Technical Report

UCAM-CL-TR-946
ISSN 1476-2986

Number 946

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

End-to-end deep reinforcement learning in computer systems

Michael Schaarschmidt

April 2020

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2019 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Sidney Sussex College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

End-to-end deep reinforcement learning in computer systems

Michael Schaarschmidt

Summary

The growing complexity of data processing systems has long led systems designers to imagine systems (e.g. databases, schedulers) which can self-configure and adapt based on environmental cues. In this context, reinforcement learning (RL) methods have since their inception appealed to systems developers. They promise to acquire complex decision policies from raw feedback signals. Despite their conceptual popularity, RL methods are scarcely found in real-world data processing systems. Recently, RL has seen explosive growth in interest due to high profile successes when utilising large neural networks (deep reinforcement learning). Newly emerging machine learning frameworks and powerful hardware accelerators have given rise to a plethora of new potential applications.

In this dissertation, I first argue that in order to design and execute deep RL algorithms efficiently, novel software abstractions are required which can accommodate the distinct computational patterns of communication-intensive and fast-evolving algorithms. I propose an architecture which decouples logical algorithm construction from local and distributed execution semantics. I further present RLgraph, my proof-of-concept implementation of this architecture. In RLgraph, algorithm developers can explore novel designs by constructing a high-level data flow graph through combination of logical components. This dataflow graph is independent of specific backend frameworks or notions of execution, and is only later mapped to execution semantics via a staged build process. RLgraph enables high-performing algorithm implementations while maintaining flexibility for rapid prototyping.

Second, I investigate reasons for the scarcity of RL applications in systems themselves. I argue that progress in applied RL is hindered by a lack of tools for task model design which bridge the gap between systems and algorithms, and also by missing shared standards for evaluation of model capabilities. I introduce Wield, a first-of-its-kind tool for incremental model design in applied RL. Wield provides a small set of primitives which decouple systems interfaces and deployment-specific configuration from representation. Core to Wield is a novel instructive experiment protocol called progressive randomisation which helps practitioners to incrementally evaluate different dimensions of non-determinism. I demonstrate how Wield and progressive randomisation can be used to reproduce and assess prior work, and to guide implementation of novel RL applications.

Acknowledgements

First and foremost, I want to thank my supervisor, Eiko Yoneki, for supporting me in taking on this endeavour in the first place. When starting out, the intersection of systems and reinforcement learning was unknown but exciting territory. I owe it to her constant encouragement and unending patience in shaping out ideas to eventually find my place between these two fields.

The projects at the centre of this dissertation would also not have been possible without collaborators and supporters. I am particularly indebted to Kai Fricke for our long and productive collaborations in open source reinforcement learning, RLgraph, and Wield. I also thank Sven Mika for extensive design discussions and implementation help in RLgraph. In this context, I also thank all the open source contributors and users who through their feedback and contributions shaped my understanding of real-world reinforcement learning.

The beginning of my PhD was also enriched by Valentin Dalibard who helped me understand subtleties of optimisation techniques. I also thank Felix Gessert for our collaboration on web caching which sparked my initial interest for tuning in systems.

I am also grateful to both Google and the Computer Laboratory for supporting my research through a Google PhD fellowship and a Robert Sansom studentship respectively. I am profoundly grateful to my parents and sister for their invaluable support and advice which carried me throughout a long decade of studies. Finally, I thank Adnan Halilovic, Arne Tonsen, Kai Bruns, Felix Clausberg, and Maximilian Schuch for their friendship and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decades, the proliferation of ever more powerful data processing systems and large scale distributed computing has given rise to a myriad of tuning challenges. What are the best configuration parameters for a given workload and a particular deployment environment? How should behaviour vary under changing workloads and unforeseen inputs? Exponentially growing data volumes and the rising complexity of systems software necessitate automated approaches to these problems.

In this context, statistical machine learning methods based on learned feature representations have come into focus. Developers of systems software such as databases or schedulers are investigating them to tune all aspects of configurations and deployment. Even further, they propose to eventually replace key systems components (e.g. query planners) with learned models [PAA+17]. However, unlike application domains such as natural language processing [BCB15, VSP+17] or computer vision [KSH12, SZ15] which heavily rely on supervised training data, systems optimisation problems cannot easily be given supervised solutions. For example, resource management problems are frequently NP-complete, and practical systems rely on heuristics to identify approximate solutions.

Hence, systems designers have turned their attention to reinforcement learning (RL) algorithms as a class of methods which learn decision policies from performance signals without supervision. In combination with neural networks as function approximators (deep reinforcement learning (DRL)), they have been used to solve tasks across a growing number of domains [NCD+06, MKS+15, SHM+16, SSS+17, FL17]. In systems, recent research into RL applications spans a diverse range of domains such as scheduling [MSV+19], networking [MNA17], database management [MP18, MNM+19], and device placement [MPL+17, MGP+18]. In controlled settings, these experiments often demonstrate substantial improvement against off-the-shelf algorithms or hand-designed heuristics.

Yet, despite a wealth of successful proof-of-concepts in research, RL solutions are rarely (if at all) employed in real-world systems environments. Emerging deep RL approaches promise to yield significant performance improvements by learning to adapt to fine-grained workload properties. However, these methods come at the cost of long training times, additional hyperparameters requiring calibration, and brittle algorithms due to aggressive use of stochastic approximations. My dissertation investigates these challenges from a systems design perspective. What software primitives should be used to implement these methods? How is domain knowledge translated to RL problem representations?

Successful real-world applications exist in other domains, e.g. content recommendation at Facebook [GCL+18]. However, they are characterised by extremely large deployments

where outsized economical gains can be realised from small improvements. Problem environments of this scale only exist in few organisations, and systems deployments relying on RL must justify both development overhead and non-trivial computation cost.

Implementing and executing these machine learning methods themselves also poses new systems challenges. Reinforcement learning workloads significantly differ from supervised workloads in resource usage and communication patterns. Supervised training mechanisms have undergone significant standardisation, e.g. widespread use of synchronous distributed stochastic gradient descent. This in turn has allowed machine learning frameworks to provide training abstractions for neural networks which transparently manage distributed training from single-node to date-centre scale [AIM17]. These training mechanisms are largely independent from the neural network architecture and loss function used.

Reinforcement learning workloads in contrast continue to evolve rapidly. They are driven by empirical evaluation of new training modes and recombination of training heuristics. Where training data is usually available in advance for supervised learning (e.g. on a distributed file-system), reinforcement learning agents must interact with one or more problem environments to collect new experience. RL workloads have not found similar standardisation, as new algorithm designs continue to leverage new learning semantics and communication patterns. Research implementations suffer from performance issues and lack re-usability due to tight coupling of execution semantics and training algorithm.

This dissertation focuses on systems abstractions for deep reinforcement learning from two perspectives. First, I consider software support for RL workloads by introducing a modular programming model for deep RL. I present RLgraph, a framework which decouples execution semantics from algorithm logic to combine flexible prototyping with scalable execution. Users compose algorithms by creating dataflow between high-level logical components, and RLgraph combines them as named subgraphs into end-to-end differentiable computation graphs through a staged build process. Through this separation of concerns, RLgraph achieves high-performing, incrementally testable and reusable implementations which address the implementation difficulties around brittle RL algorithms.

Second, I investigate applications of reinforcement learning to the optimisation of data processing systems themselves. The trade-offs discussed above call into question if current deep RL approaches can close the gap to real-world deployments. Answering this question is complicated by fragmented research approaches and missing shared benchmarks for fine-granular optimisation problems in systems. New results are difficult to compare, assess, and reproduce. To begin addressing these challenges, I propose Wield, a software framework which standardises training workflows and problem modelling when applying RL to real-world systems. Developers can use Wield to iteratively identify, train, and evaluate reinforcement learning representations. Wield further introduces a novel evaluation protocol focused on distinguishing RL model capabilities under different randomisation assumptions. Where RLgraph is concerned with algorithm design and execution, Wield covers the necessary problem modelling and evaluation to make effective use of RLgraph. I use RLgraph and Wield to investigate the following thesis:

> A reinforcement learning engine that decouples logical dataflow from execution semantics can flexibly execute reinforcement learning workloads across different application semantics, and resolve the tension between robust and scalable implementations and fast prototyping.
>
> Further, new software tools and evaluation mechanisms are needed to analyse

and ultimately overcome the gap between experimental successes and real-world deployments of RL in systems optimisation.

1.1 Contributions

In this dissertation, I make two principal contributions to introduce an end-to-end software stack for reinforcement learning in computer systems:

1. My first contribution is **RLgraph**, an architecture for decoupling definition of reinforcement learning algorithms from their execution semantics. RLgraph separates logical dataflow design from execution through a staged build process. The main novelty compared to prior deep RL implementations lies in the incremental dataflow instantiation where any arbitrary combination of sub-graphs can be executed and tested individually. This allows users to compose new algorithms through reusable components, and to then execute these algorithms on different deep learning frameworks and distributed execution engines. The resulting implementations achieve robust learning performance and high sampling throughput as a result of incremental testing and strict separation of concerns.

