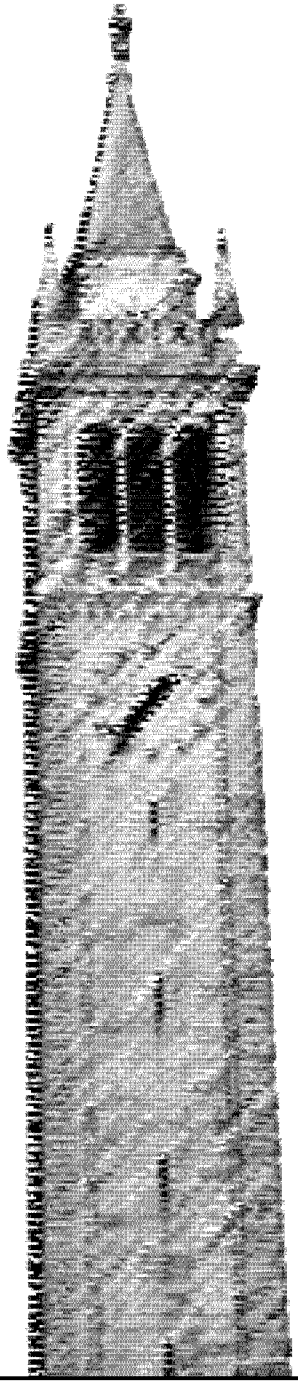
Scalable Reinforcement Learning Systems and their Applications



Eric Liang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-48
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-48.html

May 11, 2021

Scalable Reinforcement Learning Systems and their Applications

by

Eric K Liang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Michael I. Jordan
Professor Barna Saha

Spring 2021

Scalable Reinforcement Learning Systems and their Applications

Abstract

Scalable Reinforcement Learning Systems and their Applications

by

Eric K Liang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

The past few years have seen the growth of deep reinforcement learning (RL) as a new and powerful optimization technique. Similar to deep supervised learning, deep RL has demonstrated the ability to solve problems previously thought to be unapproachable with machine learning (e.g., fine motor control of robotics, sports and video gaming), and substantial improvements over heuristic solutions for existing problems (e.g., in systems optimization, e-trading, advertising, robotic control). Like deep learning, deep reinforcement learning is necessarily computationally intensive. Because of this, researchers and practitioners in the field of deep RL frequently leverage parallel computation, which has led to a plethora of new algorithms and systems.

This thesis looks at deep RL from the systems perspective in two ways: how to design systems that scale the computationally demanding algorithms used by researchers and practitioners, and conversely, how to apply reinforcement learning to expand the state of the art in systems. We study the distributed primitives needed to support the emerging range of large-scale RL workloads in a flexible and high-performance way, as well as programming models that can enable RL researchers and practitioners to easily compose distributed RL algorithms without in-depth systems knowledge. We synthesize the lessons learned in RLlib, a widely adopted open source library for scalable reinforcement learning. We investigate the applications of RL and ML for improving systems, specifically the examples of improving the speed of network packet classifiers and database cardinality estimators.

To my parents *Yulin and Ji-fuh*, for their hard work, love, and support.

Contents

List of Figures

List of Tables

Acknowledgments

I am very grateful to the many people without whom this thesis would not have been possible. They have inspired my work and helped me grow professionally and as a person. I would like to thank:

My advisor Ion Stoica for bringing me to Berkeley and guiding me throughout my PhD. He has been an endless source of ideas and optimism about the potential for real-world impact. Watching our projects grow to maturity, starting from nothing more than a few thoughts in a document, has taught me many lessons.

I would like to thank all the members of the Ray team, including Robert Nishihara, Philipp Moritz, Richard Liaw, Stephanie Wang, Edward Oakes, Sven Mika, Ameer Haj-Ali, Alexey Tumanov, Si-Yuan Zhuang, Melih Elibol, Simon Mo, and Romil Bhardwaj. I learned a lot from our collaborations.

The RISElab and its systems, databases, and RL seminars were the source of many insights and friendships. I would like to thank my colleagues from my research groups, including Amog Kamsetty, Ankur Dave, Benjamin Hindman, Charles Lin, Chenggang Wu, Daniel Ho, Danyang Zhuo, Eugene Vinitsky, Eyal Sela, Frank Luan, Ionel Gog, Karl Krauth, Kristian Hartikainen, Lisa Dunlap, Lydia Liu, Michael Alan Cheng, Michael Luo, Michael Whittaker, Mitchell Stern, Nilesh Tripuraneni, Peter Schafhalter, Qifan Pu, Roy Fox, Vikram Sreekanti, Vlad Feinberg, Xin Wang, Yifan Wu, Zhanghao Wu, and Zhuohan Li.

I would like to thank my quals and thesis committee, Mike Jordan, Joey Gonzalez, and Barna Saha for their support and feedback on my thesis.

I would like to thank Xin Jin and Hang Zhu for their contributions to the NeuroCuts project, without which our results would not have been possible. I will never forget the feeling of having designed an algorithm that clearly, convincingly set the state of the art.

I greatly enjoyed my collaboration with Zongheng Yang during my PhD. It was fun being the only two systems students in a cutting-edge ML class, and seeing our class project grow into a significant and impactful research agenda.

I would like to thank the Bonita social pod: Aaron Davidson, Ahir Reddy, Debra Hsu, Kelly Huynh, Michael Treger, and Yahui Xiong, for their companionship and good spirits during the dark days of the COVID-19 pandemic.

My research journey began long before I entered the Berkeley PhD program. My early-career mentors: Michael Armbrust, James Hendricks, and Nevin Heintze showed me the challenges and rewards of working in the systems field. I will always remember the times spent pouring over data and whiteboarding in the Google offices. When I was an undergraduate, Dan Garcia, David Patterson, Mike Franklin, and Avideh Zakhor provided opportunities for trying my hand at teaching and research, which made all the difference.

Finally, I thank my family, who provided me with assistance every step of the way. In particular, my parents Yulin and Ji-fuh Liang, whose hard work and encouragement gave me the environment to learn and thrive. This thesis is dedicated to them.

Chapter 1

Introduction

Many fields today are seeing new solutions from deep reinforcement learning (deep RL), a new and powerful optimization technique that has emerged in the past few years, building on the success of deep supervised learning. While many of these solutions are coming to domains traditionally in the domain of machine learning and AI—for example, drug design [122], or gaming [6, 2, 138, 115], the field of systems has also seen a remarkable upwelling of RL inspired approaches. Areas previously thought to be too complex or too amenable to expert knowledge to be optimized—for example, data structures [90, 169], task scheduling [100, 108, 119], compiler phase optimization [64, 79], database query optimization [102, 117, 76, 85], chip layouts [104]—are now seeing ML and RL-based solutions that significantly surpass decades of research towards the state of the art. Since effective design of systems is so pervasive to our infrastructure, the promise of a better performing and more principled approach is immense. For many decades systems design has been powered by heuristics: pieces of code which encode the knowledge of practitioners and seek to optimize for the common case, or robustness, or some combination of the above. It has only recently been possible to think of learning these approaches, making it feasible to more directly optimize for end objectives.

Deep RL today, while still a complex and narrowly effective tool, is increasingly often "good enough" to generate great results in systems research. This is the reason we are seeing such a widespread interest in RL-based solutions. That is not to say that deep RL is easy to use—practitioners still need to determine the right optimization objectives and problem framing—only that when a problem formulation is found to be amenable to deep RL, good results become almost inevitable. Intuitively, this can be the case because systems problems, though complex to represent, are sometimes "simpler" than activities such as robotics control and video gaming that cannot be solved at all with heuristics. Hence, it is of great interest to accelerate the removal of the practical barriers for applying RL to systems problems today: in scaling up computation, and engineering tractable environments for learning, which together will enable rapid experimentation and implementation of large-scale RL applications.

As a concrete example of some of these barriers to applying RL today, consider a practitioner attempting to optimize the bus schedule for a city's metro network. An engineer can

design and deploy a heuristic in a matter of a few dozen or hundreds of lines of code. The same engineer trying to optimize the schedule with deep RL would have to come up with a conceptual framing of the problem with an RL objective, choose an algorithm, algorithm hyperparameters, and neural network design. They would study the relevant literature to glean ideas from similar problems. They would likely have to build and deploy a parallel training system to evaluate their approach, and require several iterations before they start seeing reasonable results. Each iteration may take weeks or more of work, depending on how efficiently they are able to leverage their compute, and could require substantial re-implementation effort should new approaches need to be evaluated.

In other words, the new avenues for optimization opened by deep RL pose new, challenging systems problems that must be solved before RL can become truly practical. First among these problems is reducing the overhead of experimentation with deep RL at scale. This means both reducing the overhead on the practitioner implementing a proposed solution, and efficient execution of RL algorithms at large scale to save in compute costs. The engineer in the previous example should not have to spend time re-writing parallel code when they want to try out a new problem decomposition not well supported by existing libraries. Systems designed for workloads such as data processing or high-performance computing are not flexible enough to support the many different types of RL computations, which exacerbates the problem. New abstractions need to be designed that can flexibly support RL training and experimentation at large scale. The second problem, which is broader, is assembling the methods by which RL can be tractably applied to solve system problems. This involves identifying a suitable framing of the optimization objective, design of the simulation environment, and selection of the RL approach.

This thesis looks at the confluence of systems and reinforcement learning in both ways: how to design systems that scale the computationally demanding algorithms used by researchers and practitioners, and conversely, how to apply deep RL to solve systems problems in new ways. The first half of this thesis overviews the design and evolution of RLlib, a scalable and widely adopted open source library for distributed reinforcement learning. We study the distributed primitives needed to support the emerging range of large-scale RL workloads in a flexible and high-performance way, as well as programming models that can enable RL researchers and practitioners to easily compose distributed RL algorithms without in-depth systems knowledge. The second half of this thesis looks at how we can move towards leveraging RL and ML for improving the design of systems, specifically diving into examples improving the performance of packet classification trees and database cardinality estimators. We see how heuristics can be incorporated into learning based approaches, and how RLlib's scalability and flexibility accelerates research. RLlib and the other projects in this thesis have been developed together with many collaborators at Berkeley and from the wider open source community. The thesis is organized as follows:

- In Chapter 2, we provide an overview of key concerns in deep reinforcement learning landscape from the systems perspective. The design of both systems for RL and RL for systems is challenging due to the breadth of algorithms and applications, here we

hope to shed some light on the fundamental problems and concepts to prepare the reader for the next chapters.

- In Chapter 3, we describe an early version of RLlib, a system for distributed reinforcement learning we built on the Ray distributed system. Our initial work here focused on providing a small set of primitives for scaling RL computations. The challenges raised by early adopters of RLlib in academia and industry motivated further architectural improvements to Ray and new abstractions for RL training. This material was previously published in [92].

- In Chapter 4, we introduce new dataflow-oriented abstractions for distributed RL that seek to make the composition of high-performance algorithms more accessible to non-system experts. These abstractions also provide a unifying view into distributed RL from the dataflow perspective. This material was previously published in [91].

- In Chapter 5, we look backwards and answer the question "How can reinforcement learning be used to improve the design of systems?". We show advancements in the state of the art on packet classification, a fundamental problem in computer networking, which we were able to explore at scale thanks to the flexibility of RLlib. This material was previously published in [90].

- In Chapter 6, we look deeply at one piece of the RL for databases puzzle—constructing an efficient simulation environment. We demonstrate order of magnitude speedups on state-of-the-art deep generative modeling approaches to cardinality estimation, making it practical to leverage these new models in training environments. This material was previously published in [93].

- In the Conclusion we overview the impact of RLlib as an open source project, and cover areas of future work relevant to this thesis.

Chapter 2

Reinforcement Learning and Systems

To better understand the concerns for systems support for reinforcement learning, let us first discuss the central principles of RL and the concerns of its practitioners (e.g., researchers and applied RL engineers). In this chapter we overview RL basics along with the requirements for "Systems for RL", followed by the concerns of "RL for Systems" applications. We focus primarily on systems concerns instead of the learning theory.

2.1 RL Algorithm Basics

The goal of an RL algorithm is typically to improve the performance of an agent's *policy* with respect to an objective defined through an *environment* (e.g., simulator). An environment is typically formulated as a single Markov Decision Process (MDP). In this framing, a rollout (or simulation from start to end in the environment) begins with the environment at some initial state, $s_0 \in S$. At each step t, the agent executes an action $a_t \in A$ and receives a reward r_t; the environment transitions from the current state $s_t \in S$ to the next state $s_{t+1} \in S$. The goal is to maximize the total reward received by the agent, i.e., $\sum_t \gamma^t r_t$ where γ is a discounting factor used to prioritize more recent rewards.

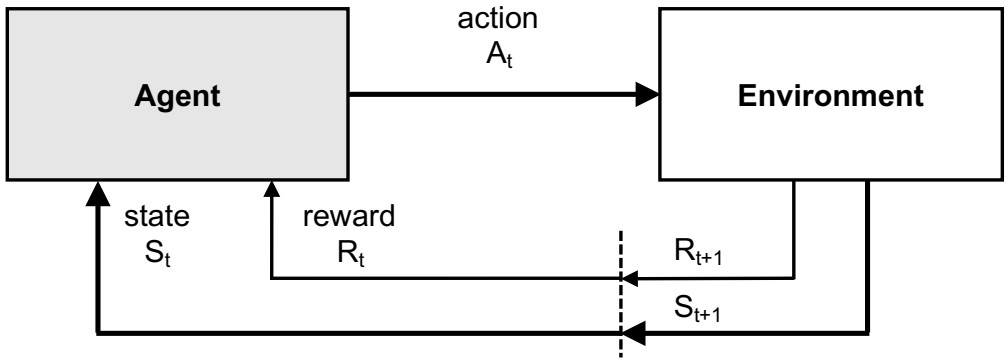


Figure 2.1: A typical RL environment formulated as a Markov Decision Process.

A textbook implementation of an RL algorithm would involve executing the above rollout steps serially, followed by an optimization routine. However in practice, RL training is computationally expensive, as both the rollout and optimization phases involve policy computations. The policy is usually defined as a deep neural network, which can range from several KB to several hundred MB in size. The simulator itself may also be quite slow to run. In order to achieve results in a timely way, parallelism and batching is often used at multiple levels in RL systems to reduce the computation time and take advantage of hardware accelerators. As a consequence, it is more useful to think of RL algorithm implementations as being comprised of the following logical steps of rollout, replay, and optimization. Each of these steps can contain substantial internal parallelism, and the steps can execute concurrently with respect to each other:

Rollout: To generate experiences, the policy, which outputs actions to take given environment observations, is run against the environment to collect batches of data. The batch consists of observations, actions, rewards, and episode terminals and can vary in size (10s to 10000s of steps). In general, rollouts may involve environments hosted on remote machines, interacting directly with the physical world, or data indirectly gathered through log files.

Replay: On-policy algorithms (e.g., PPO [134], A3C [105]) collect new experiences from the current policy to learn. On the other hand, off-policy algorithms (e.g., DQN [106], SAC [54]) can leverage experiences from past versions of the policy as well. For these algorithms, a *replay buffer* of past experiences can be used. The size of these buffers ranges from a few hundred to millions of steps.

Optimization: Experiences, either freshly collected or replayed, can be used to improve the policy. Typically this is done by computing and applying a gradient update to the policy and value neural networks. While in many applications a single GPU suffices to compute gradient updates, it is sometimes desirable to leverage multiple GPUs within a single node, asynchronous computation of gradients on multiple CPUs [105], or many GPUs spread across a cluster [162].

In Table 2.1 we summarize how a selection of RL algorithms realize above steps in different ways. All but the most naive (or single-node GPU accelerated) algorithms leverage distributed execution for rollouts, which means rollouts can be scaled across multiple machines. Optimization is typically centralized for algorithms that communicate experiences over the network, but distributed for those that compute gradients decentrally and synchronize them asynchronously (A3C) or via allreduce (DD-PPO [162]). Specialized single-node implementations [8, 144] leverage multiple hardware accelerators on a single machine for both rollouts and optimization. Very large scale RL systems such as IMPALA [36] or Rapid [12] may leverage GPUs spread across multiple machines, but have some degree of logical centralization of optimization vs rollout processes. In this thesis we will show that RLlib pro-

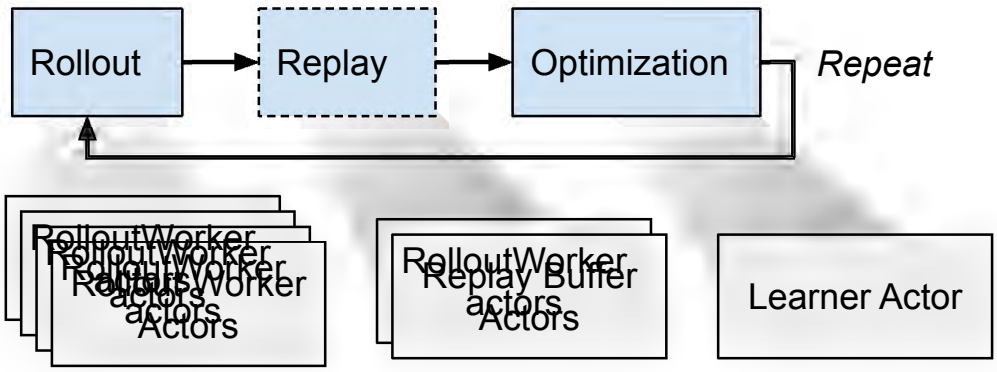


Figure 2.2: Most RL algorithms can be defined in terms of the basic steps of rollout, replay, and optimization. These steps are commonly parallelized across multiple actor processes. Depending on the implementation, these actors may be logically realized as separate operating system threads, processes, processes on different machines, or components thereof.

vides a sufficiently flexible and high-performance system substrate that can realize a broad range of these architectures.

	Rollout	**Optimization**	**Communication**
Single-threaded	centralized	centralized	None
A3C family [105]	distributed	distributed	gradients, weights
PPO family [134]	distributed	centralized	experiences, weights
DD-PPO [162]	distributed	distributed	gradients
GA3C [8], accelerated PPO [144]	centralized	centralized	experiences (unbatched), weights
IMPALA [36], Ape-X [63]	distributed	centralized and distributed	experiences, weights
SEED [37], Rapid [12]	distributed simulation, centralized inference	centralized and distributed	experiences (unbatched), weights

Table 2.1: The range of distributed RL algorithms. Despite sharing similar optimization objectives (i.e., policy gradient or Q-learning), these algorithms have very different performance characteristics that arise from differences in how they implement the steps of rollout, optimization, and communication of experiences for optimization.

2.2 Systems for Reinforcement Learning

At a high level, general purpose systems for reinforcement learning have to deal with the following concerns. Note that this list is not meant to be exhaustive, but covers the main concerns addressed by RLlib. This thesis focuses on the subset relevant to supporting *computation patterns*, including patterns for multi-agent computations. The other aspects are equally important for practical RL systems, but we believe are currently addressable with *good API design*. The interested reader may refer to the RLlib user-facing documentation (at `https://ray.readthedocs.io`) for coverage of these other aspects of RL systems design.

1. **Computation patterns**: RL algorithms span a broad range of computational requirements, from synchronous to asynchronous, from using CPUs to GPUs, from single-threaded to distributed across a cluster of machines, to name a few. In hapter 3, we discuss how RL systems can flexibly meet such requirements at high performance.

2. **Multi-agent computations**: Environments may consist or be decomposed into multiple cooperative or competing agents. Perhaps surprisingly, multi-agent scenarios can not only affect the algorithmic aspects of training but also the overall computational pattern of the algorithm (Chapter 4), though in most cases the concerns are shared with those of single-agent algorithms.

3. **Numerical definitions**: The RL system needs to provide an API for researchers and practitioners to define and customize policy and auxiliary losses for the algorithm. This API has to be sufficiently flexible to support new algorithms from the literature. While this thesis does not focus on API concerns (the reader can refer to the RLlib documentation), Chapters 3 and 4 touch on these issues.

4. **Batch training**: It is often desirable to be able to take advantage of data collected offline that is relevant to RL training. In Chapter 3 we briefly discuss how offline batch training can be unified with online simulations.

5. **Customization of existing algorithms**: Users often wish to customize the neural network architecture of their agents and other algorithm hyper-parameters.

6. **Complex environments**: Beyond simple games, real-world environments involve complex structured data (i.e., variable-length lists and structs of arrays), which the agent must be able to interpret. Environments may be interacted with in *vectorized* form for efficiency.

RLlib, the core system presented in this thesis, aims to provide a scalable and universal library for RL addressing the above concerns from researchers and practitioners of RL. In pursuit of this universality, we have had to sacrifice on certain other aspects. For example, enabling the system to flexibly execute many different computation patterns necessarily requires certain abstractions to be introduced. We discuss this and other concerns in the concluding chapter.

2.3 Reinforcement Learning for Systems

Computer systems involve many hand-engineered heuristics. We see these in every-day data structures (e.g., how splits are decided in tree-like data structures [11, 53]), task scheduling in operating systems [14] and distributed systems [128, 173], and query optimization in databases [7]. The deep learning revolution has prompted a wave of learning-based approaches which promise better performance compared to many of these heuristics. In this section we overview, conversely, some of the main concerns [57] that must be met when seeking to apply reinforcement learning and machine learning to systems. We also look at a couple examples addressing these concerns:

1. **Environment formulation**: Many advances in applied RL for systems do not involve fundamental innovation in RL theory. Rather, the insights come from the formulation of the RL environment and objective that enables tractable learning. In Chapter 5 we look at one such example involving an RL solution to the packet classification problem. More generally, this motivates having a general purpose, unified RL system which can support many different types of RL computations. We describe how RLlib meets this requirement in Chapter 3 and 4.

2. **Deploy-time overheads**: One of the major obstacles for the practicality of learning-based approaches is the deploy-time overhead. Whereas a heuristic may typically involve a trivial amount of computation (e.g., to decide which machine to schedule a task on), a deep neural network may both require expensive inference hardware and take a large amount of data as input. There are several avenues of attack here; one promising class of approaches is applying reinforcement learning to *produce a static artifact*. For example, in the packet classification problem (Chapter 5), instead of training a neural network to classify packets, we instead train a neural network to generate an optimized data structure that can in turn classify packets, eliminating deploy-time overhead as a concern.

3. **Train-time overheads**: One advantage of a heuristic is that you don't need to train it. In contrast, RL agents can take thousands to millions of experiences to reach an optimized policy. For some classes of problems this isn't an issue, but for real-world systems that are difficult to simulate, the overhead can be prohibitive. For example, consider a use of case of optimizing "big data" queries. Running a single query, never mind millions, can cost hundreds to thousands of dollars, especially if the generated execution plan isn't well optimized. In Chapter 6, we investigate an approach that can be used for such scenario: using deep unsupervised learning to model the environment (i.e., data record cardinalities), which enables fast runtime simulation (i.e., estimated execution costs). More generally, *model-based reinforcement learning* seeks to learn general purpose models of the environment to accelerated RL training.

4. **Robustness to new scenarios**: Finally, one weakness of learned approaches is the often lack of generalization to new scenarios not seen in the training data. Safe gen-

eralization is of course its own sub-field in RL. There can be case-specific mitigations, for example, in scenarios like packet classification (Chapter 5) we can limit the space of generated data structures to only provably correct ones (i.e., the agent can't make a correctness mistake, only reduce its performance). Approaches such as deep unsupervised learning learn an unbiased model of the data from which information can be extracted for new scenarios without the typical generalization problem. In Chapter 6, we specifically investigate the high-quantile performance of the model on unseen queries, showing robustness beyond those provided by heuristic approaches. However, generalization concerns can still apply to agents trained on the model.

To tie things together, we observe that RL for system requires a great deal of experimentation. In this way systems for RL can accelerate the work of RL for systems. Conversely, the requirements that surface from creative RL approaches to problems inform the design of better systems for RL. Over the last few years, RLlib's design and requirements have evolved via feedback from users across dozens of different academic labs and industry groups, each often raising novel use cases.

Chapter 3

Primitives for Distributed Reinforcement Learning

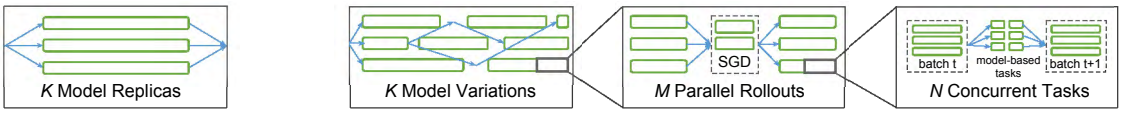
Reinforcement learning (RL) algorithms involve the deep nesting of highly irregular computation patterns, each of which typically exhibits opportunities for distributed computation. In this chapter, we examine how these computation patterns can be implemented in a flexible programming model based on top-down hierarchical control. This method of writing algorithms encapsulates the parallelism and resource requirements of RL computations within short-running compute tasks. We demonstrate the benefits of this principle through RLlib: a library that provides scalable software primitives for RL. These primitives enable a broad range of algorithms to be implemented with high performance, scalability, and substantial code reuse. RLlib is available as part of the open source Ray project [1]. This is the first of two chapters on RLlib; Chapter 4 focuses on higher-level abstractions for composing more complex training workflows, whereas this focuses on the core distributed primitives required for enabling both basic and more complex computation patterns. The interested reader may also want to refer to the Ray paper [107] and Ray ownership paper [159] for further details on the implementation considerations of the system architecture described here.

3.1 Introduction

Advances in parallel computing and composition through symbolic differentiation have been fundamental to the recent success of deep learning. Today, there are a wide range of deep learning frameworks [123, 1, 20, 68] that enable rapid innovation in neural network design and facilitate training at the scale necessary for progress in the field.

In contrast, while the reinforcement learning community enjoys the advances in systems and abstractions for deep learning, there has been comparatively less progress in the design of systems and abstractions that directly target reinforcement learning. Nonetheless, many of the challenges in reinforcement learning stem from the need to scale learning and

[1] RLlib documentation can be found at http://rllib.io



(a) Deep Learning (b) Reinforcement Learning

Figure 3.1: In contrast with deep learning, RL algorithms leverage parallelism at multiple levels and physical devices. Here, we show an RL algorithm composing derivative-free optimization, policy evaluation, gradient-based optimization, and model-based planning (Table 3.2).

simulation while also integrating a rapidly increasing range of algorithms and models. As a consequence, there is a fundamental need for composable parallel primitives to support research in reinforcement learning.

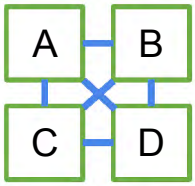
In the absence of a single dominant computational pattern (e.g., tensor algebra) or fundamental rules of composition (e.g., symbolic differentiation), the design and implementation of reinforcement learning algorithms can often be cumbersome, requiring RL researchers to directly reason about complex nested parallelism. Unlike typical operators in deep learning frameworks, individual components may require parallelism across a cluster (e.g., for rollouts), leverage neural networks implemented by deep learning frameworks, recursively invoke other components (e.g., model-based subtasks), or interface with black-box third-party simulators. In essence, the heterogeneous and distributed nature of many of these components poses a key challenge to reasoning about their parallel composition. Meanwhile, the main algorithms that connect these components are rapidly evolving and expose opportunities for parallelism at varying levels. Finally, RL algorithms manipulate substantial amounts of state (e.g., replay buffers and model parameters) that must be managed across multiple levels of parallelism and physical devices.

The substantial recent progress in RL algorithms and applications has resulted in a large and growing number of RL libraries [16, 32, 55, 59, 74, 131]. While some of these are highly scalable, few enable the composition of components at scale. In large part, this is due to the fact that many of the frameworks used by these libraries rely on communication between long-running program replicas for distributed execution; e.g., MPI [49], Distributed TensorFlow [1], and parameter servers [88]). As this programming model ignores component boundaries, it does not naturally encapsulate parallelism and resource requirements within individual components.[2] As a result, reusing these distributed components requires the insertion of appropriate control points in the program, a burdensome and error-prone process (Section 3.2). The absence of usable encapsulation hinders code reuse and leads to error prone reimplementation of mathematically complex and often highly stochastic algorithms. Even worse, in the distributed setting, often large parts of the distributed communication and execution must also be reimplemented with each new RL algorithm.

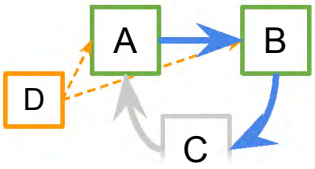
[2]By *encapsulation*, we mean that individual components specify their own internal parallelism and resources requirements and can be used by other components that have no knowledge of these requirements.

We believe that the ability to build scalable RL algorithms by composing and reusing existing components and implementations is essential for the rapid development and progress of the field.[3] Toward this end, we argue for structuring distributed RL components around the principles of logically centralized program control and parallelism encapsulation [47, 118]. We built RLlib using these principles, and as a result were not only able to implement a broad range of state-of-the-art RL algorithms, but also to pull out scalable primitives that can be used to easily compose new algorithms.

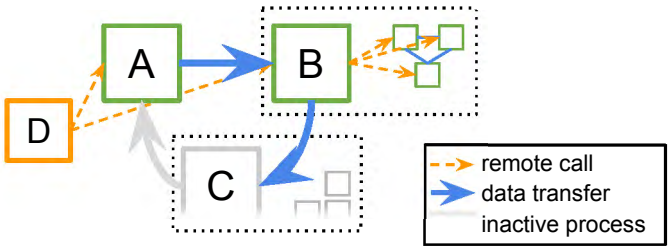
Irregularity of RL training workloads



(a) Distributed Control



(b) Logically Centralized Control



(c) Hierarchical Control

Figure 3.2: Most RL algorithms today are written in a fully distributed style (a) where replicated processes independently compute and coordinate with each other according to their roles (if any). We propose a hierarchical control model (c), which extends (b) to support nesting in RL and hyperparameter tuning workloads, simplifying and unifying the programming models used for implementation.

Modern RL algorithms are highly irregular in the computation patterns they create (Table 3.1), pushing the boundaries of computation models supported by popular distribution frameworks. This irregularity occurs at several levels:

1. The duration and resource requirements of tasks differ by orders of magnitude depending on the algorithm; e.g., A3C [105] updates may take milliseconds, but other algorithms like PPO [134] batch rollouts into much larger granularities.

[3] We note that composability *without* scalability can trivially be achieved with a single-threaded library and that all of the difficulty lies in achieving these two objectives simultaneously.

2. Communication patterns vary, from synchronous to asynchronous gradient-based optimization, to having several types of asynchronous tasks in high-throughput off-policy algorithms such as Ape-X and IMPALA [63, 36].

3. Nested computations are generated by model-based hybrid algorithms (Table 3.2), hyperparameter tuning in conjunction with RL or DL training, or the combination of derivative-free and gradient-based optimization within a single algorithm [138].

4. RL algorithms often need to maintain and update substantial amounts of state including policy parameters, replay buffers, and even external simulators.

Table 3.1: RL spans a broad range of computational demand.

Dimension	DQN/Laptop	IMPALA+PBT/Cluster
Task Duration	∼1ms	minutes
Task Compute	1 CPU	several CPUs and GPUs
Total Compute	1 CPU	hundreds of CPUs and GPUs
Nesting Depth	1 level	3+ levels
Process Memory	megabytes	hundreds of gigabytes
Execution	synchronous	async. and highly concurrent

As a consequence, the developers have no choice but to use a hodgepodge of frameworks to implement their algorithms, including parameter servers, collective communication primitives in MPI-like frameworks, task queues, etc. For more complex algorithms, it is common to build custom distributed systems in which processes independently compute and coordinate among themselves with no central control (Figure 3.2a). While this approach can achieve high performance, the cost to develop and evaluate is large, not only due to the need to implement and debug distributed programs, but because composing these algorithms further complicates their implementation (Figure 3.3). Moreover, today's computation frameworks (e.g., Spark [175], MPI) typically assume regular computation patterns and have difficulty when sub-tasks have varying durations, resource requirements, or nesting.

