University of California
Santa Barbara

Methods for Efficient Deep Reinforcement Learning

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Samuel Brooks Green

Committee in charge:

Professor Çetin Kaya Koç, Co-Chair
Professor Ömer Eğecioğlu, Co-Chair
Professor Timothy Sherwood
Dr. Craig M. Vineyard, Sandia National Laboratories

December 2019

The Dissertation of Samuel Brooks Green is approved.

Professor Timothy Sherwood

Dr. Craig M. Vineyard, Sandia National Laboratories

Professor Ömer Eğecioğlu, Committee Co-Chair

Professor Çetin Kaya Koç, Committee Co-Chair

December 2019

Methods for Efficient Deep Reinforcement Learning

by

Samuel Brooks Green

To my parents.

Acknowledgements

My interest in reinforcement learning originated from a seminar Professor Ömer Eğecioğlu facilitated on the topic. In addition to that, I would like to express my appreciation for Professor Eğecioğlu's constant encouragement and support, from the time I applied to UCSB, and throughout my Ph.D. effort.

I'm also fortunate to have Professor Timothy Sherwood on my committee. I benefited from many conversations with Professor Sherwood regarding selection of high-quality research directions and how to best communicate the results of those efforts. Professor Sherwood also demonstrated the value of adding kindness to professional rigor.

Dr. Craig Vineyard, at Sandia National Laboratories, has been an invaluable addition to my committee. Dr. Vineyard mentored me and proposed and supported numerous experimental directions which have become the core components of my current and future research.

Finally, I would like to thank Professor Çetin Kay Koç for inviting me to pursue a Ph.D. at UCSB. Through Professor Koç, I was guided to the exciting topics of neuromorphic engineering and efficient autonomy, which eventually resulted in this dissertation. I have benefited from countless new concepts, adventures, and lessons from the time Professor Koç has spent on me.

Curriculum Vitæ
Samuel Brooks Green

Education

2019	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2009	M.S. in Applied Mathematics, University of Central Arkansas.
2006	B.S. in Computer Science, University of Central Arkansas.

Publications

S. Green, C. M. Vineyard, and Ç. K. Koç, "RAPDARTS: Resource-aware progressive differentiable architecture search," in *International Joint Conference on Neural Networks*. IEEE, Glasgow, UK, July 19–24, 2020 (In Review).

S. Green*, C. Vineyard*, W. M. Severa, and Ç. K. Koç, "Benchmarking event-driven neuromorphic architectures," in *International Conference on Neuromorphic Systems*. ACM, Knoxville, Tennessee, July 23–25, 2019.

S. Green*, J. Luo*, P. Feghali, G. Legrady, and Ç. K. Koç, "Visual diagnostics for deep reinforcement learning policy development," *NVIDIA GPU Technology Conference*, San Jose, California, March 18–21, 2019.

S. Green, J. B. Aimone, "Memristors learn to play," *Nature electronics* 2(3), 96, March, 2019.

S. Green, C. M. Vineyard, and Ç. K. Koç, "Distillation strategies for proximal policy optimization," *IEEE Transactions on Neural Networks and Learning Systems*, submitted June 5, 2019 (In Review).

S. Green* and J. Luo*, "Bridging reinforcement learning and creativity: implementing reinforcement learning in processing," *SIGGRAPH Asia Courses*, December 2018.

S. Green*, J. Luo*, P. Feghali, G. Legrady, and Ç. K. Koç, "Reinforcement learning and trustworthy autonomy," *Cyber-Physical Systems Security*, Ç. K. Koç, editor, pp. 191–217, Springer, 2018.

S. Green, C. M. Vineyard, and Ç. K. Koç, "Mathematical optimizations for deep learning," *Cyber-Physical Systems Security*, Ç. K. Koç, editor, pp. 69–92, Springer, 2018.

S. Green, C. M. Vineyard, and Ç. K. Koç, "Impacts of mathematical optimizations on reinforcement learning policy performance," in *International Joint Conference on Neural Networks*. IEEE, Rio de Janeiro, Brazil, July 8–13, 2018.

S. Green, "Inductive image editing based on learned stylistic preferences," US Patent Nr. 9,558,428 B1. 2017.

*Denotes equal contribution.

S. Green, İ. Çiçek, and Ç. K. Koç, "Continuous-time computational aspects of cyber-physical security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 59–62, Santa Barbara, CA, August 2016.

S. Green, "FPGA coprocessing for computational mathematics," M.S. Thesis, University of Central Arkansas, 2009.

Abstract

Methods for Efficient Deep Reinforcement Learning

by

Samuel Brooks Green

Reinforcement learning (RL) is a broad family of algorithms for training autonomous agents to collect rewards in sequential decision-making tasks. Shortly after deep neural networks (DNNs) advanced, they were incorporated into RL algorithms as high-dimensional function approximators. Recently, "deep" RL algorithms have been used for many applications that were once only approachable by humans, e.g., expert-level performance at the game of Go and dexterous control of a high degree-of-freedom robotic hand. However, standard deep RL approaches are computationally, and often financially, expensive. High cost limits RL's real-world application, and it will slow research progress.

In this dissertation, we introduce methods for developing efficient DNN-based RL agents. Our approaches for increasing efficiency draw upon recent developments for the optimization of DNN inference. Specifically, we present quantization, parameter pruning, parameter sharing, and model distillation algorithms that reduce the computational cost of DNN-based policy execution. We also introduce a new algorithm for the automatic design of DNNs which attain high performance while meeting specific resource constraints like latency and power. Intuition, which is backed by empirical results, states that a naive reduction in DNN model capacity should lead to a reduction in model performance. However, our results prove that by taking a principled approach, it is often possible to maintain high agent performance while simultaneously lowering the computational expense of decision-making.

Finally, a policy must be evaluated on hardware, and currently, there is an explosion

of non von Neumann architectures for the acceleration of neural algorithms. We analyze one such device, and we propose rigorous methods for analysis of such devices for future applications.

Contents

Chapter 1

Introduction

How do you become an expert at something? Most likely it is through a combination of intense practice, instruction from a teacher, innate talent, and studying the work of other experts. Similarly, *reinforcement learning* (RL) is a branch of machine learning that attempts to create expert agents using various algorithmic approximations of the previous strategies. RL is distinguished from other machine learning methods by its use of a sequential decision-making agent that attempts to maximize the long-term collection of rewards within an environment. RL agents learn which actions, in which contexts, lead to the most rewards.

RL agents use a *policy* for action selection. In mammals, vision processing accounts for a significant portion of neural activity. Likewise *deep* RL (DRL) policies often process visual or spatial inputs with the bulk of the policy's computational expense attributable to the execution of convolutional neural networks (CNNs) or other types of deep neural networks (DNNs).

As an example of how computationally expensive DRL can be, consider DeepMind's AlphaGo Zero (AGZ) reinforcement learning solution to the game of Go, which beat the world's top Go player in 2016 [1]. AGZ used a large CNN for its policy. Compu-

tational cost was not clearly provided by DeepMind in the AGZ publication. However, Facebook Artificial Intelligence Research (FAIR) duplicated the AGZ work and provided a cost: 2,000 NVIDIA V100 GPUs running for 9 days [2]. Executing FAIR's Go re-implementation code on Google Cloud would currently cost [3]:

$$2,000 \text{ GPUs} \times \frac{\$2.48}{\text{GPU Hr}} \times 216 \text{ Hr} \approx \$1,071,360. \tag{1.1}$$

The bulk of AGZ's computational cost, and therefore financial cost, is its CNN-based policy which uses one input convolutional block followed by 19 residual blocks. The convolutional block has the following structure:

- 3×3 convolution with 256 channels and stride 1
- Batch normalization
- Rectified linear unit (ReLU)

The convolutional block output is input to a residual tower where each of the 19 residual blocks have the following structure:

- 3×3 convolution with 256 channels and stride 1
- Batch normalization
- ReLU
- 3×3 convolution with 256 channels and stride 1
- Batch normalization
- Skip-connection that adds the input
- ReLU

Go is partially-observable (because of a no-repeat rule in the game and because the current player is not indicated from the current board state) so the AGZ designers provide

the CNN with a history of recent board positions as well as an indicator for the current player. This is achieved by using 17 binary 19×19 (the game board size) channels as input to the policy. The first eight channels capture empty/occupy intersections of the current player's stones for the last eight game states. The second eight channels capture the opponent's stone intersections for the last eight game states. The 17th channel is set to all 1s, if it is black's move, or all 0s, if it is white's move.

Given AGZ's network architecture and input tensor shape, approximately eight billion multiply-accumulate operations (MACs) per forward-pass are required. The AGZ network is used for Monte-Carlo Tree Search (MCTS) during game play, using 1,600 game simulations to select each move. During each game simulation, the CNN is evaluated once. Therefore, over 12 trillion MACs are required for the AGZ agent to pick a single move. This is why the AGZ network architecture is the primary contributor of the computational and financial cost for training and then deploying AGZ.

The computational intensity of DRL will only increase as tasks become more challenging. Even now we are seeing DRL agents which receive as sensory input a mix of vision, audio, and text input [4, 5]. In addition to being required to process multimodal sensory inputs, agents must often perform in situations which require memory, because their instantaneous sensory inputs do not contain complete information about the state of their environment. Extending agents with complex memory capabilities further complicates design and increases processing cost.

The high computational expense of deep reinforcement learning will limit its application when constrained by power, memory, or latency. However, by extending deep neural network efficiency techniques to deep reinforcement learning, it is possible to both maintain agent performance and reduce the computational expense of the agent's decision-making.

DNNs are a type of directed acyclic graph, where each vertex in the graph is a

primitive operation, e.g. one of the following operations: convolution, pooling, attention, matrix-vector multiplication, recurrent operation, or nonlinearity operation. We refer to a DNN's specific set of primitive operations and the connectivity between operations as the DNN's *architecture*.

Similarly, we classify existing DNN optimizations into two categories: *primitive optimizations* and *architectural optimizations*. Primitive optimizations are low-level, and they are concerned with obtaining the most benefit from the least amount of resources. This would be analogous to the development of a construction brick which can support the most pressure and has the best insulation properties using the least amount of material. There are five primary primitive optimization strategies:

- Pruning: reduces the number of parameters, which, in turn, reduces the total number of MAC operations, amount of traffic required to transfer parameters, and storage requirements.
- Quantization: lowers the number of bits of precision representing neural network inputs, parameters, or activations, which lowers both memory requirements and silicon required for processing elements.
- Weight Sharing and Compression: forces parameters to share values, thus decreasing memory storage and traffic.
- Model Distillation: the training of a smaller network to mimic the behavior of larger network, reducing the number of parameters and lowering latency.
- Filter Decomposition: modifies convolutional filter designs such that the number of parameters and multiple-accumulate operations are reduced.

Architectural optimizations are concerned with the construction of DNNs, given specific primitive operations with which to build from. Using another construction analogy,

the Passivhaus architectural standard achieves energy optimization through careful selection of which construction materials to use, HVAC flow, and the geographic orientation of the building [6].

DNN architecture design has historically been a manual process, where various neural network architecture and primitive combinations were iteratively hand-selected and tested on a dataset until something performed satisfactorily. *Neural Architecture Search* (NAS) has recently started to take over hand-crafted architectural optimization.

NAS methods automate strategies for discovery of high performing neural architectures. A DRL approach was the first post-AlexNet NAS method with state-of-the-art performance on CIFAR-10 [7, 8]. The DRL approach was quickly followed by a high performance Evolutionary Strategy (ES) based method [9]. While both the DRL and ES methods discovered high performance architectures, their use came at the cost of thousands of GPU hours.

Gradient-based NAS (GBNAS) methods have the benefit of being directly optimized through gradient descent and consequently complete the search faster than other NAS methods [10]. The search process alternates between temporarily fixing one set of parameters, i.e. assuming they are constant, and updating the other set of parameters. The GBNAS approaches provide no convergence guarantees, but they work well in practice.

Both primitive and architectural optimizations have been pursued extensively in supervised learning literature, but, to our knowledge, this dissertation represents the first comprehensive investigation into their application for efficient DRL.

There are close similarities between supervised learning and DRL, but analyzing the effect of optimizations on DRL should be considered explicitly. In both cases, image, audio, text, or other modalities, are input into a DNN for feature extraction. The resulting features are used for classification, in the context of supervised learning, action selection, in the context of DRL, or regression, which is used by both supervised learning

and DRL. The primary difference between supervised learning and DRL is the *domain* of the inputs to the DNN.

In machine learning, the domain is the distribution over which the inputs to a function are valid. In supervised learning, the training set is ideally drawn from the same distribution that the test set (or real-world examples) will be drawn from later. Extrapolation occurs when inputs are drawn from a distribution that is different from which training has occurred.

DRL is more complex. In DRL, an agent may spend some time in one setting, e.g. a room. Observations from that setting will be used for DNN training. Eventually the agent may get to another setting, e.g. go outside. If the observational characteristics of the new setting are from a new distribution, then the DNN-based policy experiences *domain shift*, i.e. extrapolation occurs. Domain shift can cause collapse in agent performance.

The literature for DNN optimizations in supervised learning perform their evaluations on common benchmarks, e.g. CIFAR-10, CIFAR-100, ImageNet, in the case of CNNs. In that scenario, there is a known and fixed test set used by the community to compare results. The test set is essentially a previously agreed upon "holdout" subset of the training set. So research on primitive and architectural optimizations for supervised learning when using benchmarks for evaluation benefits from stable distributions.

On the other hand, domain shift is guaranteed to occur in non-trivial DRL environments. A priori, it is unclear how primitive and architectural optimizations will react under the presence of domain shift in the DRL setting. For this reason it is worthwhile to specifically consider the impacts of DNN optimizations on the performance of DRL agents.

The optimizations discussed thus far are essentially concerned with the co-design of neural architectures and their constituent primitive operations to achieve efficient performance on observations from a domain of interest. Ultimately, however, a neural

architecture must be executed on a specific hardware platform. NVIDIA GPUs, and, increasingly, Google's TPUs, are dominant for training DNNs, because they can efficiently perform high-precision tensor arithmetic over large training batches. However, for decision-making (i.e. inference), a CPU, GPU, FPGA, or custom accelerator may be preferable.

Taking hardware attributes and constraints into account is the next logical step when considering methods for efficient DRL. For example, there is no reason to prune individual convolutional filter parameters if hardware cannot take advantage of sparse convolution. And, for example, there is no reason to quantize numbers to single-bit precision, if the target processor's ALU only supports a minimum of 8-bit arithmetic.

While this dissertation is primarily focused on DNN-based RL, we again point out that RL is a family of algorithms that learn behavior to maximize long-term accumulation of rewards. There is no requirement that RL algorithms use deep neural networks, indeed deep learning did not exist when the fundamental algorithms of RL were created [11, 12, 13]. However, before DNNs, the RL methods which used classical function approximation were not able to process high-dimensional observations well.

DNNs are a subset of *neuromorphic engineering* which is concerned with biologically plausible models of intelligence. To put things in perspective, DNNs are an advancement over the McCulloch-Pitts neuron model which is essentially a dot-product and nonlinearity function [14]. The McCulloch-Pitts model was created in 1943. Since then, the field of neuroscience has advanced, and the field of computational neuroscience has appeared. These fields have created new mathematical models of neural processing. Many of the more biologically plausible neuromorphic computing models are based on spiking neuron models, which incorporate time and amplitude, unlike DNNs which only use amplitude [15].

Spiking neuron models are efficiently processed by *event-driven* architectures [16, 17,

18]. These are architectures that only perform computation when certain events occur, unlike common tensor processors which cannot benefit from sparse computations.

Thus far, the neuromorphic community has not found a training algorithm that trains spiking neuron models to perform as well as backpropagation trains DNNs. Partly this has to do with the fact that the deep learning community's popular datasets are conducive to DNN processing, while spiking algorithms would benefit from event-driven datasets that are expensive to collect or generate. If and when the neuromorphic community is able to train event-driven models as well as backpropagation trains DNNs, then there will be opportunity to perform deep reinforcement learning at very low-power.

1.1 Dissertation Outline

- Chapter 2 introduces the basic mathematical concepts of Reinforcement Learning. Using Microsoft's AirSim, we also provide a case-study of reinforcement learning's application to drone flight control.

- Chapter 3 provides an overview of the central methods for primitive operation optimization: pruning, quantization, weight sharing and compression, model distillation, and filter decomposition.

- Chapter 4 introduces a resource-aware neural architecture search technique. This method enables the discovery of DNNs which meet predefined resource constraints. As a use-case, we show how to discover CNN architectures requiring less than a specified number of parameters.

- Chapter 5 investigates the application of primitive operation optimizations to reinforcement learning. In this chapter, we apply pruning, quantization, and compression to reinforcement learning policies and study how robust they are to the

optimizations.

- Chapter 6 develops a distillation algorithm for reinforcement learning which enables agents with relatively small CNN-based policies to achieve performance on par with agents using larger CNN-based policies.

- Chapter 7 considers neuromorphic hardware. In the first section, we analyze a recent memristive reinforcement learning accelerator. In the second section, we present an objective approach to the evaluation of neuromorphic hardware.

1.2 Permissions and Attributions

- The content of Chapter 2 is the result of a collaboration with Jieliang Luo, Peter Feghali, George Legrady, and Çetin Kaya Koç. Part of the content previously appeared in *Cyber-Physical Systems Security*, Ç. K. Koç, editor, pages 191–217, Springer, 2018. It is reproduced here with permission from Springer.

- The content of Chapter 3 is the result of collaborations with Craig M. Vineyard, and Çetin Kaya Koç. The content previously appeared in *Cyber-Physical Systems Security*, Ç. K. Koç, editor, pages 69–92, Springer, 2018. It is reproduced here with permission from Springer.

- The content of Chapter 4 is the result of collaborations with Craig M. Vineyard, and Çetin Kaya Koç. It is currently in pre-publication.

- The content of Chapter 5 is the result of collaborations with Craig M. Vineyard, and Çetin Kaya Koç. The content previously appeared at the *International Joint Conference on Neural Networks*, Rio de Janeiro, Brazil, July 8–13, 2018. It is reproduced here with permission from IEEE.

- The content of Chapter 6 is the result of collaborations with Craig M. Vineyard, and Çetin Kaya Koç. The content is currently in review with *IEEE Transactions on Neural Networks and Learning Systems*. It is reproduced here with permission from IEEE.

- The content of Chapter 7 is the result of collaborations with Craig M. Vineyard, James B. Aimone, William Severa, and Çetin Kaya Koç. Part of the content previously appeared at *International Conference on Neuromorphic Systems (ICONS)*, Knoxville, Tennessee, July 23–25, 2019. It is reproduced here with permission from the ACM. Part of the content previously appeared in *Nature Electronics* 2(3), 96, March, 2019. It is reproduced here with permission from Springer Nature.

Chapter 2

Deep Reinforcement Learning

2.1 Introduction

Reinforcement learning (RL) is a family of control techniques that arose from a combination of applying psychological models of operant conditioning with mathematical techniques of dynamic programming [19]. In RL, there is an *agent* that uses a *policy* to decide *actions*, given *state observations* and *rewards* from an *environment*.

The agent may be something physical, like a robot, or it could be more virtual, like a program that chooses the advertisement to show a website visitor. Likewise, the environment may be physical or virtual. In general, actions can be discrete or continuous,

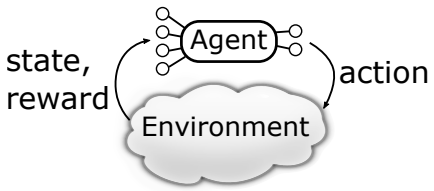


Figure 2.1: The deep reinforcement learning environment-agent cycle. An environment provides state observations and rewards to an agent. State observations are input to a neural network-based policy which decides actions. An agent then makes an action with the goal of maneuvering into states with the highest rewards.

depending on the agent's capability. Rewards can be continuous or discrete and *dense* or *sparse*. An example of dense reward would be distance from the agent to a target at any given moment. An example of spare reward would be 0 at all time steps until some goal is reached and then 1 for only that time step. Finally, state observations represent the information an agent is able to detect from the environment. For example, a self-driving car may receive observations from only a CCD camera on it's front bumper, or it could be surrounded by LIDAR, RADAR, CCD, and other sensors.

The agent's *policy* π maps state observations to actions. The goal of all RL algorithms is to find an optimal policy π^* that maximizes the *return*, which is defined as the expected sum of rewards over some sequence of state-action pairs. The environment provides state observations and rewards. After receiving a new observation, the policy chooses which action the agent should take. The policy must learn to make actions to maneuver the agent into states with high rewards. An illustration of the agent-environment interaction paradigm is shown in Fig. 2.1.

The optimal policy is then formally defined as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau \sim p}\left[\sum_{t} r(s_t, a_t)\right] \tag{2.1}$$

where the *rollout* τ represents a sequence of states and actions, p is the trajectory distribution, r is the *reward function* mapping states s and actions a to rewards, and subscript t ranges across time steps in the rollout.

The basic methods of RL have been developed over the past several decades, but early RL algorithms did not work very well for high dimensional observations, e.g. images. Specifically, classical RL techniques using images for observations would require a table with each entry in the table corresponding to exactly one configuration of the image's RGB values. For example, a 256×256 8-bit RGB image has $256 \times 256 \times (2^8)^3 \approx 1 \times 10^{12}$

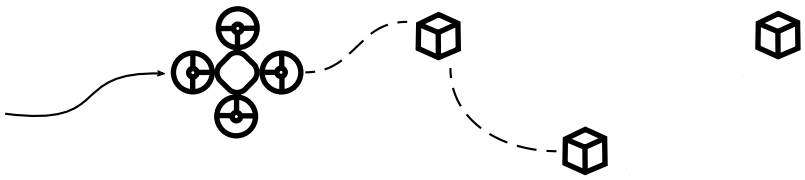


Figure 2.2: In this chapter we train a drone to use its camera to perform path planning. Training is performed via reinforcement learning, and the goal is to learn camera-to-action mappings that allow the drone to collect cubes.

possible unique values and would require a table with that many entries.

Function approximation can be used to map similar observations to similar actions. However, function approximation-based RL did not work well until 2013, when DeepMind developed new algorithms for using deep neural networks (DNNs) with RL. By doing so, in their seminal Atari-dominating RL work, agents learned to play many Atari video games at superhuman levels of performance [20]. Since then, there has been a regular stream of deep RL (DRL) algorithm improvements [21, 22, 23, 24]. The application space of DRL has naturally lagged behind theoretical results, and we expect to see further real-world results in the future. Following results, there will be increasing interest in hardware optimized for deep reinforcement learning.

In the remainder of this chapter, we introduce *Markov Decision Processes*, which are mathematical abstractions of many real-world decision making tasks. Then we introduce the *Vanilla Policy Gradient* (VPG) method, which is a foundational reinforcement learning algorithm. Finally, we apply VPG in the context of a physics-based simulator that enables experimentation with self-driving and flight applications (Fig. 2.2).

2.2 Markov Decision Processes

Markov Decision Processes (MDPs) are the mathematical models that reinforcement learning was developed to solve. MDPs have states in which an agent exists, and the out-

comes of actions depend only on the current state, not on past states and actions; in this sense MDPs are *memoryless*. The memoryless property is captured in the environment's state-transition and reward function notation:

$$\begin{aligned} p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) &= p(s_{t+1}|s_t, a_t), \\ r(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) &= r(s_{t+1}|s_t, a_t). \end{aligned} \tag{2.2}$$

The state-transition and reward functions in Eq. 2.2 specify that function outcomes depend only on the current state and action, and are independent of past states and actions.

A second defining feature of MDPs is that the state-transition and/or reward functions could be *stochastic*, which means their return values are drawn from some underlying probability distributions. In standard RL settings, these distributions must be *stationary* which means the probabilities do not shift over time. Methods exist for using RL in nonstationary environments. Investigating such advanced methods is critical for using RL in safety-critical applications. For example, state-transition and reward distributions may shift from what was observed during training in the event of an anomalous situation, e.g. an emergency. In that case it could be disastrous were the agent to follow its policy decisions blindly. For that reason, consider the methods introduced in this chapter as an introduction to what is possible, but safety mechanisms should be put in place for real-world RL applications.

Within an MDP, agents may observe their current state and make actions that attempt to affect the future state. The agent's objective is to maximize collection of rewards. An example 3-state MDP[1] is given in Fig. 2.3. In this example, the initial state is s_0, and the agent has two action options: a_0 and a_1. If the agent chooses action a_0 it is

[1]In the notation for this example, the subscripts for actions a denote "options", versus the usual meaning, which is time in this chapter.

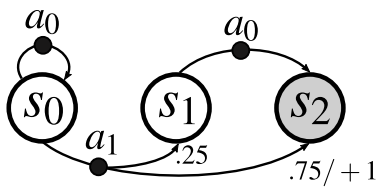


Figure 2.3: Example Markov Decision Process. There are three states and two actions, a_0 and a_1. Unless otherwise indicated, the state transition probability is 1 and reward is 0. Transition from s_0 to s_2 is the most interesting with $r(s_2|s_0, a_1) = 1$ and $p(s_2|s_0, a_1) = .75$.

guaranteed to stay in state s_0, denoted by $p(s_0|s_0, a_0) = 1$. If the agent chooses action a_1, there is a 25% probability that it will transition to s_1, denoted by $p(s_1|s_0, a_1) = .25$, and a 75% probability it will transition to s_2, denoted by $p(s_2|s_0, a_1) = .75$. The environment returns reward of 0 for all state transitions except for $s_0 \rightarrow s_2$, and in this case it returns $r(s_2|s_0, a_1) = 1$.

In the context of Fig. 2.3 the agent should always select action a_1, as it is the only action that leads to a non-zero reward. While *we* can see that is the solution, an RL agent must learn it.

2.3 Reinforcement Learning Approaches

As illustrated in Fig. 2.4, the approaches for finding the optimal policy π^* in Eq. 2.1 can be separated into three families of methods: value-based methods, policy-based methods, and model-based methods. Value-based methods, e.g. Q-Learning, are closer to RL's historical roots in dynamic programming. They use the learned value of states and actions to find a policy that will transfer the agent into states with more value [12, 25, 26]. Policy-based methods directly learn to optimize an action-making policy via maximizing a reward function [24]. Model-based approaches require the agent have a representation of the environment which provides a prediction of the reward the environment will yield when a given action is taken [27]. This model of the environment enables learning the

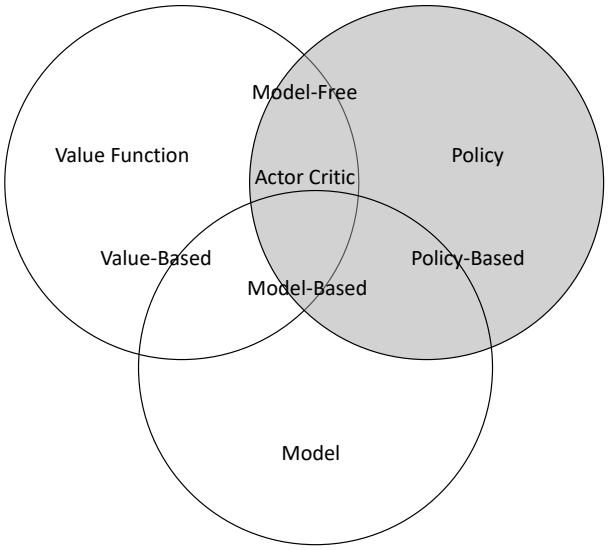


Figure 2.4: Reinforcement learning taxonomy of approaches. Value-based methods learn the long-term results of states or state-action pairs. Policy-based methods learn to directly optimize a policy. Model-based methods leverage given or learned mechanics of the environment. This introductory chapter is focused on a policy-based method.

optimal policy by providing a prediction of how the environment will behave so the best actions to take may be identified. It is also possible to mix value-based, policy-based, and model-based methods into hybrid RL algorithms [28, 29].

Because they serve as the basis for most deep RL approaches, we focus on policy-based methods in this chapter, highlighted in gray in the figure, and described in more detail next.

2.4 Vanilla Policy Gradient

In the context of reinforcement learning, our first-order objective was defined in Eq. 2.1 as the sum of rewards, but here we will refine it. As stated in Section 2.2, MDPs often have stochastic state transition and reward functions; for that reason the *objective* $J(\theta)$ of the agent is actually to maximize the expected sum of rewards under the *trajectory distribution* (defined in Eq. 2.8 below). This is achieved by discovering

optimal policy (i.e. deep neural network) parameters $\theta^\star$ that maximize the objective function $J(\theta)$:

$$\theta^\star = \arg\max_\theta \mathbb{E}_{\tau\sim p_\theta} \sum_{t=0}^{T-1} r(s_t, a_t) = \arg\max_\theta J(\theta), \tag{2.3}$$

where τ is the *trajectory* of state-action pairs $(s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ and p_θ is the trajectory distribution which is conditioned on the policy parameters.