2. My second contribution is **Wield**, a software tool for incremental model design in reinforcement learning. While algorithmic frameworks like RLgraph facilitate rapid implementation of novel algorithms, systems primitives for mapping domain problems to reinforcement learning models themselves are lacking. Moreover, reasoning about model capabilities is difficult due to highly sensitive algorithms and a lack of common evaluation protocols. To facilitate systematic model construction, Wield introduces a novel classification scheme called progressive randomisation which helps delineate model capabilities under different randomisation assumptions.

The designs, architectures and algorithms presented in this dissertation are the result of my own work. However, collaborators have helped me implement several of the components described in this dissertation.

The RLgraph project (§3) is a collaboration with Sven Mika who contributed to design and implementation of many modules but especially to the static build mechanism (§3.3.2) and the component class (§3.3.1). For RLgraph's experimental evaluation, he also in particular contributed the IMPALA implementation used to demonstrate distributed TensorFlow (TF) capabilities (§4.5). Kai Fricke implemented the cloud orchestration for Ray and distributed TensorFlow, and helped carrying out learning and throughput experiments for these engines (§4). In the Wield project, Kai Fricke further helped with implementation and experiments on the device placement case study (§6.3).

1.2 Dissertation outline

The remainder of my dissertation is structured as follows:

- **Chapter 2** gives an overview of reinforcement learning as the central technique my work is based on. Here, I focus on algorithmic aspects independent of systems issues. I further give a brief overview of iterative optimisation techniques commonly used in systems tuning.

- **Chapter 3** begins with an analysis of RL workloads. I first characterise RL applications in difference to supervised learning and identify common implementation patterns. I then survey existing frameworks and RL abstractions to motivate key design problems in RL. This motivates RLgraph, my programming model and execution engine for deep reinforcement learning which is introduced in this chapter.

- **Chapter 4** evaluates how algorithms designed with RLgraph can be built towards different execution contexts. I further show that RLgraph generates high-performing execution plans by comparing training performance and throughput to both a recent framework and a tuned-standalone implementation. Finally, I illustrate how RLgraph enables composition of new algorithmic variants by motivating and implementing a distributed algorithm variant.

- **Chapter 5** introduces Wield, a framework for task modelling towards systems applications of reinforcement learning. I first analyse the practical gap between decades of experimental work in systems applications of RL and the lack of real world deployments. Second, I introduce progressive randomisation as an experimental protocol. Finally, I discuss mechanisms such as learning from imperfect demonstrations to incorporate domain knowledge and improve training results in applied RL.

- **Chapter 6** evaluates the application of Wield via case studies on compound indexing in databases and automated device placement for machine learning workloads. I further use progressive randomisation to reproduce and assess prior published work.

- **Chapter 7** summarises conclusions of this dissertation. I further describe a number of future directions for both RLgraph and Wield.

1.3 Related publications

As part of the work described in this dissertation, I have authored the following peer-reviewed papers and preprints:

[**SFY19**] Michael Schaarschmidt, Kai Fricke, Eiko Yoneki. "Wield: Systematic Reinforcement Learning With Progressive Randomization". *arXiv preprint arXiv:1909.06844, 2019.*

[**WSY19**] Jeremy Welborn, Michael Schaarschmidt, Eiko Yoneki. "Learning Index Selection with Structured Action Spaces". *arXiv preprint arXiv:11909.07440, 2019.*

[**SMFY19**] Michael Schaarschmidt, Sven Mika, Kai Fricke, Eiko Yoneki. "RLgraph: Modular Computation Graphs for Deep Reinforcement Learning". In: *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML), Palo Alto, CA, April 2019.*

[**SGDY16**] Michael Schaarschmidt, Felix Gessert, Valentin Dalibard, Eiko Yoneki. "Learning Runtime Parameters in Computer Systems with Delayed Experience Injection". In: *NIPS Deep Reinforcement Learning Workshop, Barcelona, Spain, December 2016.*

I have also co-authored the following publications which have influenced my understanding of optimisation in computer systems, but did not directly contribute to the work in this dissertation:

[**GSW+17**] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, Norbert Ritter. "Quaestor: Query Web Caching for Database-as-a-Service Providers". In: *Proceedings of the 43rd International Conference on Very Large Databases (PVLDB 2017), Munich, Germany, August 2017.*

[**DSY17**] Valentin Dalibard, Michael Schaarschmidt, Eiko Yoneki. " BOAT: Building Auto-Tuners with Structured Bayesian Optimization". In: *World Wide Web Conference, Systems and Infrastructure Track (WWW), Perth, Australia, April, 2017.*

[**DSY16**] Valentin Dalibard, Michael Schaarschmidt, Eiko Yoneki. " Tuning the Scheduling of Distributed Stochastic Gradient Descent with Bayesian Optimization". In: *NIPS Workshop on Bayesian Optimization, Barcelona, Spain, December 2016.*

Chapter 2

Background

The contributions of my dissertation are in systems applications and computational aspects of reinforcement learning (RL). This chapter hence surveys different aspects of reinforcement learning and optimisation.

I begin by reviewing key RL concepts with focus on recent techniques combining RL with deep neural networks (§2.1). In Section 2.2, I discuss the heuristics and algorithms which characterise the wave of novel applications in deep reinforcement learning. Finally, I connect a number of related black-box optimisation techniques to RL (§2.3).

2.1 Reinforcement learning

2.1.1 The reinforcement learning problem

The reinforcement learning problem (Fig. 2.1) is described by an agent interacting with an environment represented as a sequence of states $s \in S$ [SB17]. In each state s_t, the agent selects an action a_t from its action space A, observes a reward r_t, and advances to a new state s_{t+1}. The agent seeks to maximise cumulative expected rewards $\mathbb{E}[\sum_t \gamma^t r_t]$ where future rewards are discounted by $\gamma \in [0, 1]$. To this end, the agent seeks to improve its behavioural policy $\pi(a|s)$ from which it selects its actions to learn the optimal policy π^*.

Both the environment and π may be stochastic, i.e. taking the same action on the same state may result in different transitions. The policy may be represented as a probability distribution from which actions are sampled. Formally, Markov Decision Processes (MDP) are used to describe the reinforcement learning problem where state transitions are modeled via transition probabilities $P_a(s_t, s_{t+1})$, and transitions only depend on s_t and a_t (satisfy the Markov property). An MDP is defined through the set of states S, the action space A, the transition probabilities $P_a(s_t, s_{t+1})$, and the reward function $R_a(s_t, s_{t+1})$ describing rewards observed after performing actions.

In practice, the state is often only partially observable. In the framework of Partially Observable Markov Decision Processes (POMDP), environment dynamics are assumed to be governed by an MDP, but only partial or noisy state information is available. Applied problems especially in computer systems typically require significant "state engineering". This is because unlike in domains like games or robotics, where a natural state representation may arise from the current frame of a video game or the physical sensors of the robots, data processing systems expose large amounts of system information at different time scales and formats. In this case, RL practitioners must manually select,

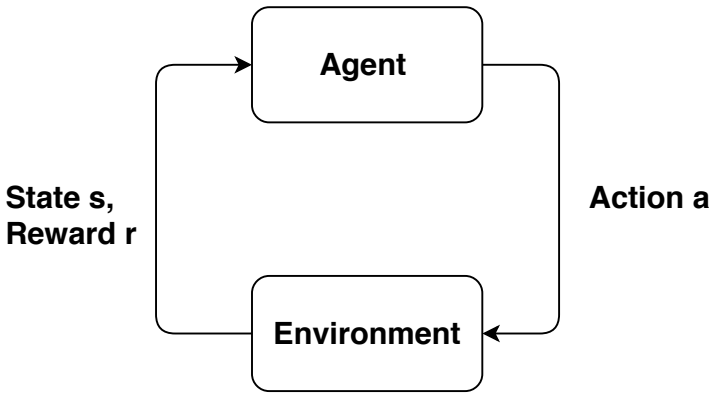


Figure 2.1: The standard reinforcement learning problem.

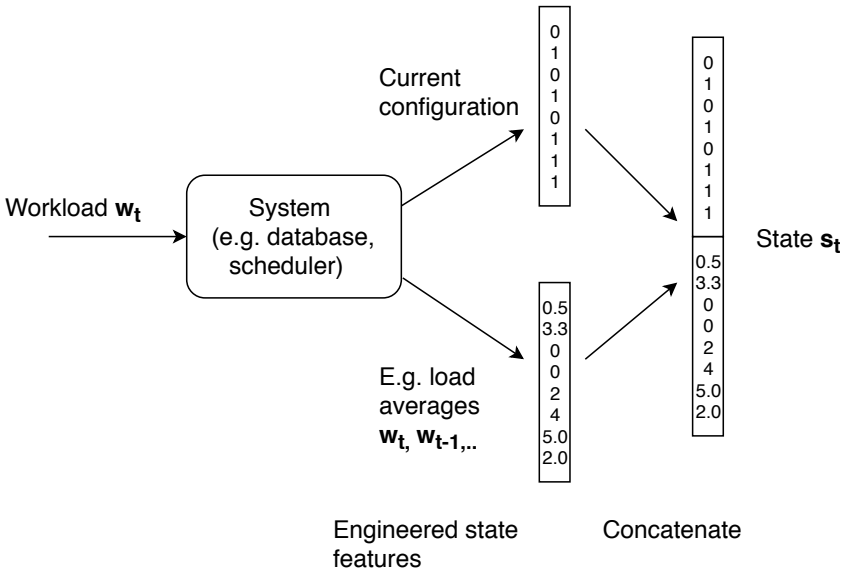


Figure 2.2: States can require extensive feature engineering to combine workload metrics and system configuration, e.g. by simple concatenation.

pre-process and aggregate information which they believe to form the system state (Figure 2.2). I discuss state and action representations for systems applications in Chapter 5.

