Abstract of “Building an Intelligent Agent to Design Neural Networks” by Linnan Wang, Ph.D., Brown University, Feb 2022.

Designing Artificial Intelligence (AI) is still reserved for experts, and the existing design paradigm follows a data-driven approach: domain experts start with a hypothetical model, verify the model on a task-specific dataset to acquire performance metrics, then revise the model based on prior experiences, hoping to improve the model in the next loop. This thesis seeks to build an intelligent agent to substitute domain experts in this design process. I start with formalizing the current design process as a computational model, upon which I further investigate issues to the algorithmic efficiency and system utilization to build an agent that algorithm and system can synergistically work together. Specifically, I propose a new black box solver, Latent Action Monte Carlo Tree Search (LA-MCTS), to address the sample efficiency and build a deep learning framework to expand the design space far beyond the available GPU DRAM. These results collectively provide a partial path toward AI democratization by creating a practical MCTS-based AI agent that efficiently designs complex AI without experts in a reasonable amount of time.

Building an Intelligent Agent to Design Neural Networks

by

Linnan Wang

B. E., University of Electronic Science and Technology of China, 2011

M. Sc., Rutgers University, 2013

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

February 2022

© Copyright 2022 by Linnan Wang

This dissertation by Linnan Wang is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date Sep 2, 2021	[illegible]
	Rodrigo Fonseca, Director

Recommended to the Graduate Council

Date **Aug. 2, 2021**	[illegible]
	Yuandong Tian, Reader Facebook
Date 01/08/2021	[illegible]
	George Konidaris, Reader Brown University
Date Aug 2 2021	[illegible]
	Stephen Bach, Reader Brown University

Approved by the Graduate Council

Date	
	Andrew G. Campbell Dean of the Graduate School

Acknowledgements

I came to U.S. to study Ph.D. in 2011, and it toke me 10 years to finally present this thesis to you. This would not be possible without the help of so many brilliant people across the world. I'm extremely grateful to my mentors for their support and guidance in this journey, my collaborators for their help and splendid ideas, my friends for their encouragement, and my family for their unconditioned love and support.

First, I want to thank my advisor, Rodrigo Fonesca. The biggest thing I learned from Rodrigo is to pursue your interests. Rodrigo works in the distributed system, but he always supports and encourages me to explore artificial intelligence. I'm incredibly grateful to have the research freedom to work on the topic of this thesis that genuinely excites me in the last four years. On top of this, Rodrigo is both greatly wise, respectful, and humble. I can recall many occasions in my early Ph.D., where I claim to build a deep learning system to replace TensorFlow and PyTorch, and I know how foolish it sounds now. But he never laughed at me, encouraged and guided me to look into the specific problems in these systems. Some of my absolute memorable experiences at Brown are these moments Rodrigo and I spent hours in his office to think through a tricky problem on the whiteboard. These conversations eventually turned into several exciting ideas and publications that constitute this thesis. I want to sincerely thank Rodrigo for his support for giving me the freedom to explore, being respectful to my reckless ideas, guiding me in the right direction, and being such an outstanding mentor and advisor. I am so lucky to have had the opportunity to work with you!

To my mentor Yuandong Tian, I want to thank him for showing me that there is always someone wiser than me. I feel very fortunate to intern with Yuandong at Facebook AI Research, where I had a chance to learn from some of the most brilliant minds in AI today. Yuandong is immeasurably wise. Whenever I was stuck on a problem, he always quickly summarized the intrinsic nature of the problem and then led me in the right direction. This thesis also greatly benefited from his insight that the definition of actions in MCTS is critical to efficiency. This insight eventually grew into a new search method for black-box optimizations, produced several publications in path planning, multi-objective optimizations, and neural architecture search with numerous applications that constitute most of this thesis. Our time doing research together is one of the most memorable times during my study, and I look forward to continuing our collaborations in the future.

Many of my works wouldn't have been possible without the help from my collaborators. I have been working closely with Yiyang Zhao in the last five years, and we have been through ups and downs together. Yiyang is an accountable research buddy and a true friend to me. Doing a Ph.D. is hard, but having a close friend like Yiyang makes it a lot easier. From the bottom of my heart, I truly appreciate our time together. I look forward to our continued collaboration on conquering the most challenging problems together. Next, I'm immensely grateful to Yanyi Guo for being with me from 2013 to 2018. Although our path diverted, I cherish our memory and sincerely wish you the best of luck in your future. I also want to thank Yuu Jinnai, John Meehan, Jeff Rasley, George Boscia, Hang Liu, Jinmian Ye, Zenglin Xu, Jianxiong Xiao, Junyu Zhang, and Shuohang Wang for their help, trust, and time in doing research together with me. Finally, I want to thank George Konidaris and Maurice Herlihy for providing valuable guidance and thank Tim Kraska and Carsten Binnig for recruiting me to Brown.

Here I want to thank my two other friends, Alex Zhang and Fuguo Jiang (in Memoriam). Alex is such a nice guy that helped me a lot in English writing; Fuguo taught me to be a man of integrity, one of the most important lessons in my life. I also want to thank the staff and faculties at Brown CS. Lauren is very resourceful to the department rules and requirements, and I also learned a lot from our program director David Laidlaw. I want to thank Saining Xie, Teng Li, Yuxiong He, Samyam Rajbhandari, and Yi Yang for hosting my internships at Facebook, Microsoft, and NEC Labs.

Finally, I want to thank my family. To my father, Xinli Wang, I want to thank him for his support. He bought my first laptop in 2004, which nearly cost him a whole year's salary. I love playing computers since I was a kid, and that laptop is the best gift I ever had; and it later motivates me to pursue a career in computer science. Before the university, I was never a good student. I want to thank my mother, Qiuxiang Liu, and stepfather Mingfang Wu for their encouragement and unconditioned love. To my wife, Ying Wang, I'm forever grateful for your love and support, and the sacrifice you made to our family. Marrying you is the best decision I ever made.

To Michelle Wang, your mother and I are so blessed to have you as our daughter! Your smiles always cure me, and I wish you good health and joy throughout your life.

Contents

List of Tables

List of Figures

Part I

Preliminaries

Chapter 1

Introduction

The resurgence of connectionism since early 2010 marks a remarkable new chapter for Artificial Intelligence (AI). The boom of General-Purpose computing on Graphics Processing Units (GPGPU) and Big Data all serve as a catalyst to fuel AI's exponential growth consistently. At this wave of AI revolution, Deep Neural Networks (DNNs) are no doubt the backbone technology behind the scene, manifested by the countless successful applications in many areas, including the superhuman AI for multiplayer poker [Brown and Sandholm, 2019], the self-taught AI for Go [Silver et al., 2017, Tian et al., 2019a], the near-human performance of self-driving cars [DMV, 2020] at Cruise Automation and Waymo, the Grandmaster level II AI for Starcraft II [Vinyals et al., 2019], the super-radiologists AI in the breast cancer screening [McKinney et al., 2020], and real-time super-translator AI for speech to speech translation [Jia et al., 2019]. Besides, the prevailing trend of DNN based unsupervised learning in both Natural Language Processing (NLP) [Devlin et al., 2018] and Computer Vision (CV) [He et al., 2019b] are also achieving presumably impossible things. These examples strongly demonstrate the dominance of DNNs in various sub-fields of AI today for solving problems that have resisted the best attempts of the AI community for many years [LeCun et al., 2015].

In the era of Big Data and Big GPU, the success of DNNs is by no means a coincidence. For decades, the conventional Machine Learning techniques, e.g., Support Vector Machine (SVM) [Cortes and Vapnik, 1995], were majorly limited by the proper craft of feature vectors to the raw data, which requires considerable and painstakingly engineering from domain experts [LeCun et al., 2015]. This conventional approach is usually inefficient in practice, especially at internet giants such as Facebook and Google, due to the massive amount of user images or texts to be handled. Whereas DNNs successfully alleviate the burden of feature engineering at the cost of extra computations, that is no longer a problem with the emergence of GPGPU since

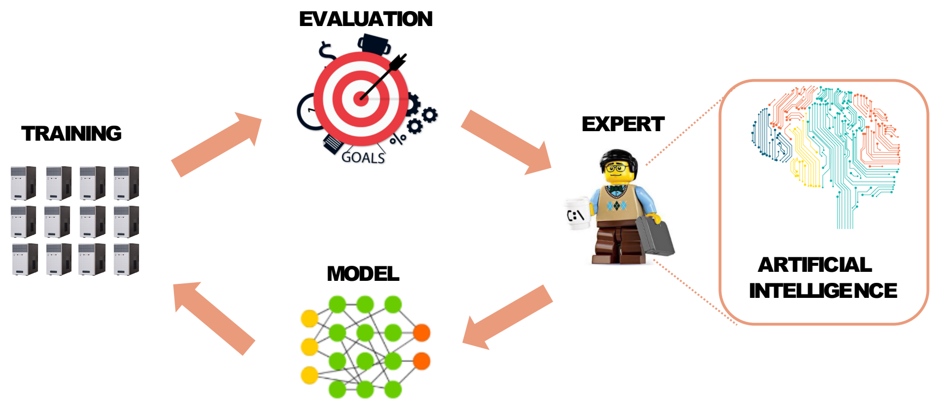


Figure 1.1: This thesis intends to build an Artificial Intelligence that replaces domain experts in the data-driven design process of deep neural networks.

2006. Specifically, DNNs directly learn the intricate structures of any high-dimensional data such as images, videos, voices, and texts using several non-linear operators to sequentially transform the data into layers of abstract representations to be processed by other Machine Learning primitives. One classical success is applying DNNs at the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2011 [Krizhevsky et al., 2012], which significantly outperformed all the conventional feature-based methods and ignited the Deep Learning revolution afterward.

With the advent of big and powerful DNNs, machine learning practitioners have gradually shifted their focus to the structural engineering of neural architectures [Xie, 2018]. Though DNNs alleviate the burden of feature engineering, a deep, wide, and densely connected neural network is not a viable solution to all tasks. The structure of a DNN needs to be carefully tailored for a particular task before working well. Therefore, finding useful structural priors has been one of the principal goals in the development of DNNs. For example, the convolution operators in Convolution Neural Networks (CNNs) make CNN faster to run and far more accurate than Multi-layer Perceptions (MLPs) in computer vision tasks using the sparse connections among neurons. Another notable example is the residual connection in ResNet [He et al., 2016], which drastically improves the performance of conventional CNNs such as AlexNet. By accumulating helpful structure priors, machine learning practitioners have transformed their workflow to the structural assembly of these structural priors, then fine-tuning the model by their experience in an iterative manner.

To be specific, the existing paradigm of neural network design follows a data-driven trial and error. Domain experts start with a hypothetical model built from structure priors, verifying the model on some task-specific dataset to acquire performance metrics. Then, the experts revise the model according to their previous experiences to improve the model in the next iteration. Domain experts keep iterating this process, maybe a few hundred times, until reaching a model that satisfies the design requirements. This process can be pretty tricky for a human to handle, especially considering the prolonged design process typically lasting for months or even years, as the DNN training is rather challenging. For example, the complex design choices induce a high-dimensional model space with an immense amount of model possibilities, e.g., the design space of non-linear Convolution Neural Networks (CNN) can quickly go beyond 10^{30} possibilities [Zoph et al., 2018b]. Besides the accuracy, the requirements sometimes prefer considering multiple factors together, e.g., resource requirements, which further complicates the design to trade-off multi-objectives. Therefore, it is urgent to find a principled method to efficiently tackle these challenges that emerged from the recent progress, which is the core research question this thesis seeks to answer. The following statement articulates the central research question studied by this thesis,

> *How to formulate the DNN design into a computational model that is feasible on the existing computing hardware? Taking it one step further, can we build an intelligent agent to replace the domain experts in artificial intelligence design entirely?*

This research question also has a tremendous social impact. Countries like China or the United States have established their own AI initiatives to facilitate AI innovation as part of their national strategies. For example, U.S. executive order 13859 stresses the necessity of sustained advancements in AI and details five pillars for bolstering AI, including greater investment in AI R&D, the elimination of barriers to AI development, enhanced AI training of the workforce, and the encouragement of international standards that foster the success of American AI innovation. Seeking solutions to the proposed research question will improve targeting and utilization of investment capital in AI R&D, e.g., domain experts can devote more of their scarce time and monetary resources to innovation. It will also significantly lower the bar for general workers to use AI and drastically accelerate the AI workforce's training to meet the urgent demand from the market. Therefore, studying the proposed research question is a crucial step toward AI democratization.

1.1 Problem Formulation and Technical Challenges

At first glance, this research question seems to seek an Artificial General Intelligence (AGI), solving it could solve the rest of the questions in AI. Nevertheless, AGI is too broad for now, and we have to simplify the problem into a concise theoretical framework to make concrete progress. Therefore, we start with formalizing the data-driven DNN design before elaborating on the technical challenges from both the algorithm and system sides.

1.1.1 Problem Formulation

Let's first define the notations for formulating the problem. The layer-wise data processing nature of DNNs defines itself to be a data-flow graph, and all the existing types of DNNs, including CNN, Transformer, Generative Adversarial Network (GAN), or Recurrent Neural Network (RNN) confirm to this representation [Tch, 2017]. Representing a data-flow graph is a well-studied problem, and one simple solution is to use the adjacent matrix and node list [Cormen et al., 2009]. This allows us to reasonably assume that a vector $\mathbf{x}$ can fully specify the structure of any DNNs, by concatenating the flattened adjacent matrix and node list together. It is a general practice to place several constraints in designing a neural network, such as setting the maximal depth of a neural network. Here we use Ω to denote the feasible region for $\mathbf{x}$. A neural network $\mathbf{x}$ also carries a set of parameters θ for training, denoted as $\mathbf{x}(\theta)$. In evaluating a neural network $\mathbf{x}(\theta)$, we apply different metrics based on tasks. For example, top-1 accuracy is widely used for image recognition in CV, but f1 score is the key metric for the question-answering in NLP [Rajpurkar et al., 2016]. Here we denote the evaluation metric as ϕ. Please note the evaluation may also consider other factors such as the latency or memory usage besides the performance, so ϕ can be multi-objective. D_{train} and D_{val} represent the dataset for training and testing, respectively.

Now we're ready to formulate the data-driven design paradigm of DNNs as an optimization problem:

$$\begin{aligned} \mathbf{x}^* &= \text{argmax}_{\mathbf{x}} \phi(\mathbf{x}(\theta^*, D_{val})), \text{where } \mathbf{x} \in \Omega \\ \text{s.t. } \theta^* &= \text{argmax}_{\theta} \phi^{'}(\mathbf{x}(\theta, D_{train})) \end{aligned} \tag{1.1}$$

This optimization problem demonstrates two intimately connected steps. The first line represents the architecture search that maximizes the performance (ϕ) by varying the design of networks $\mathbf{x}$ using the test dataset D_{val}, given θ^* as the optimal parameters of $\mathbf{x}$. The final output is the best architecture performing on the test dataset. The second line outputs the optimal parameters θ^* for the architecture from the first step by training toward D_{train}, i.e., training the neural network. In this formulation, we will find footing to make the study of the proposed research question concrete.

1.1.2 Technical Challenges

The above optimization problem should be easy to solve with many existing algorithms, such as grid search or gradient descent if the design of DNN only involved a few parameters and the training of DNN was fast. However, many hurdles render the problem challenging to solve, requiring both algorithmic and system innovations to make it practical. We summarize these challenges as follows.

- *High-dimensional combinatorics*: the problem formulation seeks an optimal configuration of architecture vector $\mathbf{x}$ on the feasible region Ω. This is a traditional problem widely studied in combinatorics that finds the best structure or solution among several possibilities. The key challenge of the combinatorial problem is that the feasible region grows exponentially with the number of problem dimensions, also known as the curse of dimensionality [Bellman, 1966]. Configuring a practical neural network generally requires more than 20 parameters, making a sweep of all the possible parameters impossible. For example, NASNet is one of the most popular search spaces for image recognition. The widely adopted configuration of the NASNet search space has 10^{21} candidate architectures that require a vector of 28 discrete variables to fully specify [Wang et al., 2019a]. Therefore, sample efficiency is crucially important in solving the problem with such a vast search space.

- *Non-convex optimization*: our research problem is also non-convex with many local optimums in the solution space Ω. The construction of DNN utilizes many non-convex operators [Bengio et al., 2006]; therefore, both the meta-design of DNN and itself are non-convex functions, which is also empirically corroborated by NASBench-101 that contains over $4.2 * 10^5$ the architecture and final evaluation accuracy pairs [Ying et al., 2019]. Due to the multi-modal nature of the problem itself, gradient descent, hill climbing, or other greedy-based search methods can easily trap into a local optimum, yielding a sub-optimal solution. An efficient global algorithm is desired to explore a local optimum intelligently and exploit a promising region to find a good solution.

- *Expensive neural network training*: today's big neural network is trained with a massive amount of data, which dramatically increases the computation cost in training. For example, ResNet-50 takes 29 hours to finish 90-epoch ImageNet training on eight NVIDIA P100 GPUs, while 81 hours to complete BERT pre-training on 16 v3 Google TPUs [You, 2020]. These data suggest that our formulation of DNN design can quickly become intractable to the current hardware without a highly sample-efficient solver, such as using the grid search or exhaustive search. On the other hand, reducing the cost of neural network training for the second step of Eq. 1.1 will also increase the number of samples evaluated in a given time, i.e., the first step of Eq. 1.1. So, fully utilizing the heterogeneous distributed system to train neural networks is also very important.

- *Utilization of heterogeneous systems*: undoubtedly, solving Eq. 1.1 requires a tremendous amount of computations, urging us to harness the massive computing power from a large-scale distributed system to approach the problem. Whereas, the distributed system today has evolved to be highly heterogeneous: a workstation not only has one CPU, but also other co-processors such as NVIDIA GPU, Google TPU, or Intel Phi. These co-processors are inserted in PCI-E and are equipped with an independent DRAM, requiring explicit communications to move the data back to on-device DRAM if the data is located on CPU DRAM. Besides the heterogeneity of processors, the network bandwidth is also highly heterogeneous. For example, the bandwidth for inter-GPU communications via PCI-E 3.0 is 15.75 GB/s, while NVLink is up to astonishingly 50 GB/s. All of these heterogeneities greatly complicate the program to maximize the utilization of computing resources.

1.2 Contributions

With the main research question in play, I now summarize my contributions from years of study to build an intelligent agent that designs DNNs. I study the research question in Eq. 1.1 by drawing insights from the recent success of Monte Carlo Tree Search (MCTS) in playing Go [Tian et al., 2019a], and we tackle the sample-efficiency from a brand new perspective of learning the search space partition using MCTS. To sustain the computation needs from algorithms, we also developed novel software systems to support large DNNs and a new parallelization strategy to accelerate the distributed DNN training with a theoretical guarantee. We have developed an intelligent agent driven by a novel MCTS based solver named Latent Action Monte Carlo Tree Search (LA-MCTS). This agent has designed many SoTA models for computer vision tasks using up to 500 GPUs at Facebook, such as finding a neural network that reaches 99% top-1 accuracy on CIFAR-10. LA-MCTS is also an award-wining algorithm in the black-box optimization challenge at NeurIPS-2020 [Sazanovich et al., 2020, Kim et al., 2020b], which has over 68 global participating teams, including companies such as NVIDIA, Huawei, and IBM [BBContest, 2017]. Therefore, these contributions allow me to confidently make the following thesis statement:

Thesis Statement

With algorithmic improvements and new system designs, it is possible to build an intelligent agent to replace domain experts in designing neural networks for a variety of AI applications using the existing computing infrastructure.

1.2.1 Details of Contributions

Developing a practical agent to assist domain experts in designing neural networks requires new algorithms to tackle the challenge from computational complexity and new systems to embody the algorithms efficiently. In summary, this thesis has made the following contributions.

Alg1: Propose the principle of learning search space partition using MCTS to improve the sample efficiency of existing black-box solvers such as Bayesian Optimization.
Application: Training a neural network is very expensive, so the number of networks sampled by the design process is one of the major bottlenecks. This principle allows the agent to get a better result by training fewer networks.

Alg2: Propose the principle of using multiple super-nets to improve the performance estimation of neural networks in the design space.
Application: Super-net is an over-parameterized network used for predicting the performance of networks in a design space. It replaces network training with performance prediction, but the prediction is quite inaccurate. This principle uses multiple super-nets to enhance the predicted ranking correlation without compromising the speed.

Alg3: Propose the principle of Fourier transform-based gradient sparsification to accelerate distributed DNN training with a theoretical guarantee to recover the accuracy.
Application: Distributed DNN training can significantly accelerate network training, while communication is the major bottleneck in this process. This principle allows to drop up to 90% of communications to expedite distributed training while maintaining the same accuracy as the case without sparsification.

Sys1: Propose the principle of out-of-core GPU computations that pipeline computations and communications over multiple streams to eliminate the communication cost.

Application: We have seen a growing interest in designing very large DNN, especially in NLP, while the limited GPU DRAM places an undesired constraint. This principle allows the agent to design large DNNs with memory requirements far beyond the GPU DRAM without compromising the speed.

Sys2: Propose the principle of using GPU DRAM as a cache to alleviate communications in the pipelined stream computations on GPUs.

Application: In supporting the large-scale DNN on the GPU, the out-of-core computations consistently swap out operands to save the memory. This principle reduces unnecessary communication by fully exploiting the free space on GPU DRAM.

Each principle provides one piece to the jigsaw puzzle of building an intelligent agent to design DNNs. All of these principles together construct a concrete and coherent solution to the proposed research question. We have evaluated each principle on various specific yet essential domains, and some principles are also independently replicated and assessed by the community in public contests. These results consistently corroborate that these algorithm and system principles are very effective at improving the sample efficiency and sustaining the computation needs from algorithms. Based on these principles, we have developed a working agent capable of designing DNNs for various tasks to exceed SoTA results using hundreds of GPUs at Facebook. The rest of this thesis is organized as follows:

PART 1. This part focuses on improving sample efficiency. By drawing insights from the recent success of AlphaGo [Silver et al., 2016], we build the first agent using Monte Carlo Tree Search (MCTS) for a better exploration policy to jump out of local optima. MCTS trades off the exploration and exploitation based on the visiting statistics at individual states, while the static and coarse-grained ϵ-strategy simply treats all the states the same. In Chapter 2, I describe the development of our MCTS-based agent and elaborate the background of how we approach the research problem. I empirically show that the exploration is critical to help the agent jump out of a local optimum in the design space, thereby finding a better network than other agents that use greedy approaches. While we were developing the initial agent, we made the critical insight that the action space of MCTS is vital to the sample efficiency. This observation is empirically verified by a controlled experiment with the setup of two experiments being the same except for using two different action spaces for MCTS. This result motivates us to learn the action space for MCTS to best fit the performance metric to be optimized. Chapter 3 presents a new algorithm to fulfill this goal, named Latent Action Monte

Carlo Tree Search (LA-MCTS), and a scalable and fault resilient software architecture for parallelizing LA-MCTS. The latent actions in LA-MCTS are SVM boundaries learned from previously collected samples to partition the search space into good and bad regions. Solvers such as Bayesian Optimization can focus on the promising region rather than the entire search space. When LA-MCTS is paired with Bayesian optimization, it introduces many benefits, such as avoiding the over-exploring in high-dimensional problems and significantly reducing the complexity for the acquisition optimization. LA-MCTS is also independently replicated and verified in the black-box optimization challenge at NeurIPS-2020. Specifically, JetBrains research [2] and KAIST [3] independently replicated 'learning search space partition' in LA-MCTS and won 3rd and 8th places among 68 global participating teams, including companies and institutions such as NVIDIA, Huawei, Oxford, KAIST, IBM, Preferred Networks, and Innovatrics. Finally, the following publications contribute to this part of the thesis.

Wang, Linnan, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. "Neural architecture search using deep neural networks and monte carlo tree search." Proceedings of the AAAI Conference on Artificial Intelligence, (AAAI, 2020).

Wang, Linnan, Rodrigo Fonseca, and Yuandong Tian. "Learning search space partition for black-box optimization using monte carlo tree search." Proceedings of the Neural Information Processing Systems Conference, (NeurIPS, 2020).

Wang, Linnan, Saining Xie, Teng Li, Rodrigo Fonseca, and Yuandong Tian. "Sample-efficient neural architecture search by learning actions for monte carlo tree search." IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI, 2021).

PART.2. The next part describes our effort to expedite the evaluation of neural networks. The data-driven design process requires evaluating many DNN configurations, and the most straightforward approach is to train each neural network from scratch. However, training a neural network is quite time-consuming, not to mention training thousands of them in the entire design process. Reducing the evaluation cost of neural networks is critical. Otherwise, the computations can easily be prohibitive to the current hardware. This chapter presents two solutions, either using multiple super-nets to predict the performance of a neural network without training or training a network on distributed systems. Using the super-net only requires training the super-net itself, while the super-net predicts the performance of the rest networks. Though the performance prediction is inaccurate, the super-net approach is speedy. The training approach is very accurate in the performance evaluation for training every neural network from scratch, but the cost is too high for people with access to a few GPUs. So the training approach is suitable for chasing SoTA results with many GPUs, while the super-net approach is suitable for getting a working solution with a few GPUs. Chapter 6 describes the few-shot NAS developed together with Zhao et al. [Zhao et al., 2020] to drastically improve the performance prediction by only training a few more super-nets. Our approach hits a middle ground between super-net methods and the training methods, therefore being both accurate and fast in the performance evaluation. We present extensive results on various search methods, datasets, and tasks to demonstrate the effectiveness of the few-shot NAS. Chapter 5 presents a new Fast Fourier Transform (FFT) based gradient sparsification technique to accelerate the distributed DNN training by reducing the size of communications. We present a theoretical bound to show how the ratio of dropped information affects the convergence and accuracy and point out a solution to recover the accuracy. Many empirical experiments also show that our FFT sparsification can accelerate the distributed DNN training and maintain a good accuracy for incurring less approximation error using FFT. The following publications contribute to this part of the thesis.

Zhao, Yiyang, Linnan Wang, Yuandong Tian, Rodrigo Fonseca, and Tian Guo. ”Few-shot neural architecture search.” Proceedings of the International Conference on Machine Learning, (ICML, 2021).

Wang, Linnan, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. ”FFT-based Gradient Sparsification for the Distributed Training of Deep Neural Networks.” Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, (HPDC, 2020).

PART.3. I then turn to the system designs to support algorithms. It is a standard practice to train neural networks on the GPU today, while limited GPU DRAM places an undesired restriction on the network design domain. We have seen a growing interest in training large neural networks such as language models [Brown et al., 2020] or using the super-net for the performance predictions [Pham et al., 2018a]. All of these models require memory footprint far beyond the GPU DRAM. Chapter 6 presents a new deep learning framework to support very large neural networks on the limited GPU DRAM. The framework features three new system memory optimization techniques, including liveness analysis, unified tensor pool, and cost-aware recomputation. Together they effectively reduce the network-wide peak memory usage down to the maximal memory usage among layers. On a 12GB GPU DRAM, our system can train ResNet up to 1920 layers, which is 3.2x deeper than TensorFlow and 12.63x deeper than Torch on the same GPU. The following publication contributes to this part of the thesis.

Wang, Linnan, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. ”Superneurons: Dynamic GPU memory management for training deep neural networks.” Proceedings of the 23rd ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, (PPoPP, 2018).

PART.4. The three parts above should collectively present a coherent solution to build an MCTS based agent to design neural networks. Finally, I show that our agent can design DNNs for various tasks to beat SoTA results.

Chapter 2

Background and Related Work

Using computer simulations to design the neural network via a data-driven approach has been tried since the 1980s [Schaffer et al., 1992]. Yet those early attempts to solve this daunting problem, while yielding some notable success in some small-scale tasks, had also encountered many difficulties, especially when the community did not fully recognize the power of neural networks at that time. For example, the limited computing power in the 1980s is one major hurdle to try out thousands of design configurations. Besides, the lack of large-scale labeled datasets also limits neural networks to unleash their remarkable capabilities in tasks such as translation and image recognition. With the advent of powerful GPU and ImageNet in the 2010s, the community recognized neural networks as a powerful technology. But it was not until 2016 that the interests in automating the design of neural networks have been stirred once again for showing the case that an algorithm-designed neural network for the first time reached the SoTA human-designed network on the ImageNet using 500 GPUs [Zoph and Le, 2016]. And that time, the technology was named Neural Architecture Search (NAS).

This chapter provides a review of recent advances in automating the design of neural networks and explains the basic technical details and their pros and cons as a prerequisite to introducing our work. Much of our review is engineering focused, which means that we will scrutinize the methodology from building a practical agent to design neural networks. Therefore, we're unconcerned about questions such as whether the perception in a neural network captures the actual biological structure or not. Instead, we focus on building a practical agent to use existing operators such as convolution, pooling, or activation to assemble a neural network as a data flow graph for solving various tasks in natural language processing or computer vision. With this goal, we seek new efficient algorithms and build a reliable distributed system infrastructure so that

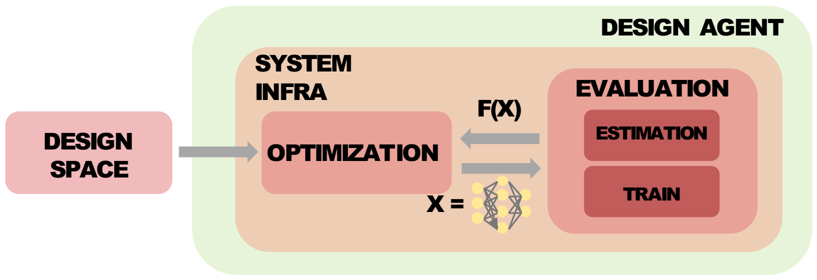
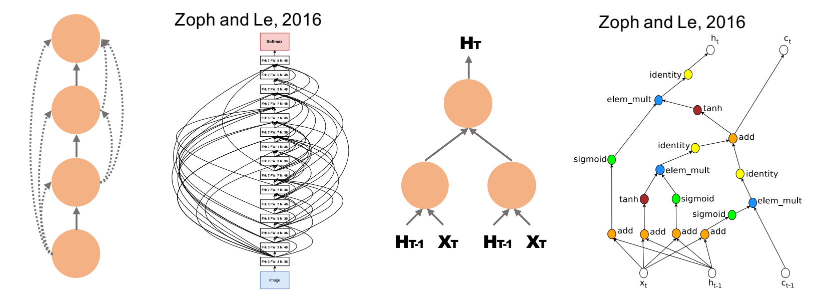


Figure 2.1: The abstraction of key components in the neural architecture search.

the algorithm and system can work together synergistically.

After surveying a variety of journals and proceedings, we abstract any existing NAS agent into 3 key modules, which are *design space*, *optimization* and *evaluation*. Fig. 2.1 provides an overview of these components. The *design space* represents the desired way to create the neural networks, such as setting the maximal depth, the types of operators to use, or constructing linear or non-linear structures [Wang et al., 2018a]. The design space varies from task to task, and these constraints imposed on the design space can alternatively be viewed as constraints to bound the following architecture optimization. Here we assume any architecture can be represented by a vector $\mathbf{x}$, and the *optimization* intends to find the optimal configuration of $\mathbf{x}$ on the feasible region bounded by the design space. The *optimization* module proposes an architecture $\mathbf{x}$ for the *evaluation* module to get the performance of the network $f(\mathbf{x})$. Please note that the metrics used by evaluation can be multi-objective, then the optimization finds the optimal Pareto front to make the trade-off among all the objectives. The extremely costly evaluation of a neural network gives rise to techniques for approximating the performance of a neural network without training. Therefore, the *evaluation* module contains a sub-module for the performance estimation and a sub-module to get true performance by training. Finally, with the new evaluated neural network, the *optimization* module augments the data to inform the next decision.

We organized the rest of this chapter to focus on the recent advances in these three modules. Sec. 2.1 reviews several design spaces for various tasks. Many methods can be applied in the optimization module, such as reinforcement learning, Bayesian optimization, evolutionary algorithms, gradient descent, Monte Carlo tree search, etc. We discuss the pros and cons of these algorithms for NAS in sec. 2.2. Finally, we demystify the estimation and the training methods in the evaluation module so that readers can clearly understand the trade-off between the computations and final accuracy. Though there is a rich literature on NAS in the 1980s, the rest of this chapter primarily focuses on recent works after 2016.



(a) The CNN design space (b) The searched CNN structure (c) The RNN design space (d) The searched RNN structure

Figure 2.2: The design space used by [Zoph and Le, 2016].

2.1 Design Space

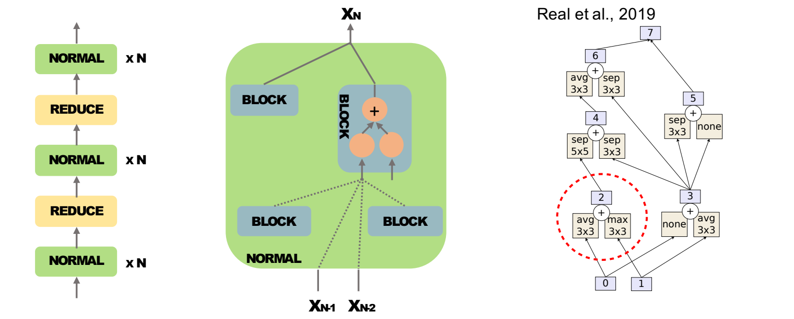
2.1.1 Graph Based Search Space

Perhaps the most intuitive way to represent a neural network is the graph. Despite their different functionalities, the 27 types of existing neural networks [Tch, 2017] suggests the graph structure. Using a graph-based design space, Zoph and Le [Zoph and Le, 2016] are the first to perform a large-scale NAS on 500 GPUs by showing that an algorithm-designed neural network can perform nearly as good as the SoTA network designed by the expert. Fig. 2.2 shows the design space used in [Zoph and Le, 2016]. For example, the tunable components for a CNN are the length of the neural network and the configurations of the convolution layer, including filters, strides, and the source of input. For generating recurrent neural networks, they use the basic motif shown in Fig. 2.2(c) to construct a hierarchical search space. Each node in this motif can choose operators from element-wise multiplications, activation functions, addition, and other basic operations. By recursively applying these motifs, Fig. 2.2(d) suggests the design of this search space can incorporate the formulation of LSTM.

Here we review two design spaces inspired by this early work in Fig. 2.2 that are also widely used in this thesis and NAS literature today.

NASNet Search Space

Fig. 2.2 suggests the hierarchical motif pattern used for designing RNN is more generic than the sequential CNN search space, then Zoph et al. [Zoph et al., 2018b] explores this motif pattern for designing CNN and

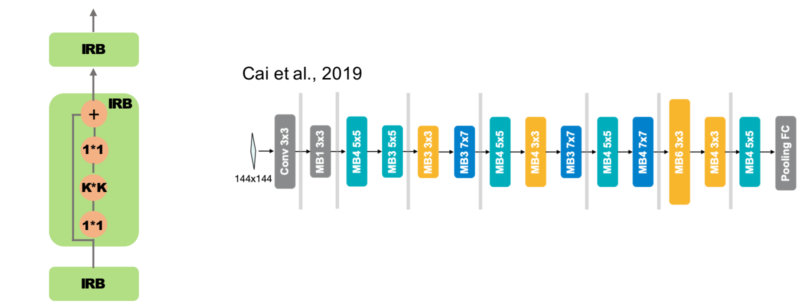


(a) The NASNet network structure (b) The cell structure (c) The searched cell structure

Figure 2.3: The illustration of NASNet search space.

propose the NASNet search space. It turns out that the NASNet search space works astonishingly well for image recognition. Notably, using the NASNet search space, their algorithm found a family of networks termed NASNet that beat the best human-designed networks on ImageNet by a non-negligible margin. Though priors works [Zoph and Le, 2016, Baker et al., 2016] conducted a large-scale search on CIFAR-10, it is the first time that NAS pushes SoTA results on ImageNet that is an authoritative dataset widely recognized by academia and industry and far more challenging than CIFAR-10 in any dimensions.

A network in the NASNet design space constitutes two types of normal and reduce cells sequentially stacked together by following the pattern in Fig.2.3(a). The image sequentially flows through these cells, and the output of the last normal cell feeds into a softmax to produce a probabilistic distribution. The normal cell maintains the input and output dimensions with the padding, while the reduce cell halves the height and width by setting the stride to 2. Fig. 2.3(b) demonstrates the internal hierarchical structure of a cell, and Fig. 2.3(c) provides an example of NASNet cell. Please note we only describe normal cells here, as the reduce cell shares the same structure. Fig. 2.3(b) suggests the two inputs of a cell are from the outputs of the last two cells. A cell has blocks in it, and each block also has two inputs, which can be any outputs of proceeding blocks or the inputs to the cell inputs, e.g., in Fig. 2.3(b). Finally, all the outstanding outputs of blocks are concatenated in the channel dimension as the cell's output. A block consists of a left and a right branch, and a branch consists of one or multiple basic network layers sequentially linked together. The outputs of both branches add together as the block's output. We have extensively used this NASNet search for evaluations.



(a) The EfficientNet design space (b) The searched CNN

Figure 2.4: The illustration of EfficientNet search space.

EfficientNet Search Space

The branches in a NASNet cell complicate the design; therefore networks from this search space are generally slower to execute than the AlexNet style linear networks. In many scenarios, the latency of a network is critical in practice, motivating researchers to look for a simplified search space to design good performing yet fast CNN. The EfficientNet search space is one of the most popular search spaces for this purpose. Compared to the NASNet search space, the EfficientNet only comprises sequentially connected Inverted Residual Block (IRB) [Tan and Le, 2019a]. Fig 2.4(a) demonstrates the 3 components inside a IRB. After the first convolution with the 1-by-1 kernel size to scale up the channels, there is a depth-wise k-by-k convolution followed by another 1-by-1 convolution to scale back the channels. So the design space searches for the expansion ratio induced by two 1-by-1 convolutions, the filter, and kernel size for the middle depthwise convolution. We use this search space to design low flops models.

While the search spaces discussed so far are acyclic graphs, some networks can have the cyclic graph pattern. For example, the parameter sharing in ALBERT [Lan et al., 2019] diverts the output from attention or several attention layers back to its input, looping for a few times. The design space should vary from tasks to tasks, and the EfficientNet and NASNet search spaces are majorly for tasks in computer visions. Therefore, the design of search space can be diverse to cover a broad scope of tasks.

2.1.2 Graph Generator based Search Space

Apart from explicitly building neural networks, another interesting direction is to find a meta-function that generates neural networks following certain patterns. One example is using graph generators. For example, [Xie et al., 2019] draw insights from random graph theory, using Erdős–Rényi [Erdős and Rényi, 1960],

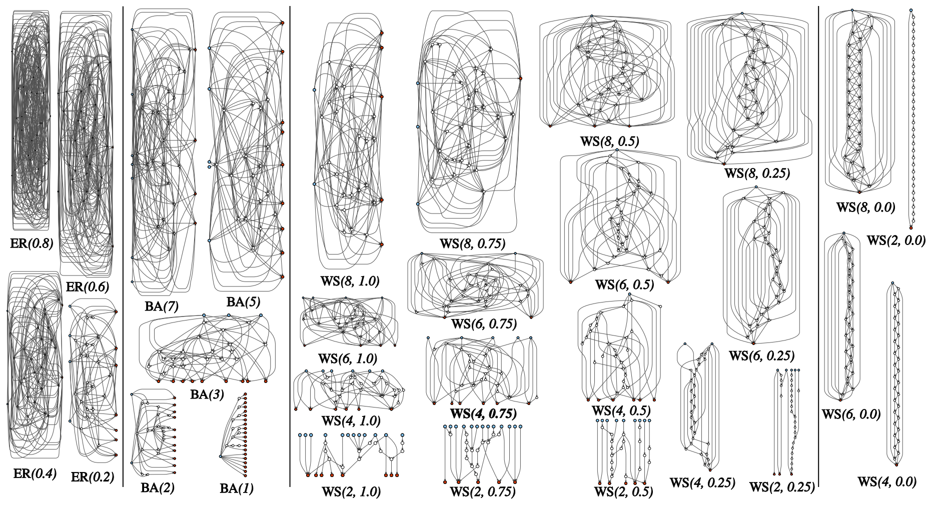


Figure 2.5: The illustration of networks generated by Erdős–Rényi(ER), Barabási–Albert(BA) and Watts–Strogatz(WS) random graph generators. These models takes in a few parameters to generate graph that follows the certain pattern. Figure courtesy of [Xie et al., 2019].

Barabási–Albert model [Albert and Barabási, 2002] and Watts–Strogatz model [Watts and Strogatz, 1998] to generate 3 classes of randomly connected neural networks. Fig. 2.5 demonstrates some networks generated by these three random graph generators. Though these graphs look pretty random, their results are surprisingly better than or comparable to all existing hand-designed networks [Xie et al., 2019, You et al., 2020]. The result introduces a new perspective to learn a meta-function to represent various search spaces so that the optimization can tweak the hyper-parameters for the graph generators to find a class of good networks. Please note that the graph generator [Li et al., 2018b] is not the only way to represent search spaces; it can be any meta-function to delegate a set of search spaces. Though this thesis does not explore this direction, it can be great future work.

2.2 The Optimization Module

The optimization module maximizes predefined evaluation metrics by tuning the network architectures. It proposes a new network design $\mathbf{x}$ that satisfies the definitions of design space, queries the network performance $f(\mathbf{x})$ from the evaluation module. This section reviews a broad spectrum of optimization methods and discusses each techniques' pros and cons in the context of NAS.

2.2.1 Black Box Optimization

Black-box optimization refers to solve a set of problems without algebraic formulation of the system. Yet there are some advances in the deep learning theory [Du et al., 2018], an analytical formulation of meta-design of neural networks is still largely unknown. Therefore, it is natural to treat neural networks as a black box and use this class of techniques to optimize the design. Actually, it has been a long history to combine the black-box optimization and the neural network design; some early works can be dated back to the 1980s [Belew and McInerney, 1989, Changeux, 1980, Schaffer et al., 1992].

The black box optimization solves the following problem preferably with fewer samples $\mathbf{x}$:

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in X} f(\mathbf{x}) \tag{2.1}$$

Though we don't know the formulation of system f, the system returns $f(\mathbf{x})$ given an input $\mathbf{x}$. This mechanism gives rise to a set of techniques that maximize f by trying different configurations of $\mathbf{x}$ based on the previous experience. Here we review a few prominent black-box solvers, including evolutionary algorithm and Bayesian optimization.

Grid Search

Grid search is among one of the simplest yet prevalent methods to solve black-box optimization. It specifies a set of values for each problem dimension, then performs a Cartesian product of these sets to generate a set of points that take a sweep of search space [Bergstra and Bengio, 2012]. Another alternative technique to generate near uniformly distributed points are Latin hypercube sampling [McKay et al., 2000] or Sobol sequences [Sobol', 1967]. Apparently, the grid search is only limited to small scale problem due to the curse of dimensionalities, so people usually practiced grid search in an iterative manner such that reducing the search space at iteration N+1 to the adjacent regions of good points at iteration N. Grid search renders two major drawbacks when applied to NAS. First, it evaluates many samples in sweeping the search space, while the cost of evaluating a neural network, even by the performance prediction, is non-trivial. Second, the grid search lacks a mechanism to systematically balance the exploration and exploitation, as the global optimum is not necessarily located around the good points at iteration N.

Evolutionary Algorithm

Evolutionary Algorithm (EA) is a set of optimization techniques inspired from Darwinism [Huxley, 1860] that 'all species arise and develop through the natural selection of small and inherited variations that increase the individual's ability to survive and reproduce' [Darwin, 1909]. By mimicking the biological evolution, EA is an iterative algorithm that gradually improves the solution over generations. First, EA starts with a random population of individuals, with their performance measured by a fitness function. Second, EA weeks out the low-performing individuals and only keeps the top-performing candidates for reproducing offspring through crossover or mutations. Finally, the new offspring and the high-performing individual will form a new population for the next iteration. Here we can view the individual as a candidate solution to the problem and the fitness function as the objective function. The mutation and crossover are equivalent to perturbing the current candidate solutions. While the process of EA seems to be simple, it can solve very complex problems in practice [Salimans et al., 2017]. EA is a vast family of algorithms; here, we focus on three prominent evolutionary strategies for choosing individuals in reproduction.

- Simple Evolution: this is the simplest strategy that uses a threshold to keep individuals with the highest fitness scores for the reproduction, e.g., choosing the top 10%. Let's represent an individual as a vector; then, the crossover randomly selects two survivors for recombining a segment of them to form a new individual, and the mutation simply randomly perturbs a segment of the survivor. One drawback of this approach is that we need to define the mutation and crossover rules for different tasks explicitly. An alternatively generic approach is to sample new individuals from a multi-variate Gaussian $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and the update rule of $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is as follows

$$\boldsymbol{\mu}^{t+1} = \frac{1}{N} \sum_{\mathbf{x} \in \mathcal{X}^t} \mathbf{x}^t \tag{2.2}$$

$$\boldsymbol{\Sigma}^{t+1} = \mathbf{I} \tag{2.3}$$

  where $\mathcal{X}^t$ is the group of survivors that surpass the fitness threshold at iteration t, and $\mathbf{I}$ is the identity matrix. The update rule suggests that the simple evolution strategy only adapts $\boldsymbol{\mu}$ from top individuals in the population, so the improvement is bounded by $\mathbf{I}$ at each iteration.

- Covariance Matrix Adaptation Evolution Strategy (CMA-ES): following the idea of sampling from a multivariate Gaussian $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, CMA-ES adapts both $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ based on top individuals. There are several advanced versions of CMA-ES [Hansen, 2006], here we describe the simplest form to demonstrate the gist of CMA-ES. Similarly, CMA-ES employs a threshold to pick the candidates ($\mathcal{X}$) with top fitness scores, and adjust $\boldsymbol{\mu}$ according to individuals $\mathbf{x} \in \mathcal{X}$. The key change is to adjust variance $\boldsymbol{\Sigma}$ as follows,

$$\boldsymbol{\Sigma}_{ij}^{t+1} = \frac{1}{N} \sum_{\mathbf{x} \in \mathcal{X}^t} (\mathbf{x}_i - \boldsymbol{\mu}_i^t)(\mathbf{x}_j - \boldsymbol{\mu}_j^t) \tag{2.4}$$

where $\boldsymbol{\Sigma}_{ij}^{t+1}$ represents the covariance located at the ith row and jth column at iteration t+1, and $\boldsymbol{\mu}_i$ is the ith element of $\boldsymbol{\mu}$. Since $\boldsymbol{\Sigma}$ controls the size of the sampling region, the above equation suggests the sampling region grows if survivors sparsely scatter in the sampling region when the algorithm is making an improvement, and shrink otherwise. Therefore, CMA-ES is generally faster than simple evolution.

- Natural Evolutionary Strategies (NES): previous approaches all discard individuals with a low fitness score, while discarded individuals may provide a better estimate about what not to do. So the improvement of NES over CMA-ES is to use all the individuals in updating $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ by calculating the gradient toward $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ to maximize the expected fitness under the search distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, i.e.

$$J(\theta) = \int f(\mathbf{x}) \pi(\mathbf{x}|\boldsymbol{\theta}) d\mathbf{x} \tag{2.5}$$

where $\pi(\mathbf{x}|\boldsymbol{\theta})$ is a probability distribution function, and $\boldsymbol{\theta}$ is the parameter of this density, e.g. $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ for the density of Gaussian distribution. $f(\mathbf{x})$ is the fitness function. By using the same log-likelihood trick as in the REINFORCE algorithm [Williams, 1992], we can calculate $\nabla_\theta J(\theta)$ as follows:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{k=1}^{N} f(\mathbf{x}_k) \nabla_\theta log \pi(\mathbf{x}_k, \boldsymbol{\theta}) \tag{2.6}$$

The complete derivation is straightforward in [Wierstra et al., 2014], and they also provide the update equations for using the multi-variate Gaussian distribution.

These evolutionary strategies all suggest that there are numerous evaluations of candidate solution $\mathbf{x}$ inside the fitness function at each iteration. Even if it can pair with a surrogate model [Jin, 2011], Evolutionary Algorithm still requires many samples to work well. In NAS, $\mathbf{x}$ is equivalent to a design of neural network and evaluating $\mathbf{x}$ is either from training or predicting (Fig. 2.1). Training a neural network is very expensive, while the prediction is also non-trivial as we need to evaluate the masked Supernet on a test dataset. Therefore, Evolutionary Algorithm usually needs many GPUs to work well.

Muller et al. [Miller et al., 1989] and Kitano [Kitano, 1990] pioneer in applying Evolutionary Algorithm to the design of neural networks. Their work uses the genetic algorithm to propose architectures and use the back-propagation to train and evaluate an architecture. [Stanley and Miikkulainen, 2002] further explores the idea of evolving both the topology of a neural network and the weights. Due to the previous success, many researchers also explore Evolutionary Algorithm for NAS in recent years. [Real et al., 2019a] proposes Regularized Evolution to design a neural network using the NASNet search space. The performance of searched network beats the best human-designed network on ImageNet, and they also show many downstream tasks such as object detection can be improved using the searched network. Though Regularized Evolution is similar to the simple Evolution described above, they performed experiments on 500 GPUs to get this astonishingly good result. Deploying neural networks to mobile devices is also an important topic today. In this scenario, we need to trade-off multiple objectives on the Pareto front in the design optimization. NSGA [Deb et al., 2002] is a popular multiple-objective Evolutionary Algorithm, and [Lu et al., 2019] et al. applies NSGA to find the Pareto front in designing a fast and good neural network.

Bayesian Optimization

Bayesian optimization is an efficient global optimization method that utilizes the Bayesian rules to optimize a black-box function. It is known to be sample efficient for using Gaussian Process as a surrogate model; therefore, Bayesian Optimization seems to be a great fit to the optimization module since evaluating a neural network is very expensive. Bayesian optimization consists of two major parts, a surrogate model that maintains a probabilistic belief about the function and an acquisition function that determines where to sample next. The surrogate model captures the prior belief about the function by modeling toward all the previous samples $(\mathbf{x}, f(\mathbf{x}))$, then it yields the posterior distribution to the acquisition function for sampling the next $\mathbf{x}$. The new $\mathbf{x}$ and $f(\mathbf{x})$ will be merged into the dataset to update the surrogate in the next iteration. Here we mainly review the Gaussian Process and several commonly used acquisition functions in Bayesian optimization.

- Gaussian Process is inarguably the foundation to the Bayesian Optimization. The beauty of Gaussian Process Regression (GPR) is the non-parametric modeling that the posterior predictive distribution can be acquired by marginalizing all the possible models without learning it. Let's consider a general regression problem,

$$y = \mathbf{w}^T\phi(\mathbf{x}) + \epsilon \tag{2.7}$$

where ϕ is a kernel function. This regression is generic as we can use different kernels to perform the various types of regression. So different $\mathbf{w}$ corresponds to different predictive models. Given the dataset $D = \{(\mathbf{x_1}, y_1), ..., (\mathbf{x_n}, y_n)\}$, the posterior predict distribution is

$$P(y|D,\mathbf{x}) = \int_{\mathbf{w}} P(y, \mathbf{w}|\mathbf{x}, D)d\mathbf{w} \tag{2.8}$$

$$= \int_{\mathbf{w}} P(y|\mathbf{x}, \mathbf{w}, D)P(\mathbf{w}|D)d\mathbf{w} \tag{2.9}$$

The elegance of GPR is that we are averaging all the possible results from all the possible $\mathbf{w}$, and $P(y|D,\mathbf{x})$ is a Gaussian distribution if the priors are also Gaussian likelihood, i.e.

$$P(y_*|D,\mathbf{x}) \sim \mathcal{N}(\boldsymbol{\mu}_{y^*|D}, \boldsymbol{\Sigma}_{y^*|D}) \tag{2.10}$$

and $\mathbf{x}^*$ and $y^* = f(\mathbf{x}^*)$ are predictive test points and results, respectively. Let's assume y follows a Gaussian $\sim \mathcal{N}(\mu, \boldsymbol{\Sigma})$ and we can center the distribution at 0 by subtracting the mean estimated from samples. Because an entry in Σ measures the similarity between two data points, here we can use a kernel function to compute Σ instead, such as RBF kernel below

$$\boldsymbol{\Sigma}_{ij} = \tau e^{\frac{-\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\sigma^2}} \tag{2.11}$$

If we represent $\boldsymbol{\Sigma}$ as $\left| \begin{smallmatrix} \boldsymbol{K} & \boldsymbol{K}_* \\ \boldsymbol{K}_*^T & \boldsymbol{K}_{**} \end{smallmatrix} \right|$. $\boldsymbol{K}$ represents the kernel results among training samples, $\mathbf{K}_*$ represents the kernel results among training samples $\mathbf{x} \in D$ and predicting points $\mathbf{x}^*$, and $\boldsymbol{K}_{**}$ represents the kernel results among predicting points. Then the conditional distribution in Eq. 2.10 is

$$\mu_{y^*|D} = K_*^T K^{-1} y \tag{2.12}$$

$$\Sigma_{y^*|D} = K_{**} - K_*^T K^{-1} K_* \tag{2.13}$$

Now we complete the introduction of GPR, and more information can be found at [Weinberger, 2017]. Bayesian optimization further utilizes $\mu_{y^*|D}$ and $\Sigma_{y^*|D}$ in the acquisition to select samples.

K^{-1} in Eq. 2.12 indicates the complexity cubically increases with the number of training samples since the matrix inverse is an $O(n^3)$ operation. While most real-world problems are high dimensional, and reliably optimizing a complex function requires many evaluations, which has motivated many works to scale up BO by approximating the expensive Gaussian Process (GP), such as using Random Forest in SMAC [Hutter et al., 2011a], Bayesian Neural Network in BOHAMIANN [Springenberg et al., 2016], and the tree-structured Parzen estimator in TPE [Bergstra et al., 2011]. BOHB [Falkner et al., Ster] further combines TPE with Hyperband [Li et al., 2017] to achieve strong any time performance. Using a sparse GP is another way to scale up BO [Seeger et al., 2003, Hensman et al., 2013]. However, sparse GP only works well if there exists sample redundancy, which is barely the case in high-dimensional problems.

