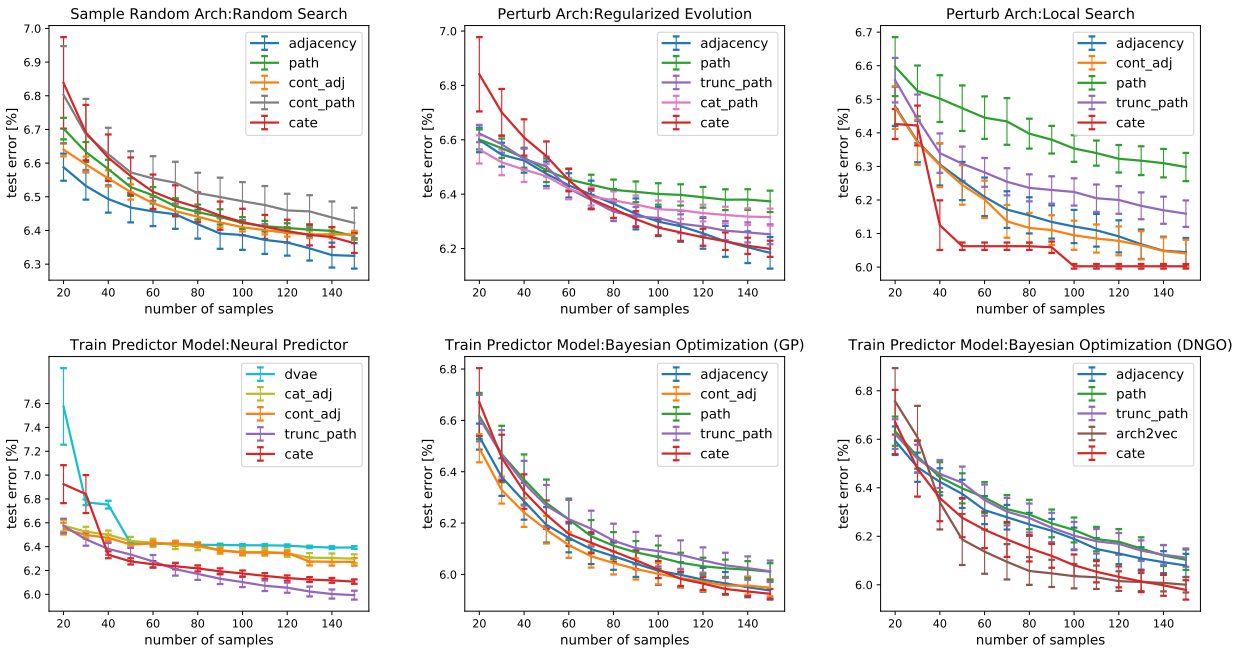


Figure 4.2: Comparison between CATE and other architecture encoding schemes under different subroutines on NAS-Bench-101: *sample random architecture* (top left), *perturb architecture* (top middle, top right), and *train predictor model* (bottom left, bottom middle, bottom right). It reports the test error of 200 independent runs given 150 queried architectures.

pool in each iteration. The search stops when the number of samples reaches a pre-defined budget.

Experiments

We describe two NAS benchmarks used in our experiments.

Pre-training

NAS-Bench-101. The NAS-Bench-101 search space [10] consists of $423,624$ architectures. Each architecture has its pre-computed validation and test accuracies on CIFAR-10. The cell includes up to 7 nodes and at most 9 edges with the first node as input and the last node as output. The intermediate nodes can be either 1×1 convolution, 3×3 convolution, or 3×3 max pooling. We use the number of network parameters as the computational attribute $\mathbf{P}$ for architecture pair sampling. We set δ to $2,000,000$ and K to 2. The ablation studies on δ

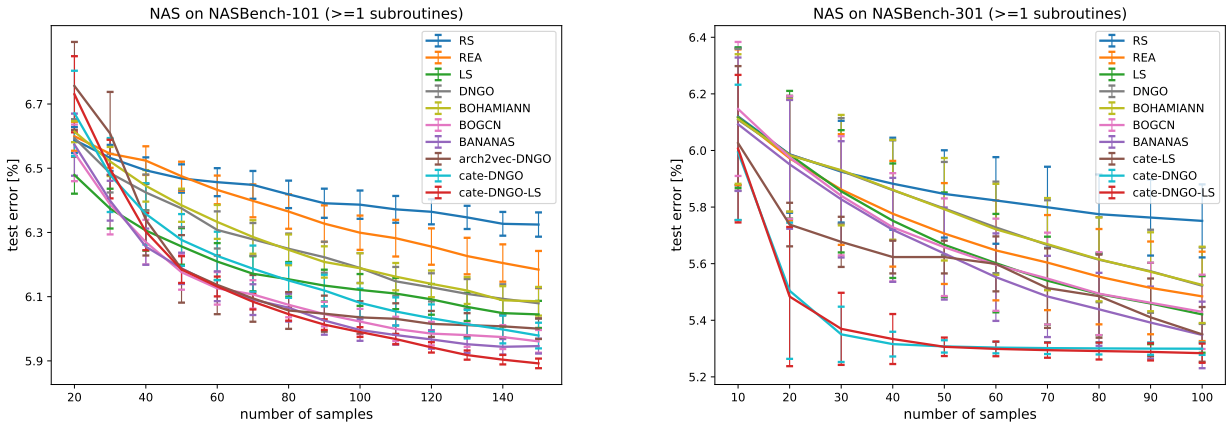


Figure 4.3: Comparison between CATE and SOTA NAS methods on NAS-Bench-101 (left) and NAS-Bench-301 (right). It reports the test error of 200 independent runs. The error bars denote the variance of the test error. The number of queried architectures is set to 150 for NAS-Bench-101 and 100 for NAS-Bench-301.

and K are summarized in Section 4. We split the dataset into 95% training and 5% held-out test sets for pre-training.

NAS-Bench-301 [128] is a new surrogate benchmark on the DARTS [12] search space that is much larger than NAS-Bench-101. It was created by fully training $60,000$ architectures that is *stratified by the NAS methods*[1] with a good coverage and then fitting a surrogate model that can estimate the accuracy (with noise) at epoch 100 and the training time for any of the remaining 10^{18} architectures. To convert the DARTS search space into one with the same input format as NAS-Bench-101, we add a summation node to make nodes represent operations and edges represent data flow. Following [113], we use the same cell for both normal and reduction cell, allowing roughly 10^9 DAGs without considering graph isomorphism. More details about the DARTS/NAS-Bench-301 and a cell transformation example are included in Appendix 1.3. We randomly sample $1,000,000$ architectures in this search space, and use the same data split used in NAS-Bench-101 for pre-training. We use network FLOPs as the computational attribute $\mathbf{P}$ for architecture pair sampling. We set δ to

[1]We suggest referring to C.2 in [128] for a detailed description on the data collection.

$5,000,000$ and K to 1. Since some NAS methods we compare against use the same GIN [93] surrogate model used in NAS-Bench-301, to ensure fair comparison, we thus followed [128] to use XGB-v1.0 and LGB-runtime-v1.0 which utilizes gradient boosted trees [149, 150] as the regression model.

Model and Training. We use a $L = 12$ layer Transformer encoder and a $L_c = 24$ layer cross-attention Transformer encoder, each has 8 attention heads. The hidden state size is $d_h = d_c = 64$ for all the encoders. The hidden dimension is $d_{ff} = 64$ for all the feed-forward layers. We employ AdamW [151] as our optimizer. The initial learning rate is 1e-3. The momentum parameters are set to 0.9 and 0.999. The weight decay is 0.01 for regular layer and 0 for dropout and layer normalization. We trained our model with batch size of 1024 on NVIDIA Quadro RTX 8000 GPUs. It takes around 4GB GPU memory for NAS-Bench-101 and 9GB GPU memory for NAS-Bench-301. The validation loss converges well after 10 epochs of pretraining, which takes 1.2 hours on NAS-Bench-101 and 7.5 hours on NAS-Bench-301.

Comparison with Different Encoding Schemes

In our first experiment, we compare CATE with eleven architecture encoding schemes under three major encoding-dependent subroutines described in Section 4 on NAS-Bench-101. These encoding schemes include (1-3) one-hot/categorical/continuous adjacency matrix encoding [10], (4-6) one-hot/categorical/continuous path encoding and (7-9) their corresponding truncated counterparts [50], (10) D-VAE [53], and (11) arch2vec [2]. For continuous encodings, we use L2 distance as the distance metric. To examine the effectiveness of the encoding schemes themselves, we compare different encoding schemes under the same search subroutine.

Figure 4.2 illustrates our results. For each subroutine, we show the top-five best-

performing encoding schemes. Overall, despite there is no overall best encoding, we found that CATE is among the top five across all the subroutines.

Specifically, for *sample random architecture* subroutine, random search using adjacency matrix encoding performs the best. The random search using continuous encodings performs slightly worse than the adjacency encodings possibly due to the discretization loss from vector space into a fixed number of bins of same size before the random sampling.

For *perturb architecture* subroutine, CATE is on par with or outperforms adjacency encoding and path encoding because it is pre-trained to preserve strong computation locality information. This advantage allows the evolution or local search to find architectures with similar performance in local neighborhood more easily. Interestingly, we observe very small deviation using local search with CATE. This indicates that it always converges to some certain local minimums across different initial seeds. Since NAS-Bench-101 already exhibits locality in edit distance, encoding computation makes architectures even closer in terms of accuracy and thus benefits the local search.

For *train predictor model* subroutine, we have four observations: 1) Adjacency matrix encodings perform less effective with neural predictor and DNGO. It is possibly that edit distance cannot fully reflect the closeness of architectures w.r.t their actual performance. 2) Path encoding performs well with neural predictor but worse than other encodings with Bayesian optimization. 3) D-VAE and arch2vec, two encodings learned via variational autoencoding, perform well only with some certain NAS methods. It could be attributed to their challenging training objective which easily leads to overfitting. 4) CATE is competitive with neural predictor and outperforms all the other encodings with Bayesian optimization. This is because neighboring computation-aware encodings correspond with similar accuracies. Moreover, the training objective in CATE is more efficient compared to the standard

NAS methods	NAS-Bench-101	NAS-Bench-301
Prev. SOTA [50]	5.92	5.35
CATE-DNGO-LS (ours)	**5.88**	**5.28**

Table 4.1: Comparison between CATE and state-of-the-arts. Final test error [%] given 150 queried architectures on NAS-Bench-101 and 100 queried architectures on NAS-Bench-301. The result is averaged over 200 independent runs.

VAE loss [99] used by D-VAE and arch2vec.

Comparison with Different NAS Methods

In our second experiment, we compare the neural architecture search performance based on CATE encodings with state-of-the-art NAS algorithms on NAS-Bench-101 and NAS-Bench-301. Existing NAS algorithms contain one or more encoding-dependent subroutines.

We consider six NAS algorithms that contain one encoding-dependent subroutine: random search (RS) [116] (*sample random arch.*), regularized evolution (REA) [18] (*perturb arch.*), local search (LS) [121] (*perturb arch.*), DNGO [110] (*train predictor*), BOHAMIANN [152] (*train predictor*), arch2vec-DNGO [2] (*train predictor*), and two NAS algorithms that contain more than one encoding-dependent subroutine: BOGCN [46] (*perturb arch.*, *train predictor*) and BANANAS [50] (*sample random arch.*, *perturb arch.*, *train predictor*).

We compare these eight existing NAS algorithms with CATE-DNGO: a NAS algorithm based on CATE encodings with the DNGO subroutine (*train predictor*), and CATE-DNGO-LS: a NAS algorithm based on CATE encodings with the combination of DNGO and LS subroutines (*train predictor*, *perturb arch.*) as described in Section 4.

Figure 4.3 and Table 4.1 summarize our results. We have three major findings from Figure 4.3: 1) Architecture encoding matters especially in the large search space. The right plot shows that CATE-DNGO and CATE-DNGO-LS in DARTS search space not only

converge faster but also lead to better final search performance given the same budgets. 2) Local search (LS) is a strong baseline in both small and large search spaces. As mentioned in Section 4, performing LS using CATE leads to better results compared to other encodings. 3) NAS algorithms that use more than one encoding-dependent subroutine in general perform better than NAS algorithms with just one subroutine. Specifically, BOGCN and BANANAS that have multiple subroutines perform better than the single-subroutine NAS algorithms such as REA, DNGO, and BOHAMIANN. Moreover, CATE-DNGO-LS leads to the best performing result in both NAS-Bench-101 and NAS-Bench-301 search spaces. Meanwhile, the improvement of CATE-DNGO-LS versus CATE-DNGO shrinks in larger search space, indicating that the larger search space is more challenging to encode.

NAS-Bench-301 uses a surrogate model trained on 60k architectures to predict the performance of all the other architectures in the DARTS search space. The performance of the other architectures, however, can be inaccurate. Given that, we further validate the effectiveness of CATE-DNGO-LS in the *actual* DARTS search space by training the queried architectures from scratch. We set the budget to 100 and 300 queries, separately. Each queried architecture is trained for 50 epochs with a batch size of 96, using 32 initial channels and 8 cell layers. The average validation error of the last 5 epochs is computed as the label. These values are chosen to be close to the proxy model used in DARTS. It takes about 3.3 GPU days to finish the search with 100 quries and 10.3 GPU days with 300 queries. See Figure 4.4 for the best found cells. To ensure fair comparison, we compare CATE-DNGO-LS to methods $[2, 12, 50, 116]$ that use the common test evaluation script which is to train for 600 epochs with cutout and auxiliary tower.