2.1.2 Episodes and experience

In this dissertation, I consider the episodic reinforcement learning problem where agent interactions with the environment are partitioned into finite length episodes. Instead of operating under an infinite time horizon, the agent discounts rewards towards the end of an episode which is marked by a terminal state. After reaching a terminal state, all further actions let the agent remain in this state, and no further rewards are given. Many applied problems naturally lend themselves to episode semantics. For example, games may represent a single level which can be won or lost as an episode. In computer systems, processing a single workload instance (e.g. a set of queries) and measuring its performance is often used to represent individual episodes.

Episode semantics are of great relevance to computational aspects of reinforcement learning. They can form logical units of execution for sample trajectories. Between

episodes, the environment needs to be reset to restore it to an initial state s_0 (which may be sampled from a distribution of initial states). For some problems, resetting the environment (e.g. by re-deploying or reconfiguring a system) can take substantial time.

The term "sample trajectories" above refers to the fact that the methods I discuss in the following are *Monte Carlo methods*. They operate on incomplete knowledge of the environment, and computations are based on sample averages. The agent collects *experience* in the form of sample trajectories (sequences of states, actions, rewards and terminal information), and averages returns for each state and action pair.

A core idea towards practically solving RL problems is the concept of using *value functions* to evaluate sample trajectories. A value function $V_\pi(s)$ of a policy π is used by an agent to estimate expected returns of being in a state s and taking actions according to π. Formally, the state-value function (for an episode of length T) $V_\pi(s)$ is given as:

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} | s_t = s \right] \tag{2.1}$$

Value functions hence enable the evaluation of a given trajectory under a policy. In deep reinforcement learning, neural networks are used to represent value functions (§2.2).

An extensive survey of RL methods is given by Sutton and Barto [SB17]. Here, I briefly summarise temporal-difference and policy-gradient RL algorithms.

2.1.3 Temporal difference learning

Temporal difference (TD) algorithms update their value function estimates based on a combination of observed reward and their own prior estimates. They are hence bootstrap methods. After visiting a state and observing a reward, initially random (or initialised with some inductive bias) value function estimates are updated. For example, in Q-learning [WD92], a popular TD-method, a state-action value function called Q-function $Q_\pi(s, a) = \mathbb{E}[R_t | s_t = s, a]$ is learned to estimate for each state and action pair the expected discounted rewards from taking a in s and following π after. The aim of Q-learning is to identify the optimal Q-function

$$Q^*(s, a) = \max_\pi Q_\pi(s, a) \tag{2.2}$$

The optimal policy for the agent follows by simply selecting the action with the optimal value in each step (i.e. highest estimated returns). The value function $V_\pi(s)$ can then be expressed via Q and the expectation across all actions:

$$V_\pi(s) = \mathbb{E}_{a \sim \pi}[Q_\pi(s, a)] \tag{2.3}$$

It then follows that $V^*(s) = \max_a Q^*(s, a)$. When expressing Q^* and V^* through a Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma V^*(s') | s, a, \pi \right], \tag{2.4}$$

one can then substitute V^* using the relation above:

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]. \tag{2.5}$$

Here, s' is an often-used shorthand for s_{t+1} (with $s = s_t$). This equation provides a simple update algorithm as any state and action pair (s, a) can update its q-value estimate by

combining the reward observed after taking a and arriving in s' with the return estimate from s' itself. TD methods hence combine dynamic programming with Monte Carlo methods (as discussed by Sutton [SB17]). During training, the agent greedily selects the action with the highest estimated q-value or a random action with some probability ϵ to explore (ϵ-*exploration*).

Note that Q-learning does not require episode semantics due to its incremental nature. It can simply update after each step without concerning itself with prior or future events, as they are entirely encapsulated within Q. The discount factor γ modulates how many steps ahead the agent incorporates into its action selection. While Q-learning converges to the optimal policy in the case of finite MDPs [WD92], visiting and computing Q-values for all pairs (s, a) is impractical for real-world problems. I describe how Q-learning algorithms are implemented in practice using neural networks in Section 2.2.

2.1.4 Policy gradients

In Q-learning, the policy is implicitly derived through greedy action selection. In contrast, policy gradient methods directly represent the policy $\pi(a|s, \theta)$ via a differentiable parameter vector $\theta \in \mathbb{R}^D$. While policy gradient methods often still use value functions to evaluate trajectories and learn θ, the value function is not needed to select actions, as they are directly output by π. In the following, I will sometimes omit θ in subscripts when referring to π, i.e. I will write Q_π for Q_{π_θ}.

In practice, π is represented as a parametrised probability distribution, e.g. a Gaussian distribution may be used for continuous actions, or a categorical distribution for discrete actions. Importantly, policy gradient methods can be used to learn both stochastic and deterministic policies (the deterministic policy gradient theorem [SLH$^+$14] is beyond the scope of this dissertation). Consider a normal distribution $\mathcal{N}(\mu, \Sigma|s, \theta)$ parametrised by θ which estimates mean μ and covariance Σ for each s. A deterministic π would select action μ as the maximum likelihood estimate, while a stochastic policy would sample from $\mathcal{N}(\mu, \Sigma)$.

In Q-learning, the Q-function must be evaluated for all actions for action selection. Sample requirements can hence become impractical for large action spaces, and using a continuous action space in Q-learning requires discretisation. Policy gradient methods can naturally handle continuous action spaces via parametrised probability distributions. They also directly include exploration by sampling and learning to adjust co-variance estimates, and hence do not require bolt-on techniques such as epsilon-exploration.

A key result used as the basis for modern policy gradient methods is the policy gradient theorem. The policy gradient theorem provides an analytical expression to estimate the gradient of a performance measure $J(\theta)$ with respect to policy parameters θ:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in S} \mu(s) \sum_{a \in A} Q_\pi(s, a) \pi_\theta(a|s) \tag{2.6}$$

$$\propto \sum_{s \in S} \mu(s) \sum_{a \in A} Q_\pi(s, a) \nabla_\theta \pi_\theta(a|s) \tag{2.7}$$

Here, $\mu(s)$ refers to the on-policy state distribution. By expanding this term and substi-

tuting the derivative of the logarithm ($ln(x)' = 1/x$), the following expression arises:

$$\sum_{s \in S} \mu(s) \sum_{a \in A} Q_\pi(s,a) \nabla_\theta \pi_\theta(a|s) = \sum_{s \in S} \mu(s) \sum_{a \in A} \pi_\theta(a|s) Q_\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \tag{2.8}$$

$$= \mathbb{E}_{s \sim \mu, a \sim \pi}[Q_\pi(s,a) \nabla_\theta ln\ \pi(a|s)] \tag{2.9}$$

For the detailed proof I refer the reader to the original paper [SMS+99] or the latest edition of Sutton and Barto's seminal textbook [SB17]. Crucially, the derivative of $J(\theta)$ does not depend on the derivative of $\mu(s)$ This means policy gradient methods can improve π without knowledge of system dynamics. They simply repeatedly estimate $\nabla_\theta J(\theta)$ by collecting new state transitions and using the expression above.

Gradient estimation in policy gradient methods can be improved by separately training a baseline which estimates $V_\pi(s)$. This allows to use the so called advantage $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$ for $Q_\pi(s,a)$. Intuitively, the policy gradient update modifies the distribution to increase the likelihood of good actions, where goodness is judged by Q_π. An empirical estimate of Q_π is the (discounted) reward observed when taking a in s. However, the absolute reward value is less relevant than how much better the reward for a specific a in s was than the average estimate of the state value ($V_\pi(s)$). This difference is called the *advantage*. Various other heuristics exist to reduce the variance of gradient estimates without introducing bias. Schulman et al. discuss common variants in their work on generalised advantage estimation [SML+15].

In the formulation above, policy gradients are called an *on-policy* method because updates are based on actions carried out by π. In contrast, an *off-policy* method can learn from experience collected by any policy. Q-learning algorithms are off-policy because the bootstrap estimate of $Q_\pi(s', a')$ utilises greedy action selection ($\max_{a'}$), as opposed to the action that was actually taken in the next state (which may differ due to exploration). Recent research has explored various hybrid methods where e.g. a baseline is learned off-policy to improve performance [GLT+17, GLG+17]. Due to fewer hyper-parameters, simpler variants often prevail in practice.

2.2 Deep reinforcement learning

2.2.1 Function approximation with deep neural networks

Reinforcement learning algorithms rely on functions of the state which map states to Q-values or probability distributions of actions. Real-world problems usually either have extremely large discrete state spaces (e.g. all legal positions on a chess board), or continuous state spaces. This means that during learning or at test time, almost every state is unseen. The concept of *function approximation* in deep RL refers to using a neural network as a non-linear function approximator.

While various other state representations exist (e.g. linear models, Gaussian processes), this dissertation focuses on reinforcement learning with neural networks.