- The acquisition function (ϕ) selects where to sample next using the Gaussian process priors. There are many strategies to implement ϕ, but the process can be generalized as follows:

$$maximize \qquad \phi(\boldsymbol{\mu}_{y^*|D}, \boldsymbol{\Sigma}_{y^*|D}, y^+) \tag{2.14}$$

$$s.t. \qquad \mathbf{x}^* \in D \tag{2.15}$$

where $\mathbf{x}^*$ is the predictive point, and y^+ is the best $f(\mathbf{x})$ sampled so far. Please note K_* and K_{**} in the equation 2.12 are computed from $\mathbf{x}^*$ and current samples $\mathbf{x}$. Here we review 2 commonly used ϕ in the Bayesian optimization.

The first one is Probabilistic Improvement (PI). The intuition of PI is to maximize the probability that the subsequent evaluation will improve upon the current best y^+. Let's denote ψ as the cumulative distribution function of the standard Gaussian distribution, so ϕ_{PI} is defined as follows:

$$\phi_{PI}(\mathbf{x}^*) = 1 - P(f(\mathbf{x}^*) < y^+) = 1 - P(\frac{f(\mathbf{x}^*) - \boldsymbol{\mu}_{y^*|D}}{\boldsymbol{\Sigma}_{y^*|D}} < \frac{y^+ - \boldsymbol{\mu}_{y^*|D}}{\boldsymbol{\Sigma}_{y^*|D}}) \tag{2.16}$$

$$= \psi(\frac{y^+ - \boldsymbol{\mu}_{y^*|D}}{\boldsymbol{\Sigma}_{y^*|D}}) \tag{2.17}$$

Expected Improvement is another acquisition function that maximizes the average improvement w.r.t $f(\mathbf{x}^*)$ after sampling at $\mathbf{x}^*$, i.e.

$$\phi_{EI}(\mathbf{x}^*) = \mathbf{E}(max(f(\mathbf{x}^*) - y^+, 0)) \tag{2.18}$$

[Jones et al., 1998] shows that ϕ_{EI} can be analytically solved, yielding the following update rule

$$\phi_{EI}(\mathbf{x}^*) = (\boldsymbol{\mu}_{y^*|D} - y^+ - \xi)\psi(\frac{\boldsymbol{\mu}_{y^*|D} - y^+ - \xi}{\boldsymbol{\Sigma}_{y^*|D}}) + \boldsymbol{\Sigma}_{y^*|D} p(\frac{\boldsymbol{\mu}_{y^*|D} - y^+ - \xi}{\boldsymbol{\Sigma}_{y^*|D}}) \tag{2.19}$$

Optimizing the acquisition function is not trivial, especially when $\mathcal{X}$ is high dimensional, while a sub-optimal solution to the acquisition optimization will deteriorate the performance. Existing Bayesian methods such as SMAC [Bergstra et al., 2011], TPE [Bergstra et al., 2011], and BOHB [Falkner et al., Ster] use iterated local search/evolutionary algorithm to optimize the acquisition, and [Wilson et al., 2018] provides an in-depth discussion of the impact of different optimization algorithms on the performance. In this thesis, chapter 5 proposes a new MCTS based method that learns to partition the search space; then, the acquisition function can be easily optimized inside a partition instead of the entire search space.

The existing acquisition function also tends to be myopic, leading to over-exploring the boundary of a search space [Oh et al., 2018]. Therefore, ML practitioners usually observe that the performance of Bayesian optimization is as poor as a random search, especially in high-dimensional problems. There is much work to specifically study high-dimensional Bayesian optimization [Wang et al., 2013, Kawaguchi et al., 2015, McIntire et al., 2016, Chen et al., 2012, Kandasamy et al., 2015, Wang et al., 2016c]. One category of methods decomposes the target function into several additive structures [Kandasamy et al., 2015, Gardner et al., 2017], which limits its scalability by the number of decomposed structures for training multiple GP. Besides, learning a good decomposition remains challenging. Another category of methods is to transform a high-dimensional problem into low-dimensional subspaces. REMBO [Wang et al., 2016c] fits a GP in low-dimensional spaces and projects points back to a high-dimensional space that contains the global optimum with a reasonable probability. Binois et al. [Binois et al., 2020] further improve the distortion from Gaussian projections in REMBO. While REMBO works empirically, HesBO [Nayebi et al., esBO] is a theoretical sound framework for BO that optimizes high-dimensional problems on low dimensional sub-spaces embeddings; In BOCK [Oh et al., 2018], Oh et al. observed that existing BO spends most evaluations near the boundary of a search space due to the Euclidean geometry, and it proposed transforming the problem into a cylindrical space to avoid over-exploring the boundary. Recent works explore space partitioning and local modeling that fits local models in promising regions to achieve strong empirical results in high-dimensional problems. For example, EBO [Wang et al., 2017b] uses an ensemble of local GP on the partitioned problem space. Based on the same principle of local modeling as EBO, recent trust-region BO (TuRBO) [Eriksson et al., uRBO] has outperformed other high-dimensional BO on a variety of tasks. However, their space partitions follow a fixed criterion (e.g., K-ary uniform partitions) independent of the objective to be optimized. Following the learning path, one under-explored direction is to learn the space partition, and this thesis explores this direction in Chapter 5.

Monte Carlo Tree Search

Besides the recent success in games [Silver et al., 2016, Schrittwieser et al., 2019, Silver et al., 2017], Monte Carlo Tree Search (MCTS) is also widely used in robotics planning and black box optimization [Buşoniu et al., 2013, Munos, 2014, Weinstein and Littman, 2012, Mansley et al., 2011]. The formal introduction of MCTS can be found in chapter 3. Here we focus on reviewing recent advances of MCTS in black-box optimization.

When MCTS is applied to optimize a function, it can be seen as an optimistic sampling strategy where, at each round, MCTS explores the space believed to be promising according to previous samples [Munos, 2011]. Alternatively, we can view MCTS as partitioning the search space into regions of different performance, then sample from the promising region. Munos pioneers in this line of research by proposing Deterministic Optimistic Optimization (DOO) and Simultaneous Optimistic Optimization (SOO) [Munos, 2011]. DOO uses a tree structure to partition the search space by recursively bifurcating the region with the highest upper bound, i.e., optimistic exploration, while SOO relaxes the Lipschitz condition in DOO on the objective function. HOO [Bubeck et al., 2011] is a stochastic version of DOO. While prior works use K-ary partitions, Kim et al. show Voronoi [Kim et al., 2020a] partition can be more efficient than previous linear partitions in high-dimensional problems. In this thesis, based on the idea of space partitioning, we extend current works by learning the space partition so that the partition can adapt to the distribution of $f(\mathbf{x})$. Besides, the sample efficiency of MCTS alone is not attractive for lacking a surrogate model such as GP in Bayesian Optimization. We further show that MCTS and Bayesian optimization can work synergistically together: the learned space partition helps Bayesian optimization avoid over-exploring by bounding within a small region and reduce the complexity in optimizing the acquisition function. Chapter 5 provides more details to our approach to learning the search space partition, and Chapter 4 discusses applying our new MCTS method to build the optimization module in the NAS agent.

2.2.2 Reinforcement Learning

Reinforcement Learning (RL) studies the problem of learning a policy for an agent to act in an environment so that the cumulative reward can be maximized. RL is one of 3 basic pillars in machine learning alongside

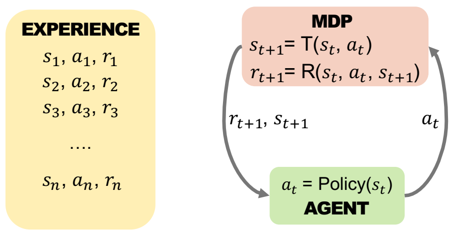


Figure 2.6: The workflow of RL that interacts with a MDP.

with the supervised and unsupervised learning. This section briefly reviews of basic concepts in RL and their applications to NAS.

Background

Markov Decision Process (MDP) [Bellman, 1957] is a general mathematical framework to formalize the agent-environment interactions in RL. The NAS agent builds a neural network by wiring various operators together, so the process can be modeled by a discrete time MDP that defines the reward and the probability of destination states after taking an action at the current state. The discrete time MDP consists of six tuple, (S, A, R, T, γ), where S represents states in MDP that describe all the possible configurations of this discrete world; A is a set of possible actions; R is a reward function that outputs a reward given the current state after taking an action to the next state; T is a transition function that outputs the probability of arriving the next state after taking an action on the current state; γ is a discount factor to regulate the preference over a long-term and shot-term rewards. There are many kinds of MDPs designed for different scenarios. For example, continuous time MDP is often used in the robotics such as the joint rotation control in a robot arm. An agent may not observe its current state in an MDP such as the case in Elevator Control [Cassandra, 1998], so this kind of problems use Partially Observable Markov Decision Process (POMDP) to model the environment.

While MDP is a very generic in the above formulation, modeling the neural network design only requires a deterministic MDP. In NAS, an architecture corresponds to a state so the actions are adding a new layers from a pool of operators or modify the hyper-parameters in existing layers. Every time an agent modifies an architecture (s), the next architecture ($s^{'}$) is always certain, i.e. $s \times a \rightarrow s^{'}$ is deterministic. The reward of a state is the performance of the network represented by the state. Fig. 2.6 depicts how RL interacts with MDP in the context of NAS. Given an architecture s_t, an agent selects an action a_t following the current policy that modifies the architecture to s_{t+1}. The agent collects the reward r_{t+1} after taking the action a_t. The agent loops this action for indefinite steps until reaching a terminating condition, and all the past state, action and reward will be stored into the experience to train the policy. RL tries to learn a good policy for an agent that maximizes the long term discounted reward,

$$\sum_{t=1}^{\infty} r_t \gamma^t \tag{2.20}$$

Here the agent tries to find a path with the largest cumulative performance sum of networks located on the path. However, NAS mostly concerns about a single architecture that delivers the best performance. RL also optimizes a policy for the entire states, so the number of samples required to find a good policy for all states is inevitably higher than the black box optimization that only finds a good solution.

RL algorithms

RL algorithms, in general, can be classified into model-based RL, model-free RL, and policy-based RL. This three-category is based on the functions to be estimated. Model-based RL estimates the reward ($\hat{R}$) and transition ($\hat{T}$) functions. If the action-value function, e.g. ($\hat{Q}$), is estimated, it is the model-based RL. Policy-based RL directly estimates the policy π. When combined with deep neural networks, model-free and policy-based RL, a.k.a deep RL methods, have demonstrated great success in many challenging domains from Atari [Mnih et al., 2013] to robotics [Levine et al., 2016]. Here we only focus on reviewing the model-free and policy-based RL methods.

Q-learning [Watkins and Dayan, 1992] is one of the classical algorithms in model-free RL. Q-learning utilizes a Q-function to build the policy, which can be viewed as a giant lookup table that inputs the current state s and action a and outputs a score. At any states, Q-learning follows the policy of

$$\pi_q(a|s) = \text{argmax}_{a'} Q(s, a^{'}) \tag{2.21}$$

which takes the action yielding the highest q score. So the task is to learn the optimal Q-function (Q^*) from the past experiences (s_t, a_t, r_t) to maximize the cumulative rewards. At each step, Q-learning updates the Q-function as follows,

$$\hat{Q}(s_t, a_t) = \hat{Q}(s_t, a_t) + \alpha(r_t + \gamma * argmax_{a \in \mathcal{A}} \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t)) \tag{2.22}$$

where s_{t+1} represents the next state after taking action a at the current state s_t, γ is the discounted factor and α is the learning rate. Every step generates an entry of s_t, a_t and r_t to update the Q-function. Using a ϵ exploration strategy, Q-learning is theoretically guaranteed to converge to the optimal policy Q^* [Melo, 2001]. There are also many works to extend the Q-learning to the deep RL regime by implementing the Q-function with a deep neural network, such as Deep Q Network [Mnih et al., 2015].

Baker et al. [Baker et al., 2016] implemented the first Q-learning based NAS agent, which treats a state s as a concrete neural architecture and an action a as adding a layer or modifying the hyper-parameters of an existing layer. Training a neural network is very expensive, so training every new network at a new state s is not desired. Baker et al. ingenuously introduce the terminal action to solve this question. Specifically, the agent can choose to terminate at any state when the network represented by the state is actually trained. So the reward for selecting the terminal action is the actual network accuracy, while the reward for transiting among non-terminal states is 0. The terminal action allows the agent to build a very deep neural network before the actual network training. Following this design of MDP and the update rule in Eq. 2.22, they found a network that achieves the top-1 accuracy of 95.38 on CIFAR-10.

The policy-based RL methods directly update the parameterized policy w.r.t the cumulative reward by the gradient descent. Weng provides a comprehensive review of various policy gradient methods in [Weng, 2018]. Here we focus on REINFORCE [Williams, 1992], which has been applied in NAS to achieve great success. Let's define a parameterized stochastic policy as π_θ, e.g., implemented with a deep neural network. So the expected return is

$$J(\theta) = E(\sum_{t=0}^{T} r_t | \pi_\theta) = \sum_{t=i}^{T} P(s_t, a_t | \tau) r_{t+1} \tag{2.23}$$

where i is a starting point, and $P(s_t, a_t|\tau)$ is the probability of s_t, a_t on the trajectory τ. By using the log trick, Williams derives the update rule for a parameterized policy π_θ is as follows,

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t)(\sum_{t^{'}=t+1}^{T} \gamma^{t^{'}-t-1} r_{t^{'}}) \tag{2.24}$$

The full derivation of the above equation is available in [Williams, 1992].

Zoph and Le utilize REINFORCE to implement a NAS agent [Zoph and Le, 2016] that designs a network to achieve the competitive performance as the best human-designed network. They use a Recurrent Neural Network (RNN) to generate the architecture and update the parameters of RNN using REINFORCE. This agent samples an architecture from a Recurrent Neural Network then evaluates the network to get the accuracy as the reward. Finally, the agent computes the gradient w.r.t the rewards and sampled networks following the equation 2.23. This process suggests that the agent needs evaluate a network at every update step, and they use 800 GPUs to evaluate 800 networks concurrently. After training 12800 architectures, the agent finds a network that achieves 96.35% top1 accuracy on CIFAR-10. Although REINFORCE has shown NAS as a promising direction, the results are from massive computations using hundreds of GPUs. However, very few groups have access to hundreds of GPUs running for weeks. The lack of GPUs naturally motivates researchers to look into several possible directions to improve the efficiency, either from enhancing the sample efficiency [Wang et al., 2019a] or expediting the network evaluation [Pham et al., 2018a]. Indeed the goal of RL is to find a policy that maximizes the cumulative rewards, while NAS mainly concerns the performance of the final architecture. Therefore, methods from black-box optimization are more suitable than RL in NAS.

2.2.3 Gradient Descent

Another popular approach is to formalize the architecture optimization and the network training as a bi-level optimization, solved by gradient descent. The formulation of this bi-level optimization is as follows,

$$\min_\alpha \mathcal{L}_{val}(w^*(\alpha), \alpha) \tag{2.25}$$

$$s.t.\ w^*(\alpha) = \text{argmin}_w \mathcal{L}_{train}(w, \alpha) \tag{2.26}$$

which is exactly the same as equation 1.1. The second optimization is the bottleneck of bi-level optimization as training a neural network is extremely expensive, while the gradient descent approximates the architecture gradient by using only one step of the weight update in the network training:

$$\nabla_\alpha \mathcal{L}_{val}(w^*(\alpha), \alpha) \approx \nabla_\alpha \mathcal{L}_{val}(w - \eta * \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha) \tag{2.27}$$

this approximation avoids training the network until the convergence [Liu et al., 2018b]; therefore, it can solve the problem within days, even using 1 GPU. Alternatively, black box optimizations or reinforcement learning either evaluate an architecture by training from scratch or approximate the optimal weight by transferring from a trained Supernet. We will discuss the pros and cons of each method in sec. 2.3.

The above approximation significantly lowers the computation requirement for performing NAS, which has stirred much interest to study gradient-based methods. One direction is to improve the gradient method itself. For example, P-DARTS [Chen et al., 2019b] observes there is a depth gap in the architecture during the search and evaluation steps in the search for CNN on the NASNet search space. Then, P-DARTS proposes progressively increasing the network depth during the search for CNN to match the evaluation setting. Besides, Zela et al. [Zela et al., 2019] observes the regularization robustifies the gradient-based method, either via augmenting data for training or adding an L2 regularization in the training step. One problem with these methods is that these task-dependent settings can change from tasks to tasks or search space to search space. The effectiveness of these methods on different tasks is not guaranteed since most of their evaluations are limited to building a CNN. Most importantly, one of the critical evaluation criteria is the tabular results of final CNN accuracy on CIFAR-10 or ImageNet; while the accuracy can be improved by the tricks irrelevant to the methods such as distillation [Cai et al., 2019a], data augmentation [Nayman et al., 2019a], and Exponential Moving Average (EMA) [Dai et al., 2020]. Although the gradient-based methods are cheap to run, the problem is the aggressive approximation of the optimal weight parameters by only one update. Therefore, the networks found by gradient-based algorithms are generally worse than other methods that train every network from scratch on the same search space [Wang et al., 2021, Luo et al., 2018a].

2.3 Evaluation Module

The evaluation module takes a network architecture proposed from the optimization module to output the network's performance. The simplest evaluation method is to train the network with SGD until convergence. However, training a neural network is highly time-consuming, especially on a large dataset and the data-driven design process constantly evaluates the architectures during the progress of the search. The high cost naturally motivates us to look into solutions to utilize many GPUs to tackle the computation challenges. While parallelizing the search over many GPUs requires careful consideration of the underlying heterogeneous system, an intelligent job scheduler specifically designed to handle the computation pattern in NAS is desired. On the other hand, not everyone can access hundreds of GPUs; this gives rise to a set of approximation techniques that quickly estimates the performance of architecture so that NAS can be done within a few days on a GPU. These approximation techniques are also no magic, trading the evaluation quality for speed. We will also discuss these trade-offs for each method discussed below.

Parallelization

The GPU of today can easily deliver massive computing power such that one NVIDIA V100 server node delivers up to the same performance as 135 CPU only server nodes. This huge advances in the computing

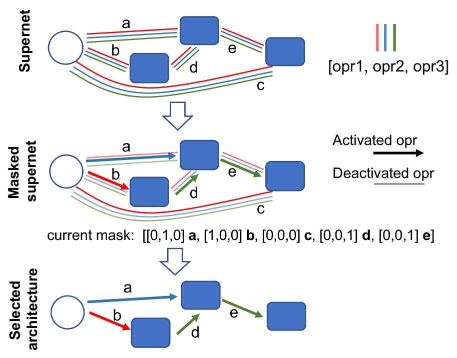


Figure 2.7: Integrating supernet with search algorithms. Before the search comes into play, we pre-train a *supernet* by applying a random mask at each iteration until the convergence. The supernet remains static in the search. When a search algorithm proposes a network, we transform the supernet to the target architecture by multiplying a mask to deactivate the extra operations on a compound edge, or deactivate the entire edge.

infrastructure enables researchers to roll out unprecedently large scale NAS using hundreds of GPUs. For example, the RL agent implemented by Zoph and Le [Zoph and Le, 2016] utilizes 800 GPUs to concurrently train sampled neural networks to update the RNN controller. They parallelize the computations using the parameter server [Li et al., 2014], where they have several shards of a parameter server, that store the shared parameters for several RNN controller replicas. Each controller replica samples a few different architectures, sending each of architecture to a downward GPU computing node to train and evaluate the architecture. The accuracy of the architecture is used as the reward, which is sent back to the parameter servers to calculate the policy gradient. The gradient is subsequently broadcast to update the controller, and the update is in asynchronous fashion to avoid the straggler problem. Besides architecting on the parameter server, this thesis proposes a scalable and fault tolerant client and server style job scheduler to support our MCTS based NAS agent. We will elaborate the details in Chapter 3.

The parallelization accurately evaluates an architecture by training toward the convergence, but the drawback is requiring hundreds of GPUs. Therefore this approach is limited to a small group of researchers who have such access in big tech companies or national labs. While it is possible to rent the GPU node on the cloud, the cost is very high. Recently, the emerging serverless computing seems to be a promising solution to reduce down the cost by taking advantage of spot instances on the cloud. Serverless computing has shown very promising in other application domains [Fouladi et al., 2017], and Ray library has demonstrated some excellent results [Liaw et al., 2018] in the large scale hyper-parameter search using the serverless. Although we did not exploit this direction in this thesis, I strongly believe this is a promising direction and leave it as future work.

Performance Prediction

Another interesting direction is to predict the performance of a neural network. The methods in this category are often distinguished by features used in the prediction. There are numerous works to predict the performance of a network based on the progress of training. For example, Rasley et al. [Rasley et al., 2017] build a cloud scheduler that provisions more computing resources for neural networks that show good loss trace in the early stage. Similarly, Li et al. develop hyperband [Li et al., 2017] to dynamically allocate the resource w.r.t the training progress along with the search. This early stopping technique is also employed by Bowen et al. to save computations from terminating the low-performing networks in the early stage [Baker et al., 2017b]. In general, these works usually use a surrogate model trained with the loss traces of evaluated networks. When a network comes in, the algorithm uses the surrogate model to extrapolate the loss progress in the next several epochs. If the trend predicted from the surrogate model does not look promising, the system terminates the training to provision resources to good candidates. Another group of works tries to learn a predictor that directly predicts the final accuracy of an architecture. For example, in chapter 3, we have discussed several performance predictors using MLP and RNN, and Shi et al. [Shi et al., 2019a] further shows that Graph Neural Network performs better than other regressors. Although several works demonstrate great results [Chau et al., 2020, Shi et al., 2019a, Wen et al., 2020], their search is exclusively guided by the predictor, which can easily trap into a local optimum without implementing any exploration mechanism.

Supernet

Pham et al. [Pham et al., 2018a] proposes a weight sharing scheme to avoid the model re-training using an over-parameterized supernet, which can transform to any architectures in the search space by deactivating extra edges to the network as shown in Fig. 2.7. Many works formulate the training of supernet and architecture optimization as integrated bi-level optimizations solved by SGD [Liu et al., 2018b], while recent works [Guo et al., 2019a, Sciuto et al., 2019] show this also can be decoupled. We choose to separate the two procedures as it enables us to evaluate various algorithms on the same supernet. We train the supernet by applying a random mask at each iteration. After training the supernet, we fix the parameters, then evaluate various search methods with the supernet. For example, a search algorithm samples an architecture, masking the supernet to assess the architecture, e.g., in Fig. 2.7; then the performance of the architecture is estimated from the performance of masked supernet on the test dataset.

Part II

Sample-Efficient Neural Architecture Search using Monte Carlo Tree Search

Chapter 3

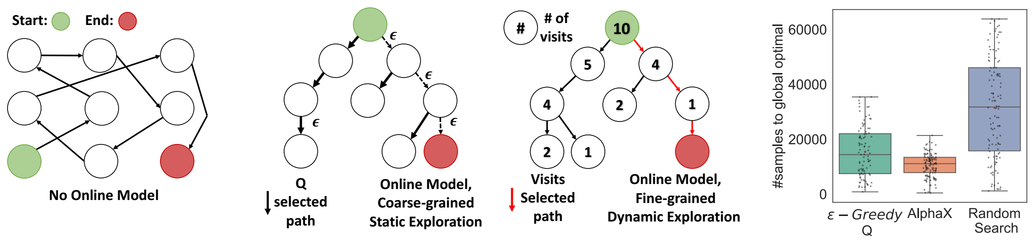
Building the NAS agent using MCTS

3.1 Motivation

Designing efficient neural architectures is extremely laborious. A typical design iteration starts with a heuristic design hypothesis from domain experts, followed by the design validation with hours of GPU training. The entire design process requires many of such iterations before finding a satisfying architecture. Neural Architecture Search has emerged as a promising tool to alleviate human effort in this trial and error design process, but the tremendous computing resources required by current NAS methods motivate us to investigate both the search efficiency and the network evaluation cost.

AlphaGo/AlphaGoZero [Silver et al., 2016, Tian et al., 2019a] have recently shown super-human performance in playing the game of Go, by using a specific search algorithm called Monte-Carlo Tree Search (MCTS). Given the current game state, MCTS gradually builds an online model for its subsequent game states to evaluate the winning chance at that state, based on search experiences in the current and prior games, and makes a decision. The search experience is from the previous search trajectories (called rollouts) that have been tried, and their consequences (whether the player wins or not). Different from the traditional MCTS approach that evaluates the consequence of a trajectory by random self-play to the end of a game, AlphaGo uses a predictive model (or value network) to predict the consequence, which enjoys much lower variance. Furthermore, due to its built-in exploration mechanism using Upper Confidence bound applied to Trees (UCT) [Kocsis and Szepesvári, 2006], based on its online model, MCTS dynamically adapts itself to the most promising search region, where good consequences are likely to happen.

Inspired by this idea, we build AlphaX, a NAS agent that uses MCTS for efficient architecture search



(a) random search (b) greedy methods (c) MCTS (d) Performance

Figure 3.1: Comparisons of NAS algorithms: (a) *random search* makes independent decision without using prior rollouts (previous search trajectories). An online model is to evaluate how promising the current search branch based on prior rollouts, and *random search* has no online model. (b) Search methods guided by online performance models built from previous rollouts. With static, coarse-grained exploration strategy (e.g., ϵ-greedy in *Q-learning*), they may quickly be stuck in a sub-optimal solution; and the chance to escape is exponentially decreasing along the trajectory. (c) AlphaX builds online models of both performance and visitation counts for adaptive exploration. The numbers in nodes represent values. (d) Performance of different search algorithms on NASBench-101. AlphaX is 3x, 1.5x more sample-efficient than *random search* and *ϵ-greedy* based *Q-learning*.

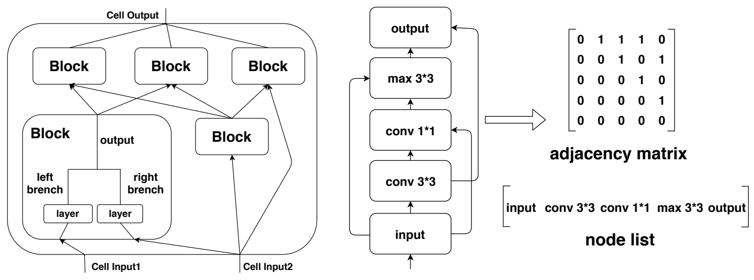
with Meta-DNN as a predictive model to estimate the accuracy of a sampled architecture. Compared with Random Search, AlphaX builds an online model which guides the future search; compared to greedy methods, e.g. Q-learning, Regularized Evolution or Top-K methods, AlphaX dynamically trades off exploration and exploitation and can escape from locally optimal solutions with fewer search trials. Fig. 3.1 summarizes the trade-offs. Toward a practical MCTS-based NAS agent, AlphaX has two novel features: first, a highly accurate multi-stage meta-DNN to improve the sample efficiency; and second, the use of transfer learning, together with a scalable distributed design, to amortize the network evaluation costs. As a result, AlphaX is the first MCTS-based NAS agent that reports the competitive performance.

3.2 AlphaX: A Scalable MCTS Design Agent

Please note the focus of this chapter is to describe our first MCTS based NAS agent, therefore you may find the content is engineering focused accompanied with lots of implementation details.

3.2.1 Design, State and Action Space

Design Space: the neural architectures for different domain tasks, e.g. object detection and image classification, follow fundamentally different designs. This renders different design spaces for the design agent. AlphaX is flexible to support various search spaces with an intuitive state and action abstraction. Here we



(a) NASNet Cell (b) NASBench DAG

Figure 3.2: Design space: (a) the cell structure of NASNet and (b) the DAG structure of NASBench-101. Then the network is constructed by stacking multiple cells or DAGs.

provide a brief description of two search spaces used in our experiments.

- *NASNet Search Space*: [Zoph et al., 2017] proposes searching a hierarchical Cell structure as shown in Fig.3.2. There are two types of Cells, Normal Cell ($NCell$) and Reduction Cell ($RCell$). $NCell$ maintains the input and output dimensions with the padding, while $RCell$ reduces the height and width by half with the striding. Then, the network is constituted by stacking multiple cells.

- *NASBench Search Space*: [Ying et al., 2019] proposes searching a small Direct Acyclic Graph (DAG) with each node representing a layer and each edge representing the inter-layer dependency as shown in Fig.3.2b. Similarly, the network is constituted by stacking multiple such DAGs.

State Space: a state represents a network architecture, and AlphaX utilizes states (or nodes) to keep track of past trails to inform future decisions. We implement a state as a map that defines all the hyper-parameters for each network layer and their dependencies. We also introduce a special terminal state to allow for multiple actions. All the other states can transit to the terminal state by taking the terminal action, and the agent only trains the network, from which it reaches the terminal. With the terminal state, the agent freely modifies the architecture before reaching the terminal. This enables multiple actions for the design agent to bypass shallow architectures.

Action Space: an action morphs the current network architecture, i.e. current state, to transit to the next state. It not only explicitly specifies the inter-layer connectivity, but also all the necessary hyper-parameters for each layer. Unlike games, actions in NAS are dynamically changing w.r.t the current state and design spaces. For example, AlphaX needs to leverage the current DAG (state) in enumerating all the feasible actions of 'adding an edge'. In our experiments, the actions for the NASNet search domain are adding a new layer in the left or right branch of a Block in a cell, creating a new block with different input combinations,

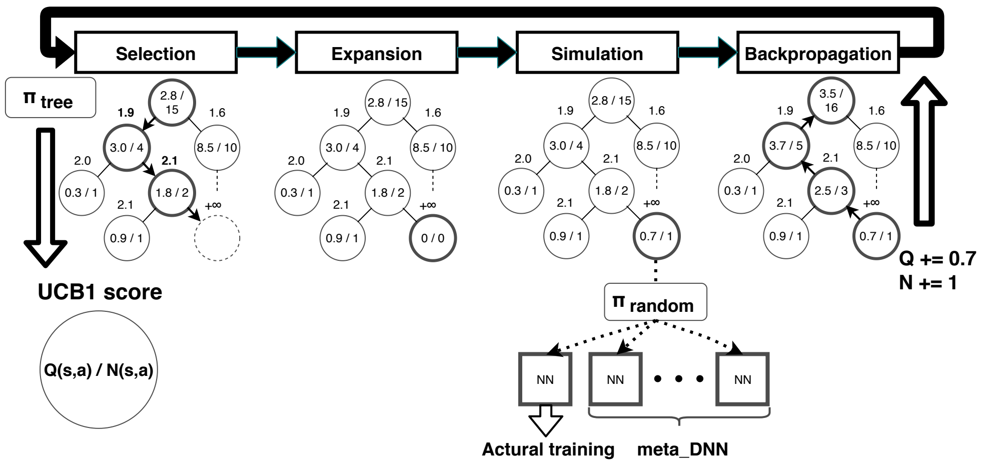


Figure 3.3: An overview of AlphaX search procedures, please see details in sec. 3.2.2.

and the terminating action. The actions for the NASBench search domain are either adding a node or an edge, and the terminating action.

3.2.2 Search Procedure

This section elaborates the integration of MCTS and metaDNN. The purpose of MCTS is to analyze the most promising move at a state, while the purpose of meta-DNN is to learn the sampled architecture performance and to generalize to unexplored architectures so that MCTS can simulate many rollouts with only an actual training in evaluating a new node. The superior search efficiency of AlphaX is due to balancing the exploration and exploitation at the finest granularity, i.e. state level, by leveraging the visiting statistics. Each node tracks these two statistics: 1) $N(s,a)$ counts the selection of action a at state s; 2) $Q(s,a)$ is the expected reward after taking action a at state s, and intuitively $Q(s,a)$ is an estimate of how promising this search direction is. Fig.3.3 demonstrates a typical searching iteration in AlphaX, which consists of Selection, Expansion, Meta-DNN assisted Simulation, and Backpropagation. We elucidate each step as follows.

Selection traverses down the search tree to trace the current most promising search path. It starts from the root and stops till reaching a leaf. At a node, the agent selects actions based on UCB1 [Auer et al., 2002]:

$$\pi_{tree}(s) = \arg\max_{a \in A} \left(\frac{Q(s,a)}{N(s,a)} + 2c\sqrt{\frac{2\log N(s)}{N(s,a)}} \right), \tag{3.1}$$

where $N(s)$ is the number of visits to the state s (i.e. $N(s) = \sum_{a \in A} N(s,a)$), and c is a constant. The first term ($\frac{Q(s,a)}{N(s,a)}$) is the exploitation term estimating the expected accuracy of its descendants. The second term ($2c\sqrt{\frac{2\log N(s)}{N(s,a)}}$) is the exploration term encouraging less visited nodes. The exploration term dominates $\pi_{tree}(s)$ if $N(s,a)$ is small, and the exploitation term otherwise. As a result, the agent favors the exploration in the beginning until building proper confidences to exploit. c controls the weight of exploration, and it is empirically set to 0.5. We iterate the tree policy to reach a new node.

Expansion adds a new node into the tree. $Q(s,a)$ and $N(s,a)$ are initialized to zeros. $Q(s,a)$ will be updated in the simulation step.

Meta-DNN assisted Simulation randomly samples the descendants of a new node to approximate $Q(s,a)$ of the node with their accuracies. The process is to estimate how promising the search direction rendered by the new node and its descendants. The simulation starts at the new node. The agent traverses down the tree by taking the uniform-random action until reaching a terminal state, then it dispatches the architecture for training. The more simulation we roll, the more accurate estimate of this search direction we get. However, we cannot conduct many simulations as network training is extremely time-consuming. AlphaX adopts a novel hybrid strategy to solve this issue by incorporating a meta-DNN to predict the network accuracy in addition to the actual training. We delay the introduction of meta-DNN to sec.3.2.3. Specifically, we estimate $q = Q(s,a)$ with

$$Q(s,a) \leftarrow \left(Acc(sim_0(s')) + \frac{1}{k}\sum_{i=1..k} Pred(sim_i(s')) \right)/2 \tag{3.2}$$

where $s' = s + a$, and $sim(s')$ represents a simulation starting from state s'. Acc is the actually trained accuracy in the first simulation, and $Pred$ is the predicted accuracy from Meta-DNN in subsequent k simulations. If a search branch renders architectures similar to previously trained good ones, Meta-DNN updates the exploitation term in Eq.4.3 to increase the likelihood of going to this branch.

Backpropagation back-tracks the search path from the new node to the root to update visiting statistics. Please note we discuss the sequential case here, and the backpropagation will be split into two parts in the distributed setting. With the estimated q for the new node, we iteratively back-propagate the information to its ancestral as:

$$\begin{aligned} &Q(s,a) \leftarrow Q(s,a) + q, \quad N(s,a) \leftarrow N(s,a) + 1 \\ &s \leftarrow parent(s), \quad a \leftarrow \pi_{tree}(s) \end{aligned} \tag{3.3}$$

until it reaches the root node.

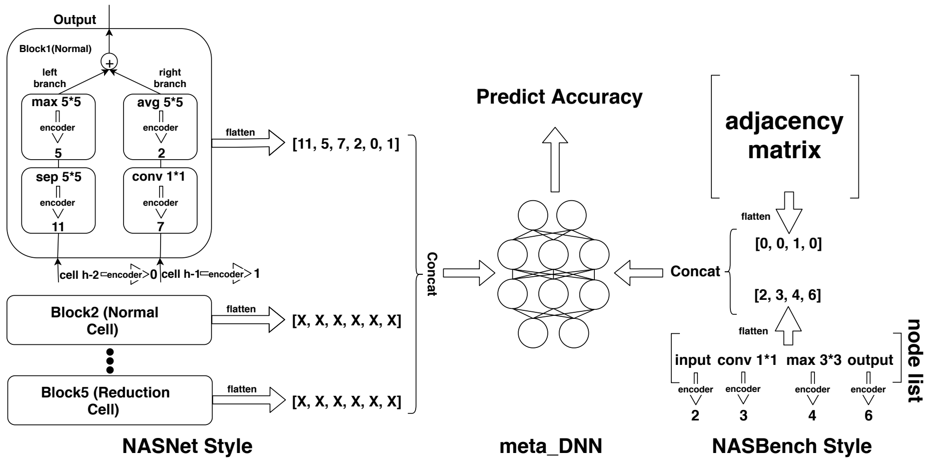


Figure 3.4: Encoding scheme of NASBench and NASNet.

3.2.3 The design of Meta-DNN and its related issues

Meta-DNN intends to generalize the performance of unseen architectures based on previously sampled networks. It provides a practical solution to accurately estimate a search branch with many simulations without involving the actual training (see the metaDNN assisted simulation for details). New training data is generated as AlphaX advances in the search. So, the learning of Meta-DNN is end-to-end. The input of Meta-DNN is a vector representation of architecture, while the output is the prediction of architecture performance, i.e. test accuracy.

The coding scheme for NASNet architectures is as follows: we use 6-digits vector to code a $Block$; the first two digits represent up to two layers in the left branch, and the 3rd and 4th digits for the right branch. Each layer is represented by a number in $[1, 12]$ to represent 12 different layers. We use 0 to pad the vector if a layer is absent. The last two digits represent the input for the left and right branch, respectively. For the coding of block inputs, 0 corresponds to the output of the previous $Cell$, 1 is the previous, previous $Cell$, and $i + 2$ is the output of $Block_i$. If a block is absent, it is [0,0,0,0,0,0]. The left part of Fig.3.4 demonstrates an example of NASNet encoding scheme. A $Cell$ has up to 5 blocks, so a vector of 60 digits is sufficient to represent a state that fully specifies both $RCell$ and $NCell$. The coding scheme for NASBench architectures is a vector of flat adjacency matrix, plus the nodelist. Similarly, we pad 0 if a layer or an edge is absent. The right part of Fig.3.4 demonstrates an example of NASBench encoding scheme. Since NASBench limits nodes ≤ 7, 7×7 (adjacency matrix)+ 7 (nodelist) = 56 digits can fully specify a NASBench architecture.

Now we cast the prediction of architecture performance as a regression problem. Finding a good metaDNN is heuristically oriented and it should vary from tasks to tasks. We calculate the correlation between predicted accuracies and true accuracies from the sampled architectures in evaluating the design of metaDNN. Ideally, the metaDNN is expected to rank an unseen architecture in roughly similar to its true test accuracy, i.e. corr = 1. Various ML models, such as Gaussian Process, Neural Networks, or Decision Tree, are candidates for this regression task. We choose Neural Networks as the backbone model for its powerful generalization on the high-dimensional data and the online training capability. More ablations studies for the specific choices of metaDNN are available in sec.3.3.2.

3.2.4 Transfer Learning

As MCTS incrementally builds a network with primitive actions, networks falling on the same search path render similar structures. This motivates us to incorporate transfer learning in AlphaX to speed network evaluations up. In simulation (Fig. 3.3), AlphaX recursively traverses up the tree to find a previously trained network with the minimal edit distance to the newly sampled network. Then we transfer the weights of overlapping layers, and randomly initialize new layers. In the pre-training, we train every sample for 70 epochs if no parent networks are transferable, and 20 epochs otherwise. Fig. 3.8 provides a study to justify the design.

3.2.5 Distributed AlphaX

It is imperative to parallelize AlphaX to work on a large scale distributed systems to tackle the computation challenges rendered by NAS. Fig.3.5 demonstrates the distributed AlphaX. There is a master node exclusively for scheduling the search, while there are multiple clients (GPU) exclusively for training networks. The general procedures on the server side are as follows: 1) The agent follows the selection and expansion steps described in Fig.3.3. 2) The simulation in MCTS picks a network $arch_n$ for the actual training, and the agent traverses back to find the weights of parent architecture having the minimal edit distance to $arch_n$ for transfer learning; then we push both $arch_n$ and parent weights into a job queue. We define $arch_n$ as the selected network architecture at iteration n, and $rollout\_from(arch_n)$ as the node which it started the rollout from to reach $arch_n$. 3) The agent *preemptively backpropagates* $\hat{q} \leftarrow \frac{1}{k}\sum_{i=1..k} Pred(sim_i(s'))$ based only on predicted accuracies from the Meta-DNN at iteration n.

$$\begin{aligned} &Q(s,a) \leftarrow Q(s,a) + \hat{q}, \quad N(s,a) \leftarrow N(s,a) + 1, \\ &s \leftarrow parent(s), \quad a \leftarrow \pi_{tree}(s). \end{aligned} \tag{3.4}$$

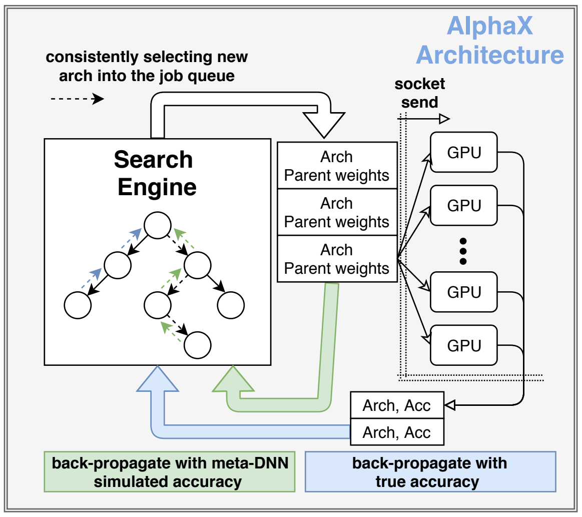

AlphaX Architecture
consistently selecting new arch into the job queue
socket send
Search Engine
Arch Parent weights
Arch Parent weights
Arch Parent weights
GPU
GPU
GPU
GPU
Arch, Acc
Arch, Acc
back-propagate with meta-DNN simulated accuracy
back-propagate with true accuracy

Figure 3.5: Distributed AlphaX: we decouple the original back-propagation into two parts: one uses predicted accuracy (green arrow), while the other uses the true accuracy (blue arrow). The pseudocode for the whole system is available in Sec.3.4

4) The server checks the receive buffer to retrieve a finished job from clients that includes $arch_z$, acc_z. Then the agent starts the second backpropagation to propagate $q \leftarrow \frac{acc_z + \hat{q}}{2}$ (Eq. 3.2) from the node the rollout started ($s \leftarrow rollout\_from(arch_z)$) to replace the backpropagated $\hat{q}$ with q:

$$\begin{aligned} Q(s,a) &\leftarrow Q(s,a) + q - \hat{q}, \\ s \leftarrow parent(s), &\quad a \leftarrow \pi_{tree}(s). \end{aligned} \tag{3.5}$$

The client constantly tries to retrieve a job from the master job queue if it is free. It starts training once it gets the job, then it transmits the finished job back to the server. So, each client is a dedicated trainer. We also consider the fault-tolerance by taking a snapshot of the server's states every few iterations, and AlphaX can resume the searching from the breakpoint using the latest snapshot.

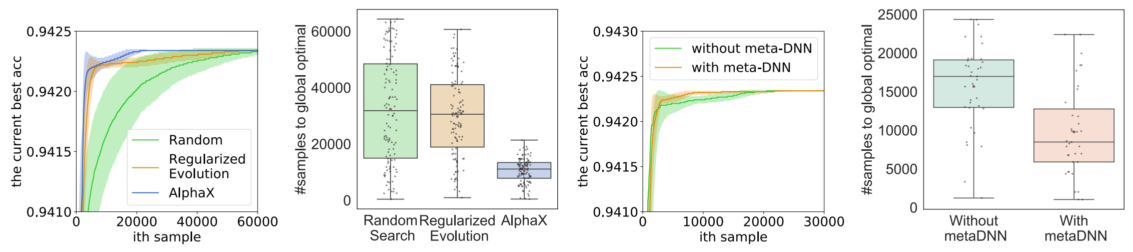
3.3 Experiments

3.3.1 Evaluations of architecture search

First, we perform the search on the NASNet search space using 8 NVIDIA 1080 TI. One GPU works as a server, while the rest work as clients. To further speedup network evaluations, we early terminated the

Model	Params	Err	GPU days	M
NASNet-A+cutout [Zoph et al., 2017]	3.3M	2.65	2000	20000
AmoebaNet-B+cutout [Real et al., 2019a]	2.8M	$2.50_{\pm 0.05}$	3150	27000
DARTS+cutout [Liu et al., 2018b]	3.3M	$2.76_{\pm 0.09}$	4	-
AlphaX+cutout (32 filters)	2.83M	$2.54_{\pm 0.06}$	12	1000
PNAS [Liu et al., 2018a]	3.2M	$3.41_{\pm 0.09}$	225	1160
ENAS [Pham et al., 2018a]	4.6M	3.54	0.45	-
NAONet [Luo et al., 2018a]	10.6M	3.18	200	1000
AlphaX (32 filters)	2.83M	$3.04_{\pm 0.03}$	12	1000
NAS v3[Zoph and Le, 2016]	7.1M	4.47	22400	12800
Hier-EA [Liu et al., 2017]	15.7M	$3.75_{\pm 0.12}$	300	7000
AlphaX+cutout (128 filters)	31.36M	$2.16_{\pm 0.04}$	12	1000

Table 3.1: The comparisons of our NASNet search results to other state-of-the-art results on CIFAR-10. M is the number of sampled architectures in the search.



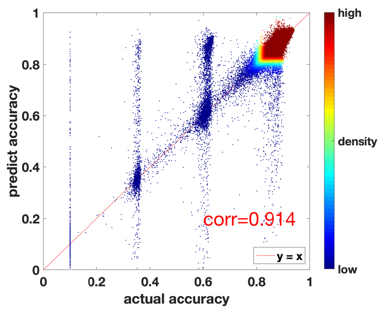
(a) best acc progression (b) #samples to the best (c) best acc progression (d) #samples to the best

Figure 3.6: Finding the global optimum on NASBench-101: AlphaX is 3x, 2.8x faster than Random Search and Regularized Evolution on NASBench-101 (nodes $\leq$ 6). The results are from 200 trails with different random seeds. (c) and (d) show the performance of AlphaX in cases of with/without meta-DNN on NASBench-101

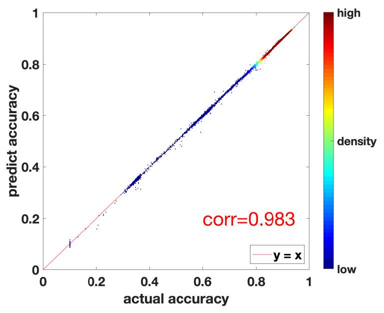
training at 70th epoch during the pre-training. We selected the top 20 networks from the pre-training and fine-tuned them additional 530 epochs to get the final accuracy. For the ImageNet training, we constructed the network with the same $RCell$ and $NCell$ searched on CIFAR10 following the accepted standard, i.e. the mobile setting, defined in [Zoph et al., 2017]. In total, AlphaX sampled 1000 networks.

To further examine the sample efficiency, we evaluate AlphaX on the recent NAS dataset, NASBench-101 [Ying et al., 2019]. NASBench enumerates all the possible DAGs of nodes $\leq$ 7, constituting of (420k+) networks and their final test accuracies. This enables bypassing the computation barrier to fairly evaluate the sample efficiency. In our experiments, we limited the maximal nodes in a DAG $\leq$ 6, i.e. constructing a subset of NASBench-101 that contains 64521 valid networks. This allows us to quickly repeat each algorithm for 200 trials. The search target is the network with the highest mean test accuracy (the global optimum) at 108th epochs, which can be known ahead by querying the dataset. We choose Random Search (RS) [Sciuto et al., 2019] and Regularized Evolution (RE) [Real et al., 2019a] as the baseline. RE delivers competitive results according to Table. 6.4 in the NASNet search space, and RS finds the global optimal in expected n/2, where n is the dataset size. Fig. 3.6 demonstrates AlphaX is 2.8x and 3x faster than RE and RS, respectively. As we analyzed in Fig. 3.1, Random Search lacks an online model. Regularized Evolution only mutates on top-k performing models, while MCTS explicitly builds a search tree to dynamically trade off the exploration and exploitation at individual states. Please note that the slight difference in Fig. 3.6a actually reflects a huge gap in speed as indicated by Fig. 3.6b. According to NASBench-101, there are abundant architectures with minor performance difference to the global optimum. Therefore, it is fast to find the top 5% architectures, but extremely hard in reaching the global optimum.

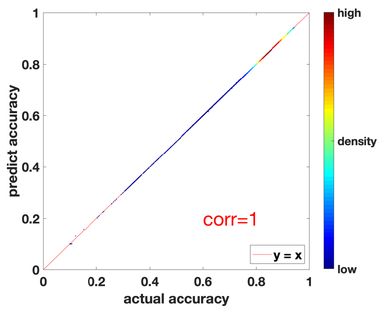
3.3.2 Component Evaluations



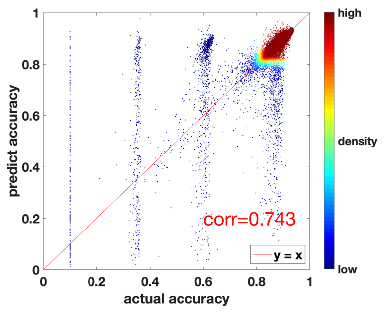
(a) RNN train



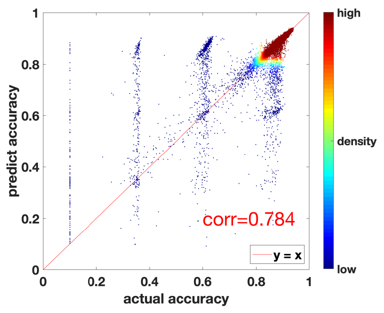
(b) MLP train



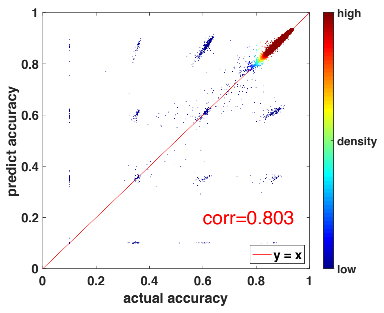
(c) multi-stage train



(d) RNN test



(e) MLP test



(f) multi-stage test

Figure 3.7: meta-DNN design ablations: True v.s. predicted accuracies of MLP, RNN and multi-stage MLP on architectures from NASBench. The scatter density is highlighted by color to reflect the data distribution; Red means high density, and blue otherwise.

First, we exam the impact of meta-DNN design. The metric in evaluating metaDNN is the correlation between the predicted v.s. true accuracy. We used 80% NASBench for training, and 20% for testing. Since DNNs have shown great success in modeling complex data, we start with Multilayer Perceptron (MLP) and

Recurrent Neural Network (RNN) on building the regression model. The hidden state size of LSTM is 100, and the embedding size is also 100. The final LSTM hidden state goes through a fully-connected layer to get the final validation accuracy. The multi-stage model is an ensemble of multiple MLP models. The structure of an MLP is 512→2048→2048→512→1. Fig. 3.7d and Fig .3.7e demonstrate the performance of MLP (corr=0.784) is 4% better than RNN (corr=0.743), as the MLP (Fig. 3.7b) performs much better than RNN (Fig. 3.7a) in the training set. However, MLP still mispredicts many networks around 0.1, 0.4 and 0.6 and 0.8 (x-axis) as shown in Fig. 3.7e. This clustering effect is consistent with the architecture distribution in Fig. **??** for having many networks around these accuracies. To alleviate this issue, we propose a multi-stage model, the core idea of which is to have several dedicated MLPs to predict different ranges of accuracies, e.g. [0, 25%], along with another MLP to predict which MLP to use in predicting the final accuracy. Fig. 3.7f shows a multi-stage model successfully improves the correlation by 1.2% from MLP, and the mispredictions have been greatly reduced. Since the multi-stage model has achieved corr = 1 on the training set, we choose it as the backbone regression model for AlphaX. Fig. 3.6 demonstrates our meta-DNN is effective to sustain NAS.

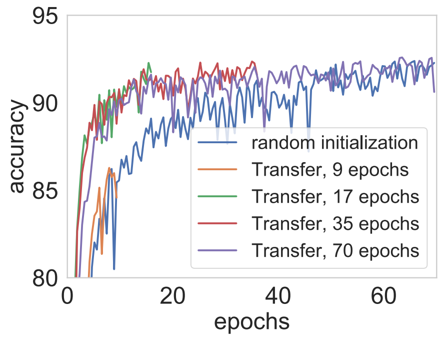


Figure 3.8: Validation of transfer learning: transferring weights significantly reduces the number of epochs in reaching the same accuracy of random initialization (Transfer 17 $\rightarrow$ 70 epochs v.s. random initialization), but insufficient epochs loses accuracy (Transfer, 9 epochs).

The transfer learning significantly speeds network evaluations up, and Fig. 3.8 empirically validates the effectiveness of transfer learning. We randomly sampled an architecture as the parent network. On the parent network, we added a block with two new 5x5 separable conv layers on the left and right branch as the child network. We trained the parent network toward 70 epochs and saved its weights. In training the child network, we used weights from the parent network in initializing the child network except for two new conv layers that are randomly initialized. Fig. 3.8 shows the accuracy progress of transferred child network at different training epochs. The transferred child network retains the same accuracy as training from scratch (random initialization) with much fewer epochs, but insufficient epochs lose the accuracy. Therefore, we chose 20 epochs in pre-training an architecture if transfer learning applied.

3.4 Additional Details

Here I present the pseudocode of distributed AlphaX. Algorithm 3 describes the search engine; Algorithm 2 describes the procedures of server that implement a client-server communication protocol to send an architecture searched by MCTS to a client for training. Algorithm 1 describes the procedures of client that consistently listen new architectures from the server for training, and send the accuracy and network back to the server after training.