Table 4.2 summarizes our results. As shown, CATE-DNGO-LS (small budget) achieves competitive performance (2.55% avg. test error) with much less search cost and CATE-

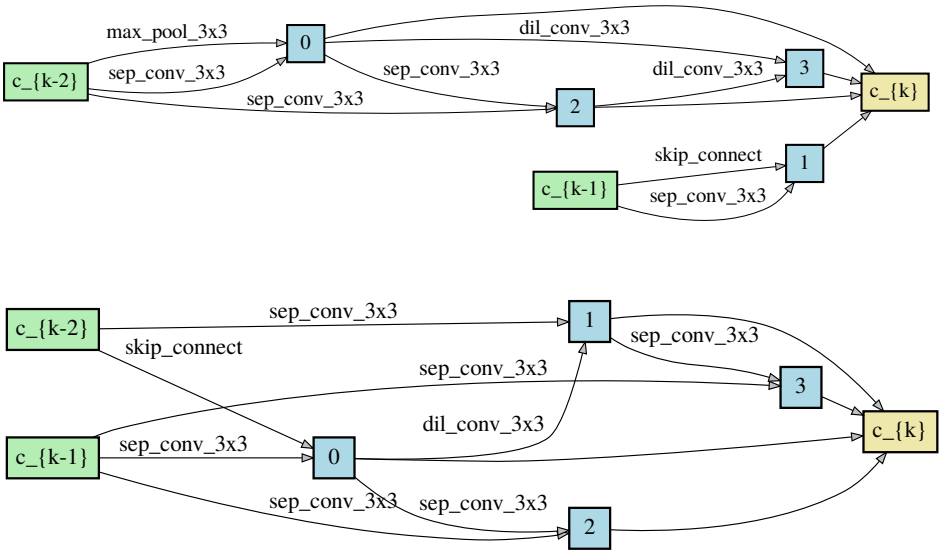


Figure 4.4: Top: Best found cell from CATE-DNGO-LS given the budget of 100 samples. Bottom: Best found cell from CATE-DNGO-LS given the budget of 300 samples.

DNGO-LS (large budget) achieves superior performance (2.46% avg. test error) with similar search cost compared to other sampling-based search methods [2, 50] in the actual DARTS search space. This is consistent with our observation in NAS-Bench-301. We report the transfer learning results on ImageNet [71] in Table 4.3.

Ablation Study

Finally, we conduct ablation studies on different hyperparameters involved in CATE. We use CATE-DNGO as the NAS method and report the final NAS test error [%] given 150 queried architectures on NAS-Bench-101. The result is averaged over 200 independent runs.

Architecture Pair Sampling Hyperparameters

We plot the histogram of model parameters on NAS-Bench-101 in Figure 4.6. As shown, the architectures are neither normally nor uniformly distributed in this search space in terms of model parameters. This motivates us to use a sliding window-based architecture pair selection to avoid the unbalanced sampling as proposed in Section 4. The choice of δ and

NAS Methods	Avg. Test Error (%)	Params (M)	Search Cost (GPU days)
RS [116]	3.29 ± 0.15	**3.2**	4
DARTS [12]	2.76 ± 0.09	3.3	4
BANANAS [50]	2.67 ± 0.07	3.6	11.8
arch2vec-BO [2]	2.56 ± 0.05	3.6	9.2
CATE-DNGO-LS (small budget)	2.55 ± 0.08	3.5	**3.3**
CATE-DNGO-LS (large budget)	**2.46 ± 0.05**	4.1	10.3

Table 4.2: NAS results in DARTS search space using CIFAR-10.

K and their effects on the downstream NAS are summarized in Table 4.4. We found that strong computation locality (*i.e.* small δ) usually leads to better results. The choice of neighborhood size K does not have a significant effect on NAS performance. Therefore, we choose small K for faster pretraining. For NAS-Bench-301, we use the FLOPs as the computational attributes $\mathbf{P}$ and observe the same trend as in NAS-Bench-101 on the selection of δ and K.

Transformer Hyperparameters

We studied the effect of the number of cross-attention Transformer blocks L_c and the hidden dimension of the feed-forward layer d_{ff} on CATE. We fix δ and K for pre-training as mentioned in Section 4. The downstream NAS result is summarized in Table 4.5. It shows that larger L_c and d_{ff} usually lead to better NAS performance, which indicates that deep contextualized representations are beneficial to downstream NAS.

Choice of Mask Type

We studied pretraining CATE with direct/indirect dependency mask and summarize its downstream NAS results in Table 4.6. CATE trained with indirect dependency mask outper-

NAS Methods	Params (M)	Mult-Adds (M)	Top-1 Test Error (%)
SNAS [59]	4.3	522	27.3
DARTS [12]	4.7	574	26.7
BayesNAS [131]	**4.0**	**440**	26.5
arch2vec-BO [2]	5.2	580	25.5
BANANAS (ours)	5.1	576	26.3
CATE-DNGO-LS (small budget)	5.0	556	26.1
CATE-DNGO-LS (large budget)	5.8	642	**25.0**

Table 4.3: Transfer learning results on ImageNet using CATE.

δ \ K	1	2	4	8
1×10^6	6.02	5.95	5.99	5.95
2×10^6	6.02	**5.94**	6.04	5.96
4×10^6	**5.94**	6.03	6.05	5.99
8×10^6	6.05	6.04	6.11	6.04

Table 4.4: Effects of δ and K on architecture pair sampling.

forms the direct one in both benchmarks, indicating that capturing long-range dependency helps preserve computation information in the encodings.

Uni/Bidirectional Encoding

As mentioned in Section 4, we also considered encoding the architecture in a bidirectional manner where both the output node hidden vector from the original DAG and the input node hidden vector from the reversed one are extracted and then concatenated together. Note that d_c in the cross-attention Transformer encoder will be doubled due to the concatenation. We compare the results of unidirectional and bidirectional encodings in Table 4.7. As

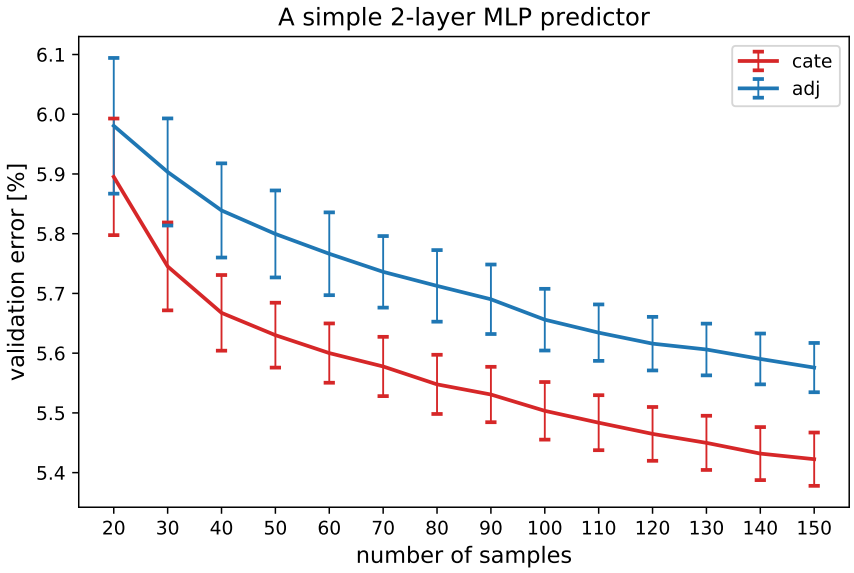


Figure 4.5: Performance on the out-of-training search space. It reports the validation error of 500 independent runs.

d_{ff} \ L_c	6	12	24
64	6.07	5.99	5.95
128	6.01	**5.94**	5.95
256	5.97	**5.94**	**5.94**

Table 4.5: Effects of L_c and d_{ff} on pretraining CATE.

shown, bidirectional encoding does not necessarily improve the results. Therefore, we keep unidirectional encoding in other experiments due to its simplicity and better performance.

Corruption Rate

By default, we randomly select 20% operations from each architecture within the pair for masking in the pairwise pre-training. We also experimented corruption rates of 15% and 30%. As shown in Table 4.8, overall, we find that the corruption rate has a limited effect on the NAS performance. Note that the number of nodes in our search space is much smaller compared to the number of tokens in the sequence modeling tasks. Given that, using larger

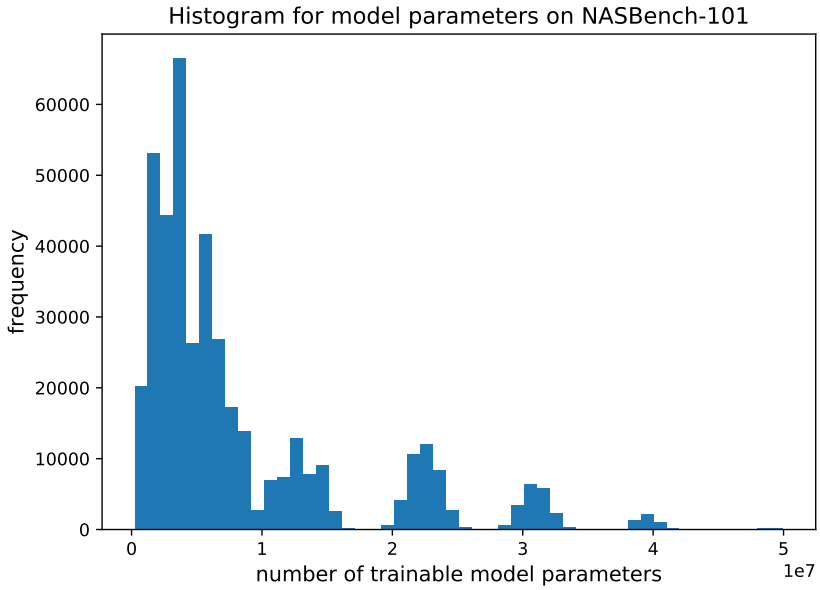


Figure 4.6: Histogram of model parameters on NAS-Bench-101.

Mask type	NAS-Bench-101	NAS-Bench-301
Direct	6.03	5.35
Indirect	**5.94**	**5.30**

Table 4.6: Direct/Indirect dependency mask selection.

corruption rate may slow down the training convergence and result in degraded performance. Based on these results, we use 20% corruption rate for other experiments.

Conclusion

In this chapter, we presented CATE, a new computation-aware architecture encoding method based on Transformers. Unlike encodings with fixed transformations, we show that the computation information of neural architectures can be contextualized through a pairwise learning scheme trained with MLM. Our experimental results show its effectiveness and scalability along with three major encoding-dependent NAS subroutines in both small and large search spaces. We also show its superior generalization capability outside the training search space. We anticipate that the methods presented in this work can be extended to

Encoding	NAS-Bench-101	NAS-Bench-301
Unidirectional	**5.88**	**5.28**
Bidirectional	5.89	5.30

Table 4.7: Unidirectional encoding vs. bidirectional encoding. We report the final NAS test error [%] given 150 queried architectures on NAS-Bench-101 and 100 queried architectures on NAS-Bench-301. The result is averaged over 200 independent runs.

Corruption Rate	NAS-Bench-101	NAS-Bench-301
15%	5.89	**5.28**
20%	**5.88**	**5.28**
30%	5.93	5.29

Table 4.8: NAS results under different corruption rates.

encode even larger search spaces (*e.g.* TuNAS [153]) to study the effectiveness of different downstream NAS algorithms.

Chapter 5

NAS-Bench-x11 and the Power of Learning Curve

In the past few years, algorithms for neural architecture search (NAS) have been used to automatically find architectures that achieve state-of-the-art performance on various datasets [5, 7, 12, 18]. In 2019, there were calls for reproducible and fair comparisons within NAS research [61, 87, 116, 154] due to both the lack of a consistent training pipeline between papers and experiments with not enough trials to reach statistically significant conclusions. These concerns spurred the release of tabular benchmarks, such as NAS-Bench-101 [10] and NAS-Bench-201 [11], created by fully training all architectures in search spaces of size 423 624 and 6 466, respectively. These benchmarks allow researchers to easily simulate NAS experiments, making it possible to run fair NAS comparisons and to run enough trials to reach statistical significance at very little computational cost or carbon emissions [155]. Recently, to extend the benefits of tabular NAS benchmarks to larger, more realistic NAS search spaces which cannot be evaluated exhaustively, it was proposed to construct *surrogate benchmarks* [156]. The first such surrogate benchmark is NAS-Bench-301 [156], which models the DARTS [12] search space of size 10^{18} architectures. It was created by fully training 60 000 architectures (both drawn randomly and chosen by top NAS methods) and then fitting a surrogate model that can estimate the performance of all of the remaining architectures. Since 2019, dozens of papers have used these NAS benchmarks to develop new algorithms [3, 46, 49, 50, 138].

An unintended side-effect of the release of these benchmarks is that it became significantly easier to devise *single fidelity* NAS algorithms: when the NAS algorithm chooses to evaluate an architecture, the architecture is fully trained and only the validation accuracy at the final epoch of training is outputted. This is because NAS-Bench-301 only contains the

architectures' accuracy at epoch 100, and NAS-Bench-101 only contains the accuracies at epochs 4, 12, 36, and 108 (allowing single fidelity or very limited multi-fidelity approaches). NAS-Bench-201 does allow queries on the entire learning curve (every epoch), but it is smaller in size (6 466) than NAS-Bench-101 (423 624) or NAS-Bench-301 (10^{18}). In a real world experiment, since training architectures to convergence is computationally intensive, researchers will often run *multi-fidelity* algorithms: the NAS algorithm can train architectures to any desired epoch. Here, the algorithm can make use of speedup techniques such as learning curve extrapolation (LCE) [23, 24, 26, 28]. Although multi-fidelity techniques are often used in the hyperparameter optimization community [28, 30, 31, 35, 36], they have been under-utilized by the NAS community in the last few years.