Logically centralized control for distributed RL

It is desirable for a single programming model to capture all the requirements of RL training. This can be done without eschewing high-level frameworks that structure the computation. Our key insight is that for each distributed RL algorithm, an equivalent algorithm can be written that exhibits logically centralized program control (Figure 3.2b). That is, instead of having independently executing processes (**A**, **B**, **C**, **D** in Figure 3.2a) coordinate among themselves (e.g., through RPCs, shared memory, parameter servers, or collective communication), a single *driver program* (**D** in Figure 3.2b and 3.2c) can delegate algorithm sub-tasks to other processes to execute in parallel. In this paradigm, the worker processes **A**, **B**, and **C**

passively hold state (e.g., policy or simulator state) but execute no computations until called by **D**. To support nested computations, we propose extending the centralized control model with *hierarchical delegation of control* (Figure 3.2c), which allows the worker processes (e.g., **B**, **C**) to further delegate work (e.g., simulations, gradient computation) to sub-workers of their own when executing tasks.

Building on such a logically centralized and hierarchical control model has several important advantages. First, the equivalent algorithm is often easier to implement in practice, since the distributed control logic is entirely encapsulated in a single process rather than multiple processes executing concurrently. Second, the separation of algorithm components into sub-routines (e.g., do rollouts, compute gradients with respect to some policy loss), enables code reuse across different execution patterns. Sub-tasks that have different resource requirements (e.g., CPUs vs GPUs) can be placed on different machines, reducing compute costs as we show in Section 6.5. Finally, distributed algorithms written in this model can be seamlessly nested within each other, satisfying the parallelism encapsulation principle.

Logically centralized control models can be highly performant, our proposed hierarchical variant even more so. This is because the bulk of data transfer (blue arrows in Figure 3.2) between processes happens out of band of the driver, not passing through any central bottleneck. In fact many highly scalable distributed systems [175, 18, 26] leverage centralized control in their design. Within a single differentiable tensor graph, frameworks like TensorFlow also implement logically centralized scheduling of tensor computations onto available physical devices. Our proposal extends this principle into the broader ML systems design space.

The contributions of RLlib are as follows.

1. We propose a general and composable hierarchical programming model for RL training (Section 3.2).

2. We describe RLlib, our highly scalable RL library, and how it builds on the proposed model to provide scalable abstractions for a broad range of RL algorithms, enabling rapid development (Section 3.3).

3. We discuss how performance is achieved within the proposed model (Section 3.4), and show that RLlib meets or exceeds state-of-the-art performance for a wide variety of RL workloads (Section 6.5).

3.2 Hierarchical Parallel Task Model

As highlighted in Figure 3.3, parallelization of entire programs using frameworks like MPI [49] and Distributed Tensorflow [1] typically require explicit algorithm modifications to insert points of coordination when trying to compose two programs or components together. This limits the ability to rapidly prototype novel distributed RL applications. Though the example in Figure 3.3 is simple, new hyperparameter tuning algorithms for long-running training

tasks; e.g., HyperBand, Population Based Training (PBT) [87, 66] increasingly demand fine-grained control over training.

We propose building RL libraries with hierarchical and logically centralized control on top of flexible task-based programming models like Ray [107]. Task-based systems allow subroutines to be scheduled and executed asynchronously on worker processes, on a fine-grained basis, and for results to be retrieved or passed between processes.

Relation to existing distributed ML abstractions

Though typically formulated for distributed control, abstractions such as parameter servers and collective communication operations can also be used within a logically centralized control model. As an example, RLlib uses allreduce and parameter-servers in some of its policy optimizers (Figure 3.4), and we evaluate their performance in Section 6.5.

Ray implementation of hierarchical control

We note that, within a single machine, the proposed programming model can be implemented simply with thread-pools and shared memory, though it is desirable for the underlying framework to scale to larger clusters if needed.

We chose to build RLlib on top of the Ray framework, which allows Python tasks to be distributed across large clusters. Ray's distributed scheduler is a natural fit for the hierarchical control model, as nested computation can be implemented in Ray with no central task scheduling bottleneck.

To implement a logically centralized control model, it is first necessary to have a mechanism to launch new processes and schedule tasks on them. Ray meets this requirement with *Ray actors*, which are Python classes that may be created in the cluster and accept remote method calls (i.e., tasks). Ray permits these actors to in turn launch more actors and schedule tasks on those actors as part of a method call, satisfying our need for hierarchical delegation as well.

For performance, Ray provides standard communication primitives such as `aggregate` and `broadcast`, and critically enables the *zero-copy* sharing of large data objects through a shared memory object store. As shown in Section 6.5, this enables the performance of RLlib algorithms. We further discuss framework performance in Section 3.4.

3.3 Abstractions for Reinforcement Learning

To leverage RLlib for distributed execution, algorithms must declare their policy π, experience postprocessor ρ, and loss L. These can be specified in any deep learning framework, including TensorFlow and PyTorch. RLlib provides *rollout workers* and *policy optimizers* that implement strategies for distributed policy evaluation and training.

```
if mpi.get_rank() <= m:
    grid = mpi.comm_world.split(0)
else:
    eval = mpi.comm_world.split(
        mpi.get_rank() % n)
...
if mpi.get_rank() == 0:
    grid.scatter(
        generate_hyperparams(), root=0)
    print(grid.gather(root=0))
elif 0 < mpi.get_rank() <= m:
    params = grid.scatter(None, root=0)
    eval.bcast(
        generate_model(params), root=0)
    results = eval.gather(
         result, root=0)
    grid.gather(results, root=0)
elif mpi.get_rank() > m:
    model = eval.bcast(None, root=0)
    result = rollout(model)
    eval.gather(result, root=0)
```

(a) Distributed Control

```
@ray.remote
def rollout(model):
    # perform a rollout and
    # return the result

@ray.remote
def evaluate(params):
    model = generate_model(params)
    results = [rollout.remote(model)
       for i in range(n)]
    return results

param_grid = generate_hyperparams()
print(ray.get([evaluate.remote(p)
   for p in param_grid]))
```

(b) Hierarchical Control

Figure 3.3: Composing a distributed hyperparameter search with a function that also requires distributed computation involves *complex nested parallel computation patterns*. With MPI (a), a new program must be written from scratch that mixes elements of both. With hierarchical control (b), components can remain unchanged and simply be invoked as remote tasks.

Defining the Policy

RLlib's abstractions are as follows. The developer specifies a policy model π that maps the current observation o_t and (optional) RNN hidden state h_t to an action a_t and the next RNN state h_{t+1}. Any number of user-defined values y_t^i (e.g., value predictions, TD error) can also be returned:

$$\pi_\theta(o_t, h_t) \Rightarrow (a_t, h_{t+1}, y_t^1 \dots y_t^N) \tag{3.1}$$

Most algorithms will also specify a trajectory postprocessor ρ that transforms a batch $X_{t,K}$ of K $\{(o_t, h_t, a_t, h_{t+1}, y_t^1 \dots y_t^N, r_t, o_{t+1})\}$ tuples starting at t. Here r_t and o_{t+1} are the reward and new observation after taking an action. Example uses include advantage estimation [133] and goal relabeling [5]. To also support multi-agent environments, experience batches $X_{t,K}^p$ from the P other agents in the environment are also made accessible:

$$\rho_\theta(X_{t,K}, X_{t,K}^1 \dots X_{t,K}^P) \Rightarrow X_{post} \tag{3.2}$$

Gradient-based algorithms define a combined loss L that can be descended to improve the policy and auxiliary networks:

$$L(\theta; X) \Rightarrow loss \tag{3.3}$$

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
avg_grad = aggregate(grads)
local_graph.apply(avg_grad)
weights = broadcast(
    local_graph.weights())
for ev in evaluators:
    ev.set_weights(weights)
```

(a) Allreduce

```
samples = concat([ev.sample()
    for ev in evaluators])
pin_in_local_gpu_memory(samples)
for _ in range(NUM_SGD_EPOCHS):
   local_g.apply(local_g.grad(samples)
weights = broadcast(local_g.weights())
for ev in evaluators:
    ev.set_weights(weights)
```

(b) Local Multi-GPU

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
    grad, ev, grads = wait(grads)
    local_graph.apply(grad)
    ev.set_weights(
        local_graph.get_weights())
    grads.append(ev.grad(ev.sample()))
```

(c) Asynchronous

```
grads = [ev.grad(ev.sample())
    for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
    grad, ev, grads = wait(grads)
    for ps, g in split(grad, ps_shards):
        ps.push(g)
    ev.set_weights(concat(
        [ps.pull() for ps in ps_shards])
    grads.append(ev.grad(ev.sample()))
```

(d) Sharded Param-server

Figure 3.4: Pseudocode for four RLlib policy optimizer step methods. Each step() operates over a local policy and array of remote evaluator replicas. Ray remote calls are highlighted in orange; other Ray primitives in blue (Section 3.4). *Apply* is shorthand for updating weights. Minibatch code and helper functions omitted. The param server optimizer in RLlib also implements pipelining not shown here.

Finally, the developer can also specify any number of utility functions u^i to be called as needed during training to, e.g., return training statistics s, update target networks, or adjust annealing schedules:

$$u^1 \dots u^M(\theta) \Rightarrow (s, \theta_{update}) \tag{3.4}$$

To interface with RLlib, these algorithm functions should be defined in a *policy* class with the following methods:

```
abstract class rllib.Policy:
    def act(self, obs, h): action, h, y*
    def postprocess(self, batch, b*): batch
    def gradients(self, batch): grads
    def get_weights; def set_weights;
    def u*(self, args*)
```

Policy Evaluation

For collecting experiences, RLlib provides a `RolloutWorker` class that wraps a policy and environment to add a method to `sample()` experience batches. Rollout worker instances can be created as Ray remote actors and *replicated* across a cluster for parallelism. To make their usage concrete, consider a minimal TensorFlow policy gradients implementation that extends the `rllib.TFPolicy` helper template:

```
class PolicyGradient(TFPolicy):
    def __init__(self, obs_space, act_space):
        self.obs, self.advantages = ...
        pi = FullyConnectedNetwork(self.obs)
        dist = rllib.action_dist(act_space, pi)
        self.act = dist.sample()
        self.loss = -tf.reduce_mean(
            dist.logp(self.act) * self.advantages)

    def postprocess(self, batch):
        return rllib.compute_advantages(batch)
```

From this policy definition, the developer can create a number of policy evaluator replicas `ev` and call `worker.sample.remote()` on each to collect experiences in parallel from environments. RLlib supports OpenAI Gym [13], user-defined environments, and also batched simulators such as ELF [150]:

```
workers = [rllib.RolloutWorker.remote(env=SomeEnv, policy=PolicyGradient)
    for _ in range(10)]
print(ray.get([worker.sample.remote() for worker in workers]))
```

Policy Optimization

RLlib separates the implementation of algorithms into the declaration of the algorithm-specific *policy* and the choice of an algorithm-independent *policy optimizer*. The policy optimizer is responsible for the performance-critical tasks of distributed sampling, parameter updates, and managing replay buffers. To distribute the computation, the optimizer operates over a set of rollout worker replicas.

To complete the example, the developer chooses a policy optimizer and creates it with references to existing rollout workers. The async optimizer uses the worker actors to compute gradients in parallel on many CPUs (Figure 3.4c). Each `optimizer.step()` runs a round of remote tasks to improve the model. Between steps, policy replicas can be queried directly, e.g., to print out training statistics:

Table 3.2: RLlib's policy optimizers and rollout workers capture common components (Rollout, Replay, Gradient-based Optimizer) within a logically centralized control model, and leverages Ray's hierarchical task model to support other distributed components.

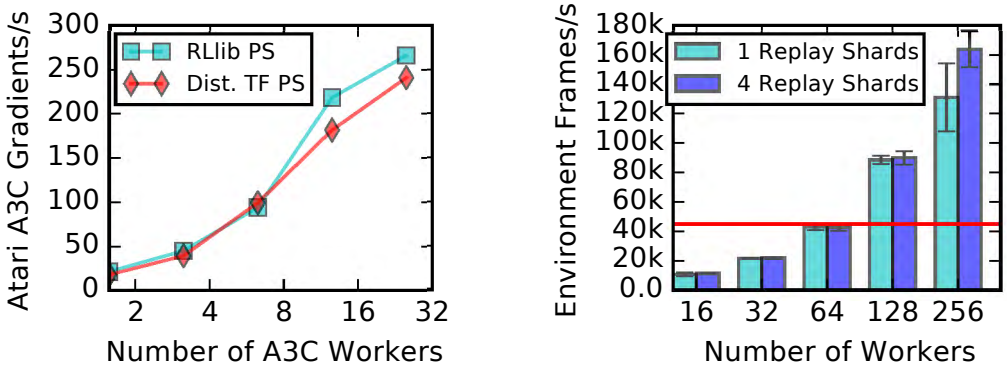
Algorithm Family	Rollout	Replay	Grad. Opt.	Other Distributed Components
DQNs	X	X	X	
Policy Gradient	X		X	
Off-policy PG	X	X	X	
Model-Based/Hybrid	X		X	Model-Based Planning
Multi-Agent	X	X	X	
Evolutionary Methods	X			Derivative-Free Optimization
AlphaGo	X	X	X	MCTS, Derivative-Free Optimization

```
optimizer = rllib.AsyncPolicyOptimizer(
    policy=PolicyGradient, workers=rollout_workers)
while True:
    optimizer.step()
    print(optimizer.foreach_policy(
        lambda p: p.get_train_stats()))
```

Policy optimizers extend the well-known gradient-descent optimizer abstraction to the RL domain. A typical gradient-descent optimizer implements $step(L(\theta), X, \theta) \Rightarrow \theta_{opt}$. RLlib's policy optimizers instead operate over the local policy G and a set of remote evaluator replicas, i.e., $step(G, ev_1 \ldots ev_n, \theta) \Rightarrow \theta_{opt}$, capturing the sampling phase of RL as part of optimization (i.e., calling `sample()` on rollout worker to produce new simulation data).

The policy optimizer abstraction has the following advantages. By separating execution strategy from policy and loss definitions, specialized optimizers can be swapped in to take advantage of available hardware or algorithm features without needing to change the rest of the algorithm. The policy class encapsulates interaction with the deep learning framework, allowing algorithm authors to avoid mixing distributed systems code with numerical computations, and enabling optimizer implementations to be improved and reused across different deep learning frameworks.

As shown in Figure 3.4, by leveraging centralized control, policy optimizers succinctly capture a broad range of choices in RL optimization: synchronous vs asynchronous, allreduce vs parameter server, and use of GPUs vs CPUs. RLlib's policy optimizers provide performance comparable to optimized parameter server (Figure 3.5a) and MPI-based implementations (Section 6.5). Pulling out this optimizer abstraction is easy in a logically centralized control model since each policy optimizer has full control over the distributed computation it implements.



(a) Sharded Param. Server (b) Ape-X in RLlib

Figure 3.5: RLlib's centrally controlled policy optimizers match or exceed the performance of implementations in specialized systems. The RLlib parameter server optimizer using 8 internal shards is competitive with a Distributed TensorFlow implementation tested in similar conditions. RLlib's Ape-X policy optimizer scales to 160k frames per second with 256 workers at a frameskip of 4, more than matching a reference throughput of ∼45k fps at 256 workers, demonstrating that a single-threaded Python controller can efficiently scale to high throughputs.

Completeness and Generality of Abstractions

We demonstrate the completeness of RLlib's abstractions by formulating the algorithm families listed in Table 3.2 within the API. When applicable, we also describe the concrete implementation in RLlib:

DQNs: DQNs use y^1 for storing TD error, implement n-step return calculation in ρ_θ, and the Q loss in L. Target updates are implemented in u^1, and setting the exploration ϵ in the user-defined method u^2.

DQN implementation: To support experience replay, RLlib's DQN uses a policy optimizer that saves collected samples in an embedded replay buffer. The user can alternatively use an asynchronous optimizer (Figure 3.4c). The target network is updated by calls to u^1 between optimizer steps.

Ape-X implementation: Ape-X [63] is a variation of DQN that leverages distributed experience prioritization to scale to many hundreds of cores. To adapt our DQN implementation, we created rollout workers with a distribution of ϵ values, and wrote a new high-throughput policy optimizer (∼200 lines) that pipelines the sampling and transfer of data between replay buffer actors using Ray primitives. Our implementation scales nearly linearly up to 160k environment frames per second with 256 workers (Figure 3.5b).

Policy Gradient / Off-policy PG: These algorithms store value predictions in y^1, implement advantage estimation using ρ_θ, and combine actor and critic losses in L.

PPO implementation: Since PPO's loss function permits multiple SGD passes over sample data, when there is sufficient GPU memory RLlib chooses a GPU-optimized policy optimizer (Figure 3.4b) that pins data into local GPU memory. In each iteration, the optimizer collects samples from evaluator replicas, performs multi-GPU optimization locally, and then broadcasts the new model weights.

A3C implementation: RLlib's A3C can use either the asynchronous (Figure 3.4c) or sharded parameter server policy optimizer (3.4d). These optimizers collect gradients from the rollout workers to update the canonical copy of θ.

DDPG implementation: RLlib's DDPG uses the same replay policy optimizer as DQN. L includes both actor and critic losses. The user can also choose to use the Ape-X policy optimizer with DDPG.

Model-based / Hybrid: Model-based RL algorithms extend $\pi_\theta(o_t, h_t)$ to make decisions based on model rollouts, which can be parallelized using Ray. To update their environment models, the model loss can either be bundled with L, or the model trained separately (i.e., in parallel using Ray primitives) and its weights periodically updated via u^1.

Multi-Agent: Rollout workers can run multiple policies at once in the same environment, producing batches of experience for each agent. Many multi-agent algorithms use a centralized critic or value function, which we support by allowing ρ_θ to collate experiences from multiple agents.

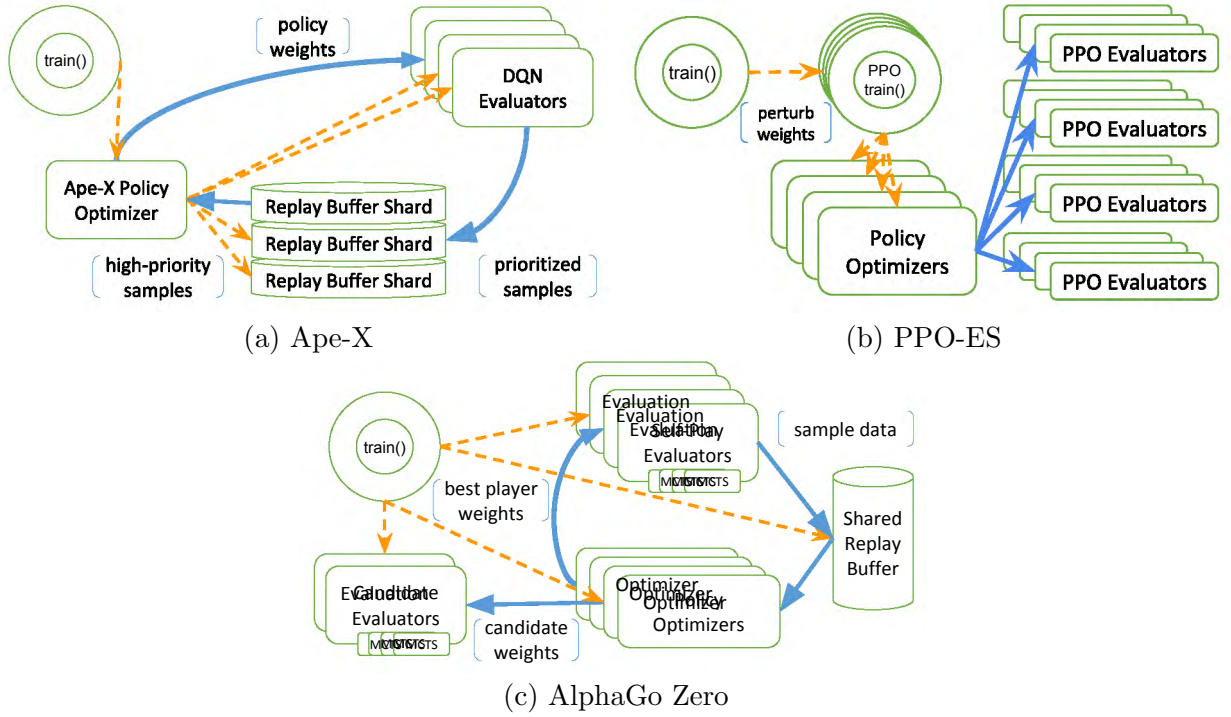
Evolutionary Methods: Derivative-free methods can be supported through non-gradient-based policy optimizers.

Evolution Strategies (ES) implementation: ES is a derivative-free optimization algorithm that scales well to clusters with thousands of CPUs. We were able to port a single-threaded implementation of ES to RLlib with only a few changes, and further scale it with an aggregation tree of actors (Figure 3.8a), suggesting that the hierarchical control model is both flexible and easy to adapt algorithms for.

PPO-ES experiment: We studied a hybrid algorithm that runs PPO updates in the inner loop of an ES optimization step that randomly perturbs the PPO models. The implementation took only $\sim$50 lines of code and did not require changes to PPO, showing the value of encapsulating parallelism. In our experiments, PPO-ES converged faster and to a higher reward than PPO on the Walker2d-v1 task. A similarly modified A3C-ES implementation solved PongDeterministic-v4 in 30% less time.

AlphaGo: We sketch how to scalably implement the AlphaGo Zero algorithm using a combination of Ray and RLlib abstractions.

1. *Logically centralized control of multiple distributed components*: AlphaGo Zero uses multiple distributed components: model optimizers, self-play rollout workers, candidate model evaluators, and the shared replay buffer. These components are manageable



(a) Ape-X (b) PPO-ES

(c) AlphaGo Zero

Figure 3.6: Complex RL architectures are easily captured within RLlib's hierarchical control model. Here blue lines denote data transfers, orange lines lighter overhead method calls. Each train() call encompasses a batch of remote calls between components.

as Ray actors under a top-level AlphaGo policy optimizer. Each optimizer step loops over actor statuses to process new results, routing data between actors and launching new actor instances.

2. *Shared replay buffer*: AlphaGo Zero stores the experiences from self-play evaluator instances in a shared replay buffer. This requires routing game results to the shared buffer, which is easily done by passing the result object references from actor to actor.

3. *Best player*: AlphaGo Zero tracks the current best model and only populates its replay buffer with self-play from that model. Candidate models must achieve a $\geq 55\%$ victory margin to replace the best model. Implementing this amounts to adding an `if` block in the main control loop.

4. *Monte-Carlo tree search*: MCTS (i.e., model-based planning) can be handled as a subroutine of the policy, and optionally parallelized as well using Ray.

HyperBand and Population Based Training: Ray includes distributed implementations of hyperparameter search algorithms such as HyperBand and PBT [87, 66]. We were able to use these to evaluate RLlib algorithms, which are themselves distributed, with the

addition of ∼15 lines of code per algorithm. We note that these algorithms are non-trivial to integrate when using distributed control models due to the need to modify existing code to insert points of coordination (Figure 3.3). RLlib's use of short-running tasks avoids this problem, since control decisions can be easily made between tasks.

3.4 Framework Performance

In this section, we discuss properties of Ray [107] and other optimizations critical to RLlib.

Single-node performance

Stateful computation: Tasks can share mutable state with other tasks through Ray actors. This is critical for tasks that operate on and mutate stateful objects like third-party simulators or neural network weights.

Shared memory object store: RL workloads involve sharing large quantities of data (e.g., rollouts and neural network weights). Ray supports this by allowing data objects to be passed directly between workers without any central bottleneck. In Ray, workers on the same machine can also read data objects through shared memory without copies.

Vectorization: RLlib can batch policy evaluation to improve hardware utilization (Figure 3.7), supports batched environments, and passes experience data between actors efficiently in columnar array format.

Distributed performance

Lightweight tasks: Remote call overheads in Ray are on the order of ∼200s when scheduled on the same machine. When machine resources are saturated, tasks spill over to other nodes, increasing latencies to around ∼1ms. This enables parallel algorithms to scale seamlessly to multiple machines while preserving high single-node throughput.

Nested parallelism: Building RL algorithms by composing distributed components creates multiple levels of nested parallel calls (Figure 3.1). Since components make decisions that may affect downstream calls, the call graph is also inherently dynamic. Ray supports this by allowing any Python function or class method to be invoked remotely as a lightweight task. For example, `func.remote()` executes `func` remotely and immediately returns a placeholder result which can later be retrieved or passed to other tasks.

Resource awareness: Ray allows remote calls to specify resource requirements and utilizes a resource-aware scheduler to preserve component performance. Without this, distributed components can improperly allocate resources, causing algorithms to run inefficiently or fail.

Fault tolerance and straggler mitigation: Failure events become significant at scale [10]. RLlib leverages Ray's built-in fault tolerance mechanisms [107], reducing costs with preemptible cloud compute instances [4, 45]. Similarly, stragglers can significantly impact the performance of distributed algorithms at scale [25]. RLlib supports straggler mitigation in a generic way via the `ray.wait()` primitive. For example, in PPO we use this to drop the slowest evaluator tasks, at the cost of some bias.

Data compression: RLlib uses the LZ4 algorithm to compress experience batches. For image observations, LZ4 reduces network traffic and memory usage by more than an order of magnitude, at a compression rate of ∼1 GB/s/core.

3.5 Evaluation

Sampling efficiency: Policy evaluation is an important building block for all RL algorithms. In Figure 3.7 we benchmark the scalability of gathering samples from rollout worker actors. To avoid bottlenecks, we use four intermediate actors for aggregation. Pendulum-CPU reaches over 1.5 million actions/s running a small 64×64 fully connected network as the policy. Pong-GPU nears 200k actions/s on the DQN convolutional architecture [106].

Large-scale tests: We evaluate the performance of RLlib on Evolution Strategies (ES), Proximal Policy Optimization (PPO), and A3C, comparing against specialized systems built *specifically for those algorithms* [114, 59, 116] using Redis, OpenMPI, and Distributed TensorFlow. The same hyperparameters were used in all experiments. We used TensorFlow to define neural networks for the RLlib algorithms evaluated.

RLlib's ES implementation scales well on the Humanoid-v1 task to 8192 cores using AWS m4.16xl CPU instances [3]. With 8192 cores, we achieve a reward of 6000 in a median time of 3.7 minutes, which is over twice as fast as the best published result [129]. For PPO we evaluate on the same Humanoid-v1 task, starting with one p2.16xl GPU instance and adding m4.16xl instances to scale. This cost-efficient local policy optimizer (Table 3.3) outperformed the reference MPI implementation that required multiple expensive GPU instances to scale.

We ran RLlib's A3C on an x1.16xl machine and solved the PongDeterministic-v4 environment in 12 minutes using an asynchronous policy optimizer and 9 minutes using a sharded param-server optimizer, which matches the performance of a well-tuned baseline [116].

Multi-GPU: To better understand RLlib's advantage in the PPO experiment, we ran benchmarks on a p2.16xl instance comparing RLlib's local multi-GPU policy optimizer with

one using an allreduce in Table 3.3. The fact that different strategies perform better under different conditions suggests that policy optimizers are a useful abstraction.

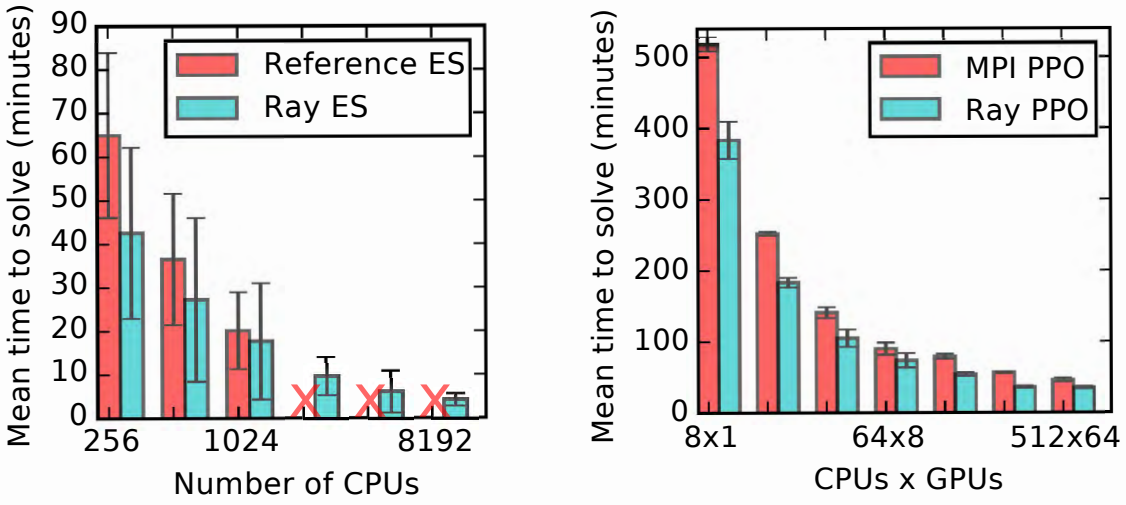
Policy Optimizer	Gradients computed on	Environment	SGD throughput
Allreduce-based	4 GPUs, Workers	Humanoid-v1	330k samples/s
		Pong-v0	23k samples/s
	16 GPUs, Workers	Humanoid-v1	**440k samples/s**
		Pong-v0	**100k samples/s**
Local Multi-GPU	4 GPUs, Driver	Humanoid-v1	**2.1M samples/s**
		Pong-v0	N/A (out of mem.)
	16 GPUs, Driver	Humanoid-v1	1.7M samples/s
		Pong-v0	**150k samples/s**

Table 3.3: A specialized multi-GPU policy optimizer outperforms distributed allreduce when data can fit entirely into GPU memory. This experiment was done for PPO with 64 Evaluator processes. The PPO batch size was 320k, The SGD batch size was 32k, and we used 20 SGD passes per PPO batch.

Figure 3.7: Policy evaluation throughput scales nearly linearly from 1 to 128 cores. PongNoFrameskip-v4 on GPU scales from 2.4k to ∼200k actions/s, and Pendulum-v0 on CPU from 15k to 1.5M actions/s. We use a single p3.16xl AWS instance to evaluate from 1-16 cores, and a cluster of four p3.16xl instances from 32-128 cores, spreading actors evenly across the cluster. Rollout workers (evaluators) compute actions for 64 agents at a time, and share the GPUs on the machine.

3.6 Related Work

There are many reinforcement learning libraries [16, 32, 55, 59, 74, 131]. These often scale by creating long-running program replicas that each participate in coordinating the distributed computation as a whole, and as a result do not generalize well to complex architectures.



(a) Evolution Strategies

(b) Proximal Policy Optimization

Figure 3.8: The time required to achieve a reward of 6000 on the Humanoid-v1 task. RLlib implementations of ES and PPO outperform highly optimized reference optimizations.

RLlib instead uses a hierarchical control model with short-running tasks to let each component control its own distributed execution, enabling higher-level abstractions such as policy optimizers to be used for composing and scaling RL algorithms.

Outside of reinforcement learning, there has been a strong effort to explore composition and integration between different deep learning frameworks. ONNX [103], NNVM [29], and Gluon [44] sit between model specifications and hardware to provide cross-library optimizations. Deep learning libraries [123, 1, 20, 68] provide support for the gradient-based optimization components that appear in RL algorithms.

3.7 Conclusion

RLlib is an open source library for reinforcement learning that leverages fine-grained nested parallelism to achieve state-of-the-art performance across a broad range of RL workloads. It offers both a collection of reference algorithms and scalable abstractions for easily composing new ones.

Chapter 4

Distributed Reinforcement Learning as a Dataflow Problem

After reading Chapter 3, the reader should have an understanding of the distributed primitives required to support a broad range of RL algorithms at scale. However, the astute reader will have noticed that the critical *policy optimizer* abstraction still needs to be written directly on top of low-level system primitives. While built-in policy optimizers are sufficient for many use cases, practically speaking this means advanced users of RLlib may still need to work with low-level systems components to compose novel distributed algorithms. In this chapter, we re-examine the challenges posed by distributed RL and try to view it through the lens of an old idea: distributed dataflow. We show that viewing RL as a dataflow problem leads to highly composable and performant implementations. We propose RLlib flow, a hybrid actor-dataflow programming model for distributed RL, and validate its practicality by porting the full suite of algorithms in RLlib.

4.1 Introduction

The past few years have seen the rise of deep reinforcement learning (RL) as a new, powerful optimization method for solving sequential decision making problems. As with deep supervised learning, researchers and practitioners frequently leverage parallel computation, which has led to the development of numerous distributed RL algorithms and systems as the field rapidly evolves.