Over the past decade, deep neural network architectures (e.g. convolutional [KSH12] or recurrent [HS97]) have seen practical success in a wide range of applied domains such as computer vision [SZ14, KSH17], natural language processing [BCB15, WSC+16, BGLL17], and robotics [FL17, LPK+18]. The concept of training neural networks with backpropagation can be traced back over 30 years (a detailed history of neural network

training is available by Schmidhuber [Sch15]). However, recent progress in data processing capabilities (hardware accelerators) and software support from open source machine learning software have led to an unparalleled rise in research into all aspects of neural network training. Popular deep learning frameworks such as TensorFlow [ABC+16], PyTorch [PGC+17], MXnet [CLL+15] or CNTK [SA16] enable developers to prototype, train, and deploy neural network representation from high-level building blocks.

These developments have provided a fertile environment for the surge of interest in combining neural networks with reinforcement learning. In deep RL, the network receives states as inputs and outputs Q-values, state-values, or parameters for a probability distribution. The term "outputs" must be clarified insofar the final network layers must be specifically engineered to support various action representations.

In deep RL, the neural network architecture typically consists of at least two components. A variable number of hidden layers learns a representation of input features. A flexible number of action layers is bolted on top of the hidden layers to represent different action outputs (e.g. parameters of a normal distribution) or value estimates. This is in contrast to classification tasks where the network simply outputs final class probabilities.

2.2.2 Deep reinforcement learning

A turning point for deep reinforcement learning as a new subfield came via Mnih et al.'s work on learning to play Atari games from video input via deep convolutional architectures [MKS+13, MKS+15]. The neural network's parameters θ are updated iteratively (indexed by i) using the following loss function (π subscripts omitted for readability):

$$J_i(\theta_i) = \mathbb{E}_{s,a\sim\pi}[(y_i - Q(s,a;\theta_i))^2] \tag{2.10}$$

with $y = r + \gamma\ max_{a'}Q(s',a';\theta_{i-1})$ (the Q-target). The gradient of the loss function

$$\nabla_{\theta_i} J_i(\theta_i) = \mathbb{E}_{s,a\sim\pi}[(r + \gamma\ max_{a'}Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i}Q(s,a;\theta_i)] \tag{2.11}$$

is then optimised via mini-batch stochastic gradient descent. Deep Q-Network (DQN) extends Q-learning by a number of heuristics used to stabilise training. First, a *replay memory* [Lin93] is used to store observations. Instead of updating Q-values after each sample and thus training from correlated sequences, a mini-batch of uncorrelated experiences is sampled periodically to estimate the gradient. This allows for training samples to be reused and also avoids feedback loops encountered in naive Q-learning where training is dominated by sampling from one state region. Second, a so called *target network* is used to improve training stability. As noted above, Q-learning uses a bootstrap estimate of the next state (the Q-target y) to update the Q-value for a given state-action pair. This can cause divergence and oscillation if the agent gets stuck in feedback loops, e.g. due to noise in the environment.

DQN avoids this via a fixed Q-target where a second neural network (the target network) with fixed parameters θ_{i-t} (for some lag t) is used to estimate updates, while the training network with current parameters is employed for action selection. The parameters of the fixed value function are periodically updated by copying the parameters of the training network.

Algorithm 1 illustrates the DQN algorithm with ϵ-exploration and a state preprocessing function ϕ which down-samples and transforms input images. DQN's conceptual simplicity

Algorithm 1 DQN algorithm as introduced by Mnih et al.

Initialise replay memory D to capacity N
Initialise action-value Q-function with random weights θ
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select random action a_t
    Otherwise select $a_t = max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t and observe reward r_t and image x_t
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition in replay memory
    Sample uniformly at random from D: (s_j, a_j, r_j, s_{j+1})
    Set

$$y_i = \begin{cases} r_j, & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \; max_{a'} Q(s_{t+1}, a'; \theta), & \text{for non-terminal } \phi_{j+1} \end{cases}$$

    Perform an gradient descent step on $(\gamma_j - Q(s_j, a_j, \theta))^2$
  end for
end for

and off-policy nature makes it a popular choice for new practitioners, as the replay memory D can be initialised from prior experience trajectories.

While many of the elements in DQN had been researched in prior work, Mnih et al.'s contribution lies in combining them with various other heuristics to learn different Atari tasks with the same architecture. For example, successfully solving Atari tasks with the original DQN architecture requires to combine multiple game-frames into a single state since a single frame does not capture motion. Further, both gradients and reward values were clipped to specific ranges to stabilise updates.

Even when using numerous custom heuristics and tuned hyper-parameters, DQN (and other early DRL approaches) are highly sensitive to weight initialisation and small changes in parameters (§3.1). Further, tens of millions of frames were required to solve Atari tasks. More difficult environments (e.g. Montezuma's revenge) could not be learned at all due to sparse rewards. Nonetheless, DQN-style algorithms remain among the most popular applied algorithms. Since Mnih. et al.'s initial work on DQN, a plethora of variants and improvements have been proposed. I summarise key developments below.

2.2.3 Common heuristics

A growing body of research is investigating all aspects of value estimation, reward discounting, exploration, gradient estimation, and experience prioritisation. A number of core techniques has emerged over the past years which are now canonically used in RL implementations. Replay memories are usually implemented with prioritised experience replay [SQAS15] where experience tuples are prioritised according to their loss values to focus training on high-loss states. Double DQN describes a technique where action selection and evaluation are decoupled in the update step by using the training network to select the action for s', and the target network to estimate its Q-values [HGS16] which significantly improves training performance. In dueling DQN, a separate value-stream is

introduced to the network to decompose Q-values into state-value and advantage estimates $Q(s,a) = A(s,a) + V(s)$ [WSH+16]. A further common technique is to not use 1-step rewards to bootstrap Q-estimates but a forward view of multiple steps (n-step Q-learning) [SB17]. Less commonly used is distributional DQN where instead of computing expected returns, the discrete distribution of rewards is approximated [BDM17]. The so-called *Rainbow* algorithm combines these and other improvements into a single architecture [HMvH+18]. Many of these techniques have in turn created inspired bodies of work, but enumerating them is beyond the scope of this dissertation.

Similar to rapid developments in Q-learning, a multitude of new policy optimisation algorithms has emerged. Trust region policy optimisation (TRPO) addresses several problems of naive policy gradients [SLA+15]. Specifically, policy performance can collapse from a single update as the gradient descent step to update parameters θ can unfavourably shift the resulting actions for many states. This is particularly problematic when using few environment samples, or when environment noise leads to imprecise gradient estimates. TRPO limits updates in the policy space (as opposed to parameter space θ) by solving a constrained optimisation where the constraint is expressed via the Kulback-Leibler divergence between prior and updated policy. This optimisation is solved via natural gradient descent combined with line-search.

Despite being theoretically attractive and also more data efficient than naive policy gradients, TRPO has seen limited practical success due being difficult to implement and computationally expensive to scale as a second-order method. Proximal policy optimisation (PPO) attains similar performance but implements a conceptually simpler optimisation which can be executed via simple stochastic gradient descent [SWD+17]. This is achieved by bounding the likelihood ratio between prior and updated policy through a clipped objective function. In a popular variant, collected sample trajectories are repeatedly sub-sampled, and multiple updates are applied until a pre-defined threshold in KL-divergence is reached. PPO-variants are widely used in practice due to being significantly more sample efficient and robust than policy gradients.

Inquiries closer to systems are concerned with parallel, distributed and asynchronous training methods. I discuss these execution mechanisms separately in §3.1 in the context of deriving the requirements for RLgraph, my framework for designing and executing RL algorithms. The multitude of subtle algorithmic variations means practitioners must choose which heuristics apply to their problem. For example, a problem with expensive evaluation (i.e. time to obtain a reward) can benefit from a more expensive optimisation to minimise sample requirements. Algorithmic frameworks must hence provide a set of well-tested standard implementations and enable transparent configurations of variants.

2.2.4 Reinforcement learning terminology

For clarity, I contextualise the algorithms above in relation to other RL sub-fields. The algorithms I discussed are principally single-task, model-free reinforcement learning algorithms. Solving real-world problems with physical agents (e.g. robots) may require solving a varied sequences of tasks (e.g. navigate, open doors, manipulate objects).

My analysis of systems design challenges also focuses on single-task learning. First, multi-task approaches are often based on combinations of single-task approaches. Second, solving non-trivial instances of single-task problems remains difficult and requires significant analysis, hyperparameter tuning, and large numbers of samples. I then separately explain

how design assumptions translate to multi-task scenarios. In the same vein, I first discuss single-agent scenarios. N.b. the distinction where multi-agent scenarios can both refer to multiple agents learning a single task, or multiple agents training on independent or causally related tasks.

The term *model-free* refers to the fact that the algorithms above learn a policy or value function without using a prediction of the next state or next reward. In model-based reinforcement learning, the agent learns a model $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ of global system dynamics, i.e. $s_{t+1} = f(s_t, a_t)$, to use for action evaluation. Historically, model-based control has been particularly successful in physical control systems such as robotics where motion trajectories may follow known dynamics with closed-form solutions. A survey of these traditional approaches is given by Kober et al. [KBP13]. In the context of deep reinforcement learning, exploratory approaches have tested network architectures incorporating planning modules [HS18, WRR$^+$17].

2.3 Iterative optimisation

In this section, I briefly summarise a number of related optimisation techniques. The overview restricts itself to the relationship of these methods to RL. For comprehensive treatments, I refer the reader to e.g. Audet's survey on black box optimisation [Aud14] and a review of Bayesian optimisation [SSW$^+$16]).