In this work, we fill in this gap by releasing NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11, surrogate benchmarks with full learning curve information for train, validation, and test loss and accuracy for all architectures, significantly extending NAS-Bench-101, NAS-Bench-301, and NAS-Bench-NLP [157], respectively. With these benchmarks, researchers can easily incorporate multi-fidelity techniques, such as early stopping and LCE into their NAS algorithms. Our technique for creating these benchmarks can be summarized as follows. We use a training dataset of architectures (drawn randomly and chosen by top NAS methods) with good coverage over the search space, along with full learning curves, to fit a model that predicts the full learning curves of the remaining architectures. We employ three techniques to fit the model: *(1)* dimensionality reduction of the learning curves, *(2)* prediction of the top singular value coefficients, and *(3)* noise modeling. These techniques can be used in the future to create new NAS benchmarks as well. To ensure that our surrogate benchmarks are highly accurate, we report statistics such as Kendall Tau rank correlation and Kullback Leibler divergence between ground truth learning curves and

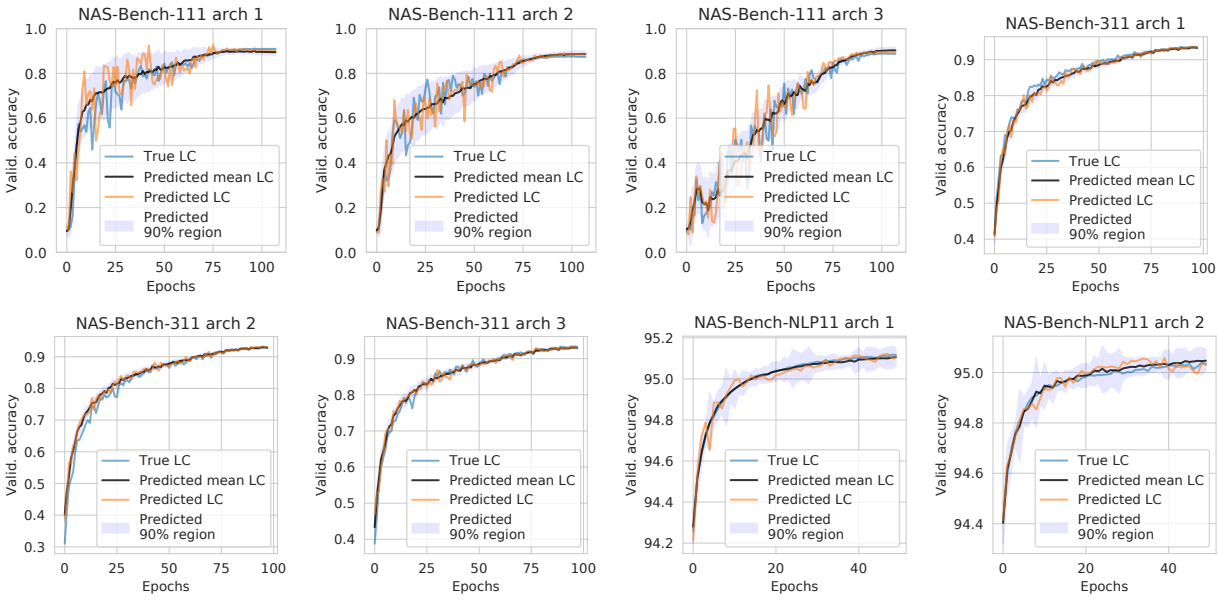


Figure 5.1: Each image shows a true learning curve vs. a learning curve predicted by one of our surrogate models, with and without predicted noise modeling. We also plot the 90% confidence interval of the predicted noise distribution.

predicted learning curves on separate test sets. See Figure 5.1 for examples of predicted learning curves on the test sets.

To demonstrate the power of using the full learning curve information, we present a framework for converting single-fidelity NAS algorithms into multi-fidelity algorithms using LCE. We apply our framework to popular single-fidelity NAS algorithms, such as regularized evolution [18], local search [121], and BANANAS [50], all of which claimed state-of-the-art upon release, showing that they can be further improved across four search spaces. Finally, we also benchmark multi-fidelity algorithms such as Hyperband [30] and BOHB [31] alongside single-fidelity algorithms. Overall, our work bridges the gap between different areas of AutoML and will allow researchers to easily develop effective multi-fidelity and LCE techniques in the future.

Our contributions. We summarize our main contributions below.

- We develop a technique to create surrogate NAS benchmarks that include the full training

information for each architecture, including train, validation, and test loss and accuracy learning curves. This technique can be used to create future NAS benchmarks on any search space.

- We apply our technique to create NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11, which allow researchers to easily develop multi-fidelity NAS algorithms that achieve higher performance than single-fidelity techniques.
- We present a framework for converting single-fidelity NAS algorithms into multi-fidelity NAS algorithms using learning curve extrapolation, and we show that our framework allows popular state-of-the-art NAS algorithms to achieve further improvements.

Related Work

NAS has been studied since at least the late 1980s [15, 16, 158] and has recently seen a resurgence [7,18,21,57,132,159]. Weight sharing algorithms have become popular due to their computational efficiency [12, 38–43]. Recent advances in performance prediction [2, 44–50] and other iterative techniques [31, 51] have reduced the runtime gap between iterative and weight sharing techniques. For detailed surveys on NAS, we suggest referring to [5, 52].

Learning curve extrapolation

Several methods have been proposed to estimate the final validation accuracy of a neural network by extrapolating the learning curve of a partially trained neural network. Techniques include fitting the partial curve to an ensemble of parametric functions [23], predicting the performance based on the partial trained neural network configurations [24], summing the training losses [25], using the basis functions as the output layer of a Bayesian neural network [26], using previous learning curves as basis function extrapolators [27], using the positive-definite covariance kernel to capture a variety of training curves [28], or using a Bayesian recurrent neural network [29]. While in this work we focus on multi-fidelity op-

Benchmark	Size	Queryable	Based on	Full train info
NAS-Bench-101	423k	yes		no
NAS-Bench-201	6k	yes		yes
NAS-Bench-NLP	10^{53}	no		no
NAS-Bench-301	10^{18}	yes	DARTS	no
NAS-Bench-ASR	8k	yes		yes
NAS-Bench-111	423k	yes	NAS-Bench-101	yes
NAS-Bench-311	10^{18}	yes	DARTS	yes
NAS-Bench-NLP11	10^{22}	yes	NAS-Bench-NLP	yes

Table 5.1: Overview of existing NAS benchmarks. We introduce NAS-Bench-111, -311, and -NLP11.

timization utilizing learning curve-based extrapolation, another main category of methods lie in bandit-based algorithm selection [30–34], and the fidelities can be further adjusted according to the previous observations or a learning rate scheduler [35–37].

NAS benchmarks

NAS-Bench-101 [10], a tabular NAS benchmark, was created by defining a search space of size 423 624 unique architectures and then training all architectures from the search space on CIFAR-10 until 108 epochs. However, the train, validation, and test accuracies are only reported for epochs 4, 12, 36, and 108, and the training, validation, and test losses are not reported. NAS-Bench-1shot1 [86] defines a subset of the NAS-Bench-101 search space that allows one-shot algorithms to be run. NAS-Bench-201 [11] contains 15 625 architectures, of which 6 466 are unique up to isomorphisms. It comes with full learning curve information on three datasets: CIFAR-10 [160], CIFAR-100 [160], and ImageNet16-120 [161]. Recently, NAS-Bench-201 was extended to NATS-Bench [162] which searches over architecture size as well as architecture topology.

Virtually every published NAS method for image classification in the last three years evaluates on the DARTS search space with CIFAR-10 [163]. The DARTS search space [12] consists of 10^{18} neural architectures, making it computationally prohibitive to create a tabular benchmark. To overcome this fundamental limitation and query architectures in this much larger search space, NAS-Bench-301 [156] evaluates various regression models trained on a sample of 60 000 architectures that is carefully created to cover the whole search space. The surrogate models allow users to query the validation accuracy (at epoch 100) and training time for any of the 10^{18} architectures in the DARTS search space. However, since the surrogates do not predict the entire learning curve, it is not possible to run multi-fidelity algorithms.

NAS-Bench-NLP [157] is a search space for language modeling tasks. The search space consists of 10^{53} LSTM-like architectures, of which 14 322 are evaluated on Penn Tree Bank [164], containing the training, validation, and test losses/accuracies from epochs 1 to 50. Since only 14 322 of 10^{53} architectures can be queried, this dataset cannot be directly used for NAS experiments. NAS-Bench-ASR [165] is a recent tabular NAS benchmark for speech recognition. The search space consists of 8 242 architectures with full learning curve information. For an overview of NAS benchmarks, see Table 5.1.

Generating Learning Curves

In this section, we describe our techniques to create a surrogate model that outputs realistic learning curves. We apply these techniques to create NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11. Our techniques apply to any type of learning curve, including train/test losses and accuracies. For simplicity, the following presentation assumes validation accuracy learning curves.

Given a search space $\mathbb{D}$, let $(\boldsymbol{x}_i, \boldsymbol{y}_i) \sim \mathbb{D}$ denote one datapoint, where $\boldsymbol{x}_i \in \mathbb{R}^d$ is the

architecture encoding (e.g., one-hot adjacency matrix [6, 10]), and $\boldsymbol{y}_i \in [0,1]^{E_{\max}}$ is a learning curve of validation accuracies drawn from a distribution $Y(\boldsymbol{x}_i)$ based on training the architecture for $E_{\max}$ epochs on a fixed training pipeline with a random initial seed. Each learning curve $\boldsymbol{y}_i$ can be decomposed into two parts: one part that is deterministic and depends only on the architecture encoding, and another part that is based on the inherent noise in the architecture training pipeline.

Formally, $\boldsymbol{y}_i = \mathbb{E}[Y(\boldsymbol{x}_i)] + \boldsymbol{\epsilon}_i$, where $\mathbb{E}[Y(\boldsymbol{x}_i)] \in [0,1]^{E_{\max}}$ is fixed and depends only on $\boldsymbol{x}_i$, and $\boldsymbol{\epsilon}_i \in [0,1]^{E_{\max}}$ comes from a noise distribution Z_i with expectation 0 for all epochs. In practice, $\mathbb{E}[Y(\boldsymbol{x}_i)]$ can be estimated by averaging a large set of learning curves produced by training architecture $\boldsymbol{x}_i$ with different initial seeds. We represent such an estimate as $\bar{\boldsymbol{y}}_i$.

Our goal is to create a surrogate model that takes as input any architecture encoding $\boldsymbol{x}_i$ and outputs a distribution of learning curves that mimics the ground truth distribution. We assume that we are given two datasets, $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$, of architecture and learning curve pairs. We use $\mathcal{D}_{\text{train}}$ (often size $> 10\,000$) to train the surrogate, and we use $\mathcal{D}_{\text{test}}$ for evaluation. We describe the process of creating $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ for specific search spaces in the next section. In order to predict a learning curve distribution for each architecture, we split up our approach into two separate processes: we train a model $f : \mathbb{R}^d \to [0,1]^{E_{\max}}$ to predict the deterministic part of the learning curve, $\bar{\boldsymbol{y}}_i$, and we train a noise model $p_\phi(\boldsymbol{\epsilon} \mid \bar{\boldsymbol{y}}, \boldsymbol{x})$, parameterized by ϕ, to simulate the random draws from Z_i. See Figure 5.2 for a summary of our entire surrogate creation method (assuming SVD).

Surrogate Model Training

Training a model f to predict mean learning curves is a challenging task, since the training datapoints $\mathcal{D}_{\text{train}}$ consist only of a single (or few) noisy learning curve(s) $\boldsymbol{y}_i$ for each $\boldsymbol{x}_i$. Furthermore, $E_{\max}$ is typically length 100 or larger, meaning that f must predict a high-

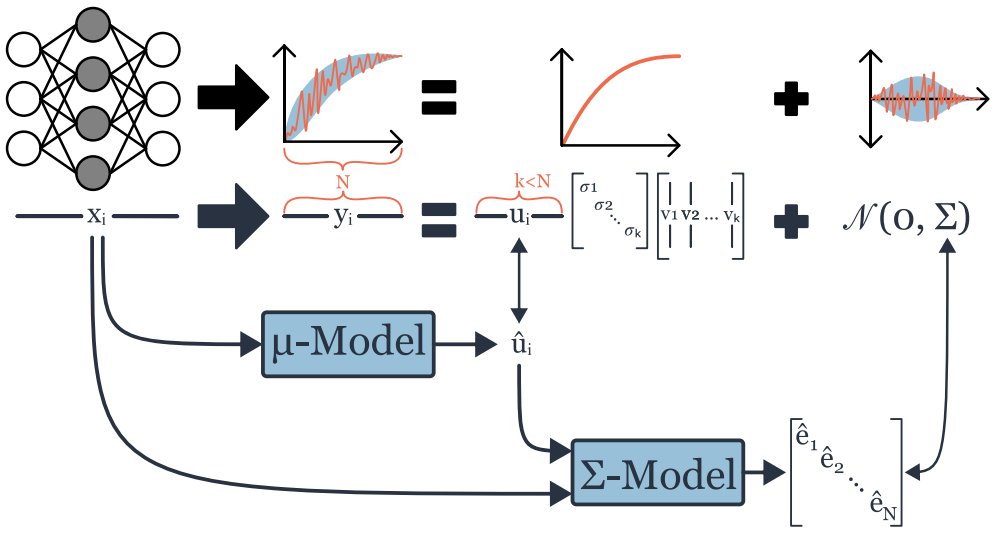


Figure 5.2: A summary of our approach to create surrogate benchmarks that output realistic learning curves. Compression and decompression functions are learned using the training set of learning curves (in the figure, SVD is shown, but a VAE can also be used). The compression also helps to de-noise the learning curves. A model (μ-model) is trained to predict the compressed (de-noised) learning curves given the architecture encoding. A separate model (Σ-model) is trained to predict each learning curve's noise distribution, given the architecture encoding and predicted compressed learning curve. A realistic learning curve can then be outputted by decompressing the predicted learning curve and sampling noise from the noise distribution.