Despite the high-level of abstraction that RL algorithms are defined in (i.e., as a couple dozen lines of update equations), their implementations have remained quite low level (i.e., at the level of message passing). This is particularly true for *distributed* RL algorithms, which are typically implemented directly on low-level message passing systems or actor frameworks [60]. Libraries such as Acme [61], RLgraph [132], RLlib [92], and Coach [16] provide unified abstractions for defining single-agent RL algorithms, but their user-facing APIs only allow algorithms to execute within the bounds to predefined distributed execution patterns or

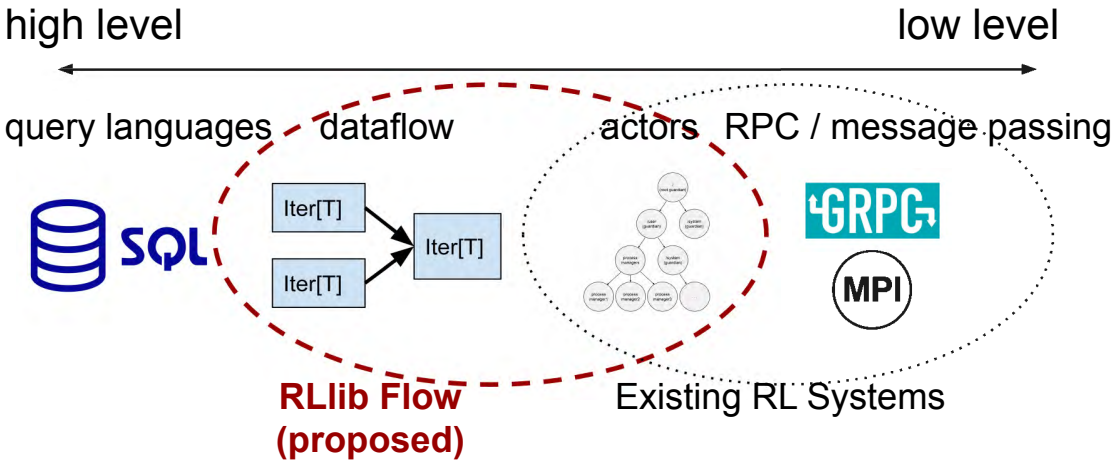


Figure 4.1: We propose RLlib flow, a hybrid actor-dataflow model for distributed RL. RLlib flow enables implementation of distributed RL algorithms in terms of their high-level dataflow.

"templates". While the aforementioned libraries have been highly successful at replicating a large number of novel RL algorithms introduced over the years, showing the generality of their underlying actor or graph-based computation models, the needs of many researchers and practitioners are often not met by their abstractions. As the authors of one such distributed RL library (RLlib), we have observed this firsthand from our users:

First, RL practitioners are typically not systems engineers. They are not well versed with code that mixes together the logical dataflow of the program and system concerns such as performance and bounding memory usage. This leads to a high barrier of entry for most RL users to experimenting with debugging existing distributed RL algorithms or authoring new distributed RL approaches.

Second, even when an RL practitioner is happy with a particular algorithm, they may wish to *customize* it in various ways. This is especially important given the diversity of RL tasks (e.g., single-agent, multi-agent, meta-learning). While many customizations within common RL environments can be anticipated and made available as configuration options (e.g., degree of parallelism, batch size), it is difficult for a library author to provide enough options to cover less common tasks that necessarily alter the distributed pattern of the algorithm (e.g., interleaved training of different distributed algorithms, different replay strategies).

Our experience is that when considering the needs of users considering novel RL applications and approaches, RL development requires a significant degree of programming flexibility. Advanced users want to tweak or add various distributed components (i.e., they need to write programs). In contrast to supervised learning, it is more difficult to provide a fixed set of abstractions for scaling RL training.

As a result, it is very common for RL researchers or practitioners to eschew existing infrastructure, either sticking to non-parallel approaches, which are inherently easier to understand and customize [17, 59], or writing their own distributed framework that fits their needs. The large number of RL frameworks in existence today is evidence of this, especially considering the number of these frameworks aiming to be "simpler" versions of other frameworks.

In this chapter, we re-examine the challenges posed by distributed RL in the light of these user requirements, drawing inspiration from prior work in the field of data processing and distributed dataflow. To meet these challenges, we propose RLlib flow, a hybrid actor-dataflow programming model for distributed RL. Like streaming data systems, RLlib flow provides a small set of operator-like primitives that can be composed to express distributed RL algorithms. Unlike data processing systems, RLlib flow explicitly exposes references to actor processes participating in the dataflow, permitting limited message passing between them in order to more simply meet the requirements of RL algorithms. The interaction of dataflow and actor messages is managed via special sequencing and concurrency operators.

We implement RLlib flow in RLlib and show through case studies how it can flexibly express different algorithms and meet diverse customization requirements. RLlib flow is released as open source in RLlib, enabling many new use cases and the deletion of thousands of lines of code.

We want to note that while we do cover some implementation details of RLlib flow, our focus is on *defining a general RL dataflow model* that could inform future RL library implementations. Our contributions are as follows: nosep

1. We examine the needs of distributed RL algorithms and RL practitioners from a dataflow perspective, identifying key challenges (Section 4.2 and 4.3).
2. We propose RLlib flow, a hybrid actor-dataflow programming model that can simply and efficiently express distributed RL algorithms (Section 4.4 and 5.5).
3. We show quantitatively that RLlib flow simplifies algorithm implementations in RLlib and reduces overheads found in data-oriented systems (Section 6.5).

4.2 Distributed Reinforcement Learning

We first discuss the relevant computational characteristics of distributed RL algorithms, starting with the common *single-agent training* scenario, where the goal is to optimize a single agent's performance in an environment, and then discuss the computational needs of emerging *multi-agent*, *model-based*, and *meta-learning* training patterns.

Single-Agent Training

Training a single RL agent—the most basic and common scenario—consists of applying the steps of rollout, replay, and optimization repeatedly until the policy reaches the desired

performance. Synchronous algorithms such as A2C and PPO apply the steps strictly sequentially. Parallelism may be leveraged internally within each step. Asynchronous algorithm variations such as Ape-X, A3C, APPO, and IMPALA pipeline and overlap the rollout and optimization steps asynchronously to hit higher data throughputs. Rate limiting [61] can be applied to control learning dynamics in the asynchronous setting.

Multi-Agent Training

In multi-agent training, there are multiple acting entities in the environment (e.g., cooperating or competing agents). While there is a rich literature on multi-agent algorithms, we note that the *dataflow structure* of multi-agent training is similar to that of single-agent—as long as all entities are being trained with the same algorithm and compatible hyperparameters. Intuitively, for a single algorithm, the changes necessary to support multi-agent consist of replicating the rollout, replay, and optimization steps per agent.

However, problems arise should it be required to customize the training of any of the agents in the environment. For example, in a two-agent environment, one agent may desire to be optimized at a higher frequency (i.e., smaller batch size). This fundamentally alters the training dataflow—there are now two iterative loops executing at different frequencies. Furthermore, if these agents are trained with entirely different algorithms, there is a need to compose two different distributed dataflows. We have seen several users of RLlib encounter both of these scenarios in their work.

Model-Based and Meta-Learning Algorithms

Model-based algorithms seek to learn the transition dynamics of the environment to improve the sample efficiency of training. This can be thought of as adding a supervised training step on top of standard distributed RL, where an ensemble of one or more dynamics models are trained from environment-generated data. Handling the data routing, replay, optimization, and stats collection for these models naturally adds complexity to the distributed dataflow graph, "breaking the mold" of standard model-free RL algorithms. Using RLlib flow, we have implemented two state of the art model-based algorithms: MB-MPO [23] and Dreamer [56]. Similarly, meta-learning algorithms such as MAML [40] leverage additional computation structures (i.e., nested inner optimization steps) as they seek to learn a policy amenable for quick adaptation to new environments.

A Case for a Higher Level Programming Model

Given that existing distributed RL algorithms are already implementable using low level actor and RPC primitives, it is worth questioning the value of defining a higher level computation model. Our experience is that RL is more like data analytics than supervised learning. Advanced users want to tweak or add various distributed components (i.e., they need to program), and there is no way to have a "one size fits all" (i.e., Estimator interface

• Example: synchronous dataflow (A2C), with updates

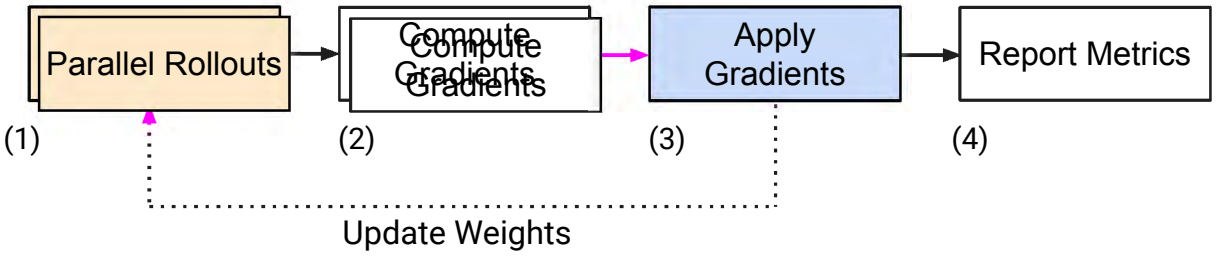


Figure 4.2: The dataflow of the A3C parallel algorithm. Each box is an operator or iterator from which data items can be pulled from. Here operators (1) and (2) represent parallel computations, but (3) and (4) are sequential. Black arrows denote synchronous data dependencies, pink arrows asynchronous dependencies, and dotted arrows actor method calls. Training metrics are pulled from the output operator ("Report Metrics"), which drives the computation.

from supervised learning). We believe that, beyond the ability to more concisely and cleanly capture *single-agent* RL algorithms, the computational needs of more advanced RL training patterns motivate higher level programming models like RLlib flow.

4.3 Reinforcement Learning vs Data Streaming

The key observation behind RLlib flow is that the dataflow graph of RL algorithms are quite similar to those of data streaming applications. Indeed, RL algorithms can be captured in general purpose dataflow models. However, due to several characteristics, they are not a perfect fit, even for dataflow models that support iterative computation.

In this section we examine the dataflow of the A3C algorithm (Figure 4.2) to compare and contrast RL with streaming dataflow. A3C starts with (1) parallel rollouts across many experiences. Policy gradients are computed in parallel based on rollouts in step (2). In step (3), the gradients are asynchronously gathered and applied on a central model, which is then used to update rollout worker weights. Importantly, each box or *operator* in this dataflow may be *stateful* (e.g., `ParallelRollouts` holds environment state as well as the current policy snapshot).

Similar to data processing topologies, A3C is applying a transformation to a data stream (of rollouts) in parallel (to compute gradients). This is denoted by the black arrow between (1) and (2). There is also a non-parallel transformation to produce metrics from the computation, denoted by the black arrow between (3) and (4) However, zooming out to look at the entire dataflow graph, a few differences emerge. There are asynchronous data dependencies (pink arrow) and method calls (dotted pink arrow):

Asynchronous Dependencies: RL algorithms often leverage asynchronous computation to reduce update latencies and eliminate stragglers [105]. In RLlib flow, we represent these

with a pink arrow between a parallel and sequential iterator. This means items will be fetched into the sequential iterator as soon as they are available, instead of in a deterministic ordering. The level of asynchrony can be configured to increase pipeline parallelism.

Message Passing: RL algorithms, like all iterative algorithms, need to update upstream operator state during execution (e.g., update policy weights). Unlike iterative algorithms, these updates may be fine-grained and asynchronous (i.e., update the parameters of a particular worker), as well as coarse-grained (i.e., update all workers at once after a global barrier). RLlib flow allows method calls (messages) to be sent to any actor in the dataflow. Ordering of messages in RLlib flow with respect to dataflow steps is guaranteed if synchronous data dependencies (black arrows) fully connect the sender to the receiver, providing *barrier semantics*.

Consistency and Durability: Unlike data streaming, which has strict requirements such as exactly-once processing of data [174], RL has less strict consistency and durability requirements. This is since on a fault, the entire computation can be restarted from the last checkpoint with minimal loss of work. Message or data loss can generally be tolerated without adverse affect on training. Individual operators can be restarted on failure, discarding any temporary state. This motivates a programming model that minimizes overhead (e.g., avoids state serialization and logging cost).

4.4 A Dataflow Model for Distributed RL

Here we formally define the RLlib flow hybrid actor-dataflow programming model. RLlib flow consists of a set of dataflow operators that produce and consume *distributed iterators* [46]. These distributed iterators can represent parallel streams of a data items `T` sharded across many actors (`ParIter[T]`), or a single sequential stream of items (`Iter[T]`). It is important to note that these iterators are *lazy*, they do not execute computation or produce items unless requested. This means that an entire RLlib flow execution graph is inactive until the user attempts to pull metrics from the output operator.

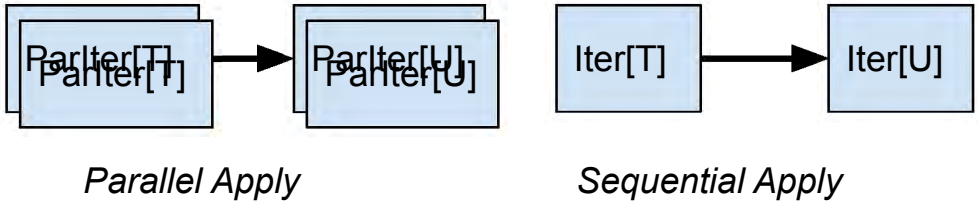
Creation and Message Passing: RLlib flow iterators are always created from an existing set of actor processes. In Figure 4.2, the iterator is created from a set of rollout workers that produce experience batches given their current policy. Also, any operator may send a message to any source actor (i.e., a rollout worker, or replay buffer) during its execution. In the A3C example, the update weights operation is a use of this facility. The order guarantees of these messages with respect to dataflow steps depends on the barrier semantics provided by *sequencing operators*. The sender may optionally block and await the reply of sent messages.

Transformation: As in any data processing system, the basic operation of data transformation is supported. Both parallel and sequential iterators can be transformed with the

```
create(Seq[SourceActor[T]]) -> ParIter[T]
send_msg(dest: Actor, msg: Any) -> Reply
```

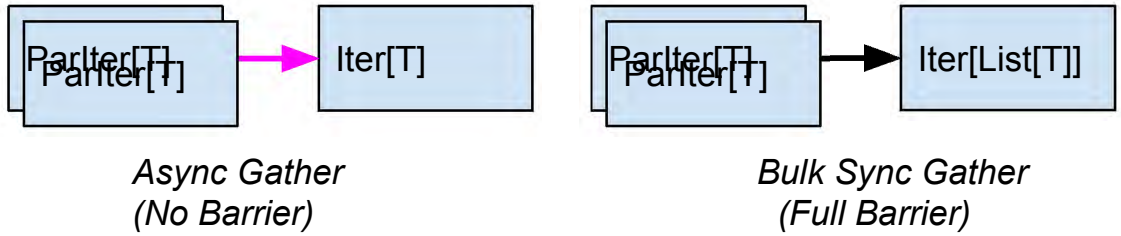
`foreach` operator. The transformation function can be stateful (i.e., in Python it can be a callable function class that holds state in class members, and in the case of sequential operators it can reference local variables via closure capture). In the A3C example, `foreach` is used to compute gradients for each batch of experiences, which depends on the current policy state of the source actor. In the case of the `ComputeGradients` step, this state is available in the local process memory of the rollout worker, and is accessible because RLlib flow schedules the execution of parallel operations onto the source actors.

```
foreach(ParIter[T], T => U) -> ParIter[U]
foreach(Iter[T], T => U) -> Iter[U]
```



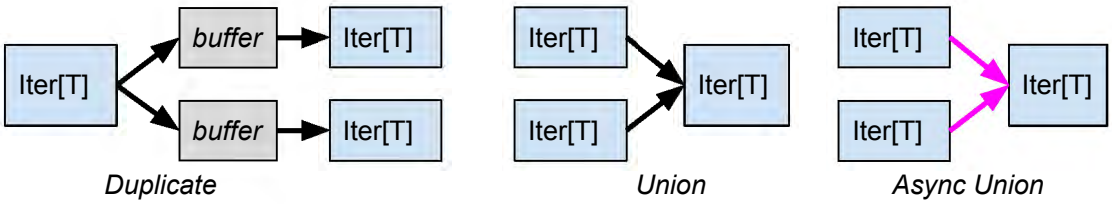
Sequencing: To consume a parallel iterator, the items have to be serialized into some sequential order. This is the role of sequencing operators. Once converted into a sequential iterator, `next` can be called on the iterator to fetch a concrete item from the iterator. The `gather_async` operator is used in A3C, and gathers computed gradients as fast as they are computed for application to a central policy. For a deterministic variation, we could have instead used `gather_sync`, which waits for one gradient from each shard of the iterator before returning. The sync gather operator also has *barrier semantics*. Upstream operators connected by a synchronous dependencies (black arrows) are fully halted between item fetches. This allows for the source actors to be updated prior to the next item fetch. Barrier semantics do not apply across asynchronous dependencies, allowing the mixing of synchronous and async dataflow fragments separated by pink arrows.

```
gather_async(ParIter[T],
        num_async: Int) -> Iter[T]
gather_sync(ParIter[T]) -> Iter[List[T]]
next(Iter[T]) -> T
```



Concurrency: Complex algorithms may involve multiple concurrently executing dataflow fragments. Concurrency (`union`) operators govern how these concurrent iterators relate to each other. For example, one may wish two iterators to execute sequentially in a round robin manner, execute independently in parallel, or rate limiting progress to a fixed ratio [61]. Additionally, one might wish to duplicate (`split`) an iterator, in which case buffers are automatically inserted to retain items until fully consumed. In this case, the RLlib flow scheduler tries to bound memory usage by prioritizing the consumer that is falling behind.

```
split(Iter[T]) -> (Iter[T], Iter[T])
union(List[Iter[T]],
  weights: List[float]) -> Iter[T]
union_async(List[Iter[T]]): Iter[T]
```



Scheduling and Nesting: In our implementation of RLlib flow, scheduling is static. Parallel transformations are executed on the nearest upstream source actor. Sequential transformations are executed on the process calling `next` on the iterator. RLlib flow iterators can be nested through the underlying actor model: a new parallel iterator can be produced from a set of actors each consuming an independent iterator. This enables nested or tree-like parallel topologies.

4.5 Implementation

We implemented RLlib flow on the Ray distributed actor framework [107] as two separate modules: a general purpose parallel iterator library (1241 lines of code), and a collection of RL specific dataflow operators (1118 lines of code) (Figure 4.3). We then ported the full suite of 20+ RL algorithms in RLlib to RLlib flow, replacing the *original implementations built directly on top of low-level actor and RPC primitives.*

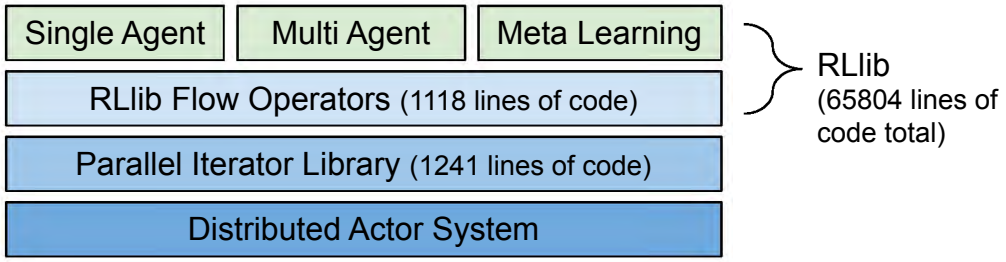


Figure 4.3: High-level architecture diagram of RLlib flow, which is now part of RLlib.

Only the portions of code in RLlib related to distributed execution were changed (less than 5% overall), which allows us to fairly evaluate against the previous implementation. In this section we overview how a few important algorithm patterns were realized in RLlib flow.

Example: A3C and A2C

As previously seen in Figure 4.2, A3C is straightforward to express in RLlib flow. Listing 4.6 shows pseudocode for A3C in RLlib flow, and Listing 4.2 shows the synchronous A2C variation. Note that several operators take references to actors in their constructor (e.g., `ParallelRollouts`, `ReportMetrics`). This allows these operators to send messages to these actors during operator execution.

Listing 4.1: A3C Pseudocode.

```
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[Gradients]
grads = ParallelRollouts(workers)
    .par_for_each(ComputeGradients())
    .gather_async()
# type: Iter[TrainStats]
apply_op = grads
    .for_each(ApplyGradients(workers))
# type: Iter[Metrics]
return ReportMetrics(apply_op, workers)
```

The main difference between A3C and A2C is use of the `gather_async` vs `gather_sync`. In addition, A3C computes gradients as a parallel transformation prior to sequencing, whereas A2C computes and applies gradients internally in the `TrainOneStep` operator.

Listing 4.2: A2C Pseudocode.

```
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[Gradients]
grads = ParallelRollouts(workers)
    .par_for_each(ComputeGradients())
    .gather_async()
# type: Iter[TrainStats]
apply_op = grads
    .for_each(ApplyGradients(workers))
# type: Iter[Metrics]
return ReportMetrics(apply_op, workers)
```

Example: Ape-X Prioritized Experience Replay

Ape-X [63] (Figure 4.4) is a high-throughput variation of DQN. It is notable since it involves multiple concurrent sub-flows (experience storage, experience replay), sets of actors (rollout actors, replay actors), and actor messages (updating model weights, updating replay buffer priorities). The sub-flows (`store_op`, `replay_op`) can be composed in RLlib flow as follows using the `Union` operator (Listing 4.3).

Listing 4.3: Ape-X Pseudocode.

```
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: List[ReplayActor]
replay_buffer = create_replay_actors()
# type: Iter[Rollout]
rollouts = ParallelRollouts(workers)
    .gather_async()
# type: Iter[_]
store_op = rollouts
    .for_each(StoreToBuffer(replay_buffer))
    .for_each(UpdateWeights(workers))
# type: Iter[TrainStats]
replay_op = ParallelReplay(replay_buffer)
    .gather_async()
    .for_each(UpdatePriorities(workers))
```

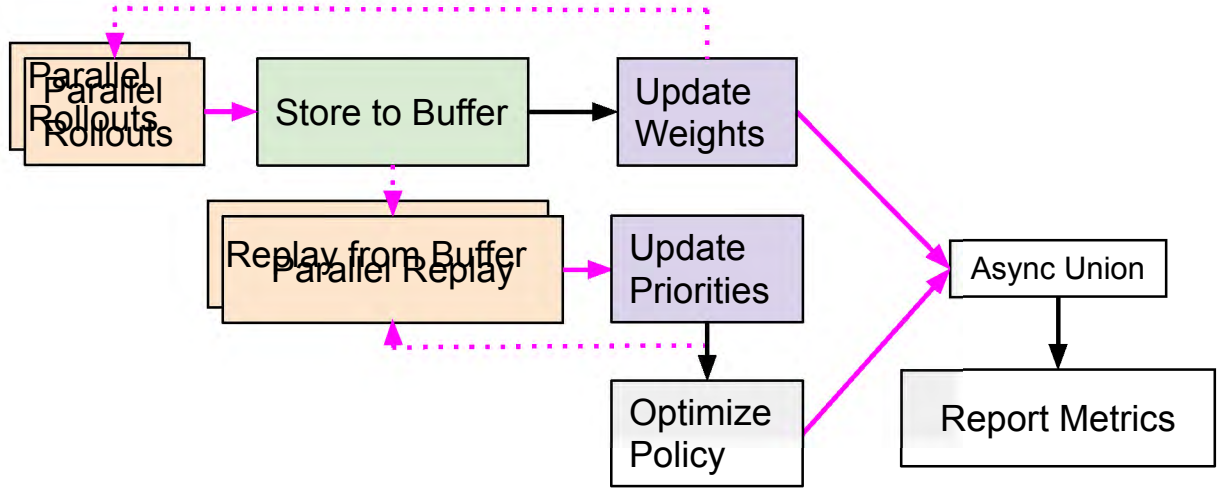


Figure 4.4: RLlib dataflow diagram for Ape-X. Two dataflow fragments are executed concurrently to optimize the policy.

```
    .for_each(TrainOneStep(workers))
# type: Iter[Metrics]
return ReportMetrics(
    Union(store_op, replay_op), workers)
```

Example: Multi-Agent Training

Similar to in Ape-X, multi-agent training can involve the composition of different training dataflows (i.e., PPO and DQN). Figure 4.5 shows the combined dataflow for an experiment that uses DQN to train certain policies in an environment and PPO to train others.

In an actor or RPC-based programming model, this type of composition is difficult because dataflow and control flow logic is intermixed. However, it is easy to express in RLlib flow using the `Union` operator (Listing 4.4).

Listing 4.4: Twin Trainer Pseudocode.

```
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[Rollout], Iter[Rollout]
r1, r2 = ParallelRollouts(workers).split()
# type: Iter[TrainStats], Iter[TrainStats]
ppo_op = ppo_plan(
    Select(r1, policy="PPO"), workers)
dqn_op = dqn_plan(
    Select(r2, policy="DQN"), workers)
```

```
# type: Iter[Metrics]
return ReportMetrics(
    Union(ppo_op, dqn_op), workers)
```

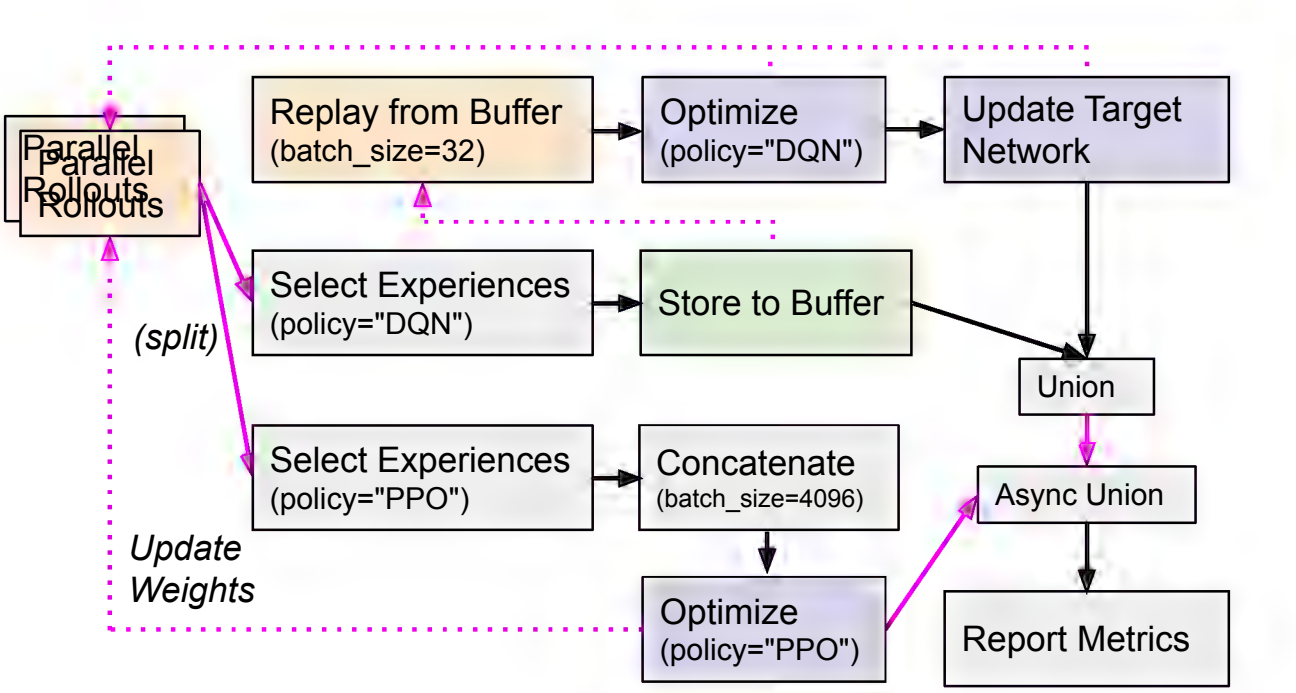


Figure 4.5: RLlib dataflow diagram for concurrent multi-agent training of DQN and PPO agents in an environment.

4.6 Evaluation

In our evaluation, we answer the following questions: nosep

- What is the quantitative improvement in code complexity with RLlib flow?
- Can RLlib flow execute RL workloads with high performance?
- How does RLlib flow compare to an off-the-shelf streaming system in terms of flexibility and performance for RL tasks?

Implementation Complexity

Lines of Code: In Table 4.1 we compare the original implementations of algorithms in RLlib to their size after porting to RLlib flow. No functionality was lost in the RLlib flow

	Original	RLlib flow	+shared	Ratio
A3C	87	9	52	1.6-9.6×
A2C	154	25	50	3.1-6.1×
DQN	239	87	139	1.7-2.7×
PPO	386	79	225	1.7-4.8×
Ape-X	250	126	216	1.1-1.9×
IMPALA	694	89	362	1.9-7.8×
MAML	370*	136	136	2.7×

Table 4.1: Lines of code for several prototypical algorithms implemented with the original RLlib vs our proposed RLlib flow-based RLlib. Our new RLlib flow-based RLlib has a consistent simplification in implementation for various algorithms. **Original MAML from `https://github.com/jonasrothfuss/ProMP`*

re-implementations. We count all lines of code directly related to distributed execution, including comments and instrumentation code, but not including utility functions shared across all algorithms. For completeness, for RLlib flow we include both an minimal (`RLlib flow`) and conservative (`+shared`) estimate of lines of code. The conservative estimate includes lines of code in shared operators that are arguably specific to the algorithm (e.g., a multi-GPU training operator for PPO). Overall, we observe between a 1.9-9.6× (optimistic) and 1.1-3.1× (conservative) reduction in lines of code with RLlib flow. Notably, the most complex algorithm (IMPALA) shrunk from 694 to 89-362 lines. **Readability**: We believe RLlib flow provides several key benefits for the readability of RL algorithms:

1. The high-level dataflow of an algorithm is visible at a glance in very few lines of code, allowing readers to understand and modify the execution pattern without diving deep into the execution logic.

2. Execution logic is organized into individual operators, each of which has a consistent input and output interface (i.e., transforms an iterator into another iterator). In contrast to building on low-level RPC systems, this provides a way for developers to decompose their algorithms into reusable modules.

3. Performance concerns are isolated into the lower-level parallel iterator library. Developers do not need to deal with low-level concepts such as batching or flow-control.

Flexibility: As evidence of RLlib flow's flexibility, a summer intern working on RLlib was able to implement several model-based (e.g., MB-MPO) and meta-learning algorithms (e.g., MAML), neither of which fit into previously existing execution patterns in RLlib flow. This was only possible due to the flexibility of RLlib flow's model that mixes together the actor and dataflow model. Listing 4.5 concisely expresses MAML's dataflow (also shown in Figure 4.6) [40]. The MAML dataflow involves nested optimization loops; workers collect pre-adaptation

data, perform inner adaptation (i.e., individual optimization calls to an ensemble of models spread across the workers), and collect post-adaptation data. Once inner adaptation is complete, the accumulated data is batched together to compute the meta-update step, which is broadcast to all workers. RLlib flow captures MAML in 139 lines compared to a baseline of ≈370 lines (Table 4.1), showing the flexibility and conciseness provided by combining actors with dataflow.