Algorithm 1 Client

```
1: **Require:** Start working once building connection to the server
2: while True do
3:    if The client is connected to server then
4:       $network \leftarrow$ Receive()
5:       $accuracy \leftarrow$ Train($network$)
6:       Send ($network, accuracy$) to the Server
7:    else
8:       Wait for re-connection
9:    end if
10: end while=0
```

Algorithm 2 Server

```
1: while $size(TASK\_QUEUE) > 2$ do
2:    while no idle client do
3:       Continue          $\rightarrow$ Wait for dispatching jobs until there are idle clients
4:    end while
5:    Create a new connection to a random idle client
6:    $network \leftarrow TASK\_QUEUE$.pop()
7:    Send $network$ to a Client
8:    if Received_Signal() then
9:       $network, accuracy \leftarrow$ Receive_Result()
10:      $acc(network) \leftarrow accuracy$
11:      $state \leftarrow rollout\_from(network)$
12:      Backpropagation($state$, $(accuracy - \hat{q}(state))/2$, 0)
13:      Train the meta-DNN with a new data $(network, accuracy)$
14:   else
15:      Continue
16:   end if
17: end while=0
```

Algorithm 3 Search Engine (MCTS)

```
1:  function Expansion($state$)
2:      Create a new node in a tree for state.
3:      for all $action$ available at $state$ do
4:      $Q(state, action) \leftarrow 0, \quad N(state, action) \leftarrow 0$
5:      end for
6:  end function
7:
8:  function Simulation($state$)
9:  $action \leftarrow none$
10:     while $action\ is\ not\ term$ do
11:         randomly generate an $action$
12:         $next\_net \leftarrow$ Apply$(state, action)$ {Apply returns the next state when $action$ is applied to $state$}
13:     end while
14: end function
15:
16: function Backpropagation($state$, $q$, $n$)
17:     while $state\ is\ not\ root$ do
18:         $state \leftarrow$ parent$(state)$
19:         $Q(state, action) \leftarrow Q(state, action) + q$
20:         $N(state, action) \leftarrow N(state, action) + n$
21:     end while
22: end function
23:
24: Require: Start from the root
25: while $episode < MAX\_episode$ do
26:     Server()
27:     $cur\_state \leftarrow root\_node$
28:     $i \leftarrow 0$
29:     while $i < MAX\_tree\_depth$ do
30:         $i \leftarrow i + 1$
31:         $next\_action \leftarrow$ Selection$(cur\_state)$
32:         if $next\_state$ not in tree then
33:             $next\_state \leftarrow$ Expansion$(next\_action)$
34:             $T_t \leftarrow$ Simulation$_t(next\_state)$ for $t = 0...k$
35:             $TASK\_QUEUE$.push$(T_0)$
36:             $rollout\_from(T_0) \leftarrow next\_state$
37:             $\hat{q}(next\_state) \leftarrow \frac{1}{k}\sum_{i=1..k} Pred(T_i)$
38:             Backpropagation$(next\_state, \hat{q})$
39:         end if
40:     end while
41: end while=0
```

3.5 Conclusion

The focus of this section is to describe the implementation details of AlphaXm, our first MCTS based NAS agent. Compared to prior works, AlphaX is the first practical MCTS based agent that achieves competitive results on CIFAR-10 in a reasonable amount of computations using a predictive model. To harness many GPUs, we also describe the distributed architect of AlphaX. In the next chapter, and we will present a new MCTS algorithm that learns the latent action to improve the sample efficiency.

Chapter 4

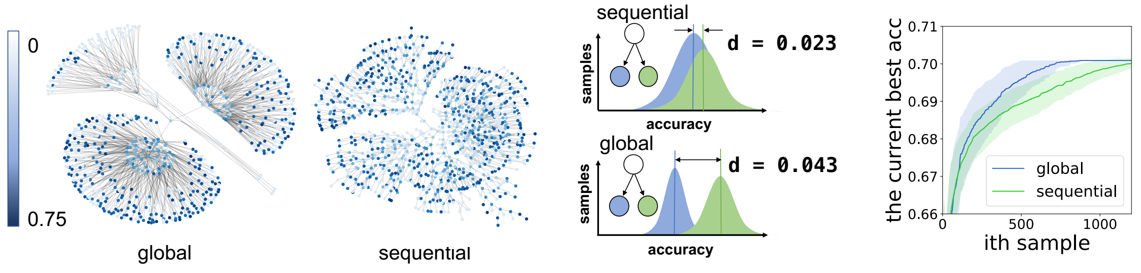
Sample-Efficient NAS by Learning Actions for MCTS

4.1 Motivation

While NAS is impressive to find a good network architecture in a large search space, one component that is often overlooked is how to design the action space. Existing MCTS based NAS approaches utilize manually designed action space, which is not directly related to the performance metric to be optimized (e.g., accuracy), leading to sample-inefficient explorations of architectures. To demonstrate the importance of action space in MCTS, we start with a motivating example. Consider a simple scenario of designing a plain Convolutional Neural Network (CNN) for CIFAR-10 image classification. The primitive operation is a Conv-ReLU layer. Free structural parameters that can vary include network depth $L = \{1, 2, 3, 4, 5\}$, number of filter channels $C = \{32, 64\}$ and kernel size $K = \{3 \times 3, 5 \times 5\}$. This configuration results in a search space of 1,364 networks. To perform the search, there are two natural choices of the action space: `sequential` and `global`. `sequential` comprises actions in the following order: adding a layer l, setting kernel size K_l, setting filter channel C_l. The actions are repeated L times. On the other hand, `global` uses the following actions instead: {Setting network depth L, setting kernel size $K_{1,...,L}$, setting filter channel $C_{1,...,L}$}. For these two action spaces, MCTS is employed to perform the search. Note that both action spaces can cover the entire search space but have very different search trajectories.

Fig. 4.1(a) visualizes the search for these two action spaces. Actions in `global` clearly separates

Can we learn to partition Ω into good/bad regions, as the case in global, from samples to boost the search efficiency?



(a) Visualization of MCTS search trees (b) Reward distribution at two child nodes (c) Search efficiency

Figure 4.1: Illustration of motivation: (a) visualizes the MCTS search trees using `sequential` and `global` action space. The node value (*i.e.* accuracy) is higher if the color is darker. (b) For a given node, the reward distributions for its children. d is the average distance over all nodes. `global` better separates the search space by network quality and provides distinctive rewards in recognizing a promising path. (c) As a result, `global` finds the best network much faster than `sequential`. This motivates us to learn actions to partition the search space for the efficient architecture search.

desired and undesired network clusters, while actions in `sequential` lead to network clusters with a mixture of good or bad networks in terms of performance. As a result, the accuracy distribution of two branches (Fig. 6.1(b)) are separable for `global`, which is not the case for `sequential`. We also demonstrate the overall search performance in Fig.6.1(c) that `global` finds desired networks much faster than `sequential`.

This observation suggests that changing the action space can lead to very different search behavior and thus potentially better sample efficiency. In this case, an early exploration of network depth is critical. Increasing depth is an optimization direction that can potentially lead to better model accuracy. One might come across a natural question from this motivating example: is it possible to find a principle way to distinguish a good action space from a bad action space for MCTS? Is it possible to *learn an action space* such that it can best fit the performance metric to be optimized?

4.2 Learning Latent Actions for MCTS

Based on the above observations, we propose LaNAS that learns *latent actions* and prioritizes the search accordingly. To achieve this goal, LaNAS iterates between *learning* and *searching* stage. In the learning stage, LaNAS models each action as a *linear constraint* that bi-partitions the search space Ω into high-performing and low-performing regions. Such partitions can be done recursively, yielding a hierarchical tree structure, where some leaf nodes contain very promising regions. In the searching stage, LaNAS applies

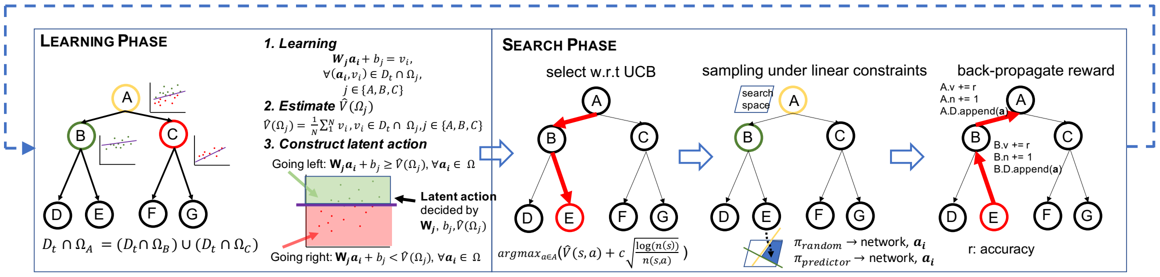

LEARNING PHASE
1. Learning
$W_j a_i + b_j = v_i$, $\forall (a_i, v_i) \in D_t \cap \Omega_j$, $j \in \{A, B, C\}$
2. Estimate $\hat{V}(\Omega_j)$
$\hat{V}(\Omega_j) = \frac{1}{N}\sum_1^N v_i, v_i \in D_t \cap \Omega_j, j \in \{A, B, C\}$
3. Construct latent action
Going left: $W_j a_i + b_j \geq \hat{V}(\Omega_j), \forall a_i \in \Omega$
Latent action decided by $W_j, b_j, \hat{V}(\Omega_j)$
Going right: $W_j a_i + b_j < \hat{V}(\Omega_j), \forall a_i \in \Omega$
$D_t \cap \Omega_A = (D_t \cap \Omega_B) \cup (D_t \cap \Omega_C)$
SEARCH PHASE
select w.r.t UCB
sampling under linear constraints
back-propagate reward
search space
A.v += r
A.n += 1
A.D.append(a)
B.v += r
B.n += 1
B.D.append(a)
$argmax_{a \in A}(\hat{V}(s,a) + c\sqrt{\frac{\log(n(s))}{n(s,a)}})$
π_{random} → network, a_i
$\pi_{predictor}$ → network, a_i
r: accuracy
A
B
C
D
E
F
G

Figure 4.2: An overview of LaNAS: Each iteration of LaNAS comprises a search and learning phase. The search phase uses MCTS to sample networks, while the learning phase learns a linear model between network hyper-parameters and accuracies.

Monte Carlo Tree Search (MCTS) on the tree structure to sample architectures. The learned actions provide an abstraction of search space for MCTS to do an efficient search, while MCTS collects more data with adaptive exploration to progressively refine the learned actions for partitioning. The iterative process is jump-started by first collecting a few random samples.

The following defines a list of notations used in this paper: $\mathbf{a}_i$ represents the ith sampled architecture; v_i is the performance metric of $\mathbf{a}_i$; D_t is the set of collected $(\mathbf{a}_i, v_i)$ at the search step t; Ω represents the entire search space; Ω_j represents the partition of Ω at the tree node j; $D_t \cap \Omega_j$ is the samples classified in Ω_j; $V(\Omega_j)$ is the mean performance metric in Ω_j; $\hat{V}(\Omega_j)$ is the estimated $V(\Omega_j)$ from $D_t \cap \Omega_j$; $f_j(\mathbf{a}_i)$ is the predicted performance by the regressor on node j; $n(s)$ represents the #visits of tree node s; $v(s)$ is the value of tree node s.

4.2.1 Learning Phase

In the learning phase at iteration t, we have a dataset $D_t = \{(\mathbf{a}_i, v_i)\}$ obtained from previous explorations. Each data point $(\mathbf{a}_i, v_i)$ has two components: $\mathbf{a}_i$ represents an architecture in specific encoding (e.g., width=512 and depth=5, etc) and v_i represents the performance metric estimated from training, or from pre-trained dataset such as NASBench-101, or estimated from a supernet in one-shot NAS.

At any iteration, our goal is to learn a good action space from D_t that splits Ω so that the performance of architectures is similar within each partition, but across partitions, the architecture performance can be easily ranked from low to high based on partitions. This split can be recursively done to form a hierarchy; and our motivating example in Fig. 6.1 suggests such partitions can help prioritize the search towards more promising regions, and improve the sample efficiency. In particular, we model the recursive splitting process as a tree. The root node corresponds to the entire model space Ω, while each tree node j corresponds to a region Ω_j. At each tree node j, we partition Ω_j into two disjoint regions $\Omega_j = \cup_{k\in(good,bad)}\Omega_k$, such that $\hat{V}(\Omega_{good}) > \hat{V}(\Omega_{bad})$ on each nodes. Therefore, a tree of these nodes recursively partitions the entire search space into different performance regions to achieve the target behavior. The following illustrates the algorithms in detail.

At each node j, we learn a regressor that embodies a latent action to split the model space Ω_j. The linear regressor takes the portion of the dataset that falls into its own region $D_t \cap \Omega_j$, then the average performance of a region is estimated by

$$\hat{V}(\Omega_j) = \frac{1}{N} \sum_{v_i \in D_t \cap \Omega_j} v_i \tag{4.1}$$

To partition Ω_j into Ω_{good} and Ω_{bad}, we learn a linear regressor f_j

$$\underset{(\mathbf{a}_i, v_i)\in D_t \cap \Omega_j}{\text{minimize}} \sum (f_j(\mathbf{a}_i) - v_i)^2 \tag{4.2}$$

Once learned, the parameters of f_j and $\hat{V}(\Omega_j)$ form a linear constraint that bifurcates Ω_j into a good region ($> \hat{V}(\Omega_j)$) and a bad region ($\leq \hat{V}(\Omega_j)$). A visualization of this process is available in Fig. 4.2 (learning phase). For convenience, the left child always represents the good region. The partition threshold $\hat{V}(\Omega_j)$, combined with parameters of f_j, forms two latent actions at node j,

$$\begin{aligned} \textbf{go-left} &: f_j(\mathbf{a}_i) > \hat{V}(\Omega_j) \\ \textbf{go-right} &: f_j(\mathbf{a}_i) \leq \hat{V}(\Omega_j), \forall \mathbf{a}_i \in \Omega \end{aligned}$$

For simplicity, we use a full tree to initialize the search algorithm, leaving the tree height as a hyper-parameter. Fig. 4.6 provides guidance in selecting the tree height. Because the tree recursively splits Ω, partitions represented by leaves follow $V(\Omega_{leftmost}) > ... > V(\Omega_{rightmost})$, with the leftmost leaf representing the most promising partition. Experiments in Sec. 4.3.2 validate the effectiveness of the proposed method in achieving the target behavior.

Note that we need to initialize each node classifier properly with a few random samples to establish an initial boundary in the search space. An ablation study on the number of samples for initialization is provided in Fig. 4.6(c).

4.2.2 Search Phase

Once actions are learned, the search phase follows. The search uses learned actions to sample more architectures $\mathbf{a}_i$, and get v_i either via training or predicted from a supernet, then store $(\mathbf{a}_i, v_i)$ in dataset D_t to refine the action space in the next iteration. Note that during the search phase, the tree structure and the parameters of those classifiers are fixed and static. The search phase decides which region Ω_j on tree leaves to sample.

Given existing samples, a trivial go-left strategy, i.e. greedy-based search, can be used to exclusively exploit the most promising Ω_k. However, the search space partitions or the latent actions learned from current samples can be sub-optimal such that the best model is located on any non-leftmost tree leaves. There can be good model regions that are hidden in the right (or bad) leaves that need to be explored.

To avoid this issue in a pure go-left search strategy, we integrate Monte Carlo Tree Search (MCTS) into the proposed search tree to enable adaptive explorations of different leaves. Besides, MCTS has shown great success in high dimensional tasks, such as Go [Tian et al., 2019a] and NAS [Wang et al., 2019c]. MCTS avoids trapping into a local optimum by tracking both the number of visits and the average value on each node. For example, MCTS will choose the right node with a lower value if the left node with a higher value has been frequently visited before. More details are available in the paragraph of select w.r.t UCT below.

Like MCTS, our search phase also has *select*, *sampling* and *backpropagate* stages. LaNAS skips the *expansion* stage in regular MCTS since we use a static tree. At each iterations, previously sampled networks and their performance metrics in D_t are reused and redirected to (maybe different) nodes, when initializing visitation counts $n(s)$ and node values $v(s)$ for the tree with updated action space. The details of these 3 steps are as follows.

1) ***select w.r.t UCB***: UCB [Auer et al., 2002] is defined by

$$\pi_{UCB}(s) = \arg\max_{a \in A} \left(\hat{V}(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \right), \tag{4.3}$$

π_{UCB} chooses the action that yields the largest UCB score. In our case, the action is either going left or right on any non-leaf nodes. At the node s, $\hat{V}(s,a)$ represents the estimated value of the child node by taking action a, e.g. the value of left child by going left. $n(s)$ is the number of visits of the node s, which corresponds to the number of samples falling on the node represented partition. Similarly $n(s,a)$ represents the number of visits of the next node. Starting from $root$, we follow π_{UCB} to traverse down to a leaf.

In π_{UCB}, the first term of $\hat{V}(s,a)$ represents the average value of next node after taking the action a at the node s. By the construction of search tree, the average value of left child node is higher than the right. Therefore, π_{UCB} degenerates to a pure go-left strategy if we set $c = 0$. The second term of $\log n(s)/n(s, a)$ represents the exploration, and c is a hyper-parameter. $n(s)$ is same regardless of the action taken at node s, but the number of visits on the next node $n(s, a)$ can be drastically different. So a less visiting node with smaller $n(s, a)$ can increase π_{UCB}. Therefore, π_{UCB} favors the node without any samples (dominated by the exploration term), or the node value is significantly higher (dominated by the exploitation term). Because of this mechanism, our algorithm can jump out of a sub-optimal action space learnt from current samples.

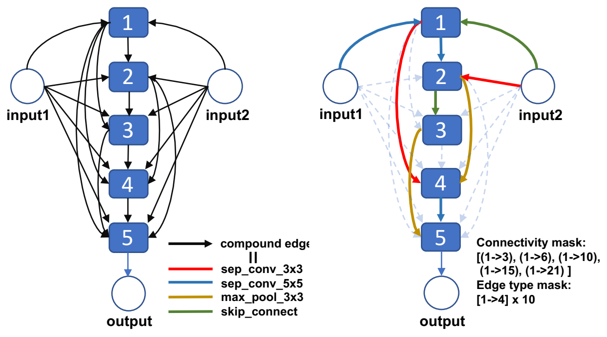
2) ***sampling from a leaf***: *select* traverses a path from the root to a leaf, which defines a set of linear constrains for sampling. A node j defines a constraint l_j of $f_j(\mathbf{a}_i) \geq \hat{V}(\Omega_j), \forall \mathbf{a}_i \in \Omega$ if the path chooses the left child, and $f_j(\mathbf{a}_i) < \hat{V}(\Omega_j)$ otherwise. Therefore, the constraints from a path collectively enclose a partition Ω_j for proposing the new samples. Fig. **??** visualizes the process of partitioning along a search path.

Within a partition Ω_j, a simple search policy is to use reject sampling: random sample until it satisfies the constraints. This is efficient thanks to limited numbers of constraints [Gilks et al., 1995, Hörmann, 1995]. Other strategies, e.g. Bayesian optimizations, can also be applied to sample from Ω_j. Here we illustrate the implementation of both π_{bayes} and π_{random}.

- *Random search based* π_{rand}: each component in the architecture encoding is assigned to a uniform random variable, and the vector of these random variables correspond to random architectures. For example, NASBench uses 21 Boolean variables for the adjacent matrix and 5 3-values categorical variables for the node list. The random generator uses 26 random integer variables, with 21 variables to be uniformly distributed in the set of [0, 1] indicating the existence of an edge, and 5 variables to be uniformly distributed in the set of [0, 1, 2] indicating layer types. π_{rand} outputs a random architecture as long as it satisfies the path constraints.

- *Bayesian search based* π_{bayes}: a typical Bayesian optimization search step consists of 2 parts: training a surrogate model using a Gaussian Process Regressor (GPR) on collected samples D_t; and proposing new samples by optimizing the acquisition function, such as Expected Improvement (EI) or Upper Confidence Bound (UCB). However, GPR is not scalable to large samples; and we used meta-DNN in [Wang et al., 2019c] to replace GPR. Training the surrogate remains unchanged, but we only compute EI for architectures in the selected partition Ω_j, and returns $\mathbf{a}_i$ with the maximum EI.

The comparisons of π_{bayes} to π_{random} is in Fig. 4.5. For the rest of the paper, we use π_{random} in LaNAS for simplicity.



(a) supernet (b) masked supernet

Figure 4.3: The cell structure of supernet used in searching nasnet. The supernet structure of normal and reduction cell are same. (a) Each edge is a compound edge, consisting of 4 independent edges with the same input/output to represent 4 layer types. (b) Each node allows for two inputs from previous nodes. To specify a NASNet architecture, we use 5 variables for defining connectivity among nodes, and 10 variables for defining the layer type of every edge. Supernet can transform to any network in the search space by applying the mask.

3) ***back-propagate reward***: after evaluating the sampled network, LaNAS back-propagates the reward, i.e. accuracy, to update the node statistics $n(s)$ and $v(s)$. It also back-propagates the sampled network so that every parent node j keeps the network in $D_t \cap \Omega_j$ for training.

There are multiple ways to evaluate the performance of architecture, such as training from scratch, or predicted from a supernet as the case in one-shot NAS [Pham et al., 2018a]. Among these methods, training every architecture from scratch (re-training) gives the most accurate v_i but is extremely costly. While one-shot NAS is fairly cheap to execute as it only requires one-time training of a supernet for predicting v_i for $\forall \mathbf{a}_i \in \Omega$, the predicted v_i is quite inaccurate [Sciuto et al., 2019]; therefore the architecture found by one-shot NAS is generally worse than the re-training approaches as indicated in Table. 9.1. In this paper, we try both one-shot based and training based evaluations. The integration of one-shot NAS is described as follows.

4.2.3 Integrating with one-shot NAS

The key bottleneck in NAS is the expensive evaluation that trains a network from scratch. [Pham et al., 2018a] proposes a weight sharing scheme to avoid the model re-training using a supernet, which can transform to any architectures in the search space by deactivating extra edges to the network. One popular approach for one-shot NAS is to formulate the training of supernet and the search on supernet as an integrated bi-level optimizations [Liu et al., 2018b], while recent works [Guo et al., 2019a, Sciuto et al., 2019] show this also can be done by separating the training and the search. Our work primarily focuses on search efficiency, and we choose to separate the two procedures as it enables us to evaluate various algorithms on the same supernet (in Fig. 4.4(c)). We train the supernet by applying a random mask at each iteration following the same training pipeline/hyper-parameters in DARTs. After training the supernet, we fix the parameters of the supernet, then evaluate various search methods onto it. For example, LaNAS samples $\mathbf{a}_i$, masking the supernet to evaluate $\mathbf{a}_i$ as illustrated by Fig. 4.3; then v_i is the validation accuracy evaluated from the masked supernet. The result $(\mathbf{a}_i, v_i)$ is stored in D_t to guide the future search. The following elaborates the design of the supernet for the NASNet search space, its training/search details, and the masking process.

The design of supernet

We have used two designs of supernet in this paper: one is for NASNet search space [Zoph et al., 2018b] to be evaluated on CIFAR10, and the other is for EfficientNet search space [Tan and Le, 2019a] to be evaluated on ImageNet.

- *Supernet for NASNet search space*: our supernet follows the design of NASNet search space [Zoph et al., 2018b], the network of which is constructed by stacking multiple normal cells and reduction cells. Since the search space of normal/reduction cells is the same, the structure of supernet for both cells is also the same, shown in Fig. 4.3a. The supernet consists of 5 nodes, and each node connects to all previous nodes. While a NASNet only takes 2 inputs, we enforce this logic by masking. Each edge consists of 4 independent edges that correspond to 4 types of layers.

- *Supernet for EfficientNet search space*: we reused the supernet from [Cai et al., 2019a], please refer to Once-For-All for details.

Transforming supernet to a specific architecture

There are two steps to transform a supernet to a target architecture by masking. Here we illustrate it on the NASNet search space, and the procedures on EfficientNet are same.

1) *Specifying an architecture*: The NASNet search space specifies two inputs to a node, which can be any previous nodes. Therefore, we used 5 integers to specify the connections of 5 nodes, and each integer enumerates all the possible connections of a node. For example, node 4 in Fig. 4.3a has 5 inputs, there are 5 possibilities $C(5, 1)$ if two inputs are same, and 10 possibilities $C(5, 2)$ for different inputs, adding up to 15 possible connections. Similarly, the possibilities for node1,2,3,5 are 3, 6, 10 and 21. Therefore, we use a vector of 5 integers with the range of $1 \to 3$, $1 \to 6$, $1 \to 10$, $1 \to 15$, and $1 \to 21$ to represent possible

connections. After specifying the connectivity, we need to specify the layer type for each edge. In our experiments, a layer can be one of 3x3 separable convolution, 5x5 separable convolution, 3x3 max pooling and skip connect. Considering there are 10 edges in a NASNet cell, we use 10 integers ranging from 1 to 4 to represent the layer type chosen for an edge. Therefore, a NASNet can be fully specified with 15 integers (Fig. 4.3b).

2) *encoding to mask*: we need to change the encoding of 15 integers to a mask to deactivate the edges. Since the supernet in Fig 4.3a has 20 edges, we use a 20x4 matrix, with each row as a vector to specify a layer or deactivation. The conversion is straightforward; if an edge is activated in the encoding, the edge is a one-hot vector, or a vector of 0s otherwise.

Training supernet

As explained in sec. 4.2.3, we apply a random mask to each training iterations. We re-used the training pipeline from DARTs [Liu et al., 2018b]. To generate the random mask, we used 15 random integers (explained in generating random masks above) to generate a random architecture with their ranges specified in Fig. 4.3b; then we transform the random encoding to a random mask, which is subsequently applied on supernet in training.

Searching using pre-trained supernet

After training the supernet, it is fixed during search. A search method proposes an architecture $\mathbf{a}_i$ for the evaluation; we mask the supernet to $\mathbf{a}_i$, then evaluate the masked supernet to get v_i for $\mathbf{a}_i$. The new $\mathbf{a}_i$, v_i pair is stored in D_t to refine the search decision in the next iteration. Since the evaluation of $\mathbf{a}_i$ is reduced to evaluating masked supernet on the validation dataset, this has greatly reduced the computation cost, enabling a search algorithm to sample thousands of $\mathbf{a}_i$ in a reasonable amount of time.

4.2.4 Partition Analysis

The sampling efficiency is closely related to the partition quality of each tree node. Here we seek an upper bound for the number of samples in the leftmost leaf (the most promising region) to characterize the sample efficiency. LaNAS shows more speedup w.r.t random search as the size of the search space grows. Details are in sec 4.2.4.

Assumption 1. *Given a search domain Ω containing finite samples N, there exists a probabilistic density f*

such that $P(a < v < b) = \int_a^b f(v)dv$*, where* v *is the performance of a network* $\mathbf{a}$.

With this assumption, we can count the number of networks in the accuracy range of $[a, b]$ by $N * P(a \leq v \leq b)$. Since $v \in [0, 1]$ and the standard derivation $\sigma_v < \infty$, the following holds ([Mallows, 1991])

$$|E(\overline{v} - M_v)| < \sigma_v \tag{4.4}$$

$\overline{v}$ is the mean performance in Ω, and M_v is the median performance. Note $v \in [0, 1]$, and let's denote $\epsilon = |\hat{v} - \overline{v}|$. Therefore, the maximal distance from $\hat{v}$ to M_v is $\epsilon + \sigma_v$; and the number of networks falling between $\hat{v}$ and M_v is $N * max(\int_{\hat{v}-\epsilon-\sigma_v}^{M_v} f(v)dv, \int_{M_v}^{\hat{v}+\epsilon+\sigma_v} f(v)dv)$, denoted as δ. Therefore, the root partitions Ω into two sets that have $\leq \frac{N}{2} + \delta$ architectures.

Theorem 1. *Given a search tree of height =* h*, the sub-domain represented by the leftmost leaf contains at most* $2 * \delta_{max}(1 - \frac{1}{2^h}) + \frac{N}{2^h}$ *architectures, and* δ_{max} *is the largest partition error from the node on the leftmost path.*

The theorem indicates that LaNAS is approximating the global optimum at the speed of $N/2^h$, suggesting 1) the performance improvement will remain near plateau as $h \uparrow$ (verified by Fig 4.6(a)), while the computational costs ($2^h - 1$ nodes) exponentially increase; 2) the performance improvement w.r.t random search (cost $\sim N/2$) is more obvious on a large search space (verified by Fig.5 (a)→(c)).

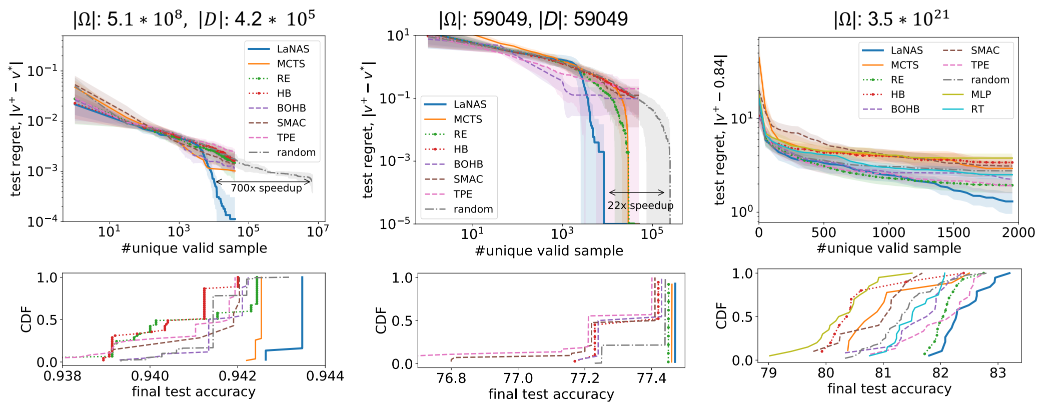
Proof of Theorem: In the worst scenario, the left child is always assigned with the large partition; and let's recursively apply this all the way down to the leftmost leaf h times, resulting in $\delta^h + \frac{\delta^{h-1}}{2} + \frac{\delta^{h-2}}{2^2} + ... + \frac{N}{2^h} \leq 2 * \delta_{max}(1 - \frac{1}{2^h}) + \frac{N}{2^h}$. δ is related to ϵ and σ_v; note $\delta \downarrow$ with more samples as $\epsilon \downarrow$, and σ_v becomes more accurate.

4.3 Experiments

4.3.1 Evaluating the search performance

Setup for benchmarks on NAS datasets

NAS datasets record architecture-accuracy pairs for the fast retrieval by NAS algorithms to avoid time-consuming model retraining. This makes repeated runs of NAS experiments in a tractable amount of computing time to truly evaluate search algorithms. We use NASBench-101 [Ying et al., 2019] as one benchmark



(a) NASBench-420K (b) ConvNet-60K (c) Supernet

Figure 4.4: The top row shows the time-course of test regrets of different methods (test regret between current best accuracy v^{+} and the best in dataset v^{*} with the interquartile range), while the bottom row illustrates Cumulative Distribution Function (CDF) of v^{+} for each method at $4 * 10^{4}$ unique valid samples. ConvNet-60K compensates NASBench to test the case of $|D| = |\Omega|$, and supernet compensates for the case of $|\Omega| \gg |\Omega_{nasbench}|$, where $|D|, |\Omega|$ are the size of the dataset and search space, respectively. LaNAS consistently demonstrates the best performance in 3 cases.

that contains over $4.2 * 10^{5}$ NASNet CNN models with edges ≤ 9 and nodes ≤ 7. To specify a network, search methods need 21 boolean variables for the adjacent matrix, and 5 3-value categorical variables for the node list[1], defining a search space of $|\Omega| = 5 * 10^{8} \gg$ the size of dataset $|D| = 4.2 * 10^{5}$. In practice, NASBench returns 0 for the missing architectures, which potentially introduces a bias in evaluations. Besides, NASBench is still several orders of smaller than a search space in practice, e.g. NASNet [Zoph et al., 2018b] $|\Omega| \sim 10^{20}$. To resolve these issues, we curate a ConvNet dataset having $5.9 * 10^{4}$ samples to cover the case of $|D| = |\Omega|$, and a supernet with $|\Omega| = 3.5 * 10^{21}$ to cover the case of $\Omega \gg \Omega_{nasbench}$ for benchmarks.

The curation of ConvNet-60K follows similar procedures in collecting 1,364 networks in sec.4.1, free structural parameters that can vary are: network depth $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, number of filters $C = \{32, 64, 96\}$ and kernel size $K = \{3 \times 3\}$, defining a $\Omega = 59049$. We train $\forall \mathbf{a}_i \in \Omega$ for 100 epochs, collect their final test accuracy v_i, store $(\mathbf{a}_i, v_i)$ in the dataset D. This small VGG-style, no residual connections, plain ConvNet search space can be fully specified with 10 3-value categorical variables, with each representing a type of filters.

We use a supernet on NASNet search space, and sec. 4.2.3 provides the details about the curation and the usage of a supernet in evaluating the search efficiency.

For NASBench-101, we used the architecture encoding of CIFAR-A in NASBench benchmarks from

[1]this is the best encoding scheme with the minimal missing architectures, which is also used in NASBench baselines.

this repository[2], as the discrepancy between the size of the dataset and the search space is the minimal. Specifically, an architecture is encoded with 21 Boolean variables and 5 3-values categorical variables, with each value corresponding to 3 layer types, i.e. 3x3 convolution, 1x1 convolution, and max-pool. The 21 Boolean variables represent the adjacent matrix in NASBench, while the 5 categorical variables represent the nodelist in NASBench. Therefore, $|\Omega| = 2^{21} * 3^5 = 5.1 * 10^8$. For ConvNet-60K, we used 10 3-values categorical variables to represent a VGG style CNN up to depth = 10, with each value correspond to 3 types of convolution layers , i.e. (filters=32, kernel = 3), (filters=64, kernel = 3) and (filters=96, kernel = 3). Therefore, $|\Omega| = 59049$. 3) *Supernet*: Since supernet implements the NASNet search space, the encoding of a supernet is same as the one used for NASBench-101.

We adopt the same baselines established by NASBench-101, and the same implementations from this public release[3]. These baselines cover diverse types of search algorithms. Regularized Evolution (RE) is a type of evolutionary algorithm that achieves SoTA performance for image recognition. While traditional BO method suffers from the scalability issue (e.g. the computation cost scales $\mathcal{O}(n^3)$ with #samples), random forest-based Sequential Model-based Algorithmic Configuration (SMAC) and Tree of Parzen Estimators (TPE) are two popular solutions by using a scalable surrogate model. HyperBand (HB) is a resource-aware (e.g. training iterations or time) search method, and Bayesian optimization-based HyperBand (BOHB) extended HB for strong any time performance. In addition to baselines in NASBench-101, we also added MCTS to validate latent actions. We have extensively discussed LaNAS v.s. these baselines in sec 4.3.1.

In LaNAS, the height of the search tree is 8; we used 200 random samples for the initialization, and #select = 50 (the number of samples from a selected partition, see sec. 4.3.1).

Details about Ensuring Fairness

1) The encoding scheme decides the size of the search space, thereby significantly affecting the performance. We ensure LaNAS and MCTS to use the same encoding as NASBench baselines on both datasets.

2) We noticed NASBench baselines allow the same architecture to be sampled at different steps, and we modified LaNAS and MCTS to be consistent with baselines (by not removing samples from the search space).

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

[3]https://github.com/automl/nas_benchmarks

3) We choose the number of unique, valid[4] samples instead of time to report the performance as model-free methods such as random search can easily iterate through the search space in a short time.

4) The hyper-parameters of baselines are set w.r.t ablation studies in NASBench-101, and we also tuned LaNAS and MCTS for the benchmark.

5) Each method is repeated 100 runs with different random seeds.

Empirical results

The top row of Fig. 4.4 shows the mean test regret, $|v^+ - v^*|$ where v^+ is the current best and v^* is the best in a dataset, along with the 25th and 75th percentile of each method through the course of searching, and the bottom row shows the robustness of methods at 40000 UVS on NASBench and ConvNet-60K, and 2000 UVS on supernet, respectively. Noted not all baselines are guaranteed to reach the global optimum, 40000 is the maximum UVS collected in 3 CPU days for all baselines on datasets, and 2000 is the maximum UVS on supernet in 3 GPU days.

We made the following observations: 1) *strong final performance*: LaNAS consistently demonstrates the strongest final performance on 3 tasks. On NASBench (Fig. 4.4(a)), the final test error of LaNAS is 0.011%, an order of smaller than the second best (0.137%); Similarly, on supernet (Fig. 4.4(c)), the highest test accuracy found by LaNAS is 83.5%, 0.75% better than the second best. 2) *good for one-shot NAS*: the strong final performance of LaNAS is more relevant to one-shot NAS as shown in Fig 4.4(c), as evaluations are fairly cheap. 3) *performance behavior*: across 3 experiments, the performance of LaNAS is comparable to Random Search in the few hundreds of samples ~ 500, and surpass baselines afterward. As an explanation for this behavior, we conduct a set of controlled experiments in Appendix. 4.3.2. We conclude that LaNAS needs a few hundreds of samples to accurately estimate boundaries, thereby good performance afterward. 4) *faster in larger* Ω: LaNAS is 700x and 22x faster than *random* in reaching similar regrets on NASBench-101 and ConvNet-60K. The empirical results validate our analysis that better performance w.r.t random search are observable on a larger search space.

Discussions of baselines v.s. LaNAS

Like existing SMBO methods, LaNAS uses a tree of linear regressor as surrogate S in predicting the performance of unseen samples, and its S is proven to be quite effective as the resulting partitions clearly separate

[4]we define valid samples as the samples in NASBench, and invalid as those in the search space but not in NASBench.

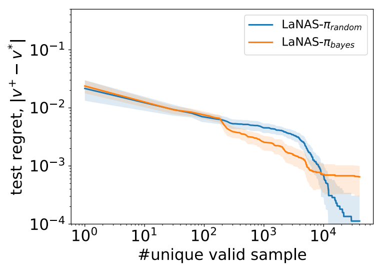
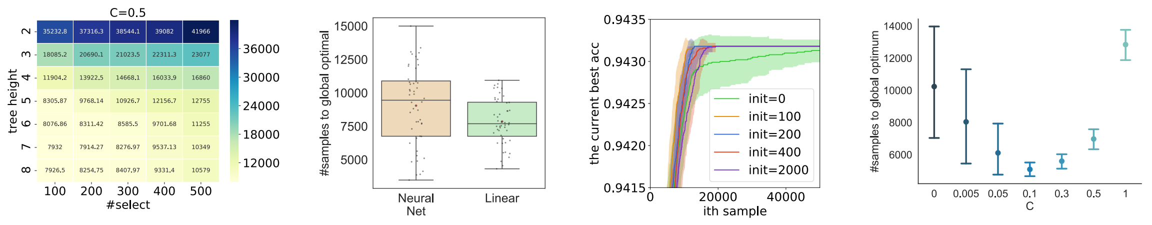


Figure 4.5: Comparisons of π_{bayes} to π_{random} in sampling from the selected partition Ω_j

good/bad Ω_j (validated by Fig. 4.7(a), and Fig. 4.7(b) in Appendix 4.3.2). Besides, LaNAS uses π_{ucb} in MCTS as the acquisition to trade-off between exploration and exploitation; All together makes LaNAS more efficient than non-SMBO baselines. For example, RS relies on blind search, leading to the worst performance. RE utilizes a static exploration strategy that maintains a pool of top-K architectures for random mutations, not making full use of previous search experience. MCTS builds online models of both performance and visitation counts for adaptive exploration. However, without learning action space, the performance model at each node cannot be highly selective, leading to inefficient search (Fig. 6.1). The poor performance of HB attributes to the low-rank correlation between the performance at different budgets (Fig.7 in Supplement S2 of [Ying et al., 2019]).

Compared to Bayesian methods, LaNAS learns the state partitions to simplify optimization of the acquisition function ϕ. With learned actions, optimization is as simple as a quick traverse down the tree to arrive at the most performant region Ω_j, regardless of the size of $|\Omega|$ and the dimensionality of tasks, and random sample a proposal within. Therefore, LaNAS gets a near-optimal solution to $\max_{\mathbf{a}_i \in \Omega} \phi(\mathbf{a}_i)$ but without explicit optimization. In contrast, Bayesian methods such as SMAC, TPE, and BOHB use iterated local search/evolutionary algorithm to propose a sample, which quickly becomes intractable on a high dimensional task, e.g. NAS with $|\Omega| > 10^{20}$. As a result, a sub-optimal solution to $\max_{\mathbf{a}_i \in \Omega} \phi(\mathbf{a}_i)$ leads to a sub-optimal sample proposal, thereby sub-optimal performance (shown in Fig **??**, Appendix). Consistent with [Wang et al., 2014], our results in Fig. 4.4 also confirms it. For example, Bayesian methods, BOHB in particular, perform quite well w.r.t LaNAS on ConvNet (the low dimensional task) especially in the beginning when LaNAS has not learned its partitions well, but their relative performances dwindle on NASBench and supernet (high dimensional tasks) as the dimensionality grows, $|\Omega_{supernet}| \gg |\Omega_{nasbench}| \gg |\Omega_{convnet}|$. Therefore, LaNAS is more effective than Bayesian methods in high-dimensional tasks.



(a) tree height v.s. #select (b) choice of classifiers (c) #samples for initialization (d) the impact of C in UCB

Figure 4.6: Ablation study: (a) the effect of different tree heights and #select in MCTS. The number in each entry is #samples to reach global optimal. (b) the choice of predictor for splitting search space. (c) the effect of #samples for initialization toward the search performance. (d) the effect of hyper-parameter c in UCB on NASBench performance.

Hyper-parameters tuning in LaNAS

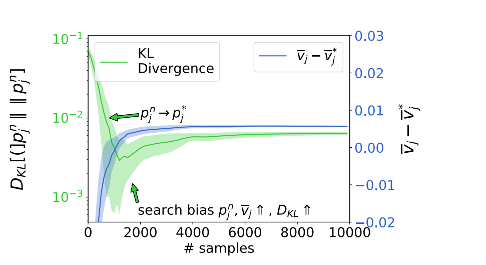
The effect of tree height and #selects: Fig. 4.6(a) relates tree height (h) and the number of selects (#selects) to the search performance. Each entry represents #samples to find v^* on NASBench, averaged over 100 runs. A deeper tree leads to better performance since the model space is partitioned by more leaves. Similarly, small #select results in more frequent updates of action space allowing the tree to make up-to-date decisions, and thus leads to improvement. On the other hand, the number of classifiers increases exponentially as the tree goes deeper, and a small #selects incurs a frequent learning phase. Therefore, both can significantly increase the computation cost.

Choice of classifiers: Fig.4.6(b) shows that using a linear classifier performs better than an multi-layer perceptron (MLP). This indicates that adding complexity to the decision boundary of actions may not help with the performance. Conversely, performance can get degraded due to potentially higher difficulties in optimization.

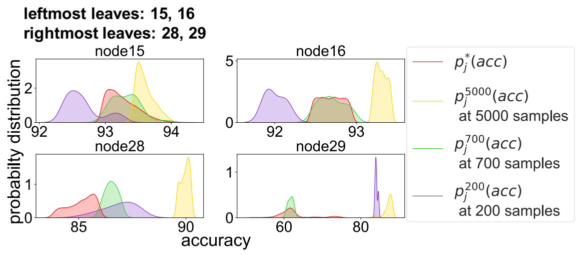
#samples for initialization: We need to initialize each node classifier properly with a few samples to establish the initial boundaries. Fig.4.6(c) shows cold start is necessary (init $>$ 0 is better than init = 0), and a small init=100-400 converges to top 5% performance much faster than init=2000, while init=2000 gets the best performance a little faster.

The effect of c in UCB: Fig. 4.6(d) shows that the exploration term, $c\sqrt{\frac{log(n_{curt})}{n_{next}}}$, improves the performance as c increases from 0 to 0.1, while using a large c, e.g. $>$ 0.5, is not desired for over-exploring. Please noted the optimal c is small as the maximum accuracy = 1. In practice, we find that setting c to $0.1*$max accuracy empirically works well. For example, if a performance metric in the range of [0, 100], we recommend setting c = 10.

Using π_{bayes} v.s. π_{random} for sample proposal: though π_{bayes} is faster in the beginning, π_{random} delivers the better final result due to the consistent random exploration in the most promising partition. Therefore, we used π_{random} for simplicity and good final performance through this paper.



(a) Search dynamics: KL-divergence and mean distance



(b) Search dynamics: sample distribution vs. dataset distribution

Figure 4.7: Evaluations of search dynamics:(a) KL-divergence of p_j and p_j^* dips and bounces back. $\bar{v} - \bar{v}^*$ continues to grow, showing the average metric $\bar{v}$ over different nodes becomes higher when the search progresses. (b) sample distribution p_j approximates dataset distribution p_j^* when the number of samples $n \in [200, 700]$. The search algorithm then zooms into the promising sub-domain, as shown by the growth of $\bar{v}_j$ when $n \in [700, 5000]$.

4.3.2 Analysis of LaNAS

To validate the effectiveness of latent actions in partitioning the search space into regions with different performance metrics, and to visualize the search phase of LaNAS, we look into the dynamics of sample distributions on tree leaves during the search. By construction, left nodes contain regions of the good metric while the right nodes contain regions of the poor metric. Therefore, at each node j, we can construct *reference* distribution $p_j^*(v)$ by training toward a NAS dataset to partition the dataset into small regions with concentrated performances on leaves, i.e. using regression tree for the classification. Then, we compare $p_j^*(v)$ with the estimated distribution $p_j^n(v)$, where n is the number of accumulated samples in $D_t \cap \Omega_j$ at the node j at the search step t. Since the *reference* distribution $p_j^*(v)$ is static, visualizing $p_j^n(v)$ to $p_j^*(v)$ and calculating $D_{KL}[p_j^n \| \| p_j^*]$ enables us to see variations of the distribution over partition Ω_j on tree leaves w.r.t growing samples to validate the effectiveness of latent actions and to visualize the search.

We used NASBench-101 that provides us with the true distribution of model accuracy, given any subset of model specifications, or equivalently a collection of actions (or constraints). In our experiments, we use a complete *binary* tree with the height of 5. We label nodes 0-14 as internal nodes, and nodes 15-29 as leaves. By definition, $\bar{v}_{15}^* > \bar{v}_{16}^* ... > \bar{v}_{29}^*$ reflected by $p_{15,16,28,29}^*$ in Fig. 4.7b.

At the beginning of the search ($n = 200$ for random initialization), $p_{15,16}^{200}$ are expected to be smaller than $p_{15,16}^*$, and $p_{28,29}^{200}$ are expected to be larger than $p_{15,16}^*$; because the tree still learns to partition at that time. With more samples ($n = 700$), p_j starts to approximate p_j^*, manifested by the increasing similarity between $p_{15,16,28,29}^{700}$ and $p_{15,16,28,29}^{*}$, and the decreasing D_{KL} in Fig. 4.7a. This is because MCTS explores the under-explored regions, and it explains the comparable performance of LaNAS to baselines in Fig. 4.4. As the search continues ($n \to 5000$), LaNAS explores deeper into promising regions and p_j^n is biased toward the region with good performance, deviated from p_j^*. As a result, D_{KL} bounces back in Fig. 4.7a. These search dynamics show how our model adapts to different stages during the course of the search, and validate its effectiveness in partitioning the search space.

The mean accuracy of $p_{15}^{700,5000} > p_{16}^{700,5000} > p_{28}^{700,5000} > p_{29}^{700,5000}$ in Fig. 4.7(b) indicates that LaNAS successfully minimizes the variance of rewards on a search path making architectures with similar metrics concentrated in a region, and LaNAS correctly ranks the regions on tree leaves. These manifest that LaNAS fulfills the online partitioning of Ω.

4.4 Related Work

Sequential Model Based Optimizations (SMBO) is a classic black box optimization framework [Hutter et al., 2011a], that uses a surrogate S to extrapolate unseen region in Ω and to interpolate the explored region with existing samples. In the scenarios of expensive function evaluations $f(\mathbf{a}_i)$, SMBO is quite efficient by approximating $f(\mathbf{a}_i)$ with $S(\mathbf{a}_i)$. SMBO proposes new samples by solving $\max_{\mathbf{a}_i} \phi(\mathbf{a}_i)$ on S, where ϕ is a criterion, e.g. Expected Improvement (EI) [Jones, 2001] or Conditional Entropy of the Minimizer (CEM) [Villemonteix et al., 2009], that transforms the value predicted from S for better trade-off between exploration and exploitation.

Bayesian Optimization (BO) is also an instantiation of SMBO [Wang et al., 2013, Gardner et al., 2014] that utilizes a Gaussian Process Regressor (GPR) as S. However, GPR suffers from the issue of $\mathcal{O}(n^3)$ where n is #samples. To resolve these issues, [Hutter et al., 2011a] replaces GPR with random forests, called SMAC-random forest, to estimate $\hat{\mu}$ and $\hat{\sigma}$ for predictive Gaussian distributions, and [Bergstra et al., 2011] proposes Tree-structured Parzen Estimator (TPE) in modeling Bayesian rule. Though both resolves the cubic scaling issue, as we thoroughly explained in sec 4.3.1, the key limitation of Bayesian approaches is at auxiliary optimization of acquisition function on an intractable search space. Similarly, recent predictor-based methods [Shi et al., 2019b] have achieved impressive results on NASBench by predicting every unseen architecture from the dataset. Without predicting over the entire dataset, their performance can drastically deteriorate. LaNAS eliminates this undesired constraint in BO or predictor-based methods, being scalable regardless of problem dimensions, while still captures promising regions for sample proposals.

Besides the recent success in games [Silver et al., 2016, Tian et al., 2019a], Monte Carlo Tree Search (MCTS) has also been used in robotics planning, optimization, and NAS [Buşoniu et al., 2013, Munos, 2014, Weinstein and Littman, 2012, Mansley et al., 2011]. AlphaX is the first MCTS based NAS agent to explore the search space assisted with a value function predictor. However, the action space of AlphaX is manually defined w.r.t the search space. In sec. 4.1, we have clearly demonstrated that manually defined search space provides a confusing reward signal to the search, therefore low sample efficiency. In contrast, LaNAS learns the action space to partition the search space into good and bad regions, providing a stronger reward signal to guide the search; and [Wang et al., 2020a] extends LaNAS to be a generic black box meta-solver.

XNAS [Nayman et al., 2019a] and many other existing NAS methods in Table. 9.1 are specifically designed to improve the search by exploiting the architecture characteristics in the NASNet or EfficientNet search space, while LaNAS is a new generic search algorithm applicable to the much broader scope of tasks. For example, P-DARTS [Chen et al., 2019b] observes there is a depth gap in the architecture during the search and evaluation steps in the search for CNN on the NASNet search space. Then, P-DARTS proposes progressively increasing the network depth during the search for CNN, so that the network depth can match the evaluation setting. But these task-dependent settings can change from tasks to tasks, or search space to search space. LaNAS treats NAS as a black box function without making any assumptions to the underlying search space, being adaptable to different problems. In the black box optimization challenge at NeurIPS-2020, LaNAS is proven to be effective in solving 216 different ML tasks [Sazanovich et al., 2020].

4.5 Additional Details

Algorithm 4 get_ucb($\bar{X}_{next}$, n_{next}, n_{curt}) in Alg. 6

```
1: c = 0.1
2: if $n_{next} = 0$ then
3:     return $+\infty$
4: else
5:     return $\frac{\bar{X}_{next}}{n_{next}} + 2c\sqrt{\frac{2log(n_{curt})}{n_{next}}}$
6: end if=0
```

Algorithm 5 LaNAS search procedures

```
1: while acc < target do
2:     for $n \in Tree.N$ do
3:         $n.g.train()$
4:     end for
5:     for $i = 1 \to \#selects$ do
6:         $leaf, path = ucb\_select(root)$
7:         $constraints = get\_constraints(path)$
8:         $network = sampling(constraints)$
9:         $acc = network.train()$
10:        $back\_propagate(network, acc)$
11:    end for
12: end while=0
```

Algorithm 6 ucb_select($c = root$) in Alg. 5

```
1: $path = []$
2: while $c$ not $leaf$ do
3:     $path.add(c)$
4:     $l_{ucb} = get\_ucb(c.left.\bar{X}, c.left.n, c.n)$
5:     $r_{ucb} = get\_ucb(c.right.\bar{X}, c.right.n, c.n)$
6: end while
7: while $c$ not $leaf$ do
8:     $path.add(c)$
9:     if $l_{ucb} > r_{ucb}$ then
10:        $c = c.left$
11:    else
12:        $c = c.right$
13:    end if
14: end while
15: return $path, c$ =0
```

Algorithm 7 get_constraints(path) in Alg. 5

```
1: $constraints$ = []
2: for $node \in s\_path$ do
3:     $\mathbf{W}, b = node.g.params()$
4:     $\bar{X} = node.\bar{X}$
5:     if $node$ on left then
6:         $constraints.add(\mathbf{Wa} + b \geq \bar{X})$
7:     else
8:         $constraints.add(\mathbf{Wa} + b < \bar{X})$
9:     end if
10: end for
11: return $constraints$ =0
```

4.6 Conclusion

This work presents a novel MCTS based search algorithm that learns action space for MCTS. With its application to NAS, LaNAS has proven to be more sample-efficient than existing approaches, validated by the cases with and without one-shot NAS on a diverse of tasks. The proposed algorithm is not limited to NAS and has been extended to be a generic gradient-free algorithm [Wang et al., 2020a], applied to different challenging black-box optimizations.

Chapter 5

Latent Action Monte Carlo Tree Search

5.1 Motivation

Black-box optimization has been extensively used in many scenarios, including Neural Architecture Search (NAS) [Zoph and Le, 2016, Wang et al., 2019a, Wang et al., 2019c], planning in robotics [Kim et al., 2020a, Buşoniu et al., 2013], hyper-parameter tuning in large scale databases [Pavlo et al., 2017] and distributed systems [Fischer et al., 2015], integrated circuit design [Mirhoseini et al., 2020], etc.. In black-box optimization, we have a function f without explicit formulation and the goal is to find $\mathbf{x}^*$ such that

$$\mathbf{x}^* = \arg\max_{\mathbf{x}\in X} f(\mathbf{x}) \tag{5.1}$$

with the fewest samples ($\mathbf{x}$). In this paper, we consider the case that f is deterministic.

Without knowing any structure of f (except for the local smoothness such as Lipschitz-continuity [Goldstein, 1977]), in the worst-case, solving Eqn. 5.1 takes exponential time, i.e. the optimizer needs to search every $\mathbf{x}$ to find the optimum. One way to address this problem is through *learning*: from a few samples we *learn* a surrogate regressor $\hat{f} \in \mathcal{H}$ and optimize $\hat{f}$ instead. If the model class $\mathcal{H}$ is small and f can be well approximated within $\mathcal{H}$, then $\hat{f}$ is a good approximator of f with much fewer samples.

Many previous works go that route, such as Bayesian Optimization (BO) and its variants [Brochu et al., 2010, Frazier, 2018, Shahriari et al., 2015, Bubeck et al., 2011]. However, in the case that f is highly nonlinear and high-dimensional, we need to use a very large model class $\mathcal{H}$, e.g. Gaussian Processes (GP) or Deep Neural Networks (DNN), that requires many samples to fit before generalizing well. For example, Oh et al [Oh et al., 2018] observed that the myopic acquisition in BO over-explores the boundary of a search space, especially in high dimensional problems. To address this issue, recent works start to explore space partitioning [Kim et al., 2020a, Munos, 2011, Wang et al., 2014] and local modeling [Eriksson et al., uRBO, Wang et al., 2017b] that fits local models in promising regions, and achieve strong empirical results in high dimensional problems. However, their space partitions follow a fixed criterion (e.g., K-ary uniform partition) that is independent of the objective to be optimized.