The Vanilla Policy Gradient method uses gradient ascent to adjust the policy parameters in a direction that increases $J(\theta)$ [13]. For notation convenience let $r(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t)$, and by the definition of expectation, the objective can be written as:

$$J(\theta) = \mathbb{E}_{\tau\sim p_\theta}\, r(\tau) = \sum_\tau p_\theta(\tau) r(\tau), \tag{2.4}$$

where $p_\theta(\tau)$ is the probability of a specific trajectory, and there may be a finite or countably infinite different number of trajectories τ. Taking the gradient of $J(\theta)$ with respect to θ then gives:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau p_\theta(\tau) r(\tau) = \sum_\tau \nabla_\theta p_\theta(\tau) r(\tau). \tag{2.5}$$

For reasons that will become clear, we recall the following identity:

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)), \tag{2.6}$$

allowing us to rewrite Eq. 2.5 as:

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_\tau p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)) r(\tau), \\ &= \mathbb{E}_{\tau\sim p_\theta}\, \nabla_\theta \log(p_\theta(\tau)) r(\tau). \end{aligned} \tag{2.7}$$

We now explain why the identity in Eq. 2.6 was used. The probability of a sampled

(i.e. experienced) trajectory τ has a probability that can be explicitly calculated only if the underlying state-transition function is known:

$$p_\theta(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t), \tag{2.8}$$

where $p(s_0)$ is the probability of starting the trajectory in state s_0 and is independent of θ, and $\pi_\theta(a_t|s_t)$ is the probability of the selected action given the state observation s_t. To better understand the notation $\pi_\theta(a_t|s_t)$, note that the policy is *stochastic*. In other words, when the policy is given a state observation s_t, the output of $\pi_\theta(s_t)$ is a vector of probabilities derived from the $softmax$ function[2]. In the discrete action-space environments considered here, there is one output probability per possible action. A random action is then drawn from the given probability distribution, and the probability of the selected action is denoted $\pi_\theta(a_t|s_t)$.

In real-world problems, the environment's state transition function $p(s_{t+1}|s_t, a_t)$ is not known, so $p_\theta(\tau)$ would be impossible to calculate. However:

$$\begin{aligned} \log p_\theta(\tau) &= \log\big(p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)\big) \\ &= \log p(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t), \end{aligned} \tag{2.9}$$

and replacing $\log p_\theta(\tau)$ in Eq. 2.7 with its expanded form gives:

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim p_\theta} \nabla_\theta \Big[\log p(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)\Big] r(\tau), \\ &= \mathbb{E}_{\tau \sim p_\theta} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) r(\tau). \end{aligned} \tag{2.10}$$

[2] $softmax(x_i|\boldsymbol{x}) := \frac{\exp(x_i)}{\sum_{j=1}^{|\boldsymbol{x}|} \exp(x_j)}$, where $\boldsymbol{x}$ is a vector of reals.

In this form, we are able to approximate the gradient. Recall that π_θ is a neural network (or some other differentiable function), so the gradient of its log may be calculated explicitly given each a_t and s_t over the trajectory. Also, we know the sum of rewards $r(\tau)$ for each trajectory. Finally, the outer expectation is approximated by performing N *episodes*, i.e. experiencing multiple trajectories, and then averaging the sums giving:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{n,t}|s_{n,t}) r(\tau_n). \tag{2.11}$$

After having obtained an approximation of the objective's gradient, we may use it to update the neural network parameters with standard stochastic gradient ascent:

$$\theta = \theta + \alpha \nabla_\theta J(\theta), \tag{2.12}$$

where α is the learning rate and whose appropriate value must be experimentally found.

The Vanilla Policy Gradient method works surprisingly well for a broad range of problems, and there are many improvements that have been made to it to increase its performance. Understanding the method presented here is a good foundation for approaching current literature. The derivation of the Vanilla Policy Gradient method as presented above is credited to [30].

2.4.1 Vanilla Policy Gradient Method in AirSim

We now apply the Vanilla Policy Gradient method to a cube collection task, illustrated in Fig. 2.5. We have extended Microsoft's AirSim simulator to support teaching a drone how to autonomously navigate a sequence of visual waypoints. This is captured by randomly distributing cubes in front of a drone at the start of each episode. The drone receives a reward for each cube it reaches. In this environment, the drone has a discrete

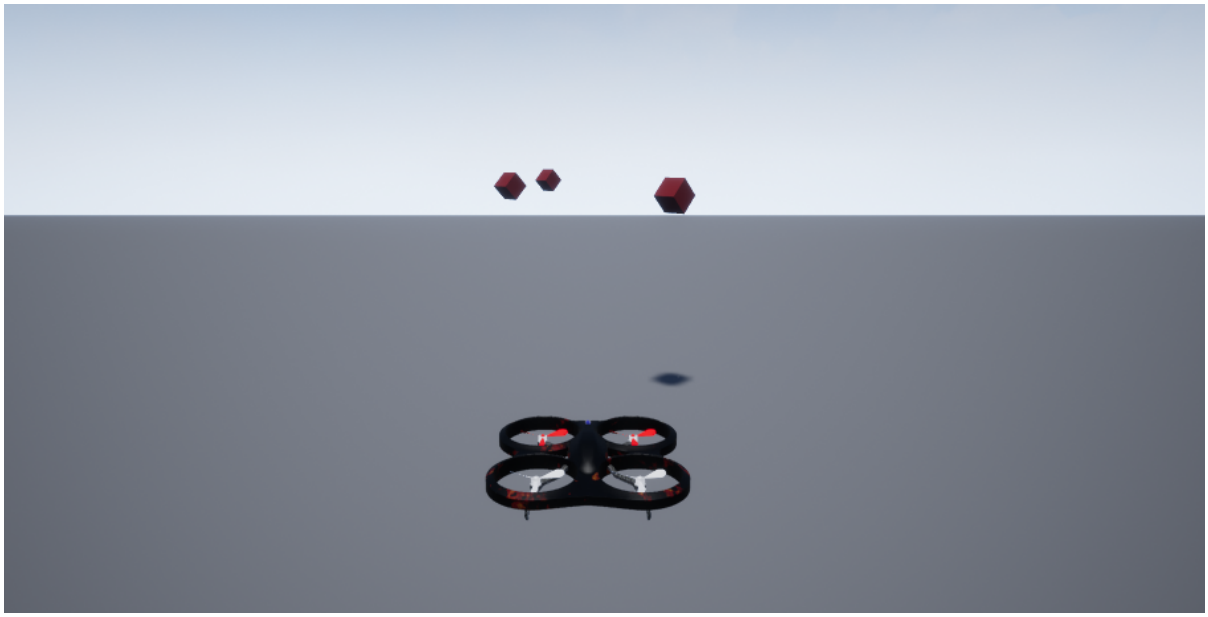


Figure 2.5: Example of our custom AirSim environment. After the environment is reset, cubes are placed randomly in front of the drone. The drone has a camera that provides input to a reinforcement learning agent. The agent contains a CNN-based policy and learns how to use the camera input to infer control decisions for collection of cubes.

action space of forward, left, and right. The observation space is continuous and is derived from a camera mounted on the drone. The source code for these experiments is available at `https://github.com/RodgerLuo/CPS-Book-Chapter`.

A convolutional neural network is used to represent the policy. We will refine the objective (from Eq. 2.3) of finding network parameters that maximize the expected sum of rewards across all time steps in the episode:

$$\theta^{\star} = \arg\max_{\theta} \mathbb{E}_{\tau \sim p_{\theta}} \sum_{t=0}^{T-1} r(s_t, a_t).$$

In the cube collection task, a reward of 1 is provided by the environment each time step a cube is collected by the drone, and 0 reward when no cube is collected, so the objective is to collect as many cubes as possible in each episode (i.e. during time step $t = 0...T-1$, where $T-1$ is the step when the last cube is collected or the drone has gone out of

bounds). In the context of Vanilla Policy Gradient, this objective was discovered by taking its gradient (from Eq. 2.11):

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{n,t}|s_{n,t}) r(\tau_n),$$

and then updating the neural network parameters based on the gradient. Recall that π_θ is the neural network, and, in the drone collection task, $\pi_\theta(a_{n,t}|s_{n,t})$ is the output probability[3] of going left, right, or forward, given input pixels from the drone's camera.

One weakness of Eq. 2.11 for our context is that the rewards are sparse, because there are only three cubes total to collect in each trajectory. If the episode return $r(\tau_n)$ is used directly as the reinforcing signal then *entire* trajectory probabilities are increased or are unchanged. This results in high variance in performance between each episode. An approach to get faster results in the cube collection task is to "smooth" the attribution of rewards from later stages to earlier stages by applying a *discounted return* to the gradient at each time step. The discounted return is defined as:

$$g_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_{T-1} = \sum_{k=t}^{T-1} \gamma^{k-t} r_k, \tag{2.13}$$

where $\gamma \in [0, 1]$ is the discount rate. The resulting $\boldsymbol{g}$ vector of Eq. 2.13 is also normalized[4] in the cube collection task. Using $\boldsymbol{g}$ we update Eq. 2.11 to give:

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) g_t, \tag{2.14}$$

where we are only collecting a single trajectory between applications of gradient ascent.

[3]Softmax of the network's logits.

[4]Normalization is defined as $\boldsymbol{g} \leftarrow (\boldsymbol{g} - \mu(\boldsymbol{g}))/\sigma(\boldsymbol{g})$, where scalar operations are applied element-wise to the vector.

There are much better approaches than using the discounted return, and our source code example is parameterized to allow experimentation with other reward function alternatives.

Algorithm 1: Vanilla Policy Gradient algorithm in the context of the AirSim cube collection task.

Input: Old policy parameters θ, learning rate α, tuple of (observations s, actions a, rewards r) from last cube collection episode
Output: Updated policy θ

Apply Eq. 2.13 to obtain discounted rewards g from a
Normalize g
Set $sum\_of\_grads = 0$
for $t = 0 \dots T-1$ **do**
  $sum\_of\_grads = sum\_of\_grads + g_t \nabla_\theta \log \pi_\theta(a_t|s_t)$
end
$\theta = \theta + \alpha sum\_of\_grads$
return θ

We summarize the use of the Vanilla Policy Gradient method in the context of our AirSim cube collection task in Algorithm 1. This algorithm is implemented in the provided source code and will train a drone to collect cubes based on visual observations.

2.5 Summary

In this chapter we introduced the Markov Decision Process, which is the sequential decision making mathematical model that reinforcement learning solves. We also introduced the basic families of reinforcement learning: policy-based, value function-based, and model-based. We then provided a derivation of the Vanilla Policy Gradient method, which is the foundational policy-based algorithm. The VPG method was then used to train a drone to collect cubes in Microsoft AirSim.

Chapter 3

Mathematical Optimizations for Deep Learning

3.1 Introduction

Modern DNN architectures require billions of floating-point multiplications and additions (MACs) for inference of a single input. Without careful design, this results in high power consumption and high latency. Fossil-fuel powered vehicles, for example, can support high energy demands, but efficient, battery powered systems cannot. Additionally, modern large DNNs have high latency, but low latency is required for real-time autonomous applications. This chapter provides a unified view of the leading methods for mathematically-optimized deep learning inference. The methods introduced in this chapter will be applied to deep RL algorithms and applications in following chapters.

To motivate the need for optimizations, it is helpful to consider first-order power and silicon area requirements for DNN inference. Table 3.1 provides a list of energy and die area required for various operator and operand sizes. Observe that a single 32-bit floating-point multiplication (denoted "32b FP Mult") requires 20 times more power and

Operation	Energy (pJ)	Area (um)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32 FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read	5	N/A
32b DRAM Read	640	N/A

Table 3.1: Energy and die area costs for various operations (45nm) [31]. Quantized operators and operands are preferred for low-power and low-resource applications. FP stands for floating-point.

12 times more area than 8-bit integer multiplication ("8b Mult"). Also observe that the power cost of a 32-bit DRAM read is more than 100 times the cost of floating-point multiplication. For this reason, efficient DNN implementations should prioritize the minimization of off-chip DRAM access first, followed by reducing operand and operator sizes. Naturally these two approaches complement one another.

DNN optimizations are useful only during the inference operation. Currently, training a DNN to reach state of the art performance requires the backpropagation algorithm, which uses gradient descent to make many small adjustments to the neural network parameters. These small adjustments must be calculated and stored using full-precision accumulation. Therefore the optimizations discussed in this chapter are not primarily aimed at making training more efficient, but they are intended to make inference more efficient.

To further emphasize the need for inference efficiency, consider the number of operations required to evaluate various modern DNNs, given in Table 3.2. This table provides a first-order estimate for MAC and memory costs for popular DNN architectures. Power

estimates assume 32-bit floating-point arithmetic and are derived from Table 3.1. MAC costs capture the power requirement for each network to perform the necessary operations for providing a single inference. The memory cost is best-case and assumes parameters are read from DRAM only once per inference; actual memory costs will be higher if intermediate results must be transferred back to DRAM during inference of the network. In Table 3.2 note that even though the number of MACs are much greater than the number of parameters, the high DRAM read cost results in the power consumed between the two to be roughly equivalent.

Metrics	LeNet 5	AlexNet	Overfeat fast	VGG 16	GoogLeNet v1	ResNet 50
Parameters	60k	61M	146M	138M	7M	25.5M
Read Cost (8b)	10μJ	10mJ	23mJ	22mJ	1mJ	4mJ
Read Cost (32b)	38μJ	39mJ	93mJ	88mJ	4mJ	16mJ
MACs	341k	724M	2.8G	15.5G	1.43G	3.9G
MAC Cost (8b)	.1μJ	167μJ	644μJ	3565μJ	329μJ	897μJ
MAC Cost (32b)	2μJ	3mJ	13mJ	71mJ	7mJ	18mJ

Table 3.2: Number and cost of parameters and MACs for popular deep neural network architectures. Cost estimates are based on Table 3.1 and from architecture statistics provided in [32]. Note that memory costs are typically higher than MAC costs.

The process of DNN training may be thought of as an exploration over a parameter space to find values which will solve an inference task. As will be expanded on, the parameters found using standard training methods result in DNNs which are overparameterized, which means they have redundancy. When the DNN performs satisfactorily during cross-validation, backpropagation is no longer needed, and optimizations may be applied to decrease parameter redundancy. The goal of mathematical optimizations for deep learning is to find the most compact network which performs satisfactorily at

its assigned real-world inference tasks.

DNN architectures are composed of various layer types: convolutional, fully-connected, dropout, pooling, and others. Each layer type was developed to solve a particular weakness and each classification problem is best solved by a different architecture, or combination of layers. Convolutional and fully-connected layers represent the greatest computational expense in DNN inference, and optimizing these layer types is the focus of this chapter. Both convolutional and fully-connected layers require repeated multiplication and addition, but they typically use different algorithmic steps. Adapting notation of [33], we represent an L-layer DNN as $\langle \mathcal{I}, \mathcal{W}, \mathcal{O} \rangle$, where:

- $\mathcal{I}_l \in \mathbb{R}^{c_{in} \times x \times y}$ and $\mathcal{W}_l \in \mathbb{R}^{c_{in} \times w \times h \times c_{out}}$ are layer l's input tensors and parameter tensors respectively. c_{in} represents the number of *input channels* and c_{out} represents the number of *output channels*[1]. x and y are the width and height of each input channel, and w and h are the width and height of each filter.

- $\mathcal{O}_l \in \{*, \cdot, \text{other}\}$ specifies whether the layer's operation type is convolution ($*$), fully-connected ($\cdot$), or some other less computationally expensive type.

Convolutional layers convolve a $\mathbb{R}^{c_{in} \times w \times h \times c_{out}}$ parameter filter tensor with a $\mathbb{R}^{c_{in} \times x \times y}$ input tensor, where (w,x) and (h,y) represent the widths and heights of the two respective tensors and may be different sizes, and c_{in} and c_{out} represent the number of input and output channels. In particular the (w, h) for parameter filters are often smaller than the (x, y) for inputs. c is the number of channels in the given layer; this value is equal for both the parameter filter tensor and input tensor. As illustrated in Figure 3.1 (top), each step in the convolution requires a sum of products between elements of the parameter filter and elements of the receptive field of the input filter.

[1] Also called input filter maps (ifmaps) and output filter maps (ofmaps) in literature.

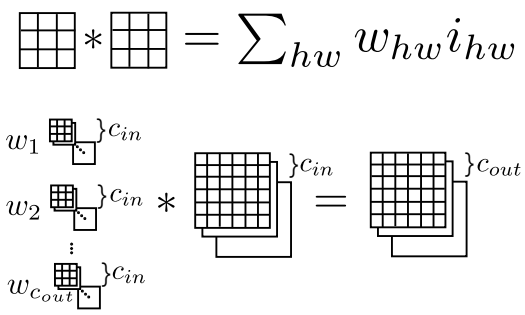


Figure 3.1: Convolutional layers convolve a parameter filter with an input. Filters are usually 5×5, 3×3, or 1×1. Each step of the convolution involves multiplying and accumulating elements of the parameter filter with a *receptive field* of the input. The top illustration represents the basic convolution operation ($*$). The lower illustration represents c_{out}, c_{in}-channel filters which are convolved with a c_{in}-channel input tensor, which results in an c_{out}-channel output tensor.

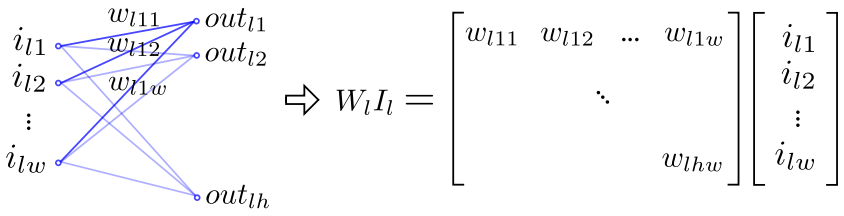


Figure 3.2: Fully-connected layers flatten the input tensor into a vector and multiply by a parameter matrix with the same number of columns as the vector and as many rows as desired.

Note that what is shown in Figure 3.1 (top) only depicts convolution of a single channel. If there are multiple channels, then the summation is also over all channels. Figure 3.1 (bottom) shows a higher-level view, where each c_{in}-channel parameter filter is convolved with the c_{in}-channel input tensor. When multiple channels are included in the convolution, each output of the convolution becomes the triple-sum across the channels. The number of parameter filters in a layer equals the number of channels in the output tensor: if there are c_{out} parameter filters, there will be c_{out} channels in the output tensor.

Computation for fully-connected layers requires a single matrix-vector product. The input tensor $\mathcal{I}_l \in \mathbb{R}^{c_{in} \times x \times y}$ is flattened to a vector $\in \mathbb{R}^{c_{in} \cdot x \cdot y}$. The parameter tensor is denoted $\mathcal{W} \in \mathbb{R}^{w \times h}$, where $w = c_{in} \cdot x \cdot y$ (from the input tensor dimensions) and h is equal

to the number of desired output units from the fully-connected layer. An illustration of a fully-connected layer is given in Fig. 3.2.

After a parameter filter $\mathcal{W}$ is convolved with an input $\mathcal{I}$ in a convolutional layer, or the matrix-vector product between parameters and layer inputs is produced for a fully-connected layer, the resulting matrix of vector entries are typically passed through a nonlinearity function $\sigma : \mathbb{R} \to \mathbb{R}$. A commonly used nonlinearity is the rectified-linear unit (ReLU), which is defined as:

$$\sigma_{\mathrm{ReLU}}(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{else.} \end{cases} \tag{3.1}$$

But more extreme nonlinearities exist, such as the binarized activation function which outputs only two values, –1 and 1:

$$\sigma_b(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ -1 & \text{else.} \end{cases} \tag{3.2}$$

The choice of nonlinearity function influences the performance and computational cost of inference. Specifically, using the binarized activation function can lead to the elimination of floating-point and fixed-point arithmetic during inference, as detailed in Subsection 3.3.2.

Both convolutional and fully-connected layers require many memory access and MAC operations, but a variety of numerical optimizations may be applied to DNN inference. Some optimizations reduce power and some optimizations reduce both power and latency. Furthermore, it is possible to optimize a DNN and maintain classification accuracy, but there also exist extreme optimization methods which result in unavoidable accuracy loss.

Depending on the application, decreased accuracy may be worth the reduction in power and latency.

The remainder of this chapter provides an introduction to the common approaches of DNN mathematical optimization: pruning, quantization, parameter sharing and compression, model distillation, and filter decomposition.

3.2 Pruning

Pruning applies to fully-connected and convolutional layers and eliminates each layer's smallest parameters, which has the consequence of reducing the number of MAC operations, the amount of traffic required to transfer parameters, and storage requirements. The typical procedure is to train the network until the desired accuracy is reached and then to prune the smallest p^{th}-percentile of parameters by setting them to zero. Pruning is followed by fine-tuning the remaining parameters, which can be accomplished using the same dataset as used during initial training.

In [35], the authors report 9× and 13× reduction in parameters for AlexNet and VGG-16 with no impact on test accuracy. A histogram of the normalized frequency of parameters is given in Fig. 3.3, where the smallest 50^{th}-percentile is delineated with two vertical lines. In practice, one would pick the percentile threshold for each layer heuristically, that is, the percentile threshold would be a hyperparameter for each layer. This process is represented in Algorithm 2.

After pruning, the resulting DNN will be sparse, with many parameters set to zero. Standard architectures, like GPUs, are currently not designed to take advantage of sparsity and will perform multiplication regardless if one of the operands is zero. In order to benefit from pruning, the architecture must be designed in such a way as to take advantage of sparsity. This will add edge cases to standard logic design. For example, consider

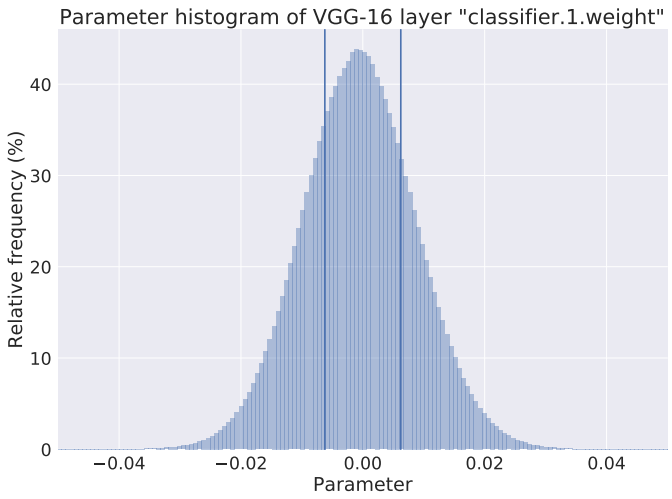


Figure 3.3: Histogram of parameters of the first fully-connected layer in VGG-16. The name "classifier.1.parameter" corresponds to the VGG-16 implementation found in torchvision [34]. The two vertical lines correspond to thresholds of values smaller than the 50^{th}-percentile. These values may be pruned (permanently set to zero) and the remaining values fine-tuned with no loss in accuracy [35]. The same procedure may be applied to all other layers in the network.

a product summation tree, which can parallelize MAC operations. Even if the tree is designed to ignore products with a zero operand, it must still take into account that the zero product must be passed to the next tree level at the appropriate time. Recently, architectures for handling sparse dataflows have been developed. One such architecture reduces the amount of "wasted" logic required for ignoring zero-products by only passing non-zero products to processing elements downstream [36].

3.3 Quantization

Before 2015, most DNNs were trained using 32-bit floating-point arithmetic. In this section we summarize approaches for using reduced precision, or *quantized*, arithmetic for DNN inference. Quantization reduces the amount of parameter data that must be transferred from DRAM to processing elements. Additionally, quantized arithmetic is less

Algorithm 2: Pruning

Require: L-layer DNN $\langle \mathcal{I}, \mathcal{W}, \mathcal{O}, \mathcal{P} \rangle$, where $\mathcal{I}_l$ and $\mathcal{W}_l$ are layer l's input tensors and parameter tensors respectively, and $\mathcal{O}_l$ specifies whether the layer's type is convolutional, fully-connected, or some other type, and $\mathcal{P}$ is the pruning percentile for each layer.

Ensure: Pruned and fine-tuned network parameters $\mathcal{W}$.

1. Initial training:
Perform standard training of DNN until satisfactory performance is achieved.

2. Pruning:
For each layer l in $\langle \mathcal{I}, \mathcal{W}, \mathcal{O}, \mathcal{P} \rangle$, eliminate parameters in $\mathcal{W}_l$ which are less than layer l's p^{th} percentile, where $p = \mathcal{P}_l$.

3. Fine-tuning:
Perform standard re-training of remaining parameters $\mathcal{W}$, until maximum performance is achieved.

expensive in terms of power and silicon area than full-precision arithmetic. Quantization may be applied to parameters, activations, or both parameters and activations. We emphasize that quantization techniques using <16-bits currently only provide efficiency benefits during inference, because backpropagation requires accumulation of small values, and therefore ≥16-bits.

It appears that 8-bit or 16-bit quantization is adequate for most DNN inference tasks. For example, Google's DNN accelerator, the Tensor Processing Unit (TPU), exclusively uses 8-bit or 16-bit integer arithmetic [37]. The TPU (and the successor TPUv2) has become a critical component of Google's computing ecosystem. Additionally, NVIDIA's Pascal architecture was designed to support 16-bit floating-point and 8-bit integer arithmetic.

In this section we focus on extreme quantization methods which binarize parameters and activations. Binarization usually has a large negative impact on performance, but we present techniques in Subsections 3.3.1 and 3.3.2 which reduce the impact.

Note that in this section we will sometimes use a unified notation which applies

to both convolutional and fully-connected layers. In a convolutional layer, a c-channel parameter filter $\mathcal{W} \in \mathbb{R}^{c \times w \times h}$ is convolved with an input $\mathcal{I} \in \mathbb{R}^{c \times w \times h}$. Convolution is performed by $\mathcal{W} * \mathcal{I}$. At a specific receptive field, the core operation may be interpreted as the inner-products between vectors. In this section, we sometimes use the notation $\mathcal{W}^\top \mathcal{I}$ to denote the convolution of a filter with a specific receptive field. Simultaneously, the $\mathcal{W}^\top \mathcal{I}$ notation captures the partial calculation of a fully-connected layer.

3.3.1 Binary parameters

In 2015, BinaryConnect [38] was an early DNN quantization method, and exemplifies the field's approach to quantization. During inference, BinaryConnect quantizes full-precision DNN parameters $\mathcal{W}$ to $\{-1, 1\}$, using the sign function:

$$w^{(b)} = \begin{cases} +1 & \text{if } w \geq 0, \\ -1 & \text{else.} \end{cases} \tag{3.3}$$

Equation 3.3 discards real-valued information, but, in doing so, it also eliminates the need for floating-point multiplication during inference. Instead, signed floating-point addition may be used for unit activation input calculations. During backpropagation, the error caused by quantization is used to update the real-valued $\mathcal{W}$s. After training is complete, full-precision parameters and arithmetic are no longer required and may thereafter be discarded. From a hardware perspective, memory overhead is $32\times$ less when using BinaryConnect-derived parameters. However, this technique has an accuracy cost. When using the AlexNet DNN architecture, BinaryConnect achieves 61% top-5 accuracy on ImageNet, compared to 80.2% accuracy when using AlexNet with 32-bit full-precision accuracy [33].

In Algorithm 3 we outline the steps of BinaryConnect. Note here that we separate the

bias terms from $\mathcal{W}$, where normally it is included in that tensor for notation convenience. The reason here is that the bias is always added, even with full-precision arithmetic, so there is no benefit to quantize it. Also note the `clip` function in Algorithm 3 limits the full-precision parameters to between $[-1, 1]$.

Algorithm 3: BinaryConnect [38]

Require: Inputs $\mathcal{I}$, targets y, previous full-precision parameters $\mathcal{W}$, biases b, learning rate η, and objective function J.

Ensure: Updated $\{-1, 1\}$-valued parameters $\mathcal{W}^{(b)}$ and real-valued bias b.

1. Forward propagation:

$\mathcal{A}_0 = \mathcal{I}$

for $l = 1$ to L

  for k^{th} filter in l^{th} layer

    $\mathcal{W}_{lk}^{(b)} \leftarrow \texttt{binarize}(\mathcal{W}_{lk})$ using Equation 3.3

    $\mathcal{A}_{lk} \leftarrow \mathcal{W}_{l}^{(b)} * \mathcal{A}_{(l-1)k} + b_{lk}$

2. Backward propagation:

Initialize output layer's activation gradient $\frac{\partial J}{\partial A_L}$ using y, $\mathcal{A}_L$, and J

for $l = L$ to 2

  for k^{th} filter in l^{th} layer

    Compute $\frac{\partial J}{\partial A_{(l-1)k}}$ knowing $\frac{\partial J}{\partial A_{lk}}$ and $\mathcal{W}_{lk}^{(b)}$

3. Update parameters:

Compute $\frac{\partial J}{\partial \mathcal{W}_{lk}}$ and $\frac{\partial J}{\partial b_{lk}}$, knowing $\frac{\partial J}{\partial A_{lk}}$ and $A_{(l-1)k}$

$\mathcal{W} \leftarrow \texttt{clip}(\mathcal{W} - \eta \frac{\partial J}{\partial \mathcal{W}})$

$b \leftarrow b - \eta \frac{\partial J}{\partial b}$

Not made explicit in Algorithm 3 is how the gradient signal passes through the binarization function given in Equation 3.3. This is required for calculation of $\partial J / \partial \mathcal{W}_{lk}^{(b)}$. We cannot merely take the derivative of the binarization function, because it is 0 everywhere except at $\mathcal{W} = 0$, where the function is discontinuous. To handle this, the authors used a variant of the *Straight-Through Estimator* (STE) during backpropagation [39]. The

modified STE is defined as:

$$\texttt{STE}(x) = \begin{cases} 0 & \text{if } x < -1, \\ 1 & \text{if } -1 \leq x \leq 1, \\ 0 & \text{if } x > 1. \end{cases} \tag{3.4}$$

During backpropagation, the derivative of the parameter binarization function (Eq. 3.3) is calculated with respect to each full-precision parameter: $\frac{d\mathcal{W}_{lk}^{(b)}}{d\mathcal{W}_{lk}}$. Because the parameter binarization function is discontinuous, its derivative must be estimated, which is achieved by replacing it with the `STE` evaluated at the full-precision parameter. Multiplying by the `STE` during backpropagation has the effect of canceling the gradient when the full-precision parameter's magnitude is too large.