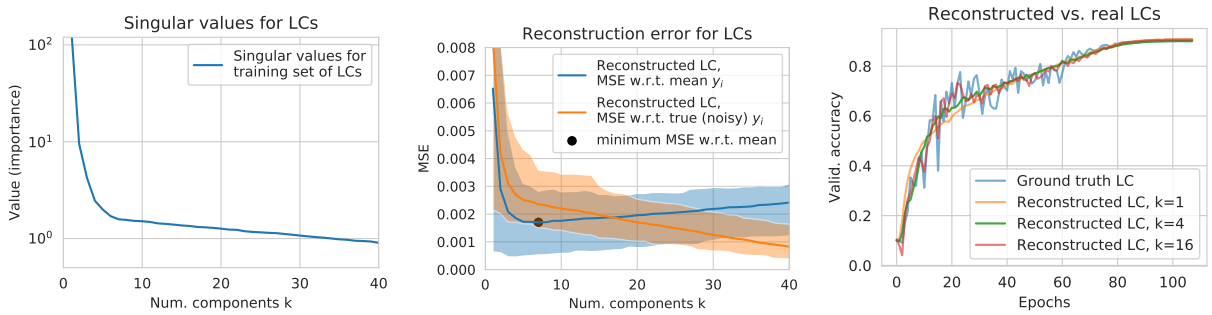


Figure 5.3: The singular values from the SVD decomposition of the learning curves (LC) (left). The MSE of a reconstructed LC, showing that $k = 6$ is closest to the true mean LC, while larger values of k overfit to the noise of the LC (middle). An LC reconstructed using different values of k (right).

dimensional output. We propose a technique to help with both of these challenges: we use the training data to learn compression and decompression functions $c_k : [0,1]^{E_{\max}} \to [0,1]^k$ and $d_k : [0,1]^k \to [0,1]^{E_{\max}}$, respectively, for $k \ll E_{\max}$. The surrogate is trained to predict *compressed* learning curves $c_k(\boldsymbol{y}_i)$ of size k from the corresponding architecture encoding $\boldsymbol{x}_i$, and then each prediction can be reconstructed to a full learning curve using d_k. A good compression model should not only cause the surrogate prediction to become significantly faster and simpler, but should also reduce the noise in the learning curves, since it would only save the most important information in the compressed representations. That is, $(d_k \circ c_k)(\boldsymbol{y}_i)$ should be a less noisy version of $\boldsymbol{y}_i$. Therefore, models trained on $c_k(\boldsymbol{y}_i)$ tend to have better generalization ability and do not overfit to the noise in individual learning curves.

We test two compression techniques: singular value decomposition (SVD) [166] and variational autoencoders (VAEs) [99], and we show later that SVD performs better. We give the details of SVD here and describe the VAE compression algorithm in Section 5.

Formally, we take the singular value decomposition of a matrix S of dimension $(|\mathcal{D}_{\text{train}}|, E_{\max})$ created by stacking together the learning curves from all architectures in $\mathcal{D}_{\text{train}}$. Performing the truncated SVD on the learning curve matrix S allow us to create

functions c_k and d_k that correspond to the optimal linear compression of S. In Figure 5.3 (left), we see that for architectures in the NAS-Bench-101 search space, there is a steep dropoff of importance after the first six singular values, which intuitively means that most of the information for each learning curve is contained in its first six singular values. In Figure 5.3 (middle), we compute the mean squared error (MSE) of the reconstructed learning curves $(d_k \circ c_k)(\boldsymbol{y}_i)$ compared to a test set of ground truth learning curves averaged over several initial seeds (approximating $\mathbb{E}[Y(\boldsymbol{x}_i)]$). The lowest MSE is achieved at $k = 6$, which implies that $k = 6$ is the sweet spot where the compression function minimizes reconstruction error without overfitting to the noise of individual learning curves. We further validate this in Figure 5.3 (right) by plotting $(d_k \circ c_k)(\boldsymbol{y}_i)$ for different values of k. Now that we have compression and decompression functions, we train a surrogate model with $\boldsymbol{x}_i$ as features and $c_k(\boldsymbol{y}_i)$ as the label, for architectures in $\mathcal{D}_{\text{train}}$. We test LGBoost [150], XGBoost [167], and MLPs for the surrogate model.

Noise Modeling

The final step for creating a realistic surrogate benchmark is to add a noise model, so that the outputs are *noisy* learning curves. We first create a new dataset of predicted $\boldsymbol{\epsilon}_i$ values, which we call residuals, by subtracting the reconstructed mean learning curves from the real learning curves in $\mathcal{D}_{\text{train}}$. That is, $\hat{\boldsymbol{\epsilon}}_i = \boldsymbol{y}_i - (d_k \circ c_k)(\boldsymbol{y}_i)$ is the residual for the ith learning curve. Since the training data only contains one (or few) learning curve(s) per architecture $\boldsymbol{x}_i$, it is not possible to accurately estimate the distribution Z_i for each architecture without making further assumptions. We assume that the noise comes from an isotropic Gaussian distribution, and we consider two other noise assumptions: *(1)* the noise distribution is the same for all architectures, and *(2)* for each architecture, the noise in a small window of epochs are iid. In Section 5, we estimate the extent to which all of these assumptions hold

true. Assumption *(1)* suggests two potential noise models: *(i)* a simple sample standard deviation statistic, $\boldsymbol{\sigma} \in \mathbb{R}_{+}^{E_{\max}}$, where

$$\sigma_j = \sqrt{\frac{1}{|\mathcal{D}_{\text{train}}| - 1} \sum_{i=1}^{|\mathcal{D}_{\text{train}}|} \hat{\epsilon}_{i,j}^2}.$$

To sample the noise using this model, we sample from $\mathcal{N}(\mathbf{0}, \text{diag}(\boldsymbol{\sigma}))$. *(ii)* The second model is a Gaussian kernel density estimation (GKDE) model [168] trained on the residuals to create a multivariate Gaussian kernel density estimate for the noise. Assumption *(2)* suggests one potential noise model: *(iii)* a model that is trained to estimate the distribution of the noise over a window of epochs of a specific architecture. For each architecture and each epoch, the model is trained to estimate the sample standard deviation of the residuals within a window centered around that epoch.

Evaluation of the Surrogate Model

Overall, we test two compression methods, three surrogate models, and three noise models. For each approach, we evaluate both the predicted mean learning curves and the predicted noisy learning curves using held-out test sets $\mathcal{D}_{\text{test}}$. To evaluate the mean learning curves, we measure the coefficient of determination (R^2) [169] and Kendall Tau (KT) rank correlation [170], both at the final epoch and averaged over all epochs. To measure KT rank correlation for a specific epoch n, we find the number of concordant, P, and discordant, Q, pairs of predicted and true learning curve values for that epoch. The number of concordant pairs is given by the number of pairs, $((\hat{y}_{i,n}, y_{i,n}), (\hat{y}_{j,n}, y_{j,n}))$, where either both $\hat{y}_{i,n} > \hat{y}_{j,n}$ and $y_{i,n} > y_{j,n}$, or both $\hat{y}_{i,n} < \hat{y}_{j,n}$ and $y_{j,n} < y_{i,n}$. We can then calculate $KT = \frac{P-Q}{P+Q}$. While this metric can be used to compare surrogate predictions and has been used in prior work [156], the KT value is affected by the inherent noise in the learning curves (even an

oracle would achieve a KT value smaller than 1.0, because architecture training is noisy). Finally, we evaluate the Kullback Leibler (KL) divergence between the ground truth distribution of noisy learning curves, and the predicted distribution of noisy learning curves on a test set. Since we can only estimate the ground truth distribution, we assume the ground truth is an isotropic Gaussian distribution. Then we measure the KL divergence between the true and predicted learning curves for architecture i by the following formula:

$$D_{KL}(\boldsymbol{y}_i||\hat{\boldsymbol{y}}_i) = \frac{1}{2E_{\max}}\left[\log\frac{|\Sigma_{\hat{\boldsymbol{y}}_i}|}{|\Sigma_{\boldsymbol{y}_i}|} - E_{\max} + (\boldsymbol{\mu}_{\boldsymbol{y_i}} - \boldsymbol{\mu}_{\hat{\boldsymbol{y}}_{\boldsymbol{i}}})^T\Sigma^{-1}_{\hat{\boldsymbol{y}}_i}(\boldsymbol{\mu}_{\boldsymbol{y_i}} - \boldsymbol{\mu}_{\hat{\boldsymbol{y}}_{\boldsymbol{i}}}) + tr\left\{\Sigma^{-1}_{\hat{\boldsymbol{y}}_i}\Sigma_{\boldsymbol{y}_i}\right\}\right]$$

where $\Sigma_{\{\boldsymbol{y}_i,\hat{\boldsymbol{y}}_i\}}$ is a diagonal matrix with the entries $\Sigma_{\{\boldsymbol{y}_i,\hat{\boldsymbol{y}}_i\}_{k,k}}$ representing the sample variance of the k-th epoch, and $\boldsymbol{\mu}_{\{\boldsymbol{y}_i,\hat{\boldsymbol{y}}_i\}}$ is the sample mean for either learning curves $\{\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i\}$.

Surrogate Benchmark Creation

Now we describe the creation of NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11. As described above, we test two different compression methods (SVD, VAE), three different surrogate models (LGB, XGB, MLP), and three different noise models (stddev, GKDE, sliding window) for a total of eighteen distinct approaches. See Section 5 for a full ablation study, and Table 5.2 for a summary using the best techniques for each search space. See Figure 5.1 for a visualization of predicted learning curves from the test set of each search space using these models.

First, we describe the creation of NAS-Bench-111. Since the NAS-Bench-101 tabular benchmark [10] consists only of accuracies at epochs 4, 12, 36, and 108 (and without losses), we train a new set of architectures and save the full learning curves. Similar to prior work [156, 171], we sample a set of architectures with good overall coverage while also fo-

cusing on the high-performing regions exploited by NAS algorithms. Specifically, we sample 861 architectures generated uniformly at random, 149 architectures generated by 30 trials of white2019bananas, local search, and regularized evolution, and all 91 architectures which contain fewer than five nodes, for a total of 1101 architectures. We kept our training pipeline as close as possible to the original pipeline. See Section 5 for the full training details. We find that SVD-LGB-GKDE achieves the best performance. Because the tabular benchmark already exists, we can substantially improve the accuracy of our surrogate by using the accuracies from the tabular benchmark (at epochs 4, 12, 36, 108) as additional features along with the architecture encoding. This substantially improves the performance of the surrogate, as shown in Table 5.2. Note that the large difference between the average KT and last epoch KT values show that the learning curves are very noisy (which is also evidenced in Figure 5.1).

Next, we create NAS-Bench-311 by using the training data from NAS-Bench-301, which consists of 40 000 random architectures along with 26 000 additional architectures generated by evolution [18], Bayesian optimization [50, 172, 173], and one-shot [12, 39, 174, 175] techniques in order to achieve good coverage over the search space. Again, SVD-LGB-GKDE achieves the best performance, which achieves an average and last epoch KT values of 0.728 and 0.788, respectively. This is comparable to the final KT of 0.817 reported by NAS-Bench-301 [156], despite optimizing for the full learning curve rather than only the final epoch. Furthermore, our KL divergences in Table 5.2 surpasses the top value of 16.4 reported by NAS-Bench-301 (for KL divergence, lower is better).

Finally, we create NAS-Bench-NLP11 by using the NAS-Bench-NLP dataset consisting of 14 322 architectures drawn uniformly at random. Due to the extreme size of the search space (10^{53}), we restrict architectures to a maximum of 12 nodes (reducing the size to 10^{22}), and we

Benchmark	Avg. R^2	Final R^2	Avg. KT	Final KT	Avg. KL
NAS-Bench-111	0.529	0.630	0.531	0.645	2.016
NAS-Bench-111 (w. accs)	0.630	0.853	0.611	0.794	1.710
NAS-Bench-311	0.779	0.800	0.728	0.788	0.905
NAS-Bench-NLP11	0.326	0.314	0.505	0.475	-
NAS-Bench-NLP11 (w. accs)	0.878	0.895	0.878	0.844	-

Table 5.2: Evaluation of the surrogate benchmarks on test sets. For NAS-Bench-111 and NAS-Bench-NLP11, we use architecture accuracies as additional features to improve performance.

achieve an average and final epoch KT of 0.505 and 0.475, respectively. To create a stronger surrogate, we add the first three epochs of the learning curve as features in the surrogate. This improves the average and final epoch KT values to 0.878 and 0.844, respectively. To use this surrogate, any architecture to be predicted with the surrogate must be trained for three epochs. Note that the small difference between the average and last epoch KT values indicates that the learning curves have very little noise, which can also be seen in Figure 5.1. Since there are no architectures trained multiple times on the NAS-Bench-NLP dataset [157], we cannot compute the KL divergence.

The Power of Learning Curve Extrapolation

Now we describe a simple framework for converting single-fidelity NAS algorithms to multi-fidelity NAS algorithms using learning curve extrapolation techniques. We show that this framework is able to substantially improve the performance of popular algorithms such as regularized evolution [18], white2019bananas [50], and local search [121, 129].

A single-fidelity algorithm is an algorithm which iteratively chooses an architecture based on its history, which is then fully trained to $E_{\max}$ epochs. To exploit parallel resources, many single-fidelity algorithms iteratively output several architectures at a time, instead of just

one. Our framework makes use of learning curve extrapolation (LCE) techniques [23, 24] to predict the final validation accuracies of all architecture choices after only training for a small number of epochs. After each iteration of getting candidates, only the architectures predicted by `LCE()` to begin the top percentage of validation accuracies of history are fully trained. For example, when the framework is applied to local search, in each iteration, all neighbors are trained to E_{few} epochs, and only the most promising architectures are trained up to E_{max} epochs. This simple modification can substantially improve the runtime efficiency of popular NAS algorithms by weeding out unpromising architectures before they are fully trained.