2.3.1 Bayesian optimisation

Bayesian optimisation (BO) is a black-box optimisation technique based on building a probabilistic model of an objective function. In common variants, A Gaussian Process (GP) with mean

$$m(x) = \mathbb{E}[f(x)] \tag{2.12}$$

and covariance function (or kernel)

$$k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))] \tag{2.13}$$

is a stochastic process where each finite subset of random variables can be described via a multi-variate normal distribution (c.f. Rasmussen's handbook on GP modeling [RW06]):

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')) \tag{2.14}$$

An acquisition function is used to identify the next best point to evaluate with a common choice being the expected improvement. The GP is updated to condition the model on each new observation by analytically computing the posterior predictive distribution. A tutorial with practical details on acquisition and covariance functions is given by Snoek et al. [SLA12]. While popular for hyper-parameter optimisation, Gaussian progress regression can also be used to address reinforcement learning problems. For example, PILCO is a model-based RL algorithm which solves continuous control tasks by fitting a dynamics model with a GP [DR11].

GP-based approaches are attractive in data-sparse environments as they can often identify substantial improvements within few (<25-50) trials. As non-parametric models,

a main drawback is computational efficiency in the number of samples, as updating the posterior requires an expensive matrix inversion. Neural network approaches in contrast succeed when feature-rich representations of input data can be learned. Moreover, GP performance relies on the kernel as an accurate model of the distance between two points in the input region. For non-smooth objective functions (e.g. in combinatorial problems), BO typically fails to find an efficient solution unless provided with a hand-crafted kernel.

2.3.2 Random-search and evolutionary methods

Various other approaches may be used to solve tuning problems. Simple random search has been shown to be effective for black-box optimisation tasks [BBBK11]. Mania et al. showed that an augmented random search could also identify high-performing policies for continuous control tasks commonly used as benchmarks for policy gradient algorithms [MGR18]. Random search variants are particularly attractive on tasks which are cheap to evaluate, as e.g. the variant explored by Mania et al. is over an order of magnitude more computationally efficient than common policy gradient methods.

A similar argument was made earlier by Salimans et al. who argued evolutionary methods could provide an alternative to RL due to admitting simpler distributed experimentation [SHCS17]. Evolutionary methods mutate parameters judged by a fitness function. While sacrificing an order of magnitude in sample complexity against gradient-based methods, evolutionary methods require minimal synchronisation between parallel actors. Random search or evolutionary methods also rely on fewer hyper-parameters than Q-learning or policy gradient algorithms.

In this dissertation, I investigate optimisation problems in computer systems where runtime is almost entirely dominated ($> 95\%$, c.f. Chapter 6) by waiting on the problem environment to execute and evaluate decisions. The trade-off offered by computationally cheap methods is hence undesirable.

2.4 Reinforcement learning in computer systems

This section presents recent representative application areas of RL in systems. In Chapter 5, I classify prior work in detail through my evaluation protocol. The works here both address RL for blackbox optimisation where RL competes with the techniques discussed in the prior section. They also include dynamic decision-making problems where an RL agent fully replaces algorithmic components (e.g. a scheduler).

Early work on RL in systems can be found between 1990 and 2000 in routing (Q-routing) and protocol optimisation [BL94, KM97, KM99]. These works compare tabular Q-learning variants to traditional shortest path algorithms and note the future potential of function approximators. Practical implementations of using neural networks in combination with RL in systems can be found as early as 2006 in Tesauro et al.'s work on server resource allocation [TJDB06]. The authors encoded as state the current and prior request arrival rates, and decided which server to allocate to respond to a request. A one-layer neural network was trained via Q-learning. Learning was further helped by initially making online decisions using a rule. A decade after Tesauro's work, Mao et al. [MAMK16] again used RL for resource management, and applied policy gradients to a simplified simulated scheduling problem.

The broader availability of deep RL implementations has since given rise to new research into many traditional systems domains. The dominating theme is that traditional systems research has yielded algorithms and systems carefully optimised for a multitude of objectives and classes of workloads (e.g. generic patterns like read-only workloads, long-running transactions). RL in contrast promises to make fine-grained per instance decisions which exploit high-granular workload properties.

Recent applications include investigations into many networking problems which can benefit from existing protocol simulators. Valadarsky et al. for example learn simulated routing decisions on small topologies but report difficulties training on larger network structures [VSST17]. Neural packet classification [LZJS19] learns a stochastic policy to generate decision trees which minimise classification time. Pensieve [MNA17] proposes to adapt client-side video streaming rates using policy gradients.

In databases, Marcus et al. utilised proximal policy optimisation to investigate learning join orders in PostgreSQL [MP18]. Join ordering is an attractive learning task as execution time is low, i.e. either obtaining a cost estimate from the query planner or executing the query directly to evaluate runtime. Their initial join-order project was later extended to a RL-based query optimiser which bootstraps training from the PostgreSQL optimiser [MNM+19]. Krishnan et al. [KYG+18] completed a similar study of RL applicability in join ordering using DQN. Durand et al. further proposed to tune data-partitioning with DQN [DPP+18]. In NoDBA, Sharma et al. [SSD] investigated a simplified indexing problem where single-column indices are selected for a relational database.

Later work in scheduling [MSV+18a] has focused on effective state representations wherein Spark [ZCF+10] dataflow jobs are encoded via graph neural networks. Multiple output tasks are used to select parallelism and job stage to execute. The resulting controller, trained in a Spark simulator, significantly improves job completion times. Li et al. proposed a scheduler for distributed stream processing where action selection in a large action space was addressed via a k-nearest neighbour approach [LXTW18].

Mirhoseini et al. demonstrated how to use attention-based and hierarchical methods known from neural machine translation to effectively perform TensorFlow device placements [MPL+17, MGP+18]. Jia et al. later demonstrated with FlexFlow [JZA18] that a hand-crafted simulator combined with greedy search (and no RL) could significantly outperform these solutions. The hierarchical placer however does not require a simulator, and FlexFlow did not evaluate how a comparable RL algorithm would perform when trained in their simulator. In a similar context of static optimisation, Ali et al. [AHM+19] used DQN to select order and type of compiler pass optimisations using the LLVM tool-chain.

Algorithm choices in the work I surveyed were often not motivated from an RL perspective. An example of this is to learn a discrete stochastic policy with on-policy training for a deterministic problem (e.g. for join ordering [MP18]). This could be due to a lack of understanding of algorithmic properties and the applicability of specific heuristics, thus indiscriminately relying on open source implementations.

2.5 Summary

This chapter introduced the reinforcement learning problem and different approaches to solve it. First, I discussed two common classes of algorithms, Q-learning and policy gradient methods. I then highlighted how using neural networks as function approximators in conjunction with a number of other heuristics has sparked a wave of new research.

I further described follow-on developments to improve initial deep RL variants which are now canonically used in practical implementations. Finally, I surveyed a number of competing tuning methods and prior work in systems-RL.

In the next chapter, I will present an analysis of execution and design aspects of reinforcement learning algorithms, before I introduce RLgraph, my backend-agnostic engine for design and execution of RL algorithms.

Chapter 3

RLgraph: Modular computation graphs for reinforcement learning

Emerging research and novel applications of deep reinforcement learning have given rise to a multitude of computational challenges. RL applications exhibit highly varied task parallelism, communication patterns, and resource requirements. Their execution semantics are also much different from supervised workloads due to the need to interact with one or more problem environments during training. Systems design has thus far not caught up with the rapidly evolving needs of these new applications. In this chapter, I make the following contributions:

- I first present a survey of reinforcement learning workloads and design challenges. I also discuss assumptions in existing abstractions and subsequently argue why they are insufficient to meet evolving requirements (§3.1).
- Based on this analysis, I introduce RLgraph, an architecture which decouples the design of reinforcement learning mechanisms from execution semantics to support different distributed execution paradigms and flexible application semantics (§3.2 - §3.8).

I begin by discussing reinforcement learning workload characteristics to derive the requirements for RLgraph. Parts of this chapter have been published as a research paper [SMFY19].

3.1 Reinforcement learning workloads

This section motivates systems design for RL by considering the following questions:

- How are RL algorithms benchmarked, and how do these benchmarks drive research progress?
- In what ways do these workloads differ from other machine learning tasks?
- What do these differences imply for the implementation of RL software systems?

3.1.1 Use of simulators

Simulators are essential tools of RL research. The first wave of deep RL algorithms following Mnih et al.'s work have often been evaluated on the Arcade Learning Environment (ALE, [BNVB13]). ALE presents a simple action and observation interface to classical Atari 2600 games (e.g. Pong, Breakout). A single-threaded program can execute thousands of steps (state transitions) per second. The MuJoCo (for Multi-Joint dynamics with Contact [TET12]) physics engine provides a similar test bed for continuous control tasks [DCH+16]. MuJoCo tasks require agents to coordinate multiple joints to learn mechanical tasks such as walking, swimming, hopping, or controlling a humanoid. OpenAI gym (in the following as "*gym*" [BCP+16]) provides a unified interface to these benchmarks and various other simulators. Gym has become the de-facto standard for implementing environment interfaces in the open source RL community. Implementing the gym interface for a custom environment only requires implementing *step* and *reset* methods to advance the state/reset the environment, and two properties to describe the shape and types of states and actions.