Following the path of learning, one under-explored direction is to *learn the space partition*. Compared to learning a regressor $\hat{f}$ that is expected to be accurate in the region of interest, it suffices to learn a classifier that puts the sample to the right subregion with high probability. Moreover, its quality requirement can be further reduced if done recursively.

In this section, we propose LA-MCTS, a meta-level algorithm that recursively learns space partition in a hierarchical manner. Given a few samples within a region, it first performs unsupervised K-mean algorithm based on their function values, learns a classifier using K-mean labels, and partition the region into good and bad sub-regions (with high/low function value). To address the problem of mis-partitioning good data points into bad regions, LA-MCTS uses UCB to balance exploration and exploitation: it assigns more samples to good regions, where it is more likely to find an optimal solution, and exploring other regions in case there are good candidates. Compared to previous space partition method, e.g. using Voronoi graph [Kim et al., 2020a], we learn the partition that is adaptive to the objective function $f(\mathbf{x})$. Compared to the local modeling method, e.g. TuRBO [Eriksson et al., uRBO], our method dynamically exploits and explores the promising region w.r.t samples using Monte Carlos Tree Search (MCTS), and constantly refine the learned boundaries with new samples.

LA-MCTS extends LaNAS [Wang et al., 2019a] in three aspects. First, while LaNAS learns a hyper-plane, our approach learns a non-linear decision boundary that is more flexible. Second, while LaNAS simply performs uniform sampling in each region as the next sample to evaluate, we make the key observation that local model works well and use existing solvers such as BO to find a promising data point. This makes LA-MCTS a *meta-algorithm* usable to boost existing algorithms that optimize via building local models. Third, while LaNAS mainly focus on Neural Architecture Search (¡ 20 discrete parameters), our approach shows strong performance on generic black-box optimization.

We show that LA-MCTS, when paired with TurBO, outperforms various SoTA black-box solvers from Bayesian Optimizations, Evolutionary Algorithm, and Monte Carlo Tree Search, in several challenging benchmarks, including *MuJoCo locomotion tasks*, trajectory optimization, reinforcement learning, and high-dimensional synthetic functions. We also perform extensive ablation studies, showing LA-MCTS is relatively insensitive to hyper-parameter tuning. As a meta-algorithm, it also substantially improves the baselines.

The implementation of LA-MCTS can be found at https://github.com/facebookresearch/LaMCTS.

5.2 Related Work

Bayesian Optimization (BO) has become a promising approach in optimizing the black-box functions [Brochu et al., 2010, Frazier, 2018, Shahriari et al., 2015], despite much of its success is typically limited to less than 15 parameters [Nayebi et al., esBO] and a few thousand evaluations [Wang et al., 2017b]. While most real-world problems are high dimensional, and reliably optimizing a complex function requires many evaluations. This has motivated many works to scale up BO, by approximating the expensive Gaussian Process (GP), such as using Random Forest in SMAC [Hutter et al., 2011a], Bayesian Neural Network in BOHAMIANN [Springenberg et al., 2016], and the tree-structured Parzen estimator in TPE [Bergstra et al., 2011]. BOHB [Falkner et al., Ster] further combines TPE with Hyperband [Li et al., 2017] to achieve strong any time performance. Therefore, we choose BOHB in comparison. Using a sparse GP is another way to scale up BO [Seeger et al., 2003, Snelson and Ghahramani, 2006, Hensman et al., 2013]. However, sparse GP only works well if there exists sample redundancy, which is barely the case in high dimensional problems. Therefore, scaling up evaluations is not sufficient for solving high-dimensional problems.

There are lots of work to specifically study high-dimensional BO [Wang et al., 2013, Kawaguchi et al., 2015, McIntire et al., 2016, Chen et al., 2012, Kandasamy et al., 2015, Binois et al., 2015, Wang et al., 2016c, Gardner et al., 2017, Rolland et al., 2018, Mutny and Krause, 2018]. One category of methods decomposes the target function into several additive structures [Kandasamy et al., 2015, Gardner et al., 2017], which limits its scalability by the number of decomposed structures for training multiple GP. Besides, learning a good decomposition remains challenging. Another category of methods is to transform a high-dimensional problem in low-dimensional subspaces. REMBO [Wang et al., 2016c] fits a GP in low-dimensional spaces and projects points back to a high-dimensional space that contains the global optimum with a reasonable probability. Binois et al [Binois et al., 2020] further improves the distortion from Gaussian projections in REMBO. While REMBO works empirically, HesBO [Nayebi et al., esBO] is a theoretical sound framework for BO that optimizes high-dimensional problems on low dimensional sub-spaces embeddings; In BOCK [Oh et al., 2018], Oh et al observed existing BO spends most evaluations near the boundary of a search space due to the Euclidean geometry, and it proposed transforming the problem into a cylindrical space to avoid over-exploring the boundary. EBO [Wang et al., 2017b] uses an ensemble of local GP on the partitioned problem space. Based on the same principle of local modeling as EBO, recent trust-region BO (TuRBO) [Eriksson et al., uRBO] has outperformed other high-dimensional BO on a variety of tasks. In comparing to high dimensional BO, we picked SoTA local modeling method TuRBO and dimension reduction method HesBO.

Evolutionary Algorithm (EA) is another popular algorithm for high dimensional black-box optimizations. A comprehensive review of EA can be found in [Jin and Branke, 2005]. CMA-ES is a successful EA method that uses co-variance matrix adaption to propose new samples. Differential Evolution (DE) [Storn and Price, 1997] is another popular EA approach that uses vector differences for perturbing the vector population. Recently, Liu et al proposes a metamethod (Shiwa) [Liu et al., grad] to automatically selects EA methods based on hyper-parameters such as problem dimensions, budget, and noise level, etc., and Shiwa delivers better empirical results than any single EA method. We choose Shiwa, CMA-ES, and differential evolution in comparisons.

Besides the recent success in games [Silver et al., 2016, Schrittwieser et al., 2019, Silver et al., 2017, Browne et al., 2012], Monte Carlo Tree Search (MCTS) is also widely used in the robotics planning and optimization [Buşoniu et al., 2013, Munos, 2014, Weinstein and Littman, 2012, Mansley et al., 2011]. Several space partitioning algorithms have been proposed in this line of research. In [Munos, 2011], Munos proposed DOO and SOO. DOO uses a tree structure to partition the search space by recursively bifurcating the region with the highest upper bound, i.e. optimistic exploration, while SOO relaxes the Lipschitz condition of DOO on the objective function. HOO [Bubeck et al., 2011] is a stochastic version of DOO. While prior works use K-ary partitions, Kim et al show Voronoi [Kim et al., 2020a] partition can be more efficient than previous linear partitions in high-dimensional problems. In this paper, based on the idea of space partitioning, we extend current works by learning the space partition so that the partition can adapt to the distribution of $f(\mathbf{x})$. Besides, we improve the sampling inside a selected region with BO. This also helps BO from over-exploring by bounding within a small region.

5.3 Latent Action Monte Carlo Tree Search (LA-MCTS)

This section describes LA-MCTS that progressively partitions the problem space. Please refer to Table. 5.1 for definitions of notations in this paper.

The model of MCTS search tree: At any iteration t, we have a dataset D_t collected from previous

Table 5.1: Definition of notations used through this paper.

$\mathbf{x}_i$	the ith sample	$f(\mathbf{x}_i)$	the evaluation of $\mathbf{x}_i$	D_t	collected $\{\mathbf{x}_i, \text{f}(\mathbf{x}_i)\}$ from iter $1 \to t$
Ω	the entire search space	Ω_j	the partition represented by node j	$D_t \cap \Omega_j$	samples classified in Ω_j
n_j	#visits at node j	v_j	the value of node j	ucb_j	the ucb score of node j

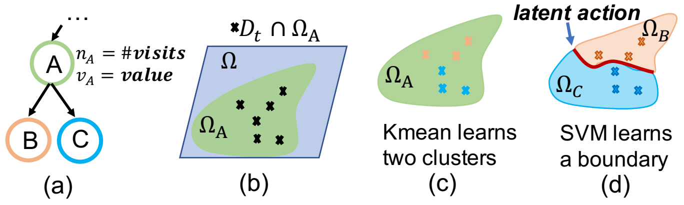


Figure 5.1: The model of latent actions: each tree nodes represents a region in the search space, and *latent action* splits the region into a high-performing and a low-performing region using $\mathbf{x}$ and $f(\mathbf{x})$.

evaluations. Each entry in D_t contains a pair of $(\mathbf{x}_i, f(\mathbf{x}_i))$. A tree node (e.g. node A in Fig. 5.1) represents a region Ω_A in the entire problem space (Ω), then $D_t \cap \Omega_A$ represents the samples falling within node A. Each node also tracks two important statistics to calculate UCB1 [Auer et al., 2002] for guiding the selection: n_A represents the number of visits at node A, which is the #sample in $D_t \cap \Omega_A$; and v_i represents the node value that equals to $\frac{1}{n_i}\sum f(\mathbf{x}_i), \forall \mathbf{x}_i \in D_t \cap \Omega_i$.

LA-MCTS finds the promising regions by recursively partitioning. Starting from the root, every internal node, e.g. node A in Fig. 5.1, use *latent actions* to bifurcate the region represented by itself into a high performing and a low performing disjoint region (Ω_B and Ω_C) for its left and right child, respectively (by default we use left child to represent a good region), and $\Omega_A = \Omega_B \cup \Omega_C$. Then a tree enforces the behavior of recursively partitioning from root to leaves so that regions represented by tree leaves (Ω_{leaves}) can be easily ranked from the best (the leftmost leaf), the second-best (the sibling of the leftmost leaf) to the worst (the rightmost leaf) due to the partitioning rule. The tree grows as the optimization progress, Ω_{leaves} becomes smaller, better focusing on a promising region (Fig. 5.9(b)). Please see sec 5.3 for the tree construction. By directly optimizing on Ω_{leaves}, it helps BO from over-exploring, hence improving the BO performance especially in high dimensional problems.

Latent actions: Our model defines *latent action* as a boundary that splits the region represented by a node into a high-performing and a low performing region. Fig. 5.1 illustrates the concept and the procedures of creating latent actions on a node. Our goal is to learn a boundary from samples in $D_t \cap \Omega_A$ to maximize the performance difference of two regions split by the boundary. We apply Kmeans on the feature vector of $[\mathbf{x}, f(\mathbf{x})]$ to find a good and a bad performance clusters in $D_t \cap \Omega_A$, then use SVM to learn a decision

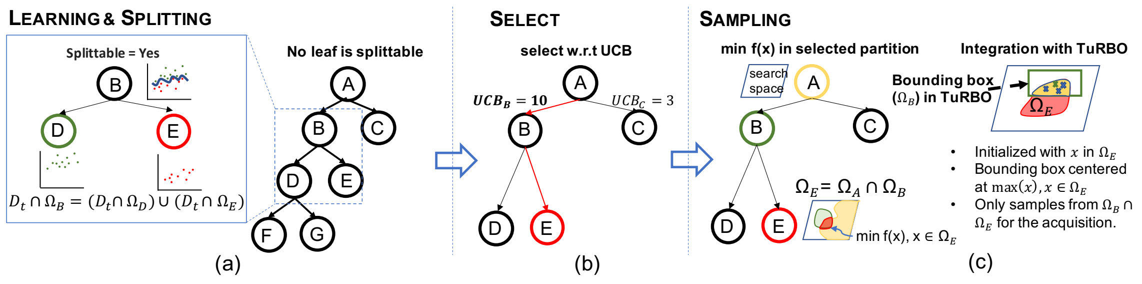

LEARNING & SPLITTING
Splittable = Yes
$D_t \cap \Omega_B = (D_t \cap \Omega_D) \cup (D_t \cap \Omega_E)$
No leaf is splittable
(a)
SELECT
select w.r.t UCB
$UCB_B = 10$
$UCB_C = 3$
(b)
SAMPLING
min f(x) in selected partition
search space
$\Omega_E = \Omega_A \cap \Omega_B$
min f(x), x ∈ Ω_E
Integration with TuRBO
Bounding box (Ω_B) in TuRBO
Initialized with x in Ω_E
Bounding box centered at $\max(x), x \in \Omega_E$
Only samples from $\Omega_B \cap \Omega_E$ for the acquisition.
(c)

Figure 5.2: The workflow of LA-MCTS: In an iteration, LA-MCTS starts with building the tree via splitting, then it selects a region based on UCB. Finally, on the selected region, it samples by BO.

boundary. Learning a nonlinear decision boundary is a traditional Machine Learning (ML) task, Neural Networks (NN) and Support Vector Machines (SVM) are two typical solutions. We choose SVM for the ease of training, and requiring fewer samples to generalize well in practices. Please note a simple node model is critical for having a tree of them. For the same reason, we choose Kmeans to find two clusters with good and bad performance. The detailed procedures are as follows:

1. At any node A, we prepare $\forall[\mathbf{x}_i, f(\mathbf{x}_i)], i \in D_t \cap \Omega_j$ as the training data for Kmeans to learn two clusters of different performance (Fig. 5.1 (b, c)), and get the cluster label l_i for every $\mathbf{x}_i$ using the learned Kmeans, i.e. $[l_i, \mathbf{x}_i]$. So, the cluster with higher average f($\mathbf{x}_i$) represents a good performing region, and lower average f($\mathbf{x}_i$) represents a bad region.

2. Given $[l_i, \mathbf{x}_i]$ from the previous step, we learn a boundary with SVM to generalize two regions to unseen $\mathbf{x}_i$, and *the boundary learnt by SVM forms the latent action* (Fig. 5.1(d)). for example, $\forall \mathbf{x}_i \in \Omega$ with predicted label equals the high-performing region goes to the left child, and right otherwise.

The search procedures

Fig. 5.2 summarizes a search iteration of LA-MCTS that has 3 major steps. 1) *Learning and splitting* dynamically deepens a search tree using new $\mathbf{x}_i$ collected from the previous iteration; 2) *select* explores partitioned search space for sampling; and 3) *sampling* solves $minimize f(\mathbf{x}_i), \mathbf{x}_i \in \Omega_{selected}$ using BO, and SVMs on the selected path form constraints to bound $\Omega_{selected}$. We omit the back-propagation as it is implicitly done in splitting. Please see [Wang et al., 2019c, Browne et al., 2012] for a review of regular MCTS.

Dynamic tree construction via splitting: we estimate the performance of a Ω_i, i.e. v_i^*, by $\hat{v}_i^* = \frac{1}{n_i}\sum f(\mathbf{x}_i), \forall \mathbf{x}_i \in D_t \cap \Omega_i$. At each iterations, new $\mathbf{x}_i$ are collected and the regret of $|\hat{v}_i^* - v_i^*|$ quickly decreases. Once the regret reaches the plateau, new samples are not necessary; then LA-MCTS splits the region using *latent actions* (Fig. 5.1) to continue refining the value estimation of two child regions. With more and more samples from promising regions, the tree becomes deeper into good regions, better guiding the search toward the optimum. In practice, we use a threshold θ as a tunable parameter for splitting. If the size of $D_t \cap \Omega_i$ exceeds the threshold θ at any leaves, we split the leaf with *latent actions*. We presents the ablation study on θ in Fig. 5.10.

The structure of our search tree dynamically changes across iterations, which is different from the pre-defined fixed-height tree used in LaNAS [Wang et al., 2019a]. At the beginning of an iteration, starting from the root that contains all the samples, we recursively split leaves using *latent actions* if the sample size of any leaves exceeds the splitting threshold θ, e.g. creating node D and node E for node B in Fig.2(a). We stop the tree splitting until no more leaves satisfy the splitting criterion. Then, the tree is ready to use in this iteration.

Select via UCB: According to the partition rule, a simple greedy based *go-left* strategy can be used to exclusively exploit the current most promising leaf. This makes the algorithm over-exploiting a region based on existing samples, while the region can be sub-optimal with the global optimum located in a different place. To build an accurate global view of Ω, LA-MCTS selects a partition following Upper Confidence Bound (UCB) for the adaptive exploration; and the definition of UCB for a node is $ucb_j = \frac{v_j}{n_j} + 2C_p * \sqrt{2log(n_p)/n_j}$, where C_p is a tunable hyper-parameter to control the extent of exploration, and n_p represents #visits of the parent of node j. At a parent node, it chooses the node with the largest ucb score. By following UCB from the root to a leaf, we select a path for sampling (Fig. 5.2(b)). When $C_p = 0$, UCB degenerates to a pure greedy based policy, e.g. *regression tree*. An ablation study on C_p in Fig. 5.10(a) highlights that the exploration is critical to the performance.

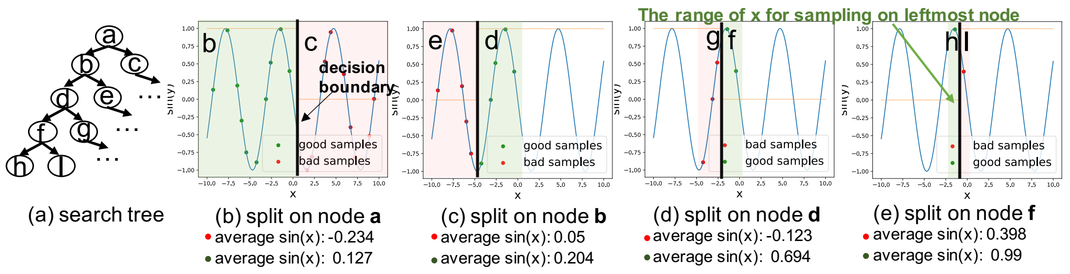


Figure 5.3: The visualization of partitioning 1d $\sin(x)$ using LA-MCTS.

Sampling via Bayesian Optimizations: *select* finds a path from the root to leaf, and SVMs on the path collectively intersects a region for sampling (e.g. Ω_E in Fig. 5.2(c)). In *sampling*, LA-MCTS solves $min f(\mathbf{x})$ on a constrained search space $\Omega_{selected}$, e.g. Ω_E in Fig. 5.2(c).

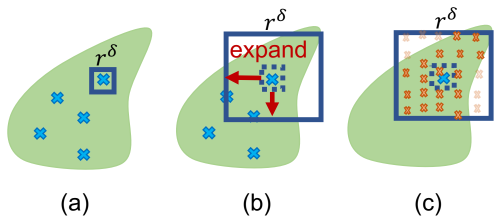


Figure 5.4: Illustration of sampling steps in optimizing the acquisition for Bayesian Optimization. We uniformly draw samples within a hyper-cube, then expand the cube and reject outliers.

Sampling with TuRBO: here we illustrate the integration of SoTA BO method TuRBO [Eriksson et al., uRBO] with LA-MCTS. We use TuRBO-1 (no bandit) for solving $minf(\mathbf{x})$ within the selected region, and make the following changes inside TuRBO, which is summarized in Fig. 5.2(c). a) At every re-starts, we initialize TuRBO with random samples only in $\Omega_{selected}$. The shape of $\Omega_{selected}$ can be arbitrary, so we use the rejected sampling (uniformly samples and reject outliers with SVM) to get a few points inside $\Omega_{selected}$. Since we only need a few samples for the initialization, the reject sampling is sufficient. b) TuRBO centers a bounding box at the best solution so far, while we restrict the center to be the best solution in $\Omega_{selected}$. c) TuRBO uniformly samples from the bounding box to feed the acquisition to select the best as the next sample, and we restrict the TuRBO to uniformly sample from the intersection of the bounding box and $\Omega_{selected}$. The intersection is guaranteed to exist because the center is within $\Omega_{selected}$. At each iteration, we keep TuRBO running until the size of trust-region goes 0, and all the evaluations, i.e. $\mathbf{x}_i$ and $f(\mathbf{x}_i)$, are returned to LA-MCTS to refine learned boundaries in the next iteration. Noted our method is also extensible to other solvers by following similar procedures.

Sampling with regular BO: following the steps described in Sec. 5.3, we select a leaf for sampling by traversing down from the root. The formulation of sampling with BO is same as using other solvers that $minf(\mathbf{x}), \mathbf{x} \in \Omega_{selected}$. $\Omega_{selected}$ is constrained by SVMs on the selected path. We optimize the acquisition function of BO by sampling, while sampling in a bounded arbitrary $\Omega_{selected}$ is nontrivial especially in high-dimensional space. For example, *rejected sampling* can fail to work as the search space is too large to get sufficient random samples in $\Omega_{selected}$; *hit-and-run* [Bélisle et al., 1993] or *Gibbs sampling* [Gelfand and Smith, 1990] can be good alternatives. In Fig. 5.4, we propose a new heuristic for sampling. At every existing samples $\mathbf{x}$ inside $\Omega_{selected}$, we draw a rectangle r^δ of length equals to δ centered at $\mathbf{x}_i$ (Fig. 5.4(a)), and $\mathbf{x}_i \in \Omega \cap D_t$, where δ is a small constant (e.g. 10^{-4}). Next, we uniformly draw random samples using sobol sequence [Sobol', 1967] inside r^δ. Since δ is a small constant, we assume all the random samples located

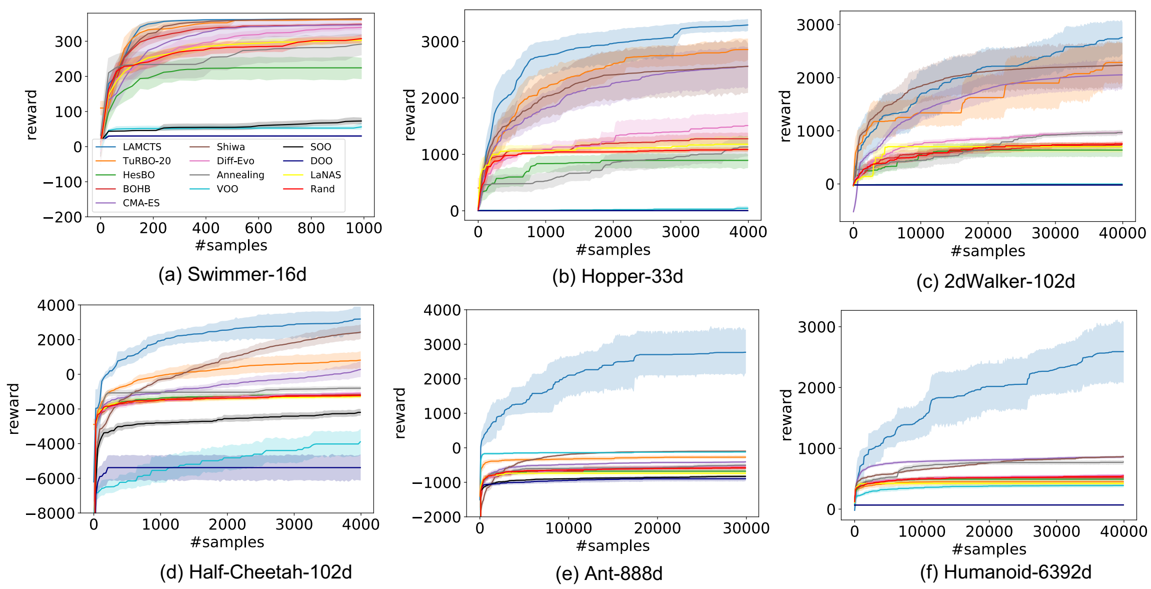


Figure 5.5: Benchmark on MuJoCo locomotion tasks: LA-MCTS consistently outperforms baselines on 6 tasks. With more dimensions, LA-MCTS shows stronger benefits (e.g. Ant and Humanoid). This is also observed in Fig. 5.8. Due to exploration, LA-MCTS experiences relatively high variance but achieves better solution after 30k samples, while other methods quickly move into local optima due to insufficient exploration.

inside $\Omega_{selected}$. Then we linearly scale both the rectangle r^{δ} and samples within r^{δ} until certain percentages (e.g. 10) of samples located outside of $\Omega_{selected}$ (Fig. 5.4(b)). We keep those samples that located inside $\Omega_{selected}$ (Fig. 5.4(c)) for optimizing the acquisition, and repeat the procedures for every existing samples in $\Omega_{selected} \cap D_t$. Finally, we propose the sample with the largest value calculated from the acquisition function.

5.4 Experiments

We evaluate LA-MCTS against the SoTA baselines from different algorithm categories ranging from Bayesian Optimization (TuRBO [Eriksson et al., uRBO], HesBO [Nayebi et al., esBO], BOHB [Falkner et al., Ster]), Evolutionary Algorithm (Shiwa [Liu et al., grad], CMA-ES [Hansen et al., ycma], Differential Evolution (DE) [Storn and Price, 1997]), MCTS (VOO [Kim et al., 2020a], SOO [Munos, 2011], and DOO [Munos, 2011]), Dual Annealing [Pincus, html] and Random Search. In experiments, LA-MCTS is defaulted to use TuRBO for sampling unless state otherwise. For baselines, we used the authors' reference implementations (see the bibliography for the source of implementations).

5.4.1 MuJoCo Locomotion Tasks

MuJoCo [Todorov et al., 2012] locomotion tasks (*swimmer*, *hopper*, *walker-2d*, *half-cheetah*, *ant* and *humanoid*) are among the most popular Reinforcement Learning (RL) benchmarks, and learning a *humanoid* model is considered one of the most difficult control problems solvable by SoTA RL methods [Salimans et al., 2017]. While the push and trajectory optimization problems used in [Eriksson et al., uRBO, Wang et al., 2017b] only have up to 60 parameters, MuJoCo tasks are more difficult: e.g., the most difficult task *humanoid* in MuJoCo has 6392 parameters.

Here we chose the linear policy $\mathbf{a} = \mathbf{W}\mathbf{s}$ [Mania et al., sARS], where $\mathbf{s}$ is the state vector, $\mathbf{a}$ is the action vector, and $\mathbf{W}$ is the linear policy. To evaluate a policy, we average rewards from 10 episodes. We want to find $\mathbf{W}$ to maximize the reward. Each component of $\mathbf{W}$ is continuous and in the range of $[-1, 1]$.

Fig. 5.5 suggests LA-MCTS consistently out-performs various SoTA baselines on all tasks. In particular, on high-dimensional hard problems such as *ant* and *humanoid*, the advantage of LA-MCTS over baselines is the most obvious. Here we use TuRBO-1 to sample $\Omega_{selected}$ (see sec. 5.3). **(a) vs TuRBO**. LA-MCTS substantially outperforms TuRBO: with learned partitions, LA-MCTS reduces the region size so that TuRBO can fit a better model in small regions. Moreover, LA-MCTS helps TuRBO initialize from a promising region at every restart, while TuRBO restarts from scratch. **(b) vs BO**. While BO variants (e.g., BOHB) perform very well in low-dimensional problem (Fig. 5.5), their performance quickly deteriorates with increased problem dimensions (Fig. 5.5(b)→(f)) due to over-exploration [Oh et al., 2018]. LA-MCTS prevents BO from over-exploring by quickly getting rid of unpromising regions. By traversing the partition tree, LA-MCTS also completely removes the step of optimizing the acquisition function, which becomes harder in high dimensions. **(c) vs objective-independent space partition**. Methods like VOO, SOO, and DOO use hand-designed space partition criterion (e.g., k-ary partition) which does not adapt to the objective. As a result, they perform poorly in high-dimensional problems. On the other hand, LA-MCTS learns the space partition that depends on the objective $f(\mathbf{x})$. The learned boundary can be nonlinear and thus can capture the characteristics of complicated objectives (e.g., the contour of f) quite well, yielding efficient partitioning. **(d) vs evolutionary algorithm (EA)**. CMA-ES generates new samples around the influential mean, which may trap in a local optimum.

Comparison with gradient-based approaches: Table 5.2 summarizes the sample efficiency of SOTA gradient-based approach on 6 MuJoCo tasks. Note that given the prior knowledge that a gradient-based approach (i.e., exploitation-only) works well in these tasks, LA-MCTS, as a black-box optimizer, will spend

Table 5.2: Compare with gradient-based approaches on MuJoCo v1; and the performance on MuJoCo v2 is similar. Despite being a black-box optimizer, LA-MCTS still achieves good sample efficiency in low-dimensional tasks (*Swimmer*, *Hopper* and *HalfCheetah*), but lag behind in high-dimensional tasks due to excessive burden in exploration, which gradient approaches lack. For more details of ARS V2-t, NG-lin, NG-rbf and TRPO-nn, please refer to [Mania et al., sARS], [Rajeswaran et al., 2017], [Rajeswaran et al., 2017] and [Mania et al., sARS], respectively.

		The average episodes (#samples) to reach the threshold				
Task	**Reward Threshold**	**LA-MCTS**	**ARS V2-t**	**NG-lin**	**NG-rbf**	**TRPO-nn**
Swimmer-v2	325	**126**	427	1450	1550	N/A
Hopper-v2	3120	2913	1973	13920	8640	10000
HalfCheetah-v2	3430	3967	1707	11250	6000	4250
Walker2d-v2	4390	N/A($r_{best} = 3523$)	24000	36840	25680	14250
Ant-v2	3580	N/A($r_{best} = 2871$)	20800	39240	30000	73500
Humanoid-v2	6000	N/A($r_{best} = 3202$)	142600	130000	130000	unknown

N/A stands for not reaching reward threshold.
r_{best} stands for the best reward achieved by LA-MCTS under the budget in Fig. 5.5.

extra samples for exploration and is expected to be less sample-efficient than the gradient-based approach for the same performance. Despite that, on simple tasks such as *swimmer*, LA-MCTS still shows superior sample efficiency than NG and TRPO, and is comparable to ARS. For high-dimensional tasks, exploration bears an excessive burden and LA-MCTS is not as sample-efficient as other gradient-based methods in MuJoCo tasks. We leave further improvement for future work.

Comparison with LaNAS: LaNAS lacks a surrogate model to inform sampling, while LA-MCTS samples with BO. Besides, the linear boundary in LaNAS is less adaptive to the nonlinear boundary used in LA-MCTS (e.g. Fig. 5.10(b)).

5.4.2 Small-scale Benchmarks

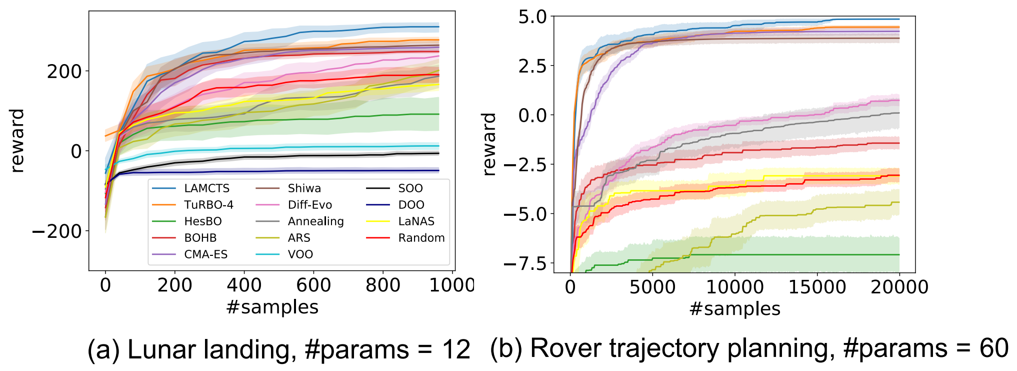


Figure 5.6: Evaluations on Lunar landing and Trajectory Optimization: LA-MCTS consistently outperforms baselines.

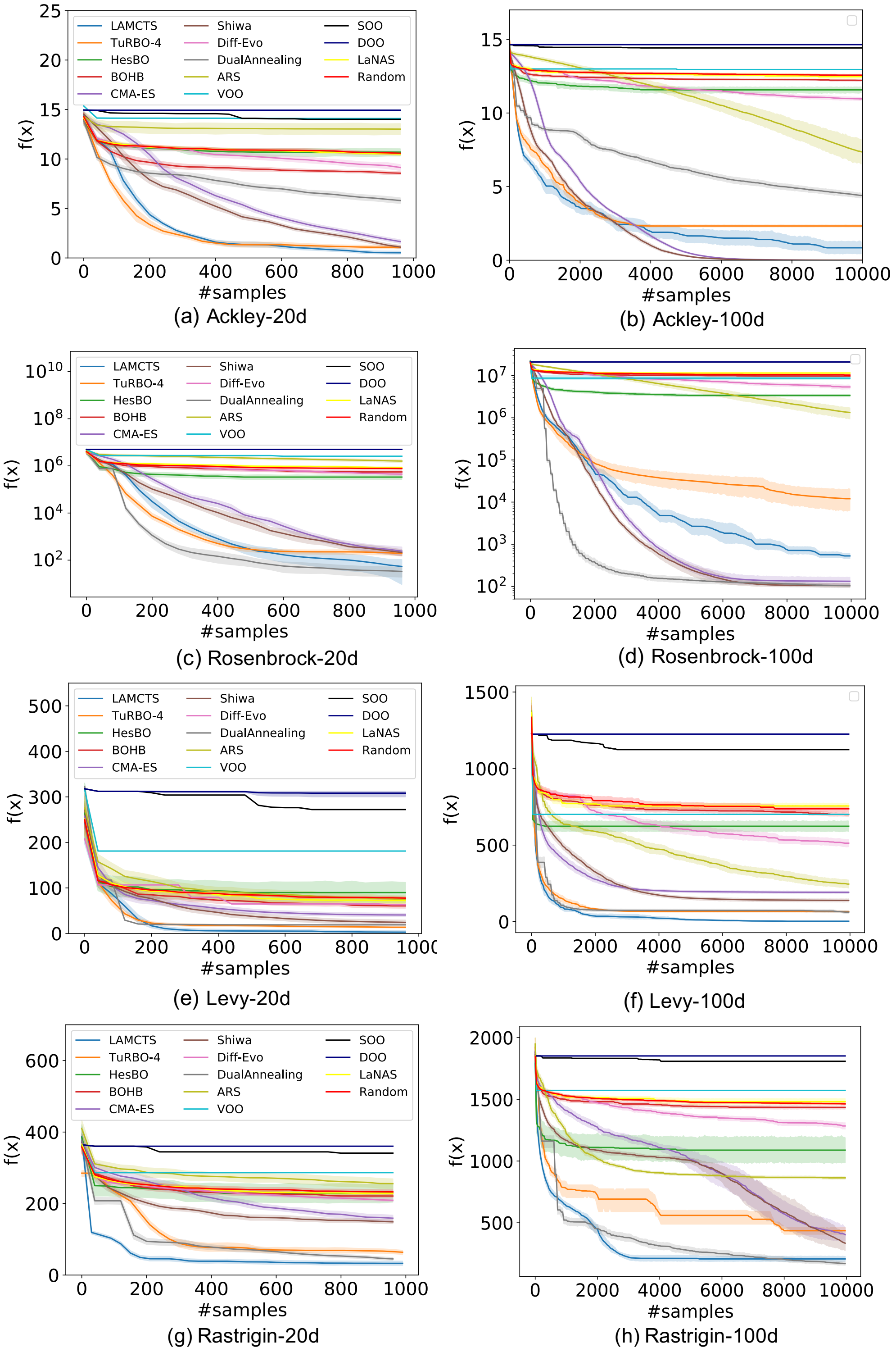


Figure 5.7: Evaluations on synthetic functions: the best method varies w.r.t functions, while LA-MCTS consistently improves TuRBO and being among top methods among all functions.

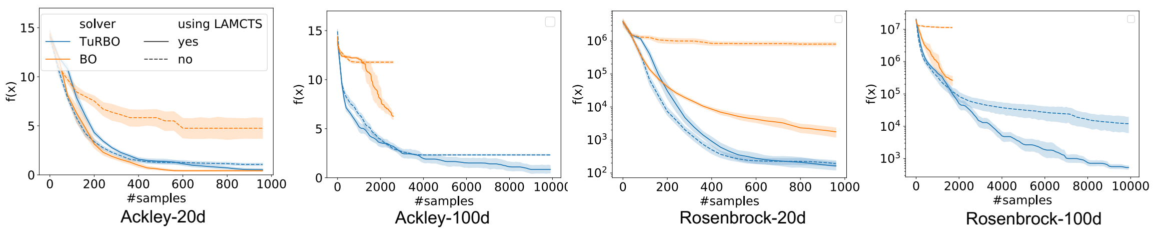


Figure 5.8: LA-MCTS as an effective meta-algorithm. LA-MCTS consistently improves the performance of TuRBO and BO, in particular in high-dimensional cases. We only plot part of the curve (each runs lasts for 3 day) for BO since it runs very slow in high-dimensional space.

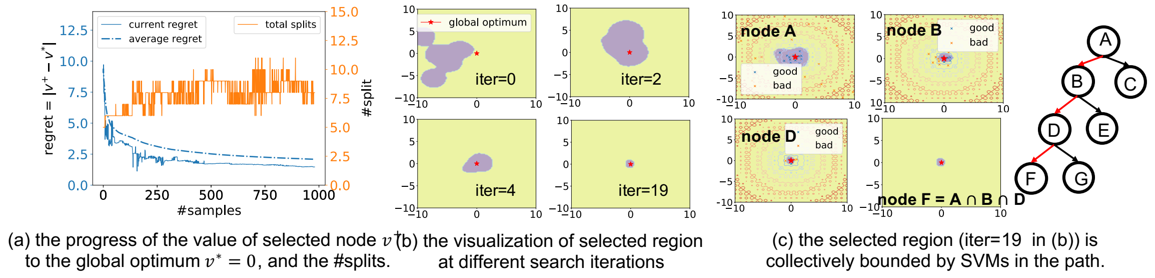


Figure 5.9: Validation of LA-MCTS: (a) the value of selected node becomes closer to the global optimum as #splits increases. (b) the visualization of $\Omega_{selected}$ in the progress of search. (c) the visualization of $\Omega_{selected}$ that takes the intersection of nodes on the selected path.

The setup of each methods can be found at Sec 5.6.1 in appendix, and figures are in Appendix 5.6.2.

Synthetic functions: We further benchmark with four synthetic functions, Rosenbrock, Levy, Ackley and Rastrigin. Rosenbrock and Levy have a long and flat valley including global optima, making optimization hard. Ackley and Rastrigin function have many local optima. Fig. 5.7 in Appendix shows the full evaluations to baselines on the 4 functions at 20 and 100 dimensions, respectively. The result shows the performance of each solvers varies a lot w.r.t functions. CMA-ES and TuRBO work well on Ackley, while Dual Annealing is the best on Rosenbrock. However, LA-MCTS consistently improves TuRBO on both functions.

Lunar Landing: the task is to learn a policy for the lunar landing environment implemented in the Open AI gym [Brockman et al., 2016], and we used the same heuristic policy from TuRBO [Eriksson et al., uRBO] that has 12 parameters to optimize. The state vector contains position, orientation and their time derivatives, and the state of being in contact with the land or not. The available actions are firing engine left, right, up, or idling. Fig. 5.6 shows LA-MCTS performs the best among baselines.

Rover-60d: the task was proposed in [Wang et al., 2017b] that optimizes 30 coordinates in a trajectory on a 2d plane, so the state vector consists of 60 variables. LA-MCTS still performs the best on this task.

5.4.3 Validation of LA-MCTS

LA-MCTS as an effective meta-algorithm: LA-MCTS internally uses TuRBO to pick promising samples from a sub-region. We also try using regular Bayesian Optimization (BO), which utilizes Expected Improvement (EI) for picking the next sample to evaluate. Fig. 5.8 shows LA-MCTS successfully boosts the performance of TuRBO and BO on Ackley and Rosenbrock function, in particular for high dimensional tasks. This is consistent with our results in MuJoCo tasks (Fig. 5.5).

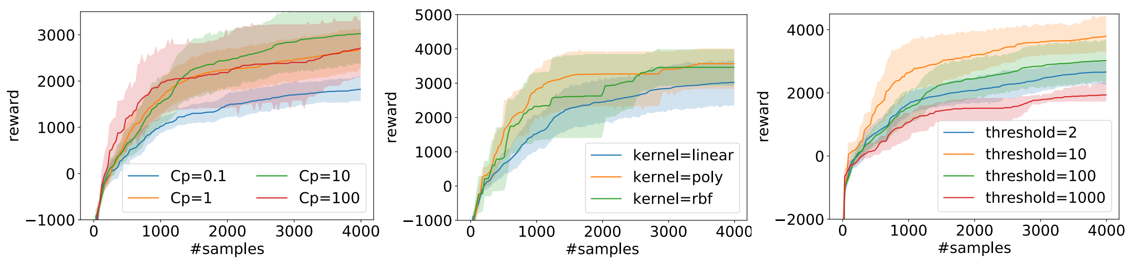
Validating LA-MCTS. Starting from the entire search space Ω, the node model in LA-MCTS recursively splits Ω into a high-performing and a low-performing regions. The value of a region v^+ is expected to become closer to the global optimum v^* with more and more splits. To validate this behavior, we setup LA-MCTS on Ackley-20d in the range of $[-5, 10]^{20}$, and keeps track of the value of a selected partition, $v_i^+ = \frac{1}{n_i}\sum f(\mathbf{x}_i), \forall \mathbf{x}_i \in D_t \cap \Omega_{selected}$, and as well as the number of splits at each steps. The global optimum of Ackley is at $v^* = 0$. We plot the progress of regret $|v_i^+ - v^*|$ in the left axis of Fig. 5.9(a), and the number of splits in the right axis of Fig. 5.9(a). Fig. 5.9 shows the regret decreases as the number of splits increases, which is consistent with the expected behavior. Besides, spikes in the regret curve indicate the exploration of less promising regions from MCTS.

Visualizing the space partition. We further understand LA-MCTS by visualizing space partition inside LA-MCTS on 2d-Ackley in the search range of $[-10, 10]^2$, which the global optimum v^* is marked by a red star at $\mathbf{x}^* = \mathbf{0}$. First, we visualize the $\Omega_{selected}$ in first 20 iterations, and show them in Fig. 5.9(b) and the full plot in Fig. 5.11(b) at Appendix. The purple indicates a good-performing region, while the yellow indicates a low-performing region. In iteration = 0, $\Omega_{selected}$ misses v^* due to the random initialization, but LA-MCTS consistently catches v^* in $\Omega_{selected}$ afterwards. The size of $\Omega_{selected}$ becomes smaller as #splits increases along the search (Fig. 5.9(a)). Fig. 5.9(c) shows the selected region is collectively bounded by SVMs on the path, i.e. $\Omega_F = \Omega_A \cap \Omega_B \cap \Omega_D \cap \Omega_F$.

5.4.4 Ablations on Hyper-parameters

Multiple hyper-parameters in LA-MCTS, including C_p in UCB, the kernel type of SVM, and the splitting threshold (θ), could impact its performance. Here ablation studies on *HalfCheetah* are provided for practical guidance.

Cp: C_p controls the amount of exploration. A large C_p encourages LA-MCTS to visit bad regions more often (exploration). As shown in Fig 5.10, too small C_p leads to the worst performance, highlighting



(a) Ablation on Cp (b) Ablation on SVM kernel (c) Ablation on splitting threshold

Figure 5.10: Ablation studies on hyper-parameters of LAMCTS.

the importance of exploration. However, a large C_p leads to over-exploration which is also undesired. We recommend setting C_p to 10% to 1% of max $f(\mathbf{x})$.

The SVM kernel: the kernel type decides the shape of the boundary drawn by each SVM. The linear boundary yields a convex polytope, while polynomial and RBF kernel can generate arbitrary region boundary, due to their non-linearity, which leads to better performance (Fig 5.10(b)).

The splitting threshold θ: the splitting threshold controls the speed of tree growth. Given the same #samples, smaller θ leads to a deeper tree. If Ω is very large, more splits enable LA-MCTS to quickly focus on a small promising region, and yields good results ($\theta = 10$). However, if θ is too small, the performance and the boundary estimation of the region become more unreliable, resulting in performance deterioration ($\theta = 2$, in Fig. 5.10).

5.5 Conclusion

The global optimization of high-dimensional black-box functions is an important topic that potentially impacts a broad spectrum of applications. We propose a novel meta method LA-MCTS that learns to partition the search space for Bayesian Optimization so that it can attend on a promising region to avoid over-exploring. Comprehensive evaluations show LA-MCTS is an effective meta-method to improve BO. In the future, we plan to extend the idea of space partitioning into Multi-Objective Optimizations.

5.6 Additional Details

5.6.1 Hyper-parameter Settings for All Baselines in the Benchmarks

Setup for MuJoCo tasks: Fig. 5.5 shows 13 methods in total, and here we describe the hyper-parameters of each method. Since we're interested in the sample efficiency, the batch size of every method is set to 1. We reuse the policy and evaluation codes from ARS [Mania et al., sARS], and the URL to the ARS implementation can be found in the bibliography. The implementations of VOO, SOO, and DOO are from here [1]; methods including CMA-ES, Differential Evolution, Dual Annealing are from the optimize module in scipy, and Shiwa is from Nevergrad[2]. Please see the bibliography for the reference implementations of BOHB and HesBO.

LA-MCTS	we use 30 samples for the initialization; and the SVM uses RBF kernel for easy tasks including *swimmer*, *hopper*, *half-cheetah*, and linear kernel for hard tasks including *2d-walker*, *ant* and *humanoid* to get over $3 * 10^4$ samples. C_p is set to 10, and the splitting threshold θ is set to 100. LA-MCTS uses TuRBO-1 for sampling, and the setup of TuRBO-1 is exactly the same as TuRBO described below. TuRBO-1 returns all the samples and their evaluations to LA-MCTS once it hits the re-start criterion.
TuRBO	we use 30 samples for the initialization, and CUDA is enabled. The rest hyper-parameters use the default value in TuRBO. In MuJoCo, we used TuRBO-20 that uses 20 independent trust regions for the best $f(x)$.
LaNAS	we use 30 samples for the initialization; the height of search tree is 8, and C_p is set to 10.
VOO	default setting in the reference implementation.
DOO	default setting in the reference implementation.
SOO	default setting in the reference implementation.
CMA-ES	the initial standard deviation is set to 0.5, and the rest parameters are default in Scipy.
Diff-Evo	default settings in Scipy.
Shiwa	default settings in Nevergrad.

[1]https://github.com/beomjoonkim/voot

[2]https://github.com/facebookresearch/nevergrad

Annealing	default settings in Scipy.
BOHB	default settings in the reference implementation.
HesBO	The tuned embedding dimensions for *swimmer*, *hopper*, *walker*, *half-cheetah*, *ant*, and *humanoid* are 8, 17, 50, 50, 400, and 1000, respectively.

Setup for synthetic functions, lunar landing, and trajectory optimization: similar to MuJoCo tasks, the batch size of each method is set to 1. The settings of VOO, DOO, SOO, CMA-ES, Diff-Evo, Dual Annealing, Shiwa, BOHB, TuRBO are the same as the settings from MuJoCo. We modify $C_p = 1$ and the splitting threshold $\theta = 20$ in LA-MCTS. Similarly, we also changed $C_p = 1$ in LaNAS. We set the upper and lower limits of each dimension in Ackley as [-5, 10], Rosenbrock is set within [-10, 10], Rastrigin is set within [-5.12, 5.12], Levy is set within [-10, 10].

Runtime: LaNAS, VOO, DOO, SOO, CMA-ES, Diff-Evo, Shiwa, Annealing are fairly fast, which can collect thousands of samples in minutes. The runtime performance of LAMCTS and TuRBO are consistent with the result in [Eriksson et al., uRBO] (see sec.G in appendix) that collects 10^4 samples in an hour using 1 V100 GPU. BOHB and HesBO toke up to a day to collect 10^4 samples for running on CPU.

5.6.2 Additional Experiment Results

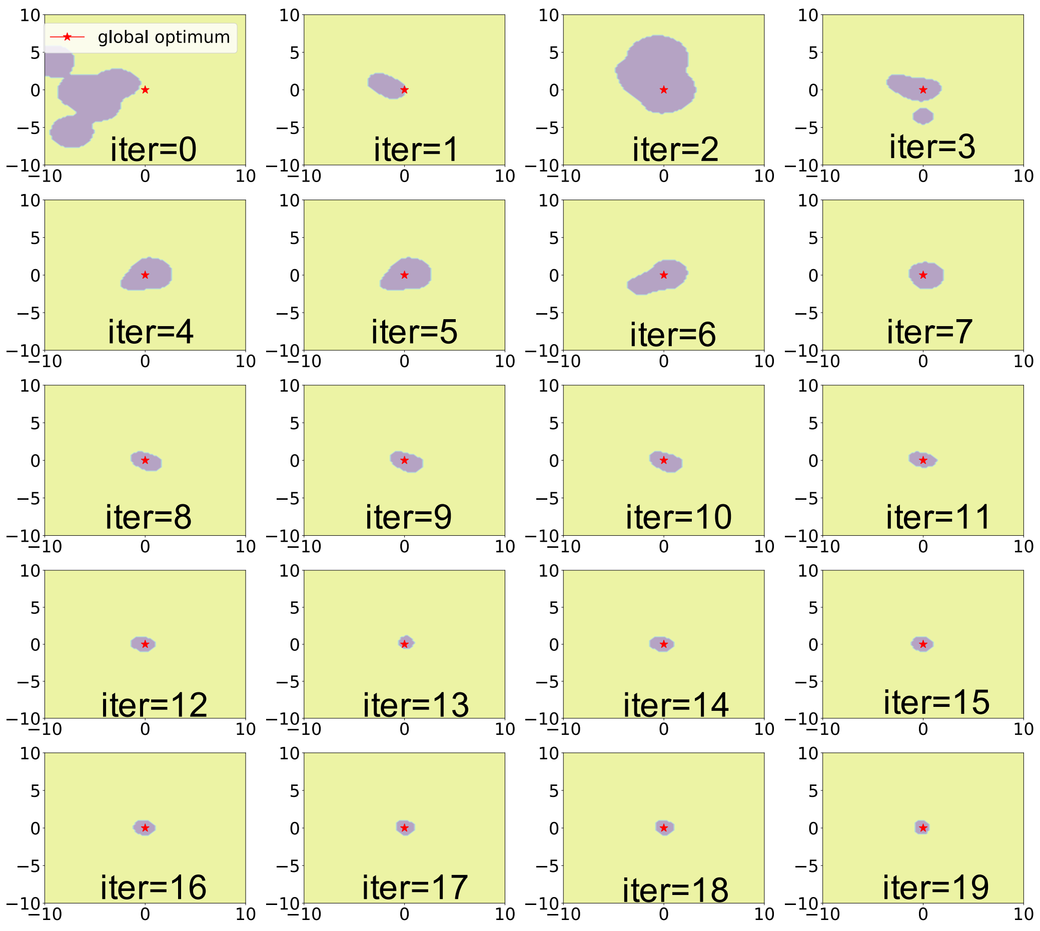


Figure 5.11: The visualization of LA-MCTS in iterations 1→20: the purple region is the selected region $\Omega_{selected}$, and the red star represents the global optimum.

Part III

Building the Evaluation Module

Chapter 6

Few-shot NAS

6.1 Motivation

Vanilla NAS requires a tremendous amount of computational costs (e.g., thousands of GPU hours) in order to find a superior neural architecture [Zoph et al., 2018a, Baker et al., 2017a, Real et al., 2019b], most of which is due to evaluating new architecture proposals by training them from scratch. To reduce the cost, one-shot NAS [Pham et al., 2018b, Liu et al., 2019] proposes to train a single supernet that represents all possible architectures in the search space. With supernet, the performance of architecture can be approximately evaluated by inheriting the corresponding weights from the supernet without training, reducing the search cost to just a few days (hours).

However, one-shot NAS suffers from degraded search performance due to the inaccurate predictions from the supernet. On NASBench-201, the best reported Kendall's Tau [Kendall., 1938] (a measurement of rank correlation) between the performance predicted from a supernet and the true performance is only 0.5748 [Yiming Hu, 2020]. Other works also have explicitly shown using supernet degrades the final performance due to the inaccurate performance predictions. For example, [Yu et al., 2019b] observes that, without using the supernet, the average performance of NAS algorithms such as ENAS and NAO is 1% higher than using it on NASBench-101, and they also conclude that the supernet never produces the true ranking. Besides, many works [Bender et al., 2018, Yu et al., 2020b, Luo et al., 2018b, Dong and Yang, 2020, Luo et al., 2020] also show that there is a non-trivial performance gap between the architectures found by one-shot NAS and vanilla NAS. Being consistent with the analysis in [Yu et al., 2019b], the main reason is that the performance predicted by the supernet has a low correlation with the true performance.

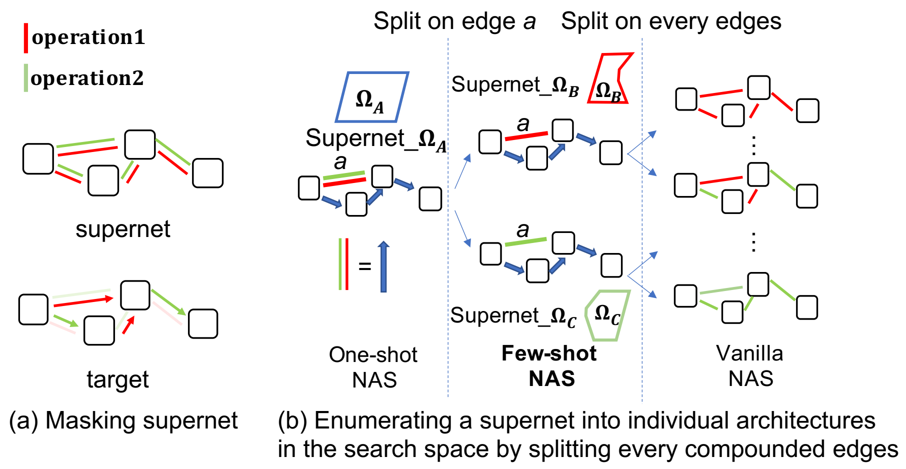
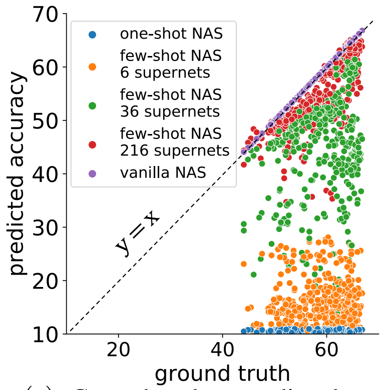


Figure 6.1: (a) masking supernet to a specific architecture. (b) the motivation of using few-shot NAS to alleviate the co-adaption impact. After splitting on edge a, supernet_Ω_B exclusively predicts architectures in Ω_B, so does supernet_Ω_C.