To summarize BinaryConnect, we take the sign of the real-valued parameters during inference. During backpropagation, the errors caused by binarization may be very small (with significant changes accumulating over many inputs) and we track those small changes in full-precision versions of the parameters. After training is complete, the full-precision parameters may be discarded, only keeping their sign information.

XNOR-Net [33] introduced a method which is almost identical to BinaryConnect, but it performs binarization in a way which achieves higher accuracy. As with BinaryConnect, parameters are binarized during inference, but then they are also scaled by a factor which attempts to compensate for the binarization. Specifically, XNOR-Net introduced the following approximation for the inner-product[2]:

$$\mathcal{W}^\top \mathcal{I} \approx \alpha \mathcal{W}^{(b)\top} \mathcal{I}, \tag{3.5}$$

[2]Note that we consider $\mathcal{W}$ and $\mathcal{I}$ to be flattened.

where $\mathcal{W}^{(b)}$ is the binarized version of $\mathcal{W}$ using Equation 3.3. This notation is slightly different than that used in Algorithm 3, where we are able to binarize the entire $\mathcal{W}$ tensor at once. But with XNOR-Net, each filter in each convolutional layer requires a separate α. To keep the notation simple, separate filters are not denoted.

To find the optimal scaling factor α, we solve the following optimization problem:

$$\begin{aligned} J(\alpha) &= \left\| \mathcal{W} - \alpha \mathcal{W}^{(b)} \right\|^2, \\ \alpha^* &= \arg\min_{\alpha} J(\alpha). \end{aligned} \tag{3.6}$$

That is, we are seeking an α which minimizes the distance between $\mathcal{W}$ and $\alpha\mathcal{W}^{(b)}$. For intuition, consider a scalar w and its binarized version $w^{(b)}$; in this case $\alpha = w/w^{(b)}$ perfectly minimizes the distance between w and $w^{(b)}$. Expanding the norm in Equation 3.6 gives:

$$J(\alpha) = \alpha^2 \mathcal{W}^{(b)\top} \mathcal{W}^{(b)} - 2\alpha \mathcal{W}^{\top} \mathcal{W}^{(b)} + \mathcal{W}^{\top} \mathcal{W}. \tag{3.7}$$

We now take the derivative of $J(\alpha)$ with respect to α, set it to zero, and solve for α:

$$\frac{dJ(\alpha)}{d\alpha} = 2\alpha \mathcal{W}^{(b)\top} \mathcal{W}^{(b)} - 2\mathcal{W}^{\top} \mathcal{W}^{(b)}. \tag{3.8}$$

Let $n = \mathcal{W}^{(b)\top} \mathcal{W}^{(b)}$, which is also equal to the number of parameters in the binarized filter. Substituting n into Equation 3.8, setting it to zero, and solving for α gives α^*:

$$\alpha^* = \frac{\mathcal{W}^{(b)\top} \mathcal{W}^{(b)}}{n} = \frac{\mathcal{W}^{(b)\top} \text{sign}(\mathcal{W})}{n} = \frac{\sum |\mathcal{W}|}{n}. \tag{3.9}$$

New α^*s must be calculated every time $\mathcal{W}$ changes, i.e. each time backpropagation is used to update the parameters, but, after the training is completed, α^* may be saved for use during inference.

Signed Multiplication		
Inputs		Output
$\mathcal{I}_i$	$\mathcal{W}_i$	$\mathcal{I}_i \times \mathcal{W}_i$
−1	−1	1
−1	1	−1
1	−1	−1
1	1	1

XNOR "Multiplication"		
Inputs		Output
$\mathcal{I}_i$	$\mathcal{W}_i$	$\overline{\mathcal{I}_i \oplus \mathcal{W}_i}$
0	0	1
0	1	0
1	0	0
1	1	1

Table 3.3: The XNOR operation captures the behavior of signed multiplication.

Using the parameter binarization methods above, we may eliminate most multiplications from inference[3], and instead we only need signed addition. If we assume 32-bit multiplication and addition, this results in 32× power reduction for parameter transfer from DRAM and $\sim 3\times$ power reduction for arithmetic. When using the AlexNet DNN architecture, XNOR-Net (binary parameters, full-precision activations) achieves 79.4% top-5 accuracy on ImageNet, compared to 80.2% accuracy when using AlexNet with 32-bit full-precision accuracy [33]. We next consider operator optimizations which become available when both parameters *and* inputs are binarized.

3.3.2 Binary parameters and Activations

If parameters and activations are binarized, then we are able to eliminate almost all floating-point (and fixed-point) calculations, resulting in extreme energy savings. Specifically, when parameters and inputs are binarized, the XNOR operation[4] may be used to calculate inner-products during inference [40]. The XNOR logic truth table is given on the right in Table 3.3. The left-hand side provides the truth table for signed multiplication between scalar values $\mathcal{I}_i \in \mathcal{I}$ and $\mathcal{W}_i \in \mathcal{W}$. Note that by mapping −1 to 0, the two tables give identical output.

[3]Multiplication by α is still necessary when using the parameter binarization technique in XNOR-Net.

[4]Not to be confused with XNOR-Net [33]. Here we are referring to the exclusive-NOR operation.

XNOR logic is simple and efficient to implement in hardware and may be used as the multiplication operator for the calculation of inner-products during inference. To use the XNOR "product" between $\mathcal{I}$ and $\mathcal{W}$ for the input into a unit's nonlinearity function, we first map all −1s to 0s, then calculate the XNOR values for both vectors. The Hamming parameter[5] (HW) of the XNOR vector result is then compared to $\#bits/2$, where $\#bits$ is the size of $\mathcal{W}$ and $\mathcal{I}$. If the Hamming parameter is greater than or equal to $\#bits/2$ then output 1, otherwise output 0. Note that after the initial mapping of −1 to 0, we no longer need to map back to −1 during the remainder of the inference procedure.

BinaryNet [40] operates similarly to BinaryConnect, with the addition that activations are also binarized. When using BinaryNet, the activation inputs are summed, as with BinaryConnect, and then the resulting sum is converted to $[-1, 1]$ using the sign function. This optimization eliminates all full-precision calculations and replaces them with signed integer calculations. As with BinaryConnect, BinaryNet requires full-precision gradient updates during training, and during backpropagation the `STE` function (Eq. 3.4) is used for both the activation and parameters. BinaryNet achieves 50.42% top-5 accuracy on AlexNet, compared to 80.2% accuracy when using the same DNN topology and 32-bit full-precision accuracy [33].

XNOR-Net also has a version which binarizes both parameters and activations. Similar to XNOR-Net's parameter-only binarization presented above, there is a scaling factor α which may (optionally) be used to reduce the error between full-precision and binarized dot products:

$$\begin{aligned} J(\alpha) &= \left\| \mathcal{I}^\top \mathcal{W} - \alpha \mathcal{I}^{(b)\top} \mathcal{W}^{(b)} \right\|^2, \\ \alpha^* &= \arg\min_\alpha J(\alpha). \end{aligned} \tag{3.10}$$

[5]Hamming parameter is defined as the number of 1s in a vector.

This is solved in the same manner as Equation 3.6, giving:

$$\alpha^* = \frac{\sum|\mathcal{I}^{(b)\top}\mathcal{W}^{(b)}|}{n} = \frac{\sum|\mathcal{I}||\mathcal{W}|}{n}. \tag{3.11}$$

Note that a separate scaling factor α^* must be solved for each receptive field and parameter filter combination *both during training and when using the neural network after training*. This high computational overhead limits the use of vanilla XNOR-Net. Fortunately, in practice, the authors of BinaryNet found that the scaling factor for binarized parameters was much more important than the scaling factor for binarized inputs, and may therefore be ignored. We summarize the parameter-scaled version of XNOR-Net with the following algorithm:

Algorithm 4: (parameter-scaled) XNOR-Net [40]

Require: Inputs $\mathcal{I}$, targets y, previous full-precision parameters $\mathcal{W}$, biases $\boldsymbol{b}$, learning rate η, and objective function J.

Ensure: Updated $\{-1, 1\}$-valued parameters $\mathcal{W}^{(b)}$, parameter scaling factors $\boldsymbol{\alpha}$, and real-valued bias $\boldsymbol{b}$.

1. Forward propagation:

$\mathcal{A}_0 = \texttt{binarize}(\mathcal{I}_0)$

for $l = 1$ to L

  for k^{th} filter in l^{th} layer

    $\alpha_{lk} = \frac{1}{n}||\mathcal{W}_{lk}||_{\ell 1}$

    $\mathcal{W}_{lk}^{(b)} \leftarrow \texttt{binarize}(\mathcal{W}_{lk})$ using Equation 3.3

    $\mathcal{A}_{lk}^{(b)} \leftarrow \texttt{binarize}\big((\alpha_{lk}\mathcal{W}_{lk}^{(b)}) * \mathcal{A}_{(l-1)k}^{(b)} + b_{lk}\big)$ using Equation 3.3

2. Backward propagation:

Initialize output layer's activation gradient $\frac{\partial J}{\partial A_L}$ using y, $\mathcal{A}_L$, and J

for $l = L$ to 2

  for k^{th} filter in l^{th} layer

    Compute $\frac{\partial J}{\partial A_{(l-1)k}^{(b)}}$ knowing $\frac{\partial J}{\partial A_{lk}^{(b)}}$ and $\mathcal{W}_{lk}$

3. Update parameters:

Compute $\frac{\partial J}{\partial \mathcal{W}_{lk}^{(b)}}$ and $\frac{\partial J}{\partial b_{lk}}$, knowing $\frac{\partial J}{\partial A_{lk}^{(b)}}$ and $A_{(l-1)k}$

$\mathcal{W} \leftarrow \texttt{clip}(\mathcal{W} - \eta\frac{\partial J}{\partial \mathcal{W}^{(b)}})$

$b \leftarrow b - \eta\frac{\partial J}{\partial b}$

Similar to the calculation of $\partial J/\partial \mathcal{W}_{lk}^{(b)}$ in Algorithm 3, both partial-derivatives $\partial J/\partial \mathcal{W}_{lk}^{(b)}$ and $\partial J/\partial \mathcal{A}_{lk}^{(b)}$ in Algorithm 4 are substituted with the `STE` function in Equation 3.4, where the inputs to `STE` are the real-valued parameter and activation respectively.

XNOR-Net using binarized inputs and parameters achieves 69.2% accuracy on AlexNet, compared to BinaryNet's 50.42%, and full-precision accuracy of 80.2%. The XNOR-Net and BinaryNet papers introduce other training tips for improved performance. The aggregate contributions of the performance techniques introduced in XNOR-Net likely account for its significant gain over BinaryNet.

3.4 Parameter Sharing and Compression

Top-performing neural networks use millions of parameters which are typically transferred from DRAM to processing elements for inference (see Table 3.2). When these parameters are transferred, DRAM energy cost can surpass arithmetic cost for performing a single inference. Parameter sharing clusters parameters into shared values and is applied after the network has reached peak performance. Once parameters have been clustered, compression may be used to transmit cluster indices instead of full-precision values. Parameter sharing coupled with data transfer compression is a method to retain the high performance typically provided by large full-precision neural networks, while simultaneously reducing the amount of data sent over DRAM [35].

3.4.1 Parameter Sharing

To apply parameter sharing, first, the DNN is trained to maximum performance using standard training methods. After training, each layer's parameters are grouped into clusters, where the number of parameters in a layer is much greater than the number of clusters. After assigning parameters to clusters, the network goes through a retraining

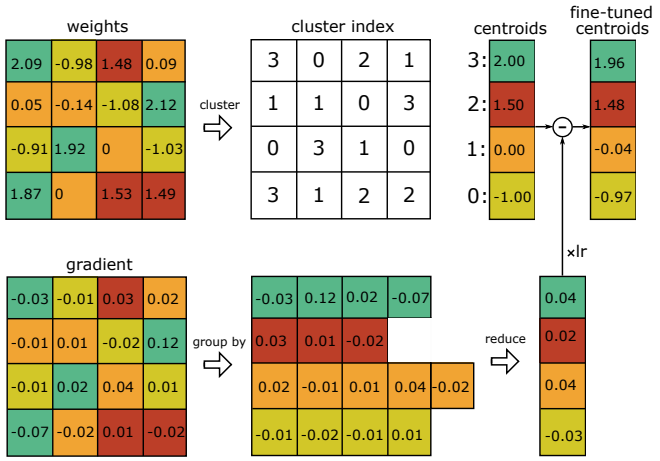


Figure 3.4: After training, 16 parameters have been clustered into 4 centroids. From that point on, clustered parameters are equal to their centroid. Partial derivatives are calculated with respect to the parameter values, as usual, but the gradients are accumulated and subtracted from the centroids [35].

phase.

For example, consider Fig. 3.4 which illustrates a 4×4 channel from some parameter filter in $\mathcal{W}$. Assume that the filter is part of a trained network. To apply parameter sharing, we use k-means clustering [35], which assigns the parameters $w \in \mathcal{W}$ to m cluster assignments $C^\star = \{c_1, c_2, \cdots, c_m\}$, such that the within-cluster sum of squares is minimized:

$$C^\star = \arg\min_{C} \sum_{i=1}^{m} \sum_{w \in c_i} |w - c_i|^2. \tag{3.12}$$

After assignment to clusters, we calculate the centroids $\tilde{w}_i$ of each cluster c_i by taking the average value of each cluster:

$$\tilde{w}_i = \frac{1}{|c_i|} \sum_{w \in c_i} w. \tag{3.13}$$

In Fig. 3.4, $m = 4$, and the top portion of the plot illustrates 16 parameters and their

associated clusters and centroids.

After clustering, parameters in the original filter are replaced by their centroid value (this is represented by the shading in Fig. 3.4). Next, the clustered parameters are fine-tuned by reusing the original training data. The key difference between standard training and the fine-tuning phase is how the parameters are updated during backpropagation. In backpropagation each parameter is changed a small amount in the direction which will improve an objective function, e.g.:

$$\mathcal{W}_{l,w} = \mathcal{W}_{l,w} - \eta \frac{\partial J(\mathcal{W})}{\partial \mathcal{W}_{l,w}}. \tag{3.14}$$

However, after clustering, we apply backpropagation to the centroid value of each parameter cluster. For example, suppose the centroid $\tilde{w}_i$ of parameter cluster c_i is to be updated using backpropagation. To update centroid $\tilde{w}_i$ we use the sum of partial derivatives with respect to parameters assigned to that cluster:

$$\tilde{w}_i = \tilde{w}_i - \eta \sum_{w \in c_i} \frac{\partial J(\mathcal{W})}{\partial w}. \tag{3.15}$$

The lower portion and the subtraction in Fig. 3.4 illustrates the gradient descent step of backpropagation when using clustering. After the fine-tuned centroids have been calculated, they will replace the previous parameter values in each cluster. The update given in Equation 3.15 is repeated until maximum performance is attained.

The steps for parameter sharing are provided in Algorithm 5. The algorithm is written from the perspective of CNNs, but adapting it for other DNN designs only requires clustering the appropriate values. For example the values in fully-connected layer could be clustered.

After parameter values have been clustered and fine-tuned, there is an opportunity

Algorithm 5: parameter Sharing

Require: Inputs $\mathcal{I}$, previously *trained* full-precision parameters $\mathcal{W}$, number of clusters m, learning rate η, objective function J.

Ensure: Clustered and fine-tuned parameters $\mathcal{W}$

1. Cluster assignment:

for $l = 1$ to L

for k^{th} filter in l^{th} layer

Assign parameters in filter k to m clusters using Equation 3.12:

$C^{\star} \leftarrow \texttt{knn}(\mathcal{W}_{lk}, m)$

Replace parameters in each cluster with centroid value using Equation 3.13:

$\mathcal{W}_{lk} \leftarrow \texttt{centroid}(\mathcal{W}_{lk}, C^{\star})$

2. Inference:

Perform standard inference using centroid-mapped parameters.

3. Fine-tuning:

Calculate standard partial derivatives with respect to parameters $\frac{\partial J(\mathcal{W})}{\partial w}$.

Update centroid values by summing partial derivatives in each cluster and using gradient decent:

$\tilde{w}_i = \tilde{w}_i - \eta \sum_{w \in c_i} \frac{\partial J(\mathcal{W})}{\partial w}$

Replace parameters in each cluster with updated centroid values.

4. Optionally repeat:

Repeat steps 2 and 3 until objective function is optimized.

to decrease the storage and traffic requirements for loading the DNN parameters from memory to an accelerator. This process is detailed in the following subsection.

3.4.2 Compression

Parameter sharing reduces the amount of data transmitted from DRAM by intentionally creating redundancy in the form of a cluster index. For example, in Fig. 3.4 we see that 16 original values are represented by 4 cluster values. Redundancy created by parameter sharing is exploitable with compression methods [35].

If a network uses b-bits of precision, then a full-precision network with n parameters requires nb-bits of transmission. After parameter sharing, only a single full-precision value (the centroid) must be transmitted for each cluster, this results in mb-bits. The

indices for m clusters are represented with $\log_2(m)$ bits, therefore transmitting n indices requires $n\log_2(m)$ bits. In general, n parameters assigned to m clusters compresses the parameters by a factor of:

$$\frac{nb}{n\log_2(m) + mb}. \tag{3.16}$$

For example, referring to Fig. 3.4, and assuming 32-bit floating-point parameters, we see that $nb = 16 \cdot 32$ and $n\log_2(m) + mb = 16 \cdot 2 + 4 \cdot 32$. Therefore, by using parameter sharing and compression, we reduce the traffic by a factor of 352.

3.5 Model Distillation

Large neural networks have a tendency to generalize better than smaller networks. Similarly, ensemble methods combine the predictions of multiple algorithms, e.g. DNNs, random forests, SVMs, logistic regression, etc., and almost always outperform the predictions from an individual algorithm. Both large networks and ensemble methods are attractive from an accuracy perspective, but many applications cannot support the time or energy it takes to perform inference using such approaches. Model distillation is the training of a smaller, more efficient, DNN to predict with the performance close to a larger DNN or ensemble [41, 42].

When training a multiclass network, first, the softmax of network logits a_i is used to calculate class probabilities:

$$\hat{y}_i = \frac{\mathrm{e}^{a_i/T}}{\sum_{j=1}^{|C|} \mathrm{e}^{a_j/T}}, \tag{3.17}$$

where C is the set of classes which the network can identify, and T is the temperature and is usually set to 1. Class probabilities are then used in the cross-entropy error function:

$$J(y, \hat{y}) = -\sum_{i=1}^{|C|} y_i \log \hat{y}_i, \tag{3.18}$$

where y is the correct training label for a given input, and $\hat{y}$ is the vector of class prediction probabilities output from the network. Using standard supervised training, y is a one-hot encoded vector, with 1 in the position of the correct label, and 0 everywhere else. Therefore, when the correct class is $i = k$, Equation 3.18 simplifies to:

$$J(\hat{y}, k) = -\log \hat{y}_k. \tag{3.19}$$

Equation 3.19 contains the objective function typically differentiated during the training of a large neural network.

The output probabilities of a previously trained large network capture rich information not available in the original training set, which only contain input examples and the correct label for each input. For example, assume a classification dataset which includes cars, trucks, and other non-vehicle classes. During training, when learning instances of car classes, only a single correct label (y, which is one-hot encoded) will be used. Once trained, if presented with a previously unseen photo of a car, the car and truck class probabilities will most likely both contain significant information regarding the correct class, whereas the potato class probability would not contain as much information. Model distillation uses all of this information.

There are various techniques to implement distillation. Initially, assume a large network has been trained to high performance, and a smaller network is to be trained with distillation. Additionally, assume we do not have access to the correct training labels. In this case, we may input random images into the large network and use *all* of its prediction probabilities $\hat{y}$ as a *soft target* for the distilled network's output $\tilde{y}$:

$$J(\tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \hat{y}_i \log \tilde{y}_i. \tag{3.20}$$

This is similar to Equation 3.18, except $y = \hat{y}$ and we have class probabilities for each entry in $\hat{y}$, so it does not simplify to Equation 3.19.

If training labels are also available, the objective function can be improved by summing Equations 3.18 and 3.20, giving:

$$J(y, \tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \alpha \hat{y}_i \log \tilde{y}_i + \beta y_i \log \tilde{y}_i, \tag{3.21}$$

where α is a hyperparameter which sets the relative importance for matching soft targets, and β sets the relative performance for selecting the correct class. In practice [42] found that α should be higher than β.

In addition to hyperparameters α and β, [42] also found that the temperature in Equation 3.17 impacts distillation performance. Higher temperatures make "softer" probability distributions. To understand why this may be important, consider the logits $[1, 2, 10]$, which have a softmax with $T = 1$ of $[1 \times 10^{-4}, 3 \times 10^{-4}, 9.995 \times 10^{-1}]$. The small probabilities slow down learning during backpropagation. However, when $T = 10$ the softmax becomes $[.22, .24, .54]$, which has ranges that will cause learning to occur more quickly with backpropagation. It can therefore be useful to use high T values for the softmax of both the large network and distilled network during the distillation phase[6]. After distillation is finished, T may be reset to 1.

Distillation is effective for transferring information from trained large networks to untrained smaller networks. In [42], a large DNN was trained to classify MNIST, resulting in 67 test errors. A smaller network, trained and tested with the sames sets as the larger network, resulted in 146 errors. However, when the smaller network was trained with distillation, it only made 74 test errors.

[6]The softmax layer is at the output and has no trainable parameters. It can therefore be replaced in the larger network with a separate temperature, with no need for retraining.

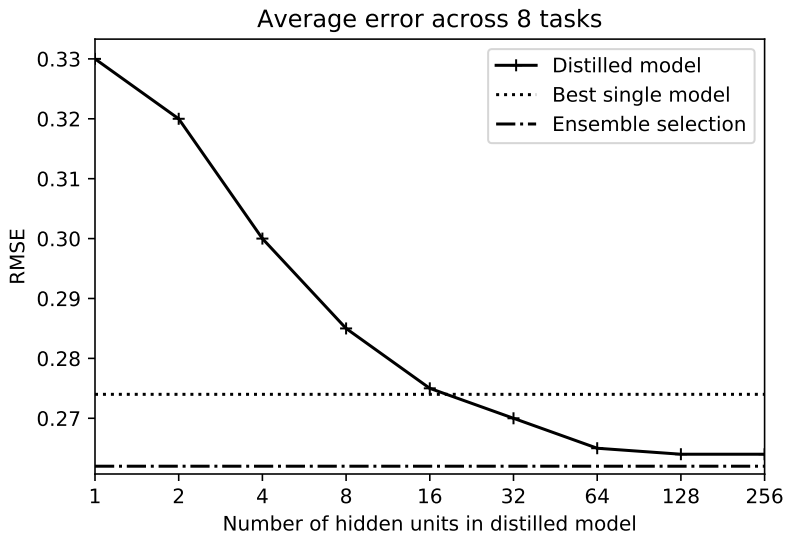


Figure 3.5: An ensemble of models was trained for eight classification tasks. Distillation was then used to train a neural network to behave like each ensemble. The plot compares average performance between the ensemble of classifiers, the best individual classifier in each ensemble, and the distilled classifiers. Once the distilled classifier has enough capacity, its average approaches the ensemble average [41].

Thus far we have discussed how to distill a DNN into a smaller network. Similar methods may be used to distill an ensemble of classifiers. In [41], eight binary classification problems were solved by an ensemble of methods, and then a neural network was trained by distillation to capture the behavior of the ensemble. The average performance of the small distilled model is given in Fig. 3.5. It can be seen that the average performance of the distilled model was similar to a giant ensemble prediction derived from SVMs, bagged trees, boosted trees, boosted stumps, simple decision trees, random forests, neural nets, logistic regression, k-nearest neighbor, and naive Bayes.

A smaller distilled model is obviously guaranteed to be more efficient than a large DNN or ensemble of models, and the distillation approaches presented in this section are a promising avenue to achieving adequate performance, given hard resource constraints. The steps for distillation are summarized in Algorithm 6.

Algorithm 6: Distillation

Require: Inputs $\mathcal{I}$, optional targets y, previously *trained* high performance network $\langle \mathcal{W}, \mathcal{O} \rangle_{large}$, *untrained* distilled network $\langle \mathcal{W}, \mathcal{O} \rangle_{dist}$

Ensure: Trained distilled network $\langle \mathcal{W}, \mathcal{O} \rangle_{dist}$

1. Inference:

$\hat{y} \leftarrow$ output probabilities of $\langle \mathcal{I}, \mathcal{W}, \mathcal{O} \rangle_{large}$

$\tilde{y} \leftarrow$ output probabilities of $\langle \mathcal{I}, \mathcal{W}, \mathcal{O} \rangle_{dist}$

2. Calculate loss:

if targets y are available

$J(y, \tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \hat{y}_i \log \tilde{y}_i + y_i \log \tilde{y}_i$

else

$J(\tilde{y}, \hat{y}) = -\sum_{i=1}^{|C|} \hat{y}_i \log \tilde{y}_i$

3. Update distilled model parameters:

$\mathcal{W}_{dist} \leftarrow \mathcal{W}_{dist} - \eta \nabla_{\mathcal{W}_{dist}} J$

4. Optionally repeat:

Repeat steps 2 and 3 until objective function is optimized.

3.6 Filter Decomposition

AlexNet introduced the first popular high-performance convolutional neural network (CNN) architecture, which has since been widely adopted and modified [43]. The AlexNet architecture won fame by winning the 2012 ImageNet Challenge, which required classification across 1,000 categories. AlexNet uses five convolutional layers, three fully-connected layers, and other less computationally expensive layers. Modern CNNs use even more convolutional layers, for example, Google's GoogLeNet-v1 CNN architecture uses 57 convolutional layers, but only one fully-connected layer.

Fully connected-layers are expensive from a bandwidth perspective, because they perform only one multiply-accumulate operation (MAC) per byte transferred over memory. Convolutional layers, however, are efficient from a bandwidth perspective, but they are expensive computationally. For example, AlexNet's three fully-connected layers require 58.6M MAC operations and 58.6M parameters, whereas AlexNet's six convolutional layers require 666M MAC operations and only 2.3M parameters. The total cost of a fully-

connected layer or convolutional layer is the total number of MACs plus total number of bytes required for the layer[7]. The choices of filter sizes in convolutional layers has a large impact on the bandwidth and computational costs of a CNN. In this section we analyze the bandwidth and computational impacts of different convolutional filter designs.

We loosely base our discussion on AlexNet, because it is well understood and the foundation of modern CNN designs. AlexNet convolutional layers use three filter shapes: 11×11, 5×5, or 3×3 and four channel depths: 96, 256, or 384. The shape of convolution filters has a significant impact on computational cost. To calculate the MAC cost for layer l's convolution operations, we first recall the notation introduced in Section 3.1, where layer l's filter tensor is denoted $\mathcal{W}_l \in \mathbb{R}^{c_{in} \times w \times h \times c_{out}}$ and layer l's input tensor is denoted $\mathcal{I}_l \in \mathbb{R}^{c_{in} \times x \times y}$. When assuming valid padding and stride of one, the number of MAC operations in a convolutional layer is found by[8]:

$$\texttt{MAC cost} = \texttt{cardinality}(\mathcal{I}_l) \times \frac{\texttt{cardinality}(\mathcal{W}_l)}{c_{in} \text{ from } \mathcal{W}_l}, \tag{3.22}$$

where `cardinality()` returns the number of elements in the input tensor. The bandwidth required for a filter, assuming 32-bit floating-point parameters, is calculated as:

$$\texttt{Byte cost} = c_{in} \times w \times h \times c_{out} \times 4 \text{ bytes}. \tag{3.23}$$

The goal of efficient CNN design is to obtain the highest classification performance, using the fewest number of MACs and parameters. Therefore from an efficiency perspective, the cost of CNN inference is:

$$\texttt{COST}() = c_1\texttt{MAC cost} + c_2\texttt{Byte cost} + c_3\texttt{CNN errors}, \tag{3.24}$$

[7]First-order estimates of power costs can be calculated using Table 3.1.

[8]Our calculations assume there is no pooling layer after convolution, which is now commonly the case.

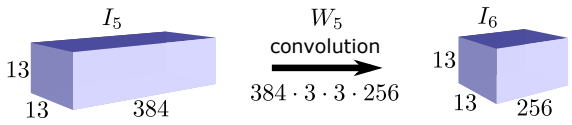


Figure 3.6: Example calculation of `MAC cost` of the fifth convolution in AlexNet. For intuition in understanding `MAC cost`, consider that each point in $\mathcal{I}_6$ is the result of applying a $384 \times 3 \times 3$ filter tensor to $\mathcal{I}_5$. Therefore the total number of MACs needed to calculate $\mathcal{I}_6$ is $256 \times 13 \times 13 \times 384 \times 3 \times 3$. This is a different perspective on the calculation than given in the main text.