Any LCE technique can be used, and in our experiments in Section 5, we use weighted probabilistic modeling (WPM) [23] and learning curve support vector regressor (LcSVR) [24]. The first technique, WPM [23], is a function that takes a partial learning curve as input, and then extrapolates it by fitting the learning curve to a set of parametric functions, using MCMC to sample the most promising fit. The second technique, LcSVR [24], is a model-based learning curve extrapolation technique: after generating an initial set of training architectures, a support vector regressor is trained to predict the final validation accuracy from the architecture encoding and partial learning curve.

Experiments

In this section, we benchmark single-fidelity and multi-fidelity NAS algorithms, including popular existing single-fidelity and multi-fidelity algorithms, as well as algorithms created using our framework defined in the previous section. In the experiments, we use our three surrogate benchmarks defined in Section 5, as well as NAS-Bench-201.

NAS Algorithms

For single-fidelity algorithms, we implemented random search (RS) [116], local search (LS) [121, 129], regularized evolution (REA) [18], and white2019bananas [50]. For multi-fidelity bandit-based algorithms, we implemented Hyperband (HB) [30] and Bayesian optimization Hyperband (BOHB) [31]. For all methods, we use the original implementation whenever possible. See Section 5 for a description, implementation details, and hyperparameter details for each method. Finally, we use our framework from Section 5 to create six new multi-fidelity algorithms: white2019bananas, LS, and REA are each augmented using WPM and LcSVR. This gives us a total of 12 algorithms.

Experimental Setup

For each search space, we run each algorithm for a total wall-clock time that is equivalent to running 500 iterations of the single-fidelity algorithms for NAS-Bench-111 and NAS-Bench-311, and 100 iterations for NAS-Bench-201 and NAS-Bench-NLP11. For example, the average time to train a NAS-Bench-111 architecture to 108 epochs is roughly 10^3 seconds, so we set the maximum runtime on NAS-Bench-111 to roughly $5 \cdot 10^5$ seconds. We run 30 trials of each NAS algorithm and compute the mean and standard deviation.

Results

We evaluate BANANAS, LS, and REA compared to their augmented WPM and SVR versions in Figure 5.4 (NAS-Bench-311) and Section 5 (all other search spaces). Across four search spaces, we see that WPM and SVR improve all algorithms in almost all settings. The improvements are particularly strong for the larger NAS-Bench-111 and NAS-Bench-311 search spaces. We also see that for each single-fidelity algorithm, the LcSVR variant often outperforms the WPM variant. This suggests that model-based techniques for extrapolating

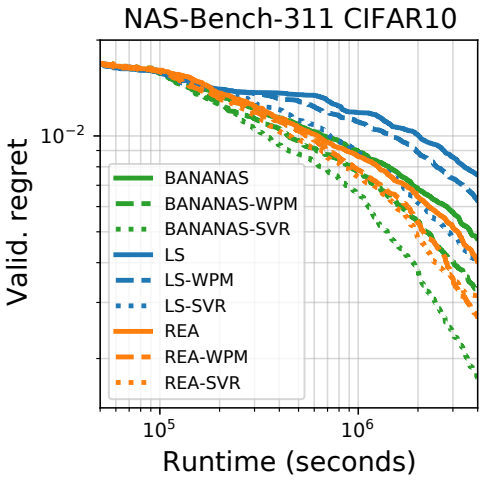


Figure 5.4: LCE framework applied to single-fidelity algorithms.

learning curves are more reliable than extrapolating each learning curve individually, which has also been noted in prior work [48].

In Figure 5.5, we compare single- and multi-fidelity algorithms on four search spaces, along with the three SVR-based algorithms from Figure 5.4. Across all search spaces, an SVR-based algorithm is the top-performing algorithm. Specifically, white2019bananas-SVR performs the best on NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11, and LS-SVR performs the best on NAS-Bench-201. Note that HB and BOHB may not perform well on search spaces with low correlation between the relative rankings of architectures using low fidelities and high fidelities (such as NAS-Bench-201 [11]) since HB-based methods will predict the final accuracy of partially trained architectures directly from the last trained accuracy (i.e., extrapolating the learning curve as a constant after the last seen accuracy). On the other hand, SVR-based approaches use a model that can learn more complex relationships between accuracy at an early epoch vs. accuarcy at the final epoch, and are therefore more robust to this type of search space.

In Section 5, we perform an ablation study on the epoch at which the SVR and WPM

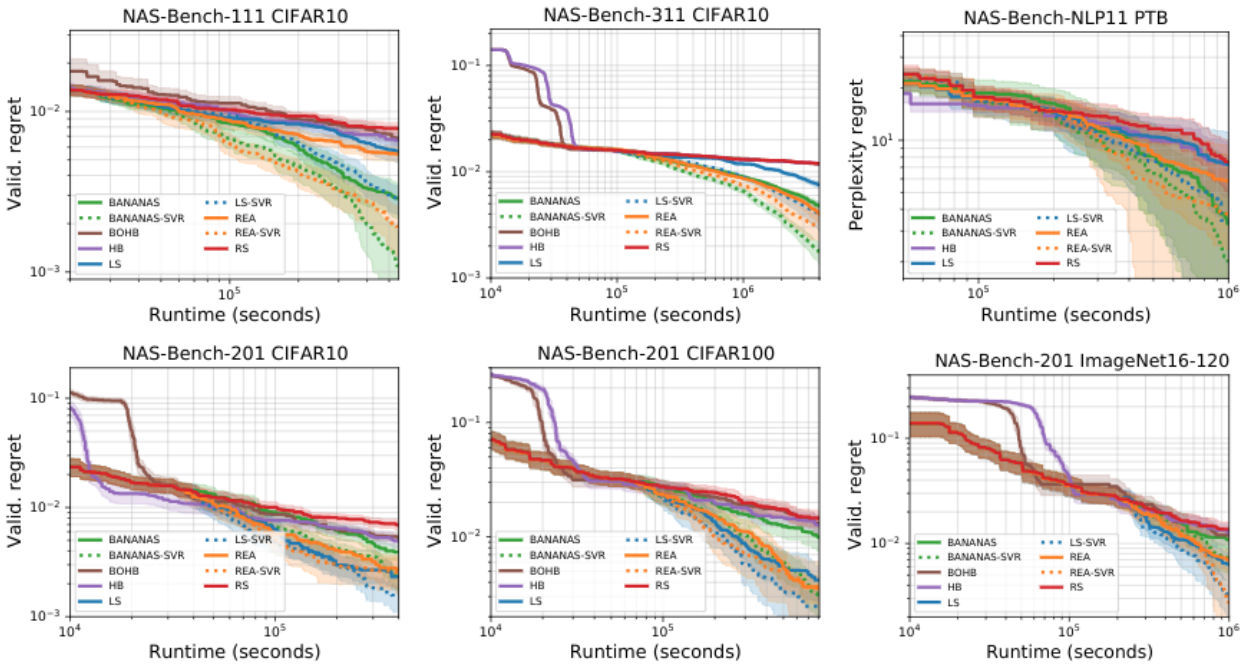


Figure 5.5: NAS results on six different combinations of search spaces and datasets. For every setting, an SVR augmented method performs best.

methods start extrapolating in our framework (i.e., we ablate E_{few}). We find that for most search spaces, running SVR and WPM based NAS methods by starting the LCE at roughly 20% of the total number of epochs performs the best. Any earlier, and there is not enough information to accurately extrapolate the learning curve. Any later, and the LCE sees diminishing returns because less time is saved by early stopping the training.

Ablation Study

Evaluation with All Combinations of Models.

We consider three different noise models as described in Section 5. Recall that we create a new dataset of predicted $\boldsymbol{\epsilon}_i$ values, which we call residuals, by subtracting the reconstructed mean learning curves from the real learning curves in $\mathcal{D}_{\text{train}}$. That is, $\hat{\boldsymbol{\epsilon}}_i = \boldsymbol{y}_i - (d_k \circ c_k)(\boldsymbol{y}_i)$ is the residual for the ith learning curve. Our three noise models are based on two different assumptions: *(1)* the noise distribution is the same for all architectures, and *(2)* for each

architecture, the noise in a small window of epochs are iid.

Now we evaluate these assumptions. In Figure 5.6 (left), we plot the residuals from the NAS-Bench-301 training set at five different epochs, showing that the distributions are roughly Gaussian, across all architectures. Recall that noise model *(i)* is a simple standard deviation statistic computed for each epoch independently, across all architectures. In Figure 5.6 (middle), we plot the autocorrelation function (ACF) averaged over all training learning curves on NAS-Bench-301. We see that there is very little autocorrelation in the learning curves, which justifies the use of the first noise model. Recall that our second noise model uses Gaussian kernel density estimation (GKDE) [168] across all learning curves. This is essentially the same as the first noise model but with the ability to capture the small amount of autocorrelation present. Finally, recall that our our third noise model does not assume that the residual distribution is similar across all architectures. Instead, it estimates the standard deviation for each epoch using a sliding window of size 10 across the epochs for each architecture. See Figure 5.6 (right) for the 90% confidence intervals of the residuals at each epoch on NAS-Bench-301. Although the sliding window noise model has the benefit of capturing different distributions for different architectures, we see that the standard deviation steadily decreases as the epoch number increases, meaning that the noise in a small window of epochs are not perfectly iid.

We run a full ablation study by testing all eighteen combinations of {SVD, VAE}, {LGB, XGB, MLP}, and {GKDE, STD, window} for NAS-Bench-111 and NAS-Bench-311. Recall that, as explained in Section 5, the first four metrics evaluate only the prediction of the mean learning curves using a held-out test set, so the noise model has no effect on the first four metrics (average R^2, final R^2, average KT, and final KT). For the final two metrics (average KL and final KL), we use a test set of learning curves consisting of five seeds of architectures,

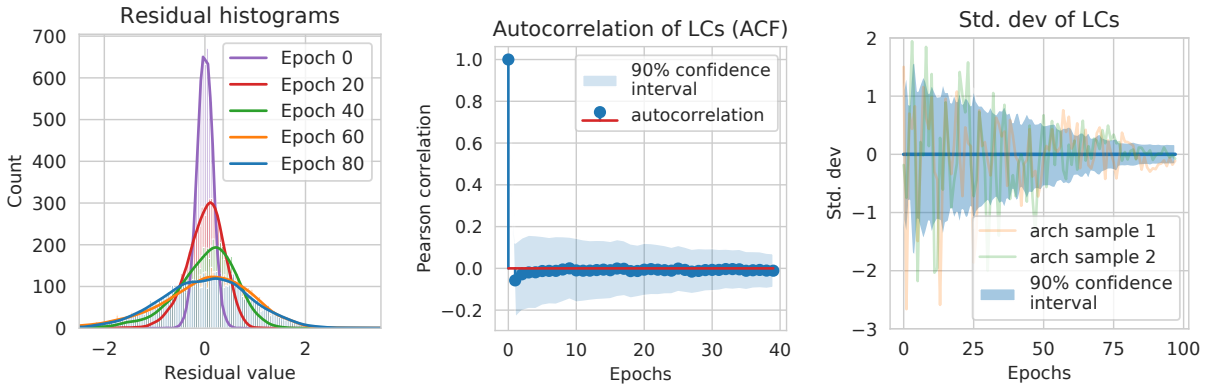


Figure 5.6: A plot of the residuals across all architectures for five different epochs (left). We see that the distributions are roughly Gaussian. A plot of the autocorrelation function (ACF) averaged over all training learning curves (middle). We see that there is only a small amount of autocorrelation. A plot of the 90% confidence intervals of the residuals at each epoch (right). All plots use the NAS-Bench-301 learning curve training set.

so that we can estimate the KL divergence between the real learning curve distribution and the predicted distribution. Note that none of the NAS-Bench-NLP architectures were trained more than once, so we are unable to test the noise models for NAS-Bench-NLP11. Across NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11, we see that SVD-LGB performs substantially better than all of the other options for the model. We also see that GKDE outperforms all other noise models on NAS-Bench-111 and NAS-Bench-311. Therefore, SVD-LGB-GKDE is the best overall combination for surrogate benchmark creation for these NAS search spaces. See Figure 5.7 for additional images of learning curves predicted by NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11. This is a continuation of Figure 5.1.