For ad-hoc optimisation and control problems in computer systems, simulators are often not readily available. On the one hand, systems applications of RL interface software systems which are easy to parallelise, copy, and inspect for internal state like simulators. On the other hand, executing training workloads on databases, compilers, stream processors or distributed data processing engines requires substantial resources. Moreover, one state transition (e.g. evaluating a configuration on a workload) may take seconds to minutes, thus requiring sample-efficient approaches. I discuss the unique challenges of RL in systems in Chapter 5.

Listing 3.1 shows the basic gym interface which is sufficient for execution in gym-compatible frameworks. The interface can be extended with additional methods to pass a random number seed, or to render the environment.

ALE and other simulators provide convenient test-beds for research due to a number of benefits:

- **Computationally cheap.** Simulators for classical games are inexpensive to execute. A single-threaded controller can scale to thousands of frames (210×160 pixels, 7 bit colour palette) per second. This includes neural network forward passes to compute actions and more expensive update operations. They are hence easily parallelised.

- **Stop-and-go execution.** Simulators such as ALE do not have real-time requirements as the game is not advanced until the step-function is invoked. This dramatically simplifies execution semantics because expensive update-steps block action requests on the network. Real-time environments require asynchronous learning and acting in separate processes.

- **Deterministic.** While real-world environments exhibit noisy and stochastic behaviour, debugging and analysis can be helped by deterministic simulators. This in turn requires to manually inject stochasticity for more realistic training. For example, Mnih et al. created a random initial state in Atari games by first taking a random number of no-op actions.

These properties have made ALE/MuJoCo simulators the most popular choices during a first wave of deep RL research following the publication of DQN. They influenced views

```
1
2  class GymInterface(object):
3    """Abstract environment for OpenAI gym execution."""
4
5    def __init__(self, observation_space, action_space):
6      # Shape, type, and number of state inputs.
7      self.observation_space = observation_space
8      # Shape, type, and number of actions.
9      self.action_space = action_space
10
11     def step(self, action):
12       """
13       Advances the environment a single step by executing an action.
14
15       Args:
16         action (any): Actions to execute.
17
18       Returns:
19         state (any): Next state observed after executing action.
20         reward (float): Reward observed after executing action.
21         terminal (bool): True if action led to a terminal state.
22         info (any): Optional meta data.
23       """
24       raise NotImplementedError
25
26     def reset(self):
27       """
28       Resets the environment and returns an initial state.
29
30       Returns:
31         state (any): Initial state after resetting (may be random).
32       """
33       raise NotImplementedError
```

Listing 3.1: OpenAI gym environment interface.

on executing training workloads (inexpensive parallelism) and evaluation. While ALE for instance provides a variety of game tasks (e.g. Pong, Breakout, Lunar Lander), quantifying task difficulty and in turn understanding algorithmic improvements on tasks has proven difficult. They are prone to over-fitting as the evaluation tests the same game the agent was trained on. Limited overlap between the systems and RL community meant a virtual absence of systematic software design and evaluation techniques, as research initially centred around variations of ad-hoc script implementations.

These challenges have called into question algorithmic improvements based on tuning problem-specific heuristics. Henderson et al. observed that subtle implementation issues and random initialisation drastically affect performance [HIB+17]. Mania et al. subsequently demonstrated that an augmented random search outperformed [MGR18] several policy optimisation algorithms on supposedly difficult control tasks. Further work on policy gradient algorithms observed that the performance of popular algorithms may depend on implementation heuristics (e.g. learning rate annealing, reward normalisation) which are not part of the core algorithm [IES+18].

In the wake of these findings, researchers have recently proposed new specialised simulators to benchmark specific properties such as generalisation capabilities (CoinRun [CKH+18]) or agent safety (e.g. DeepMind safety gridworlds [LMK+17]). To interface open source implementations of DRL algorithms, practitioners in emerging applied domains have similarly adopted gym-style interfaces in novel simulators. For example, Siemens have introduced a benchmark for industrial control tasks [HDT+17]. Others have built gym-bridges and new problem scenarios on top of existing simulators such as the ns3 networking simulator (ns3-gym [GZ18]).

The substantial sample requirements (up to billions of state transitions [BLT+16]) of model-free algorithms have resulted in implementation efforts to be overwhelmingly focused on interfacing simulators. Emerging simulator tasks can be implemented in photo-realistic environments using agent-interfaces to game engines such as Unity [JBV+18a]. Unlike cheaper early simulators, game engines compete with neural networks for accelerator resources. The trend towards high-fidelity task simulations requires larger distributed training environments to achieve higher sample throughputs. Implementations consequently need to accommodate a wide range of resource sharing scenarios. Next, I analyse how simulators have affected the development of scalable training mechanisms.

3.1.2 Distributed reinforcement learning

The light-weight nature of early simulators has given rise to many parallelisation schemes. I give an overview of key algorithms below.

The first widely popular of such schemes was the asynchronous advantage actor-critic (A3C, also by Mnih et al. [MBM+16]). In the original A3C paper, a single-node many-core machine was used to train a policy by means of workers asynchronously updating a shared global policy. Each worker interacts with its own environment simulator instance, collects a number of state trajectories (a tunable hyper-parameter), locally computes gradients for an update, and then applies this update to the globally shared policy network. After each update cycle, workers copy the global parameters (which may have been updated many times by other workers) to their local copy. Mnih et al. advertised A3C as a faster and cheaper alternative to DQN because it does not require a GPU (each actor computes its update on a CPU), while training substantially faster than a single-threaded agent. A3C

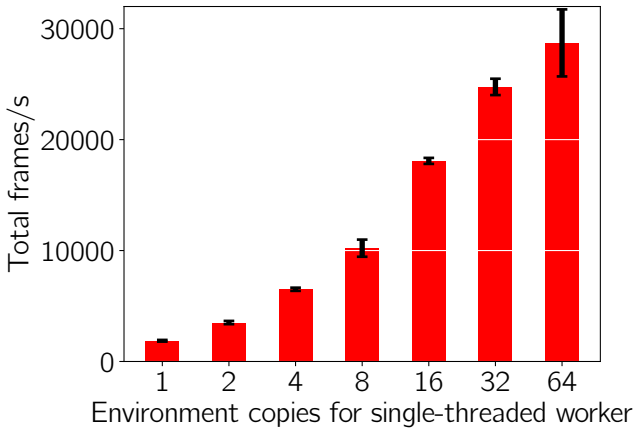


Figure 3.1: Environment vectorisation on CartPole. A single-threaded worker vectorises action selection, then sequentially acts on a list of simulator copies.

serves as an example design implicitly relying on cheap simulators.

Similar to advances in policy optimisation, distributed and parallel variants of Q-learning implement a multitude of synchronisation mechanisms. Gorila is an early distributed DQN variant which shards learners and parameter servers and uses a central replay memory in which distributed actors insert new samples [NSB+15]. Distributed prioritised experience replay (APE-X [HQB+18]) is a more recent variant encompassing several levels of parallelism. Workers asynchronously collect environment trajectories and do not compute updates. Instead, they perform preprocessing to ease computational burden on the learner. Trajectories are inserted into one or multiple replay buffers which provide prioritised sample batches to an asynchronous learner thread.

Note that in distributed training, each worker, irrespective of the distributed coordination mechanism, may itself employ further parallelism by vectorising action selection, i.e. batching states from multiple environment copies into a single forward pass. A single-threaded worker can substantially increase throughput on cheap tasks by sequentially acting on multiple environments and batching action selection in the neural network (Figure 3.1).

Figure 3.2 illustrates a single-learner task architecture with distributed sample collection. The learner schedules a set of actors which may be distributed across a cluster but which can also be separate threads or processes on the same node. Actors interact with dedicated environment copies but do not execute update operations. They batch-process new sets of states from environments, and merge trajectory data from separate environments. The learner post-processes trajectories (e.g. by computing discounted returns), then schedules updates on accelerators.

Sample collection, weight synchronisation, and updates may all be synchronous or asynchronous which gives rise to additional considerations to store trajectories. Distributed communication and coordination strategies in RL are constrained by algorithms' ability to incorporate off-policy data on the one hand, and by accelerator throughput and communication cost on the other hand.