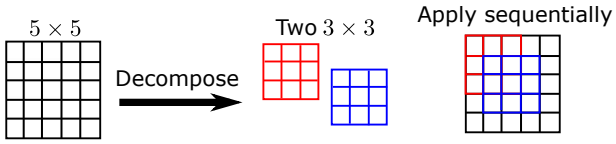


Figure 3.7: A "large" convolutional filter may be separated into two smaller filters, which retain the feature detection capabilities of the larger filter. The outputs of the smaller filters are summed. This approach is used to reduce the number of bytes required to represent filters and to reduce the number of MAC operations.

where the coefficients c depend on the priorities and budget of the CNN's designer.

To better understand Equation 3.22, consider the calculation of the number of MACs in the fifth convolutional layer of AlexNet, illustrated in Fig. 3.6. In this case $\texttt{cardinality}(\mathcal{I}_5) = 384 \times 13 \times 13$ and $\texttt{cardinality}(\mathcal{W}_5) = 384 \times 3 \times 3 \times 256$. So the the total number of MAC operations for $\mathcal{I}_5 * \mathcal{W}_5$ is $384 \times 13 \times 13 \times 3 \times 3 \times 256 = 150\text{M}$. Additionally, the size of $\mathcal{W}_5$ is $384 \times 3 \times 3 \times 256 = 885\text{k}$ parameters.

As another example, assume that instead of 3×3 filters, 5×5 filters were used in AlexNet's fifth convolutional layer. 5×5 filters cause the number of MAC operations to increase to 415M and byte cost to increase to 2.5MB. A larger filter can capture more detail, and suppose that switching to a 5×5 filter increased classification accuracy, but caused the total cost to exceed the time and energy budget allotted to the CNN. Perhaps surprisingly, there are techniques to extract the benefit of 5×5 filters without using 5×5 filters.

The concept of filter decomposition was introduced in [44], where two smaller filters

$\mathcal{W}_{l,1}$, $\mathcal{W}_{l,2}$ were applied to the input tensor $\mathcal{I}_l$ and then added (prior to the nonlinearity), giving $\mathcal{I}_{l+1} = \mathcal{I}_l * \mathcal{W}_{l,1} + \mathcal{I}_l * \mathcal{W}_{l,2}$. As shown in Fig. 3.7, instead of using a single 5×5 filter tensor in the previous case, two 3×3 tensors can be used. Specifically, instead of performing $256 \times 13 \times 13 \times 5 \times 5 \times 384 = 415$M MAC operations (requiring 2.5M parameters), $256 \times 13 \times 13 \times 3 \times 1 \times 384 \times 2 = 100$ MACs are performed (requiring 295k parameters). Similarly, a 5×1 and 1×5 filter may be used, requiring $256 \times 13 \times 13 \times 5 \times 1 \times 384 \times 2 = 166$M MAC operations (requiring 983k parameters), which is close to the original 150M MACs and 885k parameters required when using a single 3×3 filter tensor.

Going even further, [45] introduced 1×1 convolutions, which are used to create *bottleneck layers*, because they can shrink an input tensor. 1×1 filters detect correlation between corresponding parameters in each channel, which may be seen when considering their full notation: $c_{in} \times 1 \times 1 \times c_{out}$. For example, suppose we are given input $\mathcal{I}_l \in \mathbb{R}^{c_{in} \times x \times y}$, then a filter $\mathcal{W}_l \in \mathbb{R}^{c_{in} \times 1 \times 1 \times c_{out}}$ may be chosen such that $c_{out} \ll c_{in}$. Convolving $\mathcal{W}_l$ with $\mathcal{I}_l$ gives $\mathcal{I}_{l+1} \in \mathbb{R}^{c_{out} \times x \times y}$. The information from $\mathcal{I}_l$ is not lost, even though $\mathcal{I}_{l+1}$ now has fewer channels than $\mathcal{I}_l$. 1×1 convolutions capture channel correlations, compared to larger filters which capture channel and spatial correlations.

Various filter schemes can be combined. For example, a 1×1 convolution may be followed by a 3×3 or 5×5 convolution. The goal here is to extract channel correlations using the 1×1 convolution and to extract spatial (and channel) correlations using the 3×3 or 5×5 filter. Going back to our original AlexNet example, we calculated the number of MACs used for the convolution of $\mathcal{I}_5$ and $\mathcal{W}_5$ as $256 \times 13 \times 13 \times 3 \times 3 \times 384 = 150$M MACs and 885k parameters. We can reduce this by picking a smaller c_{out} size for $\mathcal{W}'_5$, e.g. 64, giving $256 \times 13 \times 13 \times 1 \times 1 \times 64 = 2.8$M MACs and 16k parameters. We may then add another convolution layer, using a $384 \times 3 \times 3$ filter $\mathcal{W}'_6$ and return to the original shape of $\mathcal{I}_6$ using $384 \times 13 \times 13 \times 3 \times 3 \times 64 = 37$M MACs and 221k parameters. We

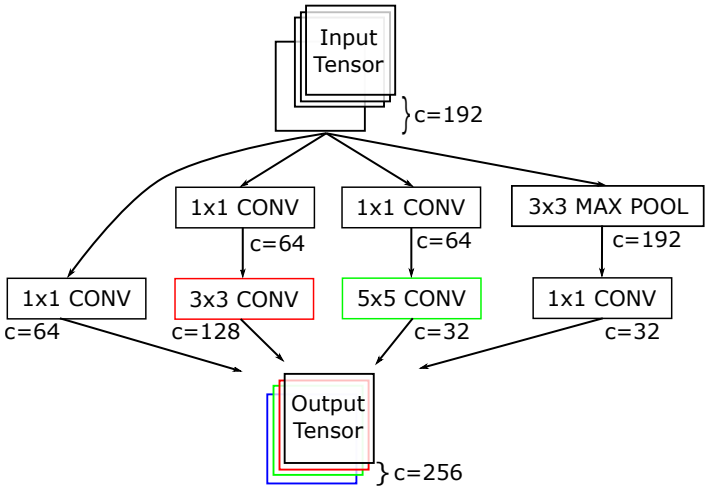


Figure 3.8: Diagram of an Inception module. Layer inputs are passed through separate 1×1 bottleneck layers, then through standard convolutional layers. This technique allows for the use of different filter sizes, without paying the computational or bandwidth cost of normal convolutional layer implementations [46].

now have extracted both channel and spatial correlations, using 1×1 and 3×3 filters and a total of 39.8M MACs and 237k parameters – much fewer than the original example which used 150M MACs and 885k parameters. Bottleneck layers followed by convolution has proven to be an effective way to increase efficiency without sacrificing accuracy.

Filter decomposition represents a fundamentally different way to improve DNN inference efficiency, compared to earlier sections. Specifically, by making careful architectural choices, high performance can be maintained and fewer parameters and MAC operations can be used. The methods introduced here may also be combined. For example, Inception is a modern CNN architecture, which combines bottleneck layers and various filter shapes to capture the benefits of every possible combination [46]. Fig. 3.8 illustrates an Inception "module", which combines many convolutional layers and outputs each combination as stacks of sub-channels. Without the 1×1 bottleneck layers, such an architecture would be much more expensive.

3.7 Summary

This chapter introduced various mathematical and algorithmic methods for optimized DNN inference:

- Eliminating "small" parameters via pruning, which reduces the required number of multiply-accumulate operations.
- Quantization, or reducing the precision, of layer inputs and/or parameters to reduce computation and data transfer costs.
- Sharing parameters between layer units and therefore enabling data transmission compression.
- Training small models to mimic larger models by distilling the information from the larger models into the smaller models.
- Separating larger convolutional filters into smaller filters, while retaining the performance of the larger filters.

These optimization methods may be used individually or may be combined for greater optimization. Note that the methods are not equivalent and should be expected to affect performance metrics in different ways.

Unfortunately most of the optimizations introduced here will result in an accuracy loss when compared to a high-performance model which was designed with no regard to computational efficiency. The trade-off between accuracy, redundancy, and precision is depicted in Figure 3.9 [47]. In general, one may expect to obtain high accuracy when using high-precision (e.g. floating-point) arithmetic (Pt. 2 in Figure 3.9), and lower accuracy when using low-precision arithmetic (Pt. 4). But low-precision arithmetic may be offset with redundancy (e.g. larger models) (Pt. 1). Likewise the errors caused

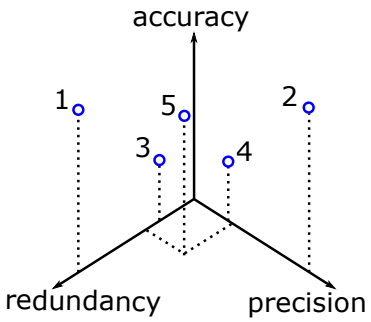


Figure 3.9: The notional trade-off between accuracy, redundancy, and precision. In general, one may prioritize any two at the expense of the third [47]. There is currently no formal proof for this plot, but most of the optimization papers referenced in this chapter report metrics across the different axes and seem to generally follow the trend of this plot.

by using low-redundancy (few parameters) models may be offset, to some extent, with high-precision arithmetic.

Ultimately, it is the DNN architect's task to find a design which achieves minimum acceptable performance, given a particular resource (e.g. latency, silicon area, power) budget. The methods introduced in this chapter facilitate this task.

Chapter 4

RAPDARTS: Resource-Aware Progressive Differentiable Architecture Search

4.1 Introduction

The optimal design of a neural architecture depends on 1) the target dataset, 2) the set of available primitive operations, 3) how the primitive operations are composed into a neural architecture and optimized, and 4) resource constraints like hardware cost, minimum accuracy, or maximum latency. In this chapter, we assume the target dataset has been provided, and we provide guidelines and analysis for searching for neural architectures under one or more hardware resource constraints.

Convolutional layers and fully-connected layers are parameter-heavy operations. Those, along with other lighter primitive operations, like pooling layers or batch normalization, may be composed into an endless variety of neural architectures. But what is the optimal neural architecture for a given dataset? There is no existing closed-form solution to that

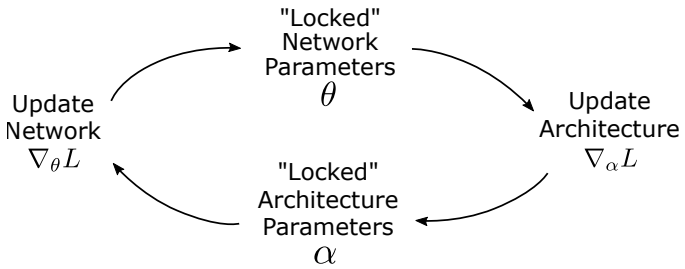


Figure 4.1: Gradient-based Neural Architecture Search (GBNAS) methods maintain two sets of parameters. *Neural network parameters* are represented by $\boldsymbol{\theta}$ and *architecture parameters* are represented by $\boldsymbol{\alpha}$. GBNAS algorithms leverage differentiable functions, parameterized by architecture parameters, to design deep neural networks, which are parameterized by network parameters. First-order optimization alternates between "locking" one set of parameters and updating the other.

question.

Historically, the highest performing neural architectures have been found by applying heuristics and a large amount of compute. Some well known examples of modern hand-crafted architectures include AlexNet [43], VGG16 [48], ResNet [49], and the Inception series [46, 50, 51]. None of these examples consider hardware, and they pursue classification performance at all cost.

Neural Architecture Search (NAS) methods automate strategies for discovery of high performing neural architectures. A reinforcement learning-based approach was the first post-AlexNet NAS method with state-of-the-art performance on CIFAR-10 [7, 8]. The RL approach was quickly followed by a high performance Evolutionary Strategy (ES) based method [9]. While both the RL and ES methods discovered high performance architectures, their use came at the cost of thousands of GPU hours.

Gradient-based NAS (GBNAS) methods have the benefit of being directly optimized through gradient descent and consequently complete the search faster than other NAS methods. The basic idea of GBNAS is given in Fig. 4.1. The search process alternates between temporarily fixing one set of parameters, i.e. assuming they are constants, and updating the other set of parameters. This approach has no convergence guarantees, but

it works well in practice.

Because neural models are now widely deployed on systems like edge devices, in cars, and running in servers, available hardware resources also have an impact on what may be considered an "optimal" neural architecture design. Hardware resource constraints are often summarized as size, weight, and power (SWaP). Resource constraints could also include maximum latency, minimum throughput, or a manufacturing budget which will determine if a custom ASIC is an option, if a COTS device is sufficient, or if something semi-custom, like an FPGA, is an option. For example, during the design of Google's TPUv1, architects were given a budget of 7 ms per inference (including server communication time) for user-facing workloads [37].

Recent efforts described below implement NAS strategies incorporating hardware resource constraints into the search. GBNAS methods capture hardware resource constraints within a differentiable loss function. This approach enables the architecture search to yield network architectures biased toward satisfying resource constraints.

In this chapter we have modified P-DARTS [52], which in-turn is based on another popular gradient-based NAS algorithm, DARTS [10], to support resource costs. We use our modified GBNAS algorithm to search for many neural architectures under various resource consumption penalties. We then use our results and observations to answer the following questions:

- What is the computational cost of searching for satisficing architectures?
- What heuristics can be used to guide the search and training process to reduce compute time?
- How reproducible are search results under random initial conditions?

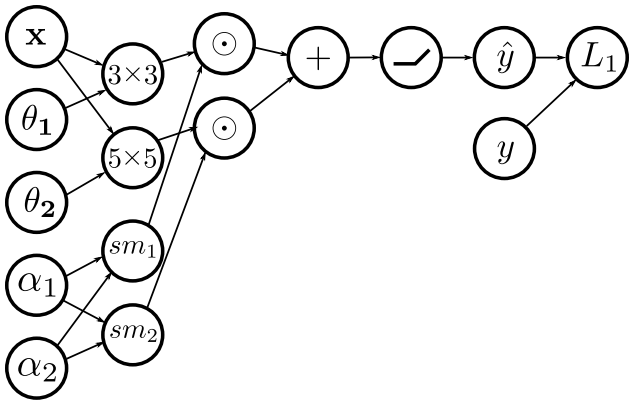


Figure 4.2: The function of DARTS architecture parameters is to scale the output of primitive operations. In this illustration the primitive operations include 3×3 and 5×5 convolutional filters parameterized by tensors $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ respectively. The output feature maps of the primitive operations are element-wise scaled $\odot$ by the softmax (sm) of architecture parameters α_1 and α_2. The scaled output feature maps are then added, thereby creating a *mixed operation*. This notional illustration shows a network with only two primitive operations, followed by a nonlinearity, producing an output prediction $\hat{y}$. In practice, there may be many mixed operations, each containing many primitive operations, forming a deep network.

4.2 Related Work

The first competitive NAS approach applied to modern image classification tasks was based on reinforcement learning (RL) [7]. In this chapter, an LSTM-based RL agent was trained to output primitive operations which were then chained together into a directed acyclic graph. After training and evaluating the graph, the agent was then encouraged or discouraged, via a positive or negative reward derived from classification accuracy, to generate similar graphs in the future or to explore and make new graphs.

The reinforcement learning NAS approach worked well and was able to achieve high accuracy, but at unheard of computational expense. It required 3,150 GPU-days to discover one of their published architectures.

Related approaches to sampling neural architectures include Markov chain Monte Carlo methods [53], evolutionary strategies [54], and genetic algorithms [55]. Similar to RL approaches, all of these optimization methods generate populations of neural

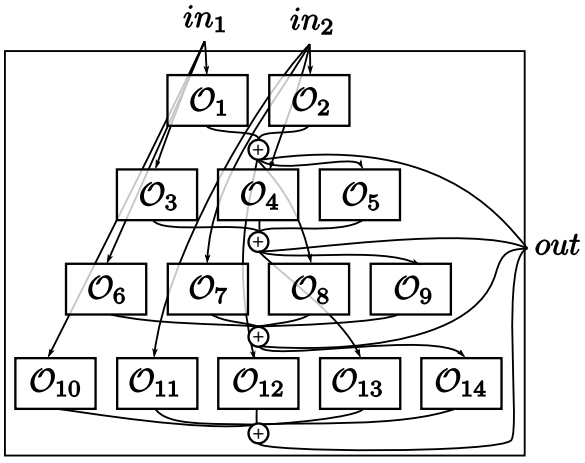


Figure 4.3: The DARTS cell architecture has 14 mixed operations (represented by $\mathcal{O}_i$) distributed among four steps with skip-connections between each step. At each step, the outputs of the mixed operations are element-wise added. The sum is then passed as an input to a mixed operation in the next step. All element-wise sums are concatenated as the cell output and fed forward to the next cell in the network.

architectures. The populations are then trained and a fitness value is derived from the classifier's final test performance. The fitness value is used to encourage or discourage the design of the next population of architectures.

Reinforcement learning, Markov chain Monte Carlo methods, evolutionary strategies, and genetic algorithms discover high-performance architectures, but they are incredibly expensive. These methods often require 100× to 1000× more compute than gradient-based methods [56].

Gradient-based neural architecture search has recently become popular because of its efficiency [10, 57, 58, 52]. GBNAS methods maintain two sets of parameters: *network parameters* $\boldsymbol{\theta}$ and architecture parameter $\boldsymbol{\alpha}$. Previous GBNAS methods have introduced various methods to optimize and use the two parameter sets. In the simplest case, optimization is achieved by optimizing one set of parameters and then the other. This first-order optimization approach is illustrated in Fig. 4.1.

Differentiable Architecture Search (DARTS) is a GBNAS technique that uses *mixed operations* to compute multiple primitive operations in parallel, followed by element-wise

summation [10]. The mixed operations are scaled by architecture parameters prior to summation. For example, as illustrated in Fig. 4.2, a 3×3 convolutional filter and a 5×5 convolutional filter can be designed such that both receive the same input feature map and both generate additively conformable output feature maps.

Extending this technique, DARTS composes 14 mixed operations into a *cell*. Eight cells are then chained to create the network. Each cell has the same connectivity and architecture parameters ($\boldsymbol{\alpha}$) for mixed operations, but the network parameters ($\boldsymbol{\theta}$) are learned independently in each primitive operation and in each cell. An illustration of the DARTS cell connectivity is given in Fig. 4.3.

DARTS has a limitation which requires the entire neural network (i.e. all cells and all mixed operations) to fit in GPU memory. This limits the depth of the neural network as well as the batch size during training. Progressive Differentiable Architecture Search (P-DARTS) mitigates the memory limitation of DARTS by 1) gradual growth in the depth of the neural network, and simultaneously 2) gradual reduction in number of primitive operations per mixed operation, thus reducing model size [52].

ProxylessNAS also extended DARTS [58]. ProxylessNAS treats the architecture parameters of each mixed operation as a probability distribution. ProxylessNAS stores a large over-parameterized network in system memory, because the network is too large to fit on a GPU. During evaluation, a subnetwork is sampled and transferred to the GPU for evaluation. Gradients are calculated and used to update the shared-weights of the over-parameterized network.

Addressing the need to search for architectures which not only strive for high accuracy, but also meet additional performance constraints, hardware-aware NAS techniques have been pursued. ProxylessNAS is particularly relevant for hardware-aware GBNAS, because it formalizes the approach to incorporating resource costs during the search. In the context of classification, ProxylessNAS creates a loss function that incorporates both

a cross-entropy loss for the classification accuracy as well as a resource loss for latency.

In this chapter we augment P-DARTS with a ProxylessNAS-style resource loss and analyze its impact on architectures discovered during the search phase.

4.3 Method

4.3.1 Resource-Aware Differentiable Neural Architecture Search

When training a convolutional neural network for classification, the goal is to obtain a model that best predicts labels from observations drawn from an underlying distribution of interest. Fitting a neural model to an underlying distribution is achieved by finding optimal network parameters $\boldsymbol{\theta}^*$ that minimize expected prediction error on an available dataset:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \big[J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) \big], \tag{4.1}$$

where J is the objective function, $\boldsymbol{x}$ are dataset observations, y are dataset labels, $\hat{p}_{data}$ is the empirical distribution, L is a prediction error loss function, and f is the neural network parameterized by $\boldsymbol{\theta}$.

Gradient-based NAS methods introduce another set of *architecture parameters* $\boldsymbol{\alpha}$, producing:

$$g(\boldsymbol{x};\boldsymbol{\theta},\boldsymbol{\alpha}). \tag{4.2}$$

We refer to g as a directed acyclic graph, or simply *graph*, to highlight that it is composed of a neural network whose control flow is modified by other non-network architecture parameters. Note the distinction between f used in Equation 4.1, which is only parameterized by network parameters, and g used in Equation 4.2, which is parameterized by both network and architecture parameters.

Architecture parameters, like network parameters, are scalar-valued tensors. Architecture parameters are used to control either the weight of primitive operations, as in [10, 52], or the probability primitive operations will take place, as in [59, 58]. In both cases, the scalar values are at least interpreted as one or more probability distributions through processing by the softmax function. In our case, the probability distribution is then used for evaluation of a mixed operation.

A mixed operation is illustrated in Fig. 4.2, and it is formalized as:

$$\mathcal{O}(\boldsymbol{x}) = \mathbb{E}\big[o(\boldsymbol{x})\big] \approx \sum_{i=1}^{N} \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)} o_i(\boldsymbol{x}) = \sum_{i=1}^{N} p_i o_i(\boldsymbol{x}), \tag{4.3}$$

where $o_i(\boldsymbol{x})$ is a primitive operation, and $\mathcal{O}(\boldsymbol{x})$ is equivalent to the expected value of the primitive operations. This formalism extends the mixed operation to the inclusion of N primitive operations that are evaluated in parallel and designed such that their outputs are additively conformable. In practice many mixed operations are used, with unique subsets of $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$ used for the calculation of each expected value, but we show only a single mixed operation here for clarity.

The inclusion of architecture parameters implies there are now two objective functions to be optimized:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L_1(g(\boldsymbol{x}, \boldsymbol{\alpha}; \boldsymbol{\theta}), y), \\ J(\boldsymbol{\alpha}) &= \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L_1(g(\boldsymbol{x}, \boldsymbol{\theta}; \boldsymbol{\alpha}), y). \end{aligned} \tag{4.4}$$

The graph evaluations in Equation 4.4 are now denoted $g(\boldsymbol{x}, \boldsymbol{\alpha}; \boldsymbol{\theta})$ and $g(\boldsymbol{x}, \boldsymbol{\theta}; \boldsymbol{\alpha})$. This notation highlights that in the case of $J(\boldsymbol{\theta})$ the graph is evaluated at input and architecture parameter constants $(\boldsymbol{x}, \boldsymbol{\alpha})$ and optimized using network parameters $\boldsymbol{\theta}$. In the second case of $J(\boldsymbol{\alpha})$ the graph is evaluated at input and network parameter constants

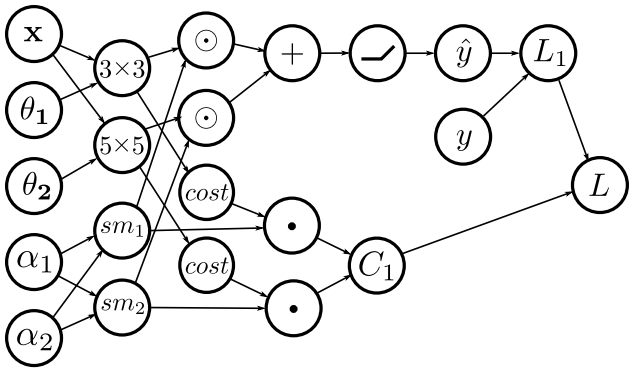


Figure 4.4: P-DARTS may be extended with the calculation of an expected resource cost (C_1) for each mixed operation. When the gradient of the expected resource cost is calculated, the more expensive primitive operations are penalized more heavily than the less expensive operations, but the penalty is balanced by how much the primitive operation contributes to classification accuracy.

$(\boldsymbol{x}, \boldsymbol{\theta})$ and optimized using architecture parameters $\boldsymbol{\alpha}$. Therefore the following bilevel optimization must be solved:

$$\begin{aligned}\boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} \left[J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L_1(g(\boldsymbol{x}, \boldsymbol{\alpha}^*; \boldsymbol{\theta}), y) \right], \\ \boldsymbol{\alpha}^* &= \arg\min_{\boldsymbol{\alpha}} \left[J(\boldsymbol{\alpha}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L_1(g(\boldsymbol{x}, \boldsymbol{\theta}^*; \boldsymbol{\alpha}), y) \right]. \end{aligned} \tag{4.5}$$

When using first-order differentiable methods, this bilevel optimization is solved by alternatingly "locking" one set of parameters and updating the other with gradient descent. Second-order optimization methods, which involve calculation of the Hessian, are also possible and slightly better in terms of accuracy, but this comes at significant computational cost. However, it is possible to approximate the second-order optimization with reduced computational cost [10].

Our method extends P-DARTS to discover neural architectures biased toward the satisfaction of resource constraints. We do this by including one or more "expected resource cost" loss terms. As mentioned previously, each of the primitive operations in a mixed operation is associated with a unique architecture parameter. P-DARTS uses

14 mixed operations in the search phase of cell architecture discovery, and there are eight primitive operations per mixed operation, so there are $14 \times 8 = 112$ architecture parameters total.

The expected value of a single mixed operation was given in Equation 4.3. We temporarily make index values of the mixed operation explicit here for clarity:

$$\mathcal{O}_k(\boldsymbol{x}_k) = \sum_{i=1}^{8} p_{k,i} \cdot o_{k,i}(\boldsymbol{x}_k), \tag{4.6}$$

where k is the mixed operation index. Note here that the probability distributions, $p_{k,i}$, are now tied to a particular mixed operation. This calculation is equivalent to the addition node in Fig. 4.2.

As introduced in ProxylessNAS, the probabilities used in the mixed operation calculation are also conducive to calculation of the expected value of various resource costs. For example, if there is a cost function that takes as input the description of each primitive operation (including the input feature map dimension information) and outputs a resource cost, it may be used for the calculation of an expected resource cost of the mixed operation:

$$\mathbb{E}\big[cost(\mathcal{O}_k(\boldsymbol{x}_k))\big] \approx \sum_{i=1}^{8} p_{k,i} \cdot cost(o_{k,i}(\boldsymbol{x}_k)). \tag{4.7}$$

The cost function may be an analytical function, e.g. number of bytes required by the model, or the cost function could be based on a simulation or a surrogate model trained from data collected from a physical device.

The expected cost of the mixed operation is differentiable with respect to the mixed operation's architecture parameters. Accordingly, the partial derivative of the expected

resource cost with respect to architecture parameter α_i is given as:

$$\begin{aligned}\frac{\partial \mathbb{E}\big[cost(\mathcal{O}(\boldsymbol{x}))\big]}{\partial \alpha_i} &\approx \frac{\partial \big[p_1 c_1 + p_2 c_2 + \cdots + p_8 c_8\big]}{\alpha_i}, \\ &= \sum_{l=1}^{8} \frac{\partial \Big[\frac{\exp(\alpha_l)}{\sum_j \exp(\alpha_j)} \cdot c_l\Big]}{\partial \alpha_i}, \\ &= \sum_{l=1}^{8} c_l p_l (\delta_{i,l} - p_i). \end{aligned} \tag{4.8}$$

where we have abbreviated $cost(o_i(\boldsymbol{x}))$ as c_i, $\delta_{i,l} = 1$ if i equals l and 0 otherwise, and we have dropped the mixed operation index k for brevity.

We denote the sum of expected mixed operation costs as:

$$C_m = \sum_{k=1}^{14} \mathbb{E}\big[cost_m(\mathcal{O}_k(\boldsymbol{x}_k))\big], \tag{4.9}$$

Note that unique m correspond to unique resource costs, e.g. C_1 could be the sum of expected mixed operation parameter sizes, and C_2 could be the sum of expected mixed operation latencies.

We denote the sum of the classification and resource losses as:

$$L = L_1 + \sum_{m=1}^{M} \lambda_m C_m, \tag{4.10}$$

where M is the number of resource costs to satisfy, and λ_m is the resource-cost hyper-parameter and controls how important the resource cost m is compared to accuracy as well as other resource costs.

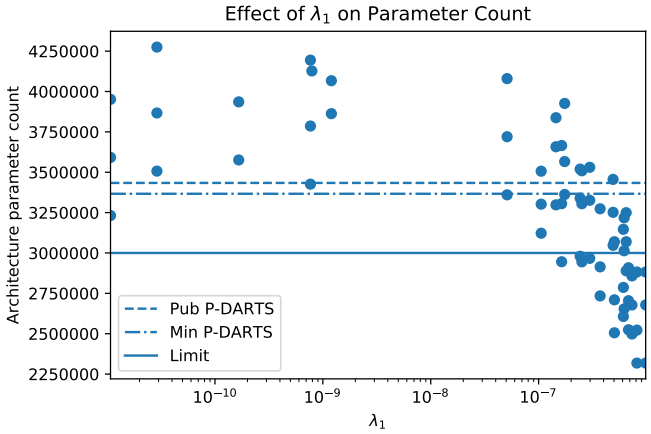


Figure 4.5: Coarse-search for resource expected parameter count hyperparameter λ_1. As λ_1 grows beyond 10^{-7}, RAPDARTS increasingly identifies architectures that require less than 3 M parameters. The publish P-DARTS architecture is marked with the dashed line. The minimum P-DARTS architecture found by us is marked with the dash-dot line. Our self-imposed budget is marked with the solid line.