Listing 4.5: MAML Pseudocode.

```
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[List[Rollouts[[
rollouts = ParallelRollouts(workers)
    .gather_sync()
# MAML Algo:
#   Loop until inner adaptation done:
#       1) Aggregate Data from Workers
#       2) Perform Inner Adaption on Data
adapt_op = rollouts
    .combine(InnerAdapt(workers, steps=5))
# Meta-update Step
train_op = adapt_op
    .for_each(MetaUpdate(workers))
# type: Iter[Metrics]
return ReportMetrics(train_op, workers)
```

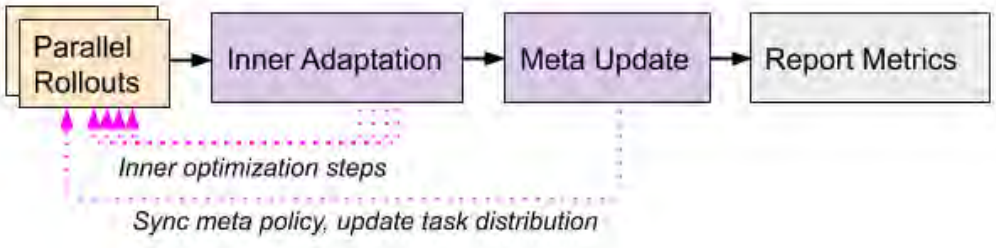


Figure 4.6: MAML dataflow, which includes a number of nested inner adaptation steps (optimization calls to the source actors) prior to update of the meta-policy. The meta-policy update and inner adaptation steps integrate cleanly into the dataflow, their ordering guaranteed by the synchronous data dependency barrier between the inner adaptation and meta update steps.

Microbenchmarks and Performance Comparison

For all the experiments, we use a cluster with an AWS p3.16xlarge GPU head instance with additional m4.16xlarge worker instances. All machines have 64 vCPUs and are connected by a 25Gbps network.

Sampling Microbenchmark: We evaluate the data throughput of RLlib flow in isolation by running RL training with a dummy policy (with only one trainable scalar). This enables us to observe how fast the framework can support sample collection from asynchronous environments. The SGD batch size and unroll length are fixed to 100K and 250, respectively. Figure 4.7a shows that our flow-based RLlib achieves comparable throughput to that of the original, and is slightly better at scale. We hypothesize that the improvements at large scale come from small optimizations such as batched RPC wait, which are easy to implement across multiple algorithms in a common way in RLlib flow.

IMPALA Throughput: We benchmark IMPALA, one of RLlib's high-throughput RL algorithms. We show that RLlib flow achieves similar or better performance compared to a previous version of IMPALA in RLlib that was built directly on top of low-level actor and RPC primitives. Figure 4.7b and Table 4.2 indicates that our new RLlib flow-based RLlib

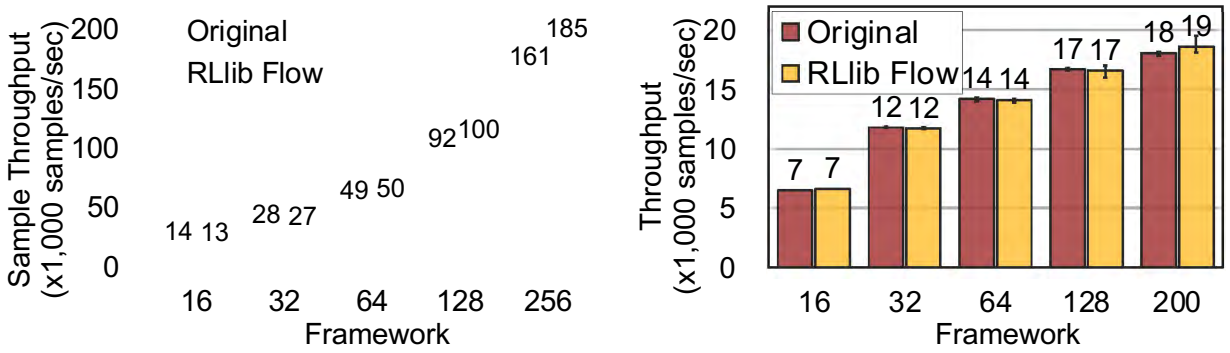
	Workers (GPUs)	Original	RLlib flow
Pendulum	2 (0)	1.2K	1.3K
Breakout	32 (4)	4.5K	4.1K
Pong	200 (2)	18.7K	18.9K

Table 4.2: Training throughput (samples/second) on different environments of Original and our RLlib flow-based RLlib. No additional overheads are introduced by RLlib flow.

achieves comparable throughput to the original RLlib on various environments. In Figure 4.7b, as the number of nodes increases, the new RLlib also shows similar scalability as the original one. That shows that RLlib flow does not impose overheads compared to a low-level implementation in a realistic scenario.

Comparison to Spark Streaming

Distributed dataflow systems such as Spark Streaming [174] and Flink [15] are designed for collecting and transforming live data streams from online applications (e.g., event streams, social media). Given the basic *map* and *reduce* operations, we can implement synchronous RL algorithms in any of these streaming frameworks. However, without consideration for the requirements of RL tasks (Section 4.3), these frameworks can introduce significant overheads. In Figure 4.8 we compare the performance of PPO implemented in Spark Streaming and RLlib flow.



(a) Sample efficiency: CartPole.

(b) Training throughput: Atari.

Figure 4.7: IMPALA before and after porting to RLlib flow.

PPO Implementation. In spark-code, we show the high-level pseudocode of our port of the PPO algorithm to Spark Streaming. Similar to our port of RLlib to RLlib flow, we only changed the parts of the PPO algorithm in RLlib that affect distributed execution, keeping the core algorithm implementation (e.g., numerical definition of policy loss and neural networks in TensorFlow) as similar as possible for fair comparison. We made a best attempt at working around aforementioned limitations (e.g., using a `binaryRecordsStream` input source to efficiently handle looping, defining efficient serializers for neural network state, and adjusting the microbatching to emulate the RLlib configuration).

Listing 4.6: Example of Spark Streaming for Distributed RL.

```
"""RL on Spark Streaming:
Iterate by saving and detecting binary states in a directory:
    1) Replicate the states to workers
    2) Sample in parallel (map)
    3) Collect the samples (reduce)
    4) Train on sampled batch
    5) Save the states and trigger next iteration
"""
# Set up the Spark cluster
sc = SparkContext(master_addr)
# Spark detects new states file in path
states = sc.binaryRecordsStream(path)
rep = states.flatMap(replicate_fn)
split = rep.repartion(NUM_WORKERS)
# Restore actor from states and sample
sample = splits.map(actor_sample_fn)
# Collect all samples from actors
reduced = sample.reduce(merge_fn)
```

```
# Restore trainer from states and train
new_states = reduced.map(train_fn)
# Save sampling/training states to path
new_states.foreachRDD(save_states_fn)
```

Experiment Setup: We conduct comparisons between the performance of both implementations. In the experiment, we adopt the PPO algorithm for the CartPole-v0 environment with a fixed sampling batch size B of 100K. Each worker samples ($B/\#$ workers) samples each iteration, and for simplicity, the learner updates the model on CPU using a minibatch with 128 samples from the sampled batch. Experiments here are conducted on AWS m4.10xlarge instances.

Data Framework Limitations: Spark Streaming is a data streaming framework designed for general purpose data processing. We note several challenges we encountered attempting to port RL algorithms to Spark Streaming:

1. Support for asynchronous operations. Data processing systems like Spark Streaming do not support asynchronous or non-deterministic operations that are needed for asynchronous RL algorithms.

2. Looping operations are not well supported. While many dataflow models in principle support iterative algorithms, we found it necessary to work around them due to lack of language APIs (i.e., no Python API).

3. Support for non-serializable state. In the dataflow model, there is no way to persist arbitrary state (i.e., environments, neural network models on the GPU). While necessary for fault-tolerance, the requirement for serializability impacts the performance and feasibility of many RL workloads.

4. Lack of control over batching. We found that certain constructs such as the data batch size for on-policy algorithms are difficult to control in traditional streaming frameworks, since they are not part of the relational data processing model.

For a single machine (the left three pairs), the breakdown of the running time indicates that the initialization and I/O overheads slow down the training process for Spark comparing to our RLlib. The former overheads come from the nature of Spark that the transformation functions do not persist variables. We have to serialize both the sampling and training states and re-initialize the variables in the next iteration to have a continuous running process. On the other hand, the I/O overheads come from looping back the states back to the input. As an event-time driven streaming system, the stream engine detects changes for the saved states from the source directory and starts new stream processing. The disk I/O leads to high overheads compared to our RLlib.

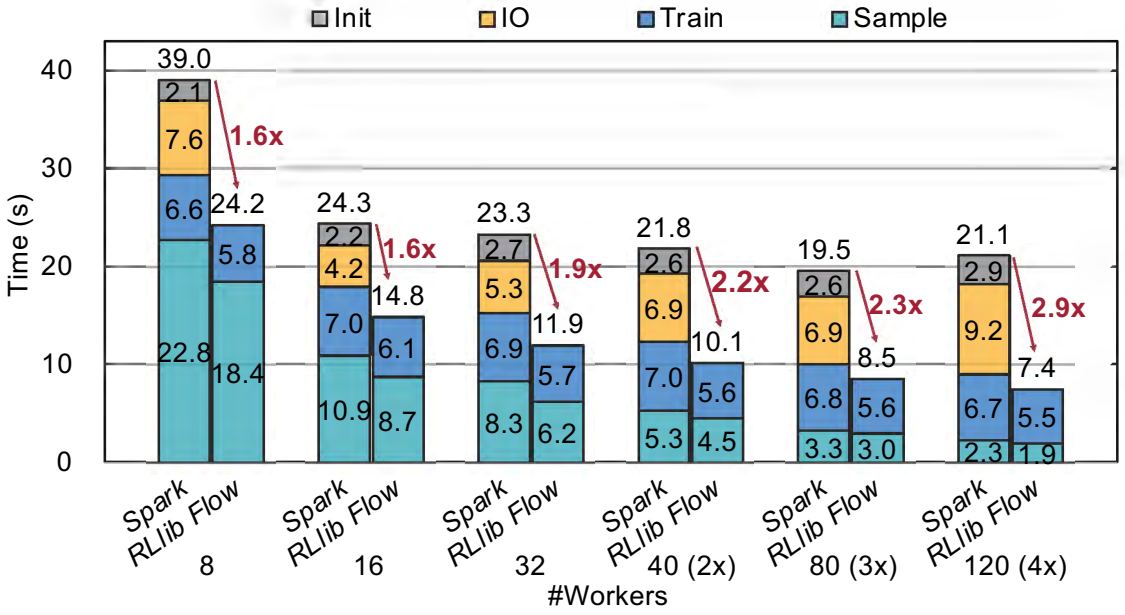


Figure 4.8: RLlib flow vs Spark Streaming PPO.

For the distributed situation (the right three pairs), the improvement of our RLlib becomes more significant against Spark, up to 2.9x. As the number of workers scales up, the sampling time decreases for both the dataflow model. Still, the initialization and I/O overheads stay unchanged, leading to lesser scalability for Spark.

4.7 Related Work

Reinforcement Learning Systems: RLlib flow is implemented concretely in RLlib, however, we hope it can provide inspiration for a new generation of general purpose RL systems. RL libraries available today range from single-threaded implementations [33, 59, 17, 78] to distributed [132, 96, 16, 92, 39, 61]. These libraries often focus on providing common frameworks for the numerical concerns of RL algorithms (e.g., loss, exploration, and optimization steps).

However, these aforementioned libraries rely on *predefined* distributed execution patterns. For example, for the Ape-X dataflow in Figure 4.4, RLlib defines this with a fixed "AsyncReplayOptimizer"[1] class that implements the topology; RLGraph uses an adapted implementation[2] from RLlib as part of their Ape-X algorithm meta-graph, while Coach does not support Ape-X[3]. These execution patterns are predefined as they are low-level, complex

[1] `https://docs.ray.io/en/releases-0.7.7/_modules/ray/rllib/optimizers/async_replay_optimizer.html`

[2] `https://github.com/rlgraph/rlgraph/blob/master/rlgraph/execution/ray/apex/apex_executor.py`

[3] `https://github.com/IntelLabs/coach`

to implement, and cannot be modified using high-level end-user APIs. In contrast, RLlib flow proposes a high-level distributed programming model for RL algorithm implementation, exposing this pattern in much fewer lines of code (Listing 4.3), and allowing free composition of these patterns by users (Listing 4.4). The ideas from RLlib flow can be integrated with any RL library to enable flexibility in distributed execution.

Distributed RL Algorithms: RLlib flow draws requirements from the distributed execution patterns of many recently proposed distributed RL algorithms. Initial approaches for scaling up RL in a distributed setting often employed asynchronous SGD [27]. Examples of such include A3C [105] and Gorila [110] which extend the actor-critic and DQN frameworks [106]. Specifically, in A3C [105], workers independently collect data, compute gradients, and send gradients to the global policy. Since then, numerous followup papers have sought upon improving existing asynchronous paradigms, such as Ape-X [63] and BA3C [2].

Another axis to distributed RL is increasing sampling throughput asynchronously from workers, which increases GPU utilization. Papers such as GA3C [8], IMPALA [36], and SEED [37] decouples learning (applying gradient updates) and acting (batch collection) by having workers asynchronously sample from a set environments, at the cost of unstable learning. IMPALA [36] and IMPACT [97] mitigates this by leveraging importance sampling.

RL can also be scaled up synchronously. A2C [105] is a synchronous variant of A3C, where the primary difference lies in workers sharing the same policy and how these workers gather data. Other policy gradient algorithms like PPO [134] and its distributed variants [162, 144, 97] are also synchronous; Distributed PPO [134] aggregates data from workers and perform mini-batch SGD on the aggregated data.

Distributed Computation Models: RLlib flow draws inspiration from both streaming dataflow and actor-based programming models. Popular open source implementations of streaming dataflow, including Apache Storm [151], Apache Flink [15], and Apache Spark [172, 174] transparently distribute data to multiple processors in the background, hiding the scheduling and message passing for distribution from programmers. In spark, we show how distributed PPO can be implemented in Apache Spark. Apache Flink's `Delta Iterate` operator can similarly support synchronous RL algorithms. However, data processing frameworks have limited asynchronous iteration support.

The Volcano (Iterator) model [46], pioneers the iterator abstraction for distributed data processing. RLlib flow builds on the Volcano model to not only encapsulate parallelism, but also to encapsulate the synchronization requirements between concurrent dataflow fragments, enabling users to also leverage actor message passing.

Naiad [109] is a low-level distributed dataflow system that supports cyclic execution graphs and message passing. It is designed as a system for implementing higher-level programming models. In principle, it is possible to implement the RLlib flow model in Naiad. Transformation operators can be placed on the stateful vertices of the execution graph. The message passing and concurrency (`Union`) operators can be represented by calling `SendBy`

and OnRecv interface on senders and receivers, which support asynchronous execution. RLlib flow's barrier semantics can be expressed with OnNotify and NotifyAt, where the former indicates all the required messages are ready, and the latter blocks execution until the notification has been received. We implemented RLlib flow on Ray instead of Naiad for practical reasons (e.g., Python support).

Deep Learning Frameworks: Deep learning frameworks such as PyTorch [39] and Tensorflow [1] provide high-level tensor algebra abstractions that can evaluate and optimize tensor programs within a single machine and across a cluster. RL frameworks leverage deep learning frameworks internally to execute tensor computations, but only a few frameworks (e.g., RLgraph) leverage deep learning frameworks for distribution. This is because RL computations are not generally differentiable across environment steps.

4.8 Conclusion

In summary, we propose RLlib flow, a hybrid actor-dataflow programming model for distributed RL. We designed RLlib flow to simplify the understanding, debugging, and customization of distributed RL algorithms RL developers require. RLlib flow provides comparable performance to reference algorithms implemented directly on low-level actor and RPC primitives, enables complex multi-agent and meta-learning use cases, and reduces the lines of code for distributed execution in a production RL library by 2-9x.

Chapter 5

Application: Optimizing Packet Classification Data Structures

In this chapter we introduce NeuroCuts, an example of applying RL to solve a systems problem. The problem at hand, packet classification, is fundamental in computer networking. This problem exposes a hard tradeoff between the computation and state complexity, which makes it particularly challenging. To navigate this tradeoff, existing solutions rely on complex hand-tuned heuristics, which are brittle and hard to optimize. As we will see, an RL based approach not only considerably improves on the performance of heuristic-based algorithms, but can be flexibly tuned to optimize for different objectives. Unlike typical RL-based approaches, NeuroCuts uses RL only as an offline step to generate an optimized data structure, which can then be deployed directly to production just like the result of a heuristic algorithm can. This class of approach, when possible to take, has the advantage of zero deploy-time overhead and guaranteed correctness. We implement NeuroCuts using RLlib, taking advantage of its multi-agent API to model the problem, and its scalability to accelerate training.

5.1 Introduction

We propose a deep reinforcement learning (RL) approach to solve the packet classification problem. There are several characteristics that make this problem a good fit for Deep RL. First, many existing solutions iteratively build a decision tree by splitting nodes in the tree. Second, the effects of these actions (e.g., splitting nodes) can only be evaluated once the entire tree is built. These two characteristics are naturally captured by the ability of RL to take actions that have sparse and delayed rewards. Third, it is computationally efficient to generate data traces and evaluate decision trees, which alleviate the notoriously high sample complexity problem of Deep RL algorithms. Our solution, NeuroCuts, uses succinct representations to encode state and action space, and efficiently explore candidate decision trees to optimize for a global objective. It produces compact decision trees optimized for

a specific set of rules and a given performance metric, such as classification time, memory footprint, or a combination of the two. Evaluation on ClassBench shows that NeuroCuts outperforms existing hand-crafted algorithms in classification time by 18 percent at the median, and reduces both classification time and memory footprint by up to 3x.

The goal of packet classification is to match a given packet to a rule from a set of rules, and to do so while optimizing the classification time and/or memory footprint. Packet classification is a key building block for many network functionalities, including firewalls, access control, traffic engineering, and network measurements [53, 154, 89]. As such, packet classifiers are widely deployed by enterprises, cloud providers, ISPs, and IXPs [9, 89, 141].

Existing solutions for packet classification can be divided into two broad categories. Solutions in the first category are hardware-based. They leverage Ternary Content-Addressable Memories (TCAMs) to store all rules in an associative memory, and then match a packet to all these rules in parallel [80]. As a result, TCAMs provide constant classification time, but come with significant limitations. TCAMs are inherently complex, and this complexity leads to high cost and power consumption. This makes TCAM-based solutions prohibitive for implementing large classifiers [154].

The solutions in the second category are software based. These solutions build sophisticated in-memory data structures—typically decision trees—to efficiently perform packet classification [89]. While these solutions are far more scalable than TCAM-based solutions, they are slower, as the classification operation needs to traverse the decision tree from the root to the matching leaf.

Building efficient decision trees is difficult. Over the past two decades, researchers have proposed a large number of decision tree based solutions for packet classification [53, 140, 124, 154, 89]. However, despite the many years of research, these solutions have two major limitations. First, they rely on hand-tuned heuristics to build the tree. Examples include maximizing split entropy [53], balancing splits with custom space measures [53], special handling for wildcard rules [140], and so on. This makes them hard to understand and optimize over different sets of rules. If a heuristic is too general, it cannot take advantage of the characteristics of a particular set of rules. If a heuristic is designed for a specific set of rules, it typically does not achieve good results on another set of rules with different characteristics.

Second, these heuristics do not explicitly optimize for a given objective (e.g., tree depth). They make decisions based on information (e.g., the difference between the number of rules in the children, the number of distinct ranges in each dimension) that is only *loosely* related to the global objective. As such, their performance can be far from optimal.

In this chapter, we propose a learning approach to packet classification. Our approach has the potential to address the limitations of the existing hand-tuned heuristics. In particular, our approach *learns* to optimize packet classification for a given set of rules and objective, can easily incorporate pre-engineered heuristics to leverage their domain knowledge, and does so with little human involvement. The recent successes of deep learning in solving notoriously hard problems, such as image recognition [77] and language translation [147], have inspired many practitioners and researchers to apply deep learning, in particular, and

machine learning, in general, to systems and networking problems [177, 170, 99, 153, 22, 67, 30, 176, 101]. While in some of these cases there are legitimate concerns about whether machine learning is the right solution for the problem at hand, we believe that deep learning is a good fit for our problem. This is notable since, when an efficient formulation is found, learning-based solutions have often outperformed hand-crafted alternatives [106, 138, 75].

There are two general approaches to apply learning to packet classification. The first is to replace the decision tree with a neural network, which given a packet will output the rule matching that packet. Unfortunately, while appealing, this end-to-end solution has a major drawback: it does not guarantee the correct rule is always matched. While this might be acceptable for some applications such as traffic engineering, it is not acceptable for others, such as access control. Another issue is that large rule sets will require correspondingly large neural network models, which can be expensive to evaluate without accelerators such as GPUs. The second approach, and the one we take in this chapter, is to use deep learning to build a decision tree. Recent work has applied deep learning to optimize decision trees for machine learning problems [112, 163, 73]. These solutions, however, are designed for machine learning settings that are different than packet classification, and aim to maximize accuracy. In contrast, decision trees for packet classification provide perfect accuracy by construction, and the goal is to minimize classification time and memory footprint.

Our solution uses deep reinforcement learning (RL) to build efficient decision trees. There are three characteristics that makes RL a particularly good fit for packet classification. First, the natural solution to build a decision tree is to start with one node and recursively split (cut) it. Unfortunately, this kind of approach does not have a greedy solution. When making a decision to cut a node, we do not know whether that decision was a good one (i.e., whether it leads to an efficient tree) before we finish building the actual tree. RL naturally captures this characteristic as it does not assume that the impact of a given decision on the performance objective is known immediately. Second, unlike existing heuristics which take actions that are only loosely related to the performance objective, the explicit goal of an RL algorithm is to directly maximize the performance objective. Third, unlike other RL domains such as as robotics, for our problem it is possible to evaluate an RL model quickly (i.e., a few seconds of CPU time). This alleviates one of the main drawbacks of RL algorithms: the non-trivial learning time due to the need to evaluate a large number of models to find a good solution. By being able to evaluate each model quickly (and, as we will see, in parallel) we significantly reduce the learning time.

To this end, we design NeuroCuts, a deep RL solution for packet classification that learns to build efficient decision trees. There are three technical challenges to formulate this problem as an RL problem. First, the tree is growing during the execution of the algorithm, as existing nodes are split. This makes it very difficult to encode the decision tree, as RL algorithms require a fixed size input. We address this problem by noting that the decision of how to split a node in the tree depends only on the node itself; it does not depend on the rest of the tree. As such, we do not need to encode the entire tree; we only need to encode the current node. The second challenge is in reducing the sparsity of rewards to accelerate the learning process; here we exploit the branching structure of the problem to provide denser

Priority	Src IP	Dst IP	Src Port	Dst Port	Protocol
2	10.0.0.0	10.0.0.0/16	*	*	*
1	*	*	[0, 1023]	[0, 1023]	TCP
0	*	*	*	*	*

Figure 5.1: A packet classifier example. Real-world classifiers can have 100K rules or more.

feedback for tree size and depth. The final challenge is that training for very large sets of rules can take a long time. To address this, we leverage RLlib [92], a distributed RL library.

In summary, we make the following contributions.

- We show that the packet classification problem is a good fit for reinforcement learning (RL).
- We present NeuroCuts, a deep RL solution for packet classification that learns to build efficient decision trees.
- We show that NeuroCuts outperforms state-of-the-art solutions, improving packet classification time by 18% at the median and reducing both time and memory usage by up to 3×.

The code for NeuroCuts is open source and is available at `https://github.com/neurocuts/neurocuts`.

5.2 Background

In this section, we provide background on the packet classification problem, and summarize the key ideas behind the decision tree based solutions to solve this problem.

Packet Classification

A packet classifier contains a list of rules. Each rule specifies a pattern on multiple fields in the packet header. Typically, these fields include source and destination IP addresses, source and destination port numbers, and protocol type. The rule's pattern specifies which packets match the rule. Matching conditions include prefix based matching (e.g., for IP addresses), range based matching (e.g., for port numbers), and exact matching (e.g., for protocol type). A packet matches a rule if each field in the packet header satisfies the matching condition of the corresponding field in the rule, e.g., the packet's source/destination IP address matches the prefix of the source/destination address in the rule, the packet's source/destination port

number is contained in the source/destination range specified in the rule, and the packet's protocol type matches the rule's protocol type.

Figure 5.1 shows a packet classifier with three rules. The first rule matches all packets with source address 10.0.0.1 and the destination addresses sharing prefix 10.0.0.0/16. Other fields are unspecified (i.e., they are $\star$) meaning that the rule matches any value in these fields. The second rule matches all TCP packets with source and destination ports in the range [0, 1023], irrespective of IP addresses (as they are $\star$). Finally, the third rule is a default rule that matches all packets. This guarantees that any packet matches at least one rule.

Since rules can overlap, it is possible for a packet to match multiple rules. To resolve this ambiguity, each rule is assigned a priority. A packet is then matched to the highest priority rule. For example, packet (10.0.0.0, 10.0.0.1, 0, 0, 6) matches all the three rules of the packet classifier in Figure 5.1. However, since the first rule has the highest priority, we match the packet to the first rule only.

Decision Tree Algorithms

Packet classification is similar to the point location problem in a multi-dimensional geometric space: the fields in the packet header we are doing classification on (e.g., source and destination IP addresses, source and destination port numbers, and protocol number) represent the dimensions in the geometric space, a packet is represented as a point in this space, and a rule as a hypercube. Unfortunately, the point location problem exhibits a hard tradeoff between time and space complexities [51].

The packet classification problem is then equivalent to finding all hypercubes that contains the point corresponding to a given packet. In particular, in a d-dimensional geometric space with n non-overlapping hypercubes and when $d > 3$, this problem has either (i) a lower bound of $O(log\ n)$ time and $O(n^d)$ space, or (ii) a lower bound of $O(log^{d-1} n)$ time and $O(n)$ space [51]. The packet classification problem allows the hypercubes (i.e., rules) to overlap, and thus is at least as hard as the point location problem [51]. In other words, if we want logarithmic computation time, we need space that is exponential in the number of dimensions (fields), and if we want linear space, the computation time will be exponential in the logarithm of the number of rules. Given that for packet classification $d = 5$, neither of these choices is attractive.

Next, we discuss two common techniques employed by existing solutions to build decision trees for packet classification: *node cutting* and *rule partition*.

Node cutting. Most existing solutions for packet classification aim to build a decision tree that exhibits low classification time (i.e., time complexity) and memory footprint (i.e., space complexity) [154]. The main idea is to split nodes in the decision tree by "cutting" them along one or more dimensions. Starting from the root which contains all rules, these algorithms iteratively split/cut the nodes until each leaf contains fewer than a predefined number of rules. Given a decision tree, classifying a packet reduces to walk the tree from the root to a leaf, and then chose the highest priority rule associated with that leaf.

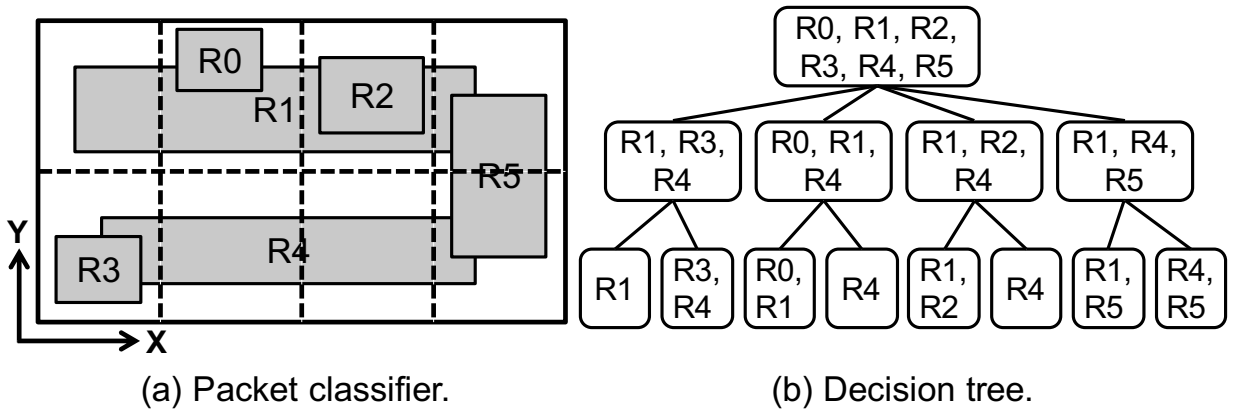


Figure 5.2: Node cutting.

Figure 5.2 illustrates this technique. The packet classifier contains six rules (R0 to R5) in a two-dimensional space. Figure 5.2(a) shows each rule as a rectangle in the space, and represents the cuts as dashed lines. Figure 5.2(b) shows the corresponding decision tree for this packet classifier. The root of the tree contains all the six rules. First, we cut the entire space (which represents the root) into four chunks along dimension x. This leads to the creation of four children. If a rule intersects a child's chunk, it is added to that child. For example, R1, R3 and R4 all intersect the first chunk (i.e., the first quarter in this space), and thus they are all added to the first root's child. If a rule intersects multiple chunks it is added to each corresponding child, e.g., R1 is added to all the four children. Next, we cut the chunk corresponding to each of the four children along dimension y. As a result, each of the nodes at the first level will end up with two children.

Rule partition. One challenge with "blindly" cutting a node is that we might end up with a rule being replicated to a large number of nodes [154]. In particular, if a rule has a large size along one dimension, cutting along that dimension will result in that rule being added to many nodes. For example, rule R1 in Figure 5.2(a) has a large size in dimension x. Thus, when cutting along dimension x, R1 will end up being replicated at every node created by the cut. Rule replication can lead to decision trees with larger depths and sizes, which translate to higher classification time and memory footprint.

One solution to address this challenge is to first partition rules based on their "shapes". Broadly speaking, rules with large sizes in a particular dimension are put in the same set. Then, we can build a separate decision tree for each of these partitions. Figure 5.3 illustrates this technique. The six rules in Figure 5.2 are grouped into two partitions. One partition consists of rules R1 and R4, as both these rules have large sizes in dimension x. The other partition consists of the other four rules, as these rules have small sizes in dimension x. Figure 5.3(a) and Figure 5.3(b) show the corresponding decision trees for each partition. Note that the resulting trees have lower depth, and smaller number of rules per node as compared to the original decision tree in Figure 5.2(b). To classify a packet, we classify it

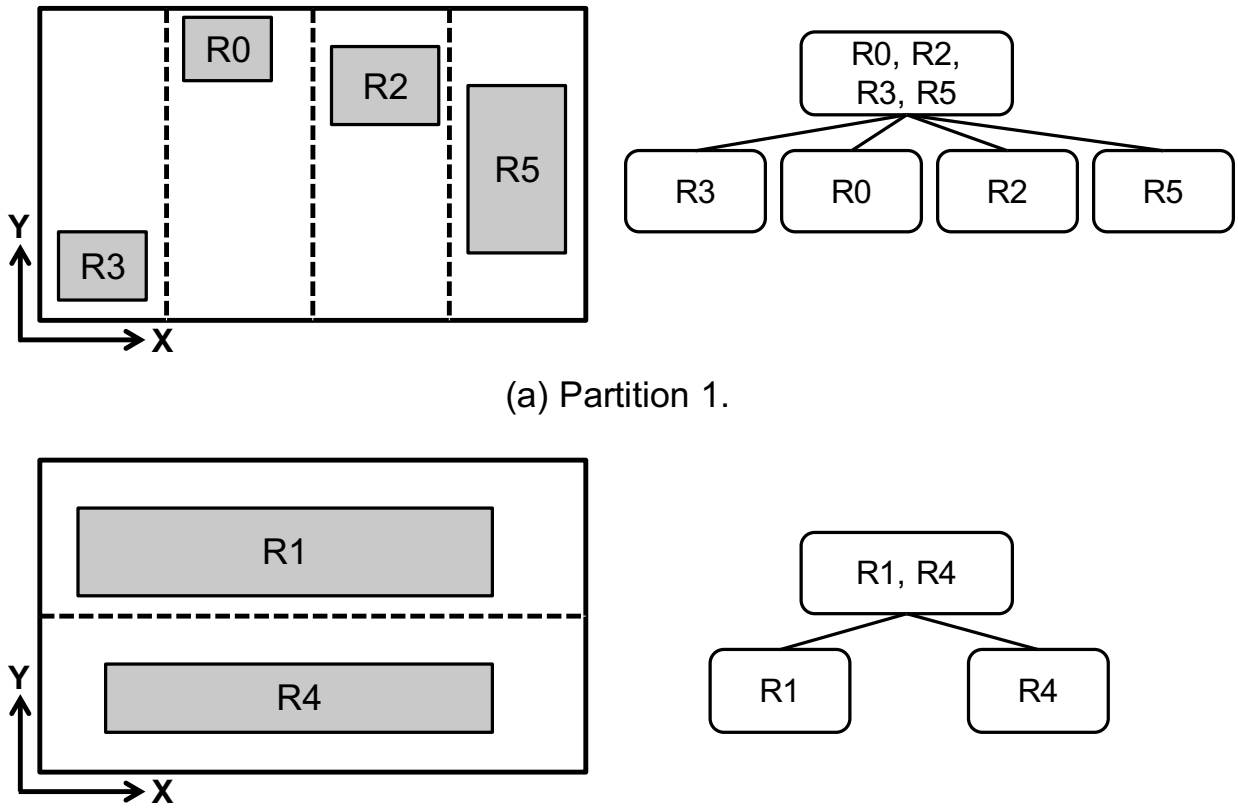


Figure 5.3: Rule partition.

against every decision tree, and then choose the highest priority rule among all rules the packet matches in all decision trees.