Research into better replay mechanisms remains an active topic [KOQ+19]. The conceptual simplicity of many distributed training schemes also leaves much to be desired for replay implementations, irrespective of algorithms used. Common distributed training schemes transmit the same states repeatedly as actors blindly recollect and retransmit

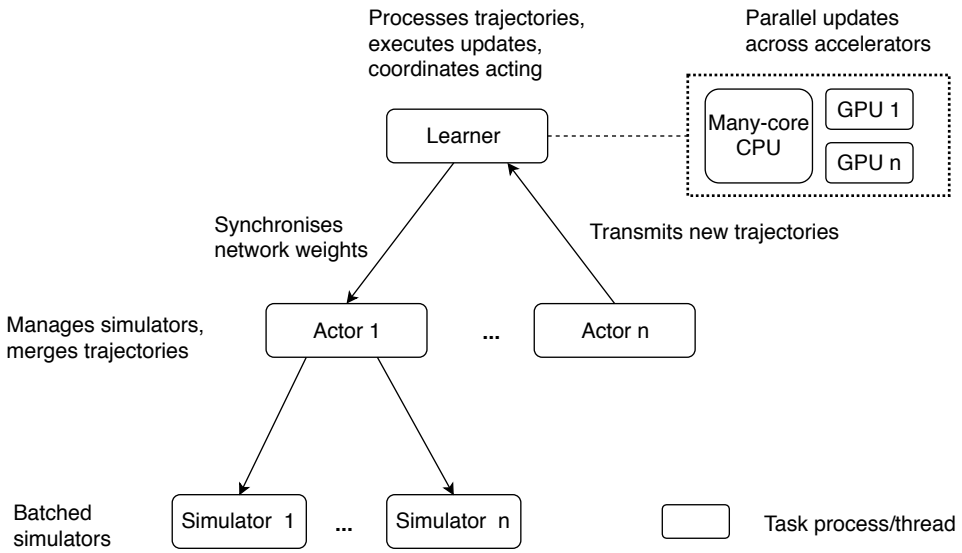


Figure 3.2: High-level overview of single-learner task parallelism. Even in single learner scenarios, multiple levels of task-parallelism require fine-tuning of a multitude of hyper-parameters for distributed coordination.

trajectories without regard for communication cost.

3.1.3 Use of accelerators

The increasing availability of GPUs and specialised accelerators [JYP+17, JYPP18] has given rise to newer distributed training schemes. In distributed policy optimisation, Babaeizadeh et al.[BFT+17] for instance suggested a GPU-variant of A3C in which both action selection and learning are queued and batched to a centralised GPU. Others observed no benefit from the noise introduced by asynchronous updates, and instead proposed a synchronous variant (A2C) to facilitate GPU batching ([WMG+17], also described by Clemente et al. in parallel [CMC17]). A central challenge in asynchronous distributed policy optimisation is incorporating off-policy data, i.e. data collected from a prior policy π_{t-k} (for a typically small integer k). The IMPALA algorithm further advanced scalable policy optimisation through a novel off-policy correction facilitating aggressive use of caching to optimise GPU throughput [ESM+18].

Decoupling sample collection from updates improves throughput and is especially useful when learning multiple problems in parallel with varying trajectory lengths. A feedback loop between algorithmic insights, improved accelerators and larger tasks in more powerful simulators drives new training schemes. To summarise key aspects of distributed RL:

- The mechanisms are generally invariant to modifications in network architecture or loss function, as long as no gradient estimation bias is introduced.
- Algorithmic improvements are in practice only selectively benchmarked for computational scalability due to cost.
- Practitioners must select between a growing body of distribution schemes even at the level of single agents solving single tasks.

Multi-learner and multi-agent scenarios create further complications in synchronising training and resource sharing. As new simulators, accelerators and algorithms advance, RL frameworks must provide mechanisms to explore new training semantics.

3.1.4 RL implementations and design problems

The growing interest in RL has given rise to a wide array of implementation styles and design patterns which can be traced back to original research implementations. Here, I describe how deep RL tasks are commonly implemented and executed on popular machine learning frameworks. The recent wave of new research and applications in deep learning has been fuelled not only by hardware improvements but also by deep learning frameworks simplifying design and training of neural networks [CLL+15, ABC+16, SA16, PGC+17].

Neural network architectures are most commonly implemented using Python, and by interfacing high level APIs provided by popular engines like TensorFlow [AIM17] or PyTorch [PGC+17]. These frameworks implement large toolboxes of neural network layers, optimisation methods (e.g. gradient descent), and high-performing numerical operations. They also map high level numerical operations to efficient GPU kernels (e.g. CUDA, OpenCL), or to multi-threaded CPU execution, thus drastically accelerating development. The core entity these frameworks operate on are tensor objects as multi-dimensional arrays with support for storing gradients for automatic differentiation. In deep learning workflows, developers typically manage I/O with Python glue code, as runtime is dominated by neural network operations executed in C/C++ on accelerators.

After network construction, there is little or no conditional branching required to perform inference or to train the network. Irrespective of the network architecture, forward passes and updates require a series of matrix multiplications and additions to perform layer operations. These numerical operations can be effectively represented via static computation graphs whereby e.g. in TensorFlow calling layer operations adds computation nodes to a dataflow graph. This computation graph can then be statically optimised (details on TensorFlow programming semantics are given by Abadi et al. [AIM17]). Alternatively, in define-by-run frameworks such as PyTorch [PGC+17], operations are executed imperatively by an interpreter which traces graph definition through program execution, thus facilitating dynamic graphs. I discuss graph construction for RL in §3.3.

As a result, supervised deep learning workflows have seen a high degree of standardisation. This allows developers to focus on effective architecture design. Listing 3.2 illustrates this using the Keras [C+15] neural network API. Layers, loss functions, optimisation mechanism and training are abstracted via an object-oriented API. Crucially, training inputs and labels are generally available in advance to facilitate straight-forward batching and I/O.

In contrast, as discussed in §3.1, RL workloads exhibit heterogeneous communication and parallelism patterns as they interact with one or more environments while processing trajectories. These workload characteristics, combined with fast-evolving empirical research, have thus far made it difficult to provide standardised software support for designing and executing reinforcement learning models.

RL algorithms are consequently not natively integrated into frameworks like TensorFlow but implemented in bolt-on libraries. Many open source RL implementations can be categorised as reference implementations. For example, OpenAI baselines [DHK+17] and Google's Dopamine [BCG+18] provide collections of well-tuned algorithms on gym

```
1
2 # Networks are assembled by sequentially stacking layers.
3 model = Sequential()
4 model.add(Dense(32, activation='relu', input_dim=10))
5 model.add(Dense(32, activation='relu'))
6 model.add(Dense(10, activation='softmax'))
7
8 sgd = SGD(lr=0.001)
9 model.compile(loss='categorical_crossentropy',
10               optimizer=sgd,
11               metrics=['accuracy'])
12
13 # High level models facilitate training and evaluation.
14 model.fit(input_data, input_labels, epochs=20, batch_size=256)
15 score = model.evaluate(test_inputs, test_labels, batch_size=256)
```

Listing 3.2: Assembling neural networks with the Keras API.

benchmarks. Nervana Coach [CLN17] contains a similar collection but with added tools for visualising progress and generic facilities for hierarchical learning and distributed training. Batched PPO [HDV17] contains a single tuned implementation of a popular algorithm (proximal policy optimisation (PPO) [SWD+17]) whereby all control flow and environment stepping have been merged into one end-to-end TensorFlow graph. Keras-rl [Pla16] provides a loose collection of algorithms (e.g. DDQN [HGS16], DDPG [LHP+16]). Horizon focuses on building end-to-end pipelines for off-policy training at Facebook [GCL+18].

These algorithm collections typically share some components between algorithms (e.g. network architectures) but ignore many of the practical considerations discussed above. Retooling such reference implementations to different execution and batching modes or device strategies (multi-GPU support, cluster environments) requires significant work. This is also because these implementations typically contain hard-coded heuristics for the use in specific simulators. As algorithm performance critically depends on tuning state preprocessing and learning heuristics [HIB+17, IES+18], developers must tediously analyse undocumented steps (e.g. reward clipping, state normalisation, gradient clipping) to re-use reference implementations.

Higher level frameworks focus on providing common APIs and abstractions. Ray RLlib [LLN+18] defines a set of abstractions for scalable RL with focus on task parallelism. RLlib relies on Ray's actor model [NMW+] to execute RL algorithms via centralised control. Ray is a distributed execution engine supporting both parallel tasks (e.g. simulations) and actors (e.g. stateful RL agents) through a unified programming interface. Ray manages resource sharing between tasks through a multi-level scheduling model where a global scheduler keeps track of shared state via an object store, and a local scheduler manages per-server resources. RLlib is a reinforcement learning library serving as an example application for Ray. At the core of RLlib's hierarchical task parallelism approach lies a set of optimiser classes. Each optimiser implements a *step()* function which distributes sampling to remote actors, manages buffers, and updates weights.

For example, an *AsyncReplayOptimizer* implements distributed prioritised experience replay [HQB+18]. Each step, the optimiser loop fetches samples from remote actors, inserts them into local replay buffers, and performs training on an asynchronous learner thread

by sampling from the buffers. The advantage of RLlib is the separation of the execution plane in the optimiser and definition of the RL algorithm's model within a policy graph. However, each optimiser encapsulates both local and distributed device execution. This means for example that only the dedicated multi-gpu optimiser class supports splitting input batches synchronously over multiple GPUs. A further disadvantage of this optimiser driven control flow is that RLlib mixes Python control flow, Ray calls, and TensorFlow calls in its components. Algorithms implemented in RLlib are hence not easily portable as training can principally only be executed on Ray. Sample pre- and post-processing are coupled with distributed training.

3.1.5 Design summary

Existing abstractions almost exclusively focus on simulator workloads with the implicit assumption of computationally cheap stop-and-go simulators. When interacting with non-simulated environments, e.g. a physical system such as robots, or data processing engines (e.g. databases), training control flow may be driven by external system events. When learning on real systems, learning is often bottlenecked by sample collection, not communication overhead or accelerator throughput. In summary, RL frameworks must:

- resolve the tension between fast prototyping through reusable components and robust interfaces,
- offer flexible distributed and local execution patterns agnostic to simulation-driven or external control flow,
- and transparently manage configurations for a large number of optional learning heuristics to incrementally test and validate training behaviour.