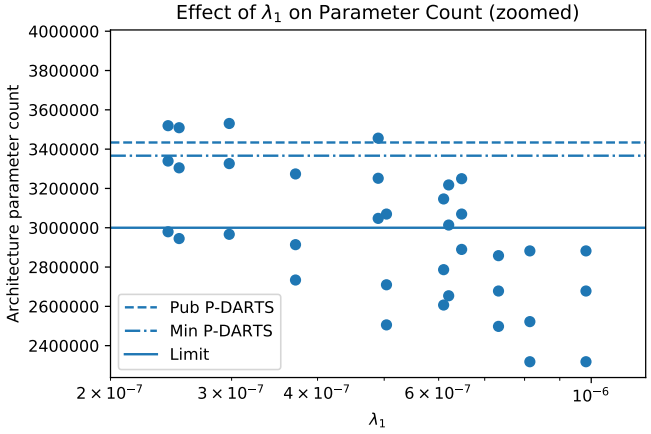


Figure 4.6: Fine-search focused $2 \times 10^{-7} < \lambda_1 < 10^{-6}$. At around $\lambda_1 = 6 \times 10^{-6}$ architectures are frequently generated which meet the 3 M parameter constraint.

The bilevel optimization in Equation 4.5 may now be slightly rewritten as:

$$\begin{aligned} \boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} \left[J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(g(\boldsymbol{x}, \boldsymbol{\alpha}^*; \boldsymbol{\theta}), y) \right], \\ \boldsymbol{\alpha}^* &= \arg\min_{\boldsymbol{\alpha}} \left[J(\boldsymbol{\alpha}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(g(\boldsymbol{x}, \boldsymbol{\theta}^*; \boldsymbol{\alpha}), y) \right], \end{aligned} \tag{4.11}$$

where only L_1 has been replaced by L. As before, this may be optimized using first or

second-order approaches. For intuition on the continued use of a single loss function L, consider Fig. 4.4. Under the assumption that a change in network parameters $\boldsymbol{\theta}$ creates no change in cost (given a fixed input feature map and primitive operation), the gradient of C_1 with respect to $\boldsymbol{\theta}$ is zero. On the other hand, a change in architecture parameters $\boldsymbol{\alpha}$ creates a change in both L_1 and C_1. So calculating the gradient of $L = L_1 + \lambda_1 C_1$ with respect to both $\boldsymbol{\theta}$ and $\boldsymbol{\alpha}$ results in the correct values.

Using the method above, we created *Resource-Aware P-DARTS* (RAPDARTS). Practically, the modification to P-DARTS requires the total expected resource cost be returned during the forward pass of an input tensor. To achieve this, during calculation of each mixed operation (Equation 4.6), we also calculate the expected resource cost (Equation 4.7). The expected cost for all mixed operations is accumulated and added to the classification loss (Equation 4.9). If multiple costs are required, e.g. model size and latency, each cost requires its own version of Equation 4.7, and must be accumulated individually from other costs.

4.4 Experiments and Results

We use RAPDARTS to search for CIFAR-10 neural architectures. We follow the architecture discovery algorithm of P-DARTS and search for cell architectures containing the same primitive operations as used by DARTS and P-DARTS, namely:

- Zero*
- Skip-Connect*
- Avg-Pool 3×3*
- Max-Pool 3×3*
- Separable 3×3 Conv.
- Separable 5×5 Conv.
- Dilated 3×3 Conv.
- Dilated 5×5 Conv.

All of the above primitive operations are standard convolutional layers except Zero which allows a cell to learn *not* to pass information. Skip-connect is a parameter-free operation which allows information to pass through the mixed operation without modification. Parameter-free primitive operations are marked with an asterisk.

In an effort to simulate a real-world constraint, we restrict ourselves such that discovered CIFAR-10 architectures must have less than 3×10^6 parameters. This constrained optimization problem may be captured as:

$$\begin{aligned} \underset{\boldsymbol{\theta},\boldsymbol{\alpha}}{\text{minimize}} \quad & L_1(g(\boldsymbol{x};\boldsymbol{\theta},\boldsymbol{\alpha}),y) \\ \text{subject to} \quad & \text{Parameter count} \leq 3 \times 10^6. \end{aligned} \tag{4.12}$$

We perform NAS adhering to this constraint using the RAPDARTS framework above.

For the purpose of baseline calculations, we first consider the unconstrained results from P-DARTS. The authors of P-DARTS provided a reference architecture discovered through their algorithm [60]. We trained and evaluated that architecture eight times using the latest version of the P-DARTS code [61]. We then used the results from the repeated training to obtain performance statistics of the published architecture.

The resulting trained models achieved $2.60 \pm .13\%$ error on the CIFAR-10 validation dataset. Additionally, the published P-DARTS architecture requires 3.4×10^6 parameters.

We then executed the P-DARTS architecture search code four times to test the ability to rediscover architectures with the performance of the published architecture. The four searches resulted in nine architectures. However, per the P-DARTS algorithm, we eliminated one architecture with more than two skip-connections in the *normal* cell (see P-DARTS paper for details on the two cell types).

None of the eight valid architectures were the same as the official P-DARTS CIFAR-10 architecture, but this is not surprising, given the size of the P-DARTS architecture search

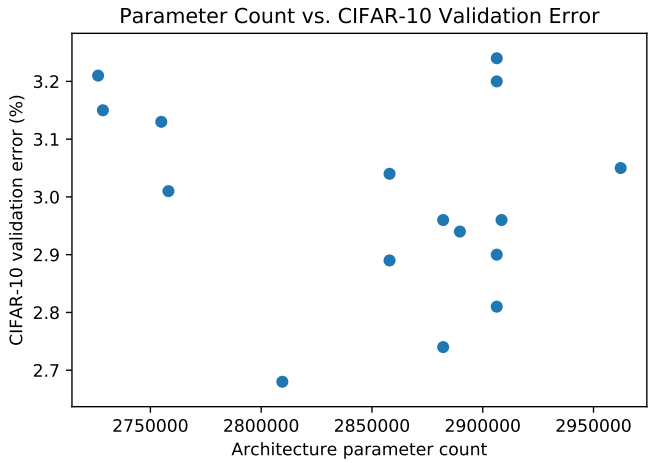


Figure 4.7: Relationship between RAPDARTS model size and trained validation error appears uncorrelated. Indicating that at this variation of model capacity, model size is not a predictor of final classifier performance.

space. Because of this, we compare our results to the statistics of various architectures discovered during our search, instead of the statistics of the single published architecture. The resulting trained models achieved $2.72 \pm .22\%$ error on CIFAR-10. The architectures required $3.9 \pm .3$ M parameters. The smallest P-DARTS model required 3.4 M parameters.

We now explore the impact of different hyperparameter values on the unconstrained multi-objective version of Equation 4.12:

$$L = L_1 + \lambda_1 C_1, \tag{4.13}$$

where C_1 is the sum of expected number of parameters in the model. As introduced in Equation 4.10, the λ_1 scalar is a hyperparameter which determines the relative importance of the resource cost explicitly and the relative importance of the accuracy of the network implicitly.

As stated in this section's introduction, our self-imposed resource budget is 3 M parameters. The default P-DARTS search does not generate models that small, however, by using RAPDARTS we are able to satisfy this constraint. To achieve this, we need to

	C10 Test Err (%)				
Architecture	**Best**	**Avg**	**Params (M)**	**Search Cost**	**Method**
AmoebaNet [62]	N/A	$\mathbf{2.55 \pm 0.05}$	2.8	3150	evolution
ASHA [63]	2.85	3.03 ± 0.13	2.2	9	random
DARTS [10]	2.94	N/A	2.9	.4	gradient
DSO-NAS [64]	N/A	2.84 ± 0.07	3.0	1	gradient
SNAS [57]	2.85	N/A	2.3	1.5	gradient
RAPDARTS (ours)	2.68	2.83 ± 0.05	2.8	12	gradient

Table 4.1: RAPDARTS CIFAR-10 error rate versus others for models with less than 3×10^6 parameters. We also include NAS results from randomly searched architectures [63]. Search Cost is measured in GPU-days. For RAPDARTS, search cost includes actual cost for all experiments for finding the 2.68% model. In total, the search and train phases required 26 GPU-days.

discover a λ_1 value to guide the architecture search. That is accomplished by finding a coarse range of suitable λ_1s and then identifying a refined λ_1.

The coarse λ_1 is identified by performing various architecture searches with λ_1s sampled randomly from a uniform distribution $\mathcal{U}([10^{-11}, 10^{-6}])$. Each search requires .3 GPU-days.

Results from the coarse-search are shown in Fig. 4.5. At approximately $\lambda_1 > 10^{-7}$, architectures begin to be generated which meet the 3×10^6 parameter count constraint. Parameter counts reduce dramatically as λ_1 approaches 10^{-6}, but we have observed that models with higher capacity tend to perform better than models with lower capacity, so it is unlikely that architectures derived from $\lambda_1 > 10^{-6}$ are preferred over those closer to the 3 M parameter threshold.

Fig. 4.6 "zooms in" on the previous figure, focusing on λ_1 sampled uniformly from $\mathcal{U}[(2 \times 10^{-7}, 10^{-6})]$. Near $\lambda_1 = 6 \times 10^{-7} \approx 1 \times 10^{-6.2}$, architectures are generated that often require less than 3 M parameters.

One final search is then performed on λ_1 sampled uniformly from $U([10^{-6.24}, 10^{-6.2}])$. This test resulted in 48 valid architectures with resulting models between 2.1 M and 2.96 M parameters. We then trained the 16 largest resulting architectures. The resulting best

model achieved 2.68% CIFAR-10 validation error and required 2.8 M parameters. The results for all 16 trained models are plotted in Fig. 4.7. As can be seen, there is no linear relationship at this scale between parameter count and CIFAR-10 accuracy. For statistical confidence, we retrained the best model eight times with different seeds and obtained $2.83\% \pm .05$ validation error.

The discovered cells corresponding to the 2.68% CIFAR-10 validation are shown in Fig. 4.8. The DARTS-based algorithms use two cell types: a "normal" cell, which maintains input and output feature map dimensionality, and a "reduce" cell, which decrease the output feature maps dimensionality.

The cell architectures discovered by RAPDARTS are noteworthy in several respects. First, the normal cell has discovered a "deep" design, similar to that discovered by P-DARTS, but only light-weight convolutional operations are used. Second, all pooling operations have been moved to the reduce cells.

Table 4.1 compares the RAPDARTS architecture with the performance of recent architectures with parameter counts less than 3 M. RAPDARTS competes favorably with the others.

We report the actual number of hours spent searching for our winning architecture, not merely the search time for a single architecture. Including both the coarse and fine-search phases, 40 different λ_1 values were used. This took a total of 12 GPU-days to compute.

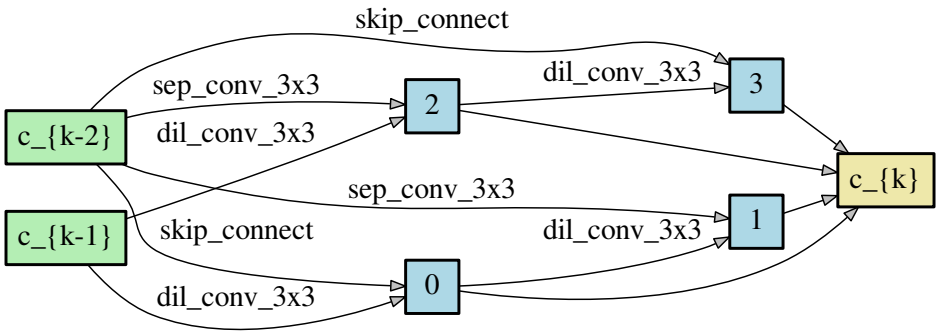
We trained 16 of the fine-search phase models to completion. Each model required less than 20 hours to train, so the 16 fine-search models took less than 14 GPU-days total to train. All experiments were performed using an NVIDIA V100 GPU.

4.5 Summary

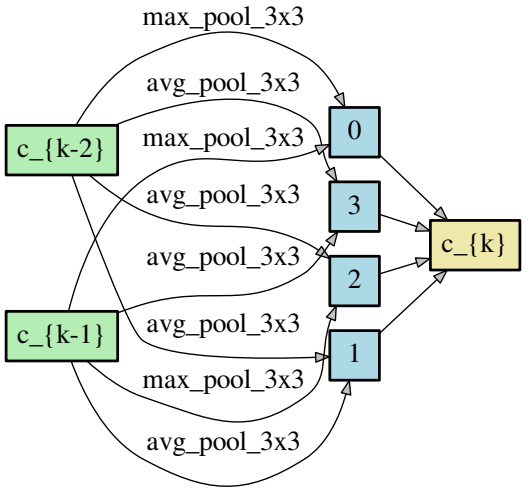
Classification accuracy achieved by neural architecture search methods now surpass hand-designed neural models. First-generation NAS methods include those based on evolutionary search and reinforcement learning. Second generation NAS methods use gradient-based optimization. In this chapter we present RAPDARTS, which augments a popular gradient-based NAS method with the ability to target neural architectures meeting specified resource constraints. We use RAPDARTS to identify a neural architecture achieving 2.68% test error on CIFAR-10. This is competitive with other existing results for models with less than 3 M parameters.

We believe third-generation methods will be gradient-based and attempt to make more aspects of the search differentiable. For example, the P-DARTS (and RAPDARTS) search begins with five cells, then grows the search network to 11 cells, and finally 17 cells. At the same time, as the network grows, less important primitive operations are dropped. The "gradual" adjustments introduced by this technique enable architecture parameters learned by gradient-descent in one phase to be useful in another. It would be preferable to make these changes even more gradually. We leave that for future work.

In conclusion, we have presented an example that optimizes two objectives: minimizing accuracy loss while keeping the number of model parameters below a resource constraint threshold. A limitation of our method is that the number of parameters required by our discovered models may not optimize other constraints, e.g. minimum latency. To address this concern, future work will focus on multiple resource constraints guided by more hardware-specific costs.



(a) Normal Cell



(b) Reduce Cell

Figure 4.8: Cells found by RAPDARTS achieving 2.68% CIFAR-10 validation error. All primitive operations are low-cost operations.

Chapter 5

Impacts of Mathematical Optimizations on Reinforcement Learning Policy Performance

5.1 Introduction

In this chapter we study the effects of adapting pruning, quantization, and compression methods to policies trained using the Vanilla Policy Gradient (VPG) method. This section introduces the primitive optimizations in the context of reinforcement learning. The following section presents the results of their application to the Vanilla Policy Gradient method.

5.1.1 Quantization

In 2015, BinaryConnect (BC) [38] was an early DNN quantization method, and it exemplifies the field's approach to quantization. During forward-propagation, BC quantizes full-precision DNN parameters to $\{-1, 1\}$, using the sign function:

$$\theta_b = \begin{cases} +1 & \text{if } \theta \geq 0, \\ -1 & \text{else.} \end{cases} \tag{5.1}$$

Equation 5.1 discards real-valued information, but, in doing so, it also eliminates the need for floating-point MACs during forward-propagation. Instead, signed floating-point addition may be used for neuron pre-activation calculations. During backpropagation, the error caused by quantization is used to update the real-valued θs. From a hardware perspective, when configured for AlexNet, memory overhead is 32× less when using BC-derived parameters. However, there is a performance loss when using quantization; with the AlexNet topology, BinaryConnect achieves 61% top-5 accuracy on ImageNet, compared to 80.2% accuracy when using the same DNN topology and 32-bit full-precision accuracy [33].

Applied to RL, BinaryConnect may be used with Vanilla Policy Gradient. VPG minimizes the cost[1]:

$$C = -\frac{1}{T}\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t)\hat{A}_t \tag{5.2}$$

where $\hat{A}_t$ is the advantage at time t. Optimal calculation of $\hat{A}_t$ is a focus of RL research, but VPG sets A_t equal to the expected sum of trajectory rewards. The cost function in Eq. 5.2 can be combined with the BinaryConnect optimization to create the BinaryConnect+VPG method as given in Algorithm 7.

In addition to BinaryConnect, we consider BinaryNet [40], which operates similarly, with the addition that activations are also binarized. When using BinaryNet, the activation inputs are summed, as with BinaryConnect, and then the resulting sum is converted to $[-1, 1]$ using the sign function. This optimization eliminates all full-precision calculations and replaces them with signed integer calculations. As with BinaryConnect,

[1]Minimizing cost and maximizing reward are equivalent, if cost equals the negative of reward.

Algorithm 7: BinaryConnect+VPG

Require: A state observation, selected action, advantage, previous parameters θ_{t-1} (parameters) and b_{t-1} (biases), and learning rate η.

Ensure: Updated $\{-1, 1\}$-valued parameters θ_t and real-valued bias b_t.

1. Forward propagation:

$\theta_b \leftarrow \text{binarize}(\theta_{t-1})$

For $k = 1$ to $L - 1$, compute activation a_k, knowing a_{k-1}, θ_b and b_{t-1}

Compute output probability of selected action using softmax

2. Backward propagation:

Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$

For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and θ_b

3. Parameter update:

Compute $\frac{\partial C}{\partial \theta_b}$ and $\frac{\partial C}{\partial b_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and a_{k-1}

$\theta_t \rightarrow \text{clip}(\theta_{t-1} - \eta \frac{\partial C}{\partial \theta_b})$

$b_t \rightarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$

BinaryNet requires full-precision gradient updates during training. As an example of the impact BinaryNet quantization has on performance, it achieves 50.42% top-5 accuracy on AlexNet [33]. BinaryNet may also be combined with VPG.

5.1.2 Compression

Many DNN models require over 500MB of model parameters to be transferred from memory to the accelerator [65]. Compression methods reduce the amount of data to be transferred, thereby reducing the most expensive power operation.

We now consider a compression method that clusters parameters in each layer [35]. First, a full-precision version of the network is trained using VPG. Next, the n b-bit parameters of each layer are clustered into k groups using an arbitrary clustering algorithm, e.g. K-Means. Finally, the network is fine-tuned. During the fine-tuning stage, in forward-propagation, each cluster is locked to the same value. During backpropagation, the individual gradients for each cluster are summed by their respective group. The sum of the gradients are then applied to the appropriate cluster parameters.

Algorithm 8: Compression+VPG

Require: Full-precision policy network parameterized by θ_{all}, learning rate η, and number of clusters k.

Ensure: Fine-tuned network incorporating real-valued clustered parameters θ_k for each layer.

1. Full-precision training:
For each state observation perform full-precision (FP32) network evaluation. Select actions from resulting output distributions.
At episode end, update all FP32 parameters θ_{all} using standard VPG and η. Repeat until maximum performance is achieved.

2. Compression:
For each layer, cluster parameters into k groups using K-Means algorithm, resulting in θ_k.

3. Fine-tuning:
For each state observation perform network evaluation using θ_k. Select actions from resulting output distribution.
Calculate gradient as usual.
Perform modified backpropagation: in each layer, sum partial derivatives associated with respective cluster.
Update θ_k using summed partial derivatives and η. Repeat Step 3 until maximum performance is achieved.

After training, when evaluating each layer, only the cluster indices must be transmitted, resulting in a compression rate of:

$$r = \frac{nb}{n\log_2(k) + kb}. \tag{5.3}$$

Complete steps for combining VPG with compression are provided in Algorithm 8.

5.1.3 Pruning

Pruning is the process of eliminating neurons or parameters. This is the oldest optimization considered by our study and dates back to LeCun, et al., 1989 [66]. In this chapter, we prune parameters with "small" absolute values, after the policy has been trained. The method is similar to that presented in the previous section, where, initially,

the full-precision network is trained. Then parameters with an absolute value less than the p^{th} percentile are set permanently to zero. Finally, the network is fine-tuned to compensate for the missing data. In [35], pruning resulted in a 9×–13× reduction in network size, while still maintaining high accuracy. See Algorithm 9 for more details.

Algorithm 9: Pruning+VPG

Require: Full-precision policy network parameterized by θ_{all}, learning rate η, and pruning threshold parameter p.

Ensure: Fine-tuned network incorporating real-valued pruned parameters θ_p for each layer.

1. Full-precision training:
For each state observation perform full-precision (FP32) network evaluation. Select actions from resulting output distribution.
Update all FP32 parameters θ_{all} using standard VPG and η. Repeat until maximum FP32 performance is achieved.

2. Pruning:
For each layer, eliminate parameters less than the layer's p^{th} percentile, resulting in θ_p.

3. Fine-tuning:
For each state observation perform network evaluation using θ_p. Select actions from resulting output distributions.
Use VPG to update θ_p. Repeat Step 3 until maximum performance is achieved.

5.2 Results

We have implemented the optimizations described above using PyTorch [67] and the popular reinforcement learning benchmark suite OpenAI Gym [68]. In particular, we have used the optimization methods on three discrete action-space environments: CartPole-v0, Acrobot-v1, and Atari Pong. We compare the optimized results to full-precision VPG (FP+VPG). While the CartPole-v0 and Acrobot-v1 are deterministic control problems, it has been shown that if an RL algorithm performs successfully on those, it is a good indication that it will perform well on a more difficult problem. This heuristic holds

true, for example, when using the Compression+VPG optimization method, as discussed below.

The CartPole-v0 benchmark is a finite-horizon, simulated physics control challenge in which a pole is attached to an un-actuated joint and balanced vertically upon a cart. The cart moves laterally along a track, and the goal is to apply force to the cart to keep the pole balanced. The agent is provided with state observations consisting of: cart position, angle of the pole, cart velocity, and rate of change of the angle. In OpenAI Gym, the agent may apply a force of +1 or -1 to the cart at each time step, and a reward of +1 is returned at each step that the cart is balanced. The environment returns the "done" signal when the pole moves more than 15 degrees from vertical, or the cart moves more than 2.4 units from the starting position, or if the pole is kept balanced for more than 200 time steps. The environment is considered solved when the agent collects an average reward of 195 over 100 episodes.

Acrobot-v1 is a two-link pendulum finite-horizon environment where only the joint between links is actuated. Initially the arm is pointed down, and it must be swung up and balanced. The agent's task is to apply joint torques such that the lower link is swung up and kept balanced. The state observations include sine and cosine of the joint angles, as well as joint velocities. In OpenAI Gym the torques may be +1, 0, or -1. The environment returns -1 reward at each step and ends in failure after 500 steps or in success if the distant link is elevated beyond a threshold before 500 steps.

The OpenAI Gym Atari environment is a wrapper for the Arcade Learning Environment and includes over 50 games. We learned agents for the classic Pong game, in which it competes against Pong's original AI agent. The state observation is game pixels, and the available actions are move up, move down, and no move. In OpenAI Gym, the environment terminates the game after either player reaches 21 points.

A single hidden layer neural network was selected as the neural architecture for all

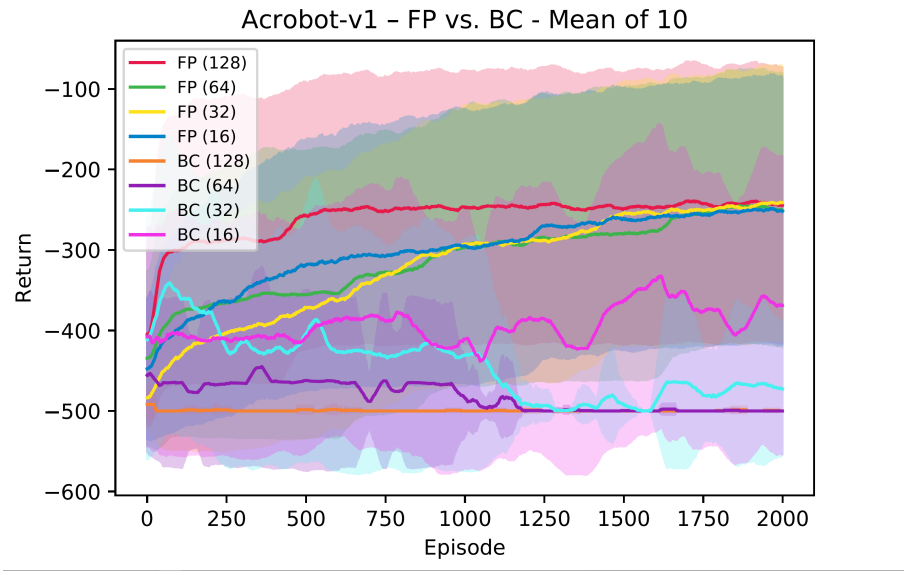


Figure 5.1: BinaryConnect+VPG (BC) performed poorly on the Acrobot-v1 task, compared to full-precision VPG (FP). When using 16 units in the hidden layer (the smallest version of BC) some learning takes place.

experiments. The input layer and output layer sizes varied depending on the state and action-spaces of the environment being solved. We varied the number of units in the hidden layer from 256 down to 16 for the CartPole-v0 and Acrobot-v1 tasks. For the Pong-v0 task we used 256 and 128 units in the hidden layer. In the given plots, performance is reported as the mean of ten separately trained policies. Standard deviation of each policy is also plotted.

Agent policies were initialized from the neural network topology described above, after which pruning (Pruning+VPG), quantization (BinaryConnect+VPG and BinaryNet+VPG), and compression (Compression+VPG) methods were applied as described in the algorithms above. In addition to the mathematically optimized methods, a full-precision policy (FP+VPG) was trained on each problem to provide a baseline. Agents were tasked with learning each of the previously listed environments. As can be observed in the broader RL literature, no single agent dominated all tasks. In our study

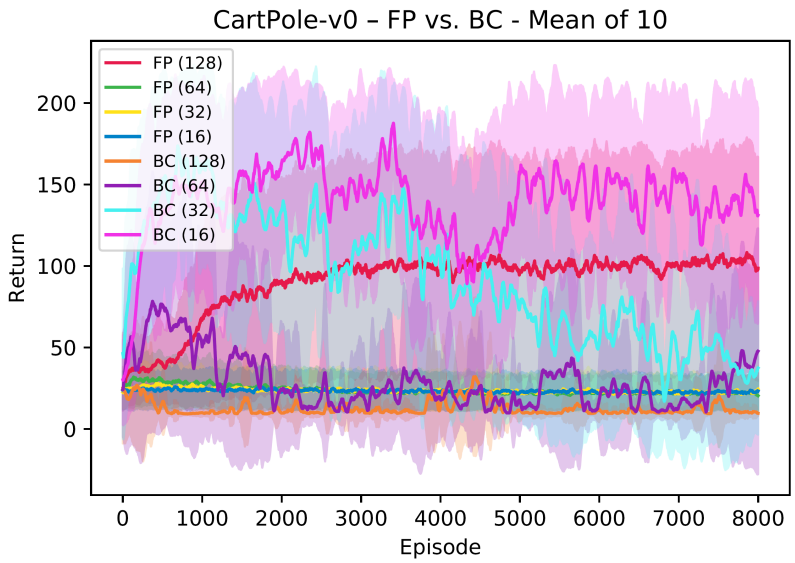


Figure 5.2: BinaryConnect+VPG (BC) with 16 and 32 hidden units performs favorably on CartPole-v0 compared to full-precision VPG (FP) with 128 hidden units.

we see that BinaryNet+VPG and BinaryConnect+VPG demonstrate erratic behavior on each task, with times of high and low performance, and overall they do not perform well. Pruning+VPG and Compression+VPG showed excellent performance on the control tasks. Compression+VPG dominated at the Pong task and seems to be the most generally useful of the methods considered here.

5.2.1 Impact of Quantization

BinaryConnect+VPG performed poorly on CartPole-v0, but it performed well on Acrobot-v1, as shown in Fig. 5.1. However, it can be observed in Fig. 5.2 that BinaryConnect+VPG with 16 hidden units dominates all other variations. This may indicate that the other policies have too many parameters for this simple task. Less convincingly, as seen in Fig. 5.1, BinaryConnect+VPG shows random spikes of marginal performance on Acrobot-v1, and it never competes with FP+VPG. On the Acrobot-v1 task, the BinaryConnect+VPG models may not have the necessary capacity to perform consistently.

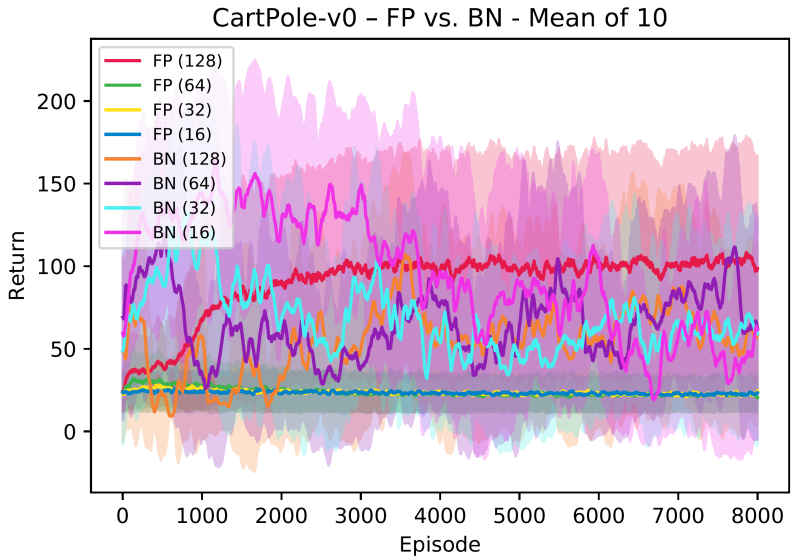


Figure 5.3: BinaryNet+VPG (BN) shows erratic behavior on CartPole-V0, but the 16 hidden unit version achieves continuous stretch of high returns around episodes 500–3,000, surpassing FP+VPG (FP).