Surrogate Training Time

We give more details for the surrogate training. For NAS-Bench-111, as discussed in Section 5, we created a new set of trained architectures with the full learning curve information. We kept the training pipeline nearly the same as in the orginal NAS-Bench-101 repository. However, instead of the TPU v2 acceleator as in the original work, we used an RTX 3070. We

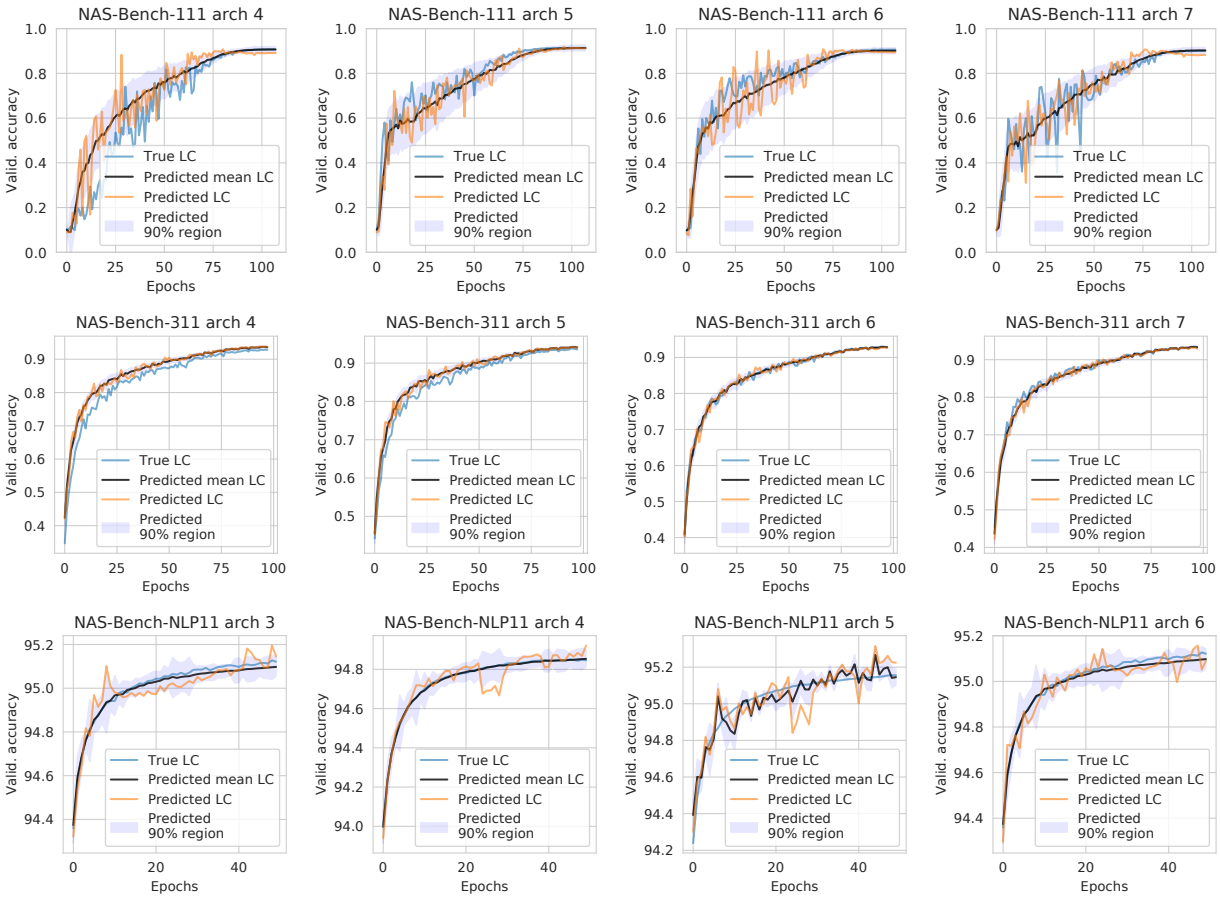


Figure 5.7: Learning curves, continued from Figure 5.1. Each image shows a true learning curve vs. a learning curve predicted by one of our surrogate models, with and without predicted noise modeling. We also plot the 90% confidence interval of the predicted noise distribution.

needed to change the batch size from 256 to 200 to account for this hardware change, which we found had a negligible affect on the final accuracy. We trained 1101 new architectures and used this new set as the new "ground-truth" when training and evaluating NAS-Bench-111. As explained in Section 5, the accuracies from the original NAS-Bench-101 benchmark were used as features to improve the performance of our surrogate, but not used as ground truth.

For NAS-Bench-311, training was straightforward. We used the original NAS-Bench-301 dataset, which already achieves good coverage [156], and we did not use any additional featuers.

For NAS-Bench-NLP11, as described earlier, it is challenging to create an accurate surrogate benchmark because there are only 14 322 evaluated architectures for a search space of total size 10^{53}. Therefore, we used two techniques to improve performance. First, we used a subset of the search space, restricting the architectures to a maximum of 12 nodes (reducing the size to 10^{22}), and we added the validation accuracies from the first three epochs of training each architecture, as features. These two techniques were shown to substantially improve the performance of NAS-Bench-NLP11. On an RTX 3070, training architectures from NAS-Bench-NLP takes about 90 seconds per epoch. Although adding in the first three epochs substantially improves the accuracy of our surrogate benchmark, it comes at the cost of query time. While NAS-Bench-111 and NAS-Bench-311 take under one second to query, a query to NAS-Bench-NLP11 now requires training an architecture for three epochs. Note that this is still a 15× speedup over performing NAS directly without a surrogate benchmark.

The Effect of Different Fidelities

We evaluate the effect of different fidelities on NAS-Bench-311. In Figure 5.8, we plot the validation regret of the SVR and WPM-based algorithms after 2×10^6 seconds, varying the initial fidelity (epoch) from which the learning curve is extracted, from 10 to 40. That is, the leftmost points run LCE by extrapolating from epoch 10 to epoch 100, and the rightmost points run LCE by extrapolating from epoch 40 to epoch 100. Note that there is a tradeoff between time saved (from only evaluating to 10 epochs vs 40) and accuracy of LCE (extrapolating from 10 epochs is more challenging than from 40 epochs). We see that overall, epoch 20 performs the best. Notably, white2019bananas-SVR and REA-SVR (two of the best-performing algorithms across all search spaces) achieve top performance at epoch 20.

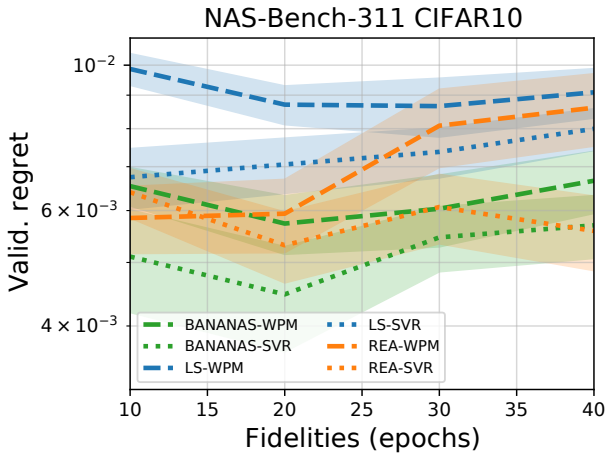


Figure 5.8: Different fidelities and their effect on NAS performance on NAS-Bench-311. The wall-clock time [s] is set to 2e6. The result are reported across 30 seeds.

Conclusion

In this work, we released three benchmarks for neural architecture search based on three popular search spaces, which substantially improve the capability of existing benchmarks due to the availability of the full learning curve for train/validation/test loss and accuracy for each architecture. Our techniques to generate these benchmarks, which includes singular value decomposition of the learning curve and noise modeling, can be used to model the full learning curve for future surrogate NAS benchmarks as well.

Furthermore, we demonstrated the power of the full learning curve information by introducing a framework that converts single-fidelity NAS algorithms into multi-fidelity NAS algorithms that make use of learning curve extrapolation techniques. This framework improves the performance of recent popular single-fidelity algorithms which claimed to be state-of-the-art upon release.

While we believe our surrogate benchmarks will help advance scientific research in NAS, a few guidelines and limitations are important to keep in mind. As with prior surrogate benchmarks [156], we give the following two caveats. *(1)* We discourage evaluating NAS

methods that use the same internal techniques as those used in the surrogate model. For example, any NAS method that makes use of variational autoencoders or XGBoost should not be benchmarked using our VAE-XGB surrogate benchmark. *(2)* As the surrogate benchmarks are likely to evolve as new training data is added, or as better techniques for training a surrogate are devised, we recommend reporting the surrogate benchmark version number whenever running experiments.

We also note the following strengths and limitations for specific benchmarks. Our NAS-Bench-111 surrogate benchmark gives strong performance even with just 1 101 architectures used as training data, due to the existence of the four extra validation accuracies from NAS-Bench-101 that can be used as additional features. While these features help to achieve high predictive power over the entire search space, the only limitation is that this technique likely cannot be used to create future surrogate benchmarks: we hope that all future surrogate works will save the full learning curve information from the start so that the creation of an after-the-fact extended benchmark is not necessary.

Since the NAS-Bench-NLP11 surrogate benchmark achieves significantly stronger performance when the accuracy of the first three epochs are added as features, we recommend using this benchmark by training architectures for three epochs before querying the surrogate. Therefore, benchmarking NAS algorithms are slower than for NAS-Bench-111 and NAS-Bench-311, but NAS-Bench-NLP11 still offers a 15× speedup compared to a NAS experiment without this benchmark. We still release the version of NAS-Bench-NLP11 that does not use the first three accuracies as features but with a warning that the observed NAS trends may differ from the true NAS trends. We expect that the performance of these benchmarks will improve over time, as data for more trained architectures become available.

Benchmark	Avg. R^2	Final R^2	Avg. KT	Final KT	Avg. KL	Final KL
NAS-Bench-111						
SVD-LGB-GKDE	**0.630**	**0.853**	**0.611**	**0.794**	**1.641**	0.516
SVD-LGB-STD	**0.630**	**0.853**	**0.611**	**0.794**	2.768	**0.383**
SVD-LGB-window	**0.630**	**0.853**	**0.611**	**0.794**	24.402	3.303
SVD-XGB-GKDE	0.329	0.378	0.408	0.429	2.743	0.580
SVD-XGB-STD	0.329	0.378	0.408	0.429	4.867	0.503
SVD-XGB-window	0.329	0.378	0.408	0.429	38.457	16.172
SVD-MLP-GKDE	0.195	0.065	0.330	0.290	4.599	0.762
SVD-MLP-STD	0.195	0.065	0.330	0.290	8.417	0.848
SVD-MLP-window	0.195	0.065	0.330	0.290	82.180	15.711
VAE-LGB-GKDE	0.267	0.218	0.462	0.617	3.788	0.829
VAE-LGB-STD	0.267	0.218	0.462	0.617	6.866	0.972
VAE-LGB-window	0.267	0.218	0.462	0.617	53.866	19.820
VAE-XGB-GKDE	0.311	0.272	0.453	0.559	3.828	0.828
VAE-XGB-STD	0.311	0.272	0.453	0.559	6.940	0.969
VAE-XGB-window	0.311	0.272	0.453	0.559	55.654	19.614
VAE-MLP-GKDE	0.218	0.007	0.386	0.369	4.583	0.844
VAE-MLP-STD	0.218	0.007	0.386	0.369	8.386	1.001
VAE-MLP-window	0.218	0.007	0.386	0.369	83.481	19.091
NAS-Bench-311						
SVD-LGB-GKDE	**0.779**	**0.800**	**0.728**	**0.788**	**0.503**	**0.548**
SVD-LGB-STD	**0.779**	**0.800**	**0.728**	**0.788**	0.919	1.036
SVD-LGB-window	**0.779**	**0.800**	**0.728**	**0.788**	1.566	4.083
SVD-XGB-GKDE	0.522	0.546	0.607	0.654	1.783	3.272
SVD-XGB-STD	0.522	0.546	0.607	0.654	3.271	5.958
SVD-XGB-window	0.522	0.546	0.607	0.654	5.282	19.432
SVD-MLP-GKDE	0.564	0.549	0.573	0.603	15.727	29.057
SVD-MLP-STD	0.564	0.549	0.573	0.603	28.833	52.515
SVD-MLP-window	0.564	0.549	0.573	0.603	45.071	167.140
VAE-LGB-GKDE	0.431	0.447	0.568	0.616	5.995	13.486
VAE-LGB-STD	0.431	0.447	0.568	0.616	11.015	24.836
VAE-LGB-window	0.431	0.447	0.568	0.616	17.510	79.773
VAE-XGB-GKDE	0.397	0.427	0.577	0.624	6.520	16.739
VAE-XGB-STD	0.397	0.427	0.577	0.624	11.978	30.368
VAE-XGB-window	0.397	0.427	0.577	0.624	18.883	97.485
VAE-MLP-GKDE	0.509	0.520	0.584	0.619	13.545	33.851
VAE-MLP-STD	0.509	0.520	0.584	0.619	24.770	61.455
VAE-MLP-window	0.509	0.520	0.584	0.619	38.593	196.246
NAS-Bench-NLP11						
SVD-LGB	0.878	**0.895**	**0.878**	**0.844**	-	-
SVD-XGB	0.877	0.806	0.856	0.820	-	-
SVD-MLP	**0.893**	0.856	0.742	0.692	-	-
VAE-LGB	0.862	0.847	0.766	0.770	-	-
VAE-XGB	0.875	0.860	0.720	0.687	-	-
VAE-MLP	0.867	0.871	0.667	0.685	-	-

Table 5.3: Evaluation of the surrogate benchmarks on test sets, with all combinations of models. For NAS-Bench-111 and NAS-Bench-NLP11, we use architecture accuracies as additional features to improve performance. As explained in Section 5, no architectures in the NAS-Bench-NLP dataset were trained more than once, so we do not compute KL divergence for NAS-Bench-NLP11.

Chapter 6

Conclusion

In this dissertation, my goal is to present to the readers all of the essence of Neural Architecture Search (NAS), through which I discuss how I have contributed to the development of NAS from one-shot approach to sampling-based approach. Chapter 1 – *Introduction* provides with the necessary knowledge to understand and build a vanilla NAS system. I walk through the history and fundamentals of NAS together with drawbacks of existing approaches, leading to the research on neural architecture representations and learning curve predictions.

Chapter 2 – *EFFICIENT ONE-SHOT NAS VIA HIERARCHICAL MASKING* starts discussing my research on one-shot NAS with based on weight-sharing. When I was developing the work for this chapter in 2019, one-shot NAS had just started from the seminal work of [38, 57]. Specifically, differentiable Neural Architecture Search [12] is one of the most popular Neural Architecture Search (NAS) methods for its search efficiency and simplicity, accomplished by jointly optimizing the model weight and architecture parameters in a weight-sharing supernet via gradient-based algorithms. Despite its potential, it still adopt hand-designed heuristics to generate architecture candidates. Specifically, at the end of the search phase, the operations with the largest architecture parameters will be selected to form the final architecture, with the implicit assumption that the values of architecture parameters reflect the operation strength. While much has been discussed about the proxyless supernet training [8], the architecture selection process has received little attention. I formulate NAS as a pruning process and propose a multi-level architecture encoding scheme to separately encode edges, operations, network parameters of the neural architectures, enabling a more flexible network architecture selection. We discard top-2 softmax selection for

all edges and opreations as used in DARTS by directly using the binarized mask output to decide the architecture topology, leading to a significantly improved architecture selection from the underlying supernets.