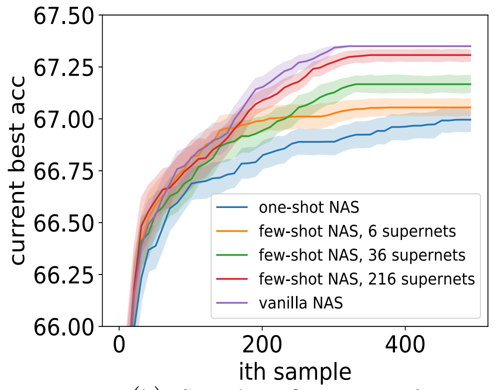
One explanation of the supernet's poor performance is the co-adaption of operations from a compound edge. Bender et al [Bender et al., 2018] show that the compound operations on an edge of the supernet can degrade the correlation between the estimated performance from a supernet and the true performance from training-from-scratch. While Bender et al primarily focused on using drop path or dropout to ensure a robust supernet for performance prediction, our method was motivated by the following observation on one-shot NAS and vanilla NAS.

One-shot NAS uses a supernet to predict the performance of a specific architecture by deactivating the extra edges w.r.t a target architecture on the supernet via masking (Fig. 6.1(a)), then perform evaluations using the masked supernet. Therefore, we can view supernet as a representation of search space Ω, and by masking, supernet can transform to any architectures in Ω. This also implies we can enumerate all the architectures in Ω by recursively splitting every compound edge in a supernet. Fig. 6.1(b) illustrates the splitting process, the root is the supernet and leaves are individual architectures in the search space Ω; the figure illustrates the case of splitting the compound edge a, and the recursively split follows similar procedures on all compound edges. In Fig. 6.1(b), one-shot NAS is the fastest but the most inaccurate in evaluations, while vanilla NAS is the most accurate in evaluations but the slowest. However, the middle ground, i.e. using multiple supernets, between one-shot NAS and vanilla NAS remains unexplored.

In a supernet, the effect of co-adaption results from combined operations on edges; therefore the evaluation of vanilla NAS is the most accurate. Based on this logic, it seems using several sub-supernets is a vanilla NAS v.s. predicted performance from one-shot NAS and few-shot NAS



(a) Ground truth vs predicted accuracy



(b) Search performance using regularized evolution

#Supernets	1	6	36	216	1296
Kendall Tau	0.013	0.12	0.26	0.63	1.0

(c) Rank correlations(Kendall Tau) for different numbers of supernets

Figure 6.2: (a) Using multi-supernets clearly improves the correlation and (c) provides the correlation score (Kendall Tau) at different numbers of supernets in (a); (b) shows the improved performance predictions result in better performance on NAS.

reasonable approach to alleviate the co-adaption effect by dissecting a compound edge into several separate sub-supernets that take charge of different sub-regions of the search space. For example, Fig. 6.1(b) shows few-shot NAS eliminates one compound edge a after splitting, resulting in two supernets for Ω_B and Ω_C, respectively. So, the predictions from resulting sub-supernet are free from the co-adaption effects from the split compound edge a.

We designed a controlled experiment to verify the assumption that *using multi-supernets improves the performance prediction*. First, we designed a search space having 1296 architectures, and trained each architecture toward the convergence to collect the final evaluation accuracy as the ground truth. Then we split the one-shot version of supernet into 6, 36, 216 sub-supernets following the procedures in Fig. 6.1(b). Finally, we trained each supernet with the same training pipeline in [Bender et al., 2018], and compared the predicted 1296 architecture performance to the ground truth using 1 (one-shot NAS), 6, 36, 216 supernets. Fig. 6.2 visualizes the results, and it indicates using multi-supernets significantly improves the correlation between predicted performance and the ground truth. Specifically, in Fig. 6.2(c) the Kendall's Tau [Kendall., 1938] ranking correlation of using 1 supernet (one-shot NAS), 6 supernets, 36 supernets, 216 supernets are 0.013, 0.12, 0.26, 0.63, respectively. As a result, the search algorithm takes fewer samples to find better networks

Table 6.1: The definition of notations used through the paper.

Ω	the whole architecture space	$\mathcal{A}$	an architecture in the architecture space
m	number of operations in the architecture space	$\mathcal{S}$	supernet
N_i	the ith node in the architecture space	n	number of nodes in the architecture space
$\Omega^{'}$	a sub-region of the whole architecture space	E_{ij}	the mixture operations between node i and j
$\mathcal{W}$	weights of neural network	$\mathcal{S}^{\Omega^{'}}$	a sub-supernet
$f(\mathcal{A})$	the evaluation of $\mathcal{A}$	$f(\mathcal{S}_{\mathcal{A}})$	the evaluation of $\mathcal{A}$ by supernet

due to more accurate performance predicted from supernets (Fig. 6.2(b)).

In this chapter, we propose *few-shot NAS* that uses multiple supernets in the architecture search. Instead of having one supernet covering the entire search space, which may be beyond its capacity and suffer from the co-adaption effect from the compound edges, we show that using multiple supernets can effectively address these issues by having each supernets modeling one part of the search space and by reducing the number of compound edges.

6.2 Methodology

In designing *few-shot NAS*, we answer the following several key questions: 1. how to divide the search space represented by the one-shot model to sub-supernets and how to choose the number of sub-supernets given a search time budget (Section 6.2.1)? 2. how to reduce the training time of multiple sub-supernets (Section 6.2.2)?; We also describe how to integrate *few-shot NAS* with existing NAS algorithms in Section 6.2.3 and Section 6.2.4.

6.2.1 The Design of Split Strategy

Our empirical observation from Section 6.1 can be summarized as following: the evaluation $f(\mathcal{S}_{\mathcal{A}}^{\Omega^k})$ of an architecture A using a sub-supernet $\mathcal{S}^{\Omega^k}$ is closer to the true accuracy $f(\mathcal{A})$ as Ω^k gets smaller, i.e., deeper in the tree. However, the prediction improvement for A diminishes with any sub-region Ω^p smaller than sub-region Ω^q where $A \in \Omega^q$. Furthermore, the time to split the initial architecture space Ω grows exponentially with tree depth. In short, the ideal split would be determined based on individual architecture and find the sub-supernet at the shallowest tree depth.

Before describing the split strategy, let's first define a generic NAS space that is compatible with one-shot NAS. We use this architecture space for introducing some necessary concepts that will be used throughout the paper. The whole architecture space Ω is represented by a directed acyclic graph (DAG) shown in Figure 6.1.

Table 6.2: Rank correlation analysis using Kendall's Tau [Kendall., 1938] for different split strategies.

#edges to split	edge choices	Mean	Std.
1	6	0.653	0.012
2	15	0.696	0.016
3	20	0.752	0.018

Each node denotes a latent state, e.g., feature maps in CNNs, and each edge represents a mixture of operations. We consider an architecture space with n nodes and m operations. Each node i is denoted as N_i where $i \in$ [1, n]; E_{ij} represents a set of m edges that connects node N_i and N_j, where m denotes the number of operations. Any architecture candidate that can be found in the space has only one edge in E_{ij}. In other words, there is exactly one operation from N_i to N_j in any architecture candidate. In addition, an available architecture at least has one edge from its predecessor node. For a search space that has n nodes, a full DAG contains $\frac{n(n-1)}{2}$ edges. Each edges has m choices from the operations, resulting in a total of $m^{n(n-1)/2}$ architectures.

To further investigate the choices of edge splitting to the architecture ranking, we calculate the Kendall's Tau (rank correlation) for splitting the same number but different choices of edges on the NASBENCH-201. We reuses the supernet from NASBENCH-201 that has 5 operation types. The supernet is a 4 nodes full DAG containing 6 independent compound edges, and each compound edges consists of the 5 parallel predefined operations. Therefore, there are 6 (C(1,6)), 15 (C(2,6)), and 20 (C(3,6)) choices to split 1, 2, and 3 edges, respectively. Each splitting of one compound edge results in 5 sub-supernets, and 5^k sub-supernets for splitting k compound edges. For example, splitting 1 compound edge results in 5 sub-supernets, but there are 6 choices on the selection of one edge to split. Here our goal is to investigate the impact of edges choices on the ranking prediction. Therefore, we trained all 6, 15, 20 edges choices for splitting 1, 2, 3 compound edges for the prediction, and present the results in Table. 6.2.

Table 6.2 shows the rank correlation after splitting by different numbers of edges. We have made the following observations: 1) Similar to the observation in Section 6.1, splitting more edges is a stronger signal to improve the rank correlation. 2) Given the same number of edges to split, the choice of splitting edges seems to have a limited impact on the rank correlation, e.g., the low standard deviation in Table. 6.2. Since this paper focuses on the insight that edge splitting on a supernet improves the rank prediction, we choose to use a random splitting strategy for now. While, the current random strategy does not rule out the possibility that a learned, task-specific strategy can do better. Or exploring the partial splitting, e.g., we split a compound edge with four independent edges into two separate compound edges, each with two independent edges. We leave these potential directions as future work. We also pre-define a training time budget T to curb the number of sub-supernets trained in the splitting process, and aborted immediately after exceeding the budget.

6.2.2 Transfer Learning

The number of sub-supernets still grows exponentially using the proposed splitting strategy, therefore the sub-supernets training can be really expensive after splitting a few compound edges, which is also against the key motivation of one-shot NAS. Here we describe how we use Transfer Learning to accelerate the training procedure of sub-supernets. Similar to how an architecture candidate $\mathcal{A}$ inherits weights $\mathcal{W}_{\mathcal{A}}$ from the supernet weights $\mathcal{W}_{\mathcal{S}}$, we allow a sub-supernet $\mathcal{S}^{\Omega'}$ to inherit weights from its parent sub-supernet. For example, in Figure 6.1(b), after training the supernet of Ω_A, the supernet of Ω_B and Ω_C can inherit the weights from shared operations in supernet of Ω_A as initialization and then start training. By using transfer learning, each sub-supernet only needs a few additional epochs to to be trained, which is far cheaper than training from scratch.

6.2.3 Integration with Gradient-based Algorithms

Gradient-based algorithms treat the NAS as a bi-level optimization problem where both the weight and architecture distribution parameters are optimized jointly at each training step [Liu et al., 2019]. To apply *few-shot NAS* to gradient-based algorithms, we start with training the supernet until it converges. Then, we split the supernet $\mathcal{S}$ to several sub-supernets following the random strategy in section 6.2.1. The weights parameters and architecture distribution parameters of each sub-supernet are subsequently initialized by inheriting from the corresponding parent supernet. We repeat the process until reaching the predefined search time budget. Lastly, we choose the sub-supernet $\mathcal{S}^{\Omega'}$ with the lowest validation loss from all the converged sub-supernets, and pick the best architecture $\mathcal{A}^*$ from the $\mathcal{S}^{\Omega'}$ based on the architecture distribution parameters.

6.2.4 Integration with Search-based Algorithms

Search-based algorithms query the performance of an architectures either by training as the case of vanilla NAS or by a pre-trained supernet as the case of one-shot NAS. The experience of this round, i.e. the performance of queried architecture, will be augmented into the memory to inform the decision of future sampling. The search repeats this process until an architecture with desired performance is found. To apply *few-shot*

NAS to search-based algorithms, we train a few sub-supernets using the randomly splitting and weight transfer, similar to the procedures described in section 6.2.3. Then we will uses the trained sub-supernets to estimate the performance of a sampled architectures. Please note each sub-supernet covers a disjoint region in the search space, so an architecture to a supernet maintains a one to one mapping. For example, if a sampled architecture $\mathcal{A}$ falls into sub-supernet $\mathcal{S}^{\Omega'}$, we will evaluate its performance $f(\mathcal{S}_{\mathcal{A}}^{\Omega'})$ by inheriting the weights $\mathcal{W}_{\mathcal{S}^{\Omega'}}$. In the end, we will pick the top performing architecture for fine-tuning.

6.3 Experiments

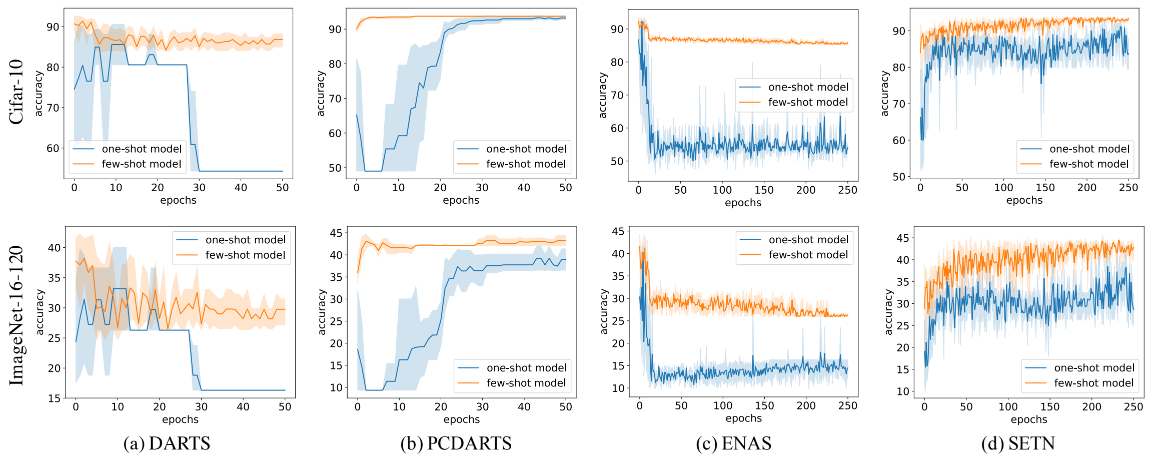


Figure 6.3: Comparison of various gradient-based algorithms in the one-shot and few-shot settings. We also plot the 75% interquartile range from 5 runs.

We first evaluate *few-shot NAS* on the impact of search performance by using different NAS algorithms. Here the evaluation metrics are search cost and accuracy. We use a variety of baselines, including DARTS, PCDARTS, ENAS, SETN, REA, REINFORCE, HB, BOHB, SMAC, and TPE [Liu et al., 2019, Xu et al., 2020, Pham et al., 2018b, Dong and Yang, 2019, Real et al., 2019b, Zoph et al., 2018a, Li et al., 2018a, Falkner et al., 2018, Hutter et al., 2011b, Bergstra et al., 2012] by directly comparing their one-shot/few-shot versions on NASBENCH-201. In addition, we also evaluate the few shot versions of DARTS, PCDARTS, and ENAS on NasBench1-shot-1 [Zela et al., 2020]. To validate *few-shot NAS* in practice, we also extend *few-shot NAS* to different open domain search problems, and show that the founded architectures by *few-shot NAS* outperform the architectures discovered by one-shot NAS by a significant margin. Particularly, the architectures searched by *few-shot NAS* reaches SoTA results on various tasks including CIFAR10, ImageNet, AutoGAN [Gong et al., 2019], and Penn Treebank.

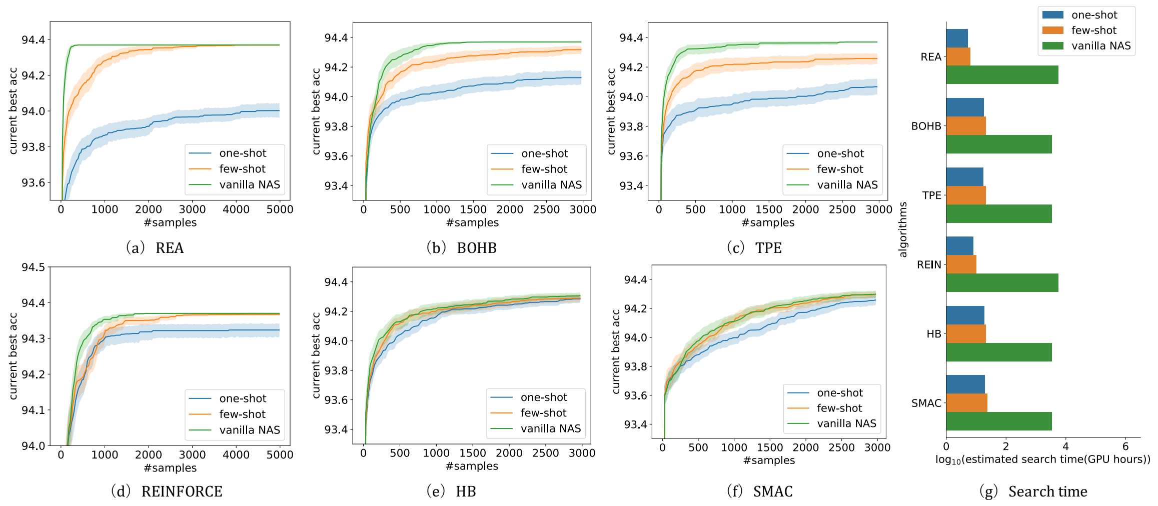


Figure 6.4: The progress of best accuracy during the search. *few-shot NAS* clearly demonstrates better results than one-shot NAS, while maintaining the similar end-to-end search time.

6.3.1 Evaluation on NasBench-201

NasBench-201 is a NAS dataset that contains 15625 architecture-accuracy pairs, along with re-implementations of several popular NAS algorithms [Dong and Yang, 2020]. The dataset allows us to thoroughly evaluate various NAS algorithms without training the actual networks.

Gradient-based Algorithms

The supernet defined for NasBench-201 has five nodes, and between two nodes there are five operations. Therefore, we will obtain 5 sub-supernets if fully splitting one compound edge that has 5 operations. For this experiment, we skip the transfer learning so that we can compare the performance of one-shot and *few-shot NAS* epoch by epoch. Here we only split one compound edge to show that even a few supernets can yield to the non-trivial improvement. Our baselines cover a list of prominent gradient-based search algorithms including DARTS and ENAS, and the following metrics are used in the evaluations: 1)the test accuracy of the final architecture found by a NAS algorithm; and 2) the search time that includes the supernet training and validation time.

Figure 6.3 shows the progress of test accuracy during the search. In training on CIFAR-10, the one-shot version of DARTS and ENAS demonstrate the bad performance, being consistent with other paper [Dong and Yang, 2020]. One potential reason is that one-shot NAS traps into a sub-optimal region due to inaccurate performance prediction. In contrast, *few-shot NAS* consistently finds a high quality of searched models

Table 6.3: Rank correlation on NASBENCH-201 using different methods.

Method	Kendall's Tau	Cost(Hours)
Random	0.0022	0
EN^{2}AS [Zhang et al., 2020]	0.378	N/A
One-shot	0.5436	6.8
Angle [Yiming Hu, 2020]	0.5748	N/A
Few-shot(5-supernets)	**0.653**	10.1
Few-shot(25-supernets)	**0.696**	18.6
Few-shot(125-supernets)	**0.752**	31.8

since multiple supernets improves the performance prediction to better guide the search than one-shot NAS. Although one-shot version of PCDARTS and SETN could find a good architecture with $> 90\%$ accuracy, it took nearly 10X more search epochs than our *few-shot NAS*. In short, we show that few-shot versions of various gradient-based algorithms consistently demonstrate better results than one-shot counterpart in terms of found architectures and the number of search epochs.

Search-based Algorithms

Being consistent with the evaluations above, we also split the first compound edge of supernet into five sub-supernets. But we used transfer learning described in section 6.2.2 for training sub-supernets this time since we can decouple the search and supernet training. Our baseline consists of six search-based algorithms that covers Bayesian optimization, evolutionary algorithm and reinforcement learning, including REA, REINFORCE, BOHB , HB, SMAC, and TPE. We report the performance of each search-based algorithms by averaging 50 independent runs with different random seeds. For details of integrating *few-shot NAS* with search methods, please refer to section 6.2.4. Two metrics is used in the evaluation. First, we denote i^{th} *best accuracy* as the best test accuracy after sampling i architectures. A good search algorithm is expected to yield a higher accuracy with fewer samples. The second metric is also the *total search time* to verify that the transfer learning helps in reducing the end-to-end time.

Result Analysis. Figure 6.4 shows the progress of best accuracy during the search. First, we observe that the few-shot verison of REA can find the global optimal in 3500 samples. Second, the few-shot versions of REA, BOHB, and TPE significantly improve the search efficiency over their one-shot counterparts, while the few-shot versions of REINFORCE, HB and SMAC are still better than one-shot versions. Figure 6.4(g) details the search time. All search-based algorithms took three to four orders of magnitude more GPU hours when using vanilla NAS than one-shot and *few-shot NAS*. *few-shot NAS* only incurs slightly more search time, roughly 10 GPU hours, than one shot NAS. Both one-shot and *few-shot NAS* complete the search within 24 hours. With a little more computing time, we do observe better rank correlation from *few-shot NAS* than other one-shot methods, which explains the good performance of *few-shot NAS* in the progress of search.

6.3.2 Deep Learning Applications

Table 6.4: Applying *few-shot NAS* on existing NAS methods on CIFAR-10 using the NASNet search space. Our results demonstrate that 1) *few-shot NAS* consistently improves the final accuracy of various one-shot based NAS methods under the same setup. Please note we only extend one-shot based DARTS, REA, and LaNAS by replacing the single supernet with 7 supernets in their public release; 2) after integrating with multiple supernets, few-shot DARTS achieves SOTA 98.72% top-1 accuracy on CIFAR-10 using the cutout [Devries and Taylor, 2017] and auto-augmentation [Cubuk et al., 2018]. Without auto-augmentation, few-shot DARTS-Small still consistently outperforms existing models that have similar parameters.

Method	Data Augmentation	#Params	Err	GPU days
NASNet-A [Zoph et al., 2018a]	cutout	3.3M	2.65	2000
AmoebaNet-B-small [Real et al., 2019b]	cutout	2.8M	2.50 ± 0.05	3150
AmoebaNet-B-large [Real et al., 2019b]	cutout	34.9M	2.13 ± 0.04	3150
AlphaX [Wang et al., 2019c]	cutout	2.83M	2.54 ± 0.06	1000
NAO [Luo et al., 2018b]	cutout	3.2M	3.14 ± 0.09	225
DARTS [Liu et al., 2019]	cutout	3.3M	2.76 ± 0.09	1
P-DARTS [Chen et al., 2019a]	cutout	3.4M	2.5	0.3
PC-DARTS [Xu et al., 2020]	cutout	3.6M	2.57 ± 0.07	0.3
Fair-DARTS [Chu et al., 2019b]	cutout	3.32M	2.54 ± 0.05	3
BayeNAS [Zhou et al., 2019a]	cutout	3.4M	2.81 ± 0.04	0.2
CNAS [Lim et al., 2020]	cutout	3.7M	2.60 ± 0.06	0.3
MergeNAS [Wang et al., 2020c]	cutout	2.9M	2.68 ± 0.01	0.6
ASNG-NAS [Akimoto et al., 2019a]	cutout	3.32M	2.54 ± 0.05	0.11
XNAS [Nayman et al., 2019b]	cutout + autoaug	3.7M	1.81	0.3
one-shot REA	cutout + autoaug	3.5M	2.02 ± 0.03	0.75
one-shot LaNas [Wang et al., 2019b]	cutout + autoaug	3.6M	1.68 ± 0.06	3
few-shot DARTS-Small	cutout	3.8M	$\mathbf{2.31\pm0.08}$	1.35
few-shot DARTS-Large	cutout	45.5M	$\mathbf{1.92\pm0.08}$	1.35
few-shot DARTS-Small	cutout + autoaug	3.8M	$\mathbf{1.70\pm0.08}$	1.35
few-shot DARTS-Large	cutout + autoaug	45.5M	$\mathbf{1.28\pm0.08}$	1.35
few-shot REA	cutout + autoaug	3.7M	$\mathbf{1.81\pm0.05}$	0.87
few-shot LaNas	cutout + autoaug	3.2M	$\mathbf{1.58\pm0.04}$	3.8

CIFAR-10 in Practice. We implement few shot versions of one gradient-based algorithm (DARTs) and two search-based algorithms including regularized evolution (REA) and LaNas [Liu et al., 2019, Real et al., 2019b, Wang et al., 2019b] in the evaluations. Their results are compared to several recent NAS algorithms listed in Table 6.4. We see that the few-shot version of DARTS outperforms the one-shot version by 0.43% in the test accuracy. Further, the best architecture reported from *few-shot NAS* is also the state-of-the-art results

Table 6.5: Applying *few-shot NAS* on existing NAS methods on ImageNet using the EfficientNet search space. Being consistent with the results on CIFAR-10 in Table. 6.4, the final accuracy from few-shot OFA and ProxylessNAS also outperforms their original one-shot version under the same setting, except for replacing the single supernet with 5 supernets. Particularly, Few-shot OFA-Large achieves SoTA 80.5% top1 accuracy at 600M FLOPS.

Method	Space	#Params	#FLOPs	Top 1 Acc(%)	GPU hours
AutoSlim [Yu and Huang, 2019]	Mobile	5.7M	305M	74.2	N/A
MobileNetV3-Large [Howard et al., 2019a]	Mobile	5.4M	219M	74.7	N/A
MnasNet-A2 [Tan et al., 2019]	Mobile	4.8M	340M	75.6	N/A
FBNetV2-L1 [Wan et al., 2020b]	Mobile	N/A	325M	77.2	600
EfficientNetB0 [Tan and Le, 2019b]	Mobile	5.3M	390M	77.3	N/A
AtomNAS [Mei et al., 2020]	Mobile	5.9M	363M	77.6	N/A
few-shot OFA_Net-Small	Mobile	5.6M	238M	**77.50**	68
MobileNetV2 [Sandler et al., 2018]	Mobile	6.9M	585M	74.7	N/A
ShuffleNet-V2 [Ma et al., 2018]	Mobile	N/A	590M	74.9	N/A
ProxylessNAS [Cai et al., 2019b]	Mobile	7.12M	465M	75.1	200
ChamNet [Dai et al., 2019]	Mobile	N/A	553M	75.4	N/A
RegNet [Radosavovic et al., 2020]	Mobile	6.1M	600M	75.5	N/A
OFA_Net [Cai et al., 2020]	Mobile	9.1M	595M	80.0	40
few-shot ProxylessNAS	Mobile	4.87M	521M	**75.91**	280
few-shot OFA_Net-Large	Mobile	9.2M	600M	**80.50**	68

on CIFAR-10. Although we used 7 supernets in this experiment, *few-shot NAS* only incurred 35% more time than the one shot NAS. Similarly, *few-shot NAS* also improved the search efficiency of one-shot REA, finding an architecture with 0.21 lower error using only 16.7% more time. Few-shot version of LaNas also decrease the test error from 1.68 to 1.58 in the one shot version using only 26.7% extra search time. All of our few-shot results outperform the one-shot results in the table by using the exactly same setup.

Neural Architecture Search on ImageNet. We selected ProxylessNAS and Once-for-All NAS(OFA) [Cai et al., 2019b, Cai et al., 2020] in this evaluation. Table 6.5 show the final result. Few-shot NAS significantly improves the accuracy on two NAS algorithms that uses one-shot model.

6.4 Related Work

Weight-sharing supernet was first proposed as a way to reduce the computational cost of NAS [Pham et al., 2018b]. Centering around supernet, a number of NAS algorithms including gradient-based [Liu et al., 2019, Xu et al., 2020, Dong and Yang, 2019] and search-based [Bender et al., 2018, Chu et al., 2019a, Guo et al., 2019b] were proposed. The search efficiency of these algorithms is dependent on the ability of supernet to approximate architecture performance. To improve the supernet approximation accuracy, Bender et al. [Bender et al., 2018] proposed a path dropout strategy that randomly drops out weights of the supernet during training. This approach improves the correlation between one-shot NAS and individual architecture accuracy by reducing weight co-adaptation. In a similar vein, Guo et al. [Guo et al., 2019b] proposed a single-path one-shot training by only activating the weights from one randomly picked architecture in forward and backward propagation. Additionally, Yu et al. [Yu et al., 2019a] found that training setup greatly impacts supernet performance and identified useful parameters and hyper-parameters. Lastly, an angle-based approach [Zhiyuan Li, 2020, Arora et al., 2019, Carbonnelle and Vleeschouwer, 2018] was proposed to improve the supernet approximation accuracy for individual architecture [Yiming Hu, 2020] and was shown to improve the architecture rank correlation. However, our few-shot models achieved better rank correlation than this angle-based approach(see table 6.3). Our work focuses on reducing the supernet approximation error by dividing the supernet to a few sub-supernets to eliminate the co-adaption among supernet operations. As such, our work is complementary and can be integrated into the aforementioned work.

6.5 Conclusion

Recently, one-shot NAS substantially reduces the computation cost by training only one supernetwork, a.k.a. *supernet*, to approximate the performance of every architecture in the search space via weight-sharing. However, the performance estimation can be very inaccurate due to the co-adaption among operations [Bender et al., 2018]. In this chapter, we propose *few-shot NAS* that uses multiple supernetworks, called *sub-supernet*, each covering different regions of the search space to alleviate the undesired co-adaption. Compared to one-shot NAS, few-shot NAS improves the performance prediction with a small increase of evaluation cost. With only up to 7 sub-supernets, few-shot NAS establishes new SoTAs: on ImageNet, it finds models that reach 80.5 top-1 at 600 MB FLOPS and 77.5 top-1 at 238 MFLOPS; on CIFAR10, it reaches 98.72 top-1 without using extra data or transfer learning.

Chapter 7

Efficient Distributed Training via Gradient Sparsification

7.1 Introduction

The performance and efficiency of distributed training of Deep Neural Networks (DNN) highly depend on the performance of gradient averaging among participating processes, a step bound by communication costs. There are two major approaches to reduce communication overhead: overlap communications with computations (lossless), or reduce communications (lossy). The lossless solution works well for linear neural architectures, e.g. VGG, AlexNet, but more recent networks such as ResNet and Inception limit the opportunity for such overlapping. Therefore, approaches that reduce the amount of data (lossy) become more suitable. In this paper, we present a novel, explainable lossy method that sparsifies gradients in the frequency domain, in addition to a new range-based float point representation to quantize and further compress gradients. These dynamic techniques strike a balance between compression ratio, accuracy, and computational overhead, and are optimized to maximize performance in heterogeneous environments.

Unlike existing works that strive for a higher compression ratio, we stress the robustness of our methods, and provide guidance to recover accuracy from failures. To achieve this, we prove how the FFT sparsification affects the convergence and accuracy, and show that our method is guaranteed to converge using a diminishing θ in training. Reducing θ can also be used to recover accuracy from the failure. Compared to STOA lossy methods, e.g., QSGD, TernGrad, and Top-k sparsification, our approach incurs less approximation error, thereby better in both the wall-time and accuracy. On an 8 GPUs, InfiniBand interconnected cluster, our techniques effectively accelerate AlexNet training up to 2.26x to the baseline of no compression, and 1.31x to QSGD, 1.25x to Terngrad and 1.47x to Top-K sparsification.

Parameter Server (PS) and allreduce-style communications are two core parallelization strategies for distributed DNN training. In an iteration, each worker produces a gradient, and both parallelization strategies rely on the communication network to average the gradients across all workers. The gradient size of current DNNs is at the scale of 10^2 MB, and, even with the state-of-the-art networks such as Infiniband, repeatedly transferring such a large volume of messages over millions of iterations is prohibitively expensive. Furthermore, the tremendous improvement in GPU computing and memory speeds (e.g., the latest NVIDIA TESLA V100 GPU features a peak performance of 14 TFlops on single-precision and memory bandwidth of 900 GB/s with HBM2) further underscores communication as a bottleneck.

Recently, several methods have shown that training can be done with a lossy gradient due to the iterative nature of Stochastic Gradient Descent (SGD). It opens up new opportunities to alleviate the communication overhead by aggressively compressing gradients. One approach to compress the gradients is *quantization*. For example, Terngrad [Wen et al., 2017] maps a gradient into [-1, 0, 1], and QSGD [Alistarh et al., 2017] stochastically quantizes gradients onto a uniformly discretized set larger than that of Terngrad. Such coarse approximation not only incurs large errors between the actual and quantized gradients as we demonstrate in Figure 7.15 [QSGD, TernGrad], but also fails to exploit the bit efficiency in the quantization (Figure 7.7). Another approach to gradient compression, *sparsification*, only keeps the top-k largest gradients [Han et al., 2015, Aji and Heafield, 2017, Alistarh et al., 2018]. Similarly, Top-k loses a significant amount of actual gradients around zeros to achieve a high compression ratio (Figure 7.15, [Top-k]). In summary, existing lossy methods greatly drop gradients, incur large approximation errors (Figure 7.15e), leading to the deterioration of the final accuracy (Table 7.2). To avoid compromising the convergence speed, both *quantization* and *sparsification* must limit the compression ratio, leading to sub-optimal improvement of the end-to-end training wall time.

In this paper, we propose a gradient compression framework that takes advantages of both $sparsification$ and $quantization$ with two novel components, FFT-based sparsification, and a range-based quantization. FFT-based sparsification allows removing the redundant information, while preserving the most relevant information (Figure 7.15 [FFT]). As a result, FFT incurs fewer errors in approximating the actual gradients (Figure 7.15e), thereby better in accuracy than QSGD, TernGrad, and Top-K (Table 7.2). We treat the gradient as a 1D signal, and drop near-zero coefficients in the frequency domain, after an FFT. Deleting some frequency components after the FFT introduces magnitude errors, but the signal maintains its distribution (Figure 7.5). As a result, the sparsification in the frequency domain can achieve the same compression ratio as in the spatial domain but preserving more relevant information.

To further improve the end-to-end training wall time, we introduce a new range-based variable precision floating point representation to quantize and compress the gradient frequencies after sparsification. Most importantly, unlike the uniform quantization used in existing approaches, the precision of representable floats in our method can be adjusted to follow the distribution of the original gradients (Figure 7.9). The novel range-based design allows us to fully exploit the precision given limited bits so that the approximation error can be further reduced. By combining *sparsification* and *quantization*, our framework delivers a higher compression ratio than the single method, resulting in shorter end-to-end training wall time than QSGD, Terngrad, and Top-k.

Lastly, our compression framework is highly efficient and scalable. The primitive algorithms in our compression scheme, such as FFT, top-k select, and precision conversions, are efficiently parallelizable and thus GPU-friendly. We resort to existing highly optimized GPU libraries such as cuFFT, Thrust, and bucketSelect [Alabi et al., 2012], while we propose a simple yet efficient packing algorithm to transform sparse gradients into a dense representation. Minimizing the computational cost of the compression is crucial for high-speed networks, such as Infiniband networks, as we analyzed in Figure 7.10.

Specifically, the contributions of this paper are as follows:

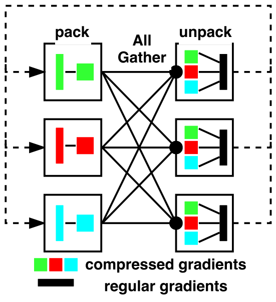
1 A novel FFT-based, tunable gradient sparsification that retains the original gradient distribution.

2 A novel range-based variable precision floating-point that allocates precision according to the gradient distribution.

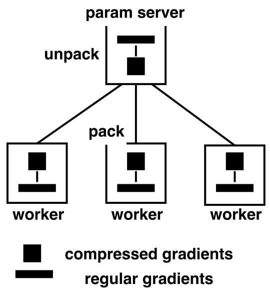
3 An analytic model to guide people when to enable compression and how to set a compression ratio according to hardware specifications.

4 The convergence proof of our methods, and its guidance in selecting a compression ratio θ, to ensure the convergence, or reduce θ to recover the accuracy. To the best of our knowledge, this paper is the first one to discuss the relationship between compression ratio and accuracy of neural networks.

5 Highly optimized system components for a compression framework that achieves high throughput on GPUs and is beneficial even on state-of-the-art Infiniband networks.



(a) BSP



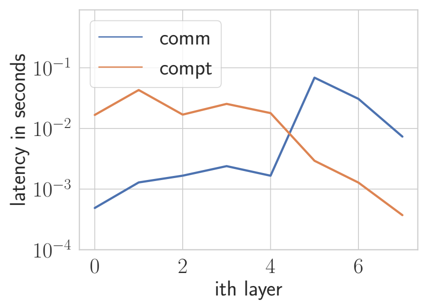
(b) PS

Figure 7.1: Two parallelization schemes of distributed DNN training:(a) Bulk Synchronous Parallel (BSP) strictly synchronizes gradients with all-to-all group communications, e.g. MPI collectives; (b) Parameter Server (PS) exchanges gradients with point-to-point communications, e.g. push/pull.

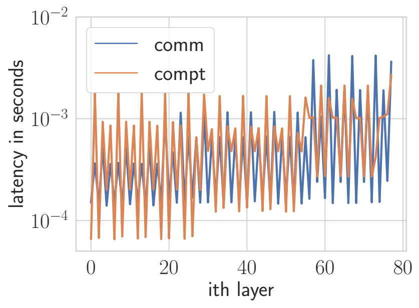
7.2 Background and Motivation

In general, there are two strategies to parallelize DNN training: *Model Parallelism* and *Data Parallelism*. *Model Parallelism* splits a network into several parts, with each being assigned to a computing node [Dean et al., 2012]. It demands extensive intra-DNN communications in addition to gradient exchanges. It largely restricts the training performance, and thereby *Model Parallelism* is often applied in scenarios where the DNN cannot fit onto a computing node [Dean et al., 2012]. The second approach, *Data Parallelism* [Wang et al., 2017a], partitions the image batch, and every computing node holds a replica of the network. In a training iteration, a node computes a sub-gradient with a batch partition. Then, nodes *all-reduce* sub-gradients to reconstruct the global one. The only communications are for necessary gradient exchanges. Therefore, current Deep Learning (DL) frameworks such as SuperNeurons [Wang et al., 2018b], MXNet [Chen et al., 2015], Caffe [Jia et al., 2014], and TensorFlow [Abadi et al., 2016] parallelize the training with *Data Parallelism* for the high-performance.

There are two common strategies to organize the communications with data parallelism: with a centralized Parameter Server (PS) (Figure 7.1b), or with all-to-all group communications, e.g., (Figure 7.1a). TensorFlow [Abadi et al., 2016], MXNet [Chen et al., 2015], and PaddlePaddle implement distributed DNN training with a Parameter Server (PS) [Li et al., 2014]. In this distributed framework, the parameter server centralizes the parameter updates, while workers focus on computing gradients. Each worker pushes newly computed gradients to the parameter server, and the parameter server updates parameters before sending the latest parameters back to workers. Though this client-server [Berson, 1992] style design easily supports fault tolerance and elastic scalability, the major downside is the network congestion on the server. Alternatively,



(a) AlexNet



(b) ResNet32

Figure 7.2: layer-wise communications (all-reduce) v.s. computations in an iteration of BSP SGD using 16 P100 (4 GPUs/node with 56Gbps FDR).

-based Bulk Synchronous Parallel SGD can better exploit the bandwidth of a high-speed, dense interconnects, such as modern Infiniband networks. Instead of using a star topology, pipelines the message exchanges at a fine granularity with adjacent neighbors in a ring-based topology. Since the pipeline fully utilizes the inbound and outbound link of every computing node, it maximizes network bandwidth utilization and achieves appealing scalability where the cost is largely independent of the number of computing nodes.

There are trade-offs between the BSP and PS schemes, with PS having better fault tolerance, and better exploits the network bandwidth. However, as we argue below, in both cases, the communication cost is high, and reducing it can yield substantial gains in training latency.

7.2.1 Communication Challenges in Distributed Training of DNNs

Communications for averaging sub-gradients is widely recognized as a major bottleneck in scaling DNN training[Zhao et al., 2017, Dean et al., 2012, Wang et al., 2017a]. With increasing data complexity and volume, and with emerging non-linear neural architectures, two critical issues exacerbate the impact of communications in the scalability and efficiency of distributed DNN training with data parallelism: I) *the increasing amounts of data to be exchanged*, and II) *the decreasing opportunity to overlap computation and communication*.

Challenge I: Enormous amounts of communications during training. DNNs are extremely effective at modeling complex nonlinearities thanks to the representation power of millions of parameters. The number of parameters dictates the size of the gradients. Specifically, the gradient sizes of AlexNet, VGG16, ResNet32,

and Inception-V4 are 250MB, 553MB, 102MB, and 170MB. Even with the highly optimized allreduce implementation on a 56 Gbps FDR network, communication overhead remains significant. For example, the communication for AlexNet, VGG16, Inception-V4 and ResNet32 at regular single-GPU batch sizes[1] consumes 64.17%,
$18.62\%, 33.07\%$ and 43.96% of an iteration time, respectively.

Challenge II: Decreasing opportunity to overlap computation and communication. One promising solution to alleviate the communication overhead is hiding the communication for averaging the gradient of the i^{th} layer by the computation of $i-1^{th}$ layer in the backward pass. This lossless technique has proven to be effective on linear networks such as AlexNet and VGG16 [Awan et al., 2017, Rhu et al., 2016], as these networks utilize large convolution kernels to process input data. Figure 7.2a demonstrates the computation time of the convolution layers is $10\times$ larger than the communication time, easy for overlapping. However, the overlapping technique is not always applicable for two reasons. $First$, the degree of overlapping is largely decided by the computation pattern of the neural network model. The opportunity for computation and communication overlap is very limited in recent neural architectures, such as Inception-V4 [Szegedy et al., 2017] and ResNet [He et al., 2016]. The sparse fan-out connections in the Inception Unit (Figure 1a in [Wang et al., 2018b]) replace one large convolution (e.g. 11×11 convolution kernel in AlexNet) with several small convolutions (e.g. 3×3 convolution kernels). Similarly, ResNet utilizes either 1×1 or 3×3 small convolution kernels. As a result, the layer-wise computational cost of ResNet is similar to or smaller than communication (Figure 7.2b); hence, it is much harder to overlap these neural networks than AlexNet. $Second$, the degree of overlapping is also impacted by the bandwidth of networks. With slower networks, there are less opportunity to overlap communications and computations. Specifically, as seen in Figure 7.2a, the computation cost of convolution layers of the AlexNet is $10\times$ larger than the communication cost with 56Gbps InfiniBand. However, when training AlexNet in a low profile network such as 1Gbps Ethernet, it becomes impossible to hide the communication cost as it is significantly larger than the computation cost.

These two challenges – increasing data exchanged, and decreasing opportunity to hide communication latency – make it attractive to look for solutions that minimize the communication cost by decreasing the communication volume. Training a neural network with imprecise gradient updates still works as parameters are iteratively refined [Aji and Heafield, 2017]. Particularly, lossy gradient compression can achieve higher compression rates and still allow the network to deliver target accuracy [Alistarh et al., 2017]. Given this, it is not surprising that several gradient compression approaches have been proposed in the literature. They

[1]the single GPU batch size for AlexNet is 64, and 16 for others.

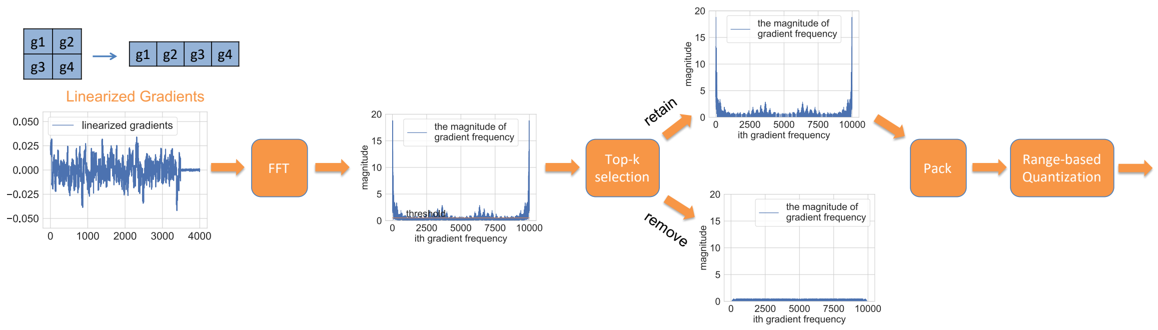


Figure 7.3: The gradient compression framework (sender).

generally fall into two categories: quantization of the gradients ([Seide et al., 2014, Wen et al., 2017, Alistarh et al., 2017, De Sa et al., 2015]), where these are represented with lower precision numbers, and sparsification ([Aji and Heafield, 2017, Alistarh et al., 2018, Wangni et al., 2018]), where small gradient are treated as zero and not transmitted. We discuss these approaches in detail in Section 7.5. As we describe next, we propose a novel gradient compression scheme that uses adaptive quantization and tunable FFT-based gradient compression that, together, achieve variable compression ratios that can maintain convergence quality, and, critically, is cheap enough computationally to be beneficial.

7.3 Methodology

7.3.1 The Compression Framework

Figure 7.3 provides a step-by-step illustration of our compression pipeline.

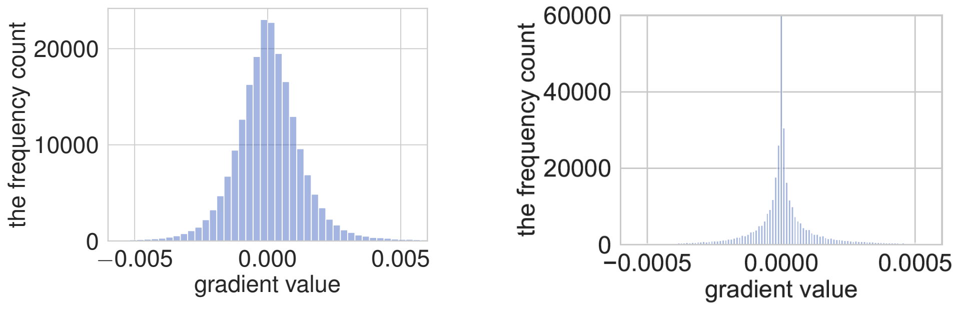
1 Linearize the gradients by re-arranging gradient tensors into a 1-d vector for Fast Fourier Transform (**FFT**), which is discussed in Section 7.3.1.

2 Truncate the gradient frequencies based on their magnitudes to sift out the **top-k** low-energy frequency components, which is discussed in Section 7.3.1.

3 Transform the frequencies' representation from 32-bit float to a new, **range-based**, N-bit float ($N < 32$) to further compress down the gradient frequency, which is discussed in Section 7.3.2.

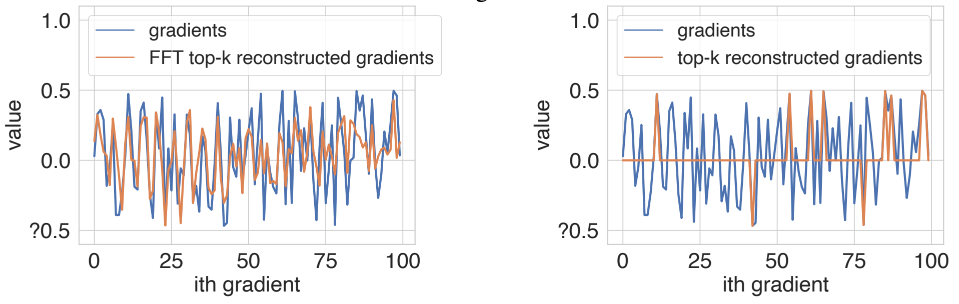
4 **Pack** sparse data into dense vector and transfer them out, which is discussed in Section 7.3.1.

On the receiver side, a similar approach (but using the inverse operations in the reverse order) is used to decompress the gradient frequency vector into gradients. Detailed discussions of compression components



(a) CIFAR-10, ResNet (b) ImageNet, AlexNet

Figure 7.4: Histogram of DNN gradients: we sampled gradients every 10^3 and 10^4 iterations in a full training.



(a) FFT Top-k (b) Top-k

Figure 7.5: FFT Top-k v.s. direct Top-k sparsificaiton: Top-k aggressively loses gradients (err=0.0246), while FFT preserves more relevant information (err=0.0209) at the same sparsification ratio.

and their motivations are as follows.

Removing redundant information with FFT based Top-K sparsification

Motivation: the gradient points to a descent direction in the high dimensional space, thereby small perturbations on gradients can be viewed as introducing local deviations along the descent direction. If such deviations are limited during the training, these imprecise descent directions still iteratively lead to a local optimum at the cost of additional iterations. This is the intuition for the gradient sparsification. Besides, Figure 7.4 indicates high redundancy in DNN gradients due to a lot of near-zero components, that may have limited contributions in updating gradients. Recently, several top-k based methods [Han et al., 2015, Aji and Heafield, 2017, Alistarh et al., 2018] have also shown the possibility to train DNNs with only the top 10% largest gradients. However, the resulting gradients, as shown in Figure 7.5, significantly deviate from the original, for entirely dropping the gradients below the threshold. This has motivated us to sparsify gradients, instead, in the frequency domain for preserving the trend of the original signal even after removing the same amount of information. For a gradient vector of length N, each gradients is $g_i = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}$ after FFT. If we sparsify on x_n, i.e. $g_i = \sum_{n}^{top_k} x_n e^{\frac{-i2\pi kn}{N}}$, g_i still preserves some of the original gradient information.

Therefore, FFT based top-k shows better results than top-k in Figure 7.5. More validations are available in the experimental section.

Our approach: The detailed computation steps of our FFT sparsification are highlighted in Figure 7.3. Recent generations of NVIDIA GPUs support mixed-precision; and computing with half-precision increases the FFT throughput up to $2\times$. So, we convert 32-bit (full-precision) gradients into 16-bit (half-precision) gradients to improve the throughput before applying FFT, and the information loss from the conversion is negligible due to the bounded gradients.

After FFT, the next step is to filter the low energy gradient in the frequency domain. We introduce a new hyper-parameter, θ, to regulate the sparsity of frequencies. Here, we only describe the procedures, and the tuning of θ is thoroughly discussed in Section 3 and experiments. If $\theta = 0.9$, we keep the top 10% frequency components in magnitude and drop the rest by resetting to zeros (Figure 7.3). The selection is implemented with either sorting or Top-k. Since Thrust[2] and cuFFT[3] provide highly optimized FFT and sorting kernels for the GPU architecture, we adopted them in our implementations.

7.3.2 Packing sparse data into a dense vector

Thresholding gradient frequencies in the last step yields a highly irregular sparse vector, and we need to pack it into a dense vector to reduce communications. The speed of packing a sparse vector is critical to the practical performance gain. Here, we propose a simple parallel packing algorithm:

1 Create a $status$ vector and mark an element in $status$ as 1 if the corresponding scalar in $sparse$ vector is non-zero (e.g., $sparse = [a, 0, b, 0, c, 0, 0]$ and $status = [1, 0, 1, 0, 1, 0, 0]$).

2 Perform a parallel prefix-sum on $status$ to generate a $location$ vector ($[1, 1, 2, 2, 3, 3, 3]$).

3 if $status[i] == 1$, write $sparse[i]$ to $dense[location[i]]$, and $dense$ vector is the packed result.

This parallel algorithm has a $689\times$ speedup over the single-threaded algorithm on a TESLA V100 with a throughput of 34 GB/s.

We need to send the status vector and the compressed gradient to perform the decompression. The status vector is a bitmap that tracks the location of non-zero elements, and its length in bits is the same as the gradient vector. Figure 7.6 shows the cost of the status vector is non-negligible after the compression ratio exceeding 20. Therefore, setting $\theta < 0.05$ is not desired.

[2]`https://developer.nvidia.com/thrust`

[3]`https://developer.nvidia.com/cufft`

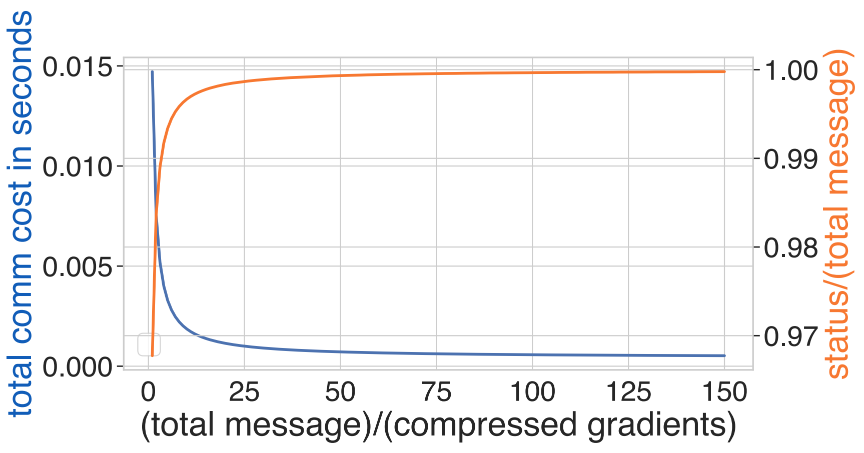


Figure 7.6: the effect of status vector: given 100 MB gradients, the improvement after dropping $> 95\%$ gradients ($\theta = 0.05$, compression ration is 20) is limited.

Figure 7.7: comparisons of quantization schemes: the uniform distribution and IEEE 754 format.

Range based Quantization

Motivation: the range of single precision IEEE-754 floating point is $[-3.4 * 10^{38}, +3.4 * 10^{38}]$, while the range of gradients and their frequencies are much smaller (e.g. [-1, +1]). This motivates us to represent the bounded gradients with fewer bits. The problem of using an N bits IEEE 754 format, as seen in Figure 7.7, is the inconsistency between the range of gradients [min, max] and the range of the IEEE representable numbers. Given N bits for IEEE 754, there are $N-2$ combinations of exponent-mantissa. The representation range is either too large or too small for gradients, regardless of which combinations to choose. Another conventional way is to equally divide the $max - min$ into 2^N, i.e., uniform quantization. Still, the actual gradient distribution is far from the uniform, and thereby it is also inefficient, as shown in Figure 7.7.