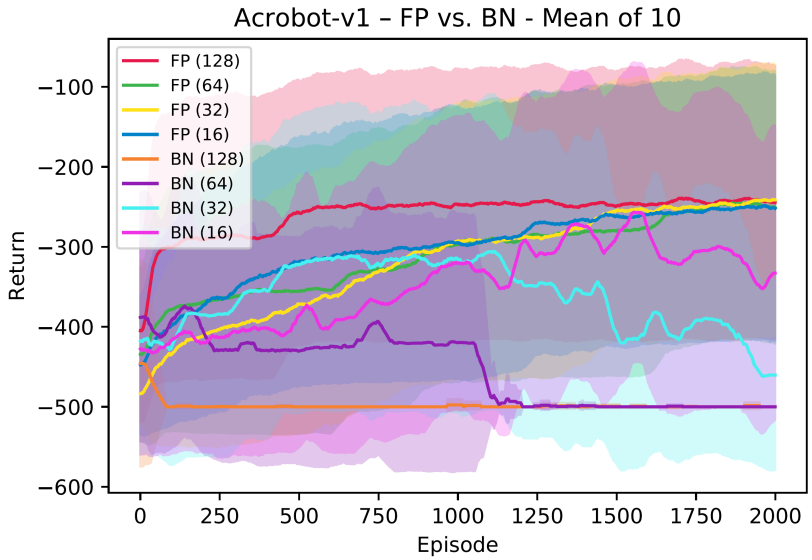


Figure 5.4: BinaryNet+VPG (BN) with 16 and 32 hidden units appear to be competitive to FP+VPG results. Initial performance indicates incompatibility with the simple update method used by VPG could be the cause of eventual performance degradation.

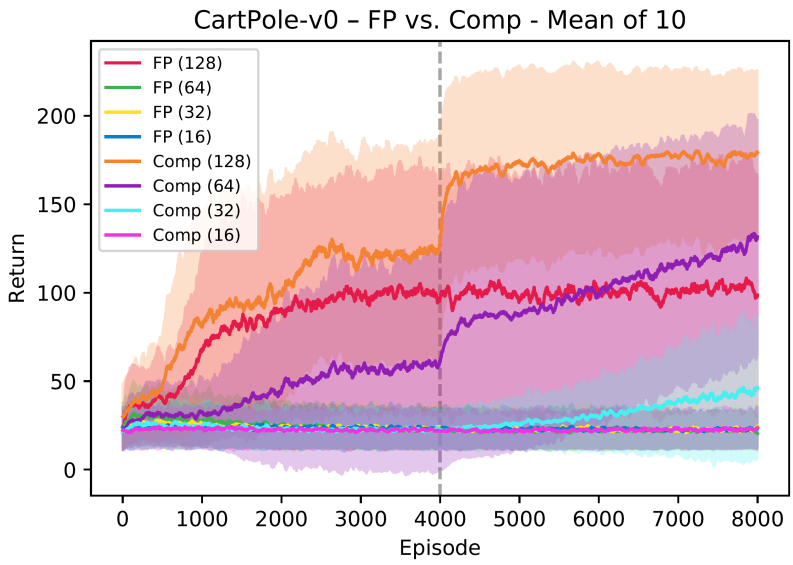


Figure 5.5: Compression+VPG (Comp) performs better than FP+VPG (FP) on CartPole-v0. Compression occurs at dotted line, after which performance of Compression+VPG increases.

In Figs. 5.3 and 5.4, it is shown that BinaryNet+VPG is a more interesting policy, with times of peak performance on both CartPole-v0 and Acrobot-v1. Its performance in Acrobot-v1 is particularly interesting and shows similar behavior to BinaryConnect+VPG, with a period of stability followed by increasing instability. Perhaps a different update strategy could prevent the instabilities.

5.2.2 Impact of Compression

Compression+VPG performed the most robustly among the mathematical optimization methods discussed in this paper. The results for Acrobot-v1, CartPole-v0, and Pong are given in Figs. 5.5, 5.6, and 5.7. As described in Algorithm 8, Compression+VPG uses a policy function which has an identical topology to the full-precision version for the first half of the episodes. After the halfway point, Step 2 of Algorithm 8 is used to compress the parameters. For our experiments, k was set to 8, which limits all parameters in each

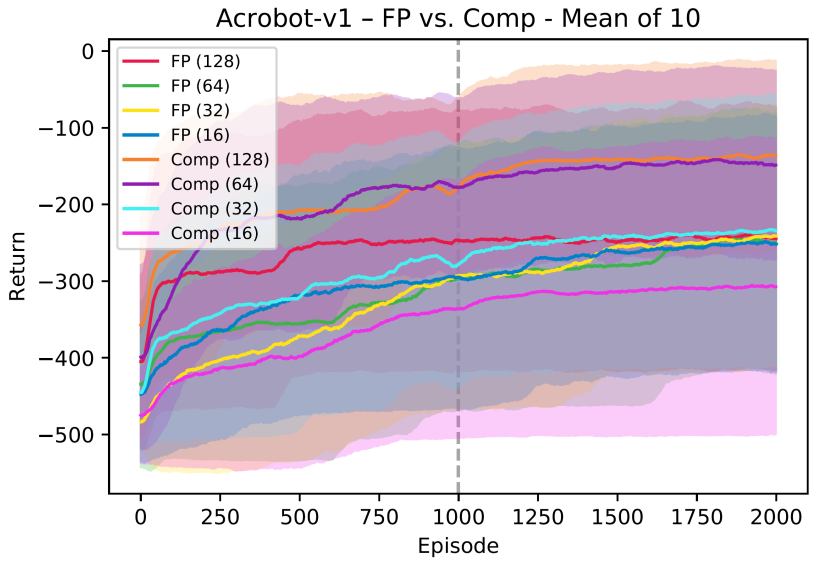


Figure 5.6: Compression+VPG performs better than FP+VPG on Acrobot-v1. Note that unlike the top and bottom plots, there is no discernible change in performance after compression for the Acrobot-v1 task.

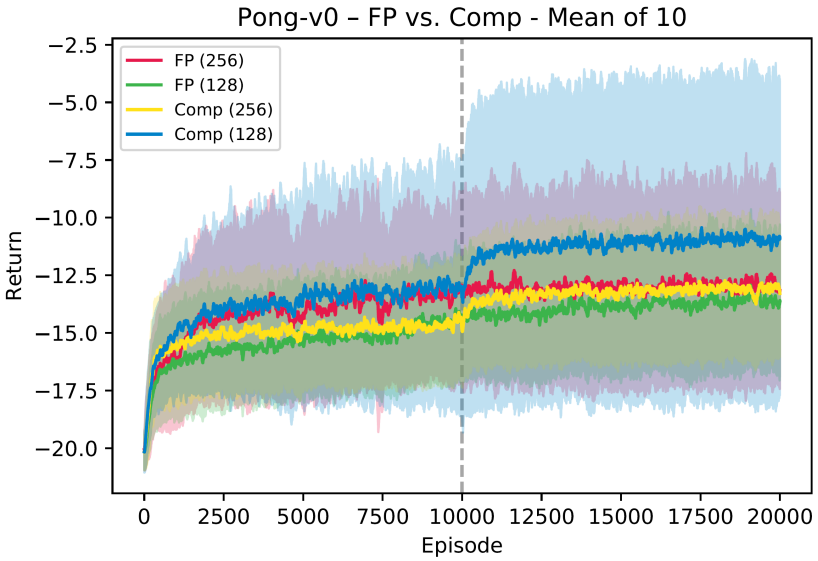


Figure 5.7: Compression+VPG applied to Pong-v0 environment shows stronger results than FP+VPG after tuning.

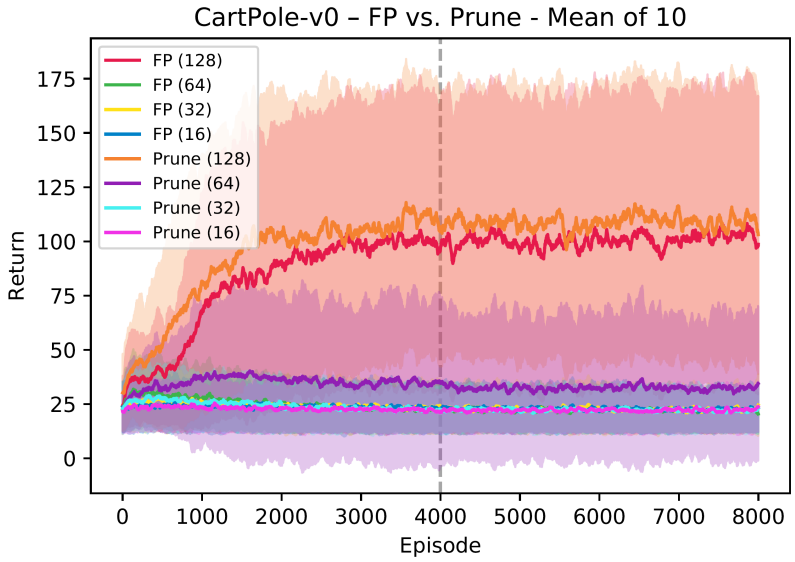


Figure 5.8: Pruning+VPG achieves equal performance to FP+VPG on CartPole-v0. Policy networks are pruned at the midpoint.

layer to 8 possible 32-bit floating-point values.

During the first half of all three figures, Compression+VPG performs the same as the baseline full-precision network, as it should, because during that time it is also a full-precision network. However, after compression takes place, we see a startling reduction in variance in one case and as well as improved returns in all cases.

5.2.3 Impact of Pruning

Pruning+VPG also exhibited excellent performance on CartPole-v0, with results shown in Figs. 5.8, 5.9, and 5.10. As with Compression+VPG, a full-precision policy is trained during the first half of each experiment, then, as described in Algorithm 9, the lower p^{th} percentile of network parameters are eliminated. In Fig. 5.8, 5.9, and 5.10 it is very promising that Pruning+VPG fully recovers after *50%* of its parameters have been removed.

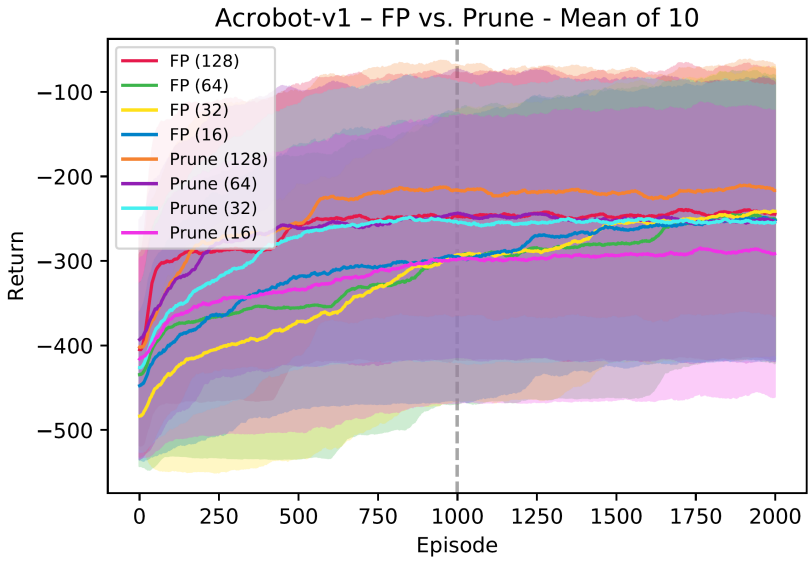


Figure 5.9: Pruning+VPG achieves equal performance to FP+VPG on Acrobot-v1. Policy networks are pruned at the midpoint.

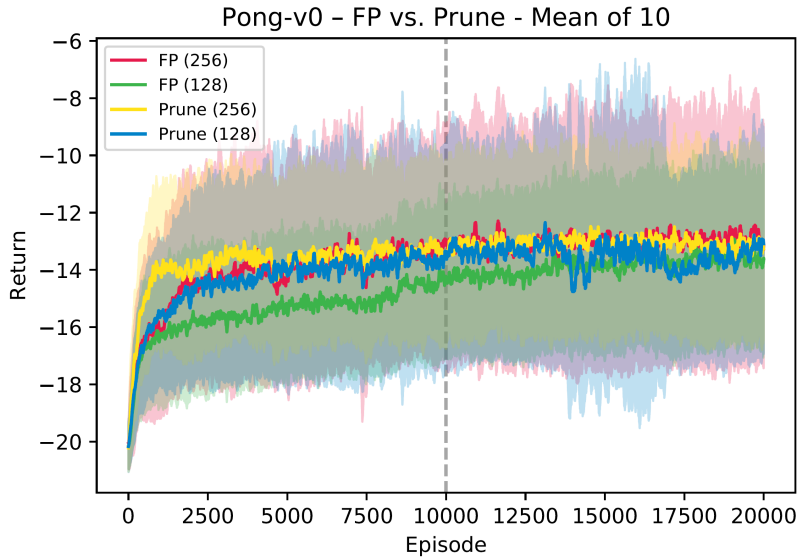


Figure 5.10: Pruning+VPG achieves equal performance to FP+VPG on Pong-v0. Policy networks are pruned at the midpoint.

5.3 Summary

To alleviate the immense computational requirements of deep neural networks it is desirable to employ optimized versions with comparable performance by taking advantage of mathematical simplifications. A suite of such mathematical optimizations has been pursued for deep neural networks and applied to domains such as image processing. Such optimization include binarization of parameters and inputs, clustering of parameters, and pruning parameters. However, it was previously unknown whether the existing optimization techniques can be readily applied to deep RL as well, without impacting the performance of the learned policy.

In this chapter, we have shown initial results indicating the strong performance that may still be achieved by deep RL, even under extreme optimization. In fact, the Compression+VPG method, which locked all parameters in each layer to 8 shared values, surpassed full-precision VPG on each environment (Figs. 5.5, 5.6, 5.7). And Pruning+VPG performed equally to VPG after fine-tuning (Figs. 5.8, 5.9, and 5.10). However, BinaryConnect+VPG and BinaryNet+VPG show promising, but very unstable behavior, which is most likely a result of the extreme quantization used for those methods. As is the case for reinforcement learning algorithms in general, we also observed that different optimizations are better suited than others for different problem domains. Furthermore it is still an open problem in RL to determine exactly how much model capacity is required for a particular task a priori.

VPG is a good baseline algorithm for optimized RL. It allows for experimentation with optimization methods, without confounding factors which would be included by more advanced policy-gradient based algorithms. However, VPG is notorious for exhibiting high variance, and therefore erratic collection of rewards, between policy updates. More advanced methods ensure lower variance and are also faster to train. As future work, we

will explore the interactions between more sophisticated RL algorithms combined with a broader array of mathematical optimizations.

Additionally, further experiments are needed to understand the trade-offs associated with applying various optimizations to different problem domains such as continuous versus discrete action-space tasks. The neural network architecture itself is also critically important and directly affects the impact various optimizations may have. For example, an over-parameterized neural network with more latent capacity than a given problem minimally needs will respond differently to the application of different optimization techniques than a minimal network for which optimizations may have a stronger impact. And, beyond assessing performance, more sophisticated implementation metrics can be analyzed such as: the number of multiplications and additions per policy action, size of policy parameters, and estimated power consumption.

In conclusion, the AI/ML communities have made great strides in the development of accurate and robust DNNs. The RL community is now incorporating such DNNs to an increasing degree and is showing results across a broad range of domains. Just as the architecture community has shown interest in DNN accelerator design, there will be increasing efforts toward deep RL accelerator design. However, because of the added complexity of RL, it is important to first understand the limitations of mathematical optimization for deep RL, before moving to the design of deep RL accelerators. This chapter shows that such a transition will be possible but future studies are required. The outcome of these studies will serve as a foundation for making architectural decisions for building RL accelerators and related neuromorphic processors.

Chapter 6

Distillation Strategies for Proximal Policy Optimization

6.1 Introduction

As introduced in Chapter 3.5, *distillation* is a method to transfer information learned by a high capacity, high parameter-count teacher neural network into a relatively low capacity, low parameter-count student neural network [42]. When using distillation, all teacher output probabilities are used as a training signal for the student, versus the single label that is normally used during training. Distillation leverages the fact that trained teacher class probabilities contain more information than a single label. For example, if an apple is presented to a classifier successfully trained to recognize images of food, then the classifier's class probabilities for apple, pear, peach, and orange most likely have some significant value compared to non-round foods. Furthermore, the low probabilities for other non-round classes provide information about what the input is *unlikely* to be.

The results presented in this chapter extend the work of [69] which used distillation to train a student neural network to match the deep Q-network (DQN) of a teacher trained

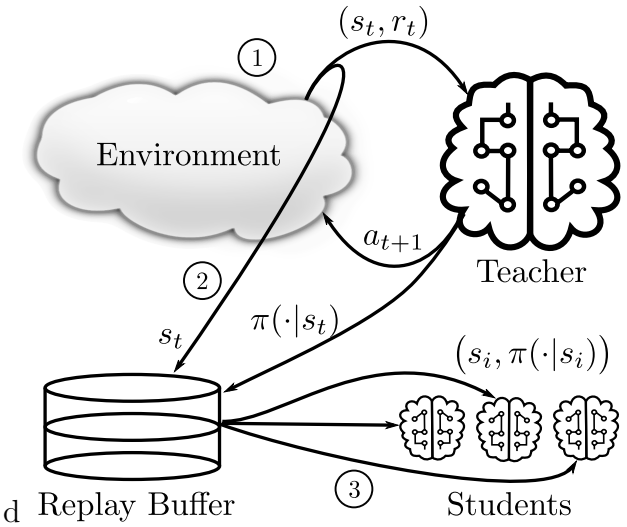


Figure 6.1: RL distillation has three phases: 1) teacher training, using standard RL algorithms, 2) using the trained teacher to interact with the environment and saving state observations and teacher's action probabilities into a *replay buffer*, and 3) the transfer of information stored in the replay buffer into student(s).

through the Deep Q-Learning algorithm. We refer to that technique as *DQN distillation*. A noteworthy feature of DQN distillation, and any variety of RL distillation in general, is that only the teacher is required to experience the environment. Once trained, the teacher may pass its knowledge to students, without the students being required to experience the environment as well, Fig. 6.1. Excellent results were obtained in [69], with the student DQNs often matching or exceeding the performance of the teacher DQNs on all tasks.

Actor-critic algorithms constitute a popular family of high performance deep RL algorithms. In the context of deep RL, actor-critic algorithms are typically composed of two networks: an actor network, which also serves as the agent's policy, and a critic network, which serves as a value function during policy improvement. DQN only uses a value function, which is queried during run-time. In this chapter, we reexamine DQN distillation in the context of the Proximal Policy Optimization algorithm, which was developed more recently than DQN and subsequently has many improvements [24]. PPO

was selected as our actor-critic algorithm because it is simple to implement and is widely used for a broad range of RL applications [70, 71, 72, 73].

As RL becomes appropriate for real-world applications, various "costs" to execute neural network forward-propagation becomes critical. Action latency, power consumption, silicon area requirements, and other design factors must be reconciled with the fact that relatively large neural networks typically provide state of the art results. RL distillation techniques will be broadly useful for neural architecture design. In particular, RL distillation will allow a machine learning engineer to 1) design the best policy, given their hardware constraints, or 2) identify minimum hardware requirements, given a satisficing agent performance metric. RL distillation methods provide the following benefits:

- Rapid student model exploration is enabled by the use of an experience replay buffer. RL distillation keeps a large replay buffer, which is populated with high quality state observations, actions, and action probabilities recorded by the teacher after its training is complete.

- Faster training times via high capacity teachers. It has been shown that high capacity agents decrease training time for both deep learning and deep RL [69, 74].

- Expensive environments, e.g. accurate physics simulations or physical systems, may only need to be experienced once by the teacher. The teacher's replay buffer may then be repeatedly used for offline actor distillation at a later date.

6.2 Background and Related Work

Distillation was proposed in [42] as a method to transfer knowledge from a trained teacher classifier neural network into an untrained student network. There are various techniques to implement neural network distillation, and here we review the version

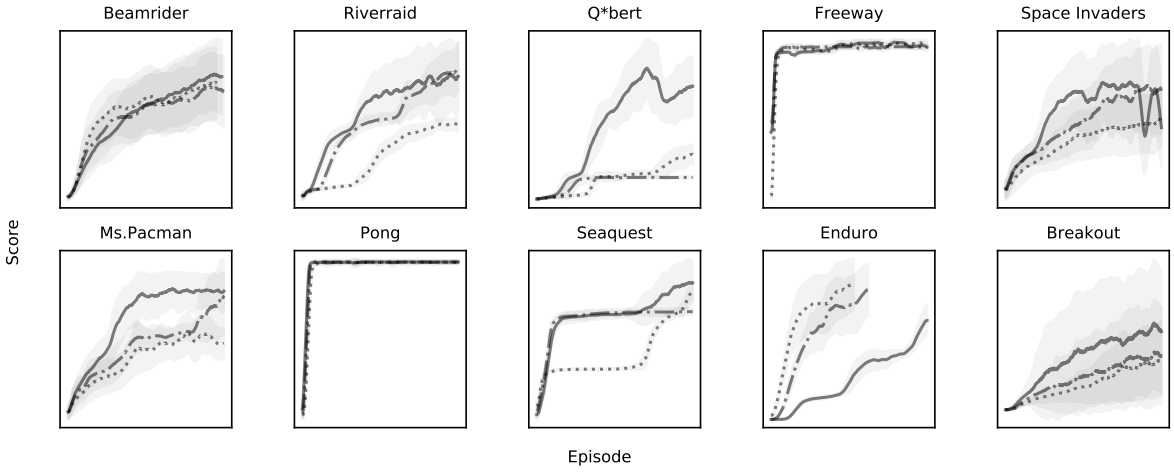


Figure 6.2: Qualitative analysis of effect of policy capacity on learning rate. Solid line (—) is high capacity, Dash-dot line (-·-) is medium capacity, and dotted line (····) is low capacity. In general, high capacity policies achieved higher performance, faster than low capacity policies. Y-axis is game score, x-axis is number of games played. Total number of steps was fixed across all games to 75×10^6 time steps. As detailed in the Results section, distillation allows for higher performing low capacity policies, compared to what they may achieve through environmental interaction alone.

most relevant to RL. Initially, assume a high capacity teacher classifier network has been trained to high performance, and a smaller network is to be trained with distillation. Additionally, assume access to the training inputs X used for teacher training, but no access to class C training labels $y \in C$. In this case, we may derive a loss function for the student network by providing training inputs $x \in X$ to the teacher network and using its class probability distribution $p_t(x) \in \mathbb{R}^{|C|}$ as a soft target for the student network's output probability distribution $p_\theta(x) \in \mathbb{R}^{|C|}$, where the student is parameterized by θ. The student's loss is defined as the distance between distributions $p_t(x)$ and $p_\theta(x)$ and may be measured using a standard metric, such as Kullback-Leibler divergence:

$$\mathcal{L}(p_\theta(x)|p_t(x)) = \sum_{i=1}^{|C|} p_{t,i}(x) \log \frac{p_{t,i}(x)}{p_{\theta,i}(x)}, \tag{6.1}$$

where $p_{t,i}(x)$ and $p_{\theta,i}(x)$ represent the probability for class i, given input x. The gradient of $\mathcal{L}$ may then be taken with respect to the student's parameters, which may then be updated using gradient descent.

As introduced by [69], distillation without labels maps to the RL setting. In the context of value-based algorithms like DQN, the output of the teacher Q-network is a vector of state-action values $q_t(s) \in \mathbb{R}^{|A|}$, where s is a state observation and A is a discrete action space. A probability distribution $p_t(s)$ may be obtained from the teacher by taking the softmax of $q_t(s)$. The state observations are then also provided to the untrained student network parameterized by θ, and its (originally random) state-action values may be interpreted as a probability vector by taking the softmax of its output, giving $p_\theta(s)$. The trained teacher Q-network is then distilled into a student network using the Kullback-Leibler divergence metric for the loss:

$$\mathcal{L}(p_\theta(s)|p_t(s)) = \sum_{i=1}^{|A|} p_{t,i}(s) \log \frac{p_{t,i}(s)}{p_{\theta,i}(s)}, \tag{6.2}$$

where $p_{t,i}(s)$ and $p_{\theta,i}(s)$ represent the probability for action i, given state observation s.

Eq. 6.2 would lead to low agent performance if used as given for DQN distillation. Recall that Q-values represent the expected return from state s, given that action a is taken, and the policy is followed thereafter. After training is complete, an agent makes its decisions by taking the action with the highest Q-value. By taking the softmax of the DQN, we are interpreting the Q-values as a probability distribution. This distribution may be relatively uniform, and because of the noise introduced during distillation, values in the student may not relatively match that of the teacher. Specifically, the Q-value for a suboptimal action in the teacher may become the highest Q-value in the student, and this would lead to degraded agent performance. The authors of [69] minimized the chance of this error by dividing all teacher Q-values by a temperature parameter $\tau = .01$, prior

	DQN	PPO Teacher	PPO Medium	PPO Low
Beamrider	**8672.4**	7500	7018	6958
Breakout	**303.9**	277	166	187
Enduro	475.6	722	827	**948**
Freeway	25.8	**34**	33	**34**
Ms.Pacman	763.5	3410	**4544**	2085
Pong	16.2	**21**	**21**	**21**
Q*bert	4589.8	**28367**	11646	18502
Riverraid	4065.3	13916	**15601**	9408
Seaquest	**2793.3**	2471	1908	2315
S. Invaders	1449.7	**1653**	1624	1312
% of DQN	100%	169%	150%	141%

Table 6.1: Comparisons of policies trained by DQN and PPO. DQN results are taken from [69]. PPO Teacher, PPO Medium, and PPO Low refer to agents trained using PPO with high, medium and low capacity policies respectively. Policy details are given in the Implementation Details section. All PPO policies were trained for 75×10^6 environment time steps and evaluated for 1×10^6 time steps. "% of DQN" is the geometric mean of the column divided by the geometric mean of the DQN column. DQN and PPO Teacher have the same architecture. Medium and low capacities have 25% and 7% of the parameters as DQN and PPO Teacher. Note that PPO Low has a geometric mean 41% higher than DQN, and that higher capacity PPO policies tend to have higher performance than lower capacity PPO policies.

to calculating the softmax. This has the effect of "sharpening" the teacher's probability distribution in $p_t(s)$, such that the highest probability is much greater than the next to highest.

After the teacher has been fully trained, a distillation training set is collected from the teacher into a replay buffer. The authors of [69] showed excellent distilled student performance across a variety of classic Atari 2600 games. Most significantly, a low capacity student network, with 7% of the parameters relative to their teacher network, performed at least as well as the teacher network.

PPO is an actor-critic algorithm which has stood out as being simple to implement and high-performance [24]. PPO is now established as a popular baseline with which to compare other RL algorithms and as a preferred algorithm for applying RL to new tasks and for applications outside RL algorithm research [75, 71]. Because of the popularity and performance of PPO, it was selected as our actor-critic algorithm.

In general, PPO learns more efficiently than the seminal DQN algorithm. Table 6.1 compares agents trained with DQN and PPO. Significantly, PPO agents with much smaller capacity (7%) achieved 41% higher than a high capacity DQN agent, when comparing geometric means.

A motivating factor for policy distillation is that it may be used to increase the sample efficiency and optimize the performance of a low capacity policy. In [69] it was speculated that a larger network accelerates learning. In [74] it was observed that high capacity policies are generally able to learn a task better and faster than low capacity policies. In the context of this chapter, the results in Fig. 6.2 also show that high capacity policies have performance advantages. In this figure, the average scores for agents using three different policy architectures are tracked during training for 75×10^6 time steps. All PPO agents were trained using Proximal Policy Optimization, as described in the Implementation Details section.

Distillation has also proven to be useful for neuromorphic hardware design. For example, the benefits of better sample efficiency and higher student performance through distillation were combined in [76] for efficient RL policy development. In this chapter, a high capacity policy trained with Double DQN, and represented by a standard convolutional neural network (CNN), was distilled into a student policy represented by a low precision spiking neural network to be executed on IBM's TrueNorth architecture. As TrueNorth has special restrictions, e.g. binary activations and ternary weights, it does not use a standard SGD algorithm. Instead TrueNorth uses the Energy-Efficient Deep Networks algorithm [77] to train a student to match a teacher's Q-values. Importantly, [76] demonstrates the viability of training a teacher policy once, using one type of algorithm, and distilling that policy into an arbitrary number of student policies, using the best training algorithm for each respective student.