Summary. Existing solutions build decision trees by employing two types of actions: node cutting and rule partition. These solutions mainly differ in the way they decide (i) at which node to apply the action, (ii) which action to apply, and (iii) how to apply it (e.g., along which dimension(s) to partition).

5.3 A Learning-Based Approach

In this section, we describe a learning-based approach for packet classification. We motivate our approach, discuss the formulation of classification as a learning problem, and then present our solution.

Why Learn?

The existing solutions for packet classification rely on hand-tuned heuristics to build decision trees. Unfortunately, this leads to two major limitations.

First, these heuristics often face a difficult *trade-off* between *performance* and *cost*. Tuning such a heuristic for a given set of rules is an expensive proposition, requiring considerable human efforts and expertise. Worse yet, when given a different rule set, one might have to do this all over again. Addressing this challenge has been the main driver of a long line of research over the past two decades [53, 140, 124, 154, 89]. Of course, one could build a general heuristic for a large variety of rule sets. Unfortunately, such a solution would not provide the best performance for a given set of rules.

Second, existing algorithms do not directly optimize for a global objective. Ideally, a good packet classification solution should optimize for (*i*) classification time, (*ii*) memory footprint, or (*iii*) a combination between the two. Unfortunately, the existing heuristics do not *directly* optimize for any of these objectives. At their core, these heuristics make greedy decisions to build decision trees. At every step, they decide on whether to cut a node or partition the rules based on simple statistics (e.g., the size of the rules in each dimension, number of unique ranges in each dimension), which are poorly correlated with the desired objective. As such, the resulting decision trees are often far from being optimal.

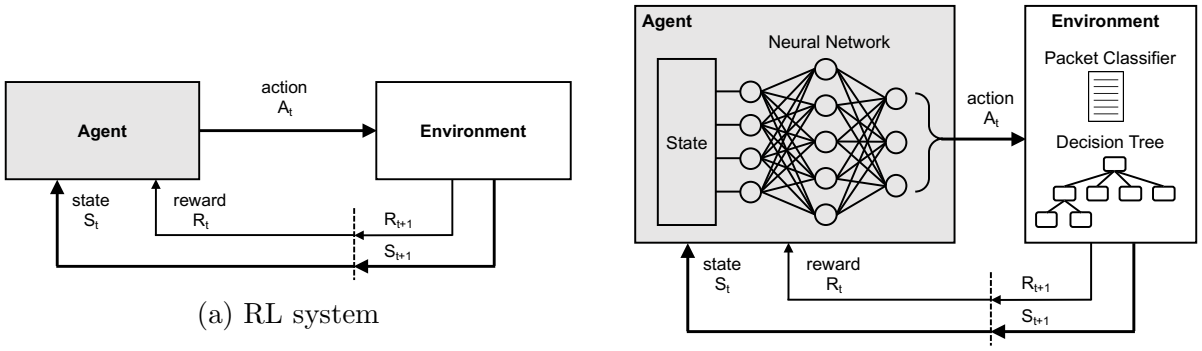
As we will see, a learning-based approach can address these limitations. Such an approach can learn to generate an efficient decision tree for a specific set of rules without the need to rely on hand-tuned heuristics. This is not to say these heuristics do not have value; in fact they often contain key domain knowledge that we show can be leveraged and improved on by the learning algorithm.

What to Learn?

Classification is a central task in machine learning literature. The recent success of using deep neural networks (DNNs) for image recognition, speech recognition and language translation has been single-handedly responsible for the recent AI "revolution" [77, 147, 48].

As such, one natural solution for packet classification would be to replace a decision tree with a DNN. In particular, such DNN will take as input the fields of a packet header and output the rule matching that packet. Related to our problem, prior work has shown that DNN models can be effectively used to replace B-Trees for indexing [75].

However, this solution has two drawbacks. First, a DNN-based classifier does not guarantee 100% accuracy. This is because training a DNN is fundamentally a stochastic process. Second, given a DNN packet classification result, it is expensive to verify whether the result is correct or not. Unlike the recently proposed learned index solution to replace B-Trees [75], the rules in packet classification are multi-dimensional and overlap with each other. If a rule matches a packet, we still need to check other rules to see if this rule has the highest priority among all matched rules.



(a) RL system

(b) NeuroCuts

Figure 5.4: (a) Classic RL system. An agent takes an action, A_t, based on the current state of the environment, S_t, and applies it to the environment. This leads to a change in the environment state (S_{t+1}) and a reward (R_{t+1}). (b) NeuroCuts as an RL system.

To avoid these drawbacks, in this chapter we propose to learn building decision trees for a given set of rules. Since the result is still a decision tree, we can guarantee correctness, and it will be easy to deploy the classifier with existing systems (hardware and software) compared to a DNN.

How to Learn?

In this section, we show that the problem of building decision trees maps naturally to RL. As illustrated in Figure 5.4(a), an RL system consists of an agent that repeatedly interacts with an environment. The agent observes the state of the environment, and then takes an action that might change the environment's state. The goal of the agent is to compute a *policy* that maps the environment's state to an action in order to optimize a reward. As an example, consider an agent playing chess. In this case, the environment is the board, the state is the position of the pieces on the board, an action is moving a piece on the board, and the reward could be 1 if the game is won, and -1, if the game is lost.

This simple example illustrates two characteristics of RL that are a particularly good fit to our problem. First, rewards are *sparse*, i.e., not every state has associated a reward. For instance, when moving a piece we do not necessary know whether that move will result in a win or loss. Second, the rewards are *delayed*; we need to wait until the end of the game to see whether the game was won or lost.

To deal with large state and action spaces, recent RL solutions have employed DNNs to implement their policies. These solutions, called Deep RL, have achieved remarkable results matching humans at playing Atari games [106], and beating the Go world champion [139]. These results have encouraged researchers to apply Deep RL to networking and systems problems, from routing, to congestion control, to video streaming, and to job scheduling [177, 170, 99, 153, 22, 67, 30, 176, 101]. Building a decision tree can be easily cast as an RL

problem: the environment's state is the current decision tree, an action is either cutting a node or partitioning a set of rules, and the reward is either the classification time, memory footprint, or a combination of the two. While in some cases there are legitimate concerns about whether Deep RL is the right solution for the problem at hand, we identify several characteristics that make packet classification a particularly good fit for Deep RL.

First, when we take an action, we do not know for sure whether it will lead to a good decision tree or not; we only know this once the tree is built. As a result, the rewards in our problem are both *sparse* and *delayed*. This is naturally captured by the RL formulation.

Second, the explicit goal of RL is to maximize the reward. Thus, unlike existing heuristics, our RL solution aims to explicitly optimize the performance objective, rather than using local statistics whose correlation to the performance objective can be tenuous.

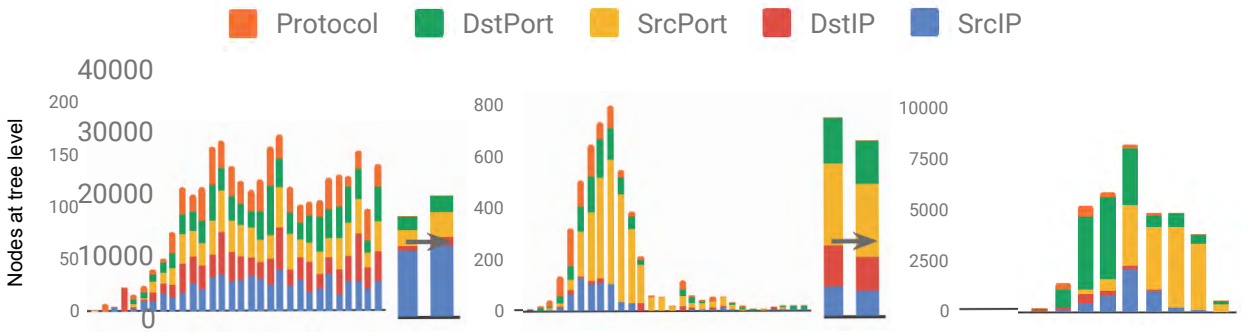
Third, one potential concern with Deep RL algorithms is sample complexity. In general, these algorithms require a huge number of samples (i.e., input examples) to learn a good policy. Fortunately, in the case of packet classification we can generate such samples cheaply. A sample, or rollout, is a sequence of actions that builds a decision tree with the associated reward(s) by using a given policy. The reason we can generate these rollouts cheaply is because we can build all these trees in software, and do so in parallel. Contrast this with other RL-domains, such as robotics, where generating each rollout can take a long time and requires expensive equipment (i.e., robots).

5.4 NeuroCuts Design

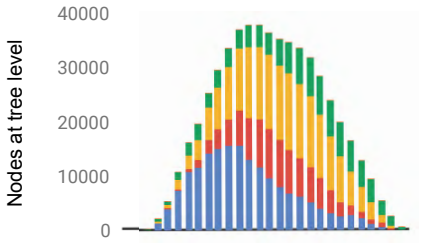
NeuroCuts Overview

We introduce the design for NeuroCuts, a new Deep RL formulation of the packet classification problem. Given a rule set and an objective function (i.e., classification time, memory footprint, or a combination of both), NeuroCuts learns to build a decision tree that minimizes the objective.

Figure 5.4(b) illustrates the framing of NeuroCuts as an RL system: the environment consists of the set of rules and the current decision tree, while the *agent* uses a model (implemented by a DNN) that aims to select the best cut or partition action to incrementally build the tree. A cut action divides a node along a chosen dimension (i.e., one of `SrcIP`, `DstIP`, `SrcPort`, `DstPort`, and `Protocol`) into a number of sub-ranges (i.e., 2, 4, 8, 16, or 32 ranges), and creates that many child nodes in the tree. A partition action on the other hand divides the rules of a node into disjoint subsets (e.g., based on the coverage fraction of a dimension), and creates a new child node for each subset. The available actions for the current node are advertised by the environment at each step, the agent chooses among them to generate the tree, and over time the agent learns to optimize its decisions to maximize the reward from the environment. Figure 5.5 visualizes the learning process of NeuroCuts.



(a) NeuroCuts starts with a randomly initialized policy that generates poorly shaped trees (left, truncated). Over time, it learns to reduce the tree depth and develops a more coherent strategy (center). The policy converges to a compact depth-12 tree (right) that specializes in cutting `SrcIP`, `SrcPort`, and `DstPort`.



(b) In comparison, HiCuts produces a depth-29 tree for this rule set that is 15× larger and 3× slower in classification time.

Figure 5.5: Visualization of NeuroCuts learning to split the **fw5_1k** ClassBench rule set. The x-axis denotes the tree level, and the y-axis the number of nodes at the level. The distribution of cut dimensions per level of the tree is shown in color.

NeuroCuts Training Algorithm

Recall that the goal of an RL algorithm is to compute a policy to maximize rewards from the environment. Referring again to Figure 5.4, the environment defines the action space A and state space S. The agent starts with an initial policy, evaluates it using multiple rollouts, and then updates it based on the results (rewards) of these rollouts. Then, it repeats this process until satisfied with the reward.

We first consider a strawman formulation of decision tree generation as a single Markov Decision Process (MDP). In this framing, a rollout begins with a tree consisting of a single node. This is the initial state, $s_0 \in S$. At each step t, the agent executes an action $a_t \in A$ and receives a reward r_t; the environment transitions from the current state $s_t \in S$ to the next state $s_{t+1} \in S$ (i.e., the updated tree and next node to process). The goal is to maximize the total reward received by the agent, i.e., $\sum_t \gamma^t r_t$ where γ is a discounting factor used to

prioritize more recent rewards.

Design challenges. While at a high level this RL formulation seems straightforward, there are three key challenges we need to address before we have a realizable implementation. The first is how to encode the variable-length decision tree state s_t as an input to the neural network policy. While it is possible to flatten the tree, say, into an 1-dimensional vector, the size of such a vector would be very large (i.e., hundreds of thousands of units). This will require both a very large network model to process such input, and a prohibitively large number of samples.

While recent work has proposed leveraging recurrent neural networks (RNNs) and graph embedding techniques [164, 160, 165] to reduce the input size, these solutions are brittle in the face of large or dynamically growing graph structures [178]. Rather than attempting to solve the state representation problem to deal with large inputs, in NeuroCuts we instead take advantage of the underlying structure of packet classification trees to design a simple and compact state representation. This means that when the agent is deciding how to split a node, it only observes a fixed-length representation of the node. All needed state is encoded in the representation; no other information about the rest of the tree is observed.

The second challenge is how to deal with the sparse and delayed rewards incurred by the node-by-node process of building the decision tree. While we could in principle return a single reward to the agent when the tree is complete, it would be very difficult to train an agent in such an environment. Due to the long length of tree rollouts (i.e., many thousands of steps), learning is only practical if we can compute *meaningful dense rewards*.[1] Such a dense reward for an action would be based on the statistics of the subtree it leads to (i.e., its depth or size).[2] This effectively reduces the delay of the rewards from $O(\text{tree size})$ to $O(log(\text{tree size}))$. Unfortunately, it is not possible to compute this until the subtree is complete. To handle this, we take the somewhat unusual step of only computing rewards for the rollout when the tree is completed, and setting $\gamma = 0$, effectively creating a series of 1-step decision problems similar to contextual bandits [81]. However, unlike the bandit setting, where an agent only makes a decision once per environment, these 1-step decisions are connected through the dynamics of the tree building process. For instance, this makes NeuroCuts amenable to techniques from the Deep RL literature such as GAE [133].

Another way of looking at the dense reward problem is that the process of building a decision tree is not really sequential but *tree-structured* (i.e., it is more accurately modeled as a branching decision process [71, 121, 38]), and we need to account for the reward calculations accordingly. In such a "branching" formulation, $\gamma > 0$, but the rewards of an action are computed as an aggregation over *multiple child states* produced by an action. For example, cutting a node produces multiple child sub-nodes, and the reward calculation may involve a `sum` or a `min` over each child's future rewards, depending on whether we are optimizing for

[1] Note that just returning -1 or -*cutSize* for each step would be a dense reward but not particularly useful.

[2] The rewards for NeuroCuts correspond to the true problem objective; we do not do "reward engineering" since that would bias the solution.

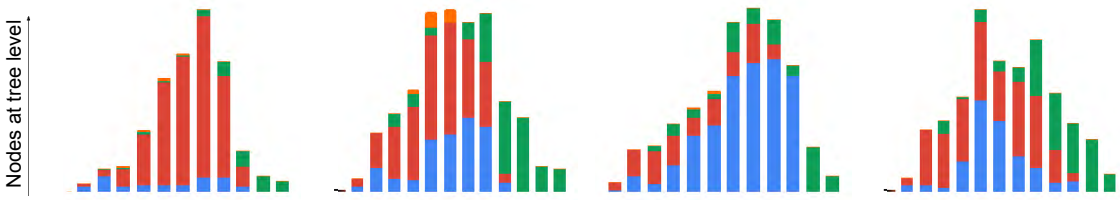


Figure 5.6: The NeuroCuts policy is stochastic, which enables it to effectively explore many different tree variations during training. Here we visualize four random tree variations drawn from a single policy trained on the **acl4_1k** ClassBench rule set. The x-axis denotes the tree level, and the y-axis the number of nodes at the level. The distribution of cut dimensions per level of the tree is shown in color.

tree size or depth. The 1-step decision problem and branching decision process formulations of NeuroCuts are roughly equivalent; in the implementation section we describe how we adapt standard RL algorithms to run NeuroCuts.

The final challenge is how to scale the solution to large packet classifiers. The decision tree for a packet classifier with 100K rules can have hundreds of thousands of nodes. The size of the tree impedes training along several dimensions. Not only does it take more steps to finish building a tree, but the execution time of each action increases as there are more rules to process. The space of trees to explore is also larger, requiring the use of larger network models and generating more rollouts to train.

State representation. One key observation is that the action on a tree node only depends on the node itself, so it is not necessary to encode the entire decision tree in the environment state. Our goal to optimize a global performance objective over the entire tree suggests that we would need to make decisions based on the global state. However, this does not mean that the state representation needs to encode the entire decision tree. Given a tree node, the action on that node only needs to make the best decision to optimize the sub-tree rooted at that node. It does not need to consider other tree nodes in the decision tree.

Formally, given tree node n, let t_n and s_n denote n's classification time and memory footprint, respectively, and T_n and S_n be the classification time and memory footprint of the entire sub-tree rooted at node n, respectively. Then, for a cut action, we have the following equations:

$$T_n = t_n + \text{max}_{i \in children(n)} T_i \tag{5.1}$$
$$S_n = s_n + \text{sum}_{i \in children(n)} S_i \tag{5.2}$$

Similarly, for a partition action, we have as an upper bound on cost, assuming serial execution:

$$T_n = t_n + \text{sum}_{i \in children(n)} T_i \tag{5.3}$$
$$S_n = s_n + \text{sum}_{i \in children(n)} S_i \tag{5.4}$$

An action, a, taken on node n only needs to optimize the sub-tree rooted at n according to the following expression,

$$V_n = \text{argmax}_{a \in A} - (c \cdot T_n + (1 - c) \cdot S_n), \tag{5.5}$$

where c is a coefficient capturing the tradeoff between classification time and memory footprint. The negation is needed since we want to *minimize* time and space complexities. We note that these values can be computed after the tree is fully built, regardless of the traversal order taken building the tree.

When $c \in \{0, 1\}$, it is easy to see that if at every tree node n we take the action that optimizes V_n, then, by induction, we end up optimizing V_r, where r is the root of the tree. In other words, we end up optimizing the global objective (reward) for the entire decision tree. For $0 < c < 1$ this optimization becomes approximate, but we find empirically that c can still be used to interpolate between the two objectives. It is important to note here that while the state representation only encodes current node n, action a taken for node n is *not local*, as it optimizes the *entire sub-tree* rooted at n.

In summary, we only need to encode the current node as the input state of the agent. This is because the environment builds the tree node-by-node, node actions need only consider their own state, and each node contains a subset of the rules of its parent (i.e., rules contained in some subspace of its parent space). Therefore, nodes in the tree can be completely defined by the ranges they occupy in each dimension. Given d dimensions, we use $2d$ numbers to encode a tree node, which indicate the left and right boundaries of each dimension for this node. The state also needs to describe the partitioning at the node, which can be handled in a similar way. We note that the set of rules for the packet classifier are *not* present in the observation space. NeuroCuts learns to account for packet classifier rules implicitly through the rewards it gets from the environment. A full description of the NeuroCuts state and action representations can be found in Table 5.1.

Training algorithm. We use an actor-critic algorithm to train the agent's policy [72]. This class of algorithms have been shown to provide state-of-the-art results in many use cases [24, 105, 134], and can be easily scaled to the distributed setting [36]. We also experimented with Q-learning [106] based approaches, but found they did not perform as well.

Algorithm 1 shows the pseudocode of the NeuroCuts algorithm, which executes as follows. NeuroCuts starts with the root node of the decision tree, s^*. The end goal is to learn an optimized stochastic policy function $\pi(a|\mathbf{s}; \theta)$ (i.e., the actor). NeuroCuts first initializes all the parameters (line 1-6), and then runs for N rollouts to train the policy and the value function (line 7-23). After each rollout, it reinitializes the decision tree to the root node (line 9). It then incrementally builds the tree by repeatedly selecting and applying an action on each non-terminal leaf node (line 11-13) according to the current policy. A terminal leaf node is a node in which the number of rules is below a given threshold.

More specifically, NeuroCuts traverses the tree nodes in depth-first-search (DFS) order (line 13), i.e., it recursively cuts the child of the current node until the node becomes a

Action Space	Tuple(Discrete(NumDims), Discrete(NumCutActions + NumPartitionActions))
Observation Space	Box(low=0, high=1, shape=(278,))
Observation Components	$\big[$BinaryString($Range^{dim}_{min}$) + BinaryString($Range^{dim}_{max}$) + OneHot($Partition^{dim}_{min}$) + OneHot($Partition^{dim}_{max}$)$\big]$ $\forall dim \in \{SrcIP, DstIP, SrcPort, DstPort, Protocol\}$ + OneHot(EffiCutsPartitionID) + ActionMask

Table 5.1: NeuroCuts action and observation spaces described in OpenAI Gym format [13]. Actions are sampled from two categorical distributions that select the dimension and action to perform on the dimension respectively. Observations are encoded in a one-hot bit vector (278 bits in total) that describes the node ranges, partitioning info, and action mask (i.e., for prohibiting partitioning actions at lower levels). When not using the EffiCuts partitioner, the $Partition^{dim}$ rule dimension coverage thresholds are set to one of the following discrete levels: 0%, 2%, 4%, 8%, 16%, 32%, 64%, and 100%.

terminal leaf. Note that the DFS order is not essential. It is used to give a way for the agent to find a tree node to cut. Other orders, such as the breadth-first-search (BFS), can be used as well. After the decision tree is built, the gradients are reset (line 14), and then the algorithm iterates over all the tree nodes to aggregate the gradients (line 15-21). Finally, NeuroCuts uses the gradients to update the parameters of the actor and critic networks (line 22), and proceeds to the next rollout (line 23).

The first gradient computation (line 19) corresponds to that for the *policy gradient loss*. This loss defines the direction to update θ to improve the expected reward. An estimation of the state value $V(s;\theta_v)$ is subtracted from the rollout reward R to reduce the gradient variance [72]. V is trained concurrently to minimize its prediction error (line 21). Figure 5.5 visualizes the learning process of NeuroCuts to build a decision tree. The NeuroCuts policy is stochastic, enabling it to effectively explore many different tree variations during training, as illustrated in Figure 5.6.

Incorporating existing heuristics. NeuroCuts can easily incorporate additional heuristics to improve the decision trees it learns. One example is adding rule partition actions. In addition to the cut action, in our NeuroCuts implementation we also allow two types of partition actions:

1. **Simple**: the current node is partitioned along a single dimension using a learned threshold.

2. **EffiCuts**: the current node is partitioned using the EffiCuts partition heuristic [154].

Scaling out to handle large packet classifiers. The pseudocode in Algorithm 1 is for a single-

Algorithm 1 Learning a tree-generation policy using an actor-critic algorithm.

Input: The root node $\mathbf{s}^*$ where a tree always grows from.

Output: A stochastic policy function $\pi(a|\mathbf{s};\theta)$ that outputs a branching action $a \in \mathcal{A}$ given a node state $\mathbf{s}$, and a value function $V(\mathbf{s};\theta_v)$ that outputs a value estimate for a node state.

Main routine:

// **Initialization**
Randomly initialize the model parameters θ, θ_v
Maximum number of rollouts N
Coefficient $c \in [0, 1]$ that trades off classification time vs. space
Reward scaling function $f(x) \in \{x, \textsc{log}(x)\}$
$n \leftarrow 0$
// **Training**
while $n < N$ **do**
    $s \leftarrow \textsc{Reset}(\mathbf{s}^*)$
    // **Build a tree using the current policy**
    while $s \neq \textsc{Null}$ **do**
        $a \leftarrow \pi(a|s;\theta)$
        $s \leftarrow \textsc{GrowTreeDFS}(s, a)$
    Reset gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    for $(s, a) \in \textsc{TreeIterator}(\mathbf{s}^*)$ **do**
        // **Compute the future rewards for the given action**
        $R \leftarrow -(c \cdot f(\textsc{Time}(s)) + (1 - c) \cdot f(\textsc{Space}(s)))$
        // **Accumulate gradients wrt. policy gradient loss**
        $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a|s;\theta)(R - V(s;\theta_v))$
        // **Accumulate gradients wrt. value function loss**
        $d\theta_v \leftarrow d\theta_v + \partial(R - V(s;\theta_v))^2/\partial\theta_v$
    Perform update of θ using d_θ and θ_v using $d\theta_v$.
    $n \leftarrow n + 1$

Subroutines:

- $\textsc{Reset}$(s): Reset the tree s to its initial state.
- $\textsc{GrowTreeDFS}$(s, a): Apply action a to tree node s, and return the next non-terminal leaf node in the tree in depth-first traversal order.
- $\textsc{TreeIterator}$(s): Non-terminal tree nodes of the subtree s and their taken action.
- $\textsc{Time}$(s): Upper-bound on classification time to query the subtree s. In non-partitioned trees this is simply the depth of the tree.
- $\textsc{Space}$(s): Memory consumption of the subtree s.

threaded implementation of NeuroCuts. This is sufficient for small classifiers. But for large classifiers with tens or hundreds of thousands of rules, parallelism can significantly improve the speed of training. In Figure 5.7 we show how Algorithm 1 can be adapted to build multiple decision trees in parallel.

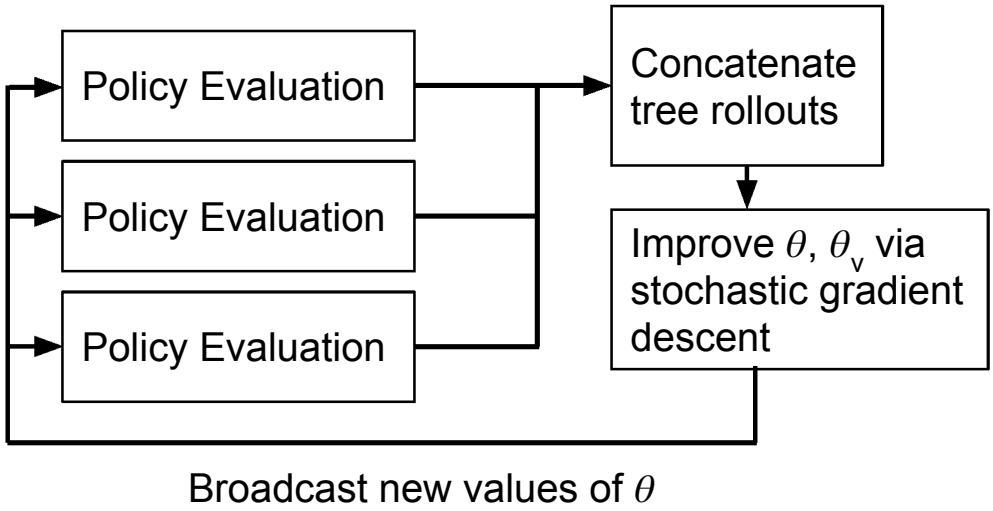


Figure 5.7: NeuroCuts can be parallelized by generating decision trees in parallel from the current policy.

Handling classifier updates. Packet classifiers are often updated by network operators based on application requirements, e.g., adding access control rules for new devices. For small updates of only a few rules, NeuroCuts modifies the existing decision tree to reflect the changes. New rules are added to the decision tree according to the existing structure; deleted rules are removed from the terminal leaf nodes. When enough small updates accumulate or a large update is made to the classifier, NeuroCuts re-runs training.

5.5 Implementation

Deep RL algorithms are notoriously difficult to reproduce [58]. For a practical implementation, we prioritize the ability to (i) leverage off-the-shelf RL algorithms, and (ii) easily scale NeuroCuts to enable parallel training of policies.

Decision tree implementation. We implement the decision tree data structure for NeuroCuts in Python for ease of development. To ensure minor implementation differences do not bias our results, we use this same data structure to implement each baseline algorithm (e.g., HiCuts, EffiCuts, etc.), as well as to implement NeuroCuts.

Branching decision process environment. As discussed in Section 5.4, the branching structure of the NeuroCuts environment poses a challenge due to its mismatch with the MDP formulation assumed by many RL algorithms. A typical RL environment defines a transition function $P_a(s_{t+1}|s_t)$ and a reward function $R_a(s, s')$. The first difference is that the state transition function in NeuroCuts returns multiple child states, instead of a single

state., i.e., $(s_t, a_t) \rightarrow \{s^0_{t+1}, ..., s^k_{t+1}\}$. Second, the final reward for NeuroCuts is computed by aggregating across the rewards of child states. More precisely, for the cut action we use `max` aggregation for classification time and `sum` aggregation for memory footprint. For the partition action, we use `sum` aggregation for both metrics.

The recursive dependence of the NeuroCuts reward calculation on all descendent state actions means that it is difficult to flatten the tree structure of the environment into a single MDP, which is required by existing off-the-shelf RL algorithms. Rather than attempting to flatten the NeuroCuts environment, our solution is to instead treat the NeuroCuts environment as a series of *independent* 1-step decision problems, each of which yields an "immediate" reward. The actual reward for these 1-step decisions is calculated once the relevant sub-tree rollout is complete.

For example, consider a NeuroCuts tree rollout from a root node s_1. Based on π_θ the agent decides to take action a_1 to split s_1 into s_2, s_3, and s_4. Of these child nodes, only s_4 needs to be further split (via a_2), into s_5 and s_6, which finishes the tree. The experiences collected from this rollout consist of two independent 1-step rollouts: (s_1, a_1) and (s_4, a_2). Taking the time-space coefficient $c = 1$ and discount factor $\gamma = 1$ for simplicity, the total reward R for each rollout would be $R = 2$ and $R = 1$ respectively.

Multi-agent implementation. Since these 1-step decisions are logically independent of each other, NeuroCuts execution can be realized as a multi-agent environment, where each node's 1-decision problem is taken by an independent "agent" in the environment. Since we want to learn a single policy, π_θ, for all states, the agents must be configured to share the same underlying stochastic neural network policy. This ensures all experiences go towards optimizing the single shared policy π_θ. When using an actor-critic algorithm to optimize the policies of such agents, the relevant loss calculations induced by this multi-agent realization are identical to those presented in Algorithm 1.

There are several ways to implement the 1-step formulation of NeuroCuts while leveraging off-the-shelf RL libraries. In Algorithm 1 we show standalone single-threaded pseudocode assuming a simple actor-critic algorithm is used. In our experiments, we use the multi-agent API provided by Ray RLlib [92], which implements parallel simulation and optimization of such RL environments.

Performance. We found that NeuroCuts often converges to its optimal solution within just a few hundred rollouts. The size of the rule set does not significantly affect the number of rollouts needed for convergence, but affects the running time of each rollout. For smaller problems (e.g., 1000 rules), this may be within a few minutes of CPU time. The computational overhead for larger problem scales with the size of the classifier, i.e., linearly with the number of rules that must be scanned per action taken to grow the tree. The bulk of time in NeuroCuts is spent executing tree cut actions. This is largely an artifact of our Python implementation, which iterates over each rule present in a node on each cut action. An optimized C++ implementation of the decision tree would further reduce the training time.

Optimizations

Rollout truncation. During the initial phase of learning, the unoptimized policy will create excessively large trees. Since NeuroCuts does not start learning until a tree is complete, it is necessary to truncate rollouts to speed up the initial phase of training. For larger classifiers, we found it necessary to allow rollouts of up to 15000 actions in length.

Depth truncation. Since valid solutions never involve trees of depth greater than a few hundred, we also truncate trees once they reach a certain depth. In our experience, depth truncation is only a factor early on in learning; NeuroCuts quickly learns to avoid creating very deep trees.

Proximal Policy Optimization. For better stability and more sample-efficient learning, in our experiments we choose to use Proximal Policy Optimization (PPO) [134]. PPO implements an actor-critic style loss with entropy regularization and a clipped surrogate objective, which enables improved exploration and sample efficiency. We report the PPO hyperparameters we used in Table 5.2. It is important to note however that this particular choice of RL algorithm is not fundamental to NeuroCuts.