Chapter 3 – *Arch2Vec* includes my deep exploration on what is the key component in NAS. From my previous work in Chapter 2, I found that one-shot NAS jointly optimizes the architecture representations (*e.g.* supernet topology) and the search methods (*e.g.* gradient decent). This could easily lead to local minimum because these two components are deeply coupled with each other: A bad optimization on architecture representations could incur bad search bias to the search strategy, and vice versa. Despite the widespread use, architecture representations learned in NAS are still poorly understood. From this perspective, I was thinking if it is possible to learn architecture representations first and then perform the downstream search. The intuition behind it is that a good representation should give us a reasonable performance even it is conducted using local search. Therefore, I propose to learn the architecture representations using edit distance closeness as its objectiveness, and the goal is to reconstruct the architecture themselves to preserve such local structure closeness. I found that compared to supervised joint optimization, unsupervised pretraining architecture representations followed by supervised fine-tuning is able to better encourage neural architectures with similar connections and operators to cluster together. This helps to map neural architectures with similar performance to the same regions in the latent space and makes the transition of architectures in the latent space relatively smooth, which significantly benefits diverse downstream search strategies. To this end, I confirm that architecture representations (encodings) are an important design decision in NAS, and my follow-up work in Chapter 4 aims to learn a better architecture representations by embracing more properties of neural architectures.

Chapter 4 – *Computation-aware neural architecture encoding* can be viewed as my continuing effort to design a better architecture representation in NAS which was introduced in Chapter 3. While most of the NAS encoders encode neural architectures using structural similarity, there is only a few work focusing a more discriminative property of neural architectures, *e.g.* computations, functions. Sometimes, the same computation can also define different functions, e.g., two identical neural architectures will represent different functions given they are trained differently since the weights of their layers will be different. Therefore, modeling functions is much harder than modeling computations since it additionally requires knowing the parameters of some operations, which are unknown before training [53]. Compared to structure-aware encodings, computation-aware encodings are able to map architectures with different structures but similar accuracies to the same region. This advantage contributes to a smooth encoding space with respect to the actual architecture performance instead of structures, which improves the efficiency of the downstream architecture search. I use Transformer as the encoder because it has been widely used as an effective and scalable generative model on sequence modeling tasks. Architectures can also be viewed as the a sequence given the specific attention module. I propose to encode the computation information of neural architectures through a pairwise learning scheme trained with MLM based on Transformers. I show its effectiveness and scalability in both small and large search spaces as well as its superior generalization capability outside the training search space.

Chapter 5 – *NAS-Bench-x11 and the Power of Learning Curves* Once the good architecture representations are obtained, they are used for the downstream search methods. The search process is conducted by fully training the sampled architectures from scratch until convergence just to get a single reward signal. Despite its stability compared to weight-sharing based methods, the computation budget is still non-negligible. Currently, two of

the most popular NAS benchmarks do not provide the full training information for each architecture. As a result, on these benchmarks it is not possible to run many types of multi-fidelity techniques, such as learning curve extrapolation,that require evaluating architectures at arbitrary epochs. This unavoidably slows down the development of multi-fidelity NAS algorithms. Therefore, we present a method using singular value decomposition and noise modeling to create surrogate benchmarks, NAS-Bench-111, NAS-Bench-311, and NAS-Bench-NLP11, that output the full training information for each architecture, rather than just the final validation accuracy. Based on this benchmark, we also introduce a learning curve extrapolation framework to modify single-fidelity algorithms, showing that it leads to improvements over popular single-fidelity algorithms given the same wall-clock budget.

Our hope is that our work will make it quicker and easier for researchers to run fair experiments and give reproducible conclusions. In particular, the surrogate benchmarks allow AutoML researchers to develop NAS algorithms directly on a CPU, as opposed to using a GPU, which may decrease the carbon emissions from GPU-based NAS research [155, 176]. In terms of our proposed NAS speedups, these techniques are a level of abstraction away from real applications, but they can indirectly impact the broader society. For example, this work may facilitate the creation of new high-performing NAS techniques, which can then be used to improve various deep learning methods, both beneficial (e.g. algorithms that reduce CO_2 emissions), or harmful (e.g. algorithms that produce deep fakes).

Lastly, I am fortunate to have gone through an exciting journey in developing Neural Architecture Search since my early days in PhD. I hope that this dissertation will provide useful background and inspirations for future research to build much more informative architecture encodings, powerful learning curve extrapolation algorithms, and their applications.

BIBLIOGRAPHY

[1] S. Yan, B. Fang, F. Zhang, Y. Zheng, X. Zeng, M. Zhang, and H. Xu, "Hm-nas: Efficient neural architecture search via hierarchical masking," in *ICCV Neural Architects Workshop*, 2019. ii, 5, 6, 7, 56

[2] S. Yan, Y. Zheng, W. Ao, X. Zeng, and M. Zhang, "Does unsupervised architecture representation learning help neural architecture search?" in *NeurIPS*, 2020. ii, 5, 6, 7, 8, 9, 53, 56, 57, 65, 67, 68, 69, 70, 71, 78

[3] S. Yan, K. Song, F. Liu, and M. Zhang, "Cate: Computation-aware neural architecture encoding with transformers," in *ICML*, 2021. ii, 5, 6, 8, 9, 75

[4] S. Yan, C. White, Y. Savani, and F. Hutter, "Nas-bench-x11 and the power of learning curves," in *NeurIPS*, 2021. ii, 6, 9

[5] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," in *JMLR*, 2019. 1, 5, 29, 33, 56, 75, 78

[6] C. White, W. Neiswanger, S. Nolen, and Y. Savani, "A study on encodings for neural architecture search," in *NeurIPS*, 2020. 4, 7, 8, 33, 53, 56, 62, 81

[7] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017. 1, 33, 56, 75, 78

[8] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," in *ICLR*, 2019. 1, 10, 13, 14, 100

[9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018. 1, 24

[10] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," in *ICML*, 2019. 2, 4, 30, 33, 37, 42, 43, 53, 56, 57, 58, 63, 65, 75, 79, 81, 86

[11] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," in *ICLR*, 2020. 2, 30, 37, 75, 79, 91

[12] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *ICLR*, 2019. 2, 3, 5, 10, 13, 14, 15, 17, 19, 21, 22, 24, 29, 30, 37, 38, 45, 46, 47, 56, 57, 64, 68, 70, 71, 75, 78, 80, 87, 100

[13] B. Zoph and K. Knight, "Multi-source neural translation," in *NAACL*, 2016. 3

[14] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, pp. 229–256, 1992. 4

[15] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms." in *ICGA*, 1989. 4, 56, 78

[16] H. Kitano, "Designing neural networks using genetic algorithms with graph generation system," in *Complex systems*, 1990. 4, 56, 78

[17] K. O. Stanley and R. A. Miikkulainen, "Evolving neural networks through augmenting topologies," in *Evolutionary Computation*, 2002. 4, 56

[18] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI*, 2019. 4, 10, 12, 14, 15, 22, 24, 33, 42, 44, 46, 47, 53, 56, 62, 67, 75, 77, 78, 87, 88, 90

[19] Z. Lu, K. Deb, E. Goodman, W. Banzhaf, and V. N. Boddeti, "Nsganetv2: Evolutionary multi-objective surrogate-assisted neural architecture search," in *ECCV*, 2020. 4, 56

[20] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *arXiv preprint arXiv:1711.00436*, 2017. 4, 53

[21] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing, "Neural architecture search with bayesian optimisation and optimal transport," in *NeurIPS*, 2018. 5, 33, 36, 53, 56, 78

[22] C. White, W. Neiswanger, and Y. Savani, "Bananas: Bayesian optimization with neural architectures for neural architecture search," in *AAAI*, 2021. 5

[23] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *IJCAI*, 2015. 5, 76, 78, 89

[24] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction," in *ICLR Workshop*, 2018. 5, 33, 53, 56, 76, 78, 89

[25] B. Ru, C. Lyle, L. Schut, M. van der Wilk, and Y. Gal, "Revisiting the train loss: an efficient performance estimator for neural architecture search," *arXiv preprint arXiv:2006.04492*, 2020. 5, 78

[26] A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter, "Learning curve prediction with bayesian neural networks," in *ICLR*, 2017. 5, 76, 78

[27] A. Chandrashekaran and I. R. Lane, "Speeding up hyper-parameter optimization by extrapolation of learning curves using previous builds," in *ECML-PKDD*, 2017. 5, 78

[28] K. Swersky, J. Snoek, and R. P. Adams, "Freeze-thaw bayesian optimization," *arXiv preprint arXiv:1406.3896*, 2014. 5, 76, 78

[29] M. Gargiani, A. Klein, S. Falkner, and F. Hutter, "Probabilistic rollouts for learning curve extrapolation across hyperparameter settings," *arXiv preprint arXiv:1910.04522*, 2019. 5, 78

[30] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," in *JMLR*, 2018. 5, 57, 76, 77, 79, 90

[31] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," in *ICML*, 2018. 5, 29, 33, 42, 44, 57, 76, 77, 78, 79, 90

[32] A. Klein, L. Tiao, T. Lienart, C. Archambeau, and M. Seeger, "Model-based asynchronous hyperparameter and neural architecture search," *arXiv preprint arXiv:2003.10865*, 2020. 5, 79

[33] Y. Huang, Y. Li, H. Ye, Z. Li, and Z. Zhang, "An asymptotically optimal multi-armed bandit algorithm and hyperparameter optimization," *arXiv preprint arXiv:2007.05670*, 2020. 5, 79

[34] N. Mallik and N. Awad, "Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization," in *IJCAI*, 2021. 5, 79

[35] K. Kandasamy, G. Dasarathy, J. B. Oliva, J. Schneider, and B. Poczos, "Multi-fidelity gaussian process bandit optimisation," in *NeurIPS*, 2016. 5, 76, 79

[36] K. Kandasamy, G. Dasarathy, J. Schneider, and B. Poczos, "Multi-fidelity bayesian optimisation with continuous approximations," in *JMLR*, 2017. 5, 76, 79

[37] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets," in *AISTATS*, 2017. 5, 79

[38] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," in *ICML*, 2018. 5, 11, 13, 56, 78, 100

[39] X. Dong and Y. Yang, "Searching for a robust neural architecture in four gpu hours," in *CVPR*, 2019. 5, 56, 78, 87

[40] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter, "Understanding and robustifying differentiable architecture search," in *ICLR*, 2020. 5, 29, 56, 78

[41] S. You, T. Huang, M. Yang, F. Wang, C. Qian, and C. Zhang, "Greedynas: Towards fast one-shot nas with greedy supernet," in *CVPR*, 2020. 5, 56, 78

[42] H. Peng, H. Du, H. Yu, Q. Li, J. Liao, and J. Fu, "Cream of the crop: Distilling prioritized paths for one-shot neural architecture search," in *NeurIPS*, 2020. 5, 56, 78

[43] K. Yu, R. Ranftl, and M. Salzmann, "Landmark regularization: Ranking guided super-net training in neural architecture search," in *CVPR*, 2021. 5, 78

[44] W. Wen, H. Liu, H. Li, Y. Chen, G. Bender, and P.-J. Kindermans, "Neural predictor for neural architecture search," in *ECCV*, 2020. 5, 33, 53, 56, 57, 62, 78

[45] X. Ning, W. Li, Z. Zhou, T. Zhao, Y. Zheng, S. Liang, H. Yang, and Y. Wang, "A surgery of the neural architecture evaluators," *arXiv preprint arXiv:2008.03064*, 2020. 5, 56, 78

[46] H. Shi, R. Pi, H. Xu, Z. Li, J. Kwok, and T. Zhang, "Bridging the gap between sample-based and one-shot neural architecture search with bonas," in *NeurIPS*, 2020. 5, 33, 36, 53, 56, 57, 62, 67, 75, 78

[47] R. Luo, X. Tan, R. Wang, T. Qin, E. Chen, and T.-Y. Liu, "Semi-supervised neural architecture search," in *NeurIPS*, 2020. 5, 56, 78

[48] C. White, A. Zela, B. Ru, Y. Liu, and F. Hutter, "How powerful are performance predictors in neural architecture search?" *arXiv preprint arXiv:2104.01177*, 2021. 5, 78, 91

[49] B. Ru, X. Wan, X. Dong, and M. Osborne, "Neural architecture search using bayesian optimisation with weisfeiler-lehman kernel," in *ICLR*, 2021. 5, 56, 75, 78

[50] C. White, W. Neiswanger, and Y. Savani, "Bananas: Bayesian optimization with neural architectures for neural architecture search," in *AAAI*, 2021. 5, 33, 36, 43, 44, 45, 53, 56, 57, 58, 62, 65, 67, 68, 69, 70, 75, 77, 78, 87, 88, 90

[51] V. Nguyen, S. Schulze, and M. Osborne, "Bayesian optimization for iterative learning," in *NeurIPS*, 2020. 5, 78

[52] L. Xie, X. Chen, K. Bi, L. Wei, Y. Xu, Z. Chen, L. Wang, A. Xiao, J. Chang, X. Zhang *et al.*, "Weight-sharing neural architecture search: A battle to shrink the optimization gap," *arXiv preprint arXiv:2008.01475*, 2020. 5, 56, 78

[53] M. Zhang, S. Jiang, Z. Cui, R. Garnett, and Y. Chen, "D-vae: A variational autoencoder for directed acyclic graphs," in *NeurIPS*, 2019. 8, 32, 38, 45, 53, 57, 58, 65, 102

[54] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018. 10, 12, 13, 14, 22, 24, 46, 47, 53, 56

[55] Y. Chen, T. Yang, X. Zhang, G. Meng, C. Pan, and J. Sun, "Detnas: Neural architecture search on object detection," *arXiv preprint arXiv:1903.10979*, 2019. 10

[56] V. Nekrasov, H. Chen, C. Shen, and I. Reid, "Fast neural architecture search of compact semantic segmentation models via auxiliary cells," in *CVPR*, 2019, pp. 9126–9135. 10