6.3 Formulation

Actor distillation (AD) is an offline technique closest in formulation to DQN distillation, with the difference being that AD distills the teacher's true actor probabilities, i.e. the teacher's policy, π_t into the student π_θ, which are both functions of state observation s. Whereas DQN distillation transfers a proxy of the teacher's value function into the student.

AD proceeds as follows: a teacher policy π_t is trained to maximum performance on the environment. After training, the trained teacher interacts with the environment during a collection phase which records the teacher's state observations and action probabilities to a replay buffer. An uninitialized student network π_θ is then trained to mimic the teacher

with mini-batch SGD using the replay buffer and a loss similar to Eq. 6.2:

$$\mathcal{L}(\pi_\theta(s)|\pi_t(s)) = \sum_{i=1}^{|A|} \pi_t(a_i|s) \log \frac{\pi_t(a_i|s)}{\pi_\theta(a_i|s)}, \tag{6.3}$$

where s and $\pi_t(\cdot|s)$ are stored in the replay buffer.

π_t and π_θ are obtained by taking the softmax of a policy network logits vector. [69] obtained better results by dividing the teacher logits by .01, prior to taking the softmax. This has the effect of sharpening the teacher's probabilities. This was necessary because Q-values, which are learned using an ϵ-greedy explore-exploit strategy, have undefined behavior when converted to a distribution. AD does not require sharpening, because the teacher policy is stochastic anyway.

Optionally, after distillation is complete, the student may be fine-tuned by allowing it to interact directly with the environment and using a standard actor-critic algorithm.

6.4 Implementation Details

In this chapter, actor distillation was used to train students on 10 different Atari environments. We analyze the effect of AD on student performance, compared to agents with the same policy architecture as the student but trained directly on the environment with no distillation. We also analyze the effect of capacity on student performance, relative to agents with the same capacity but trained directly in the environment. Finally, we study the impact of allowing a distilled student to fine-tune on the environment after distillation is complete. Our environments are provided by the Arcade Learning Environment [78] and are interfaced with OpenAI Gym [68]. Additionally, we used PPO and distillation reference codes from [79] and [80].

Our PPO architectures use a single convolutional neural network body, followed by

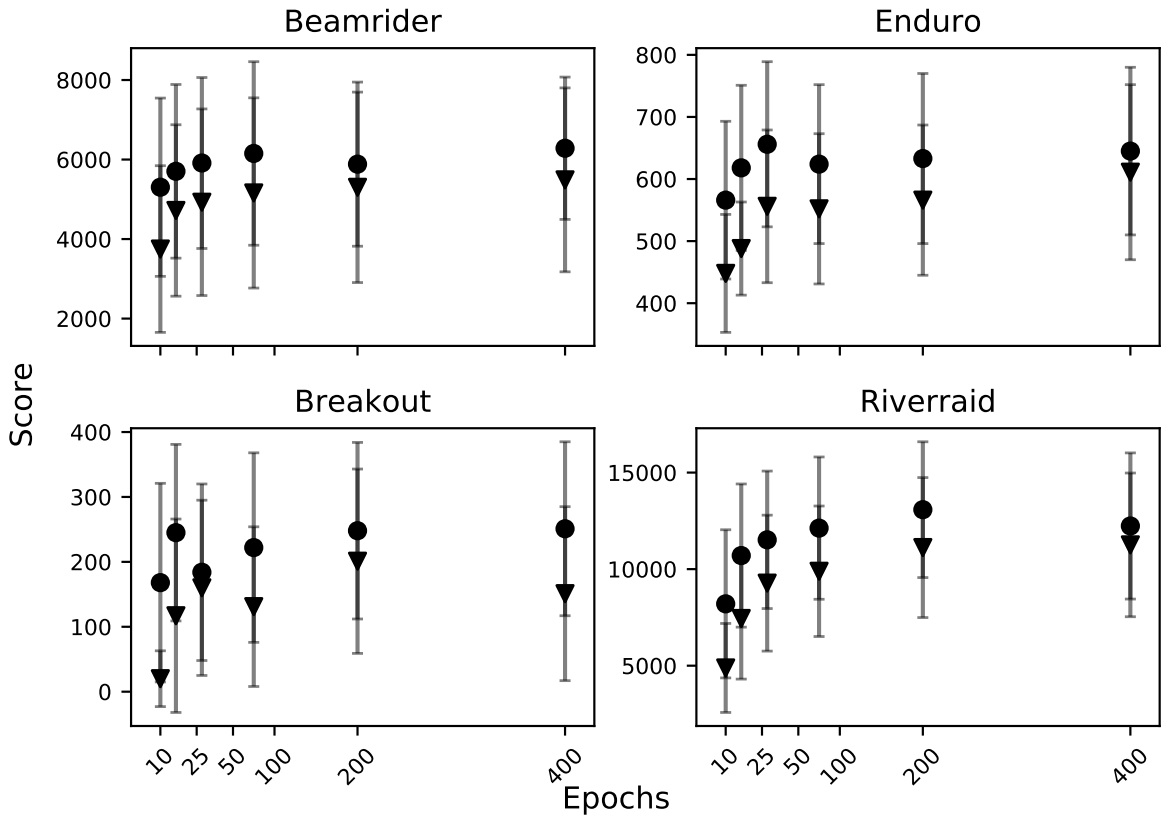


Figure 6.3: Effects of capacity on distilled performance. Student policies were distilled from between 10 and 400 epochs and then evaluated for 1×10^6 time steps. Circle ● represents medium capacity policies and triangle ▼ represents low capacity policies. Policies were reinitialized and distilled for each data point. Medium capacity policies have an advantage over low capacity policies when fewer epochs are used, but the advantage is often reduced as distillation progresses. As epochs increase, policies converge to maximum scores.

Capacity	Layer	Channels	Shape	Stride
High	Conv 1	32	8	4
	Conv 2	64	4	2
	Conv 3	64	3	1
	FC 1	n/a	512	n/a
Medium	Conv 1	16	8	4
	Conv 2	32	4	3
	Conv 3	32	3	1
	FC 1	n/a	256	n/a
Low	Conv 1	8	8	4
	Conv 2	16	4	2
	Conv 3	16	3	1
	FC 1	n/a	128	n/a

Table 6.2: Architecture details of policy feature extraction layers. High, medium, and low capacity agents were trained to play Atari. All architectures had three convolutional layers with inputs of $84 \times 84 \times 4$, followed by two fully connected layers (FC 1), followed by separate "heads": a fully-connected policy layer (π) with 3–18 units, depending on the environment, and a critic (V) unit with 512 units. The high, medium, and low capacity architectures had "bodies" with 1683456, 422912 (25%), and 106752 (6%) parameters, respectively.

two separate "heads": one for the actor and one for the critic. Two student capacities were investigated: one with medium capacity and one with low capacity, both relative to the teacher's high capacity network. In order to make a fair comparison, student network architectures were chosen to match those used for the DQN Distillation results. Network architecture details are given in Table 6.2.

For distillation and architecture baseline comparisons, the high (teacher), medium, and low capacity agents were trained for 75×10^6 time steps on each Atari environment. 16 agents ran in parallel with 2048 environment steps on each agent between each PPO update. Generalized Advantage Estimation was used to calculate returns with $\gamma = .99$ and $\tau = .95$. Within PPO, 10 epochs were used with batch sizes of 32 and clipping parameter set to .1. Adam was used with the stepsize set to 3×10^{-4}. Unlike [69], we do not divide teacher probabilities by a temperature and therefore directly use Eq. 6.3 for distillation. Tuning experiments on Beamrider, Enduro, Breakout, and Riverraid were

used for final hyperparameter selection.

6.5 Results

6.5.1 Distillation Results

After training or distillation, all agents were evaluated for 1×10^6 time steps of game play. Depending on the agent and game, 1×10^6 time steps resulted in 4 to 56 episodes per game. Results are given in Table 6.3. The bottom two rows of the table provide the geometric mean of the student versus the geometric mean of the PPO-based teacher and the geometric mean of DQN-based teacher scores reported in [69]. PPO is a more advanced algorithm than DQN, and even our low capacity PPO-trained agent obtain scores much higher than the DQN teacher.

The capacity of a student has an impact on how much information is transferred to a student. The medium capacity students obtain a geometric mean of 94% relative to the teacher, and the low capacity students obtain 85%. In general, then, it is beneficial to use larger capacity students.

Critically, as given in the Medium vs. Medium AD and Low vs. Low AD columns in Table 6.3, distilled students significantly exceed or meet the performance of equal-capacity agents which were trained directly on the environment. Recall from Table 6.2 that higher capacity policy networks typically reach higher performance, faster than lower capacity networks. By using distillation we may exploit this fact and not be penalized by it.

	Teacher	Medium	Medium AD	Medium AD tuned	Low	Low AD	Low AD tuned
Beamrider	15548 **7500±2322**	12464 7018±2183	(400) 11720 6284±1790	10278 5447±1867	12174 6958±1997	(400) 10974 5489±2313	9744 5548±1973
Breakout	442 277±115	395 166±123	(200) 425 248±136	434 **309±127**	372 187±69	(200) 414 201±142	397 159±103
Enduro	983 722±110	1288 827±223	(50) 967 656±133	1062 733±217	1386 948±181	(400) 791 611±141	1376 **1013±162**
Freeway	34 34±1	33 33±0	(10) 34 34±1	32 32±0	34 **34±0**	(10) 34 33±1	33 33±0
Ms.Pacman	3940 3410±333	8140 4544±875	(200) 4920 3413±355	6500 **5041±1263**	2090 2085±67	(100) 4830 3390±407	5450 3483±302
Pong	21 21±2	21 21±1	(10) 21 20±3	21 19±7	21 **21±0**	(10) 21 19±7	21 **21±0**
Q*bert	30075 28367±3651	11675 11646±377	(10) 29975 **28554±3301**	22100 20572±2209	20775 18502±2563	(10) 29650 23019±9190	15375 12152±1208
Riverraid	18980 13916±3552	20960 **15601±3189**	(200) 18830 13080±3517	18640 13716±2874	9910 9408±217	(400) 18630 11256±3721	18430 15593±2674
Seaquest	4580 2471±452	1980 1908±56	(10) 4980 2572±580	6060 **3708±1144**	2480 2315±103	(10) 4100 2219±465	5940 3550±1055
Space Invaders	2775 **1653±451**	3025 1624±596	(200) 2550 1432±469	2550 1569±375	2325 1312±336	(200) 2405 1382±343	2205 1288±263
% of Teacher	100%	88%	94%	100%	84%	85%	89%
% of DQN	169%	150%	160%	169%	141%	144%	151%

Table 6.3: Game scores over 1×10^6 time steps using directly-trained (Teacher, Medium, Low columns), distilled (Medium AD, Low AD), and distilled-then-tuned agents (Medium AD tuned, Low AD tuned). Teachers use high capacity architecture trained with PPO for 75×10^6 time steps. Medium and Low represent medium and low capacity architectures. AD represents actor distillation as described in Implementation Details section. The top row in each cell is agent's high score over all games. The bottom row in each cell provides mean game score $\pm$ standard deviation. Parentheses in AD columns provide number of epochs used for student distillation. "% of Teacher" columns are the geometric mean of column mean divided by geometric mean of teacher. "% of DQN" provides geometric means of students versus those obtained through DQN distillation [69]. Of note, medium capacity distilled student achieved 94% of PPO-based teacher, and 160% of DQN-based teacher results, which were, in turn, often higher than human player scores [20]. Medium capacity students distilled for 10 epochs and then allowed to train using PPO directly on the environment for 20×10^6 time steps ("Medium AD tuned") achieved Teacher's level of performance. Similar analysis applies to low capacity students.

6.5.2 Effect of Distillation Epochs

The optimal number of epochs used for distillation depends on the environment. Some games, e.g. Pong and Freeway, required 10 epochs of distillation to reach teacher performance. Others, e.g. Breakout and Ms.Pacman, required hundreds of epochs. The effects of increasing the number of distillation epochs on student evaluation performance for four games is given in Fig. 6.3. In general, higher capacity policies distill with higher final evaluation performance than lower capacity policies, but the performance difference diminishes as the number of epochs increase.

Each data point in Fig. 6.3 was created by initializing a new student policy (with random weights) and then distilling from between 10 and 400 epochs[1], and then finally evaluating the distilled student for 1×10^6 time steps in the environment. For the sake of sample efficiency, it would be preferable to have access to a proxy metric to know when further distillation is unnecessary, but we leave that for future work.

6.5.3 Fine-Tuning Results

We also studied the impact of allowing distilled students to learn in the environment, using standard PPO, after the distillation phase. Students were distilled for 10 epochs and then fine-tuned for 20×10^6 time steps, which is 27% of the number of time steps used to directly train the medium and low capacity policies. Notably, fine-tuning elevated the performance of the medium capacity distilled student to the performance of the teacher. In Table 6.3, the "Medium AD tuned" and "Low AD tuned" students have geometric means significantly higher than the students which were only distilled and not fine-tuned.

[1]Beamrider, Breakout, Enduro, and Riverraid were the only students distilled for 400 epochs.

6.6 Summary

Distillation is a robust and generally applicable optimization method. In this chapter we show that distillation may be used successfully in conjunction with Proximal Policy Optimization, a popular actor-critic reinforcement learning algorithm. The method presented here can be used during architecture search for efficient hardware and policy co-design.

Specifically, a high capacity trained teacher may be used to collect a replay buffer of state observations from the environment. Then the replay buffer and teacher probabilities may be used repeatedly to experiment with different student architectures. This method trains a low capacity reinforcement learning policy to achieve higher performance than it would have through direct interaction with the environment.

Furthermore, if it is possible for the student to also learn within the environment, we show that it is beneficial to first perform distillation followed by fine-tuning of the student.

Distillation of policies was originally in the context of a neural network trained to approximate Q-values. A limitation of Q-values is the inability to represent action values for continuous action spaces. Actor-critic methods have no such limitation. Future work can extend the ideas here to continuous action spaces.

The field of deep learning has a training heuristic called *early stopping*, which can be used to prevent overfitting. Early stopping monitors error on the training set, relative to error on a test dataset. As epochs increase, training error will always decrease, however test error will reach a minimum, before increasing again. Early stopping may be beneficial for distillation, but it is not clear. As may be seen in Fig. 6.3, distilled student performance is plateauing, but generally not dropping as epochs increase. We leave further investigation into this question for future work.

Chapter 7

Neuromorphic Engineering

7.1 Introduction

Neuromorphic engineering encompasses algorithms and architectures taking inspiration from the brain to perform computation. Neuromorphic algorithms may be executed on both von Neumann architectures (VA) and non-von Neumann architectures (NVA), or a combination of the two. NVAs have the potential to be more efficient than VAs at brain-inspired computations. This is due to NVAs closer similarity to biological neural architectures, often being highly connected and parallel, potentially low-power, and collocating memory and processing. Furthermore, both VAs and NVAs can be implemented in digital, analog, or mixed-signal hardware, with each implementation having different practical and theoretical trade-offs.

Currently the most popular route for building efficient DNN hardware is through digital implementations of NVAs, but there are alternative paths with potential efficiency gains. In this chapter, we analyze a recent analog NVA-based approach to deep RL, in Section 7.2, and we outline a benchmarking strategy for the field of neuromorphic engineering, in Section 7.3.

7.2 Memristors Learn to Play

In recent years there has been a race to leverage brain-inspired neuromorphic hardware to address the growing computational costs of deep learning. Analog arrays, particularly those consisting of programmable memristors, have been of interest because they natively and efficiently solve *vector matrix multiplications* (VMMs), which are the dominant operation of DNNs. While it has been shown that basic analog DNNs are tractable on memristor systems [81], an open question has been whether *analog neural networks* (ANNs) are suitable for more sophisticated algorithms like RL. In a recent issue of *Nature Electronics*, J. Joshua Yang and colleagues describe a hybrid digital-analog system that effectively solves classic control problems with RL [82].

The key piece of the system developed by Yang and team was a one-transistor, one resistor (1T1R) memristor crossbar of sufficient scale to fully encapsulate a 3-layer neural network capable of effectively learning control policies. Using Deep Q-Learning, the authors trained a neural network to approximate Q-values, which represent the expected future sum of rewards for given state-action pairs. Through many trial runs, where the agent can interact directly with the environment, the memristor network can be progressively trained with Deep Q-Learning to become an expert at its task.

Yang and colleagues demonstrated their hybrid RL system's ability to learn two classic RL control problems: Cart-Pole, where a cart must learn a policy to keep an unstable pole vertical by accelerating in different directions, and Mountain Car, where a car accelerates out of a valley by moving back and forth to obtain an escape velocity. There is a long way to go between these tasks and AlphaGo-level game learning; however, these early steps illustrate the long-term potential for neuromorphic implementations for solving harder problems. Many of the largest RL challenges of today can be viewed as scaled up versions of the simpler control problems solved here.

While promising, one of the apparent limitations of memristor crossbars is that their benefits are derived from physically instantiating VMMs; any operations that are not easily represented in that matrix form must be computed separately. For this reason, in their paper, Yang and colleagues leveraged conventional digital electronics to perform any calculations not directly leveraging the neural network weights. From a complexity perspective, they show that these digital calculations are far less costly than the memristor-optimized linear algebra; however, in practice it may be ideal to move entirely away from conventional electronics.

Can fully neuromorphic RL agents be achieved that leverage this approach? Notably, in addition to analog synapse arrays; radically lowered energy costs can also be achieved through event-driven "spiking" communication. Spiking hardware, so far primarily through digital CMOS, has been advanced by IBM TrueNorth [16], Intel Loihi [17], and others; and spiking algorithms can be developed for precise numerical tasks [83]. Often these spiking algorithms may not provide the blanket asymptotic scaling advantage described in Yang and colleagues' analysis of analog crossbars, but they can reduce the need for converting back and forth to conventional digital processors while offering low-power solutions to practical tasks.

The spiking and memristor communities have had limited interaction to date, however these approaches should be complementary in the long-run. For instance, it is possible to train DNNs, such as the RL network used by Yang and colleagues, to be compatible with spiking hardware; and such spiking networks are often compatible with lower precision synaptic weights [84]. This suggests that with only limited modifications, a spiking-analog hybrid approach could benefit from both analog synaptic operations and event-driven communication. As each of these approaches promise orders-of-magnitude lower power, the combination could be game changing.

Of course, there is much to do. The theoretical benefits of neural algorithms are

not fully understood, the implications of reduced precision need to be clarified, and there are engineering challenges ahead in scaling neuromorphic hardware. Furthermore, the appeal of achieving human capabilities by implementing brain-like algorithms still eludes the neuromorphic community, and AI broadly for that matter. However, the RL approach described here, while not a biologically-realistic model of decision learning, has many similarities with circuits in the brain, such as different time-scales of learning and feedback loops. Thus, the demonstration of memristor-enabled RL is an exciting step forward in seeing brain-inspired hardware reach its potential.

7.3 Benchmarking Event-Driven Neuromorphic Architectures

7.3.1 Introduction

In this section, we focus on a specific type of NVA: event-driven architectures, e.g. based on spiking neural networks, which are more biologically plausible than other mainstream NVAs, like digital deep neural network accelerators [85].

From an applications perspective, the focus on digital-systolic DNN architectures at most recent VLSI conferences seems justifiable, as the performance of contemporary DNNs is now sufficiently good that they are being used in safety-critical applications like autonomous driving, and they are computationally taxing on traditional VA's, motivating a need for alternative neuromorphic approaches. However, DNNs have only loose biological plausibility and they are only good at a narrow range of cognitive tasks. Attention [86] and Capsule Networks [87] are two recent examples which attempt to augment DNNs with greater biological plausibility, and we expect to see more examples in the future.

Given that the number and variety of possible neuromorphic approaches is unbounded, how are architecture design decisions to be made? Rigorous benchmarking has been foundational in advancing traditional computer architecture, however, as NVAs employ alternative paradigms from VAs, it is challenging if not meaningless to try and compare these architectures using solely the same metrics. Different architectural approaches are optimized for different benefits, so appropriate metrics are necessary to provide full understanding of the trade-offs and advantages each affords.

The mainstream DNN community has begun developing strong benchmarking efforts to highlight their advantages[1]. It is the intention of this work to outline benchmarking goals for the neuromorphic community. Our focus is on event-driven architectures, but the guidelines presented here may be applied to neuromorphic architecture evaluation in general. Highlighted by Fig. 7.1, we propose more extensive architectural evaluation metrics, analogous to how modern nutrition understanding has progressed to include more than just calorie counts. For example, rather than looking at single metrics like operation counts, a more complete understanding of micro-nutrients enables a greater nutritional understanding. Similarly, a more advanced understanding of multiple factors of architecture operation are needed to compare the strengths and weaknesses of computational architectures.

In the remainder of the paper, we expand on why we are focused on benchmarking event-driven architectures, intrinsic and extrinsic metrics, and benchmark candidates for neuromorphic processors.

[1] `https://mlperf.org/`

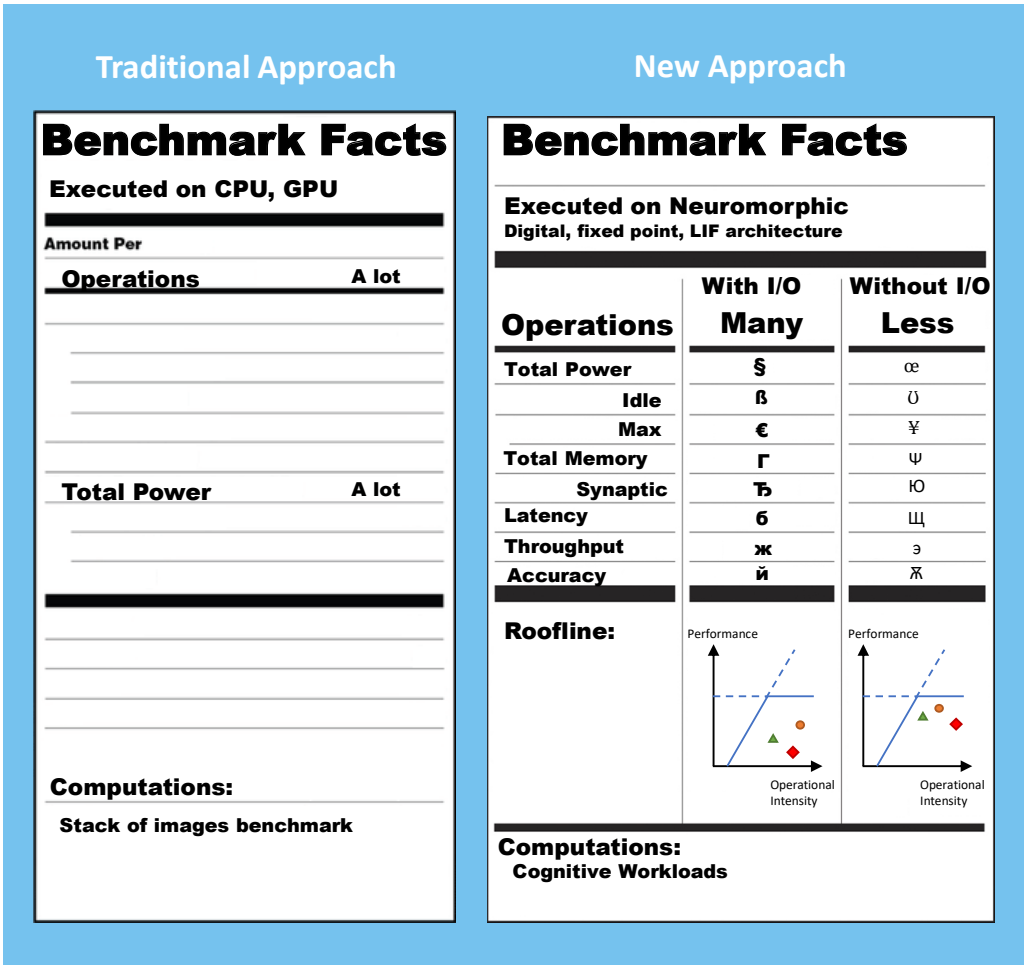


Figure 7.1: Illustration of traditional and proposed approaches to architectural benchmarking.

7.4 Event-Driven Neuromorphic Architectures

Event-driven neuromorphic architectures (EDNA), often modeled on spiking neuron models, are more biologically plausible than DNNs and offer the promise of higher efficiency for certain applications. These architectures are often able to take advantage of sparse connectivity and communication. Industry research platforms and academically available ASIC implementations of event-driven architectures currently include IBM's TrueNorth [?], University of Manchester's SpiNNaker [18], the Human Brain Project's BrainScaleS [88], and Intel's Loihi [17]. There are other ASIC and FPGA implementa-

tions, and many architectures that have yet to be physically realized [89].

7.5 Metrics

Evaluating a neuromorphic processor is nuanced. For example, the literature (or advertising material) for a processor may report "low power", but it may not report benchmarks for a dataset or a task of interest. Furthermore, other published architecture details may lack information required to compare a potential processor to its alternatives. Due to this nuance, we suggest two high-level categories of metrics: extrinsic metrics and intrinsic metrics, with a metric's category dependent on whether or not a workload must be processed to measure the metric. In this work, we provide recommendations for a variety of extrinsic and intrinsic metrics. These metrics may be used to compare and improve architecture designs.

7.5.1 Intrinsic Metrics

Intrinsic metrics may be measured without executing a workload on a processor. These metrics are simple to collect or may be gathered directly from technical manuals or publications, however, they do not provide sufficient information for a researcher to understand workload-dependent performance comparisons. They may not even indicate whether the architecture is likely to meet minimum specifications or performance requirements on tasks of interest.

Intrinsic metrics include *hardware metrics*, e.g. maximum power, idle power, silicon area, process size, clock speed, package dimensions, weight, memory, and time to reconfigure; *architecture metrics*, e.g. connectivity limits, communication limitations, reconfigurability, bit-precision options, IP protection (e.g. encryption), on-device learning availability, and built-in algorithm support; and *metadata metrics*, e.g. maturity, country

of origin, access to design files, programming support, and manufacturer.

7.5.2 Extrinsic Metrics

Extrinsic metrics require a specific workload to be executed on an architecture. By "workload", we are referring to the combination of a specific algorithm processing a specific set of data. If the input data changes, this may lead to different processing flows, and therefore to different extrinsic metrics. Extrinsic metrics include power, latency, throughput, accuracy, and roofline analysis.

The workload used to generate extrinsic metrics would ideally be matched to the type of workload for which the architecture was designed. For example, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) benchmark is a popular data set for workloads in the DNN community, but it may be an incongruous workload for most EDNAs [90]. Ideally, the neuromorphic community should have access to a suite of benchmarks which represent different brain-inspired tasks. This would allow researchers to select tasks tailored to their design, and also compare how their design performs relative to other designs on the same workload. We recognize that benchmarking event-driven systems could require hardware or datasets which are not yet widely available.

Roofline Analysis

Roofline plots are visual aids to understand performance of a workload on a particular architecture. This tool was developed for analysis of the interaction between system memory, processor performance, and application efficiency in high-performance computing (HPC) [91]. Roofline plots are espoused by a popular computer architecture book and were recently used for analysing systolic and dataflow DNN accelerators [92, 37, 93]. These plots are similarly useful for EDNAs.

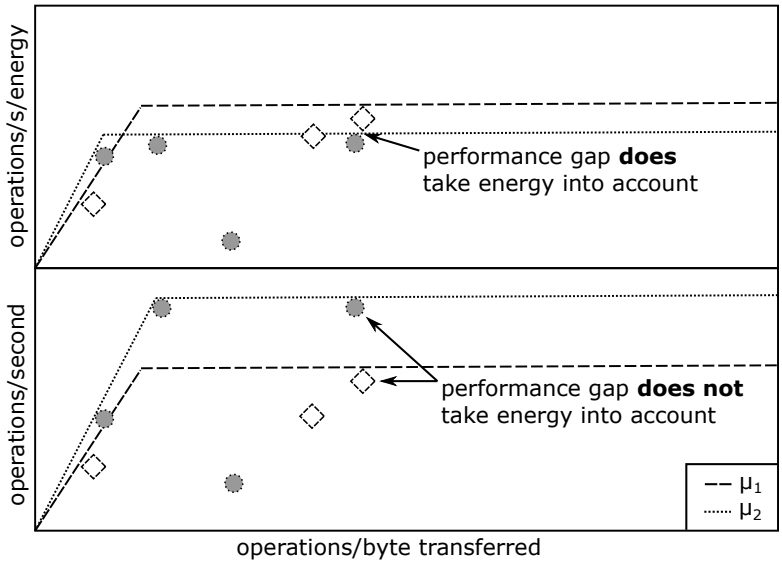


Figure 7.2: Illustrative roofline plots of two exemplar architectures (μ_1 and μ_2) comparing how various workloads utilize computational resources under two evaluation criteria. The bottom figure does not normalize for energy, but the top figure does. By normalizing for energy we can get a more accurate understanding of efficiency.

Illustrated in Fig. 7.2, the "roof" of the roofline plot is the maximum theoretical throughput at which a processor can perform some operation. The definition of "operation" is flexible. The HPC community uses floating-point operations (FLOPs). The DNN community may use multiply-accumulate operations (MACs). The neuromorphic community could use events (such as synapticops). Also, higher order operations may be defined, e.g. FLOPs/Watt.