Our approach: we propose an offset-based N-bit floating point, which intends to match the distribution of representable numbers to the real gradients. Our representation is to use the N-bit binary format of a positive number as base number $pbase$, and encode it to 0...01. The rest positive numbers are encoded as 0...01 ($pbase$) + offset. The negative numbers also follow the same rule. Therefore, the total 2^N representable numbers consist of P positive numbers and $2^N - P$ negative numbers. To match the range of real gradients, our quantization permits the manual setting of a representation range, defined by min and max. We estimate min and max from the first few iterations of gradients. Then, we tune m and eps to adjust the precision of representable numbers, as shown in Figure 7.7. m represents the number of bits left for the mantissa, and eps represents the minimal representable positive number whose corresponding N-bit binary is $pbase$. The

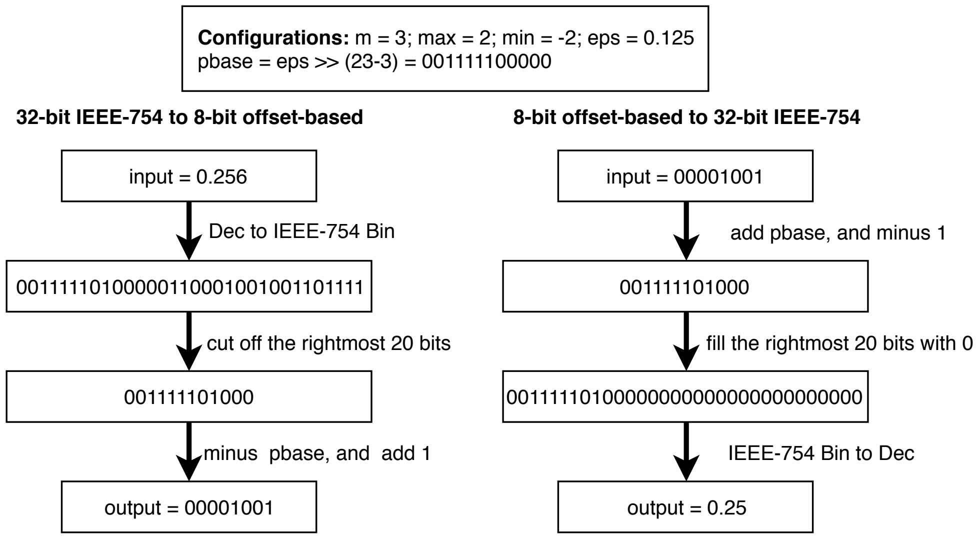


Figure 7.8: *Illustration of range based quantizer*: an example conversion of between 32 bits IEEE 754 and 8 bits our representation.

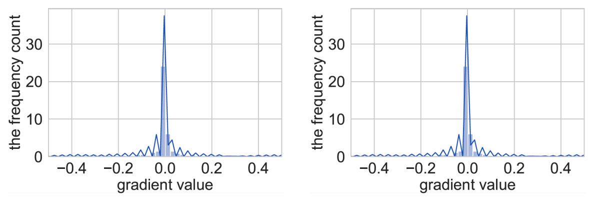
following further explains how m and eps adjust the precision:

- m: let's denote the difference between two consecutive numbers as $diff$. For m bits mantissa, the exponent increases by 1 after 2^m number, and increasing $diff = diff * 2$. Since $diff$ is exponentially growing, this creates a Gaussian like representation range that matches to real gradients. If max, min and eps are fixed, P is small for a small m, as it takes fewer numbers to increase the exponent. Similarly, a large m leads to a larger P. Therefore, m is very sensitive for precision.

- eps: with max, min and m, $diff$ is also fixed. If eps is small, it takes more steps to reach max yielding a large P; and vice versa.

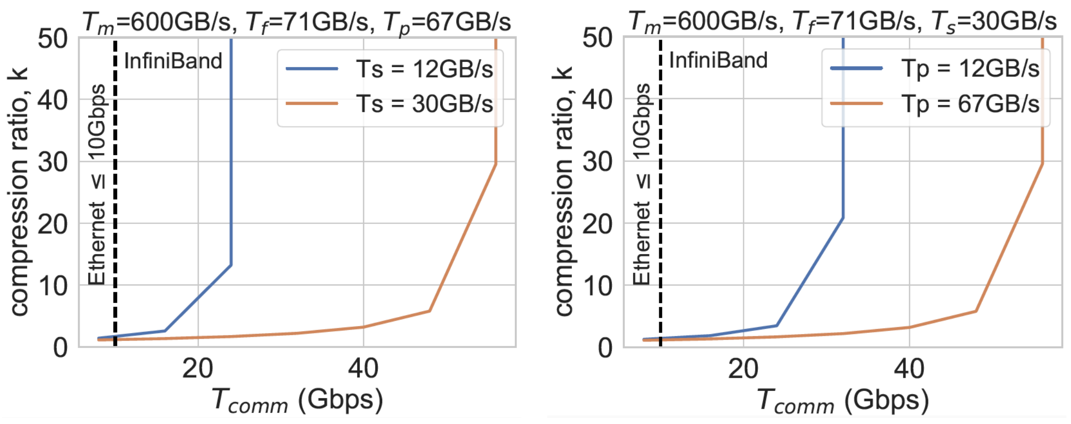
Since m and eps determine P, we need to tune them to make P close to $2^N/2$ for balancing the range of positive and negative numbers. In practice, N, min, and max are empirically decided from gradients, and the $m \in [1, N]$. We iterate every m to tune for eps. Given N, m, min, and max, we initialize eps as a reasonably small number, e.g., 0.002, then de-compress the 1..1 (the minimal representable negative number) back to FP32 with the selected eps, and the resulting number is the current actual minimal negative number $actual\_min$; if $actual\_min$ is smaller than min, we decrease eps, and increase otherwise. Following this path, P converges to $2^N/2$, a state with equal positive and negative numbers, and yielding the optimal eps.

```
Input: init(min, max)
pbase_binary = eps ¿¿ (23-m)   Input: 32bit_to_Nbit(32bit_float)
if 32bit_float ¿ max then
    32bit_float = max;
```



(a) (-0.5, 0.5)   (b) (-5, 5)

Figure 7.9: Adjustable representation range: our quantization successfully adjusts its distribution.



(a) Top-k selection   (b) Packing

Figure 7.10: Minimal compression ratio k exhibits performance benefits at different network bandwidths T_{comm}, packing throughput T_p and selection throughput T_s. It is easy to get performance improvement from a slow network, while it requires faster compression primitives to be beneficial on a fast network.

32bit_binary = 32bit_float ¿¿ (23-m)   Nbit_binary = 32bit_binary - pbase_binary + 1   **Input:** Nbit_to_32bit(Nbit_binary)

32bit_binary = Nbit_binary + pbase_binary - 1   32bit_float = 32bit_binary ¡¡ (23-m)

Alg. **??** summarizes the conversion from 32-bit IEEE 754 to our N-bit offset based float, and N is set w.r.t the precision requirement for the training. Figure 7.8 provides a step-by-step conversion between IEEE 754 and our 8 bits representation.

Figure 7.9 shows the resulting number distributions of our approach when the range is set to [-0.5, 0.5], and [-5, 5]. This shows our approach successfully adjusts representation ranges, while still maintaining similar distribution to actual gradients. This is because $diff$ increases 2x after 2^m numbers, leading to more numbers around 0, and less to max or min. Unlike prior static approach, our offset based float dynamically changes the representable range to sustain the various precision requirements from different training tasks. Besides, the float quantizations are embarrassingly data-parallel, so it is easy to achieve the high-performance.

Symbol	Explanation
T_m	Maximum throughput of precision conversion including float-to-half and range-based quantization
T_f	Maximum throughput of FFT
T_p	Maximum throughput of packing
T_s	Maximum throughput of top-k selection
T_{comm}	Maximum throughput of communication via networks
k	Overall compression ratio

Table 7.1: Symbols of equations in Section 7.3.3.

7.3.3 Sensitivity Analysis

The compression cost shall not offset the compression benefit to acquire practical performance gain. In this section, we analyze the performance of compression primitives and their impact on perceived network bandwidth. Table 7.1 defines all symbols used in the analysis. It is noted that we use the same notation T_m for both float-to-half and range-based quantization as they are O(N) algorithms and embarrassingly parallel. Given a message of size M, the cost of compression is:

$$cost_{comp} = M(\frac{2}{T_m} + \frac{1}{T_f} + \frac{1}{T_p} + \frac{1}{T_s}) \tag{7.1}$$

The communication cost after compression is :

$$cost_{comm} = \frac{M}{T_{comm}}(\frac{1}{k}) \tag{7.2}$$

So the communication cost saved by compression is:

$$saved\_cost_{comm} = \frac{M}{T_{comm}}(1 - \frac{1}{k}) \tag{7.3}$$

To compensate for the cost of compression and decompression, $2cost_{comp} < saved\_cost_{comm}$ must hold to acquire the practical performance gain, that is

$$k > \frac{1}{1 - 2T_{comm}(\frac{2}{T_m} + \frac{1}{T_f} + \frac{1}{T_p} + \frac{1}{T_s})} \tag{7.4}$$

The performance of T_m depends on the hardware characteristics (such as GPU DRAM bandwidth), and T_f depends on cuFFT. It is therefore reasonable to consider them fixed for a particular GPU hardware. T_s and

T_p depend on the libraries and algorithms applied. By varying T_s and T_p in Equation 7.4 we analyze the minimal compression ratio k that will show benefits for a particular network infrastructure. Figure 7.10 shows the relationship between k and T_{comm}. If the network throughput is low, like Ethernet, a small k could compensate for the cost of compression and decompression, which means increasing k would significantly boost the performance of communications. For example, Figure 7.10 shows that $k = 2$ is enough to compensate for the overhead of compression and decompression on a 10Gbps Ethernet. One the other hand, if the network throughput is high, like InfiniBand, a larger k would be necessary; otherwise, the overall performance will be impacted by the overhead of compression and decompression. More precisely, the red line in Figure 7.10 indicates that the minimal compression ratio k should be about 30 to exhibit any benefit on a 56Gbps InfiniBand.

Figure 7.10 also predicts that the performance of the compression primitives is crucial for high bandwidth networks. As seen in Figure 7.10.a, if T_s is 12GB/s, for any T_comp larger than $22Gbps$ no compression ratio will be able to provide any tangible communication improvement. as long as the T_comm is larger than 22 Gbps, the k is ∞, which indicates it is impossible to get any benefit from our compression framework in this case. indicating that the performance of compression primitives is critical to a high-end network like InfiniBand.

7.3.4 Convergence Analysis

In order to analyse the convergence of our proposed technique we formulate the DNN training as:

$$\min_x f(x) := \frac{1}{N} \sum_{i=1}^{N} f_i(x), \tag{7.5}$$

where f_i is the loss of one data sample to a network. For non-convex optimization, it is sufficient to prove the convergence by showing $\|\nabla f(x^t)\|^2 \leq \epsilon$ as $t \to \infty$, where ϵ is a small constant and t is the iteration. The condition indicates the function converges to the neighborhood of a stationary point. Before stating the theorem, we need to introduce the notion of Lipschitz continuity. $f(x)$ is smooth and non-convex, and ∇f are L-Lipschitz continuous. Namely,

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

For any x, y,

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2} \|x - y\|^2.$$

Assumption 2. *Suppose j is a uniform random sample from $\{1, ..., N\}$, then we make the following bounded variance assumption:*

$$\mathbb{E}[\|\nabla f_j(x) - \nabla f(x)\|^2] \leq \sigma^2, \textit{ for any } x.$$

This is a standard assumption widely adopted in the SGD convergence proof [Nemirovski et al., 2009] [Ghadimi and Lan, 2013]. It holds if the gradient is bounded.

Assumption 3. *In the data-parallel training, the gradient of each iteration is $\bar{v} = \frac{1}{p} \sum_1^p v_i$; p is the number of processes, and v_i is the gradient from the i^{th} process. Let's denote $\theta \in [0, 1]$ to control the percentage of information loss in the compression function $\hat{v_i} = T(v_i, \theta)$ that does quant(FFT-sparsification(v_i)), so $\bar{\hat{v}} = \sum_1^p \hat{v}_i$. We assume there exists a α such that:*

$$\|\bar{v} - \bar{\hat{v}}\| \leq \alpha \|\bar{v}\|.$$

So, $\hat{v}$ only loses a small amount of information with respect to $\bar{v}$, and the update from the sparsified gradient is within a bounded error range of true gradient update. It is a necessary condition for deriving the upper bound.

With our compression techniques, one SGD update becomes:

$$x^{t+1} = x^t - \eta_t (\frac{1}{P} \sum_1^p \hat{v}_i) = x^t - \eta_t \bar{\hat{v}}_t. \tag{7.6}$$

Then, we have the following lemma for one step:

Lemma 1. *Assume $\eta_t \leq \frac{1}{4L}, \theta_t^2 \leq \frac{1}{4}$. Then*

$$\frac{\eta_t}{4} \mathbb{E}[\|\nabla f(x^t)\|^2] \leq \mathbb{E}[f(x^t)] - \mathbb{E}[f(x^{t+1})] + (L\eta_t + \theta_t^2) \frac{\eta_t \sigma^2}{2b_t}. \tag{7.7}$$

Please check the supplemental material for the proof of this lemma. Summing over (7.7) for K iterations, we get:

$$\textstyle\sum_{t=0}^{K-1} \eta_t \mathbb{E}[\|\nabla f(x^t)\|^2] \leq 4(f(x^0) - f(x^K)) + \sum_{t=1}^{K-1} (L\eta_t + \theta_t^2) \frac{2\eta_t \sigma^2}{b_t}. \tag{7.8}$$

Next, we present the convergence theorem.

Theorem 2. *If we choose a fixed learning rate, $\eta_t = \eta$; a fixed dropout ratio in the sparsification function, $\theta_t = \theta$; and a fixed mini-batch size, $b_t = b$; then the following holds:*

$$\min_{0 \le t \le K-1} \mathbb{E}[\|\nabla f(x^t)\|^2] \le \frac{4(f(x^0)-f(x^{K-1}))}{K} + (L\eta+\theta^2)\frac{2\eta\sigma^2}{b}.$$

Proof. $\min_{0 \le t \le K-1} \mathbb{E}[\|\nabla f(x^t)\|^2] \le \frac{1}{K}\sum_{t=0}^{K-1} \eta_t \mathbb{E}[\|\nabla f(x^t)\|^2]$, as $\|\nabla f(x^t)\|^2 \ge 0$. By (7.8), we get the theorem. □

Theorem 3. *If we apply the diminishing stepsize, η_t, satisfying $\sum_{t=0}^{\infty} \eta_t = \infty, \sum_{t=0}^{\infty} \eta_t^2 < \infty$, our compression algorithm guarantees convergence with a diminishing drop-out ratio, θ_t, if $\theta_t^2 = L\eta_t$.*

Proof. If we randomly choose the output, x_{out}, from $\{x^0, ..., x^{K-1}\}$, with probability $\frac{\eta_t}{\sum_{t=0}^{K-1} \eta_t}$ for x^t, then we have:

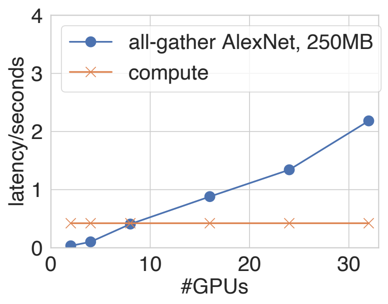
$$\mathbb{E}[\|\nabla f(x_{out})\|^2] = \frac{\sum_{t=0}^{K-1} \eta_t \mathbb{E}[\|\nabla f(x^t)\|^2]}{\sum_{t=0}^{K-1} \eta_t} \tag{7.9}$$

$$\le \frac{4(f(x^0)-f(x^*))}{\sum_{t=0}^{K-1} \eta_t} + \frac{\sum_{t=0}^{K-1}(L\eta_t+\theta_t^2)2\eta_t\sigma^2}{b\sum_{t=0}^{K-1} \eta_t}. \tag{7.10}$$

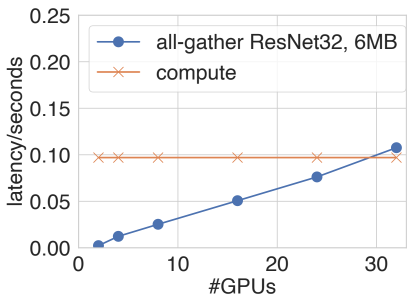
Note that $\sum_{t=0}^{K-1} \eta_t \to \infty$, while
$\sum_{t=0}^{K-1} (L\eta_t + \theta_t^2)2\eta_t\sigma^2 = \sum_{t=0}^{K-1} 4L\eta_t^2\sigma^2 < \infty$,
and we have $\mathbb{E}[\|\nabla f(x_{out})\|^2] \to 0$. □

7.4 Evaluation

Our experiments consist of two parts to assess the proposed techniques. First, we validate the convergence theory and its assumptions with AlexNet on ImageNet and ResNet32 on CIFAR10, which sufficiently cover typical workloads in traditional linear and recent non-linear neural architectures, and also provide coverage on two widely used datasets. Then, we show that the FFT-based method demonstrates better convergence and faster compression than other state-of-the-art compression methods such as QSGD [Alistarh et al., 2017], TernGrad [Wen et al., 2017], Top-k sparsification [Lin et al., 2017, Alistarh et al., 2018], as our techniques incur fewer approximation errors, while still delivering a competitive compression ratio for using both sparsification and quantization.



(a) AlexNet



(b) ResNet32

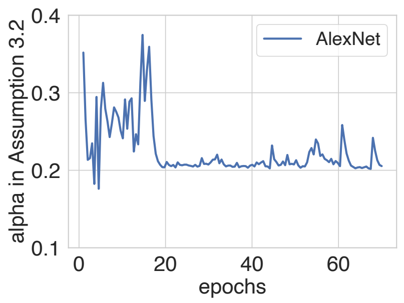
Figure 7.11: the latency for *all-gather* AlexNet and ResNet32 from 2 to 32 GPUs.

Parallelization scheme: we choose BSP for parallelization for its simplicity in the theoretical analysis: BSP follows strict synchronizations, allowing us to better observe the effects of gradient compression toward the convergence by iterations.

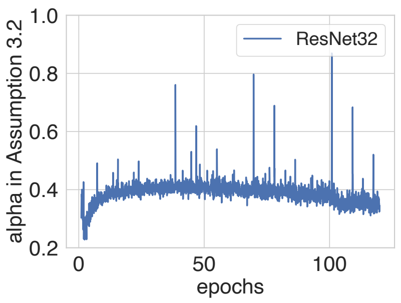
Implementation: we implemented our approach, losses SGD(no compression), QSGD, Top-K, and TernGrad in a C++ DL framework, SuperNeurons [Wang et al., 2018b]; We used the allgather collective from NVIDIA NCCL2 to exchange compressed gradients since existing communication libraries lack the support for sparse all-reduce (Figure 7.1a). Even though SGD usually uses allreduce instead of allgather as it does not have compression; for a fair comparison, we applied allgather for all algorithms to demonstrate the algorithmic benefit of our FFT compression. Every GPU has a copy of global gradients for updating parameters after all-gather local gradients. Parameters need to be synchronized after multiple iterations to eliminate the precision errors, and here we broadcast parameters every 10 iterations. It is noticed that we did not adopt communication and computation overlapping strategy as it could be another optimization method orthogonal to compression, and is not in the scope of this paper.

Training setup: The single GPU batch is set to 128 and 64 for ResNet32 and AlexNet, respectively. The momentum for both networks is set to 0.9. The learning rate for Resnet32 is 0.01 at epochs $\in [0, 130]$, and 0.001 afterwards; the learning rate for AlexNet is 0.01 at epochs $\in [0, 30]$, 0.001 at epochs $\in [30, 60]$, and 0.0001 afterwards.

Machine setup: we conducted experiments on the Comet cluster hosted at San Diego Supercomputer Center. Comet has 36 GPU nodes, and each node is equipped with 4 NVIDIA TESLA P100 GPUs and 56 Gbps FDR InfiniBand, Figure 7.11 shows the *allgather* cost almost linearly increases with the number of GPUs. This is because the total exchanged messages in *allgather* linearly increase with #GPUs [Gabriel et al., 2004]. In our experiments, we used 8 GPUs in evaluating the accuracy and performance when integrating our

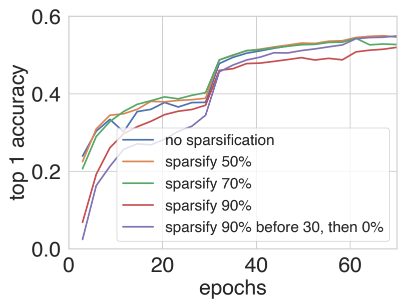


(a) AlexNet

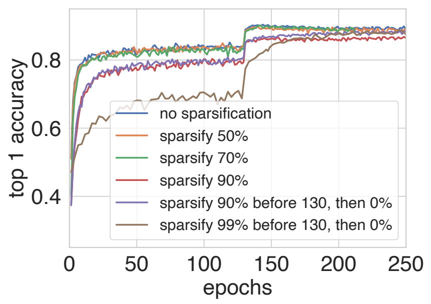


(b) ResNet32

Figure 7.12: Empirical verification of Assumption 3.



(a) AlexNet



(b) ResNet32

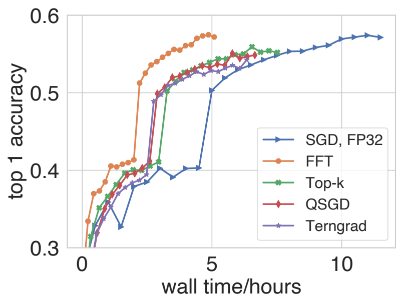
Figure 7.13: Empirical validation of Theorem 3.

compression methods in training, and up to 32 GPUs in evaluating the scalability of the distributed training.

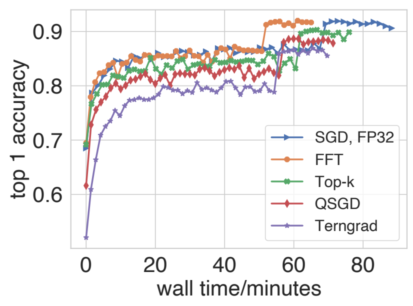
7.4.1 Validation of Theorems

Verification of assumptions: our convergence theorems rely on Assumption 2 and Assumption 3. Assumption 2 automatically holds due to the bounded gradients. Assumption 3 always hold if $p = 1$, but it can break in very rare cases for $p > 1$. For example, α does not exist if $\bar{v} = [0, 0]$, given two opposite gradients, e.g. $\bar{v}_1 = [-0.3, 0.5]$ and $\bar{v}_2 = [0.3, -0.5]$. Though the scenario is very unlikely, we empirically validate Assumption 3 on different training tasks by calculating $\alpha = \frac{\|\bar{v}-\tilde{\bar{v}}\|}{\|\bar{v}\|}$. From Figure 7.12, $\alpha \in [0, 1]$ practically sustaining Assumption 3.

Validation of theorems: Theorem 2 states a large compression ratio, i.e. large θ, can jeopardize the convergence, and theorem 3 states that our FFT-based sparsified SGD is guaranteed to converge with a diminishing compression ratio. The goal of optimization is to find a local optimum, where the gradient approximates to zero, i.e, $\mathbb{E}[\|\nabla f(x^t)\|^2] \to 0$, as $K \to \infty$. From the inequality in theorem 2, $\frac{4(f(x^0)-f(x^{K-1}))}{K} \to 0$ as $K \to \infty$, leaving $E[\|\nabla f(x^t)\|^2]$ bounded by $(L\eta + \theta^2)\frac{2\eta\sigma^2}{b}$. $L\eta\frac{2\eta\sigma^2}{b}$ is the error term from SGD, and



(a) AlexNet



(b) ResNet32

Figure 7.14: Training wall time on a 8 GPUs cluster: FFT outperforms TernGrad, QSGD and Top-k in both the speed and test accuracy. FFT is faster for a high compression ratio by combining sparsification and quantization, while the better gradient quality of FFT explains the good accuracy, as we will show in Figure 7.15.

$\theta^2 \frac{2\eta\sigma^2}{b}$ is the error term from the compression. Compared to the SGD, using a large θ in the gradient compression slacks off the bound for $\mathbb{E}[\|\nabla f(x^t)\|^2]$, causing the deterioration on both the validation accuracy and training loss. As shown in Figure 7.13, when $\theta = 0.5$ (i.e., sparsify 50%), the accuracy and loss traces of AlexNet and ResNet32 behave exactly the same as SGD (shown as no sparsification). When $\theta = 0.9$ (i.e., sparsify 90%), both the training loss and validation accuracy significantly deviate from SGD, as a large θ increases the error term $\frac{2\eta\sigma^2\theta^2}{b}$ loosening the bound for $\mathbb{E}[\|\nabla f(x^t)\|^2]$. Therefore, θ is critical to retain the same performance as SGD, and it is tricky to select θ in practice. We present Theorem.3 to resolve this issue. The theorem compensates for Theorem 2, indicating that a large θ can still deliver the same accuracy as SGD if we shrink the θ during the training. Empirical results in Figure 7.13 validate Theorem 3. For example, by setting $\theta = 0.9$ (drop 90%, red line), both AlexNet and ResNet32 fail to converge to the same case of SGD. However, it is able to bring the accuracy back to the same result as the SGD in the same epochs simply by diminishing θ from 0.9 to 0 at the 30th epoch for AlexNet, and at the 130th epoch for ResNet32. Therefore, we claim both Theorem 2 and Theorem 3 are legitimate.

Implications of theorems: these two theorems explain the relationship between the accuracy and compression ratio θ, and act as a guide to help preserve the training network accuracy by tuning the compression ratio during the training. Hence, in practice, to ensure the convergence, we can shrink θ along with the learning rate η for the condition of $\theta_t^2 = L\eta_t$. In order to recover the accuracy, we can also reduce θ as the case in Fig. 7.13 that a failure case ($\theta = 0.9$) recovers the accuracy after reducing θ to 0 in the middle of training.

Method	AlexNet top1 acc	Speedup w.r.t SGD	ResNet32 top1 acc	Speedup w.r.t SGD
SGD, FP32	56.52%	1	92.11%	1
FFT	56.61%, (+0.09%)	2.26	91.99%, (−0.12%)	1.33x
Top-K	55.07%, (−1.45%)	1.53	90.31%, (−1.80%)	1.12x
QSGD	53.54%, (−2.98%)	1.73	88.66%, (−3.45%)	1.21x
TernGrad-noclip	52.86%, (−3.66%)	1.81	86.90%, (−5.21%)	1.24x

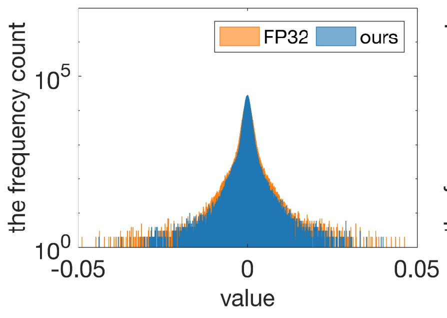
Table 7.2: Summarization of Figure 7.14: the difference of test accuracy and the speedup over lossless SGD.

7.4.2 Algorithm Comparisons

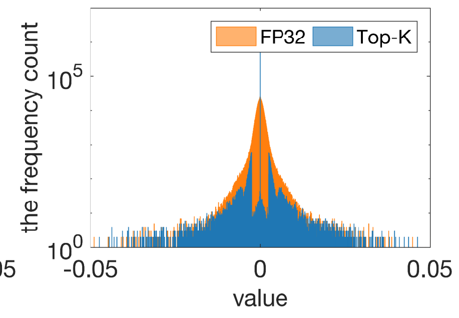
Choice of Algorithms: Here we evaluate our FFT-based techniques against 3 major gradient compression algorithms, Top-k sparsification [Lin et al., 2017, Alistarh et al., 2018, Aji and Heafield, 2017], and Terngrad [Wen et al., 2017] and QSGD [Alistarh et al., 2017]. The baseline method is SGD using 32 bits float. Top-k sparsification thresholds the gradients w.r.t their magnitude, and the compression ratio is determined by 1/(1-θ), where θ is the drop-out ratio. Please note that Top-k variant e.g. DGC [Lin et al., 2017] utilizes heuristics like error accumulation and momentum correction to boost performance. To fairly evaluate Top-k sparsification against FFT based sparsification, we evaluated the vanilla Top-k v.s. the vanilla FFT sparsification, and finding heuristics to boost FFT sparsification is orthogonal to this study. Both Terngrad and QSGD map gradients to a discrete set. Specifically, Terngrad maps each gradient to the set of $\{-1, 0, 1\} * max(|g|)$, and thus 2 bits are sufficient to encode a gradient. Instead, QSGD uses N bits to maps each gradient to a uniformly distributed discrete set containing 2^N bins. Please note TernGrad does not quantize the last classification layer to keep good performance [Wen et al., 2017], while we sparsify the entire gradients.

Algorithm Setup: Regarding Top-k and FFT based sparsification, results from Figure 7.13 and [Alistarh et al., 2018] show a noticeable convergence slowdown after $\theta > 90\%$. To maintain a reasonable accuracy, we choose θ = 85% for both top-k and FFT based sparsification. We use $min = -1$ and $max = 1$ as the boundaries, and 10 bits in initializing our N-bit quantizer. Therefore, the compression ratio for Top-k is 1/(1-θ) = 6.67x and FFT based is 21.3x with an additional 32/10 from quantizers. Terngrad uses 2 bits to encode a gradient, while we use 8 bins (3 bits) for QSGD to encode a gradient. As a result, the compression ratio of Terngrad is 16x and QSGD is 10.6x. Please note we calculate the compression ratio w.r.t gradients as gradient exchanges dominate communications in BSP. Following a similar setup in Figure 7.13, each algorithm is set to run 180 epochs on CIFAR10 and 70 epochs on ImageNet using 8 GPUs.

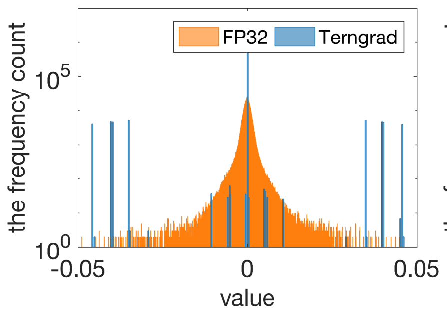
Figure 7.14 demonstrates that our framework outperforms QSGD, Terngrad, and Top-k in both the final accuracy and the training wall time on an 8 GPU cluster, and Table 7.2 summarizes the test accuracy



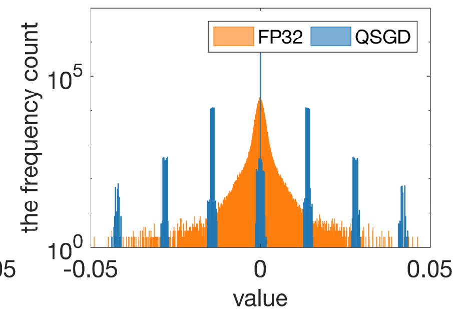
(a) Ours



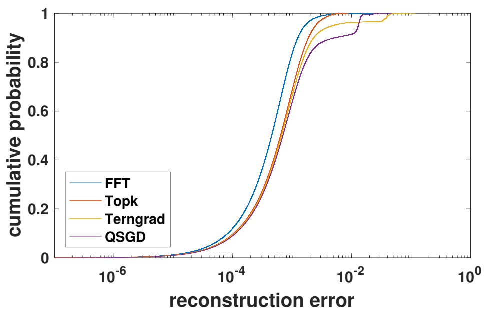
(b) Top-k



(c) Terngrad



(d) QSGD



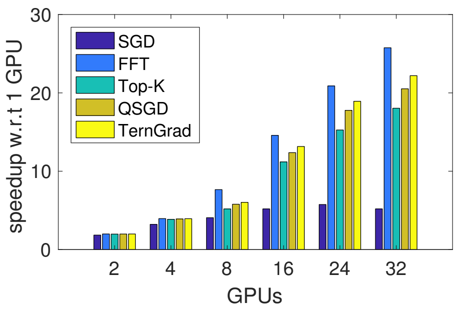
(e) reconstruction error

Figure 7.15: (a)$\rightarrow$(d): Histogram of reconstructed gradients (blue) by FFT ($\theta = 0.85$), Top-k ($\theta = 0.85$), QSGD and Terngrad v.s. the original. The reconstructed gradients by FFT is the closest to the original(FP32). (e) Cumulative error distribution of $|g_i - \hat{g}_i|$, where g_i is the i-th true gradient, and $\hat{g}_i$ is the i-th sparsified gradient. FFT incurs less errors than other approaches for 99.7% of the gradients.

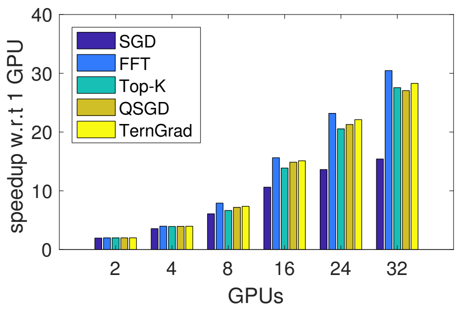
and speedup over the lossless SGD. Particularly, FFT consistently reaches a similar accuracy to SGD with the highest speedup. To further investigate the algorithmic and system advantages of the FFT method, we investigate the gradient quality and the scalability of iteration throughput.

The algorithmic advantages of FFT

We claim the algorithmic advantages of FFT for preserving the original gradient distribution and rendering fewer reconstruction errors than others. We uniformly sampled the gradients of ResNet32 every 10 epochs



(a) AlexNet



(b) ResNet32

Figure 7.16: Weak scalability from 2 to 32 GPUs: we measure the iteration throughput, and calculate the speedup w.r.t 1 GPU.

during the training. Figure 7.15 demonstrates the distribution of reconstructed gradients w.r.t the gradients before the compression. FFT is the only one that retains the original gradient distribution, though $\theta = 85\%$ frequency has been removed. In contrast, Top-k loses the peak for eliminating the near-zero elements at the same θ. Similarly, QSGD presents 7 clusters for using 8 bins to represent a gradient; and, in general, TernGrad shows 3 major clusters around {0, -0.05, 0.05} for using a quantization set of {-1, 0, 1}. Please note that Terngrad shows 11 bars; this is due to the aggregation of sparsified gradients from each node. Aside from qualitatively inspecting the gradient distribution, we also quantitatively examined the empirical cumulative distribution of the reconstruction error in Figure 7.15e. FFT demonstrates the lowest error within the range of $[10^{-5}, 10^{-2}]$. Therefore, FFT can reach better accuracy in the same training iterations.

The system advantages of FFT

Our compression framework fully exploits both the gradient sparsity and the redundancy in 32-bit floating point by further quantizing the FFT sparsified gradient. It enables FFT to deliver a much higher iteration throughput than QSGD, TernGrad, and Top-K. Following the same setting in Figures 7.14, Figure 7.16 demonstrates the iteration throughput of training AlexNet and ResNet32 from 2 to 32 GPUs. Please note that using a very large θ (e.g., 0.999) can get an impressive speedup, but it also drastically hurts the final accuracy. Here we still use $\theta = 85\%$. The gradients of AlexNet (ImageNet) is around 250 MB, while the gradients of ResNet32 (CIFAR-10) are only 6MB. Therefore, the scalability of AlexNet is generally better than ResNet32. Better results are also observable if using a slow network, e.g., 100MB Gbps. When GPUs ≤ 4, the speedup is similar as communications are intra-node through PCI-E. FFT still consistently demonstrates the highest iteration throughput for a better compression ratio when GPUs increase from 8 to 32.

7.5 Related Work

We categorize the existing lossy gradient compression into two groups: (1) *quantization* and (2) *sparsification*.

Quantization: 1-bit SGD [Seide et al., 2014] is among the first to quantize gradients to alleviate the communication cost in the distributed training. Specifically, it quantizes a 32-bit IEEE-754 float into a binary of [0, 1] to achieve a compression ratio of $32\times$. Though their methods are purely heuristic, and their empirical validations demonstrate a slight loss of accuracy, it shows the possibility to train a network with highly lossy gradients. Subsequently, several quantization methods have been proposed. Flexpoint [Köster et al., 2017] uses block floating-point encoding based on current gradient/weight values. HOGWILD! [De Sa et al., 2015] quantizes both weights and gradients into 8-bit integers by rounding off floats (i.e., low-precision training); but this idea is largely restricted by the availability of low-precision instruction sets. TernGrad [Wen et al., 2017] quantizes a gradient as [-1, 0, 1]$*|max(g)|$, while QSGD [Alistarh et al., 2017] stochastically quantizes gradients onto a uniformly discretized set. Both approaches distribute the precision uniformly across the representable range—ignoring both the distribution and the range of the gradients. As we show, gradients follow a normal distribution (Figure 7.4). In our range-based quantizer, we allocate precision for the range and the distribution of the values to better exploit the limited number of bits. Most importantly, QSGD and TernGrad damage the original gradient distribution due to limited representable values after the quantization (Figure 7.15). As a result, TernGrad and QSGD incur an observable deterioration in the final accuracy (Table 7.2).

Sparsification: Aji and Heafield [Aji and Heafield, 2017] present the very first Top-k gradient sparsification showing that the training can be done with a small accuracy loss by setting the 99% smallest gradients to zeros. Based on the Top-k thresholding, Han et al. [Han et al., 2015] propose Deep Compression, which uses heuristics like momentum correction and error accumulation to resolve the accuracy loss in the vanilla Top-k. Please note that these heuristics are orthogonal to our methods and can also be applied to improve ours. Jin et al. [Jin et al., 2019] propose DEEPSZ, which performs error-bounded lossy compression on the pruned weight. It is a modification of the SZ lossy compression framework [Di and Cappello, 2016]. Cédric et al. [Renggli et al., 2018] propose a communication sparsification approach called SPARCML. Different from ours, the SPARCML focuses on the implementation of MPI collective operations of sparse data. D. Alistarh et al. [Alistarh et al., 2018] analyze the convergence of Top-k compression. With [Alistarh et al., 2018], we noticed a significant convergence slowdown at a large sparsity. As we investigated, these Top-k methods also distort the gradient distribution at a large sparsity, yielding higher approximation errors than the original gradients. At the same sparsity (θ), our FFT method is much better at preserving the original gradient distribution and shows less approximation error and better results.

7.6 Conclusion

As indicated in Sec. 7.2, exchanging gradients is the major bottleneck for the distributed DNN training. To alleviate this communication bottleneck, this paper proposes a lossy gradient compression framework that uses an FFT-based gradient sparsification and a range-based, variable-precision, floating-point representation. We theoretically prove that our techniques preserve the convergence and the final accuracy by adapting the sparsification ratio θ during the training, and empirically verify the assumptions and the theory.

At the same sparsification ratio (θ), we show FFT preserves more gradient information than other state-of-the-art lossy methodologies including Top-K sparsification, Terngrad, and QSGD. Besides, our adaptive float quantization further improves the overall compression ratio with negligible loss of gradient information (Fig. 7.15). These advantages enable us to use a larger compression ratio in retaining the same accuracy as the lossless SGD, than other lossy methodologies to improve the scalability (Fig. 7.16) in the distributed training.

Our lossy gradient compression framework demands a highly efficient allreduce that supports communications of sparse data, while current MPI implementations, such as Open MPI or MVAPICH, lack the support of sparse collectives. Though this work uses all-gather to circumvent this issue, future research and development of a bandwidth-efficient allreduce with the sparse support are highly desired to facilitate the deployment of lossy gradient compression techniques in practice.

Part IV

System Building

Chapter 8

SuperNeurons: A Deep Learning Framework to Support Large Models

8.1 Introduction

Deep Neural Network (DNN) is efficient at modeling complex nonlinearities thanks to the unparalleled representation power from millions of parameters. This implies scaling up neural networks is an effective approach to improve the generalization performance. The Deep Learning (DL) community now widely acknowledges either going deeper or going wider on the nonlinear architecture improves the quality of image recognition tasks. For example, 9-layer AlexNet won the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) with a top-5 error of 17%. GoogLeNet (inception v1) refreshed the top-5 error rate to 6.67% with 22 inception units in 2014 ILSVRC, and ResNet further reduced the error rate down to 3.57% in 2015 ILSVRC with 152 residual units.

While DL practitioners are enthusiastically seeking deeper and wider nonlinear networks, the limited size of GPU DRAM becomes a major restriction. Training a deep network is inherently a computation-intensive task. Almost every AI lab today, either in academia or industry, is deploying the network training on GPUs for the purposes of better performance [Bahrampour et al., 2016]. Data need to be residing on GPU DRAM for the GPU computing, but the largest commercial GPU DRAM so far is 24 GB. This is still far from sufficient to accommodate a deep neural network. For example, the latest Inception v4 has 515 basic layers consuming 44.3 GB memory in the training. The deeper or wider we go, the higher memory usages will be. Therefore, this deep trend subjects the rigid GPU DRAM to the severe space insufficiency.

Major DL frameworks, such as Caffe or MXNet, have tried to alleviate the GPU memory shortage with several static memory reduction techniques. Those techniques, due to their static nature, are not well tuned to address the new data and dependency variations in non-linear networks. For example, Caffe and Torch do not fully support the data flow analysis on non-linear neural networks; the strategy of trading computation for memory in MXNet is limited because it ignores the memory variations across network layers. These limitations have motivated us to propose a dynamic approach for the emerging deep nonlinear neural architectures.

In this paper, we present the first dynamic GPU memory scheduling runtime for training deep non-linear neural networks. The runtime allows DL practitioners to explore a much deeper and wider model beyond the physical limitations of GPU memory. It utilizes tensors as the fundamental scheduling units to consist with the layer-wise computations enforced in DL performance primitives cuDNN [Chetlur et al., 2014]. The runtime seamlessly orchestrates the tensor placement, movement, allocation and deallocation so that the underlying memory operations are entirely transparent to users.

Our runtime guarantees the minimal peak memory usage, $peak_m = \max(l_i)$, at the layer-wise granularity. We denote the memory usage of the ith layer as l_i, and the superscript, e.g. l_i^f or l_i^b, as the forward/backward. The peak memory usage during the forward and backward computations is denoted as $peak_m$. First, *Liveness Analysis* recycles no longer needed tensors to reduce $peak_m$ from baseline $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$ to $\sum_{i=1}^{N} l_i^f + l_N^b$ (defined in Sec.8.3). Secondly, Unified Tensor Pool (UTP) offloads tensors in compute-intensive layers, referred to as checkpoints, to the external physical memory. This further reduces $peak_m$ from $\sum_{i=1}^{N} l_i^f + l_N^b$ to $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$. Finally, *Cost-Aware Recomputation* drops the forward results of cheap-to-compute or none-checkpoints layers and reconstructs them to reduce $peak_m$ from $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$ to $peak_m = \max(l_i)$. The final $peak_m$ indicates the largest computable network is bounded by the maximal memory usage among layers.

Our runtime also features three performance optimizations to improve the efficiency of *Liveness Analysis* and *UTP*. First, GPUs require memory allocations to create tensors and deallocations to free tensors. Thus, the highly frequent large tensor allocations/deallocations incur the non-negligible overhead in *Liveness Analysis* [Wang et al., 2016b]. The runtime successfully amortizes the cost by directly reusing memory segments from a huge pre-allocated memory pool, managed by a heap based GPU memory management utility. Secondly, UTP swaps tensors among different physical memory spaces, while modern GPUs equip with independent Direct Memory Access (DMA) engine exposing opportunities to hide communications under computations. The runtime also meticulously overlap communications with computations. However, the

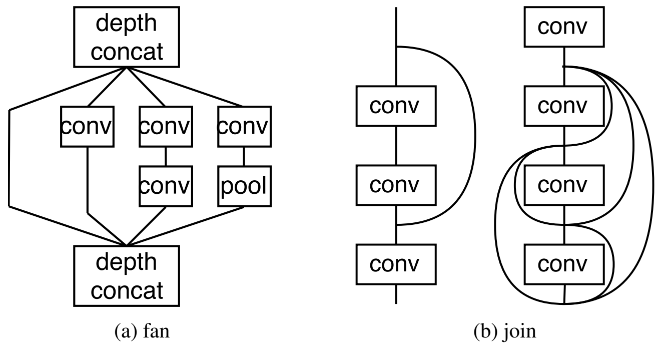


Figure 8.1: The non-linear connections in inception v4 (fan), ResNet (join, left) and DenseNet (join, right). DenseNet utilizes a full-join.

overlapping opportunity is limited given the fixed amount of computations. We propose a LRU based Tensor Cache built on GPU DRAM to minimize total communications by tensor reusing.

This paper claims the following contributions:

1. We demonstrate the new memory scheduling challenges in nonlinear neural networks, and discuss the key limitations of existing approaches.

2. By dynamically allocating memory for convolution workspaces, SuperNeurons delivers the leading performance among state-of-art DL systems on the GPU.

3. We design and implement SuperNeurons to enable DL practitioners to explore deep neural networks; and the largest computable network of SuperNeurons is only bounded by the maximum memory usage among layers.

8.2 Background and Motivation

8.2.1 Challenges for Processing Super Deep Neural Networks

Traditional Convolutional Neural Networks (CNN) [LeCun et al., 1998, Krizhevsky et al., 2012, Simonyan and Zisserman, 2014] are typically composed of several basic building layers, including Convolution (CONV), Pooling (POOL), Activation (ACT), Softmax, Fully Connected (FC), Local Response Normalization (LRN), Batch Normalization (BN), and Dropout. For linear CNNs, these layers are independent and inter-connected to their neighbors in a sequential manner: $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow n$. Recently, several deep non-linear neural architectures have been proposed to further improve the state-of-the-art accuracy on the 1K ImageNet recognition

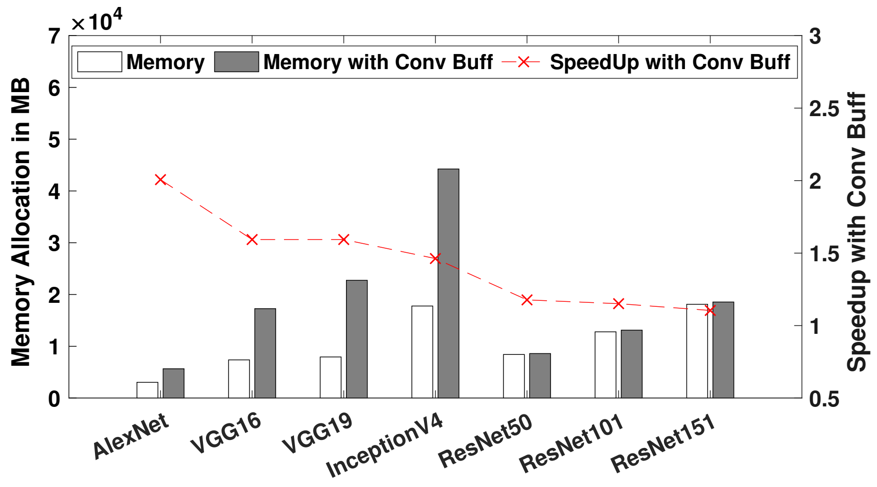


Figure 8.2: The left axis depicts the memory usages of networks. The batch size of AlexNet is 200, and the rest use 32. The right axis and red x marks depict the speedup (imgs/s) with and without convolution workspaces.

challenge, e.g., Inception v4[Szegedy et al., 2017], ResNet[He et al., 2016], and DenseNet[Huang et al., 2016]. These prominent network designs (especially the one that solves the classic gradient vanishing [Bengio et al., 1994] problem) pave the algorithmic foundation for DL practitioners to harness the unparalleled representation power brought forth by the super deep non-linear neural architectures. For example, the latest inception v4 delivers 95% top-5 accuracy with 515 basic building layers while ResNet151[1] achieves 94.3% top-5 accuracy with 567 layers. In Figure 8.1, we illustrate two classic types of non-linear connections: fan and join. Compared with the linear connection pattern, the sparse fan-out connection (Figure 8.1a) avoids one huge computing-inefficient dense layer [Szegedy et al., 2015] while the join connection prevents gradients from quickly vanishing in the back-propagation [He et al., 2016].

Training these super deep and complex non-linear neural architectures is a computation-intensive task. Due to its DL-driven novel architecture designs and massive parallelism, GPUs have been widely adopted in today's industry and academia for the efficient neural network training. However, there are critical issues for efficiently training in these newly-developed super deep non-linear neural architectures: *limited GPU resident memory* and *a high degree of variation in computational dependencies*.

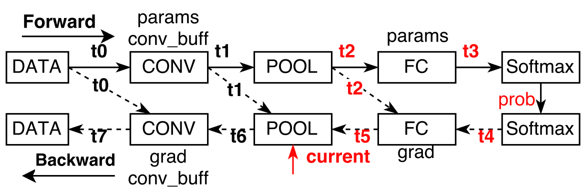
Challenge I: Limited GPU Resident Memory. The prominent deep neural architectures share a common feature: high memory demand and computation intensity. Figure 8.2 illustrates the network-wide memory usages of several recent DNNs in training with and without convolution workspaces (buffer). Among them,

[1]151 represents the number of convolutional units.

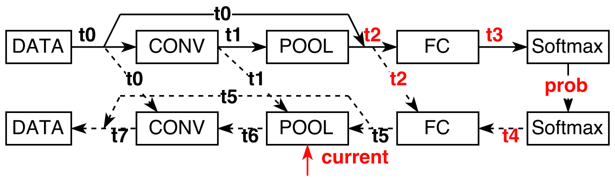
AlexNet and VGG are linear networks while the others are non-linear. We can observe that the non-linear networks demand a significant amount of GPU memory, e.g., ResNet152 and Inception v4 require up to 18.5GB and 44.3 GB at only the batch size of 32, respectively. However, these sizes are either similar to or surpass the resident memory sizes of commercial GPUs on the market today. For instance, the newest generations of NVIDIA Pascal and Volta GPUs only have 16GB with HBM2 enabled (e.g., P100 and V100) while the one with the most memory available in recent generations is Maxwell P40 with 24GB GDDR5. This limitation poses a major bottleneck for deep learning practitioners for exploring deep and wide neural architectures [Szegedy et al., 2015, Pleiss et al., 2017, Szegedy et al., 2017]. The most straightforward solution is to split the network across GPUs, i.e. Model Parallelism. However, splitting either the computations of a network or a layer incurs excessive intra-network and intra-layer communications that drastically deteriorate the performance. For example, recent work has suggested the deficiency of applying model parallelism for deep neural networks: it compromises at least 40% speed when training a network with 1.3 billion parameters from 36 GPUs to 64 GPUs [Coates et al., 2013]. To address the performance issues from Model Parallelism, Data Parallelism has been widely adopted in today's mainstream deep learning frameworks such as Caffe[Jia et al., 2014], TensorFlow[Abadi et al., 2016], Torch[Collobert et al., 2002], and MXNet[**?**]. In this model, each GPU holds a network replica; and one GPU computes one sub-gradient with a sub-batch. Subsequently, all sub-gradients are aggregated as one global gradient to update the network parameters [Wang et al., 2016a]. Although this process does not incur intra-network or intra-layer communications besides necessary gradient exchanges, it requires the network training to fit in the limited GPU DRAM. In this paper, we focus on addressing the GPU memory shortage issue for training deep neural networks under data parallelism model while taking the training performance into design considerations.

Challenge II: Variations in Computational Dependencies for Nonlinear Networks. Nonlinear networks exhibit a high degree of dependency variations while linear networks follow a fixed sequential execution pattern with predictable data dependencies [Rhu et al., 2016]. Fig.8.3 illustrates the data dependency graph for linear (a) and nonlinear (b and c) neural architectures. One typical training iteration consists of two phases: forward and backward propagation. For linear networks, data is sequentially propagated in the forward pass; and a layer's backward computation is simply contingent upon the previous layer as illustrated in Figure 8.3a. Thus their computation and dependency patterns are static regardless of the total layers involved.

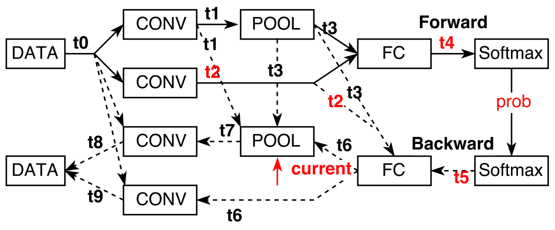
However, for nonlinear networks, a high degree of variation in computational dependencies appear. Fig.8.3b and 8.3c show two simple examples of join and fan nonlinear connections. Join connections forward a layer's output tensor to another layer, creating a dependency between two layers. For example, the join



(a) linear



(b) join (nonlinear)



(c) fan (nonlinear)

Figure 8.3: Data dependencies of different neural architectures. Tensors in red are ready to free when the computation back propagates to the POOL layer. Solid lines represent forward dependencies and dashed lines represent backward dependencies.

in Fig.8.3b forwards `t0` from DATA layer to FC layer in the forward pass. The dependency of join-based non-linear networks is non-deterministic as any two layers can be connected with a join, e.g., in DenseNet. For fan connections, it creates multiple branches in the execution flow: DATA layer forks two branches and joins them before FC layer. Separate branches, each with a different number of layers, have to finish before joining them back to the original branch, making this execution sequence nonlinear. Although the two basic nonlinear scenarios shown here are intuitive, a typical deep nonlinear network today has hundreds of joins and fans convoluted together, resulting in a complex network architecture. These significantly complicate runtime resource-management compared to the static computational pattern in linear ones. Therefore, the memory scheduling of deep non-linear neural networks demands a dynamic solution to effectively address these variations in both the execution flow and computation dependencies.

8.2.2 Limitations of GPU Memory Management in Mainstream Deep Learning Frameworks

Several static memory reduction techniques have been implemented in today's deep learning frameworks to address the GPU memory shortage at data parallelism level. For example, Caffe and Torch directly reuse the forward data tensors for the backward data propagation, which saves up to 50% of memory on a linear network [MXN,]. Although this technique works well on linear networks, it requires extra tensors to hold the future dependencies for training non-linear networks, thereby limiting the effectiveness and efficiency. Also, these frameworks still have to fit the entire network into GPU DRAM without leveraging NUMA architectures, and this level of reuse is arguably not adequate for contemporary deep nonlinear neural networks. MXNet and TensorFlow are built with a Directed Acyclic Graph (DAG) execution engine [Wu et al., 2015]. Users explicitly define the computation flow and tensor dependencies, which provide necessary information for the DAG engine to analyze the life span of tensors. Both systems then free tensors that are no longer needed in order to save memory. MXNet also implements a per-layer-based re-computation strategy that is similar to Resilient Distributed Datasets (RDD) in Spark [Zaharia et al., 2010]. Basically it frees the tensors produced by computation-inexpensive layers in the forward pass, and recomputes the freed dependencies for the backward pass by doing another forward. However, this method neglects non-uniform memory distribution of network layers, consequentially demanding large unnecessary memory usages. TensorFlow swaps long-lived data tensors from GPU DRAM to CPU DRAM, but it fails to optimize data communications between the two (e.g., utilizing pinned data transfer) which compromises at least 50% of communication speed.