5.6 Evaluation

In the evaluation, we seek to answer the following questions:

1. How does NeuroCuts compare to the state-of-the-art approaches in terms of classification time and memory footprint? (Section 5.6 and 5.6)
2. Beyond *tabula rasa* learning, can NeuroCuts effectively incorporate and improve upon pre-engineered heuristics? (Section 5.6)
3. How sensitive is NeuroCuts to the hyperparameters of the neural network architecture (Section 5.6), and the time-space coefficient c (Section 5.6)?

For the results presented in the next sections, we evaluated NeuroCuts within the space of hyperparameters shown in Table 5.2. We did not otherwise perform extensive hyperparameter tuning; in fact we use close to the default hyperparameter configuration of the PPO algorithm. The notable hyperparameters we swept over include:

- Allowed top-node partitioning (none, simple, and the EffiCuts heuristic), which strongly biases NeuroCuts towards learning trees optimized for time (none) vs space (EffiCuts), or somewhere in the middle (simple).
- The max number of timesteps allowed per rollout before truncation. It must be large enough to enable solving the problem, but not so large that it slows down the initial phase of training.

Hyperparameter	Value
Time-space coefficient c	¡set by user¿
Top-node partitioning	{none, simple, EffiCuts}
Reward scaling function f	{x, LOG(x)}
Max timesteps per rollout	{1000, 5000, 15000}
Max tree depth	{100, 500, inf}
Max timesteps to train	10000000
Max timesteps per batch	60000
Model type	fully-connected
Model nonlinearity	tanh
Model hidden layers	{256x256, 512x512}
Weight sharing between θ, θ_v	{true, false}
Learning rate	0.00005
Discount factor γ	1.0
PPO entropy coefficient	0.01
PPO clip param	0.3
PPO VF clip param	10.0
PPO KL target	0.01
SGD iterations per batch	30
SGD minibatch size	1000

Table 5.2: NeuroCuts hyperparameters. Values in curly braces denote a space of values searched over during evaluation. We found that the most sensitive hyperparameter is the top-node partitioning, which greatly affects the structure of the search problem. It is also important to ensure that the rollout timestep limit and model used are sufficiently large for the problem.

- We also experimented with values for the time-space tradeoff coefficient $c \in \{0, 0.1, 0.5, 1\}$. When $c < 1$, we used $\text{LOG}(x)$ as the reward scaling function to simplify the combining of the time and space rewards.

We ran NeuroCuts on m4.16xl AWS machines, with four CPU cores used per NeuroCuts instance to speed up the experiment. Because the neural network model and data sizes produced by NeuroCuts are quite small (e.g., in contrast to image observations from Atari games), the use of GPUs is not necessary. Our main training bottleneck was the Python implementation of the decision tree. We ran each NeuroCuts instance for up to 10 million timesteps (i.e., up to a couple thousand generated trees in total), or until convergence.

We compare NeuroCuts with four hand-tuned algorithms: HiCuts [53], HyperCuts [140], EffiCuts [154], and CutSplit [89]. We use the standard benchmark, ClassBench [148], to generate packet classifiers with different characteristics and sizes. The benchmark metrics are those from prior work: classification time (tree depth) and memory footprint (bytes per

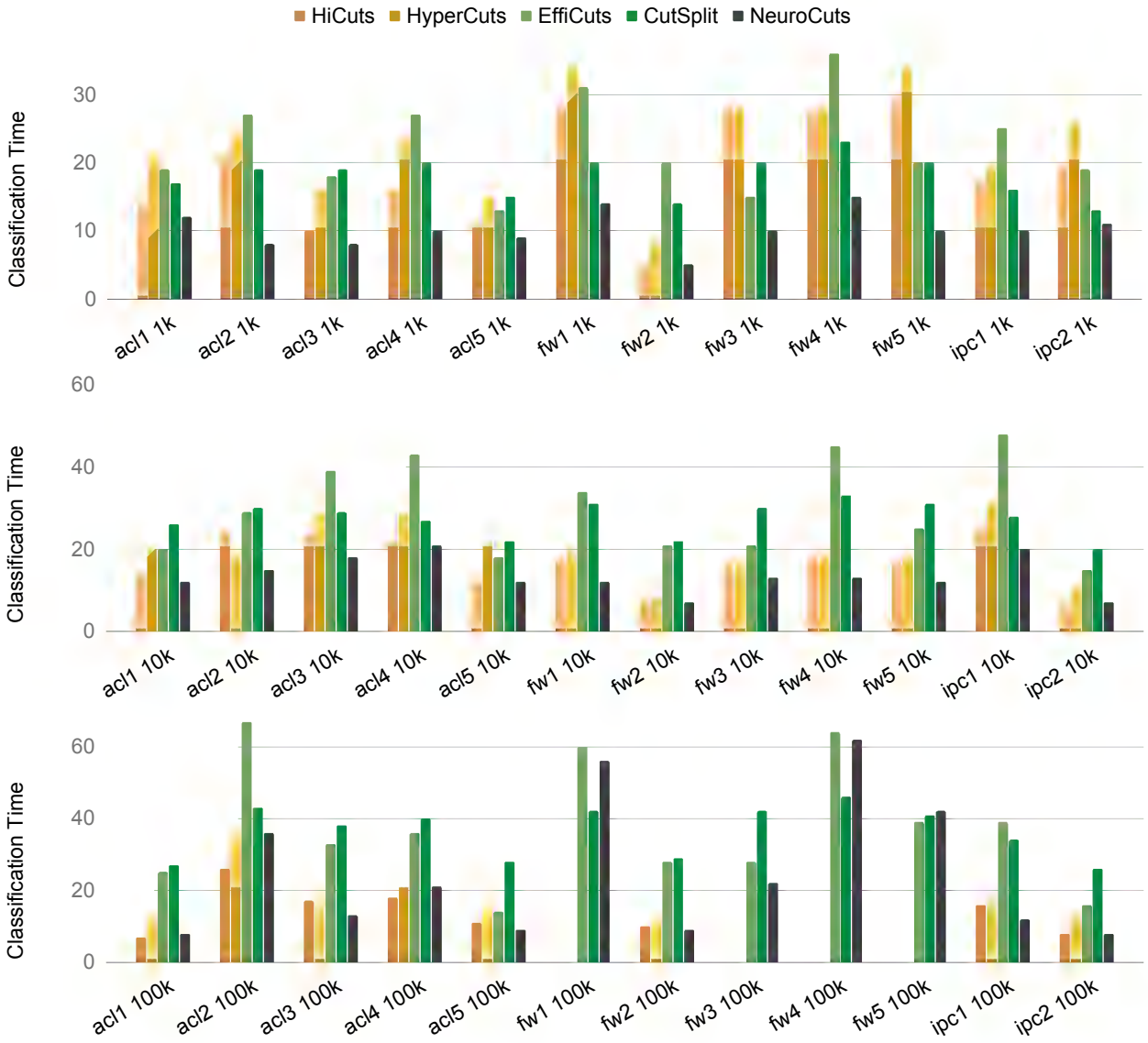


Figure 5.8: Classification time (tree depth) for HiCuts, HyperCuts, EffiCuts, and NeuroCuts (time-optimized) across the (1k, 10k, and 100k) sized rulesets. We omit four entries for HiCuts and HyperCuts that did not complete after more than 24 hours.

rule). Since we use the same underlying tree data structure for all algorithms, a lesser depth virtually guarantees more efficient traversal, and the same is true for memory footprint.

We find that NeuroCuts significantly improves over all baselines in classification time while also generating significantly more compact trees. NeuroCuts is also competitive when optimizing for memory, with a 25% median space improvement over EffiCuts without compromising in time.

Figure 5.9: Memory footprint (bytes per rule) used for HiCuts, HyperCuts, EffiCuts, and NeuroCuts (space-optimized) across the (1k, 10k, and 100k) sized rulesets. We omit four entries for HiCuts and HyperCuts that did not complete after more than 24 hours.

Time-Optimized NeuroCuts

In Figure 5.8, we compare the best time-optimized trees generated by NeuroCuts against HiCuts, HyperCuts, EffiCuts, and CutSplit in the ClassBench classifiers. NeuroCuts provides a 20%, 38%, 52% and 56% median improvement over HiCuts, HyperCuts, EffiCuts, and CutSplit respectively. NeuroCuts also does better than the minimum of all baselines in 70% of the cases, with a median all-baseline improvement of 18%, average improvement of 12%, and best-case improvement of 58%. These time-optimized trees generally correspond

to NeuroCuts runs with either no partitioning action or the simple top-node partitioning action.

Space-Optimized NeuroCuts

We again compare NeuroCuts against the baselines in Figure 5.9, this time selecting the most space-optimized trees and comparing the memory footprint (bytes per rule). As expected, NeuroCuts does significantly better than HiCuts and HyperCuts since it can learn to leverage the partition action. NeuroCut's space-optimized trees show a 40% median and 44% mean improvement over EffiCuts. In our experiments NeuroCuts does not usually outperform CutSplit in memory footprint, with a 26% higher median memory usage compared to CutSplit, though the best case improvement is still 3× (66%) *over all baselines*.

Separately, we also note that the memory footprints of the best *time-optimized trees* generated by NeuroCuts are significantly lower than those generated by HiCuts and HyperCuts, with a ¿100× *median* space improvement along with the better classification times reported in Section 5.6. However, these time-optimized trees are not competitive in space with the space-optimized NeuroCuts, EffiCuts and CutSplit trees.

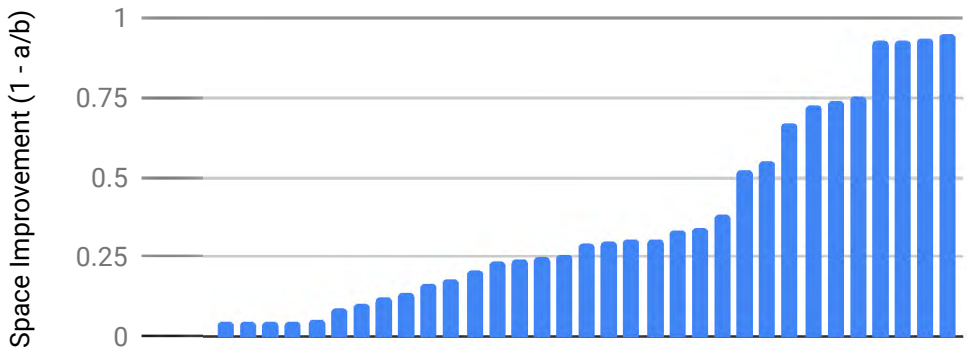
Improving on EffiCuts

In Figure 5.10 we examine a set of 36 NeuroCuts trees (one tree for each ClassBench classifier) generated by NeuroCuts with the EffiCuts partition action. This is in contrast with the prior experiments that selected trees optimized for either space or time alone. On this 36-tree set, there is a median space improvement of 29% relative to EffiCuts; median classification time is about the same. This shows that NeuroCuts is able to effectively incorporate and improve on pre-engineered heuristics such as the EffiCuts top-level partition function.

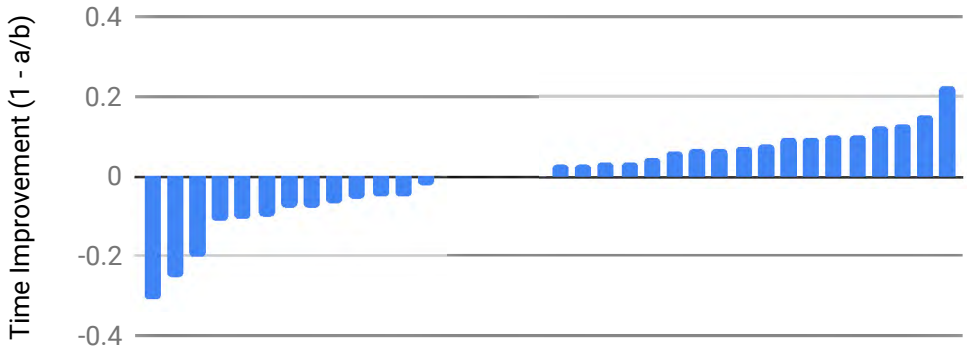
Surprisingly, NeuroCuts is able to outperform EffiCuts despite the fact that NeuroCuts does not use multi-dimensional cut actions. When we evaluate EffiCuts with these cut types disabled, the memory advantage of NeuroCuts widens to 67% at the median. This suggests that NeuroCuts could further improve its performance if we also incorporate multi-dimensional cut actions via parametric action encoding techniques [41]. It would also be interesting to, besides adding actions to NeuroCuts, consider postprocessing steps such as resampling that can be used to further improve the stochastic policy output.

Neural Network Architecture

To better understand the influence of the neural network architecture on NeuroCuts performance, we conduct an ablation study where the network size is reduced from 512x512 (hundreds of thousands of parameters) all the way down to 16x16 (a couple hundred parameters). We also consider the case where the network is trivial and does not process the observation at all, similar to a non-contextual bandit. For this study we run a single sweep



(a) NeuroCuts can build on the EffiCuts partitioner to generate trees up to 10× (90%) more space efficient than EffiCuts. In this experiment NeuroCuts did as well or better than EffiCuts on all 36 rule sets.



(b) NeuroCuts with the EffiCuts partitioner generates trees with about the same time efficiency as EffiCuts.

Figure 5.10: Sorted rankings of NeuroCuts' improvement over EffiCuts in the ClassBench benchmark. Here NeuroCuts is run with only the EffiCuts partition method allowed. Positive values indicate improvements.

across only these architecture hyperparameters, keeping all the others fixed, and use the simple partition method.

The results are shown in Figure 5.11. We observe that while the larger 64x64 network consistently outperforms 16x16, at 512x512 performance starts to be impacted due to the larger number of learnable parameters.[3] Interestingly, while the bias-only network did the worst, it still was able to generate reasonably compact trees in many cases. This suggests that NeuroCuts may operate by first learning a random distribution of actions that leads to a basic solution, and then leveraging the capacity of its neural network to specialize the

[3]We note that these results might not hold for different hyperparameters, e.g., if allowed longer training periods, larger networks may dominate.

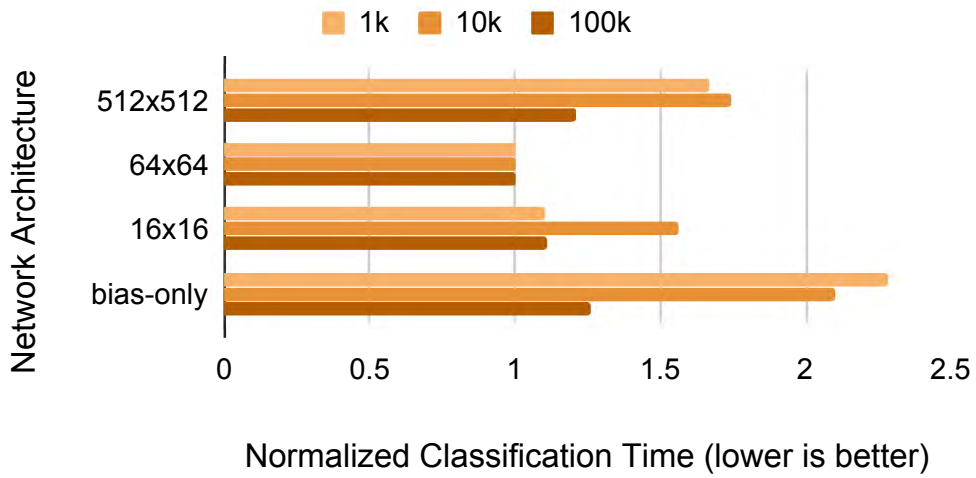


Figure 5.11: Comparison of the mean best classification time achieved by NeuroCuts across different network architectures and groups of classifiers. The *bias-only* architecture refers to a trivial neural network that does not process the observation at all and emits a fixed action probability distribution (i.e., a pure bandit). Results are normalized within classifier groups so that the best tree has a normalized time of 1. Rulesets that did not converge to a valid tree were assigned a time of 100 prior to normalization.

action distribution to different portions of the rule space.

Tuning Time vs Space

Finally, in Figure 5.12 we sweep across a range of values of c for NeuroCuts with the simple partition method and LOG(x) reward scaling. We plot the ClassBench median of the best classification times and bytes per rule found for each classifier. We find that classification time improves by $2\times$ as $c \to 1$, while the number of bytes per rule improves $2\times$ as $c \to 0$. This shows that c is effective in controlling the tradeoff between space and time.

5.7 Related Work

Packet classification. Packet classification is a long-standing problem in computer networking. Decision-tree based algorithms are a major class of algorithmic solutions. Existing solutions rely on hand-tuned heuristics to build decision trees. HiCuts [53] is a pioneering work in this space. It cuts the space of each node in one dimension to create multiple equal-sized subspaces to separate rules. HyperCuts [140] extends HiCuts by allowing cutting in multiple dimensions at each node. HyperSplit [124] combines the advantages of rule-based space decomposition and local-optimized recursion to guarantee worst-case classification time and reduce memory footprint. EffiCuts [154] introduces four heuristics, including separable

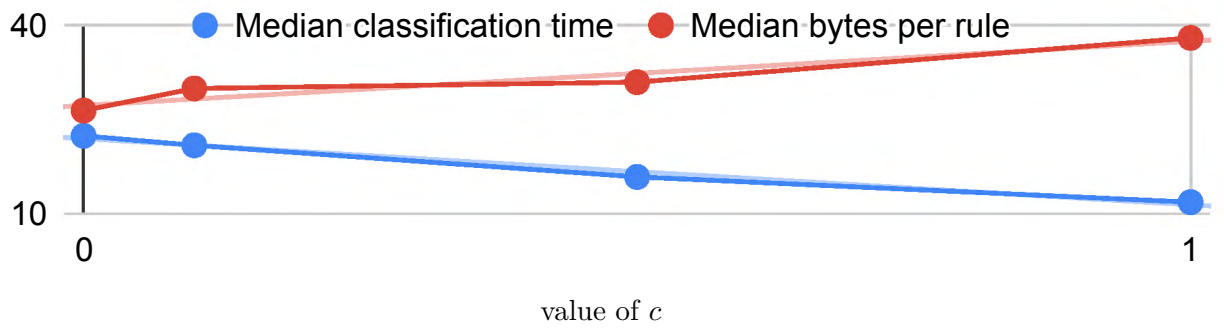


Figure 5.12: The classification time improves by 2× as the time-space coefficient $c \to 1$, and conversely, number of bytes per rule improves 2× as $c \to 0$.

trees, tree merging, equal-dense cuts and node co-location, to reduce rule replication and imbalance cutting. CutSplit [89] integrates equal-sized cutting and equal-dense cutting to optimize decision trees. Besides decision-tree based algorithms, there are also other algorithms proposed for packet classification, such as tuple space search [142], RFC [52] and DCFL [149]. These algorithms are not as popular as decision-tree based algorithms, because they are either too slow or consume too much memory. There are also solutions that exploit specialized hardware such as TCAMs, GPUs and FPGAs to support packet classification [141, 98, 69, 80, 94, 157, 146, 125]. Compared to existing work, NeuroCuts is an algorithmic solution that applies Deep RL to generate efficient decision trees, with the capability to incorporate and improve on existing heuristics as needed.

Decision trees for machine learning. There have been several proposals to use deep learning to optimize the performance of decision trees for machine learning problems [112, 163, 73]. In these settings, the objective is maximizing test accuracy. In contrast, packet classification decision trees provide perfect accuracy by construction, and the objective is minimizing classification time and memory usage.

Structured data in deep learning. There have many recent proposals towards applying deep learning to process and generate tree and graph data structures [178, 171, 50, 164, 160, 165]. NeuroCuts sidesteps the need to explicitly process graphs, instead exploiting the structure of the problem to encode agent state into a compact fixed-length representation.

Deep reinforcement learning. Deep RL leverages the modeling capacity of deep neural networks to extend classical RL to domains with large, high-dimensional state and action spaces. DQN [106, 156] is one of the earliest successes of Deep RL, and shows how to learn control policies from high-dimensional sensory inputs and achieve human-level performance in Atari 2600 games. A3C, PPO, and IMPALA [105, 134, 36] scale actor-critic algorithms to leverage many parallel workers. AlphaGo [138], AlphaGo Zero [139] and AlphaZero [137] show that Deep RL algorithms can achieve superhuman performance in many challenging

games like Go, chess and shogi. Deep RL has also been applied to many other domains like natural language processing [86] and robotics [83, 84, 82]. NeuroCuts works in a discrete environment and applies Deep RL to learn decision trees for packet classification.

Deep learning for networking and systems. Recently there has been an uptake in applying deep learning to networking and systems problems [177, 170, 99, 153, 22, 67, 30, 176, 101]. NAS [170] utilizes client computation and deep neural networks to improve the video quality independent to the available bandwidth. Pensieve [99] generates adaptive bitrate algorithms using Deep RL without relying on pre-programmed models or assumptions about the environment. Valadarsky [153] applies Deep RL to learn network routing. Chinchali [22] uses Deep RL for traffic scheduling in cellular networks. AuTO [19] scales Deep RL for datacenter-scale traffic optimization. There are also many solutions that apply deep reinforcement learning to congestion control [67, 30, 176] and resource management [101]. We explore the application of Deep RL to packet classification, and propose a new algorithm to learn decision trees with succinct encoding and scalable training mechanisms.

5.8 Conclusion

We present NeuroCuts, a simple and effective Deep RL formulation of the packet classification problem. NeuroCuts provides significant improvements on classification time and memory footprint compared to state-of-the-art algorithms. It can easily incorporate pre-engineered heuristics to leverage their domain knowledge, optimize for flexible objectives, and generates decision trees which are easy to test and deploy in any environment.

We hope NeuroCuts can inspire a new generation of learning-based algorithms for packet classification. As a concrete example, NeuroCuts currently optimizes for the worst-case classification time or memory footprint. By considering a specific traffic pattern, NeuroCuts can be extended to other objectives such as average classification time. This would allow NeuroCuts to not only optimize for a specific classifier but also for a specific traffic pattern in a given deployment.

Chapter 6

Application: Accelerating Database Cardinality Estimation

As with applied machine learning, applied RL practitioners often find the hardest part of solving the problem is coming up with the right problem framing. In RL, this means designing the environment and objective to optimize. In Chapter 5 we saw that an efficient state space formulation and problem decomposition led to state of the art results. In this chapter we'll dive deeply into one part of the puzzle of solving *database query optimization* with RL. The problem is complex—decades of study have gone into database query optimizers. We encounter a similar depth of obstacles when looking at query optimization from the RL perspective. Not only does one need to define the environment, which is challenging given the complex, tree-structured nature of database query plans, one quickly finds that executing simulations can be prohibitively expensive.

One approach of mitigating the expense of executing real database queries for training is to take a model-based approach—try to learn a simulator for the database. In the databases domain, this amounts to learning a compressed representation of the data distribution held by the database, which we use to infer the magnitude of data that flows through each node of a database query plan. This chapter introduces a practically fast inference method that can be used to accelerate RL training on databases.

6.1 Introduction

Deep autoregressive models (AR) compute *point* likelihood estimates of individual data points. However, many applications (i.e., database cardinality estimation) require estimating *range* densities, a capability that is under-explored by current neural density estimation literature. In these applications, fast and accurate range density estimates over high-dimensional data directly impact user-perceived performance. In this chapter, we explore a technique, *variable skipping*, for accelerating range density estimation over deep autoregressive models. This technique exploits the sparse structure of range density queries to avoid sampling un-

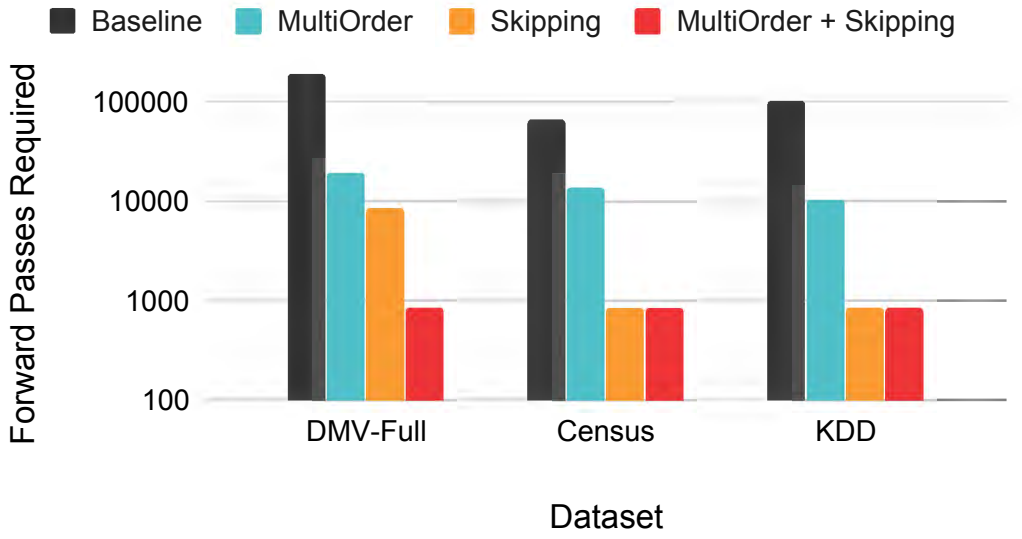


Figure 6.1: Approximate number of model forward passes required to achieve single-digit inference error at the *99th quantile*. Y-axis shown in *log scale*, lower is better. Variable skipping provides 10-100× compute savings for challenging high-quantile error targets. Refer to the Evaluation section for full results.

necessary variables during approximate inference. We show that variable skipping provides 10-100× efficiency improvements when targeting challenging high-quantile error metrics, enables complex applications such as text pattern matching, and can be realized via a simple data augmentation procedure without changing the usual maximum likelihood objective.

Deep AR models have achieved state-of-the-art density estimation results in image, video, and audio [130, 113, 155, 21, 126, 161]. Recent work has applied them to domains traditionally outside of machine learning, such as physics [136], protein modeling [127], and database query optimization [167, 168]. These use cases have surfaced the need for complex inference capabilities from deep AR models. For example, the database cardinality estimation task reduces to estimating the density mass occupied by sets of variables under sparse *range constraints*. In this problem, the database optimizer probes the fraction of records satisfying a query of high-dimensional constraints, e.g., $\Pr(\mathsf{age} > 35 \;\&\&\; \mathsf{salary} < 50K)$, and relies on accurate estimates to pick performant query execution strategies.

In this chapter, we call for attention to such *range density estimation* problems in the context of deep AR models. Given rapid advances in model capabilities, fast and accurate range density estimation has broad potential applicability to a number of domains, including databases, text processing, and inpainting (Section 6.1).

Range density estimation involves two related challenges: nolistsep

- **Marginalization**: the handling of unconstrained variables, and
- **Range Constraints**: variables that are constrained to a specific *range* or *subset* of values.

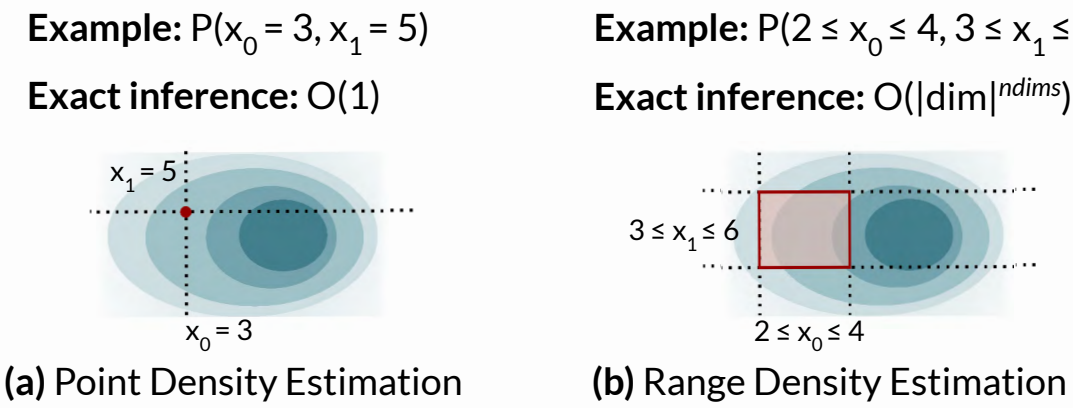


Figure 6.2: Comparison of point density and range density estimation. Naive marginalization to estimate range densities takes time proportional to the size of the query region (i.e., exponential in the number of dimensions of the joint distribution).

Exact inference or integration over the query region takes time exponential in the number of dimensions—a cost too high for all but the tiniest problems. Further, both marginalization and range constraints are difficult to implement on top of AR models since they are only trained to provide point density estimates. This motivates the use of approximate inference algorithms such as recently proposed by [167], which show that AR models can significantly improve on the state-of-the-art in range estimation accuracy while remaining competitive in latency.

Building on prior work, we distill and evaluate a more general optimization for accelerating range density estimation termed *variable skipping*. The central idea is to exploit the sparsity of range queries, by avoiding sampling through the unconstrained dimensions (i.e., those to be marginalized over) during approximate inference. A training-time data augmentation procedure randomly replaces some dimensions in the input with learnable marginalization tokens, which are trained to represent the *absence* of those dimensions. During inference, the unconstrained dimensions take on these learned values instead of being sampled from their respective domains.

Variable skipping provides two key advantages. First, by not needing to sample a concrete value for certain variables, the number of forward passes is significantly reduced from $O(|\text{Vars}|)$ (e.g., hundreds) to $O(|\text{ConstrainedVars}|)$ (e.g., a few). Second, by avoiding sampling through the (potentially large) unconstrained region, it is possible to reduce the variance of the sampling-based estimator. We show that variable skipping realizes both advantages in practice (Figure 6.1).

Reducing the computation required for estimates can significantly impact the viability of model-based estimators for the aforementioned computer systems applications. For example, in database query optimization, cardinality estimation is typically run in the inner loop of a dynamic program [135], and hence has to be executed many times in potentially unbatchable

fashion. Further, this process must be re-run for each new query as it may have different variable constraints. In this setting, reducing estimation costs from tens or hundreds of forward passes (i.e., the number of columns in a typical production database) to just a handful (i.e, the number of constraints in a typical range query) is critical for adoption. Models that include rarely queried text columns (e.g., byte pair encoded, which exacerbates the problem) may benefit further still.

We start by discussing related work, then reviewing the previously proposed approximate inference algorithm [167], termed Progressive Sampling (Section 6.3), which allows any trained autoregressive model to efficiently compute range densities. We then discuss an optimization, *variable skipping*, which allows dimensions irrelevant to a query to be skipped over at inference time, greatly reducing or eliminating sampling costs (Section 6.4). We show that, beyond accelerating range density estimation, variable skipping can enable related applications such as pattern matching and text completion. Finally, we study the performance of variable skipping (Section 6.5).

Our contributions are as follows:

1. We distill the more general concept of *variable skipping*, a training and run-time optimization that greatly reduces the variance of range density estimates.

2. To show its generality, we apply variable skipping to text models, which can then support applications such as pattern matching.

3. We evaluate the effectiveness of variable skipping across a variety of datasets, architecture, and hyperparameter choices, and compare with related techniques such as multi-order training.

4. To invite research on this under-explored problem, we open source our code and a set of range density estimation benchmarks on high-dimensional discrete datasets at `https://var-skip.github.io`.

Applications of Range Density Estimation

Range density estimation is important for the following applications, among others:

Database Systems: A core primitive in database query optimizer is cardinality estimation [135]: given a query with user-defined predicates for a subset of columns, estimate the *fraction* of records that satisfy the predicates. Applying AR models to cardinality estimation was the topic of [167].

Pattern Matching: A regular expression can be interpreted as a dynamically unrolled predicate (i.e., a nondeterministic finite automata) [62] over a series of character variables. Hence, its *match probability* can be estimated in the same way as a range query. Section 6.4 shows how this can be realized with variable skipping.

Completion and Inpainting: While an AR model can be straightforwardly used to extend a prefix in the variable ordering, completing a missing value from the *middle* of a sequence of variables requires sampling from the marginal distribution over missing values. We show that variable skipping allows this to be done efficiently (Section 6.4).

6.2 Related Work

Density Estimation with Deep Autoregressive Models have enjoyed vast interest due to their outstanding capability of modeling high-dimensional data (text, images, tabular). Efficient architectures such as MADE [42] and ResMADE [35] have been proposed, and self-attention models (e.g., Transformer [158]) have underpinned recent new advances in language tasks. Our work optimizes the approximate inference (of range density estimates) on top of such AR architectures.