[57] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," in *ICML*, 2018. 10, 12, 14, 22, 45, 56, 78, 100

[58] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *ICLR*, 2017. 10, 12

[59] S. Xie, H. Zheng, C. Liu, and L. Lin, "Snas: Stochastic neural architecture search," in *ICLR*, 2019. 10, 13, 14, 15, 22, 24, 29, 45, 47, 56, 71

[60] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Smash: One-shot model architecture search through hypernetworks," *ICLR*, 2018. 10

[61] C. Sciuto, K. Yu, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the search phase of neural architecture search," in *ICLR*, 2020. 11, 13, 25, 75

[62] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," in *arXiv:1904.00420*, 2019. 11, 13, 29

[63] R. Luo, F. Tian, T. Qin, E.-H. Chen, and T.-Y. Liu, "Neural architecture optimization," in *NeurIPS*, 2018. 14, 29, 33, 43, 53, 56, 57

[64] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742. 14, 17, 21, 23

[65] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131. 18, 19

[66] A. Mallya, D. Davis, and S. Lazebnik, "Piggyback: Adapting a single network to multiple tasks by learning to mask weights," in *ECCV*, 2018, pp. 67–82. 19

[67] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in neural information processing systems*, 2016, pp. 4107–4115. 19

[68] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ICLR*, 2015. 21, 36, 38

[69] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," in *arXiv:1708.04552*, 2017. 21

[70] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017, pp. 4700–4708. 22

[71] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009. 23, 46, 69

[72] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017. 23, 24

[73] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NeurIPS*, 2013. 29

[74] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *ACL*, 2019. 29, 55, 58

[75] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," in *OpenAI Blog*, 2018. 29

[76] A. v. d. Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu, "Conditional image generation with pixelcnn decoders," in *NeurIPS*, 2016. 29

[77] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *CVPR*, 2020. 29

[78] C. Finn, I. Goodfellow, and S. Levine, "Unsupervised learning for physical interaction through video prediction," in *NeurIPS*, 2016. 29

[79] E. Jang, C. Devin, V. Vanhoucke, and S. Levine, "Grasp2vec: Learning object representations from self-supervised grasping," in *arXiv:1811.06964*, 2018. 29

[80] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *ACM SIGKDD*, 2014. 29, 31

[81] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *ACM SIGKDD*, 2016. 29, 31

[82] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," in *Machine Learning*, 1992. 29, 35, 42, 44

[83] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *ICML*, 2017. 29, 57

[84] C. He, H. Ye, L. Shen, and T. Zhang, "Milenas: Efficient neural architecture search via mixed-level reformulation," in *CVPR*, 2020. 29

[85] Y. Shu, W. Wang, and S. Cai, "Understanding architectures learnt by cell-based neural architecture search," in *ICLR*, 2020. 29

[86] A. Zela, J. Siems, and F. Hutter, "Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search," in *ICLR*, 2020. 29, 79

[87] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," in *JMLR*, 2020. 30, 75

[88] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding." in *WWW*, 2015. 31

[89] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *KDD*, 2016. 31

[90] T. N. Kipf and M. Welling, "Variational graph auto-encoders," in *NeurIPS Workshop*, 2016. 32, 34, 38, 39

[91] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, 2017. 32

[92] P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjelm, "Deep Graph Infomax," in *ICLR*, 2019. 32

[93] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *ICLR*, 2019. 32, 34, 65

[94] B. Weisfeiler and A. Lehman, "A reduction of a graph to a canonical form and an algebra arising during this reduction." in *Nauchno-Technicheskaya Informatsia*, 1968. 32, 33

[95] J. You, J. Leskovec, K. He, and S. Xie, "Graph structure of neural networks," in *ICML*, 2020. 32, 55, 58

[96] Y. LeCun, S. Chopra, R. Hadsell, and F. J. Huang, "A tutorial on energy-based learning," in *Predicting Structured Data*, 2006. 32

[97] K. Kavukcuoglu, P. Sermanet, Y. lan Boureau, K. Gregor, M. Mathieu, and Y. L. Cun, "Learning convolutional feature hierarchies for visual recognition," in *NeurIPS*, 2010. 32

[98] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *ICML*, 2008. 32

[99] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *ICLR*, 2014. 32, 35, 67, 83

[100] A. Makhzani, J. Shlens, N. Jaitly, and I. Goodfellow, "Adversarial autoencoders," in *ICLR*, 2016. 32

[101] P. Ghosh, M. S. M. Sajjadi, A. Vergari, M. Black, and B. Scholkopf, "From variational to deterministic autoencoders," in *ICLR*, 2020. 32

[102] M. Simonovsky and N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders," in *arXiv:1802.03480*, 2018. 32

[103] W. Wang, H. Xu, Z. Gan, B. Li, G. Wang, L. Chen, Q. Yang, W. Wang, and L. Carin, "Graph-driven generative models for heterogeneous multi-task learning," in *AAAI*, 2020. 32

[104] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," in *arXiv:2006.02903*, 2020. 33

[105] C. Liu, P. Dollár, K. He, R. Girshick, A. Yuille, and S. Xie, "Are labels necessary for neural architecture search?" in *arXiv:2003.12056*, 2020. 33

[106] S. Kullback and R. A. Leibler, "On information and sufficiency," in *Annals of Mathematical Statistics*, 1951. 35

[107] J. Sietsma and R. J. Dow, "Creating artificial neural networks that generalize," in *Neural Networks*, 1991. 35

[108] G. An, "The effects of adding noise during backpropagation training on a generalization performance," in *Neural Computation*, 1996. 35

[109] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," in *arXiv:1707.06347*, 2017. 36

[110] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams, "Scalable bayesian optimization using deep neural networks," in *ICML*, 2015. 36, 62, 67

[111] R. Garnett, M. A. Osborne, and P. Hennig, "Active learning of linear embeddings for gaussian processes," in *UAI*, 2014. 36

[112] J. Mockus, "On bayesian methods for seeking the extremum and their application." in *IFIP Congress*, 1977. 36

[113] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *ECCV*, 2018. 38, 53, 56, 64

[114] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," in *JMLR*, 2008. 40

[115] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," in *JMLR*, 2012. 42, 44

[116] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *UAI*, 2019. 45, 47, 56, 62, 67, 68, 70, 75, 90

[117] I. Radosavovic, J. Johnson, S. Xie, W.-Y. Lo, and P. Dollár, "On network design spaces for visual recognition," in *ICCV*, 2019. 46

[118] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," in *CVPR*, 2020. 53

[119] B. Ru, P. Esperanca, and F. Carlucci, "Neural architecture generator optimization," in *NeurIPS*, 2020. 53

[120] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca, "Alphax: exploring neural architectures with deep neural networks and monte carlo tree search," in *AAAI*, 2020. 53, 56, 57

[121] C. White, S. Nolen, and Y. Savani, "Local search is state of the art for nas benchmarks," in *UAI*, 2021. 53, 56, 62, 67, 77, 88, 90

[122] X. Ning, Y. Zheng, T. Zhao, Y. Wang, and H. Yang, "A generic graph-based neural architecture encoding scheme for predictor-based nas," in *ECCV*, 2020. 53, 57

[123] Y. Zhang, J. Zhang, and Z. Zhong, "Autobss: An efficient algorithm for block stacking style search," in *NeurIPS*, 2020. 53

[124] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017. 55, 58

[125] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," in *JMLR*, 2020. 55, 58

[126] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *ICCV*, 2019. 55, 58

[127] R. W. Floyd, "Algorithm 97: Shortest path," in *Communications of the ACM*, 1962. 55

[128] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," in *arXiv:2008.09777*, 2020. 56, 57, 64, 65

[129] T. D. Ottelander, A. Dushatskiy, M. Virgolin, and P. A. Bosman, "Local search is a remarkably strong baseline for neural architecture search," in *International Conference on Evolutionary Multi-Criterion Optimization*, 2021. 56, 62, 88, 90

[130] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *CVPR*, 2019. 56

[131] H. Zhou, M. Yang, J. Wang, and W. Pan, "BayesNAS: A Bayesian approach for neural architecture search," in *ICML*, 2019. 56, 71

[132] R. Negrinho and G. Gordon, "Deeparchitect: Automatically designing and training deep architectures," *arXiv preprint arXiv:1704.08792*, 2017. 56, 78

[133] Y. Tang, Y. Wang, Y. Xu, H. Chen, B. Shi, C. Xu, C. Xu, Q. Tian, and C. Xu, "A semi-supervised assessor of neural architectures," in *CVPR*, June 2020. 56

[134] X. Chen and C.-J. Hsieh, "Stabilizing differentiable architecture search via perturbation-based regularization," in *ICML*, 2020. 56

[135] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, J. Marben, P. Müller, and F. Hutter, "Boah: A tool suite for multi-fidelity bayesian optimization and analysis of hyperparameters," in *arXiv preprint arXiv: 1908.06756*, 2019. 57

[136] C. Wei, Y. Tang, C. Niu, H. Hu, Y. Wang, and J. Liang, "Self-supervised representation learning for evolutionary neural architecture search," in *arXiv preprint arXiv: 2011.00186*, 2020. 57

[137] K. Choi, M. Choe, and H. Lee, "Pretraining neural architecture search controllers with locality-based self-supervised learning," in *arXiv preprint arXiv: 2103.08157*, 2021. 57

[138] C. Wei, C. Niu, Y. Tang, and J. Liang, "Npenas: Neural predictor guided evolution for neural architecture search," *arXiv preprint arXiv:2003.12857*, 2020. 57, 75

[139] J. Lukasik, D. Friede, A. Zela, F. Hutter, and M. Keuper, "Smooth variational graph embeddings for efficient neural architecture search," in *IJCNN*, 2021. 57

[140] D. Hesslow and I. Poli, "Contrastive embeddings for neural architectures," in *arXiv preprint arXiv: 2102.04208*, 2021. 58

[141] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H.-W. Hon, "Unified language model pre-training for natural language understanding and generation," in *NeurIPS*, 2019. 58

[142] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *NAACL*, 2018. 58

[143] S. Hochreiter and J. Schmidhuber, "Long short-term memory," in *Neural computation*, 1997. 58

[144] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," in *OpenAI Blog*, 2019. 58

[145] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," in *NeurIPS*, 2019. 58

[146] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," in *arXiv:1907.11692*, 2019. 58

[147] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *ACL*, 2020. 58

[148] C. E. Rasmussen and C. K. I. Williams, "Gaussian processes for machine learning," in *The MIT Press*, 2006. 62

[149] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *SIGKDD*, 2016. 65

[150] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *NeurIPS*, 2017. 65, 84

[151] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *ICLR*, 2019. 65

[152] J. T. Springenberg, A. Klein, S. Falkner, and F. Hutter, "Bayesian optimization with robust bayesian neural networks," in *NeurIPS*, 2016. 67

[153] G. Bender, H. Liu, B. Chen, G. Chu, S. Cheng, P.-J. Kindermans, and Q. Le, "Can weight sharing outperform random architecture search? an investigation with tunas," in *CVPR*, 2020. 74

[154] A. Yang, P. M. Esperança, and F. M. Carlucci, "Nas evaluation is frustratingly hard," in *ICLR*, 2020. 75

[155] K. Hao, "Training a single ai model can emit as much carbon as five cars in their lifetimes," *MIT Technology Review*, 2019. 75, 103

[156] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," *arXiv preprint arXiv:2008.09777*, 2020. 75, 80, 85, 86, 87, 95, 97

[157] N. Klyuchnikov, I. Trofimov, E. Artemova, M. Salnikov, M. Fedorov, and E. Burnaev, "Nas-bench-nlp: neural architecture search benchmark for natural language processing," *arXiv preprint arXiv:2006.07116*, 2020. 76, 80, 88

[158] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002. 78

[159] H. Hu, J. Langford, R. Caruana, S. Mukherjee, E. Horvitz, and D. Dey, "Efficient forward architecture search," in *NeurIPS*, 2019. 78

[160] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009. 79

[161] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "A downsampled variant of imagenet as an alternative to the cifar datasets," in *arXiv:1707.08819*, 2017. 79

[162] X. Dong, L. Liu, K. Musial, and B. Gabrys, "Nats-bench: Benchmarking nas algorithms for architecture topology and size," in *PAMI*, 2021. 79

[163] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search: Openreview response," 2021. [Online]. Available: https://openreview.net/forum?id=1flmvXGGJaa 80

[164] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Annual conference of the international speech communication association*, 2010. 80

[165] A. Mehrotra, A. G. C. P. Ramos, S. Bhattacharya, Ł. Dudziak, R. Vipperla, T. Chau, M. S. Abdelfattah, S. Ishtiaq, and N. D. Lane, "Nas-bench-asr: Reproducible neural architecture search for speech recognition," in *ICLR*, 2021. 80

[166] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, vol. 2, no. 2, pp. 205–224, 1965. 83

[167] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794. 84

[168] D. W. Scott, *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015. 85, 93

[169] S. Wright, "Correlation and causation," *Journal of Agricultural Research*, vol. 20, pp. 557–580, 1921. 85

[170] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938. 85

[171] K. Eggensperger, F. Hutter, H. Hoos, and K. Leyton-Brown, "Efficient benchmarking of hyperparameter optimizers via surrogates," in *AAAI*, 2015. 86

[172] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *NeurIPS*, 2011. 87

[173] C. Oh, J. M. Tomczak, E. Gavves, and M. Welling, "Combinatorial bayesian optimization using the graph cartesian product," *arXiv preprint arXiv:1902.00448*, 2019. 87

[174] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong, "Pc-darts: Partial channel connections for memory-efficient architecture search," in *ICLR*, 2019. 87

[175] X. Chen, R. Wang, M. Cheng, X. Tang, and C.-J. Hsieh, "Drnas: Dirichlet neural architecture search," in *ICLR*, 2021. 87

[176] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon emissions and large neural network training," *arXiv preprint arXiv:2104.10350*, 2021. 103