More importantly, none of aforementioned DL frameworks utilize a dynamic scheduling policy that provisions necessary memory space for deep nonlinear network training while at the same time optimizing the training speed given the existing GPU DRAM resource. In other words, these static memory-saving techniques aggressively reduce the GPU memory usage at the expense of speed. Users either painstakingly tune the performance or suffer from insufficient memory during the execution. Additionally, these frameworks either have no optimization strategy or adopt a naive method on allocating the convolution workspace (see Section 8.3.5), which is a decisive factor determining CNN training speed on the GPU. In summary, these challenges motivate us to design a dynamic scheduling runtime to provision necessary memory for the training while maximizing the memory for convolution workspaces to optimize the training speed.

8.3 Design Methodologies

This section elaborates on three memory optimization techniques and their related performance issues in SuperNeurons. From a high-level perspective, SuperNeurons provision necessary memory spaces for the training while maximizing the speed by seeking convolution workspaces within the constraint of native GPU memory size.

Notations and Baseline Definition: To facilitate the analysis of proposed techniques, we denote the forward memory usage of the ith layer as l_i^f, the backward as l_i^b. We denote the peak memory usage as $peak_m$. We use the naive network-wide tensor allocation strategy as the baseline, which allocates an independent tensor for each memory requests. Thus, the $peak_m$ of baseline is $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$. We also denote the maximal memory usage among layers as $l_{peak} = max(l_i)$, where $i \in [1, N]$, and N represents the network length. t_i represents the ith tensor.

First, *Liveness Analysis* reduces the baseline $peak_m$ to $\sum_{i=1}^{N} l_i^f + l_N^b$ by recycling free tensors amid back-propagation, demonstrating up to 50% of the memory saving. This technique is guaranteed to work on various non-linear architectures, and it is constructed in $\mathcal{O}(N^2)$. *Liveness Analysis* involves high-frequent memory operations on the large chunk memory, while native memory utilities, e.g. cudaMalloc and cudaFree, incur the nontrivial overhead. We address this issue with a preallocated heap managed by the runtime.

Secondly, *Unified Tensor Pool(UTP)* further reduces $peak_m$ to $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$, where checkpoints represent the compute-intensive layers such as FC and CONV. UTP provides a consolidated memory abstraction to external memory pools to supply for the training. Instead of using naive on-demand data transfers, it hides communications under computations. While the overlapping opportunity is limited given the fixed amount of computations, UTP further introduces a *Tensor Cache* built on GPU to reduce communications.

Finally, *Cost-Aware Recomputation* reduces $peak_m$ to $max(l_i)$, the minimum at the layer-wise granularity. The method keeps track of memory distributions among checkpoints to minimize the extra computations while ensuring $peak_m \leq max(l_i)$.

8.3.1 Prerequisites

A typical DNN network layer computes on a 4-dimension tensor indexed by batches (N), image channels (C), height (H) and width (W) (Fig.8.5). Since cuDNN operates at the layer granularity, we use tensors as the basic memory scheduling unit.

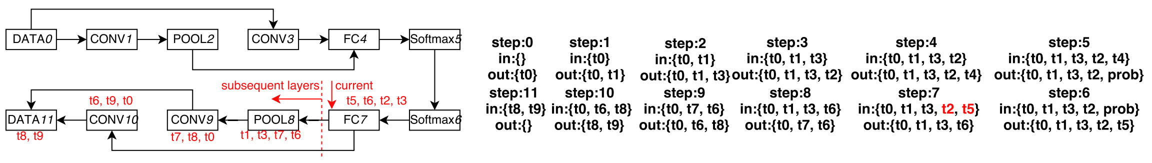

DATA0
CONV1
POOL2
CONV3
FC4
Softmax5
subsequent layers
current
t6, t9, t0
t5, t6, t2, t3
DATA11
CONV10
CONV9
POOL8
FC7
Softmax6
t8, t9
t7, t8, t0
t1, t3, t7, t6
step:0 in:{} out:{t0}
step:1 in:{t0} out:{t0, t1}
step:2 in:{t0, t1} out:{t0, t1, t3}
step:3 in:{t0, t1, t3} out:{t0, t1, t3, t2}
step:4 in:{t0, t1, t3, t2} out:{t0, t1, t3, t2, t4}
step:5 in:{t0, t1, t3, t2, t4} out:{t0, t1, t3, t2, prob}
step:11 in:{t8, t9} out:{}
step:10 in:{t0, t6, t8} out:{t8, t9}
step:9 in:{t0, t7, t6} out:{t0, t6, t8}
step:8 in:{t0, t1, t3, t6} out:{t0, t7, t6}
step:7 in:{t0, t1, t3, t2, t5} out:{t0, t1, t3, t6}
step:6 in:{t0, t1, t3, t2, prob} out:{t0, t1, t3, t2, t5}

Figure 8.4: Applying *Liveness Analysis* on the nonlinear network shown in Fig.8.3c. The number after the layer name (e.g., DATA0, CONV1, etc.) represents the step, which are calculated by Alg. 8. We mark the prerequisite tensors for a layer in red, such that t_7, t_8, t_0 are required by CONV9. Each in and out set tracks live tensors before and after the layer's computations. We can free $t2$ and $t5$ at step 7 since no subsequent dependencies from POOL8, CONV9, CONV10, and DATA11.

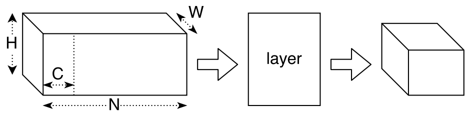


Figure 8.5: The structure of tensors used in DNN.

Algorithm 8 Construct execution steps for nonlinear neural architectures

```
Data: neural architecture definitions
Result: execution order
Function RouteConstruct (layer)
    if layer is NULL then
        return
    $layer \rightarrow counter\_inc()$
    if layer→get_counter < size of prev layers then
        return
    $computation\_route.push(layer);$
    $next\_layers = b \rightarrow get\_next();$
    for $next\_l \in next\_layers$ do
        RouteConstruct (next_l);
    $reset\ layer \rightarrow counter\ to\ 0$
```

Alg.8 describes how SuperNeurons constructs execution steps for nonlinear neural architectures. The input is the first network layer; then Alg.8 recursively explores the subsequent layers in Depth-First Searching (DFS), except that it reaches a join where all prior layers must finish before proceeding. The behavior is achieved by the counter in each layer that tracks the input dependencies (line 5 $\rightarrow$ 6 in Alg.8).

Fig.8.6 demonstrates an example execution route for a nonlinear network constructed by Alg.8. Each box represents a network layer indexed from $\mathbf{a}$ to $\mathbf{j}$. Note that this network has two fan structures (layer $\mathbf{b}, \mathbf{c}, \mathbf{d}$ and layer $\mathbf{f}, \mathbf{g}, \mathbf{h}$) nested together. Alg.8 successfully identifies layers $\mathbf{e}, \mathbf{g}$ and $\mathbf{h}$ as the prerequisites for executing $\mathbf{i}$.

8.3.2 Liveness Analysis and Its Related Issues

Liveness analysis enables different tensors to reuse the same physical memory at different time partitions. Our runtime implements a simple yet effective variant of the traditional data flow analysis constructed in

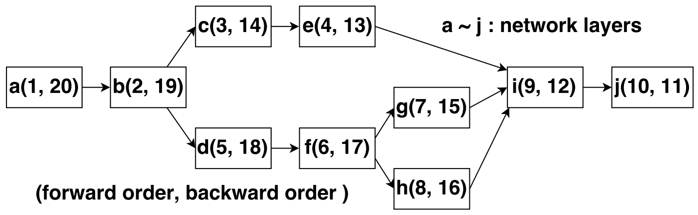


Figure 8.6: Execution route created by Algorithm 8 on a nonlinear network. The left digit represents the forward step, while the right digit represents the backward step.

Figure 8.7: The unified tensor pool provides a consolidated memory abstraction to include various physical memory pools for tensor allocations.

$\mathcal{O}(N^2)$ for various nonlinear neural networks. The general procedures are as follows:

1. We construct an in and out set for every layers to track the live tensors before and after the layer, which cost $\mathcal{O}(N)$, where N is the network length.

2. The runtime populates a layer's in and out sets by checking the dependencies of subsequent layers. It eliminates tensors in in from out if no subsequent layers need them. The cost is $\frac{N(N-1)}{2} \sim \mathcal{O}(N^2)$ as each check costs $N-1$, $N-2$, ..., 2, 1, respectively.

Fig.8.4 demonstrates the detailed procedures of *Liveness Analysis* on the network shown in Fig.8.3c. It explicitly lists the content of in and out sets at each steps. For instance, for FC7, $in = \mathbf{t0}, \mathbf{t1}, \mathbf{t3}, \mathbf{t2}, \mathbf{t5}$. It needs to create tensor $\mathbf{t6}$ to finalize the current computation. Since $\mathbf{t2}$ and $\mathbf{t5}$ are no longer needed after FC7, runtime eliminates them from FC7's out set (step:7).

Liveness Analysis reduces the baseline $peak_m = \sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$ to $\sum_{i=1}^{N} l_i^f + l_N^b$. In order to simplify the analysis, let's assume identical memory usages on every layers, i.e. $l_i^f = l_i^b$ where $i \in [1, N]$. In the network training, the results of forward pass are needed by the backward propagation[2] [Wang et al., 2017a, Chetlur et al., 2014]. Therefore, the forward total memory usages at step k is $cost_k^f = \sum_{i=1}^{k} l_i^f$, where $k \leq N$. During the back-propagation, *Liveness Analysis* frees l_i^f and l_i^b where $i \in [k+1, N]$ at the backward step k since no future dependencies on them as demonstrated in Fig.8.4. Therefore, the backward total memory usages at step k is $cost_k^b = \sum_{i=1}^{k} l_i^f + l_k^b$ and $k \leq N$. Since $l_i > 0$, the $peak_m$ is $max(max(cost_k^f), max(cost_k^b)) = \sum_{i=1}^{N} l_i^f + l_N^b$. Therefore, *Liveness Analysis* saves up to 50% memory from the baseline.

[2]Not all layers require the previous forward output for the back-propagation, again we simplify the case for the analysis.

Toward a High Performance Liveness Analysis

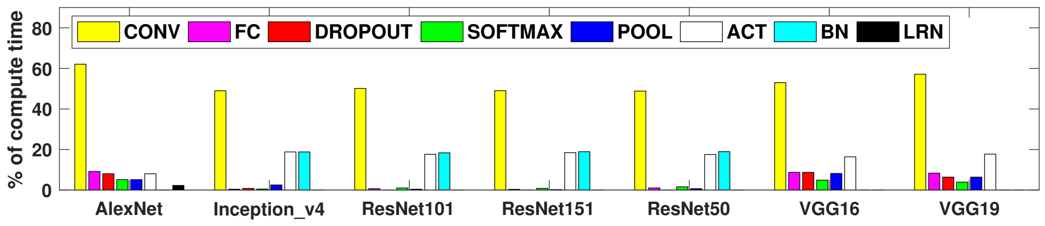
Both the empty initial in set at step 0 and the empty final out set at step 11 in Fig.8.4 demonstrates *Liveness Analysis* frequently stashes and frees tensors on the fly in a training iteration, while a typical training phase consists of millions of iterations and such intense memory operations incur nontrivial overhead if using the native *cudaMalloc* and *cudaFree* [Wang et al., 2016b]. According to the experiment, ResNet50 wastes 36.28% of the training time on memory allocations/deallocations with *cudaMalloc* and *cudaFree*. To alleviate this performance issue, we implement a fast heap-based GPU memory pool utility. The core concept is to remove the allocation/deallocation overhead by preallocating a big chunk of GPU memory as a shared memory pool. Then we divide the entire GPU memory pool into 1KB blocks as the basic storage unit. The memory pool contains a list of allocated and empty memory nodes. Each node in the two lists contains memory address, occupied blocks and node ID. For an allocation request, the memory pool finds the first node with enough free memory from the empty list. After that, it updates the empty list and creates a new node in the allocated list to track the current allocation. For a deallocation request, the memory pool locates the node in the allocated list with the ID-to-node hash-table, then the pool places the node back to the empty list.

8.3.3 Unified Tensor Pool(UTP) and Its Related Issues

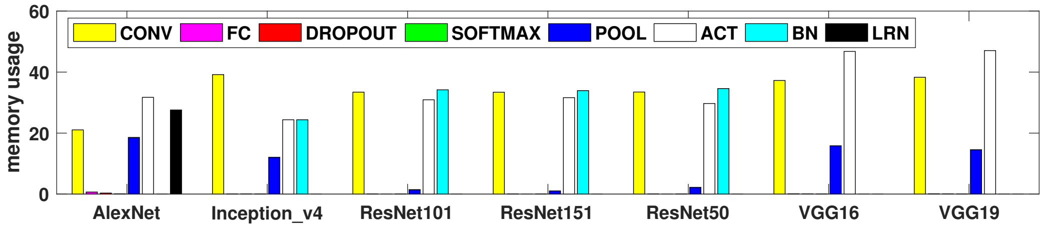
If the depth of a neural network goes to 10^3, the ImageNet training still consumes at least 10^2GB memory. Therefore, *Liveness Analysis* alone is inadequate for the emerging deep nonlinear neural architectures. We provide *Unified Tensor Pool (UTP)* to further alleviate the GPU DRAM shortage by asynchronously transferring tensors in/out of external memory. *UTP* is a consolidated memory pool abstraction for tensor allocations/deallocations, using various external physical memory such as CPU DRAM, DRAM of other GPUs, or remote CPU/GPU DRAM. In this paper, we focus on the scenario of using local CPU DRAM as an external pool for the fast and efficient interconnect, but the abstraction also applies to other cases shown in Fig.8.7. *UTP* intelligently manages the tensor placement, movement, allocation and deallocation, so that the underlying memory management is entirely transparent to DL practitioners.

Basic UTP Memory Management: Memory Offloading and Prefetching

Not all the layers are suitable for *Offloading* and *Prefetching*. We define transferring tensors from GPU to external physical pools as *Offloading*, and the reversed operation as *Prefetching*. Fig.8.8a and Fig.8.8b



(a) breakdown of execution time by layer types



(b) breakdown of memory usages by layer types

Figure 8.8: The percentages of execution time and memory usages by layer types in different networks. Note that the execution time includes both forward and backward passes.

demonstrate that POOL, ACT, BN and LRN all together occupy over 50% of the total memory, while their computations only account for an average of 20% of the entire workload. Thus, offloading these layers incurs a great overhead due to the insufficient overlapping of communications and computations. It is also not fruitful to offload on Dropout, Softmax and FC layers since they only use less than 1% of the total memory. Therefore, we only offload the tensors from CONV layers.

Offloading: the runtime asynchronously transfers the forward outputs of CONV layers to the preallocated pinned CPU memory. It records an event for this data transfer and frees the tensor's GPU memory once the event is completed. The runtime has an independent thread running in the background to check events of memory copies; and this enables GPU-to-CPU data transfers to overlap with the forward computations starting from the current CONV layer to the next one.

Prefetching: the runtime asynchronously brings the offloaded and soon to be reused tensors back to the GPU DRAM. At any CONV layers in the backward, the runtime asynchronously fetches the required tensors for the previous CONV layer. This enables the CPU-to-GPU data transfer to overlap with the backward computation starting from the current CONV layer to the previous one.

Offloading and *Prefetching* reduce $peak_m$ after *Liveness Analysis* to $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$, where $checkpoints = \{CONV\}$. Since layers in $checkpoints$ are offloaded, the total memory consumption at each backward steps is $cost(k) = \sum_{i=1}^{k}(l_i^f \notin checkpoints) + l_k^b$, where $k \in [1, N]$. The memory usage of each layers is non-negative, thus $peak_m = max(cost(k))$ is $\sum_{i=1}^{N}(l_i^f \notin checkpoints) + l_N^b$.

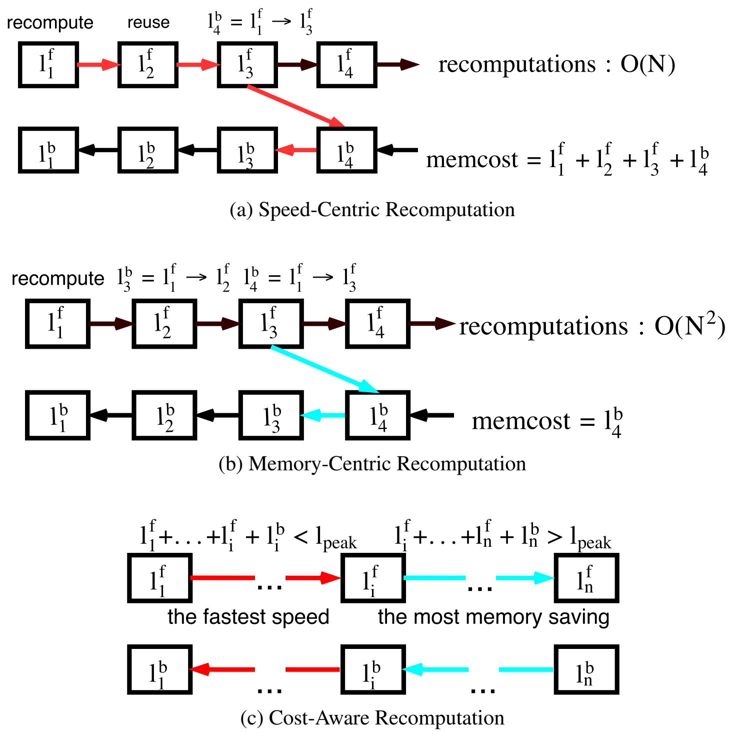

recompute
reuse
$l_4^b = l_1^f \rightarrow l_3^f$
recomputations : O(N)
memcost = $l_1^f + l_2^f + l_3^f + l_4^b$
(a) Speed-Centric Recomputation
recompute $l_3^b = l_1^f \rightarrow l_2^f$ $l_4^b = l_1^f \rightarrow l_3^f$
recomputations : O(N^2)
memcost = l_4^b
(b) Memory-Centric Recomputation
$l_1^f + \ldots + l_i^f + l_i^b < l_{peak}$
$l_i^f + \ldots + l_n^f + l_n^b > l_{peak}$
the fastest speed
the most memory saving
(c) Cost-Aware Recomputation

Figure 8.9: The speed-centric strategy only recomputes the segment once, and other backward layers within the segment will reuse the recomputed tensors. Thus, it only incurs $\mathcal{O}(N)$ additional computations, but $memcost$ is $\sum_{i=1}^{seg} l_i^f + l_{seg}^b$. The memory-centric strategy recomputes forward dependencies every time for each backward layers. Though it incurs $\mathcal{O}(N^2)$ additional computations, $memcost$ is the lowest, i.e. l_i^b. Cost-Aware Recomputation profiles the memory usages across recomputation segments. It uses the speed-centric strategy (red) if $memcost$ of a segment is less than l_{peak}, and the most memory saving strategy (blue) otherwise.

Caching Tensors on GPU DRAM

While the overlapping opportunity is limited given the fixed amount of computations in an iteration, the aforementioned on-demand *Prefetching*/*Offloading* protocol can quickly exhaust the chance. Nowadays CPU-to-GPU data movements over PCI-E, GPU-to-GPU data movements over the same PCI-E switch, and GPU-to-remote GPU over GPU-Direct RDMA deliver a practical speed of 8 GB/s, 10 GB/s, and 6 GB/s respectively but transferring Gigabytes data in each training iterations incurs the nontrivial overhead. Therefore, this on-demand tensor transfer protocol must be optimized. SuperNeurons proposes a *Tensor Cache* to exploit the temporal localities of tensors. It caches tensors on GPU DRAM to maximize their reuses and to minimize the global communications. With *Prefetching* and *Offloading*, the runtime only triggers data transfers when GPU DRAM is insufficient.

Algorithm 9 The basic LRU operations

```
Data: Tensor ($T$) and $LRU$
Result: Tensor with the GPU memory.
Function $LRU.in$ ($T$)
    $T.Lock \leftarrow false$            /* A layer will lock its dependent tensors in the computation. */
    $LRU.insertFront(T)$
Function $LRU.out$ ($T$)
    $freedMem \leftarrow 0$
    while $freedMem < T.size$ do
        $T' = LRU.getLastUnlockedTensor()$
        $freedMem = freedMem + T'.size$
        remove $T'$ from LRU list
        offload $T'.GA$ to $T'.CA$                                   /* CA is CPU Addr */
    $T.GA \leftarrow Malloc(T.size)$
Function $Check$ ($LRU$, $T$)
    $isFound \leftarrow LRU.find(T)$
    if $isFound = false$ then
        $T.GA \leftarrow Malloc(T.size)$                             /* GA is GPU Addr */
        if $T.GA = \emptyset$ then
            $T.GA \leftarrow LRU.out()$
        $LRU.in(T)$                                                    /* cache miss */
    else
        $LRU.placeToFront(T)$                                          /* cache hit */
    return $T.GA$
```

We adopt Least Recent Used (LRU) tensor replacement policy to build *Tensor Cache*. Since the back-propagation demonstrates the head-to-tail and tail-to-head computation pattern, it subjects the most recent used tensors to the earliest reusing as suggested in Fig.8.4. This motivates us to design *Tensor Cache* with a simple variant of LRU. While there are other sophisticated cache replacement policies that might better fit the scenario, thorough discussions of them fall out the scope of this paper.

Alg.9 demonstrates the three key operations of proposed LRU. 1) *LRU.in* function intends to place a tensor into LRU. Each tensor has a lock, and a tensor cannot be removed from LRU if locked. A layer will lock required tensors at calculations. LRU is implemented as a list with Most Frequently Used (MFU) at the front. 2) *LRU.out* function intends to remove enough bytes for a new tensor. It offloads the unlocked Least Recent Used tensors to CPU RAM until it has enough free memory for the new one. 3) *Check* function decides what operator to run on the tensor. It takes in a tensor to check if the tensor is in LRU based on the object address (line 2). If found, we place the tensor to the MFU position, i.e. the list front (line 9), and return the tensor's GPU address. This is the hit scenario. If not found, we call *LRU.out* to free enough memory for the new tensor before inserting it into LRU. This is the miss scenario.

8.3.4 Cost-Aware Recomputation

POOL, ACT, LRN and BN all together use an average of 50% memory, while their forward computations only account for less than 10% of the total time. This exposes an additional 50% memory savings with a fraction of performance loss by recomputing the forward dependencies in the back-propagation. Basically, the runtime frees the tensors in cheap-to-compute layers such as POOL for reconstructions. In general, there are memory-centric and speed-centric strategies for the recomputation for memory.

The speed-centric strategy keeps the recomputed tensors so that other backward layers can directly reuse them. Fig.8.9a denotes the procedures in red. At the backward step on l_4^b, it performs a forward pass from l_1^f to l_3^f to get dependencies for l_4^b. It keeps l_1^f, l_2^f so that they can be re-used for the backward computation on l_3^b and l_2^b. MXNet [Chen et al., 2016] adopts this strategy. It incurs the least $\mathcal{O}(N)$ additional computations, but $memcost$ is $\sum_{i=1}^{seg}(l_i^f) + l_{seg}^b$. $memcost$ will exceed l_{peak} if l_{peak} is within the segment.

The memory-centric strategy always recomputes the dependencies for each backward layer. In contrast to the speed-centric one, it fully exploits the memory-saving opportunity by freeing the recomputed intermediate results. For example, it recomputes $l_1^f \to l_3^f$ for l_4^b, while it recomputes $l_1^f \to l_2^f$ again for l_3^b as demonstrated by the blue lines in Fig.8.9b. The $memcost$ stays at l_i^b guaranteed to be $\leq l_{peak}$, but the strategy incurs $\mathcal{O}(N^2)$ additional computations.

We present a new *Cost-Aware Recomputation* that leverages the advantages of both methods. It is motivated by the observation that the memory costs of most recomputation segments are $\leq l_{peak}$, i.e. $\sum_{i=1}^{seg}(l_i^f) + l_{seg}^b \leq l_{peak}$. That implies we can leverage the least recomputations in the speed-centric strategy while still guaranteeing a memory usage of $\leq l_{peak}$ as in the memory-centric strategy. The general procedures of *Cost-Aware Recomputation* are as follows:

[leftmargin=*]The runtime iterates over all the layers to find $l_{peak} = max(l_i)$ as the threshold. In a recomputation segment, the runtime applies the speed-centric strategy (marked by red in Fig.8.9c) if $\sum_{i=1}^{seg}(l_i^f) + l_{seg}^b \leq l_{peak}$, and the memory-centric strategy (marked by blue in Fig.8.9c) otherwise.

Table.8.1 summarizes the extra computations for two basic strategies and *Cost-Aware Recomputation*. Our cost-aware method ensures $peak_m$ to be consistent with the memory-centric strategy, while the extra computations are comparable to the speed-centric strategy.

Cost-Aware Recomputation finally reduces $peak_m$ to $max(l_i)$. Previously, *Liveness Analysis* and *Offloading* jointly reduce the $cost_k^b$ to $\sum_{i=1}^{k}(l_i^f \notin checkpoints) + l_k^b$. Since *non-checkpoints* layers will be freed for recomputations, only the nearest *checkpoint* layer exists in the GPU memory. Thus, $cost_k^b = l_{checkpoint}$.

Table 8.1: The counts of recomputations (extra) and $peak_m$ using the speed-centric, the memory-centric and Cost-Aware Recomputation.

	speed-centric		**memory-centric**		**cost-aware**	
	extra	$peak_m$	extra	$peak_m$	extra	$peak_m$
AlexNet	14	993.018	23	886.23	17	886.23
ResNet50	84	455.125	118	401	85	401
ResNet101	169	455.125	237	401	170	401

During the recomputations, $cost_k^b$ can be either $\sum_{i=1}^{k}(l_i^f) + l_k^b \leq l_{peak}$ or l_i^b depending what recomputation strategies to use. On the other hand, *Cost-Aware Recomputation* guarantees $cost_k^b \leq l_{peak} = max(l_i)$ (see analyses above). Thus, the final network wide $peak_m = max(cost_k^b) = l_{peak}$, which is the minimal $peak_m$ achievable at the layerwise granularity.

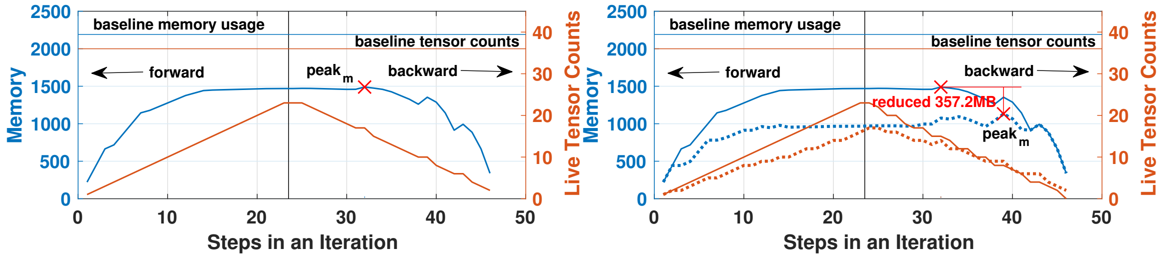
8.3.5 Finding the Best Convolution Algorithm under the Memory Constraint

The speed of CONV layers significantly impacts the training as it accounts for over 50% of total computing time (Fig.8.8). cuDNN provides several convolution algorithms, e.g. using FFT, Winograd and GEMM, for different contexts. Some of them, FFT in particular, require temporary convolution workspaces to delivery the maximal speed as demonstrated in Fig.8.2. Therefore, the memory is also a critical factor to the high-performance training.

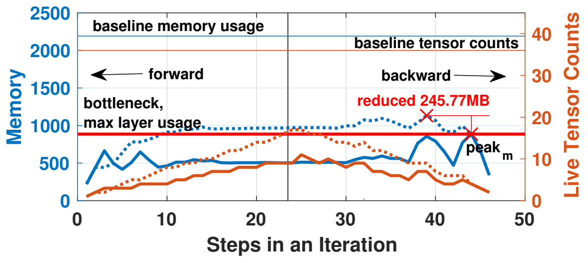
We implement a dynamic strategy for allocating convolution workspaces. It is dynamic because the memory left for convolution workspaces constantly changes in every step according to *Liveness Analysis*, *UTP* and *Cost-Aware Recomputation*. Since convolution workspaces do not affect the functionality, the allocations of functional tensors such as data and parameters are prioritized. The runtime then steps into each layer to profile the free bytes left in GPU DRAM after those memory techniques have been applied. With free memory information at individual steps, the runtime benchmarks all the memory-feasible convolution algorithms to pick the fastest one. Please note the runtime skips convolution algorithms that require more memory than it can provide. Each layer selects the fastest algorithm under the remaining GPU DRAM, and therefore maximize the performance of CONV layers and the entire training.

8.4 Evaluations

In this section, we present the results of our experimental studies that evaluate each of the memory and performance techniques in SuperNeurons. We also performed end-to-end evaluations against TensorFlow,



(a) liveness analysis (b) prefetching/offloading + liveness



(c) recomputation + previous two

Figure 8.10: The evaluations of *Liveness Analysis*, *Prefetching/Offloading* and *Cost-Aware Recomputation* on AlexNet at the batch size of 200. AlexNet has 23 layers, and a training iteration consists of 1 $\rightarrow$ 23 forward steps and 24 $\rightarrow$ 46 backward steps. The blue curve (left axis) depicts memory usages at each step, while the orange curve (right axis) depicts live tensor counts at each step. (a) demonstrates how *Liveness Analysis* affects memory usages w.r.t the baseline (horizontal lines). (b) demonstrates how *Offloading/Prefetching* improve *Liveness Analysis* by comparing the memory usages of both techniques (blue dashed lines in (b)) with *Liveness* alone (solid blue curve in (b)). Similarly, (c) demonstrates how *Cost-Aware Recomputation* improve the previous two; and dashed lines in (c) are from (b).

MXNet, Caffe and Torch on various neural networks to justify the design.

8.4.1 Components Evaluations

Memory Optimizations

We use the naive network-wide tensor allocation strategy as the baseline. Thus, the $peak_m$ of baseline is $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$, where N is the network length (defined in Sec.8.3). Since cuDNN operates at the layerwise granularity, $peak_m$ is bounded by the maximal memory usage among layers, i.e. l_{peak}.

Liveness Analysis reduces the baseline's $peak_m$ to $\sum_{i=1}^{N} l_i^f + l_N^b$. Fig.8.10a demonstrates how *Liveness Analysis* affect memory usages and live tensor counts at each forward/backward step on AlexNet. [3] Since

[3]the structure of AlexNet is CONV1→RELU1→LRN1→POOL1 →CONV2→RELU2 →LRN2→POOL2→CONV3→RELU3 →CONV4→RELU4→CONV5→RELU5→POOL5→FC1 →RELU6→Dropout1→FC2→RELU7→Dropout2 →FC3→Softmax

AlexNet has 23 layers, there are 23 forward steps and 23 backward steps. The central vertical line separates forward and backward, each of which contains 23 computational steps. The baseline allocates 36 data tensors consuming 2189.437MB, while *Liveness Analysis* uses up to 17 tensors with a peak memory usage of 1489.355MB. This demonstrates 31.9% improvement over the baseline in terms of $peak_m$. It is also observable that the location of $peak_m$ is not necessarily consistent with the peak tensor count. This confirms our claim that the memory is unevenly distributed across network layers.

To verify the cost model, i.e. $cost_k^b = \sum_{i=1}^{k} l_i^f + l_k^b$, we delve into the memory usages of the peak layer. Fig.8.10a suggests the 32th step reaches $peak_m$. This corresponds to the backward POOL5 in AlexNet, and $k = 14$ (or 46 - 32). The forward layers before POOL5 stash 5 tensors, consuming 1409.277MB ($\sum_{i=1}^{14} l_i^f$). Meanwhile the backward POOL5 stashes 3 tensors, consuming 80.078MB (l_{14}^b). Therefore, $cost_{14}^b = $ 1409.277 + 80.078 = 1489.355MB, which is consistent with the measured $peak_m$.

Prefetching and Offloading reduces the $peak_m$ after *Liveness Analysis* to $\sum_{i=1}^{N}(l_i^f \notin checkpoints)+l_N^b$. Fig.8.10b demonstrates the updated memory usages and live tensor counts after *Prefetching/Offloading* is applied on top of *Liveness Analysis*. We set CONV layers as checkpoints for offloading. The new $peak_m$ is 1132.155 MB at the 39th step or POOL2 backward. It further reduces 357.2MB off of the previous $peak_m$, for a total of 48.29% improvement over the baseline's $peak_m$. The new $peak_m$ shifts from POOL5 to POOL2 because of the number of CONV layers ahead of them. CONV1, CONV2, CONV3, and CONV4 are located before POOL5, and they consume 221.56MB, 142.38MB, 49.51MB and 49.51MB, respectively. The runtime offloads CONV $1 \sim 4$ to CPU RAM and prefetches CONV5. This leads the new memory usage of POOL5 to be 1489.355 - 221.56 - 142.38 - 49.51 = 1075.9MB, which is less than the measured new $peak_m$ 1132.155 MB at POOL2.

To verify the updated cost model, i.e. $\sum_{i=1}^{k}(l_i^f \notin checkpoints) + l_k^b$, we compare the calculated live tensor count from the model with the actual measurement. There are 2 checkpoints, CONV1 and CONV2, before POOL2, and the runtime prefetches CONV2 in the backward pass. As a result, the calculated live tensor count at POOL2 is 10 (measured live tensors before POOL2) - 1 (CONV1) = 9. This is the same as our actual measurement of 9 tensors at POOL2. Therefore, the updated cost model after *Prefetching/Offloading* is still valid.

Finally, ***Cost-Aware Recomputation*** reduces $peak_m$ to $\max(l_i)$. In theory, $\max(l_i)$ is the minimal $peak_m$ at the layerwise granularity as cuDNN needs to at least stash the tensors in a layer to compute. Fig.8.10c demonstrates stepwise memory usages and live tensor counts with all three techniques. We profile that $max(l_i) = 886.385MB$ at the backward LRN1 by iterating through every layer. Fig.8.10c demonstrates

Figure 8.11: Normalized performance with and without *Tensor Cache*. The batch size of AlexNet is 128, and 32 for the rest.

a $peak_m$ of 886 MB at the 44th step, which is the backward computation of LRN1. Therefore, the three proposed memory saving techniques successfully reduce the $peak_m$ from $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$ to $max(l_i)$.

Speed Optimizations

The runtime equips with a GPU Memory Pool and a Tensor Cache to improve the performance of memory techniques, and it also has a dynamic strategy for allocating convolution workspaces to accelerate the training speed. More specifically, the GPU Memory Pool amortizes the non-trivial overhead of high-frequency memory allocations/deallocations in *Liveness Analysis*, and Tensor Cache enables tensor reuse to minimize data transfers in *Prefetching/Offloading*. Fig.8.10c demonstrates how the GPU free space dynamically changes at each forward and backward step due to the three memory techniques. The runtime allocates convolution workspaces within the free memory at each step. As a result, the performance is optimized at individual layers under different stepwise memory constraints.

Table 8.2: The improvement of the GPU memory pool over cudaMalloc and cudaFree on various networks. The batch size for AlexNet is 128, while the rest is 16.

img/s	AlexNet	VGG16	InceptionV4	ResNet50	ResNet101	ResNet152
CUDA	359.4	12.1	6.77	21.5	11.3	7.46
Ours	401.6	14.4	10.0	32.9	18.95	13.2
speedup	1.12x	1.19x	1.48x	1.53x	1.68x	1.77x

Table 8.3: Communications with/without Tensor Cache. We benchmark the result on AlexNet by increasing the batch size from 256 to 1024.

Communications in GB	256	384	512	640	896	1024
Without Tensor Cache	2.56	3.72	4.88	6.03	8.35	9.50
Tensor Cache	0	0	0	0	0	0.88

GPU Memory Pool amortizes the non-trivial overhead of intensive memory operations in *Liveness Analysis* by preallocating a big chunk of GPU memory. Table 8.2 illustrates the performance improvement of using *GPU Memory Pool* over cudaMalloc and cudaFree. Linear networks such as AlexNet and VGG involve much fewer memory operations than nonlinear ones such as InceptionV4 and ResNet50 $\rightarrow$ 152 due to the limited depth. Therefore, the speedups on nonlinear networks (ResNet 50→152 and InceptionV4) are more significant than linear networks (AlexNet, VGG).

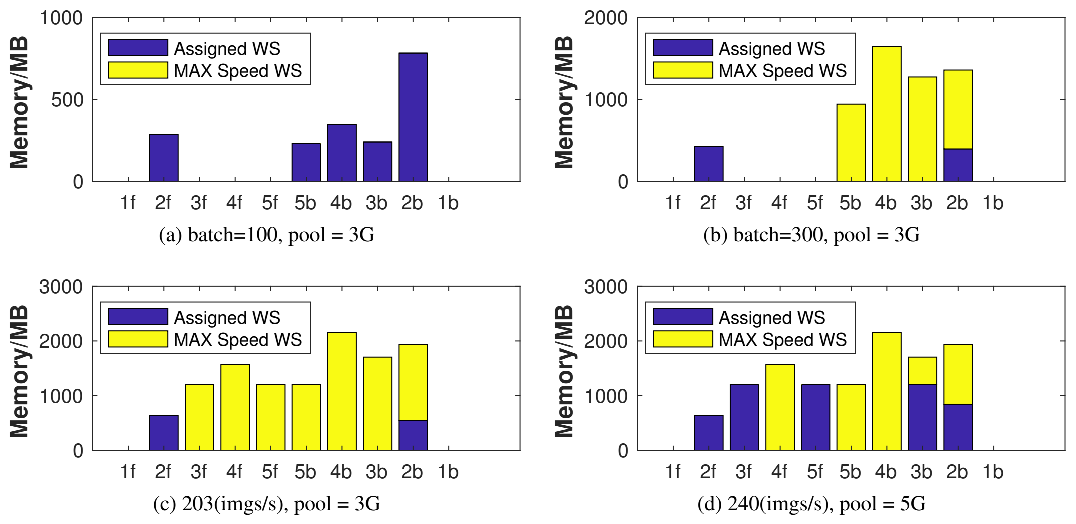


Figure 8.12: Dynamic Conv workspace allocations in the runtime. The digit in the x-axis represents the ith CONV layer, while the "f" or "b" represent the forward or backward computations.

Tensor Cache intends to reduce unnecessary data transfers in *Prefetching/Offloading*. Specifically, the offloading is unnecessary if a network can fit into the GPU DRAM. In Table 8.3, we can see *Tensor Cache* successfully avoids communications at batch sizes of 256 $\rightarrow$ 896, while the communications, in the scenario without *Tensor Cache*, linearly increase along batch sizes. The training performance will deteriorate if communications outweigh computations. Fig.8.11 demonstrates up to 33.33% performance loss without using *Tensor Cache*. It is also noticeable that the speedup on linear networks (AlexNet, VGG16) is less significant than nonlinear ones (ResNet50$\rightarrow$152, Inception). In general, the computation intensity of a linear network layer is far greater than the non-linear one. Because their communications can overlap with computations in *Prefetching/Offloading*, *Tensor Cache* does not provide the comparable speed up for AlexNet and VCG16.

Dynamic Convolution Workspace Allocation intends to optimize each layer's training speed together with the three memory techniques. Convolution workspaces are critical for high performance, while the amount of free memory for convolution workspaces constantly changes at different computing steps as demonstrated in Fig.8.10c. The runtime picks the fastest memory-feasible convolution algorithm at a particular step.

Fig.8.12a and Fig.8.12b demonstrate that the runtime automatically reduces CONV workspaces to accommodate functional tensors with the increasing batch size. Specifically, the runtime prioritizes the functional tensor allocations at batch 300 under 3 GB memory pool (Fig.8.12b), while it provisions the most workspace

Figure 8.13: Going Wider: the corresponding memory usages for the batch size in TABLE 8.5.

for the maximal speed at batch 100 (Fig.8.12a). In general, a higher speed is observable with more convolution workspaces. Fig.8.12c and Fig.8.12d demonstrate that the training speed (images per second) increases from 203 img/s to 240 img/s with additional CONV workspaces.

8.4.2 Going Deeper and Wider

Our primary goal is to enable ML practitioners to explore deeper and wider neural architectures within the limited GPU DRAM. In this section, we conduct end-to-end comparisions to TensorFlow, MXNet, Caffe and Torch with several mainstream linear networks (AlexNet, VGG16) and non-linear ones (ResNet50 $\rightarrow$ 150, Inception V4) under the same experiment setup.

Table 8.4: Going Deeper: the deepest ResNet that different frameworks can reach on a 12GB NVIDIA K40. The batch size is fixed at 16. ResNet has 4 for-loops to control its depth: $depth = 3*(n_1+n_2+n_3+n_4)+2$, where n_i is the upper limit of ith for-loop. We fix $n_1 = 6$, $n_2 = 32$, and $n_4 = 6$, while varying n_3 to increase the depth.

Depth	Caffe	MXNet	Torch	TensorFlow	SuperNeurons
ResNet	148	480	152	592	1920

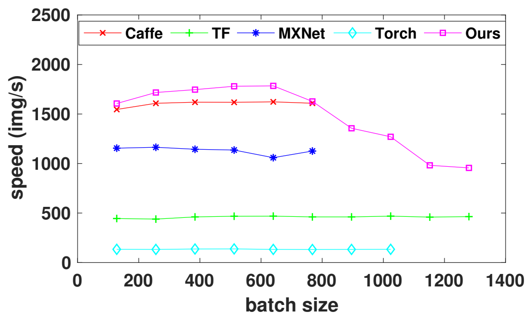
Table 8.5: Going Wider: the largest batch size that several mainstream neural architectures can reach in different frameworks with a 12GB NVIDIA K40.

peak batch	Caffe	MXNet	Torch	TensorFlow	SuperNeurons
AlexNet	768	768	1024	1408	1792
VGG16	48	64	48	80	224
InceptionV4	16	N/A	N/A	64	240
ResNet50	24	80	32	128	384
ResNet101	16	48	16	80	256
ResNet152	16	32	16	48	176

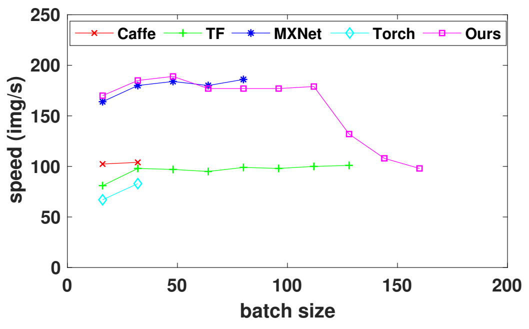
We increase the batch size to go wider. Table 8.5 presents the largest batch reachable by different frameworks before the GPU out-of-memory error. SuperNeurons consistently outperforms the mainstream frameworks on both linear and non-linear networks. On average, it handles 1.8947x larger batches than the second best. SuperNeurons can train ResNet101 at the batch of 256, which is 3x larger than the second best TensorFlow.

Fig.8.13 demonstrates the corresponding memory requirement to peak batches in Table 8.5. The translation is non-linear because of the convolution workspace. We calculate the memory requirement with $\sum_{i=1}^{N} l_i^f + \sum_{i=1}^{N} l_i^b$, and l_i is the sum of the memory usages of all tensors in the layer. It is observable that SuperNeurons handles up to a 19.8x larger model than Caffe.

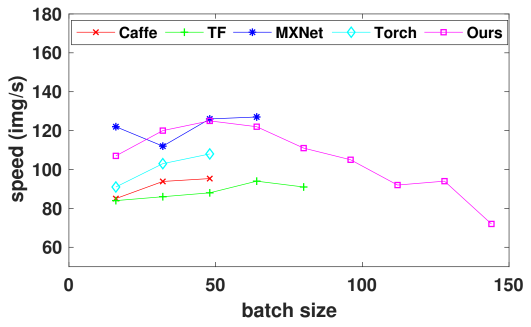
We add layers to go deeper. Table 8.4 demonstrates SuperNeurons trains 12.9730x, 12.6316x, 4.0000x, and 3.2432x deeper ResNet than Caffe, Torch, MXNet, and TensorFlow, respectively. Particularly, SuperNeurons can train a ResNet up to 2500 residual units having approximately 10^4 basic layers at the batch size of 1 on a 12GB GPU.



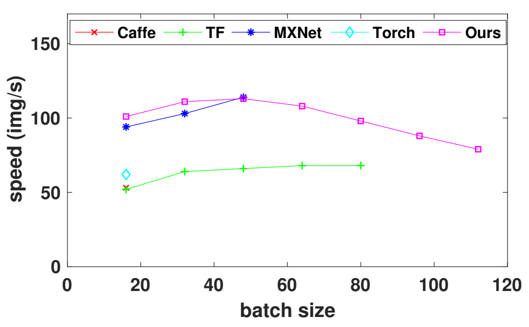
(a) AlexNet



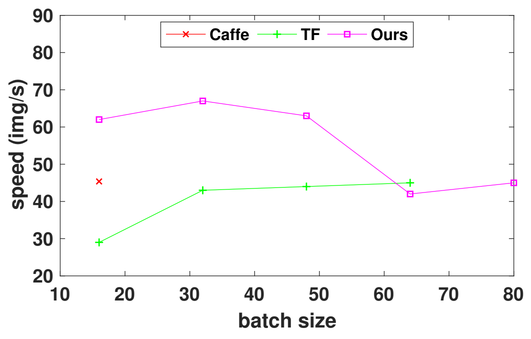
(b) ResNet50



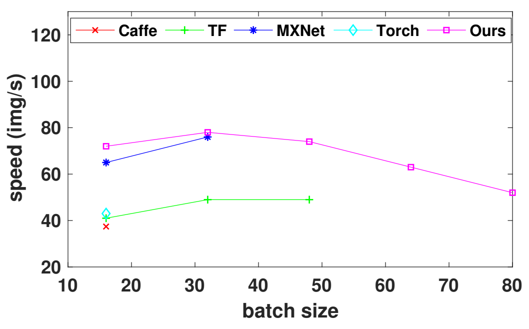
(c) VGG16



(d) ResNet101



(e) InceptionV4



(f) ResNet152

Figure 8.14: An end-to-end evaluation of different DL frameworks. We benchmark the data on a TITAN XP.

The training speed is measured by the number of processed images per second. Fig.8.14 presents an end-to-end training speed comparison of SuperNeurons to the mainstream DL systems. SuperNeurons consistently demonstrates the leading speed on various linear networks (AlexNet, VGG16) and nonlinear ones (ResNet50 $\rightarrow$ 152, Inception V4). The performance largely results from the abundant supply of convolution workspaces saved by the dynamic GPU memory scheduler. We can also observe that the speed has slowly deteriorated as batch size increases. This is due to the growing communications from more frequent tensor swapping between CPU and GPU DRAM. The performance will be at its worst when GPU memory can only accommodate one network layer. Then, the runtime has to constantly offload the current layer before proceeding to the next one.

8.5 Related Work

Several solutions have been proposed to address the GPU DRAM shortage for training large-scale neural networks. Model Parallelism provides a straightforward solution to the large network training. DistBelief [Dean et al., 2012] partitions a network across multiple machines so that each machine holds a segment of the original network. Coates et al [Coates et al., 2013] discuss another partition scheme on multi-GPUs. Model Parallelism demands huge intra-network communications for synchronization. Therefore, most DL systems parallelize the training with Data Parallelism for better performance [Jia et al., 2014, Abadi et al., 2016, Collobert et al., 2002]. In this paper, we focus on the GPU DRAM shortage issue for Data Parallelism.

Under Data Parallelism, vDNN [Rhu et al., 2016] proposes a prefetching and offloading technique to utilize the CPU DRAM as an external buffer for the GPU. It tries to overlap communications with computations by asynchronously swapping the data between CPU and GPU amid the back-propagation. The performance of this method largely depends on the communication/computation ratio. Some layers such as POOL are very cheap to compute, while the GPU processing speed is several orders of magnitude faster than the PCI-E 16x bus. In nonlinear networks, the performance will quickly deteriorate once computations are inadequate to overlap with communications. Chen et al [Chen et al., 2016] also introduces a recomputation strategy to trade computations for memory. However, their method fails to fully exploit the memory-saving opportunities and computation efficiency because it ignores the memory variations among layers.

Removing the parameter redundancy also reduces the memory usage. For example, network pruning [Han et al., 2016] removes near zero parameters, and quantization [Vanhoucke et al., 2011] or precision reduction [Judd et al., 2016] utilize low precision floats to save memory. Although the parameter reduction has immense benefits in deploying neural networks on embedded systems, parameters only account for a negligible portion of memory usage in the training. Therefore, these approaches are quite limited to training only.

8.6 Conclusion

In this paper, we focus on the GPU memory scheduling problem for training deep neural networks, and we propose a novel dynamic scheduling runtime to tackle the issue. The runtime features three memory techniques to reduce $peak_m$ to $max(l_i)$, which is the minimum at the layer-wise granularity. We also propose several performance optimizations to guarantee high performance. Evaluations against state-of-the-art DL frameworks have demonstrated the effectiveness and efficiency of proposed dynamic scheduling runtime. It creates new opportunities for DL practitioners to explore deeper and wider neural architectures, allowing the opportunity for better accuracy with even deeper and wider designs.

Part V

The Design Agent in Action

Chapter 9

Applications

We have tested our design agent in building several real-world applications, and the results consistently show that the neural network designed by the agent outperforms the manually created networks by the domain experts. For example, our agent designs the convolutional neural network for image recognition that reaches 99% top-1 accuracy on CIFAR-10, establishing new SoTA without using the extra data and exceeding the accuracy of ResNet-1001 [He et al., 2016] by 3.62%. Besides, on ImageNet, our agent designs an efficient network that reaches with SOTA 80.8% top-1 accuracy (under the mobile setting) at only 598M FLOPs, while the accuracy of manually designed ResNet-152 is 80.62% top1 but costing 18x more computations. The rest of this chapter shows the additional success of our design agent in designing recurrent neural networks, generative adversarial networks, and the backbone networks for object detection, image captioning, and neural style transfer. These data suggest that the agent proposed by this thesis can efficiently design various neural networks for different tasks.

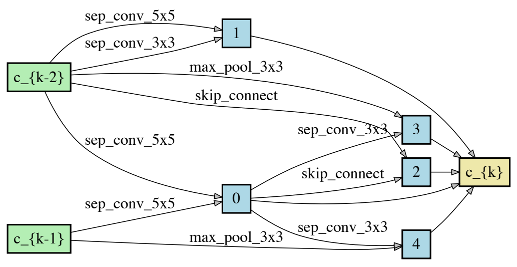
9.1 Designing Convolutional Neural Networks

To be consistent with existing literature, we also evaluate LaNAS in searching for architectures for CIFAR-10 using the NASNet search space and searching for architectures for ImageNet using the EfficientNet search space. Please refer to [Wang et al., 2019a] for more details of experiments in this section.

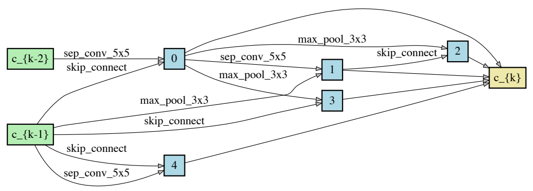
Table 9.1: Results on CIFAR-10 using the NASNet search space. LaNet-S and LaNet-L are same in structure, but the filter size of LaNet-S is 36, while LaNet-L is 128. Fig. 9.1 shows the structure of LaNet.

Model	**Data Augmentation**	**Extra Dataset**	**Params**	**Top1 err**	**M**	**GPU days**
			search based methods			
NASNet-A [Zoph et al., 2018b]	c/o	X	3.3 M	2.65	20000	2000
AmoebaNet-B-small [Real et al., 2019a]	c/o	X	2.8 M	$2.55_{\pm 0.05}$	27000	3150
AmoebaNet-B-large	c/o	X	34.9 M	$2.13_{\pm 0.04}$	27000	3150
PNASNet-5 [Liu et al., 2018a]	c/o	X	3.2 M	$3.41_{\pm 0.09}$	1160	225
NAO [Luo et al., 2018a]	c/o	X	128.0 M	2.11	1000	200
EfficientNet-B7	c/m+autoaug	ImageNet	64M	1.01		
GPipe [Huang et al., 2018]	c/m+autoaug	ImageNet	556M	1.00		
BiT-S	c/m+autoaug	ImageNet		2.49		
BiT-M	c/m+autoaug	ImageNet-21k	928M	1.09		
LaNet-S	c/o	X	3.2 M	$\mathbf{2.27}_{\pm 0.03}$	800	150
LaNet-S	c/m+autoaug	X	3.2 M	$\mathbf{1.63}_{\pm 0.05}$	800	150
LaNet-L	c/o	X	44.1 M	$\mathbf{1.53}_{\pm 0.03}$	800	150
LaNet-L	c/m+autoaug	X	44.1 M	$\mathbf{0.99}_{\pm 0.02}$	800	150
			one-shot NAS based methods			
ENAS [Pham et al., 2018a]	c/o	X	4.6 M	2.89	-	0.45
DARTS [Liu et al., 2018b]	c/o	X	3.3 M	$2.76_{\pm 0.09}$	-	1.5
BayesNAS [Zhou et al., 2019b]	c/o	X	3.4 M	$2.81_{\pm 0.04}$	-	0.2
P-DARTS ([Chen et al., 2019b])	c/o	X	3.4 M	2.5		0.3
PC-DARTS ([Xu et al., 2019])	c/o	X	3.6 M	$2.57_{\pm 0.07}$		0.3
CNAS ([Guo et al., 2020])	c/o	X	3.7 M	$2.6_{\pm 0.06}$		0.3
FairDARTS ([Chu et al., 2020])	c/o	X	3.32 M	$2.54_{\pm 0.05}$		3
ASNG-NAS [Akimoto et al., 2019b]	c/o+autoaug	X	3.9 M	$2.83_{\pm 0.14}$	-	0.11
XNAS [Nayman et al., 2019a]	c/o+autoaug	X	3.7 M	1.81		0.3
oneshot-LaNet-S	c/o	X	3.6 M	$\mathbf{2.24}_{\pm 0.02}$	-	3
oneshot-LaNet-S	c/o+autoaug	X	3.6 M	$\mathbf{1.68}_{\pm 0.06}$	-	3
oneshot-LaNet-L	c/o	X	45.3 M	$\mathbf{1.88}_{\pm 0.04}$	-	3
oneshot-LaNet-L	c/o+autoaug	X	45.3 M	$\mathbf{1.20}_{\pm 0.03}$	-	3

M: number of samples selected.
c/m: cutmix, c/o: cutout
autoaug: auto-augmentation



(a) normal cell



(b) reduction cell

Figure 9.1: The searched cell structure for LaNet.