Masked Language Models . Our variable skipping learns special MASK tokens (Section 6.4) by randomly masking inputs, which is similar to masked language models such as BERT [28] and CMLMs [43]. These models differ from AR models in optimization goals: they typically predict *only* the masked tokens conditioned on present tokens, and may assume independence among the masked tokens. We study deep AR models for two reasons: (1) our problem settings are in density estimation, and deep AR models have generally shown superior density modeling quality than other generative models; (2) the approximate inference procedure we study (Section 6.3) assumes access to autoregressive factors.

Multi-Order Training handles marginalization by training over many orders and invoking a suitable order (or an ensemble over available orders) during inference. This technique has appeared in NADE [152], MADE [42], XLNet [166], among others. Variable skipping shares the same goal of efficiently handling marginalization. These prior works have reported increased optimization difficulty as the number of orders to learn increases (some sample a fixed set of orders, while others keep sampling new orders). In the latter case, we posit that the difficulty is due to adding $n!$ input variations; in contrast, variable skipping only extends the vocabulary of each dimension by a MASK symbol, a relatively smaller increase in task difficulty. In Section 6.5, we compare variable skipping against multi-order training, and show that they can be combined to further reduce errors.

6.3 Range Density Estimation on Deep Autoregressive Models

We model a finite set of n-dimensional data points $D = \{x^{(0)}, x^{(1)}, \ldots, x^{(M)}\}$ as a discrete distribution using an autoregressive model $p_\theta(x)$, parameterized by θ. The model is trained

on D using the maximum likelihood objective:

$$\min_{\theta} \mathcal{L}(\theta) = - \mathop{\mathbb{E}}_{x \sim D} \log p_\theta(x) \tag{6.1}$$

where $p_\theta(x) = \prod_i p_\theta(x_i | x_{<i})$ for each data point x.

Range density. We consider range queries of the form

$$p_\theta(X_1 \in R_1, \ldots, X_n \in R_n), \tag{6.2}$$

where each region R_i is a subset of the domain (X_i). This formulation encapsulates unconstrained dimensions, where we simply take $R_i = (X_i)$ (the whole domain).

Background: Progressive Sampling

Exact inference of Equation 6.2 is computationally efficient only for low dimensions or small domain sizes. Approximate inference is required to scale its computation.

To solve this problem, [167] adapts classical forward sampling [70] for range likelihoods, yielding an unbiased approximate inference algorithm. The algorithm works by drawing in-range samples and re-weighting each intermediate range likelihood. Each in-range sample is drawn from the first dimension to the last (in the AR ordering). As an example, consider estimating $p_\theta(X_1 \in R_1, X_2 \in R_2, X_3 \in R_3)$. Progressive sampling draws $x_1^{MC} \sim p_\theta(X_1 | X_1 \in R_1)$ and stores $p_\theta(X_1 \in R_1)$—both tractable operations since a forward pass on the trained AR produces this single-dimensional distribution. It performs another forward pass to obtain $p_\theta(X_2 | x_1^{MC})$, which then produces a sample x_2^{MC} and the range likelihood $p_\theta(X_2 \in R_2 | x_1^{MC})$. Lastly it obtains $p_\theta(X_3 \in R_3 | x_1^{MC}, x_2^{MC})$. It can be shown that the product of all n range likelihoods, e.g.,

$$p_\theta(X_1 \in R_1) p_\theta(X_2 \in R_2 | x_1^{MC}) p_\theta(X_3 \in R_3 | x_1^{MC}, x_2^{MC})$$

is a valid Monte-Carlo estimate of the desired range density.

In the remainder of the paper, we invoke $(R_1, \ldots, R_n)$ as a black-box estimator, although our variable technique (described next) can work with other estimators for Equation 6.2.

6.4 Variable Skipping

Variable skipping works by (1) training special marginalization tokens, $_i$, for each dimension i; (2) at approximate inference, rewriting each unconstrained variable, e.g., $X_i \in (X_i)$, into a constrained variable with the singleton range, $X_i \in \{_i\}$. The training process can be interpreted as dropout of the input, or as data augmentation.

Architecture. We assume a model architecture shown in Figure 6.3: the input layer, an autoregressive core, and the output layer. For the autoregressive core, we use ResMADE [35] for tabular data and an autoregressive Transformer [158] (encoder only with correct masking) for text data. At the input layer, we embed each data point using a per-dimension trainable embedding table, denoted by (). (For text, we tie the embeddings across all dimensions since they share the same character vocabulary.) The output layer dots the hidden features with the input embeddings to produce logits.

Training-time input masking. First, we add a special token $_i$ to each dimension i's vocabulary. For each input x we uniformly draw the number of masked dimensions $n_{\text{mask}} \sim [0, |x|)$, then sample the n_{mask} positions to mask, X_{mask}. For position $i \in X_{\text{mask}}$, we replace the original representation, (x_i), by the masked representation, $(_i)$:

$$\begin{aligned}(x_i) =&(x_i)1(i \notin X_{\text{mask}}) \\ &+ (_i)1(i \in X_{\text{mask}}).\end{aligned} \tag{6.3}$$

Importantly, the objective remains the MLE for *all* autoregressive factors: we train the parameters to predict the *original values* at each dimension, given *a mix of original and masked information* at previous dimensions. In other words, we minimize the negative log-likelihood

$$-\log p_\theta(x_i|x_{<i}) = -\log p_\theta\left(x_i|\forall j < i, (x_j)\right) \tag{6.4}$$

over all dimension i. Conditioning on the mask tokens ensures that those representations are trained. Since we do not alter the *output targets* and the mask positions are chosen independently of the data, no bias is introduced.

Infer-time skipping. Given a range query (Equation 6.2), we look for each unconstrained dimension and replace its domain with a singleton set of its marginalization token:

$$\begin{aligned}& p_\theta(\dots, X_i \in R_i = (X_i), \dots) \\ \rightarrow\quad & p_\theta(\dots, X_i \in R_i' = \{\text{MASK}_i\}, \dots)\end{aligned} \tag{6.5}$$

We then invoke () which would thus *skip* the sampling for those dimensions.

Example. Suppose we have an AR model trained over the autoregressive ordering $\{\text{age}, \text{salary}, \text{city}\}$, and want to draw a sample from $p_\theta(\text{city}|\text{salary} > 50\text{k})$.

- Without skipping, first we draw $x_1 \sim p_\theta(\text{age})$, then $x_2 \sim p_\theta(\text{salary}|\text{age} = x_1, \text{salary} > 50\text{k})$, and finally $x_3 \sim p_\theta(\text{city}|\text{age} = x_1, \text{salary} = x_2)$.
- With skipping, we can directly sample $x_2 \sim p_\theta(\text{salary}|\text{age} = \text{MASK}_1, \text{salary} > 50\text{k})$, followed by $x_3 \sim p_\theta(\text{city}|\text{age} = \text{MASK}_1, \text{salary} = x_2)$.

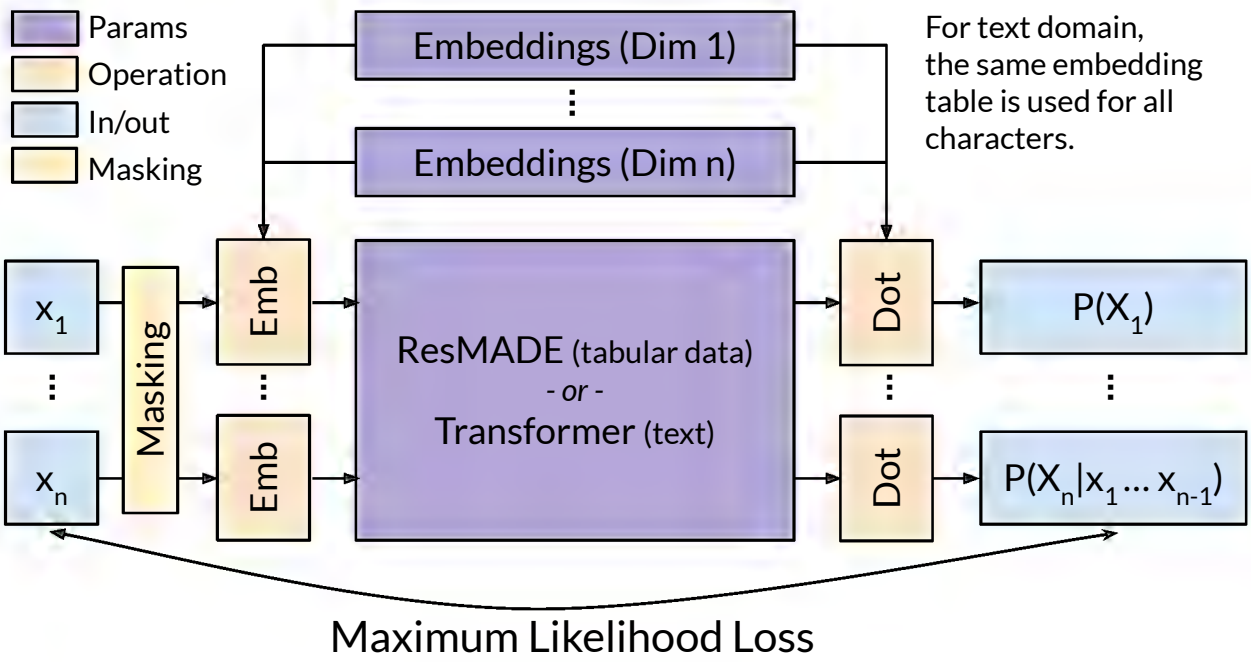


Figure 6.3: Model architecture.

Prefix Skipping for Text Pattern Matching

Any regex can be implemented as a nondeterministic finite automata (NFA) [62], which takes a stream of characters and determines acceptable next characters. We can use progressive sampling with any regex, treating its NFA like a dynamically unrolled predicate. For example, consider the regex $[\texttt{at}](\texttt{c} + |\texttt{i}+)\texttt{e}$. Possible matches include `ace`, `aiie`, and `tiiie`. Progressive sampling would work as follows: first we sample $x_1 \in \{a, t\}$, then $x_2 \in \{c, i\}$. Depending on whether $x_2 = c$ or $x_2 = i$, third we either sample $x_3 \in \{c, e\}$ or $\{i, e\}$ (this is the "dynamic" part), and so on. By retaining an NFA per sample, we obtain an estimate of the overall match probability.

However, this naive formulation is inefficient when there are long unconstrained sequences. Consider the regex $.*\texttt{icml}.*$, intended to match any string containing the token `icml`. The probability of sampling a random prefix from an AR model matching this is vanishingly small—perhaps millions of samples for a hit. To avoid this, we can try to *skip over* sequences of unconstrained characters and compute the probability of `icml` at specific offsets directly. All that would remain is sampling forward through the remainder of the variables to avoid double counting duplicate occurrences of the token. Using $m(x_i)$ to denote a string match at position i, and $m(x_{>i})$ the existence of a match at any position $> i$, the match

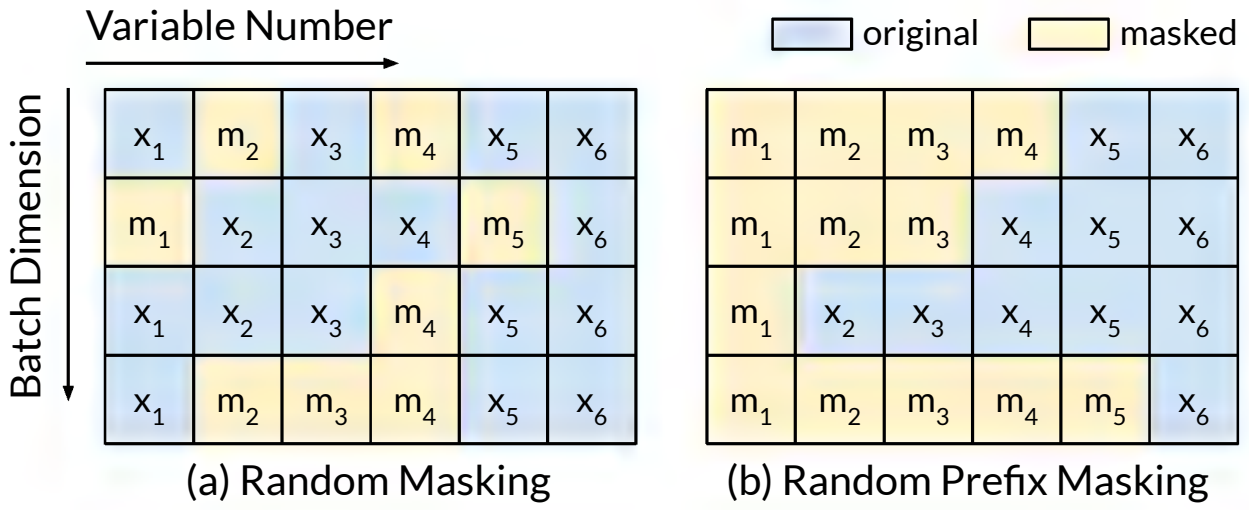


Figure 6.4: Masking strategies (Section 6.4). *(a)* For tabular data, we randomly sample the dimensions to mask out for each row. *(b)* For text, we mask a random prefix of each string, exploiting the natural left-to-right ordering.

probability is approximated as:

$$\begin{aligned} P_{\text{match}} &= \sum_{i=1}^{n} P(m(x_i)) \cdot (1 - P(m(x_{>i})|m(x_i)) \\ &\approx \sum_{i=1}^{n} p_\theta(m(x_i)|\{x_j = \text{MASK}_j : j < i\}) \\ &\quad (1 - p_\theta(m(x_{>i})|\{x_j = \text{MASK}_j : j < i\}, m(x_i)) \end{aligned} \tag{6.6}$$

Due to the need for masking contiguous prefixes, the model is trained with *random prefix masking* (Figure 6.4) to allow such contiguous characters to be skipped. We show the effectiveness of this strategy in Section 6.5, which implements simple pattern queries over an AR Transformer model.

Other Mask Patterns

Finally, we note that more structured mask patterns can be used, such as sub-sequences in text or random patches in images [34]. This allows for marginalization over complex subsets of dimensions with potential applicability to not only sample variables given prefixes of the AR ordering (i.e., from $P(x_i|x_1, \ldots, x_{i-1})$) but also variables *later on*, i.e., from $P(x_i|x_1, \ldots, x_{i-1}, x_{i+k}, \ldots, x_N)$ by marginalizing over $\{x_{i+1}, \ldots, x_{i+k-1}\}$. We leave investigation of these potential applications to future work.

Table 6.1: Datasets used in evaluation. "Domain" refers to the range of distinct values per table column (i.e., Dryad-URLs contains 78 different character values).

Dataset	Rows	Cols	Domain	Type
DMV-Full	11.6M	19	2–32K	Discrete
Census	2.5M	67	2–18	Discrete
KDD	95K	100	2–896	Discrete
Dryad-URLs	2.4M	100	78	Text

6.5 Evaluation

Our evaluation investigates the following questions:

1. How much does variable skipping improve estimation accuracy compared to baselines, and how is this impacted by the sampling budget?
2. Can variable skipping be combined with multi-order training to further improve accuracy?
3. To what extent do hyperparameters such as the model capacity and mask token distribution impact the effectiveness of variable skipping?
4. Can variable skipping be applied to related domains such as text, or is it limited to tabular data?

Overall, we find that variable skipping robustly improves estimation accuracy across a variety of scenarios. Given a certain target accuracy, skipping reduces the required compute cost by one to two orders of magnitude.

Datasets

We use the following public datasets in our evaluation, also summarized in Table 6.1. When necessary, we drop columns representing continuous data. We consider supporting continuous variables an orthogonal issue, and limit our evaluation to discrete domains:

DMV-Full [143]. Dataset consisting of vehicle registration information in New York (i.e., attributes like vehicle class, make, model, and color). We use all columns except for the unique vehicle ID (VIN). This dataset was also used in [167], but there it was restricted to 11 of the smaller columns.

KDD [31]. KDD Cup 1998 Data. We used the first hundred columns, sans noexch, zip, and pop901-3, which were especially high-cardinality. This leaves 100 discrete integer domains with 2 to 896 distinct values each.

Table 6.2: Hyperparameters for all experiments. We used a ResMADE for tabular data, and a Transformer for text.

Hyperparameter	Value
Training Epochs	20 (200 for KDD)
Batch Size	2048
Architecture	ResMADE
Residual Blocks	3
Hidden Layers / Block	2
Hidden Layer Units	256
Embedding Size	32
Optimizer	Adam
Learning Rate	5e-4
Learning Rate Warmup	1 epoch
Mask Probability	$\sim$ Uniform$[0, 1)$
Transformer Num Blocks	8
Transformer MLP Dims (d_{ff})	256
Transformer Embed Size (d_{model})	32
Transformer Num Heads	4
Transformer Batch Size	512

Census [31]. The US Census Data (1990) Data Set, which consists of a 1% sample made publicly available. We use all available columns, which range from 2 to 18 distinct values each.

Dryad-URLs [65]. For text domain experiments, we use this small dataset of 2.4M URLs, each truncated to 100 characters. This dataset was chosen to emulate a plausible `STRING` column in a relational database.

Evaluation Metric

We issue a large set of randomly generated range queries, and measure how accurately each estimator answers them. We report the multiplicative error, or *Q-error*, defined as the factor by which an estimate differs from the actual density (obtained by actually executing each query on the dataset):

$$\text{Error} := \max(\text{estimate}, \text{actual}) / \min(\text{estimate}, \text{actual})$$

Hence, a perfect estimate for a query has an error of 1.0. Moreover, we report the median, 99%-tile, and maximum Q-error across all queries. We note that the median error is typically within a fraction of 1.0 for all estimators. The reason is that most randomly generated queries

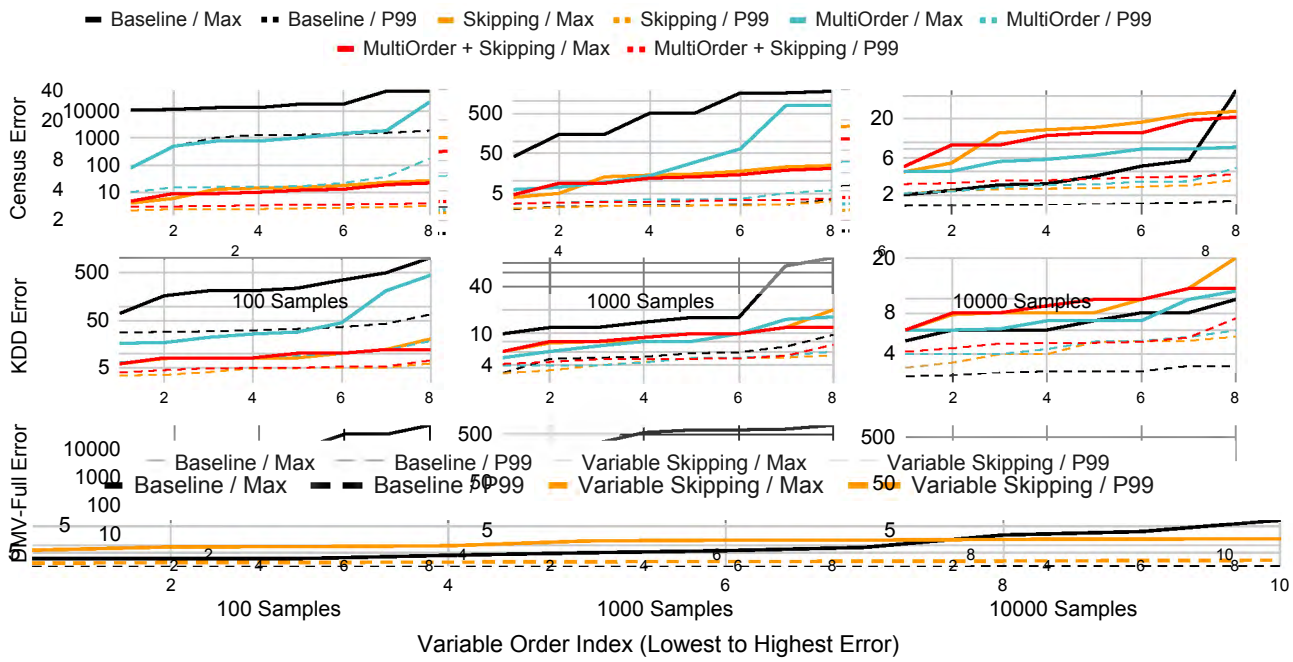


Figure 6.5: Variable skipping and skipping combined with multi-order training vs. baselines across different datasets, variable orderings, and sampling budgets. Error is plotted on the y-axis in *log scale* (lower is better). Each column reflects a 10× increase in sampling budget as we move to the right. Results for 8 different variable orders within each plot are sorted by increasing error. Variable skipping provides 10−100× max error reduction at low budgets, and still improves accuracy at high sampling budgets for large datasets such as DMV-Full. This data is also shown in tabular form in Table 6.5, which additionally reports median errors.

are "easy" (i.e., hit few cross-dimension predicate correlations), and only a few are "hard". Because of this, even a naive estimator can achieve good performance in many cases. Hence, our focus is on high quantile errors for evaluation.

Experiment Setup

For queries against tabular data, we used the experiment framework from [167], randomly drawing between 5 and 12 conjunctive variable constraints per query[1]. It is important to not have too many or too few constraints, which would skew the distribution of true density estimates towards 0.0 (too many constraints lead to little density) or 1.0 (too few constraints lead to high density) respectively.

[1]An example query for DMV-Full may be: `record_type == 1 AND city == 17 AND zip > 10000 AND model_year < 1990 AND max_weight > 5000`.

Table 6.3: The model negative log-likelihoods at convergence in bits/datapoint (evaluated using non-masked data). We also report standard deviation across multiple random order seeds.

Dataset	Baseline	Random Masking	MultiOrder(5)	MultiOrder(10)	MultiOrder(15)
Census	52.04 ± .009	52.34 ± .02	52.69 ± .02	52.79 ± .03	52.81 ± .03
DMV-Full	43.12 ± .04	43.65 ± .06	44.15 ± .05	44.53 ± .06	44.65 ± .04
KDD	107.5 ± .3	116.58 ± .2	123 ± .9	127.6 ± .4	128.4 ± .5

For text queries, we issued pattern glob queries of the form `value CONTAINS <str>`, where `<str>` is a character sequence between 3 and 5 characters in length drawn randomly from the full text corpus. This also provides a challenging spread of density from very common (e.g., `CONTAINS ".com"`), to quite rare (e.g., `CONTAINS "XVQ/i"`).

We compare between the following approaches, all of which use progressive sampling (Section 6.3) as the approximate inference procedure:

- **Baseline**: An autoregressive model queried using vanilla progressive sampling [167].
- **Skipping**: An autoregressive model trained with random input masking and queried with the variable skipping optimization enabled (Section 6.4).
- **MultiOrder**: An autoregressive model trained under multiple variable orders to enable querying an ensemble of 10 orders at inference time [152, 42].
- **MultiOrder + Skipping**: Combining the multi-order and variable skipping techniques.

The full list of training hyperparameters can be found in Table 6.2. Unless otherwise specified, we use a ResMADE [35] with 3 residual blocks, two 256-unit hidden layers per block, and an 32-unit wide embedding for each input dimension. We choose hyperparameters known to optimize for progressive sampling performance [167], but did not otherwise tune them for our experiments. In our ablations (Section 6.5) we found that the most sensitive hyperparameter to performance is the embedding size, which is closely related to model size.

The autoregressive variable ordering can significantly affect estimator variance. We thus evaluate each technique on 8 randomly chosen variable orderings and train a (fixed-order) model for each ordering[2]. For multi-order models, we train 8 distinct sets of 10 randomly chosen orders (we saw diminishing returns past 10 orders), unless specified otherwise. To ensure fairness, when not using skipping, we use a model trained without masking.

For multi-order ResMADE, to condition on the current ordering statistics each masked linear layer is allocated an additional weight matrix that shares the existing mask and has

[2]For ResMADE, this means we sample 8 sets of {input ordering, intermediate connectivity masks}.

an all-one vector as its input[3]. Due to the additional weights, we size down the hidden units appropriately to ensure that the multi-order models have about the same parameter count as other models.

Variable Skipping Performance

We evaluate the impact of the variable skipping optimization on the DMV-Full, Census, and KDD datasets. For each dataset, we generated 1000 random range queries.

In Figure 6.5 we show the results of variable skipping (orange and red lines) compared against baselines (black and turquoise lines). This data is also shown in tabular form in Table 6.5. We evaluate with sampling budgets of 100, 1000, and 10000 samples (left, center, and right columns respectively). Note that a sample refers to *all* the forward passes required to sample relevant variables (e.g., for Census a single sample takes 67 forward passes without skipping). We limit to 10k samples for cost reasons (at 10k samples, each query takes multiple seconds to evaluate even with a GPU). There are several key takeaways:

High-quantile error differentiates estimators: Across all estimators, the median error is very close to 1.0 (not shown since it is indistinguishable in log scale). However, systems applications necessarily seek to minimize the worst-case error, which does vary significantly across samplers.

Skipping significantly improves sampling efficiency: Across all datasets, variable skipping provides between 10× to 100× max error reduction at low sampling budgets (i.e., 100 samples), compared to the baseline. It also provides up to 10× improvement over the multi-order ensemble alone.

Concretely, at 100 samples the 99th-quantile error for Census is reduced from ~1000 to 2.5, KDD from ~30 to 3, and DMV-Full from ~1000 to 100. Moving up to 1000 samples, we continue to see a significant improvement at the max error, with Census improved from ~500 to 10, KDD from ~15 to 8, and DMV-Full from ~500 to 100.

Compared to multi-order, variable skipping provides 10× better max error reduction for Census and KDD at 100 samples. Interestingly, while multi-order and variable skipping provide comparable improvements for DMV-Full at 100 samples, *combining multi-order and skipping* provides a further 10× improvement in both max and 99th quantile error. This suggests that variable skipping and multi-order training are orthogonal mechanisms, and can be combined for larger datasets such as DMV-Full to reduce both error *and* inference costs.

Variable skipping can help even at high sampling budgets: On the DMV-Full dataset, variable skipping provides more than an order of magnitude reduction in max error (from ~150 to 5), even at 10000 samples. We hypothesize this is due to the large domain sizes of DMV-Full (up to 32K distinct values), which in the worst case would require a much larger

[3]This treatment has appeared in MADE [42].

number of samples to achieve low estimation error. As evidence for this, a large number of samples are required to achieve good errors even with skipping enabled. This is in contrast to the smaller CENSUS and KDD datasets where skipping achieves close to single-digit errors even with as low as 100 samples. This suggests that for even larger datasets (common in industrial settings), skipping may have an even greater impact.

We have also provided a summary of the results for Figure 6.5 in Figure 6.1. For the concrete target of 10× error at the 99th quantile, skipping significantly reduces the compute requirements over both the baseline and multi-order. This is due to both accuracy improvements that combine with those provided by multi-order ensembles, and also the reduced compute requirements of skipping.

Model Likelihoods vs. Training Scheme

Table 6.4: Variable skipping vs. vanilla progressive sampling on the text domain. Naive sampling refers to generating samples (from the learned AR model) without constraints and then filtering the generated samples to estimate the probability of matches. We include naive sampling as a baseline for this experiment since it is competitive with progressive sampling in the text domain. We measure the estimation error over 100 random pattern queries against the DRYAD-URLS dataset, and show the bootstrap standard deviation.

Num Samples	Metric	Naive Sampling	Progressive Sampling	Variable Skipping
1000	Max Error	6412 ± 1280	4628 ± 1785	**115.2** ± 25.6
1000	P99 Error	4054 ± 1386	1741 ± 1824	**89.5** ± 30.6
1000	Median Error	**1.23** ± .06	1.66 ± .15	1.39 ± .08

Training with partially masked inputs makes the learning task more difficult: the number of examples increases by a factor of 2^N, and effectively the same model is learning multiple autoregressive distributions. Table 6.3 shows that in terms of negative log-likelihoods achieved, models trained with masking do have a slightly higher NLL than baseline, as expected. However, the NLLs achieved are lower than those of multi-order models, and the gap only widens with an increased number of orders.

Even though NLLs of masked models are higher than those of baseline, Figure 6.5 shows the benefit: estimation error is significantly improved when variable skipping is enabled. This highlights *the non-perfect alignment of optimizing for point likelihoods vs. downstream range query performance*, opening up interesting future directions.

Model Size and Masking Ablations

In Figure 6.6 we study the relationship between model size and estimation accuracy. For this experiment, we vary the model embedding size among $\{2, 8, 32\}$ dimensions, and the

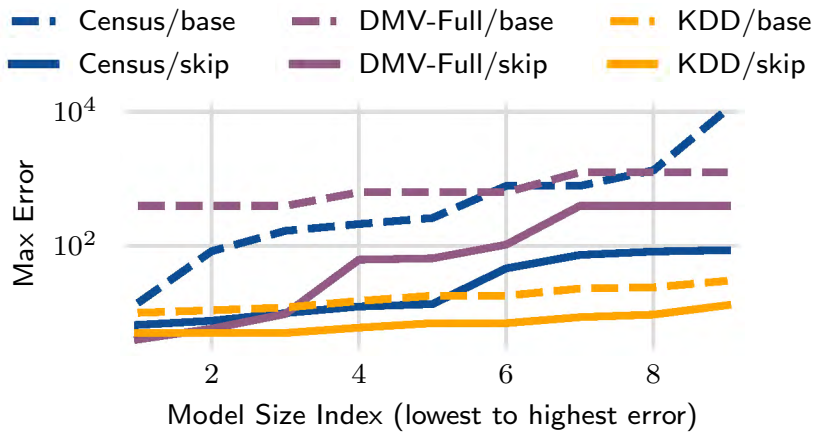


Figure 6.6: Model size vs. max estimation error on 1000 queries with 1000 samples each. The results for each dataset are sorted in increasing error. Errors are plotted on the y-axis in *log scale* (lower is better). Errors increase as the model embedding sizes are reduced from 32 to 2, and hidden layer sizes from 256 to 16. Variable skipping (solid lines) retains an advantage across this two orders of magnitude change in model capacity.

hidden layer size among $\{16, 64, 256\}$ units. We see that variable skipping retains a robust advantage across nearly two orders of magnitude variation in model size.

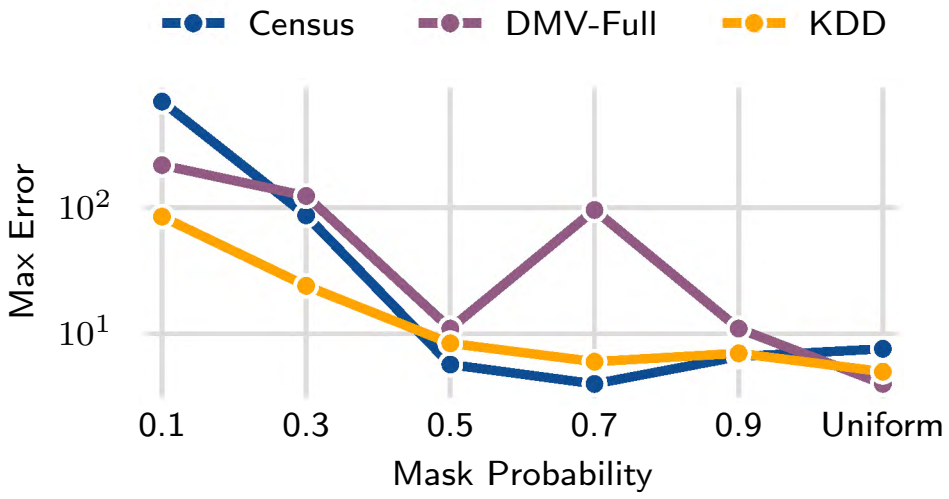


Figure 6.7: Varying the masking scheme. Here we measure the max estimator error with skipping enabled over 1000 queries with 1000 samples each, on the natural variable order. Errors are plotted on the y-axis in *log scale* (lower is better).

In Figure 6.7 we compare a few schemes for selecting the mask distribution, from fixed masking probabilities of $\{0.1, 0.3, 0.5, 0.7, 0.9\}$, vs. the random uniform scheme used in the main experiment. We see that drawing the mask probability uniformly at random obtains the lowest errors for KDD and DMV-Full, and close to optimal for Census as well, showing it to be a robust choice.