A processor receives data from memory (or directly from a sensor). Memory (or sensor) bandwidth can be saturated if a processor demands too much data, too often. The slope of the roofline indicates memory saturation at various workload intensities, where "intensity" is from the perspective of a processor. A low intensity workload would finish quickly and require data quickly.

Generating a roofline plot requires a specific workload. Creating a roofline plot for an DNN would require the DNN and all of the training data. After a workload is selected, it

can be analyzed: how many operations are required, how many bytes of data for all the code and data, how long does it actually take to run on the hardware. This information is then plotted on the roofline plot.

Roofline analysis can indicate whether a workload is processor or memory-bound or if the workload's implementation is suboptimal for the architecture. In other words, it explains how efficiently a particular workload executes on a particular piece of hardware. A roofline plot may thus be used to compare the efficiency of two architectures for processing equivalent workloads, and it may be used to understand how a system should be optimized to increase performance.

Accordingly, roofline analysis not only illustrates how well workloads are implemented on and suitable for an architecture, but, by specifying the metric(s) of interest constituting an "operation", they can also provide greater insight into the function of the architecture. Amdahl et al., in their foundational paper which first specified the notion of a computer architecture, observed that the utility of an architecture comes from problems solved rather than bits-per-microsecond [94]. This premise is demonstrated by the mathematical optimization of DNNs showing comparable performance can be achieved with computationally simpler models using fewer computations. Next we describe workloads which we propose provide greater insight into the advantages of EDNA processing than singular metrics such as FLOPs often used to measure VAs.

7.5.3 Cognitive Workloads

In the following subsections we outline high-level cognitive applications we expect to see more brain-inspired neuromorphic systems attempting to solve in the near future. The application areas are drawn from [95] and range from basic sensory processing and pattern recognition to long-term planning at multiple timescales. We model our approach

after the Seven Motifs of Scientific Computing [96], which delineates the seven basic kernels of scientific computing: structured grids, unstructured grids, dense linear algebra, sparse linear algebra, fast Fourier transforms, particles, and Monte Carlo. Each of these core algorithms have different hardware and memory access patterns, and they are instantiated in a number of open source benchmark packages. Similarly [97] identifies six kernels ubiquitous in space applications: matrix addition, fast Fourier transforms, matrix multiplication, matrix convolution, Jacobi transformation, and Kronecker product.

Realistic and interesting workloads should be processed in order to exercise a system for generation of extrinsic metrics. To understand this claim, observe that an EDNA processes potentially sparse graphs. Assuming even a deterministic architecture, a single event change in an input may lead to numerous different downstream events. And while it is possible to generate arbitrary inputs and algorithms to stress specific aspects of an architecture, we consider it to be more meaningful if such tests are tied to problems which the community is interested in solving.

Similar to how scientific software is usually composed of multiple kernels discussed above, future cognitive systems will most likely combine two or more of the following.

Feed-Forward Sensory Processing

Cognitive systems need to perform pattern classification and regression from potentially multimodal sensory inputs. Modes may include vision, audio, tactile, sonar, radar, or other more abstract data like sales transaction information. Basic, but high accuracy, classification of static images began the current trend in DNN popularity and remains a mainstay of machine learning systems [98, 49].

Recurrent Sensory Processing

Success at feed-forward sensory processing implies that the current observation contains all the information needed for prediction. However, for systems with temporal dynamics or other time-dependent behavior, some type of memory is needed for accurate prediction. Simple memory is often achieved through network recurrency, where intermediate information is retained locally and processed alongside new information, allowing temporal dependencies to be learned. See [99, 86, 100] for examples.

Top-Down Processing

DNNs gradually build up features in a bottom-up approach. For example, filters from early layers in convolutional neural network learn to detect edges, while later layers learn to detect entire shapes. This is not how mammalian brains process sensory data in general. A more biologically plausible approach at feature extraction allows higher-level processing to affect lower-level processing. This top-down approach may be modeled with Bayesian algorithms. For some efforts in this direction, please see [101, 102, 103].

Dynamical Memory and Control Algorithms

Biological neurons, and groups of neurons, and various regions of the brain can be modeled as multiple dynamical systems. This is something that neither DNNs nor current EDNA architectures commonly do. The neuromorphic architecture community must wait for tractable models to become available before tackling problems in this space. One existing effort along these lines is [104].

Cognitive Inference Algorithms, Self-Organizing Algorithms and Beyond

The frontal and subcortical parts of the brain are responsible for long-term planning from earlier processed information. Popular reinforcement learning methods represent a simple example of long-term decision making. As progress continues with more capable feed-forward sensory processing, recurrent sensory processing, Bayesian neural algorithms, and dynamical memory and control, we expect to also see progress in their consolidation in the form of powerful long-term planning algorithms. This, as well as using these subsystems for life-long learning across multiple timescales, will represent much of the future effort for the neuromorphic algorithm community. For some interesting concepts on these and other ideas, refer to [105, 95, 106].

7.5.4 The Roles of Simulation and Emulation

In the previous section, we outlined five high-level cognitive application areas. Each of these areas are currently being studied across the neuromorphic computing spectrum. The DNN community has an advantage in the space, as backpropagation performs so well in so many problem areas that these systems have become useful for commercial, scientific, military, and medical applications. The DNN community arrived at this point by applying massive amounts of compute to massive amounts of data. On the other hand, despite having theoretically computational benefit, there are no training algorithms which have yet given event-driven systems the type of workload performance as has been seen in the DNN community.

In order to show progress according to some metric (e.g. power) it must be possible to processes some meaningful workload with high performance. Unfortunately it is expensive, in terms of dollars and time, to generate large amounts of event-driven I/O for sensory data – an area where EDNA architectures should excel at processing. We

propose the development of a physics-based simulation system designed to benchmark and compare DNN, EDNA, and other neuromorphic systems. The simulation system would have the following basic characteristics:

- Ability to generate multimodal physics simulations in both the standard spatial domain for DNNs and spatiotemporal domain for EDNAs. For example, both an RGB and DVS camera could be modeled by the simulator. To accurately account for full operation costs, the simulator needs to account of differences such as in bandwidth/transmission rates or power consumption associated with the different paradigms.

- If the neuromorphic hardware has an emulator then it may be used for training directly from the simulator outputs. This approach would allow for massive parallelization for the development of neuromorphic algorithms and the collection of many extrinsic metrics, e.g. spiking events per workload.

- If the neuromorphic hardware is a low-power physical system, a simulator interface board could be developed. The interface board would translate I/O between the simulator and the neuromorphic hardware. The I/O could include both analog and digital channels. Additionally, the interface board could be designed to provide power to the low-power system, thus it would be possible to measure the neuromorphic system's power consumption.

- If the neuromorphic hardware is actually a cluster, e.g. SpiNNaker and BrainScales, then the simulator's interface board would only communicate with the neuromorphic hardware, without measuring power. If the cluster has the ability to be partitioned, then multiple simulators could be connected.

Widely available access to appropriate workloads and computational resources is cur-

rently preventing both accurate comparisons between various DNNs and EDNAs, as well as participation from a wider community. Our simulation proposal aims to create a rich, flexible, physics-based environment. Once such an environment is available, then various challenge workloads may be created, e.g. controlling an autonomous vehicle with event-based sensors or performing robotic manipulation with event-based tactile input. Once appropriate workloads are created, algorithms and architectures may be developed and applied using either hardware emulation or domain appropriate I/O. Execution of the attempted solutions will then enable collection of extrinsic metrics, which will enable benchmarking for design improvement and comparison.

7.6 Summary

In this chapter, we analyzed a memristive non-von Neumann architecture that was used for deep RL. We then turned our attention to event-driven neurophic architectures, where we defined various metrics which may be used to evaluate EDNA performance. Taking inspiration from techniques employed by conventional computer architecture, we present how roofline analysis may be used in conjunction with other advanced performance measures to develop an understanding of the strengths and weaknesses of different architectures on common workloads.

Our approach emphasizes the importance of the algorithms and data being processed for the collection of meaningful and actionable metrics. Additionally, we motivate a need for the development of cognitive workloads, inspired by the motifs of scientific computing. Such cognitive workloads will include datasets to process, but they offer more than a simple data science challenge, as their combination with a task creates a computational requirement which stresses the architectures and articulates their merits, rather than simply providing a one-dimensional metric like floating-point operation counts. To ad-

dress these needs, we also propose a simulation framework that may be used to efficiently train and evaluate EDNAs on cognitive tasks.

Chapter 8

Conclusion

Let's step back and consider the cause for deep reinforcement learning's current popularity. The deep RL algorithms now being deployed do not differ significantly from RL algorithms developed decades ago [107, 13, 12]. The reason for the current excitement is that only now are we easily and robustly able to perform high-dimensional function approximation, thanks to deep neural networks.

Before taking any given action, an RL agent must answer the following question: "In the past, in scenarios similar to the one I am in now, what did I do that eventually led to the best return?" Without function approximation, each observation-action pair must have a unique entry in a lookup table in order to track the expected return associated with the pair. This becomes infeasible for even moderately sized problems.

When applied to RL, DNNs serve as function approximators to provide expected returns or to directly recommend actions. Just as DNNs can learn to recognize cats from different angles, so too can they learn to recognize similar environment configurations from different perspectives.

Paradoxically, while DNNs have been fundamental to recent progress in RL, their computational costs are limiting both real-world application and research progress. For

example, we recall the computational cost of the DNN used by AlphaGo Zero: approximately eight billion MAC operations per inference and over 12 trillion MAC operations per move. RL solutions with costs like this may prevent deployment in the presence of resource constraints.

During the past few years, explosive academic and commercial interest in DNNs has triggered a resurgence in the computer architecture community for designing efficient neural network accelerators. Complimentary to this, the DNN algorithm community has made strides in developing efficient neural architectures. This dissertation explores some of the ways deep RL can benefit from these recent advances in DNN efficiency with a concentration on developing co-design algorithms to retain RL agent performance while minimizing computational and resource costs.

In Chapter 2, we introduced Markov Decision Processes. MDPs are an abstraction in which an agent moves from state to state by making actions. Rewards may be given to the agent during state transitions. In general, the state transition function and reward function may be deterministic or stochastic and unknown to the agent. The MDP framework is general and can model many real-world problems. RL algorithms learn which actions in which states have historically led to the greatest long-term collection of rewards. Deep RL works by using function approximators to associate states with actions or value estimates. Function approximation is the only way to effectively solve MDPs with a continuous state space, e.g. for sensor-based driving, or when there are many possible discrete states, e.g. the possible board positions in the game of Go.

Chapter 2 also introduced the basic math behind Policy Gradient algorithms. PG algorithms use neural networks to directly map state observations to actions, similar to how a classifier learns to map features to labels. During training, an agent interacts with an MDP. The rewards an agent collects during MPD interactions are then used during policy updates, which are based on gradient-ascent. If the rewards were good,

the agent's action decisions will be strengthened the next time it encounters similar state observations. Otherwise the action probabilities will be weakened. PG methods are the foundation for many modern RL algorithms and understanding the material in Chapter 2 prepares the reader to approach current literature.

Chapter 3 provided a unified introduction to five methods of optimizing DNNs. Pruning eliminates parameters with small magnitudes from a DNN. This can reduce model size and reduce the number of MACs needed for inference. Quantization reduces the precision of parameters and activations, thereby reducing model sizes and hardware logic complexity. Parameter sharing replaces parameters that are close to each other with a common value, thus enabling compression to be applied before parameter transfer. Model distillation trains a large teacher network and then uses that teacher to train a lower-capacity student network, thereby allowing the student to reach higher performance than it would otherwise. Finally, filter decomposition turns a single square convolutional filter into two vector filters: a horizontal filter and vertical filter; this leads to fewer MAC operations. Many of the optimizations introduced in this chapter can by applied to DNNs with no, or only minor, impact on predictive performance.

Chapter 3 made it clear that there are many paths toward building more efficient DNNs. Automatically exploring different DNN designs is referred to as neural architecture search, and this approach is beginning to replace hand-crafted designs. Chapter 4 introduced a NAS approach, called RAPDARTS, that takes into account both accuracy and resource use. The algorithm works by learning a cell architecture that is then connected in a chain to form a standard DNN. The contribution of RAPDARTS is that while most resource-aware NAS techniques require thousands of GPU hours to complete the search, our technique only requires hundreds of GPU hours. As an example application, we use RAPDARTS to search for neural architectures which require less than 3×10^6 parameters for use on the CIFAR-10 dataset. (This is a small number of parameters

relative to most reported CIFAR-10 NAS results.) Our results are only outperformed by a method that used thousands of GPU days for their search.

Chapter 5 studied the impact of quantization, pruning, and weight sharing on RL DNN policies for several OpenAI Gym tasks. Significantly, we presented an RL algorithm that binarizes parameters to eliminate multiplications. In the CartPole environment, several agents were able to achieve good performance with this optimization. Similarly, an algorithm that binarized weights and activations (reducing neuron calculations to signed integer addition) achieved impressive performance on Cartpole and Acrobot. The binarized optimizations are exciting for the efficiency gains, however our algorithms for pruning and weight sharing showed no reduction in performance. When using the pruning algorithm, 50% of the parameters were removed. The weight sharing algorithm restricted all parameters to eight possible 32-bit floating-point values.

In Chapter 6, we trained large teacher policies to learn many different OpenAI Gym Atari games using the PPO RL algorithm. After the teacher was finished training, one million game images were recorded from teacher game play. We then used the recorded game images and our distillation algorithm to transfer the teacher policies into smaller student policies. The distillation process involved evaluation of the student and teacher policy networks at each recorded game image. The teacher's policy output was then used to provide an error signal for the student. In general, when trained directly on the games, larger, high-capacity policy networks were able to outperform low-capacity policies. However, we showed that on average low-capacity distilled students were able to outperform low-capacity policies that were directly trained on the games. Students often achieved teacher performance when they were allowed to be fine-tuned by playing the games directly. If environment interaction is expensive, these results show that teacher observations can be used for later training of students.

Chapter 7 considered neuromorphic engineering, which encompasses both DNNs and

other more biologically plausible approaches to intelligent algorithms and hardware. We first analyzed recent literature which fabricated part of a neural network accelerator using memristors. Crossbar arrays of memristor can be used to leverage Kirchhoff's Voltage and Current Laws to almost instantaneously perform vector matrix multiplication (VMM) at low energy cost and low latency. This provides substantial savings because VMM is the most computationally intense operation when computing a fully-connected layer (i.e. one layer of a multilayer perceptron) or when computing convolution as matrix multiplication. The memristor VMM was incorporated into a hybrid digital-analog system and used to solve two OpenAI Gym classic control tasks. The cost of analog-to-digital conversion in this approach outweighs the savings from analog VMM calculation, but the results prove there are opportunities for fast and efficient memristor-accelerated RL.

Chapter 7 also considered event-driven architectures which are favored by the neuromorphic engineering community. Event-driven architectures have a computational advantage in that their processing elements only compute when neuron firing thresholds are exceeded, in contrast to standard DNN processor arrays which must compute continuously to maximize throughput. While event-driven architectures offer theoretically superior performance for neuromorphic applications, the field has yet to develop algorithms which are as compelling as deep learning. Part of the problem is that there are no established benchmarks for the neuromorphic community. In this chapter, we proposed an initial path for benchmarking these promising systems. Our proposal draws on the idea of kernels from the computer architecture community. A kernel is defined as the combination of an algorithm and the data it processes. Kernels are often used when generating roofline plots, which allow computer and software engineers to measure efficiency and to identify bottlenecks. The neuromorphic community should clearly define a hierarchy of tasks, beginning with simple feed-forward processing and ending with long-term planning. They should then treat these tasks as kernels which may then be

used for common evaluation of algorithms and hardware.

8.1 Future Work

8.1.1 Integrating Multiple Optimization Methods

The primary contribution of this dissertation is the development of DNN-based RL policy optimizations. Experimentally we have considered, in isolation, the effects of pruning, quantization, parameter sharing, and model distillation on policy performance. We have also presented a new neural architecture search algorithm for the automatic design of DNN architectures that have high performance and meet resource constraints. The next step is to integrate these approaches.

For example, policy distillation lends itself well to neural architecture search. During policy distillation, all state observations may be recorded for later distillation into a student. The student's neural architecture will impact how well it absorbs the distillation. It would be simple to use NAS to search for student architectures optimized for distillation. Furthermore, our resource-aware NAS algorithm can be used in scenarios where student architectures must meet latency or other constraints.

Furthermore, NAS, as it is described here, and as used in the majority of the literature, does not yet incorporate pruning, quantization, or parameter sharing into the search process. These primitive optimizations could be applied to the student model after the search, but this is probably not optimal. For example, if a resource constraint exists for the number of parameters in the model, then the NAS algorithm should deal with that directly, and apply pruning or other applicable methods during the search itself, and not as an afterthought. Our RAPDARTS algorithm can be extended to support primitive optimizations during the search phase.

8.1.2 Multimodal Reinforcement Learning

The DNNs used in this dissertation were either multilayer perceptrons or convolutional neural networks. No memory [99, 108], autoencoders [109], attention or transformer layers [86], or other architecture types were studied. Optimizing these other architecture types is important because they are recently being incorporated in *multimodal* RL.

Multimodal agents receive two or more sensor steams, e.g. visual, acoustic, text, joint torque, pressure, RADAR, or LIDAR [5, 110, 111]. The existing efforts that have considered multimodal agents focus on how to solve the technical challenges of integrating multiple sensors – they do not consider the computational cost or how to optimize the multimodal policy's neural architecture. The methods presented in this dissertation should be able to discover high performing and efficient multimodal neural architectures, with or without resource constraints.

8.1.3 Neuromorphic Engineering

Finally, the field of neuromorphic engineering has developed neural models, often event-driven, which are relatively more complex than current deep learning models [112]. The more-biologically-plausible models from neuromorphic engineering do yet compete with DNN-based learning, but when they do, we expect they will be used for RL. Perhaps the methods introduced here will be inspiration for optimizing event-driven neuromorphic RL agents too.

Bibliography

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[2] Y. Tian, J. Ma, Q. Gong, S. Sengupta, Z. Chen, J. Pinkerton, and L. Zitnick, "ELF opengo: an analysis and open reimplementation of alphazero," in *International Conference on Machine Learning*, 2019, pp. 6244–6253.

[3] "Google cloud gpu pricing," 2019. [Online]. Available: https://cloud.google.com/compute/gpus-pricing

[4] Y. Aytar, T. Pfaff, D. Budden, T. Paine, Z. Wang, and N. de Freitas, "Playing hard exploration games by watching youtube," in *Advances in Neural Information Processing Systems*, 2018, pp. 2930–2941.

[5] G. Wayne, C.-C. Hung, D. Amos, M. Mirza, A. Ahuja, A. Grabska-Barwinska, J. Rae, P. Mirowski, J. Z. Leibo, A. Santoro *et al.*, "Unsupervised predictive memory in a goal-directed agent," *arXiv:1803.10760*, 2018.

[6] "Passive house," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Passive_house

[7] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations*, 2017.

[8] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[9] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *International Conference on Machine Learning*, 2017, pp. 2902–2911.

[10] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *International Conference on Learning Representations*, 2019.

[11] M. Minsky, "A neural-analogue calculator based upon a probability model of reinforcement." Harvard University Psychological Laboratories, Cambridge, Massachusetts, Jan. 1952.

[12] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[13] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, 2000, pp. 1057–1063.

[14] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[15] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[16] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[17] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[18] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014.

[19] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[21] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," in *International Conference on Machine Learning*, 2015.

[22] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations*, 2016.

[23] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016.

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

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv:1312.5602*, 2013.

[26] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *AAAI Conference on Artificial Intelligence*, 2018.

[27] M. Zhang, S. Vikram, L. Smith, P. Abbeel, M. Johnson, and S. Levine, "Solar: Deep structured representations for model-based reinforcement learning," in *International Conference on Machine Learning*, 2019, pp. 7444–7453.

[28] D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution," in *Advances in Neural Information Processing Systems 31*, 2018, pp. 2451–2463.

[29] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," in *International Conference on Machine Learning*, 2019, pp. 2555–2565.

[30] S. Levine, "Lecture notes in deep reinforcement learning, cs 285." University of California, Berkeley, Jan. 2018.

[31] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE International Solid-State Circuits Conference*, 2014, pp. 10–14.

[32] W. Dally, "High-performance hardware for machine learning," in *Advances in Neural Information Processing Systems*, 2015.

[33] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[34] S. Marcel and Y. Rodriguez, "Torchvision the machine-vision package of torch," in *International Conference on Multimedia*. ACM, 2010, pp. 1485–1488.

[35] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations*, 2016.

[36] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *International Symposium on Computer Architecture*. ACM, 2017, pp. 27–40.

[37] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[38] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015.

[39] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[40] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 4107–4115.

[41] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2006, pp. 535–541.

[42] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *Stat*, vol. 1050, p. 9, 2015.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[44] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua, "Learning separable filters," in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2013, pp. 2754–2761.

[45] M. Lin, Q. Chen, and S. Yan, "Network in network," in *International Conference on Learning Representations*, 2014.

[46] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[47] D. Strukov, "Lecture notes in neuromorphic engineering, ece 594bb." University of California, Santa Barbara, Mar. 2018.

[48] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[49] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[50] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[51] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *AAAI Conference on Artificial Intelligence*, 2017.

[52] X. Chen, L. Xie, J. Wu, and Q. Tian, "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," in *International Conference on Computer Vision*, 2019.

[53] S. C. Smithson, G. Yang, W. J. Gross, and B. H. Meyer, "Neural networks designing neural networks: multi-objective hyper-parameter optimization," in *International Conference on Computer-Aided Design*. ACM, 2016, p. 104.

[54] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," in *International Conference on Learning Representations*, 2019.

[55] Z. Lu, I. Whalen, V. Boddeti, Y. Dhebar, K. Deb, E. Goodman, and W. Banzhaf, "Nsga-net: neural architecture search using multi-objective genetic algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2019, pp. 419–427.

[56] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search," *arXiv:1905.01392*, 2019.

[57] S. Xie, H. Zheng, C. Liu, and L. Lin, "Snas: stochastic neural architecture search," in *International Conference on Learning Representations*, 2019.

[58] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," in *International Conference on Learning Representations*, 2019.

[59] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," *Submitted to International Conference on Learning Representations*, 2020, under review.

[60] X. Chen, L. Xie, J. Wu, and Q. Tian, "P-darts published cifar-10 genotype," 2019. [Online]. Available: https://github.com/chenxin061/pdarts/blob/b1575e101aedb7396a89d8a7f74d0318877a1156/genotypes.py

[61] X. Chen, L. Xie, J. Wu, and Q. Tian, "P-darts source code," 2019. [Online]. Available: https://github.com/chenxin061/pdarts/tree/05addf3489b26edcf004fc4005bbc110b56e0075

[62] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789.

[63] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Conference on Uncertainty in Artificial Intelligence*, 2019.

[64] X. Zhang, Z. Huang, and N. Wang, "You only search once: Single shot neural architecture search via direct sparse optimization," *arXiv:1811.01567*, 2018.

[65] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[66] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems*, 1990, pp. 598–605.

[67] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *Advances in Neural Information Processing Systems, Autodiff Workshop*, 2017.

[68] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv:1606.01540*, 2016.

[69] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, "Policy distillation," in *International Conference on Learning Representations*, 2016.

[70] P. Long, T. Fanl, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning," in *IEEE International Conference on Robotics and Automation*, 2018, pp. 6252–6259.

[71] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine, "Learning complex dexterous manipulation with deep reinforcement learning and demonstrations," *Robotics: science and systems XIV*, Jun. 2018.

[72] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8697–8710.

[73] Y. Wang, H. Lee, and L. Lee, "Segmental audio word2vec: Representing utterances as sequences of vectors with applications in spoken term detection," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2018, pp. 6269–6273.

[74] S. Green, C. M. Vineyard, and Ç. K. Koç, "Impacts of mathematical optimizations on reinforcement learning policy performance," in *International Joint Conference on Neural Networks*. IEEE, 2018.

[75] J. Zhang, L. Tai, P. Yun, Y. Xiong, M. Liu, J. Boedecker, and W. Burgard, "Vr-goggles for robots: Real-to-sim domain adaptation for visual control," in *IEEE Robotics and Automation Letters*, vol. 4, no. 2. IEEE, 2019, pp. 1148–1155.

[76] J. L. Mckinstry, D. R. Barch, D. Bablani, M. V. Debole, S. K. Esser, J. A. Kusnitz, J. V. Arthur, and D. S. Modha, "Low precision policy distillation with application to low-power, real-time sensation-cognition-action loop with neuromorphic computing," *arXiv:1809.09260*, 2018.

[77] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing," *National Academy of Sciences*, vol. 113, no. 41, pp. 11 441–11 446, 2016.

[78] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of artificial intelligence research*, vol. 47, pp. 253–279, Jun. 2013.

[79] I. Kostrikov, "Pytorch implementations of reinforcement learning algorithms," 2018. [Online]. Available: https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail

[80] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," in *International Conference on Learning Representations*, 2018.

[81] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, p. 61, 2015.

[82] Z. Wang, C. Li, W. Song, M. Rao, D. Belkin, Y. Li, P. Yan, H. Jiang, P. Lin, M. Hu *et al.*, "Reinforcement learning with analogue memristor arrays," *Nature electronics*, vol. 2, no. 3, p. 115, 2019.

[83] S. J. Verzi, C. M. Vineyard, E. D. Vugrin, M. Galiardi, C. D. James, and J. B. Aimone, "Optimization-based computation with spiking neurons," in *International Joint Conference on Neural Networks*. IEEE, 2017, pp. 2015–2022.

[84] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone, "Training deep neural networks for binary communication with the whetstone method," *Nature Machine Intelligence*, vol. 1, no. 2, p. 86, 2019.

[85] J. L. Krichmar, W. Severa, M. S. Khan, and J. L. Olds, "Making bread: Biomimetic strategies for artificial intelligence now and in the future," *Frontiers in neuroscience*, vol. 13, p. 666, 2019.

[86] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[87] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in Neural Information Processing Systems*, 2017, pp. 3856–3866.

[88] S. Schmitt, J. Klähn, G. Bellec, A. Grübl, M. Guettler, A. Hartel, S. Hartmann, D. Husmann, K. Husmann, S. Jeltsch *et al.*, "Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system," in *International Joint Conference on Neural Networks*. IEEE, 2017, pp. 2227–2234.

[89] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *arXiv:1705.06963*, 2017.

[90] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, and et al., "Imagenet large scale visual recognition challenge," in *International Journal of Computer Vision*, vol. 115, no. 3. Springer Nature, Apr. 2015, p. 211–252.

[91] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," Lawrence Berkeley National Lab, Berkeley, California, United Sates, Tech. Rep., 2009.

[92] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2017.

[93] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE journal on emerging and selected topics in circuits and systems*, 2019.

[94] G. M. Amdahl, G. A. Blaauw, and F. Brooks, "Architecture of the ibm system/360," *IBM journal of research and development*, vol. 8, no. 2, pp. 87–101, 1964.

[95] J. B. Aimone, "Neural algorithms and computing beyond moore's law," *Communications of the ACM*, vol. 62, no. 4, p. 110, Apr. 2019.

[96] P. Colella, "Defining software requirements for scientific computing," 2004.

[97] T. M. Lovelly and A. D. George, "Comparative analysis of present and future space-grade processors with device metrics," *Journal of aerospace information systems*, vol. 14, no. 3, pp. 184–197, 2017.

[98] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[99] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[100] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," in *International Conference on Machine Learning*, 2015, pp. 2067–2075.

[101] S. Ahmad and J. Hawkins, "Properties of sparse distributed representations and their application to hierarchical temporal memory," *arXiv:1503.07469*, 2015.

[102] J. B. Tenenbaum and F. Xu, "Word learning as bayesian inference," *Proceedings of the Annual Meeting of the Cognitive Science Society*, vol. 22, no. 22, 2000.

[103] Y. Tian, A. Luo, X. Sun, K. Ellis, W. T. Freeman, J. B. Tenenbaum, and J. Wu, "Learning to infer and execute 3d shape programs," in *International Conference on Learning Representations*, 2019.

[104] C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen, "A large-scale model of the functioning brain," *Science*, vol. 338, no. 6111, pp. 1202–1205, 2012.

[105] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman, "Building machines that learn and think like people," *Behavioral and brain sciences*, vol. 40, 2017.

[106] P. Taylor, J. Hobbs, J. Burroni, and H. Siegelmann, "The global landscape of cognition: hierarchical aggregation as an organizational principle of human cortical networks and functions," *Scientific reports*, vol. 5, p. 18112, 2015.

[107] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[108] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.

[109] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *International Conference on Learning Representations*, 2013.

[110] H. Huang, V. Jain, H. Mehta, J. Baldridge, and E. Ie, "Multi-modal discriminative model for vision-and-language navigation," *arXiv:1905.13358*, 2019.

[111] X. Wang, Q. Huang, A. Celikyilmaz, J. Gao, D. Shen, Y.-F. Wang, W. Yang Wang, and L. Zhang, "Reinforced cross-modal matching and self-supervised imitation learning for vision-language navigation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 6629–6638.

[112] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris *et al.*, "Simulation of networks of spiking neurons: a review of tools and strategies," *Journal of computational neuroscience*, vol. 23, no. 3, pp. 349–398, 2007.