9.1.1 CIFAR-10

On CIFAR-10, our search space is same as NASNet ([Zoph et al., 2018b]). We used operations of 3x3 max pool, 3x3, 5x5, depth-separable convolution, and skip connection. The search target is the architectures for a reduction and a normal cell, and the number of nodes within a cell is 5. This formulates a search space of

3.5×10^{21} architectures. The setup of supernet on CIFAR-10 is consistent with the description in sec.4.2.3; We selected the top architecture collected from the search and re-trained them for 600 epochs to acquire the final accuracy in Table. 9.1. We reused the training logic from `DARTS` and their training settings.

Table. 9.1 compares our results in the context of searching NASNet style architecture on CIFAR-10. The best performing architecture found by LaNAS in 800 samples demonstrates an average accuracy of 98.37% (#filters = 32, #params = 3.22M) and 99.01% (#filters = 128, #params = 44.1M), which is better than other results on CIFAR-10 without using ImageNet or transferring weights from a network pre-trained on ImageNet. It is worth noting that we achieved this accuracy with 33x fewer samples than AmoebaNet. Besides, one-shot LaNAS also consistently demonstrates the strongest result among other one-shot variants in similar GPU time. Besides, the results on ImageNet also consistently outperform SoTA models. The performance gap between one-shot NAS and search-based NAS is largely due to inaccurate predictions of v_i from supernet [Sciuto et al., 2019, Zhao et al., 2020]. We have introduced few-shot NAS to alleviate this issue at [Zhao et al., 2020].

Table 9.2: Transferring LaNet from CIFAR-10 to ImageNet using the NASNet search space.

Model	**FLOPs**	**Params**	**top1 err**
NASNet-C ([Zoph et al., 2018b])	558M	4.9 M	27.5
AmoebaNet-C ([Real et al., 2019a])	570M	6.4 M	24.3
RandWire ([Xie et al., 2019])	583M	5.6 M	25.3
PNASNet-5 ([Liu et al., 2018a])	588M	5.1 M	25.8
DARTS ([Liu et al., 2018b])	574M	4.7 M	26.7
BayesNAS ([Zhou et al., 2019b])	-	3.9 M	26.5
P-DARTS ([Chen et al., 2019b])	557M	4.9 M	24.4
PC-DARTS ([Xu et al., 2019])	597M	5.3 M	24.2
CNAS ([Guo et al., 2020])	576M	5.3 M	24.6
LaNet	570M	5.1 M	**23.5**
oneshot-LaNet	567M	5.4 M	**24.1**

Table 9.3: Results on ImageNet using the EfficientNet search space. The LaNet architecture can be found in Table. 9.4.

Model	**FLOPs**	**Params**	**GPU days**	**top1 err**
FairDARTS ([Chu et al., 2020])	440M	4.3 M	3	24.4
FBNetV2-C ([Wu et al., 2019])	375M	5.5 M	8.3	25.1
MobileNet-V3 ([Howard et al., 2019b])	219M	5.8 M		24.8
OFA ([Cai et al., 2019a])	230M	5.4 M	1.6	23.1
FBNetV2-F4([Wan et al., 2020a])	238M	5.6 M	8.3	24.0
FairDARTS ([Chu et al., 2020])	386M	5.3 M	3	22.8
BigNAS ([Yu et al., 2020a])	242M	4.5 M		23.5
LaNet	228M	5.1 M	0.3	**22.3**
OFA ([Cai et al., 2019a])	595M	9.1 M	1.6	20.0
BigNAS ([Yu et al., 2020a])	586M	6.4 M		23.5
FBNetV3 ([Dai et al., 2020])	544M			20.5
LaNet	598M	8.2 M	0.3	**19.2**

Table 9.4: LaNet architecture found on the EfficientNet search space, i.e. results in Table. 9.3.

ID	**Block**	**Kernel**	**Stride**	**Output Channel**	**Expand Ratio**
		LaNet@228M FLOPS			
0	Conv	3	2	24	
1	IRB	3	1	24	1
2	IRB	3	2	36	3
3	IRB	3	1	36	3
4	IRB	3	2	40	3
5	IRB	3	1	40	4
6	IRB	3	1	40	4
7	IRB	5	2	80	4
8	IRB	5	1	80	4
9	IRB	3	1	80	4
10	IRB	3	1	112	4
11	IRB	3	1	112	4
12	IRB	5	2	168	6
13	IRB	4	1	168	6
14	IRB	3	1	168	6
15	IRB	3	1	168	6
16	Conv	1	1	960	
17	Conv	1	1	1280	
18	FC	1	1	1000	
		LaNet@598M FLOPS			
0	Conv	3	2	24	
0	IRB	3	1	24	1
0	IRB	3	2	36	3
0	IRB	5	1	36	4
0	IRB	3	1	36	4
0	IRB	5	2	48	4
0	IRB	5	1	48	4
0	IRB	3	1	48	6
0	IRB	3	1	48	6
0	IRB	7	2	96	6
0	IRB	5	1	96	6
0	IRB	3	1	96	6
0	IRB	5	1	96	6
0	IRB	3	1	136	6
0	IRB	5	1	136	6
0	IRB	5	1	136	6
0	IRB	3	1	136	6
0	IRB	3	2	200	6
0	IRB	5	1	200	6
0	IRB	5	1	200	6
0	IRB	3	1	200	6
16	Conv	1	1	1152	
16	Conv	1	1	1536	
18	FC	1	1	1000	

9.1.2 ImageNet

We also transfer the architecture found on CIFAR-10 to ImageNet, and Table. 9.2 demonstrates the ImageNet result using the NASNet search space. The NASNet result is not competitive, as the network is not directly searched on ImageNet, and the non-linear connections of NASNet significantly increase the latency and memory footprint. Therefore, the simplified, linearly connected Efficient-Net search space is more popular on the ImageNet that only searches for a few hyper-parameters inside the Inverted Residual Block (IRB). Specifically, the depth of an Inverted Residual Block (IRB) can vary in 2, 3, 4; the expansion ratio of an IRB is within 3, 5, 7; and the stride is within 1, 2. We show the final results in Table. 9.3.

9.2 Designing Recurrent Neural Networks

Table 9.5: Comparison with state-of-the-art language models on PTB (lower perplexity is better).

Architecture	**Perplexity**	**Params**	**GPU days)**
Variational RHN [Zilly et al., 2017]	65.4	23	–
LSTM [Merity et al., 2017]	58.8	24	–
LSTM + 5 softmax experts [Yang et al., 2017]	57.4	–	–
NAS [Zoph and Le, 2016]	64.0	25	1e4
ENAS [Pham et al., 2018a]	63.1	24	0.5
Random search baseline	59.4	23	2
DARTS [Liu et al., 2018b]	55.7	23	1
Ours	54.8	23	1.56

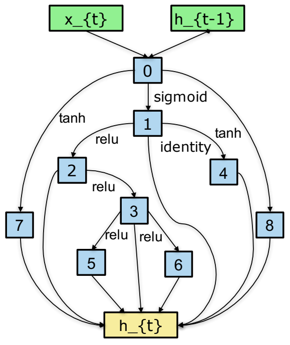
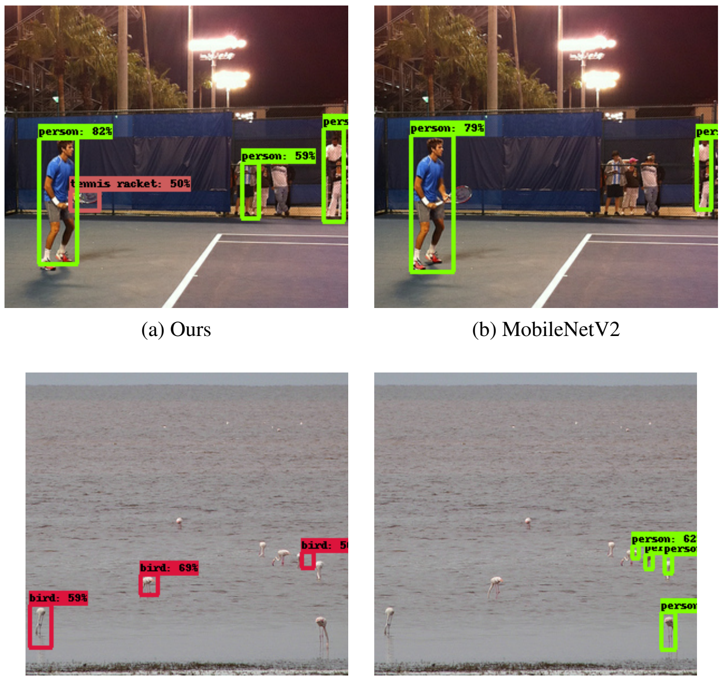


Figure 9.2: The structure of our searched RNN cell.

Here the agent is set to design an RNN for the languages modeling on Penn Treebank dataset [Marcus et al., 1994]. This experiment in this section is from the paper of [Zhao et al., 2020]. Specifically, please refer to section 2.1 in [Pham et al., 2018a] for the details of RNN design space. We prefix the embedding and the hidden size of RNN to 300. The batch size is 256, and the weight decay is 3×5^{-7}. The dropout ratio for the word embeddings is 0.2, 0.75 for cell input, 0.25 for all the hidden nodes, and 0.75 for the output layer. Finally, our agent finds an RNN cell structure that achieved the state-of-the-art test Perplexity of 54.89 with an overall cost of 1.56 GPU days, which leads to the results in Table. 9.5. The cell structure of searched RNN is in Fig. 9.4.

9.3 Designing Backbone for Detection Systems



(a) Ours (b) MobileNetV2

(c) Ours (d) MobileNetV2

Figure 9.3: The visualization of differences in Table 9.6.

Decoder	Backbone	#Channels	FLOPS(G)	AP
FPN-FCOS [Tian et al., 2019b]	R-50 [He et al., 2016]	256	169.9	37.4
NAS-FCOS [Wang et al., 2020b]	MobileNetV2 [Sandler et al., 2018]	128	39.3	32.0
NAS-FCOS [Wang et al., 2020b]	MobileNetV2 [Sandler et al., 2018]	256	121.8	34.7
FPN-FCOS [Wang et al., 2020b]	MobileNetV2 [Sandler et al., 2018]	256	105.4	31.2
FPN-FCOS [Tian et al., 2019b]	Ours	128	**35.2**	**36.5**
FPN-FCOS [Tian et al., 2019b]	Ours	256	**109.5**	**37.6**

Table 9.6: Results on test-dev set of MS COCO with different decoder, backbone and channels. R-50 abbreviates for ResNet50. All networks have the same input image resolution.

We also use our agent to search the backbone CNN for the detection system. We reuse the Efficient-Net search space described in section 9.1.2 to search for the backbone, and we use the FPN-FCOS [Tian et al., 2019b] as the decoder to train on the COCO dataset. FPN-FCOS consists of three components, a backbone, a FPN, and prediction heads. Since objects of different scales require different effective receptive fields, the input of FPN is a set of feature maps with different channels and resolutions generated from the backbone. Specifically, the size the of feature maps are (h_i, w_i, c_i), where $(h_i, w_i) = (h/2^i, w/2^i)$ given a base h and w.

Following the existing literatures [Tian et al., 2019b, Wang et al., 2020b], we set i to four and extract the four feature maps that have different resolutions and channel sizes from our searched model, and use the output tensors as the input of FPN. We set the channel numbers of FPN and prediction heads to either 128 or 256 in our experiments. We use the standard SGD optimizer with an initial learning rate of 0.005 and a norm gradient clip at 35. We train each model for 24 epochs and use the first 500 iterations as the warm-up phase. During the warm-up iterations, the learning rate starts at 0.002 and increases by 0.00033 every 50 iterations until it reaches 0.005. After the warm-up phase, we divide the learning rate by 10 at the 10^{th} and 22^{nd} epochs (i.e. to 5×10^{-4} and 5×10^{-5} respectively). We resize each image to 1333 by 800 and implement a random flip with 0.5 flip ratios. Further, we apply both the center-ness and center sampling techniques on training, following the suggestion from [Tian et al., 2019b, Wang et al., 2020b]. Table. 9.6 demonstrates the detection result on MS COCO using our searched backbone. At 256 channels, our model outperforms ResNet50 at the same setting by 0.2 AP and 60.4 G (35.6%) fewer FLOPS. Particularly, our model increases the AP scores of MobileNet backed NAS-FCOS from 32 to 36.5 in similar ($\sim$35) FLOPS.

9.4 Designing Generative Adversarial Networks

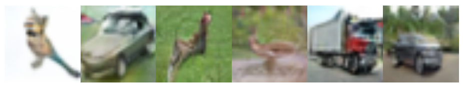
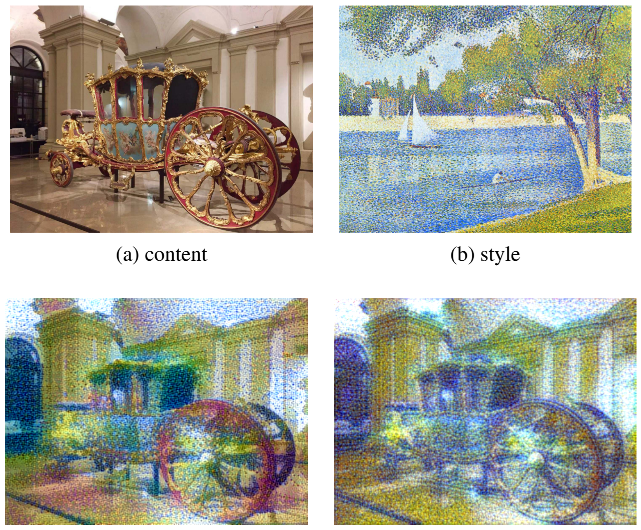


Figure 9.4: The randomly sampled CIFAR-10 images from the GAN designed by our agent.

Table 9.7: Designing the Generative Adversarial Network (GAN) to generate CIFAR-10 style images. The inception score is higher the better, while the FID score is lower the better.

Method	**Inception Score**	**FID Score**
ProbGAN[He et al., 2019a]	7.75±.14	24.60
SN-GAN[Miyato et al., 2018]	8.22±.05	21.70±.01
MGAN[Hoang et al., 2018]	8.33±.12	26.7
Improving MMD GAN[Wang et al., 2019d]	8.29	16.21
AutoGAN-top1[Gong et al., 2019]	8.55±.10	12.42
AutoGAN-top2	8.42±.06	13.67
AutoGAN-top3	8.41±.12	13.87
Ours-top1	8.60±.10	**10.73±.10**
Ours-top2	**8.63±.09**	10.89±.20
Ours-top3	8.52±.08	12.20

Here the agent is set to design GAN that consists of a generator and a discriminator network to generate



(a) content (b) style

(c) VGG (d) AlphaX-1

Figure 9.5: The neural style transfer using the network designed by our agent V.S. VGG.

CIFAR-10 style images. For more information, please refer to our few-shot NAS paper [Zhao et al., 2020]. We set up the search space of GAN following the AUTOGAN [Gong et al., 2019]. We set the learning rate of both generator and discriminator to $2e^{-4}$, and we use the hinge loss and Adam optimizer. The batch size of the discriminator is 64, and the generator is 128. The initial learning rate was set to $3.5e^{-4}$. Table 9.7 shows the top three performing GANs found by ours, other NAS algorithms in the literature, and the results of manually defined GAN. The result indicates that we improve the inception score from 8.55 to 8.63 and reduce the FID from 12.42 to 10.73, demonstrating a 16.8% improvement.

9.5 Designing Neural Style Transfer System

Here we set the agent to perform the neural style transfer by designing a CNN to replace the VGG model [Gatys et al., 2015]. We did this experiment in [Wang et al., 2019c]. We searched the network on the NASNet search space, then pre-trained the searched network on the ImageNet dataset. We set 10 as the style weight that represents the extent of style reconstruction and 0.025 as the content weight representing the extent of content reconstruction. We also search on the different combinations of the layers' outputs. Fig. 9.5 suggests that our searched network is better than VGG, especially in capturing the rich details and textures of a sophisticated style image.

9.6 NeurIPS-2020 Black Box Optimization Challenges

While demonstrating many successful cases in NAS above, the LA-MCTS proposed in this thesis is a generic solver applying to a wide range of optimization problems. In NeurIPS 2020, companies that are actively engaging in the field of black-box optimization held a contest to find the best black box solver for machine learning [bbo, 2020]. This competition has a widespread impact as black-box optimization is relevant for hyper-parameter tuning in almost every machine learning project, especially deep learning and many applications outside of machine learning such as A/B testing, optimizing system configurations many others. Our LA-MCTS is an awarding-winning algorithm in this contest. Specifically, JetBrains research and KAIST independently replicated the concept of ”learning search space partition” in LA-MCTS and won 3rd and 8th places among 68 global participating teams, including companies and institutions such as NVIDIA and Huawei Oxford, KAIST, IBM, Preferred Networks, and Innovatrics. The contest thoroughly and rigorously evaluates each team’s search algorithm on 216 different ML tasks for multiple runs to report the final result. These public results shall be a strong demonstration of LA-MCTS to solve many real-world problems.

Success is not final, failure is not fatal: it is the courage to continue that counts.
Winston Churchill

Bibliography

[MXN,] Mxnet's graph representation of neural networks. `http://mxnet.io/architecture/note_memory.html`.

[bbo, 2020] (2020). Black box optimization challenge. `https://bbochallenge.com/`.

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

[Aji and Heafield, 2017] Aji, A. F. and Heafield, K. (2017). Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*.

[Akimoto et al., 2019a] Akimoto, Y., Shirakawa, S., Yoshinari, N., Uchida, K., Saito, S., and Nishida, K. (2019a). Adaptive stochastic natural gradient method for one-shot neural architecture search. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 171–180.

[Akimoto et al., 2019b] Akimoto, Y., Shirakawa, S., Yoshinari, N., Uchida, K., Saito, S., and Nishida, K. (2019b). Adaptive stochastic natural gradient method for one-shot neural architecture search. *arXiv preprint arXiv:1905.08537*.

[Alabi et al., 2012] Alabi, T., Blanchard, J. D., Gordon, B., and Steinbach, R. (2012). Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–2.

[Albert and Barabási, 2002] Albert, R. and Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47.

[Alistarh et al., 2017] Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. (2017). Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*.

[Alistarh et al., 2018] Alistarh, D., Hoefler, T., Johansson, M., Konstantinov, N., Khirirat, S., and Renggli, C. (2018). The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*, pages 5975–5985.

[Arora et al., 2019] Arora, S., Li, Z., and Lyu, K. (2019). Theoretical analysis of auto rate-tuning by batch normalization. In *International Conference on Learning Representations*.

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

[Awan et al., 2017] Awan, A. A., Hamidouche, K., Hashmi, J. M., and Panda, D. K. (2017). S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[Bahrampour et al., 2016] Bahrampour, S., Ramakrishnan, N., Schott, L., and Shah, M. (2016). Comparative study of caffe, neon, theano, and torch for deep learning.

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

[Baker et al., 2017a] Baker, B., Gupta, O., Naik, N., and Raskar, R. (2017a). Designing neural network architectures using reinforcement learning. *International Conference on Learning Representations*.

[Baker et al., 2017b] Baker, B., Gupta, O., Raskar, R., and Naik, N. (2017b). Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*.

[BBContest, 2017] BBContest (2017). *Title*. `https://bbochallenge.com/`.

[Belew and McInerney, 1989] Belew, R. K. and McInerney, J. (1989). Using the genetic algorithm to wire feed-forward networks. *Technical abstract, University of California, San Diego, La Jolla, CA*.

[Bélisle et al., 1993] Bélisle, C. J., Romeijn, H. E., and Smith, R. L. (1993). Hit-and-run algorithms for generating multivariate distributions. *Mathematics of Operations Research*, 18(2):255–266.

[Bellman, 1957] Bellman, R. (1957). A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684.

[Bellman, 1966] Bellman, R. (1966). Dynamic programming. *Science*, 153(3731):34–37.

[Bender et al., 2018] Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. (2018). Understanding and simplifying one-shot architecture search. In *Proceedings of the 35th International Conference on Machine Learning*.

[Bengio et al., 2006] Bengio, Y., Le Roux, N., Vincent, P., Delalleau, O., and Marcotte, P. (2006). Convex neural networks. *Advances in neural information processing systems*, 18:123.

[Bengio et al., 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

[Bergstra et al., 2012] Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2012). Algorithms for hyper-parameter optimization. *Nips*.

[Bergstra and Bengio, 2012] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2).

[Bergstra et al., 2011] Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554.

[Berson, 1992] Berson, A. (1992). *Client-server architecture*. Number IEEE-802. McGraw-Hill.

[Binois et al., 2015] Binois, M., Ginsbourger, D., and Roustant, O. (2015). A warped kernel improving robustness in bayesian optimization via random embeddings. In *International Conference on Learning and Intelligent Optimization*, pages 281–286. Springer.

[Binois et al., 2020] Binois, M., Ginsbourger, D., and Roustant, O. (2020). On the choice of the low-dimensional domain for global optimization via random embeddings. *Journal of global optimization*, 76(1):69–90.

[Brochu et al., 2010] Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

[Brown and Sandholm, 2019] Brown, N. and Sandholm, T. (2019). Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[Browne et al., 2012] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43.

[Bubeck et al., 2011] Bubeck, S., Munos, R., Stoltz, G., and Szepesvári, C. (2011). X-armed bandits. *Journal of Machine Learning Research*, 12(May):1655–1695.

[Buşoniu et al., 2013] Buşoniu, L., Daniels, A., Munos, R., and Babuška, R. (2013). Optimistic planning for continuous-action deterministic systems. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 69–76. IEEE.

[Cai et al., 2019a] Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. (2019a). Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*.

[Cai et al., 2020] Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. (2020). Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*.

[Cai et al., 2019b] Cai, H., Zhu, L., and Han, S. (2019b). ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*.

[Carbonnelle and Vleeschouwer, 2018] Carbonnelle, S. and Vleeschouwer, C. D. (2018). On layer-level control of DNN training and its impact on generalization. *CoRR*, abs/1806.01603.

[Cassandra, 1998] Cassandra, A. R. (1998). A survey of pomdp applications. In *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*, volume 1724.

[Changeux, 1980] Changeux, J.-P. (1980). Genetic determinism and epigenesis of the neuronal network: Is there a biological compromise between chomsky and piaget. *Language and learning*.

[Chau et al., 2020] Chau, T., Dudziak, Ł., Abdelfattah, M. S., Lee, R., Kim, H., and Lane, N. D. (2020). Brp-nas: Prediction-based nas using gcns. *arXiv preprint arXiv:2007.08668*.

[Chen et al., 2012] Chen, B., Castro, R., and Krause, A. (2012). Joint optimization and variable selection of high-dimensional gaussian processes. *arXiv preprint arXiv:1206.6396*.

[Chen et al., 2015] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

[Chen et al., 2016] Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

[Chen et al., 2019a] Chen, X., Xie, L., Wu, J., and Tian, Q. (2019a). Progressive DARTS: bridging the optimization gap for NAS in the wild. *CoRR*, abs/1912.10952.

[Chen et al., 2019b] Chen, X., Xie, L., Wu, J., and Tian, Q. (2019b). Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1294–1303.

[Chetlur et al., 2014] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*.

[Chu et al., 2019a] Chu, X., Zhang, B., Xu, R., and Li, J. (2019a). Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *CoRR*, abs/1907.01845.

[Chu et al., 2019b] Chu, X., Zhou, T., Zhang, B., and Li, J. (2019b). Fair DARTS: eliminating unfair advantages in differentiable architecture search. *CoRR*, abs/1911.12126.

[Chu et al., 2020] Chu, X., Zhou, T., Zhang, B., and Li, J. (2020). Fair darts: Eliminating unfair advantages in differentiable architecture search. In *European Conference on Computer Vision*, pages 465–480. Springer.

[Coates et al., 2013] Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345.

[Collobert et al., 2002] Collobert, R., Bengio, S., and Mariéthoz, J. (2002). Torch: a modular machine learning software library. Technical report, Idiap.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

[Cubuk et al., 2018] Cubuk, E. D., Zoph, B., Mané, D., Vasudevan, V., and Le, Q. V. (2018). Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501.

[Dai et al., 2020] Dai, X., Wan, A., Zhang, P., Wu, B., He, Z., Wei, Z., Chen, K., Tian, Y., Yu, M., Vajda, P., et al. (2020). Fbnetv3: Joint architecture-recipe search using neural acquisition function. *arXiv preprint arXiv:2006.02049*.

[Dai et al., 2019] Dai, X., Zhang, P., Wu, B., Yin, H., Sun, F., Wang, Y., Dukhan, M., Hu, Y., Wu, Y., Jia, Y., Vajda, P., Uyttendaele, M., and Jha, N. K. (2019). Chamnet: Towards efficient network design through platform-aware model adaptation. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11390–11399.

[Darwin, 1909] Darwin, C. (1909). *The origin of species*. PF Collier & son New York.

[De Sa et al., 2015] De Sa, C. M., Zhang, C., Olukotun, K., and Ré, C. (2015). Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in neural information processing systems*, pages 2674–2682.

[Dean et al., 2012] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

[Deb et al., 2002] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Devries and Taylor, 2017] Devries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552.

[Di and Cappello, 2016] Di, S. and Cappello, F. (2016). Fast error-bounded lossy hpc data compression with sz. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[DMV, 2020] DMV, C. (2020). *Title*. `https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/disengagement-reports/`.

[Dong and Yang, 2019] Dong, X. and Yang, Y. (2019). One-shot neural architecture search via self-evaluated template network. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[Dong and Yang, 2020] Dong, X. and Yang, Y. (2020). Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations (ICLR)*.

[Du et al., 2018] Du, S., Lee, J., Tian, Y., Singh, A., and Poczos, B. (2018). Gradient descent learns one-hidden-layer cnn: Don't be afraid of spurious local minima. In *International Conference on Machine Learning*, pages 1339–1348. PMLR.

[Erdős and Rényi, 1960] Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60.

[Eriksson et al., uRBO] Eriksson, D., Pearce, M., Gardner, J., Turner, R. D., and Poloczek, M. (2019, the implementation is from *https://github.com/uber-research/TuRBO*.). Scalable global optimization via local bayesian optimization. In *Advances in Neural Information Processing Systems*, pages 5497–5508.

[Falkner et al., 2018] Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning*.

[Falkner et al., Ster] Falkner, S., Klein, A., and Hutter, F. (2018, the implementation is from *https://github.com/automl/HpBandSter*). Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*.

[Fischer et al., 2015] Fischer, L., Gao, S., and Bernstein, A. (2015). Machines tuning machines: Configuring distributed stream processors with bayesian optimization. In *2015 IEEE International Conference on Cluster Computing*, pages 22–31. IEEE.

[Fouladi et al., 2017] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. (2017). Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 363–376.

[Frazier, 2018] Frazier, P. I. (2018). A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*.

[Gabriel et al., 2004] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. (2004). Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*.

[Gardner et al., 2017] Gardner, J., Guo, C., Weinberger, K., Garnett, R., and Grosse, R. (2017). Discovering and exploiting additive structure for bayesian optimization. In *Artificial Intelligence and Statistics*, pages 1311–1319.

[Gardner et al., 2014] Gardner, J. R., Kusner, M. J., Xu, Z. E., Weinberger, K. Q., and Cunningham, J. P. (2014). Bayesian optimization with inequality constraints. In *ICML*, pages 937–945.

[Gatys et al., 2015] Gatys, L. A., Ecker, A. S., and Bethge, M. (2015). A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*.

[Gelfand and Smith, 1990] Gelfand, A. E. and Smith, A. F. (1990). Sampling-based approaches to calculating marginal densities. *Journal of the American statistical association*, 85(410):398–409.

[Ghadimi and Lan, 2013] Ghadimi, S. and Lan, G. (2013). Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization*, 23(4):2341–2368.

[Gilks et al., 1995] Gilks, W. R., Best, N. G., and Tan, K. (1995). Adaptive rejection metropolis sampling within gibbs sampling. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 44(4):455–472.

[Goldstein, 1977] Goldstein, A. (1977). Optimization of lipschitz continuous functions. *Mathematical Programming*, 13(1):14–22.

[Gong et al., 2019] Gong, X., Chang, S., Jiang, Y., and Wang, Z. (2019). Autogan: Neural architecture search for generative adversarial networks. In *The IEEE International Conference on Computer Vision (ICCV)*.

[Guo et al., 2020] Guo, Y., Chen, Y., Zheng, Y., Zhao, P., Chen, J., Huang, J., and Tan, M. (2020). Breaking the curse of space explosion: Towards efficient nas with curriculum search. In *International Conference on Machine Learning*, pages 3822–3831. PMLR.

[Guo et al., 2019a] Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., and Sun, J. (2019a). Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*.

[Guo et al., 2019b] Guo, Z., Zhang, X., Mu, H., Heng, W., Liu, Z., Wei, Y., and Sun, J. (2019b). Single path one-shot neural architecture search with uniform sampling. *CoRR*, abs/1904.00420.

[Han et al., 2016] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. (2016). Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press.

[Han et al., 2015] Han, S., Mao, H., and Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[Hansen, 2006] Hansen, N. (2006). The cma evolution strategy: a comparing review. *Towards a new evolutionary computation*, pages 75–102.

[Hansen et al., ycma] Hansen, N., Müller, S. D., and Koumoutsakos, P. (2003, the implementation is from *https://github.com/CMA-ES/pycma*). Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18.

[He et al., 2019a] He, H., Wang, H., Lee, G.-H., and Tian, Y. (2019a). Probgan: Towards probabilistic gan with theoretical guarantees. In *International Conference on Learning Representations(ICLR)*.

[He et al., 2019b] He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. (2019b). Momentum contrast for unsupervised visual representation learning. *arXiv preprint arXiv:1911.05722*.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer.

[Hensman et al., 2013] Hensman, J., Fusi, N., and Lawrence, N. D. (2013). Gaussian processes for big data. *arXiv preprint arXiv:1309.6835*.

[Hoang et al., 2018] Hoang, Q., Nguyen, T. D., Le, T., and Phung, D. (2018). MGAN: Training generative adversarial nets with multiple generators. In *International Conference on Learning Representations*.

[Hörmann, 1995] Hörmann, W. (1995). A rejection technique for sampling from t-concave distributions. *ACM Transactions on Mathematical Software (TOMS)*, 21(2):182–193.

[Howard et al., 2019a] Howard, A., Sandler, M., Chu, G., Chen, L., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. (2019a). Searching for mobilenetv3. *CoRR*, abs/1905.02244.

[Howard et al., 2019b] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. (2019b). Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324.

[Huang et al., 2016] Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. (2016). Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*.

[Huang et al., 2018] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2018). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*.

[Hutter et al., 2011a] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011a). Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer.

[Hutter et al., 2011b] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011b). Sequential model-based optimization for general algorithm configuration. In *Proceedings of the conference on Learning and Intelligent OptimizatioN (LION 5)*.

[Huxley, 1860] Huxley, T. (1860). Art. viii.-darwin on the origin of species. *Westminster Review*, pages 541–70.

[Jia et al., 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM.

[Jia et al., 2019] Jia, Y., Weiss, R. J., Biadsy, F., Macherey, W., Johnson, M., Chen, Z., and Wu, Y. (2019). Direct speech-to-speech translation with a sequence-to-sequence model. *arXiv preprint arXiv:1904.06037*.

[Jin et al., 2019] Jin, S., Di, S., Liang, X., Tian, J., Tao, D., and Cappello, F. (2019). DeepSZ: A Novel Framework to Compress Deep Neural Networks by Using Error-Bounded Lossy Compression. HPDC'19, page 159–170, New York, NY, USA. Association for Computing Machinery.

[Jin, 2011] Jin, Y. (2011). Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70.

[Jin and Branke, 2005] Jin, Y. and Branke, J. (2005). Evolutionary optimization in uncertain environments-a survey. *IEEE Transactions on evolutionary computation*, 9(3):303–317.

[Jones, 2001] Jones, D. R. (2001). A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383.

[Jones et al., 1998] Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492.

[Judd et al., 2016] Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., Jerger, N. E., and Moshovos, A. (2016). Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, page 23. ACM.

[Kandasamy et al., 2015] Kandasamy, K., Schneider, J., and Póczos, B. (2015). High dimensional bayesian optimisation and bandits via additive models. In *International Conference on Machine Learning*, pages 295–304.

[Kawaguchi et al., 2015] Kawaguchi, K., Kaelbling, L. P., and Lozano-Pérez, T. (2015). Bayesian optimization with exponential convergence. In *Advances in neural information processing systems*, pages 2809–2817.

[Kendall., 1938] Kendall., M. G. (1938). A new measure of rank correlation.

[Kim et al., 2020a] Kim, B., Lee, K., Lim, S., Kaelbling, L. P., and Lozano-Pérez, T. (2020a). Monte carlo tree search in continuous spaces using voronoi optimistic optimization with regret bounds. In *AAAI*, pages 9916–9924.

[Kim et al., 2020b] Kim, T., Ahn, J., Kim, N., and Yun, S. (2020b). Adaptive local bayesian optimization over multiple discrete variables. *arXiv preprint arXiv:2012.03501*.

[Kitano, 1990] Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex systems*, 4:461–476.

[Kocsis and Szepesvári, 2006] Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

[Köster et al., 2017] Köster, U., Webb, T., Wang, X., Nassar, M., Bansal, A. K., Constable, W., Elibol, O., Gray, S., Hall, S., Hornof, L., Khosrowshahi, A., Kloss, C., Pai, R. J., and Rao, N. (2017). Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 1742–1752. Curran Associates, Inc.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Lan et al., 2019] Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Levine et al., 2016] Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373.

[Li et al., 2017] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816.

[Li et al., 2018a] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2018a). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*.

[Li et al., 2014] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. In *OSDI*.

[Li et al., 2018b] Li, Y., Vinyals, O., Dyer, C., Pascanu, R., and Battaglia, P. (2018b). Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*.

[Liaw et al., 2018] Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. (2018). Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

[Lim et al., 2020] Lim, H., Kim, M.-S., and Xiong, J. (2020). {CNAS}: Channel-level neural architecture search.

[Lin et al., 2017] Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. (2017). Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*.

[Liu et al., 2018a] Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018a). Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34.

[Liu et al., 2017] Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. (2017). Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

[Liu et al., 2018b] Liu, H., Simonyan, K., and Yang, Y. (2018b). Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

[Liu et al., 2019] Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: Differentiable architecture search. In *International Conference on Learning Representations(ICLR)*.

[Liu et al., grad] Liu, J., Moreau, A., Preuss, M., Roziere, B., Rapin, J., Teytaud, F., and Teytaud, O. (2020, the implementation is from *https://github.com/facebookresearch/nevergrad*). Versatile black-box optimization. *arXiv preprint arXiv:2004.14014*.

[Lu et al., 2019] Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., and Banzhaf, W. (2019). Nsga-net: neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 419–427.

[Luo et al., 2020] Luo, R., Qin, T., and Chen, E. (2020). Balanced one-shot neural architecture optimization. abs/1909.10815.

[Luo et al., 2018a] Luo, R., Tian, F., Qin, T., Chen, E., and Liu, T.-Y. (2018a). Neural architecture optimization. *arXiv preprint arXiv:1808.07233*.

[Luo et al., 2018b] Luo, R., Tian, F., Qin, T., Chen, E., and Liu, T.-Y. (2018b). Neural architecture optimization. In *Advances in Neural Information Processing Systems 31*.

[Ma et al., 2018] Ma, N., Zhang, X., Zheng, H., and Sun, J. (2018). Shufflenet V2: practical guidelines for efficient CNN architecture design. *CoRR*, abs/1807.11164.

[Mallows, 1991] Mallows, C. (1991). Letters to the editor. *The American Statistician*, 45(3):256–262.

[Mania et al., sARS] Mania, H., Guy, A., and Recht, B. (2018, the implementation is from https://github.com/modestyachts/ARS). Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*.

[Mansley et al., 2011] Mansley, C., Weinstein, A., and Littman, M. (2011). Sample-based planning for continuous action markov decision processes. In *Twenty-First International Conference on Automated Planning and Scheduling*.

[Marcus et al., 1994] Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. (1994). The penn treebank: Annotating predicate argument structure. In *HUMAN LANGUAGE TECHNOLOGY: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*.

[McIntire et al., 2016] McIntire, M., Ratner, D., and Ermon, S. (2016). Sparse gaussian processes for bayesian optimization. In *UAI*.

[McKay et al., 2000] McKay, M. D., Beckman, R. J., and Conover, W. J. (2000). A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61.

[McKinney et al., 2020] McKinney, S. M., Sieniek, M., Godbole, V., Godwin, J., Antropova, N., Ashrafian, H., Back, T., Chesus, M., Corrado, G. C., Darzi, A., et al. (2020). International evaluation of an ai system for breast cancer screening. *Nature*, 577(7788):89–94.

[Mei et al., 2020] Mei, J., Li, Y., Lian, X., Jin, X., Yang, L., Yuille, A., and Yang, J. (2020). Atomnas: Fine-grained end-to-end neural architecture search. In *International Conference on Learning Representations*.

[Melo, 2001] Melo, F. S. (2001). Convergence of q-learning: A simple proof. *Institute Of Systems and Robotics, Tech. Rep*, pages 1–4.

[Merity et al., 2017] Merity, S., Keskar, N. S., and Socher, R. (2017). Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*.

[Miller et al., 1989] Miller, G. F., Todd, P. M., and Hegde, S. U. (1989). Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384.

[Mirhoseini et al., 2020] Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Bae, S., et al. (2020). Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*.

[Miyato et al., 2018] Miyato, T., Kataoka, T., Koyama, M., and Yoshida, Y. (2018). Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[Munos, 2011] Munos, R. (2011). Optimistic optimization of a deterministic function without the knowledge of its smoothness. In *Advances in neural information processing systems*, pages 783–791.

[Munos, 2014] Munos, R. (2014). From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning. *technical report*, x(x):x.

[Mutny and Krause, 2018] Mutny, M. and Krause, A. (2018). Efficient high dimensional bayesian optimization with additivity and quadrature fourier features. In *Advances in Neural Information Processing Systems*, pages 9005–9016.

[Nayebi et al., esBO] Nayebi, A., Munteanu, A., and Poloczek, M. (2019, the implementation is from *https://github.com/aminnayebi/HesBO*). A framework for bayesian optimization in embedded subspaces. In *International Conference on Machine Learning*, pages 4752–4761.

[Nayman et al., 2019a] Nayman, N., Noy, A., Ridnik, T., Friedman, I., Jin, R., and Zelnik, L. (2019a). Xnas: Neural architecture search with expert advice. In *Advances in Neural Information Processing Systems*, pages 1977–1987.

[Nayman et al., 2019b] Nayman, N., Noy, A., Ridnik, T., Friedman, I., Jin, R., and Zelnik, L. (2019b). Xnas: Neural architecture search with expert advice. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 1977–1987. Curran Associates, Inc.

[Nemirovski et al., 2009] Nemirovski, A., Juditsky, A., Lan, G., and Shapiro, A. (2009). Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4):1574–1609.

[Oh et al., 2018] Oh, C., Gavves, E., and Welling, M. (2018). Bock: Bayesian optimization with cylindrical kernels. *arXiv preprint arXiv:1806.01619*.

[Pavlo et al., 2017] Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T. C., Perron, M., Quah, I., et al. (2017). Self-driving database management systems. In *CIDR*, volume 4, page 1.

[Pham et al., 2018a] Pham, H., Guan, M., Zoph, B., Le, Q., and Dean, J. (2018a). Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR.

[Pham et al., 2018b] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018b). Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning(ICML)*.

[Pincus, html] Pincus, M. (1970, the implementation is from *https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimiz*
Letter to the editor—a monte carlo method for the approximate solution of certain types of constrained optimization problems. *Operations Research*, 18(6):1225–1228.

[Pleiss et al., 2017] Pleiss, G., Chen, D., Huang, G., Li, T., van der Maaten, L., and Weinberger, K. Q. (2017). Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*.

[Radosavovic et al., 2020] Radosavovic, I., Kosaraju, R. P., Girshick, R., He, K., and Dollar, P. (2020). Designing network design spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Rajeswaran et al., 2017] Rajeswaran, A., Lowrey, K., Todorov, E. V., and Kakade, S. M. (2017). Towards generalization and simplicity in continuous control. In *Advances in Neural Information Processing Systems*, pages 6550–6561.

[Rajpurkar et al., 2016] Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

[Rasley et al., 2017] Rasley, J., He, Y., Yan, F., Ruwase, O., and Fonseca, R. (2017). Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 1–13.

[Real et al., 2019a] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019a). Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789.

[Real et al., 2019b] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019b). Regularized evolution for image classifier architecture search. In *Association for the Advancement of Artificial Intelligence(AAAI)*.

[Renggli et al., 2018] Renggli, C., Alistarh, D., and Hoefler, T. (2018). Sparcml: High-performance sparse communication for machine learning. *CoRR*, abs/1802.08021.

[Rhu et al., 2016] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W. (2016). vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE.

[Rolland et al., 2018] Rolland, P., Scarlett, J., Bogunovic, I., and Cevher, V. (2018). High-dimensional bayesian optimization via additive models with overlapping groups. *arXiv preprint arXiv:1802.07028*.

[Salimans et al., 2017] Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

[Sandler et al., 2018] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Sazanovich et al., 2020] Sazanovich, M., Nikolskaya, A., Belousov, Y., and Shpilman, A. (2020). Solving black-box optimization challenge via learning search space partition for local bayesian optimization. *arXiv preprint arXiv:2012.10335*.

[Schaffer et al., 1992] Schaffer, J. D., Whitley, D., and Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37. IEEE.

[Schrittwieser et al., 2019] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2019). Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*.

[Sciuto et al., 2019] Sciuto, C., Yu, K., Jaggi, M., Musat, C., and Salzmann, M. (2019). Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*.

[Seeger et al., 2003] Seeger, M., Williams, C., and Lawrence, N. (2003). Fast forward selection to speed up sparse gaussian process regression. Technical report, EPFL.

[Seide et al., 2014] Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. (2014). 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[Shahriari et al., 2015] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. (2015). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175.

[Shi et al., 2019a] Shi, H., Pi, R., Xu, H., Li, Z., Kwok, J. T., and Zhang, T. (2019a). Bridging the gap between sample-based and one-shot neural architecture search with bonas. *arXiv preprint arXiv:1911.09336*.

[Shi et al., 2019b] Shi, H., Pi, R., Xu, H., Li, Z., Kwok, J. T., and Zhang, T. (2019b). Multi-objective neural architecture search via predictive network performance optimization. *arXiv preprint arXiv:1911.09336*.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484.

[Silver et al., 2017] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[Snelson and Ghahramani, 2006] Snelson, E. and Ghahramani, Z. (2006). Sparse gaussian processes using pseudo-inputs. In *Advances in neural information processing systems*, pages 1257–1264.

[Sobol', 1967] Sobol', I. M. (1967). On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4):784–802.

[Springenberg et al., 2016] Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. (2016). Bayesian optimization with robust bayesian neural networks. In *Advances in neural information processing systems*, pages 4134–4142.

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

[Storn and Price, 1997] Storn, R. and Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359.

[Szegedy et al., 2017] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, pages 4278–4284.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[Tan et al., 2019] Tan, M., Chen, B., Pang, R., Vasudevan, V., and Le, Q. V. (2019). Mnasnet: Platform-aware neural architecture search for mobile. In *Conference on Computer Vision and Pattern Recognition(CVPR)*.

[Tan and Le, 2019a] Tan, M. and Le, Q. (2019a). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR.

[Tan and Le, 2019b] Tan, M. and Le, Q. V. (2019b). Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946.

[Tch, 2017] Tch, A. (2017). *Title*. `https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464`.

[Tian et al., 2019a] Tian, Y., Ma, J., Gong, Q., Sengupta, S., Chen, Z., Pinkerton, J., and Zitnick, C. L. (2019a). Elf opengo: An analysis and open reimplementation of alphazero. *arXiv preprint arXiv:1902.04522*.

[Tian et al., 2019b] Tian, Z., Shen, C., Chen, H., and He, T. (2019b). FCOS: Fully convolutional one-stage object detection. In *Proc. Int. Conf. Computer Vision (ICCV)*.

[Todorov et al., 2012] Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE.

[Vanhoucke et al., 2011] Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, page 4.

[Villemonteix et al., 2009] Villemonteix, J., Vazquez, E., and Walter, E. (2009). An informational approach to the global optimization of expensive-to-evaluate functions. *Journal of Global Optimization*, 44(4):509.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

[Wan et al., 2020a] Wan, A., Dai, X., Zhang, P., He, Z., Tian, Y., Xie, S., Wu, B., Yu, M., Xu, T., Chen, K., et al. (2020a). Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12965–12974.

[Wan et al., 2020b] Wan, A., Dai, X., Zhang, P., He, Z., Tian, Y., Xie, S., Wu, B., Yu, M., Xu, T., Chen, K., Vajda, P., and Gonzalez, J. E. (2020b). Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Wang et al., 2020a] Wang, L., Fonseca, R., and Tian, Y. (2020a). Learning search space partition for black-box optimization using monte carlo tree search. *Advances in Neural Information Processing Systems*.

[Wang et al., 2016a] Wang, L., Wu, W., Bosilca, G., Vuduc, R., and Xu, Z. (2016a). Efficient communications in training large scale neural networks. *arXiv preprint arXiv:1611.04255*.

[Wang et al., 2016b] Wang, L., Wu, W., Xu, Z., Xiao, J., and Yang, Y. (2016b). Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. ACM.

[Wang et al., 2019a] Wang, L., Xie, S., Li, T., Fonseca, R., and Tian, Y. (2019a). Sample-efficient neural architecture search by learning action space. *arXiv preprint arXiv:1906.06832*.

[Wang et al., 2019b] Wang, L., Xie, S., Li, T., Fonseca, R., and Tian, Y. (2019b). Sample-efficient neural architecture search by learning action space. *CoRR*, abs/1906.06832.

[Wang et al., 2021] Wang, L., Xie, S., Li, T., Fonseca, R., and Tian, Y. (2021). Sample-efficient neural architecture search by learning actions for monte carlo tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[Wang et al., 2017a] Wang, L., Yang, Y., Min, R., and Chakradhar, S. (2017a). Accelerating deep neural network training with inconsistent stochastic gradient descent. *Neural Networks*.

[Wang et al., 2018a] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. (2018a). Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53.

[Wang et al., 2018b] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. (2018b). Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[Wang et al., 2019c] Wang, L., Zhao, Y., Jinnai, Y., Tian, Y., and Fonseca, R. (2019c). Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1903.11059*.

[Wang et al., 2020b] Wang, N., Gao, Y., Chen, H., Wang, P., Tian, Z., Shen, C., and Zhang, Y. (2020b). Nas-fcos: Fast neural architecture search for object detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Wang et al., 2019d] Wang, W., Sun, Y., and Halgamuge, S. (2019d). Improving MMD-GAN training with repulsive loss function. In *International Conference on Learning Representations*.

[Wang et al., 2020c] Wang, X., Xue, C., Yan, J., Yang, X., Hu, Y., and Sun, K. (2020c). Mergenas: Merge operations into one for differentiable architecture search. In Bessiere, C., editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 3065–3072. ijcai.org.

[Wang et al., 2017b] Wang, Z., Gehring, C., Kohli, P., and Jegelka, S. (2017b). Batched large-scale bayesian optimization in high-dimensional spaces. *arXiv preprint arXiv:1706.01445*.

[Wang et al., 2016c] Wang, Z., Hutter, F., Zoghi, M., Matheson, D., and de Feitas, N. (2016c). Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387.

[Wang et al., 2014] Wang, Z., Shakibi, B., Jin, L., and Freitas, N. (2014). Bayesian multi-scale optimistic optimization. In *Artificial Intelligence and Statistics*, pages 1005–1014.

[Wang et al., 2013] Wang, Z., Zoghi, M., Hutter, F., Matheson, D., and De Freitas, N. (2013). Bayesian optimization in high dimensions via random embeddings. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

[Wangni et al., 2018] Wangni, J., Wang, J., Liu, J., and Zhang, T. (2018). Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, pages 1306–1316.

[Watkins and Dayan, 1992] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

[Watts and Strogatz, 1998] Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442.

[Weinberger, 2017] Weinberger, K. (2017). *Title*. `https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote15.html`.

[Weinstein and Littman, 2012] Weinstein, A. and Littman, M. L. (2012). Bandit-based planning and learning in continuous-action markov decision processes. In *Twenty-Second International Conference on Automated Planning and Scheduling*.

[Wen et al., 2020] Wen, W., Liu, H., Chen, Y., Li, H., Bender, G., and Kindermans, P.-J. (2020). Neural predictor for neural architecture search. In *European Conference on Computer Vision*, pages 660–676. Springer.

[Wen et al., 2017] Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. (2017). Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*.

[Weng, 2018] Weng, L. (2018). Policy gradient algorithms. *lilianweng.github.io/lil-log*.

[Wierstra et al., 2014] Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J., and Schmidhuber, J. (2014). Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

[Wilson et al., 2018] Wilson, J. T., Hutter, F., and Deisenroth, M. P. (2018). Maximizing acquisition functions for bayesian optimization. *arXiv preprint arXiv:1805.10196*.

[Wu et al., 2019] Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. (2019). Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Wu et al., 2015] Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., and Dongarra, J. (2015). Hierarchical dag scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 156–165. IEEE.

[Xie, 2018] Xie, S. (2018). *Deep Representation Learning with Induced Structural Priors*. PhD thesis, UC San Diego.

[Xie et al., 2019] Xie, S., Kirillov, A., Girshick, R., and He, K. (2019). Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1284–1293.

[Xu et al., 2019] Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G.-J., Tian, Q., and Xiong, H. (2019). Pc-darts: Partial channel connections for memory-efficient architecture search. *arXiv preprint arXiv:1907.05737*.

[Xu et al., 2020] Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G.-J., Tian, Q., and Xiong, H. (2020). {PC}-{darts}: Partial channel connections for memory-efficient architecture search. In *International Conference on Learning Representations*.

[Yang et al., 2017] Yang, Z., Dai, Z., Salakhutdinov, R., and Cohen, W. W. (2017). Breaking the softmax bottleneck: A high-rank rnn language model. *arXiv preprint arXiv:1711.03953*.

[Yiming Hu, 2020] Yiming Hu, Yuding Liang, Z. G. R. W. X. Z. Y. W. . Q. G. J. S. (2020). Angle-based search space shrinking for neural architecture search.

[Ying et al., 2019] Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., and Hutter, F. (2019). Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR.

[You et al., 2020] You, J., Leskovec, J., He, K., and Xie, S. (2020). Graph structure of neural networks. In *International Conference on Machine Learning*, pages 10881–10891. PMLR.

[You, 2020] You, Y. (2020). Fast and accurate machine learning on distributed systems and supercomputers.

[Yu and Huang, 2019] Yu, J. and Huang, T. S. (2019). Network slimming by slimmable networks: Towards one-shot architecture search for channel numbers. *CoRR*, abs/1903.11728.

[Yu et al., 2020a] Yu, J., Jin, P., Liu, H., Bender, G., Kindermans, P.-J., Tan, M., Huang, T., Song, X., Pang, R., and Le, Q. (2020a). Bignas: Scaling up neural architecture search with big single-stage models. In *European Conference on Computer Vision*, pages 702–717. Springer.

[Yu et al., 2019a] Yu, K., Ranftl, R., and Salzmann, M. (2019a). How to train your super-net: An analysis of training heuristics in weight-sharing nas.

[Yu et al., 2019b] Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. (2019b). Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*.

[Yu et al., 2020b] Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. (2020b). Evaluating the search phase of neural architecture search. In *International Conference on Learning Representations*.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.

[Zela et al., 2019] Zela, A., Elsken, T., Saikia, T., Marrakchi, Y., Brox, T., and Hutter, F. (2019). Understanding and robustifying differentiable architecture search. *arXiv preprint arXiv:1909.09656*.

[Zela et al., 2020] Zela, A., Siems, J., and Hutter, F. (2020). Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search. In *International Conference on Learning Representations*.

[Zhang et al., 2020] Zhang, M., Li, H., Pan, S., Liu, T., and Su, S. (2020). One-shot neural architecture search via novelty driven sampling. In Bessiere, C., editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 3188–3194. International Joint Conferences on Artificial Intelligence Organization. Main track.

[Zhao et al., 2020] Zhao, Y., Wang, L., Tian, Y., Fonseca, R., and Guo, T. (2020). Few-shot neural architecture search. *arXiv preprint arXiv:2006.06863*.

[Zhao et al., 2017] Zhao, Y., Wang, L., Wu, W., Bosilca, G., Vuduc, R., Ye, J., Tang, W., and Xu, Z. (2017). Efficient communications in training large scale neural networks. In *Proceedings of the on Thematic Workshops of ACM Multimedia 2017*.

[Zhiyuan Li, 2020] Zhiyuan Li, S. A. (2020). An exponential learning rate schedule for deep learning.

[Zhou et al., 2019a] Zhou, H., Yang, M., Wang, J., and Pan, W. (2019a). BayesNAS: A Bayesian approach for neural architecture search. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7603–7613. PMLR.

[Zhou et al., 2019b] Zhou, H., Yang, M., Wang, J., and Pan, W. (2019b). Bayesnas: A bayesian approach for neural architecture search. *arXiv preprint arXiv:1905.04919*.

[Zilly et al., 2017] Zilly, J. G., Srivastava, R. K., Koutnık, J., and Schmidhuber, J. (2017). Recurrent highway networks. In *International Conference on Machine Learning*, pages 4189–4198. PMLR.

[Zoph and Le, 2016] Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

[Zoph et al., 2018a] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. (2018a). Learning transferable architectures for scalable image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

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

[Zoph et al., 2018b] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018b). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710.