Application to Pattern Matching in Text Domain

Finally, we show that variable skipping can be applied to the text domain for estimating the probability of pattern matches. Pattern matching (or more generally, regex matching), can be thought of as unrolling a dynamic predicate as variables are sampled (Section 6.4). Here we evaluate a simple character-level Transformer model on the Dryad-URLs dataset. We note that this is not a realistic application since scanning a dataset of this size is much faster than sampling from a model, however it demonstrates the applicability of variable skipping across domains. Table 6.4 shows that prefix skipping enables much lower variance estimates than naive sampling and vanilla progressive sampling.

6.6 Conclusion

To summarize, we identify the range density estimation task and important applications. We propose *variable skipping*, which greatly reduces sampling variance and inference latency. We validate the effectiveness of these techniques across a variety of datasets and model configurations.

Table 6.5: The full table of quantiles across all random orders evaluated in Figure 6.5. We show the mean and standard deviation of the quantiles across the random order seeds.

Samples	Metric	Dataset	Progressive	Multi-Order	Skipping	Multi + Skip
100	P50	Census	1.19 ± .01	1.53 ± .32	**1.09 ± .01**	1.21 ± .03
		KDD	1.24 ± .02	1.41 ± .16	**1.14 ± .02**	1.25 ± .05
		DMV-Full	1.22 ± .04	1.70 ± .22	**1.08 ± .01**	1.22 ± .04
	P99	Census	1150 ± 550	38.6 ± 52	**2.55 ± .29**	3.36 ± .32
		KDD	36.4 ± 11	9.82 ± 3.4	**4.58 ± .83**	5.07 ± .85
		DMV-Full	1130 ± 1300	80 ± 81	93 ± 69	**6.5 ± 1.7**
	Max	Census	24500 ± 18000	3490 ± 6700	15.1 ± 7.3	**12.3 ± 5.3**
		KDD	345 ± 280	99.9 ± 140	9.97 ± 4.1	**9.25 ± 1.9**
		DMV-Full	21200 ± 22000	1640 ± 2600	1560 ± 1700	**22.7 ± 13**
1000	P50	Census	**1.06 ± .01**	1.44 ± .31	1.09 ± .01	1.20 ± .02
		KDD	**1.08 ± .01**	1.34 ± .17	1.13 ± .02	1.25 ± .05
		DMV-Full	1.08 ± .01	1.53 ± .24	**1.05 ± .01**	1.18 ± .04
	P99	Census	2.58 ± .39	3.84 ± 1.1	**2.50 ± .28**	3.2 ± .28
		KDD	5.78 ± 1.7	4.69 ± .67	**4.41 ± .74**	5.05 ± .89
		DMV-Full	105 ± 44	11.3 ± 4.8	**4.24 ± 2.1**	4.92 ± .81
	Max	Census	798 ± 700	212 ± 330	14.7 ± 7.0	**12.8 ± 5.1**
		KDD	30.7 ± 30	9.42 ± 3.8	9.95 ± 4.1	**9.37 ± 1.9**
		DMV-Full	508 ± 200	39.4 ± 43	114 ± 100	**14 ± 6.7**
10000	P50	Census	**1.03 ± .01**	6.31 ± 1.6	1.09 ± .01	1.20 ± .24
		KDD	**1.05 ± .01**	1.33 ± .18	1.13 ± .02	1.25 ± .05
		DMV-Full	1.05 ± .01	1.48 ± .24	**1.05 ± .01**	1.18 ± .04
	P99	Census	**1.49 ± .08**	2.84 ± .70	2.48 ± .29	3.20 ± .29
		KDD	**2.99 ± .19**	4.43 ± .93	4.36 ± .75	5.08 ± .85
		DMV-Full	5.95 ± 3.4	7.23 ± 2.4	**2.89 ± .34**	4.96 ± .85
	Max	Census	8.86 ± 14	**6.31 ± 1.6**	14.7 ± 7.0	12.6 ± 5.0
		KDD	**7.0 ± 1.5**	7.57 ± 1.9	9.97 ± 4.1	9.37 ± 1.9
		DMV-Full	181 ± 78	13.5 ± 6.2	20.2 ± 40	**10.5 ± 2.6**

Chapter 7

Conclusion

This thesis started with the design of RLlib, a scalable and universal open source library for distributed reinforcement learning. Today, this project is among the most widely adopted RL libraries in both industry and academia, and sets the standard for applied use cases. Initially, our goal was to define a flexible architecture that enables support for a broad range of RL algorithms with high performance. Based on user feedback, our library evolved from merely providing system primitives to solving the other aspects of RL systems design of concern: support multi-agent computations, a functional API for numerical definitions, batch training, customizations, and support for complex environments. Addressing these user concerns led to conceptual improvements such as a dataflow-centric API, which provides abstractions that not only enable advanced multi-agent use cases, but considerably slim down and simplify large portions of RLlib's distributed execution implementation. The second half of this thesis looked back at how we leverage RL and ML for improving systems, specifically diving into examples improving the construction of packet classifiers and data cardinality estimators.

Open Source Impact

Over the past four years, RLlib has grown from a handful of parallel algorithms (A3C, DQN, PPO, and Evolution Strategies) to a toolkit for large-scale RL that features more than twenty algorithms maintained in conjunction with the open source community. We're deeply thankful to the early users who took a bet on RLlib when it was initially released and have offered countless contributions and suggestions to improve it over the years.

It's quite difficult to accurately measure the usage of an open source project, but here are a few of the interesting facts about RLlib:

- More than a dozen of RLlib's algorithms have been contributed by the community, including the MA-DDPG and QMIX multi-agent algorithms, ARS, DDPG, TD3, MARWIL, SAC, Linear UCB, Linear Thompson Sampling, and AlphaZero.
- RLlib is offered as part of the public AI platform in two of three major cloud providers (AWS Sagemaker, Azure ML), as well by several smaller industry ML platforms.

- The largest deployments of RLlib use thousands of CPUs for distributed training.
- RLlib was used by QuantumBlack to help the Emirates Team New Zealand to win America's Cup—competitive sailing's most coveted prize. To accomplish this, they used RL for training virtual sailors to evaluate boat designs also generated via RL.
- Use cases we are aware of include traffic flow, safety RL, compiler optimization, data structure layout, multi-agent research, intelligent autoscaling, video games such as Minecraft, StarCraft, and car racing, police patrol scheduling, large-scale econometrics, industrial optimization, optimizing cellular antenna coverage, trade order execution, COVID-19 drug discovery, supply chain optimization, and building AI team-mates.

More generally, RLlib has enabled users the ability to rapidly experiment with different RL approaches at scale, including helping us tackle the RL for systems problems in this thesis. With NeuroCuts, we were able to show significant improvements over twenty years of research into packet classification data structures. These advances were enabled by a novel framing of the packet classification tree problem as an RL environment, which could be executed out of the box with RLlib thanks to its flexibility. Using RLlib, research collaborators experimenting with RL for databases were able to leverage a variety of different RL approaches, including meta-learning and environment modeling techniques such as variable skipping. In a similar vein, the authors of NeuralMMO [145], a massively multi-agent simulation environment, were able to almost immediately set a new record for agent longevity after porting their environment to RLlib's multi-agent API. Because of these advantages, we believe that flexibility combined with high performance will be a key quality for RL systems and more generally ML systems in the future.

Future Work

This section lists avenues of future work we believe would be of research interest, several of which have immediate practical applications.

Automatic optimization of system hyperparameters for RL. While hyperparameter optimization is a well-known problem for applied RL, there is the potential for different avenues of attack as it relates to *system hyperparameters*. For example, RLlib has many tunables that affect execution performance, such as the level of asynchrony, worker parallelism, and pipeline parallelism. Some hyperparameters, such as the level of vectorization, episode truncation mode, and batching, also affect the learning dynamics directly. Currently users must either rely on their experience and intuition to set these system parameters, or tune these system parameters similar to other hyperparameters. A holistic approach to optimizing the overall RL execution plan could provide significant benefits.

Support for differentiable computations across multiple agents. In multi-agent scenarios, it is occasionally useful to allow for differentiable communication between agents. This can allow for efficient modeling of shared computations or communication channels between agents in the environment. Supporting this feature conflicts with existing RLlib abstractions for defining policies; it would be interesting to explore what it would take to efficiently support such differentiable communications, as it is a common user ask. Abstractions addressing the performance challenges here have been explored by the NLP community [95, 111].

Programming language techniques to simplify abstractions. Supporting a broad range of large-scale RL workloads is not without trade-offs. A key challenge of systems design is how to design abstractions that are simple yet sufficiently describe the computation so that the underlying system can optimize its execution. It would be interesting to explore how RLlib's abstractions could be simplified, perhaps with programming techniques such as static analysis or dynamic tracing, as exemplified by frameworks such as JAX.

Compiled execution plans for performance and reliability. RLlib is implemented in Ray, which implements distributed execution fully dynamically. However, this dynamic execution can sometimes incur unnecessary overheads for memory allocation and data movement. For example, in RL systems like SampleFactory [120], shared memory buffers are used to minimize copy costs between threads of execution in CPUs and GPUs. It would be interesting to explore how to combine the flexibility of dynamic execution with the safety and performance benefits of a static execution plan. At a more implementation level, performance-critical components of RLlib such as the sampler could be re-written in a compiled language for performance.

Generalizing NeuroCuts to different families of tree-based data structures. Packet classification trees can be thought of as a particular type of multi-dimensional search tree that receives only point queries and is built on static data. It would be interesting to study whether and how the NeuroCuts approach could generalize to other families of multi-dimensional search trees such as R-trees and kd-trees that have different optimization objectives and requirements for supporting updates. Indeed, supporting updates is one avenue of potential improvement to NeuroCuts, and a potential barrier for practical deployment. The result of this research could be a family of RL algorithms for tackling different types of search tree generation problems.

Integrating learned components into a production database. While there have been many promising results showing how components of databases (i.e., the cardinality estimator [168]) can be replaced with learned approaches, it remains to be seen if RL and ML based components can practically replace their hand-engineered counterparts in a production database. There are several questions here, including the choice between a modular vs.

holistic approach to integrating learned components, and whether learned components can handle advanced features such as user-defined functions and regular expression queries.

Bibliography

[1] Martın Abadi et al. "TensorFlow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Igor Adamski et al. *Distributed Deep Reinforcement Learning: Learn how to play Atari games in 21 minutes*. 2018. arXiv: 1801.02852 [cs.AI].

[3] Amazon. *Amazon EC2 Pricing*. https://aws.amazon.com/ec2/pricing. 2017.

[4] Amazon. *Scientific Computing with EC2 Spot Instances*. https://aws.amazon.com/ec2/spot/spot-and-science/. 2011.

[5] Marcin Andrychowicz et al. "Hindsight experience replay". In: *arXiv preprint arXiv:1707.01495* (2017).

[6] Kai Arulkumaran, Antoine Cully, and Julian Togelius. "Alphastar: An evolutionary computation perspective". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2019, pp. 314–315.

[7] Morton M. Astrahan et al. "System R: Relational approach to database management". In: *ACM Transactions on Database Systems (TODS)* 1.2 (1976), pp. 97–137.

[8] Mohammad Babaeizadeh et al. *Reinforcement Learning through Asynchronous Advantage Actor-Critic on a GPU*. 2017. arXiv: 1611.06256 [cs.LG].

[9] Florin Baboescu, Sumeet Singh, and George Varghese. "Packet classification for core routers: Is there an alternative to CAMs?" In: *IEEE INFOCOM*. 2003.

[10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The datacenter as a computer: An introduction to the design of warehouse-scale machines". In: *Synthesis lectures on computer architecture* 8.3 (2013), pp. 1–154.

[11] Norbert Beckmann et al. "The R*-tree: An efficient and robust access method for points and rectangles". In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 1990, pp. 322–331.

[12] Christopher Berner et al. "Dota 2 with Large Scale Deep Reinforcement Learning". In: *CoRR* abs/1912.06680 (2019). arXiv: 1912.06680. URL: http://arxiv.org/abs/1912.06680.

[13] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[14] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[15] P. Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38 (2015), pp. 28–38.

[16] Itai Caspi et al. *Reinforcement Learning Coach*. Dec. 2017. DOI: 10.5281/zenodo.1134899. URL: https://doi.org/10.5281/zenodo.1134899.

[17] Pablo Samuel Castro et al. "Dopamine: A research framework for deep reinforcement learning". In: *arXiv preprint arXiv:1812.06110* (2018).

[18] Fay Chang et al. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[19] Li Chen et al. "AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization". In: *ACM SIGCOMM*. 2018.

[20] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *NIPS Workshop on Machine Learning Systems (LearningSys'16)*. 2016.

[21] Rewon Child et al. "Generating long sequences with sparse transformers". In: *arXiv preprint arXiv:1904.10509* (2019).

[22] Sandeep Chinchali et al. "Cellular network traffic scheduling with deep reinforcement learning". In: *AAAI*. 2018.

[23] Ignasi Clavera et al. "Model-Based Reinforcement Learning via Meta-Policy Optimization". In: *CoRR* abs/1809.05214 (2018). arXiv: 1809.05214. URL: http://arxiv.org/abs/1809.05214.

[24] Ekin Dogus Cubuk et al. "AutoAugment: Learning Augmentation Policies from Data". In: *CoRR* (2018).

[25] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[26] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[27] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems 25*. 2012, pp. 1223–1231.

[28] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[29] DMLC. *NNVM Compiler: Open Compiler for AI Frameworks*. `http://www.tvmlang.org/2017/10/06/nnvm-compiler-announcement.html`. 2017.

[30] Mo Dong et al. "PCC Vivace: Online-Learning Congestion Control". In: *USENIX NSDI*. 2018.

[31] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml`.

[32] Yan Duan et al. "Benchmarking Deep Reinforcement Learning for Continuous Control". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1329–1338. URL: `http://dl.acm.org/citation.cfm?id=3045390.3045531`.

[33] Yan Duan et al. "Benchmarking deep reinforcement learning for continuous control". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. 2016.

[34] Emilien Dupont and Suhas Suresha. "Probabilistic semantic inpainting with pixel constrained CNNs". In: *arXiv preprint arXiv:1810.03728* (2018).

[35] Conor Durkan and Charlie Nash. "Autoregressive Energy Machines". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, Sept. 2019, pp. 1735–1744.

[36] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *arXiv preprint arXiv:1802.01561* (2018).

[37] Lasse Espeholt et al. "SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference". In: *International Conference on Learning Representations*. 2020.

[38] Kousha Etessami and Mihalis Yannakakis. "Recursive Markov decision processes and recursive stochastic games". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2005, pp. 891–903.

[39] WA Falcon. "PyTorch Lightning". In: *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning* 3 (2019).

[40] Chelsea Finn, Pieter Abbeel, and Sergey Levine. *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*. 2017. arXiv: `1703.03400 [cs.LG]`.

[41] Jason Gauci et al. "Horizon: Facebook's Open Source Applied Reinforcement Learning Platform". In: *arXiv preprint arXiv:1811.00260* (2018).

[42] Mathieu Germain et al. "MADE: Masked autoencoder for distribution estimation". In: *International Conference on Machine Learning*. 2015, pp. 881–889.

[43] Marjan Ghazvininejad et al. "Mask-predict: Parallel decoding of conditional masked language models". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 6114–6123.

[44] Gluon. *The Gluon API Specification*. `https://github.com/gluon-api/gluon-api`. 2017.

[45] Google. *Preemptible Virtual Machines*. `https://cloud.google.com/preemptible-vms`. 2015.

[46] G. Graefe. "Volcano— An Extensible and Parallel Query Evaluation System". In: *IEEE Trans. on Knowl. and Data Eng.* 6.1 (Feb. 1994), pp. 120–135. ISSN: 1041-4347. DOI: 10.1109/69.273032. URL: https://doi.org/10.1109/69.273032.

[47] Goetz Graefe and Diane L Davison. "Encapsulation of parallelism and architecture-independence in extensible database query execution". In: *IEEE Transactions on Software Engineering* 19.8 (1993), pp. 749–764.

[48] Alex Graves, Abdel-Rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *ICASSP*. 2013.

[49] William Gropp et al. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6 (1996), pp. 789–828.

[50] Arthur Guez et al. "Learning to search with MCTSnets". In: *arXiv preprint arXiv:1802.04697* (2018).

[51] Pankaj Gupta and Nick McKeown. "Algorithms for packet classification". In: (2001).

[52] Pankaj Gupta and Nick McKeown. "Packet classification on multiple fields". In: *SIGCOMM CCR* (1999).

[53] Pankaj Gupta and Nick McKeown. "Packet classification using hierarchical intelligent cuttings". In: *Hot Interconnects*. Aug. 1999.

[54] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).

[55] Danijar Hafner, James Davidson, and Vincent Vanhoucke. "TensorFlow Agents: Efficient Batched Reinforcement Learning in TensorFlow". In: *arXiv preprint arXiv:1709.02878* (2017).

[56] Danijar Hafner et al. *Dream to Control: Learning Behaviors by Latent Imagination*. 2020. arXiv: `1912.01603 [cs.LG]`.

[57] Ameer Haj-Ali et al. "A view on deep reinforcement learning in system optimization". In: *arXiv preprint arXiv:1908.01275* (2019).

[58] Peter Henderson et al. "Deep Reinforcement Learning that Matters". In: *CoRR* (2017).

[59] Christopher Hesse et al. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[60] Carl Hewitt, Peter Bishop, and Richard Steiger. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence". In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[61] Matt Hoffman et al. *Acme: A Research Framework for Distributed Reinforcement Learning*. 2020. arXiv: `2006.00979 [cs.LG]`.

[62] John E Hopcroft and Jeffrey D Ullman. "Introduction to Automata Theory, Languages and Computation. Adison-Wesley". In: *Reading, Mass* (1979).

[63] Dan Horgan et al. "Distributed Prioritized Experience Replay". In: *International Conference on Learning Representations* (2018). URL: `https://openreview.net/forum?id=H1Dy---0Z`.

[64] Qijing Huang et al. "Autophase: Compiler phase-ordering for hls with deep reinforcement learning". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 308–308.

[65] Michael Isard et al. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: `10.1145/1272996.1273005`. URL: `http://doi.acm.org/10.1145/1272996.1273005`.

[66] Max Jaderberg et al. "Population Based Training of Neural Networks". In: *arXiv preprint arXiv:1711.09846* (2017).

[67] Nathan Jay et al. "Internet Congestion Control via Deep Reinforcement Learning". In: *arXiv preprint arXiv:1810.03259* (2018).

[68] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[69] Kirill Kogan et al. "SAX-PAC (scalable and expressive packet classification)". In: *SIGCOMM CCR*. 2014.

[70] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[71] AN Kolmogorov and NA Dmitriev. "Stochastic branching processes". In: *Doklady Akademi Nauk SSSR*. Vol. 56. 1. 1947, pp. 7–10.

[72] Vijay R. Konda and John N. Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems*. 2000.

[73] Peter Kontschieder et al. "Deep Neural Decision Forests". In: *The IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015.

[74] Ilya Kostrikov. *PyTorch implementation of Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO) and Scalable trust-region method for deep reinforcement learning*. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr`. 2017.

[75] Tim Kraska et al. "The case for learned index structures". In: *SIGMOD*. 2018.

[76] Sanjay Krishnan et al. "Learning to optimize join queries with deep reinforcement learning". In: *arXiv preprint arXiv:1808.03196* (2018).

[77] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012.

[78] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. *Tensorforce: a TensorFlow library for applied reinforcement learning*. Web page. 2017. URL: `https://github.com/tensorforce/tensorforce`.

[79] Sameer Kulkarni and John Cavazos. "Mitigating the compiler optimization phase-ordering problem using machine learning". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2012, pp. 147–162.

[80] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. "Algorithms for advanced packet classification with ternary CAMs". In: *SIGCOMM CCR*. 2005.

[81] John Langford and Tong Zhang. "The epoch-greedy algorithm for multi-armed bandits with side information". In: *Advances in neural information processing systems*. 2008, pp. 817–824.

[82] Ian Lenz, Honglak Lee, and Ashutosh Saxena. "Deep learning for detecting robotic grasps". In: *The International Journal of Robotics Research* (2015).

[83] Sergey Levine et al. "End-to-end training of deep visuomotor policies". In: *The Journal of Machine Learning Research* (2016).

[84] Sergey Levine et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *The International Journal of Robotics Research* (2018).

[85] Guoliang Li et al. "Qtune: A query-aware database tuning system with deep reinforcement learning". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2118–2130.

[86] Jiwei Li et al. "Deep reinforcement learning for dialogue generation". In: *arXiv preprint arXiv:1606.01541* (2016).

[87] Lisha Li et al. "Hyperband: A novel bandit-based approach to hyperparameter optimization". In: *arXiv preprint arXiv:1603.06560* (2016).

[88] Mu Li et al. "Scaling distributed machine learning with the parameter server". In: *11th USENIX Symposium on Operating Systems Design and Implementation*. 2014, pp. 583–598.

[89] Wenjun Li et al. "CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification". In: *IEEE INFOCOM*. 2018.

[90] Eric Liang et al. "Neural packet classification". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 256–269.

[91] Eric Liang et al. *RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem*. 2021. arXiv: `2011.12719 [cs.LG]`.

[92] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholm Sweden: PMLR, Oct. 2018, pp. 3053–3062. URL: `http://proceedings.mlr.press/v80/liang18b.html`.

[93] Eric Liang et al. "Variable Skipping for Autoregressive Range Density Estimation". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 6040–6049.

[94] Alex X Liu, Chad R Meiners, and Yun Zhou. "All-match based complete redundancy removal for packet classifiers in TCAMs". In: *IEEE INFOCOM*. 2008.

[95] Moshe Looks et al. "Deep learning with dynamic computation graphs". In: *arXiv preprint arXiv:1702.02181* (2017).

[96] Keng Wah Loon, Laura Graesser, and Milan Cvitkovic. *SLM Lab: A Comprehensive Benchmark and Modular Software Framework for Reproducible Deep Reinforcement Learning*. 2019. arXiv: `1912.12482 [cs.LG]`.

[97] Michael Luo et al. *IMPACT: Importance Weighted Asynchronous Architectures with Clipped Target Networks*. 2020. arXiv: `1912.00167 [cs.LG]`.

[98] Yadi Ma and Suman Banerjee. "A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification". In: *ACM SIGCOMM*. 2012.

[99] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. "Neural adaptive video streaming with pensieve". In: *ACM SIGCOMM*. 2017.

[100] Hongzi Mao et al. "Learning Scheduling Algorithms for Data Processing Clusters". In: *arXiv preprint arXiv:1810.01963* (2018).

[101] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *ACM SIGCOMM HotNets Workshop*. 2016.

[102] Ryan Marcus et al. "Neo: A Learned Query Optimizer". In: *arXiv preprint arXiv:1904.03711* (2019).

[103] Microsoft. *ONNX: Open Neural Network Exchange Format*. `https://onnx.ai`. 2017.

[104] Azalia Mirhoseini et al. "Chip placement with deep reinforcement learning". In: *arXiv preprint arXiv:2004.10746* (2020).

[105] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*. 2016.

[106] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). URL: `http://arxiv.org/abs/1312.5602`.

[107] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 978-1-931971-47-8. URL: `http://dl.acm.org/citation.cfm?id=3291168.3291210`.

[108] Seyedakbar Mostafavi, Fatemeh Ahmadi, and Mehdi Agha Sarram. "Reinforcement-learning-based foresighted task scheduling in cloud computing". In: *arXiv preprint arXiv:1810.04718* (2018).

[109] Derek G Murray et al. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455.

[110] Arun Nair et al. "Massively parallel methods for deep reinforcement learning". In: *arXiv preprint arXiv:1507.04296* (2015).

[111] Graham Neubig, Yoav Goldberg, and Chris Dyer. "On-the-fly operation batching in dynamic computation graphs". In: *arXiv preprint arXiv:1705.07860* (2017).

[112] Mohammad Norouzi et al. "Efficient non-greedy optimization of decision trees". In: *Advances in Neural Information Processing Systems*. 2015, pp. 1729–1737.

[113] Aäron van den Oord et al. "WaveNet: A Generative Model for Raw Audio". In: *Arxiv*. 2016.

[114] OpenAI. *Evolution Strategies Starter Agent*. `https://github.com/openai/evolution-strategies-starter`. 2017.

[115] OpenAI. *OpenAI Dota 2 1v1 bot*. `https://openai.com/the-international/`. 2017.

[116] OpenAI. *Universe Starter Agent*. `https://github.com/openai/universe-starter-agent`. 2016.

[117] Jennifer Ortiz et al. "Learning state representations for query optimization with deep reinforcement learning". In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 2018, pp. 1–4.

[118] Heidi Pan, Benjamin Hindman, and Krste Asanović. "Composing parallel software efficiently with Lithe". In: *ACM Sigplan Notices* 45.6 (2010), pp. 376–387.

[119] Zhiping Peng et al. "Random task scheduling scheme based on reinforcement learning in cloud computing". In: *Cluster computing* 18.4 (2015), pp. 1595–1607.

[120] Aleksei Petrenko et al. "Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7652–7662.

[121] Stanley R Pliska. "Optimization of multitype branching processes". In: *Management Science* 23.2 (1976), pp. 117–124.

[122] Mariya Popova, Olexandr Isayev, and Alexander Tropsha. "Deep reinforcement learning for de novo drug design". In: *Science advances* 4.7 (2018), eaap7885.

[123] *PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration*. `http://pytorch.org/`.

[124] Yaxuan Qi et al. "Packet classification algorithms: From theory to practice". In: *IEEE INFOCOM*. 2009.

[125] Yun R. Qu et al. "Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 2015.

[126] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI Blog* 1.8 (2019), p. 9.

[127] Roshan Rao et al. "Evaluating protein transfer learning with TAPE". In: *Advances in Neural Information Processing Systems*. 2019, pp. 9689–9701.

[128] Hesham El-Rewini, Hesham H Ali, and Ted Lewis. "Task scheduling in multiprocessing systems". In: *Computer* 28.12 (1995), pp. 27–37.

[129] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: *arXiv preprint arXiv:1703.03864* (2017).

[130] Tim Salimans et al. "PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[131] Michael Schaarschmidt, Alexander Kuhnle, and Kai Fricke. *TensorForce: A TensorFlow library for applied reinforcement learning*. Web page. 2017. URL: `https://github.com/reinforceio/tensorforce`.

[132] Michael Schaarschmidt et al. "RLgraph: Modular Computation Graphs for Deep Reinforcement Learning". In: *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML)*. Apr. 2019.

[133] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *International Conference on Learning Representations (ICLR)* (2016).

[134] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[135] P Griffiths Selinger et al. "Access path selection in a relational database management system". In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.

[136] Or Sharir et al. "Deep Autoregressive Models for the Efficient Variational Simulation of Many-Body Quantum Systems". In: *Phys. Rev. Lett.* 124 (2 Jan. 2020), p. 020503. DOI: 10.1103/PhysRevLett.124.020503.

[137] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* (2018).

[138] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* (2016).

[139] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* (2017).

[140] Sumeet Singh et al. "Packet classification using multidimensional cutting". In: *ACM SIGCOMM*. Aug. 2003.

[141] Ed Spitznagel, David Taylor, and Jonathan Turner. "Packet classification using extended TCAMs". In: *IEEE ICNP*. 2003.

[142] Venkatachary Srinivasan, Subhash Suri, and George Varghese. "Packet classification using tuple space search". In: *SIGCOMM CCR*. 1999.

[143] State of New York. *Vehicle, snowmobile, and boat registrations*. catalog.data.gov/dataset/vehicle-snowmobile-and-boat-registrations. [Online; accessed March 1st, 2019]. 2019.

[144] Adam Stooke and Pieter Abbeel. *Accelerated Methods for Deep Reinforcement Learning*. 2019. arXiv: 1803.02811 [cs.LG].

[145] Joseph Suarez et al. "Neural MMO: A massively multiagent game environment for training and evaluating intelligent agents". In: *arXiv preprint arXiv:1903.00784* (2019).

[146] Weibin Sun and Robert Ricci. "Fast and flexible: parallel packet processing with GPUs and click". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 2013.

[147] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014.

[148] David E. Taylor and Jonathan S Turner. "ClassBench: A packet classification benchmark". In: *IEEE INFOCOM*. Mar. 2005.

[149] David E. Taylor and Jonathan S Turner. "Scalable packet classification using distributed crossproducing of field labels". In: *IEEE INFOCOM*. 2005.

[150] Yuandong Tian et al. "ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games". In: *CoRR* abs/1707.01067 (2017). arXiv: 1707.01067. URL: http://arxiv.org/abs/1707.01067.

[151] Ankit Toshniwal et al. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 147–156.

[152] Benigno Uria, Iain Murray, and Hugo Larochelle. "A deep and tractable density estimator". In: *International Conference on Machine Learning*. 2014, pp. 467–475.

[153] Asaf Valadarsky et al. "Learning to route with Deep RL". In: *NIPS Deep Reinforcement Learning Symposium*. 2017.

[154] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. "EffiCuts: Optimizing Packet Classification for Memory and Throughput". In: *ACM SIGCOMM*. Aug. 2010.

[155] Aaron Van den Oord et al. "Conditional image generation with pixelcnn decoders". In: *Advances in neural information processing systems*. 2016, pp. 4790–4798.

[156] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *AAAI*. 2016.

[157] Matteo Varvello et al. "Multilayer packet classification with graphics processing units". In: *IEEE/ACM Transactions on Networking* (2016).

[158] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[159] Stephanie Wang et al. "Ownership: A Distributed Futures System for Fine-Grained Tasks". In: *nsdi* (2020).

[160] Zhen Wang et al. "Knowledge Graph Embedding by Translating on Hyperplanes." In: *AAAI*. 2014.

[161] Dirk Weissenborn, Oscar Täckström, and Jakob Uszkoreit. "Scaling autoregressive video models". In: *arXiv preprint arXiv:1906.02634* (2019).

[162] Erik Wijmans et al. *DD-PPO: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames*. 2020. arXiv: `1911.00357 [cs.CV]`.

[163] Zheng Xiong, Wenpeng Zhang, and Wenwu Zhu. "Learning decision trees with reinforcement learning". In: *NIPS Workshop on Meta-Learning*. 2017.

[164] Shuicheng Yan et al. "Graph embedding and extensions: A general framework for dimensionality reduction". In: *IEEE transactions on pattern analysis and machine intelligence* (2007).

[165] Jianchao Yang et al. "Non-negative graph embedding". In: *CVPR*. 2008.

[166] Zhilin Yang et al. "XLNet: Generalized autoregressive pretraining for language understanding". In: *Advances in neural information processing systems*. 2019, pp. 5754–5764.

[167] Zongheng Yang et al. "Deep Unsupervised Cardinality Estimation". In: vol. 13. 3. VLDB, 2019, pp. 279–292.

[168] Zongheng Yang et al. "NeuroCard: one cardinality estimator for all tables". In: *arXiv preprint arXiv:2006.08109* (2020).

[169] Zongheng Yang et al. "Qd-tree: Learning data layouts for big data analytics". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 193–208.

[170] Hyunho Yeo et al. "Neural adaptive content-aware internet video delivery". In: *USENIX OSDI*. 2018.

[171] Jiaxuan You et al. "Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation". In: *arXiv preprint arXiv:1806.02473* (2018).

[172] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[173] Matei Zaharia et al. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling". In: *Proceedings of the 5th European conference on Computer systems*. 2010, pp. 265–278.

[174] Matei Zaharia et al. "Discretized streams: fault-tolerant streaming computation at scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. Farminton, Pennsylvania: ACM Press, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://dl.acm.org/citation.cfm?doid=2517349.2522737.

[175] Matei Zaharia et al. "Spark: cluster computing with working sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Vol. 10. 2010, p. 10.

[176] Yasir Zaki et al. "Adaptive congestion control for unpredictable cellular networks". In: *SIGCOMM CCR*. 2015.

[177] Ying Zheng et al. "Demystifying Deep Learning in Networking". In: *ACM SIGCOMM APNet Workshop*. 2018.

[178] Jie Zhou et al. "Graph Neural Networks: A Review of Methods and Applications". In: *arXiv preprint arXiv:1812.08434* (2018).