Many existing implementations are designed to work well in their target domain (e.g. reproducing ALE results) but neglect these considerations. They either sacrifice flexibility by only reproducing selected algorithms without intention of reuse, or focus on parallelism but restrict algorithms to narrow design assumptions.

3.2 RLGraph overview

I present RLgraph, a reinforcement learning framework designed to address these difficulties. RLgraph provides a bridge between the deep learning functionalities needed for deep RL found in popular machine learning frameworks, and the control flow needed to coordinate RL scenarios. I base RLgraph on these design principles:

- **Separating algorithms and execution.** RL algorithms require complex control flow to coordinate distributed trajectory collection and training logic. Separating these aspects is difficult but essential to avoid re-implementing execution and device strategies (§3.4.4).
- **Shared components, strict interfaces.** Deep learning frameworks enable quick prototyping of neural networks by exposing APIs to combine different types of layers. Providing a similar set of interchangeable components towards RL is complicated by the multitude of learning and execution semantics. This is exacerbated by

implementations containing definitions in multiple execution contexts, e.g. Python control flow interleaved with calls to TF runtime. Tight coupling of components, and in turn a lack of well-defined component boundaries means that re-usability is severely constrained. To ensure re-usability of both high-level abstractions and narrow heuristics, logical components should only interact using well-defined interfaces (evaluated in §4.6).

- **Training heuristics as first class citizens.** Brittle implementations arise when performance-critical heuristics are not configurable or tested, but hard-coded. Algorithms may still learn sub-optimally with faulty heuristics, as training performance variance is high even in best case scenarios. Heuristics consequently must be implemented and used as first-class citizens, and ultimately understood to be core performance elements.

- **Sub-graph testing.** An undesirable consequence of incorporating stochastic approximations at all levels are numerical sensitivity and non-determinism [NWS18b]. RL algorithms can require an overwhelming number of hyperparameters (often in excess of 25). Testing partial dataflow is highly desirable but tedious to realise in static graph frameworks. RL frameworks must provide mechanisms to incrementally build and test complex dataflow (§3.5).

I introduce RLgraph, a modular, backend-independent framework to construct and execute deep reinforcement learning algorithms. By separating logical component composition and execution, RLgraph can support static graph and define-by-run execution (§3.4). RLgraph is open source[1].

At the centre of RLgraph's design is a component graph architecture responsible for assembling and connecting algorithmic components (e.g. buffers or neural networks) as named subgraphs, and for exposing their functionality via a common API. This component graph exists independently of implementation-specific notions, and instead relies on generalised dataflow types and operations. The component graph is built into a backend-dependent computation graph via a graph builder which generates operations, graph state, device assignments, and an API registry.

A graph executor expands the component graph to add operations for local and distributed device strategies, e.g. by creating subgraph replicas for GPUs. At runtime, the graph executor serves requests to the agent API. In define-by-run mode where no static graph is constructed ahead of execution, the build process is used to validate that all defined operations can execute without error.

Figure 3.3 gives a high-level view of RLgraph in relation to deep learning frameworks and distributed execution engines. At the user-level, RLgraph exposes an agent API which allows plug-and-play execution of pre-built RL models on common simulators or custom applications. Pre-built models can be declaratively configured to enable rapid exploration of network architectures, execution models, and training heuristics. These models are implemented using RLgraph's component composition and executed on a combination of local and distributed execution engines. Next, I introduce RLgraph's design abstractions.

[1] `https://github.com/rlgraph/rlgraph`, accessed 25.08.2019

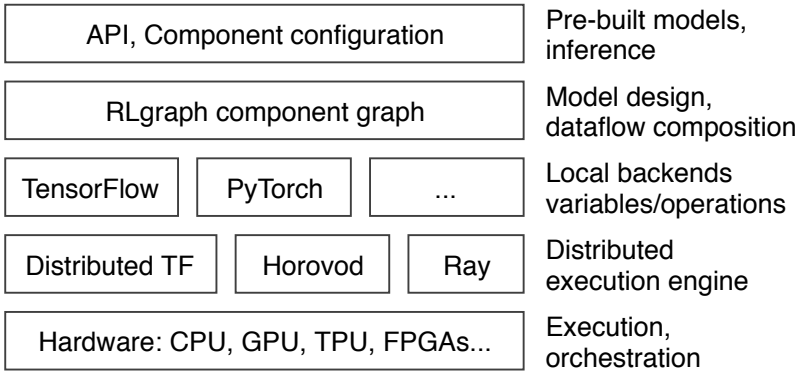


Figure 3.3: RLgraph software stack.

3.3 Design

Which software primitives or abstractions are appropriate for composing RL algorithms? Continuous environment interaction creates data streams which are processed through a series of computations which drastically differ in cost, from down-sampling images to gradient-descent steps. Dataflow programming as a series of numerical transformations is the paradigm behind popular machine learning frameworks like TensorFlow [ABC+16]. End-to-end graph representations enable compile-time optimisations (§3.6.2), model serialisation, and minimise context switches between host language and graph runtime to improve performance [YAB+18, PTS+17]. In existing implementations, I observe:

- An impedance mismatch exists between the largely functional transformations in static graphs and RL algorithms which require complex control flow and communication between high-level logical entities.
- This mismatch causes implementations to loosely tie together graph-based dataflow for numerical operations and imperative scripting to facilitate communication and logic, thus creating a host of bugs and design issues discussed in the prior section.

The aim of my design is to facilitate algorithm design at the level of logical high level components (e.g. buffers, data pre-processors, neural networks, action distributions) while preserving the advantages of end-to-end static graphs.

RLgraph must (i) allow users to group computations within familiar logical entities which encapsulate internal graph state and (ii) combine these entities into a graph while respecting different execution considerations such as distributed communication, resource assignments, and local/global state sharing. The arrangement of components into distinct coordination semantics (e.g. distributed asynchronous) must be independent of the logic contained within components.

3.3.1 Components

RLgraph's module abstraction is hence a *Component* class which encapsulates arbitrary computations via so called *graph functions*. Components provide an object-oriented interface to building static computation graphs.

Consider a replay buffer component which exposes functionality to insert trajectories and sample mini-batches according to priority weights. Implementing this buffer in an

imperative language such as Python is straight-forward, but including it as part of a static graph requires creating and managing state variables through control flow operators (e.g. to update priorities in an appropriate data structure). Composing multiple components in a re-usable way is difficult due to the impedance mismatch between class-based programming in a driver language, and functional transformations within a dataflow graph.

Existing high-level APIs for neural networks such as Sonnet, [Dee17], Keras [C+15], or Gluon [R+18] focus on assembly and training of neural networks. Implementing RL workloads in these frameworks requires mixing imperative Python control flow with deep learning graph objects, leading to the design issues discussed in §3.1.

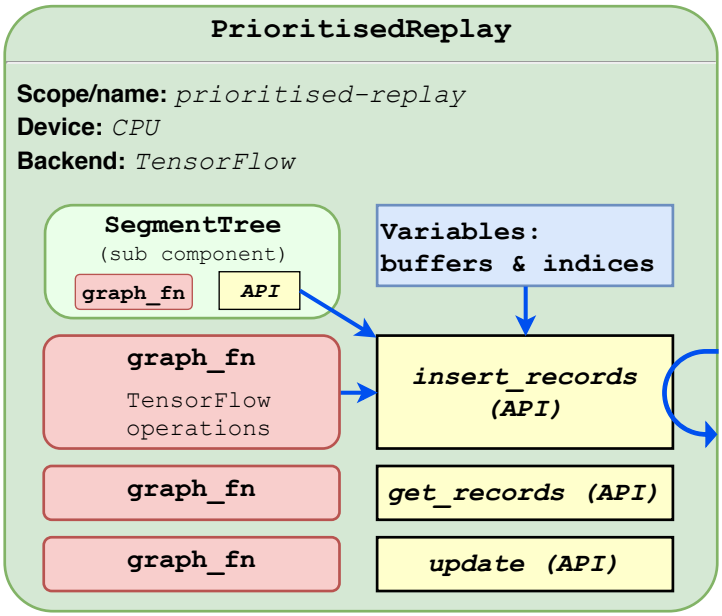


Figure 3.4: Example memory component with three API methods.

Figure 3.4 shows a concept of a prioritised replay [SQAS15] component. Components are identified by name which corresponds to the scope used when e.g. creating TensorFlow variables. Scoping is a concept to logically group state and computations into named sub-graphs. RLgraph names components and maps them to appropriate scopes if scoping is used by the underlying static graph. Component functionality is exposed via API methods which are declared via method decorators. Components encompass state and computations. They are executed on a configurable device, and state may be read from and written to from the same or a different device. When a component is built (§3.3.2), its internal state is defined based on input spaces, i.e. inputs to the graph or results of intermediate dataflow. For example, the input to a storage buffer may use a type and layout inferred from a series of transformations of graph inputs.

The difference between a native object method and an RLgraph API method is that API methods are traced in the build process into an intermediate representation. An operation registry is built which maps corresponding tensor operations to required input arguments. In define-by-run mode, the registry is used to control gradient-taping for automatic differentiation and conversion of input data to tensor types. Graph functions implement backend-specific functionality for API methods. Graph functions:

- Group dataflow computations corresponding to logical high-level entities.

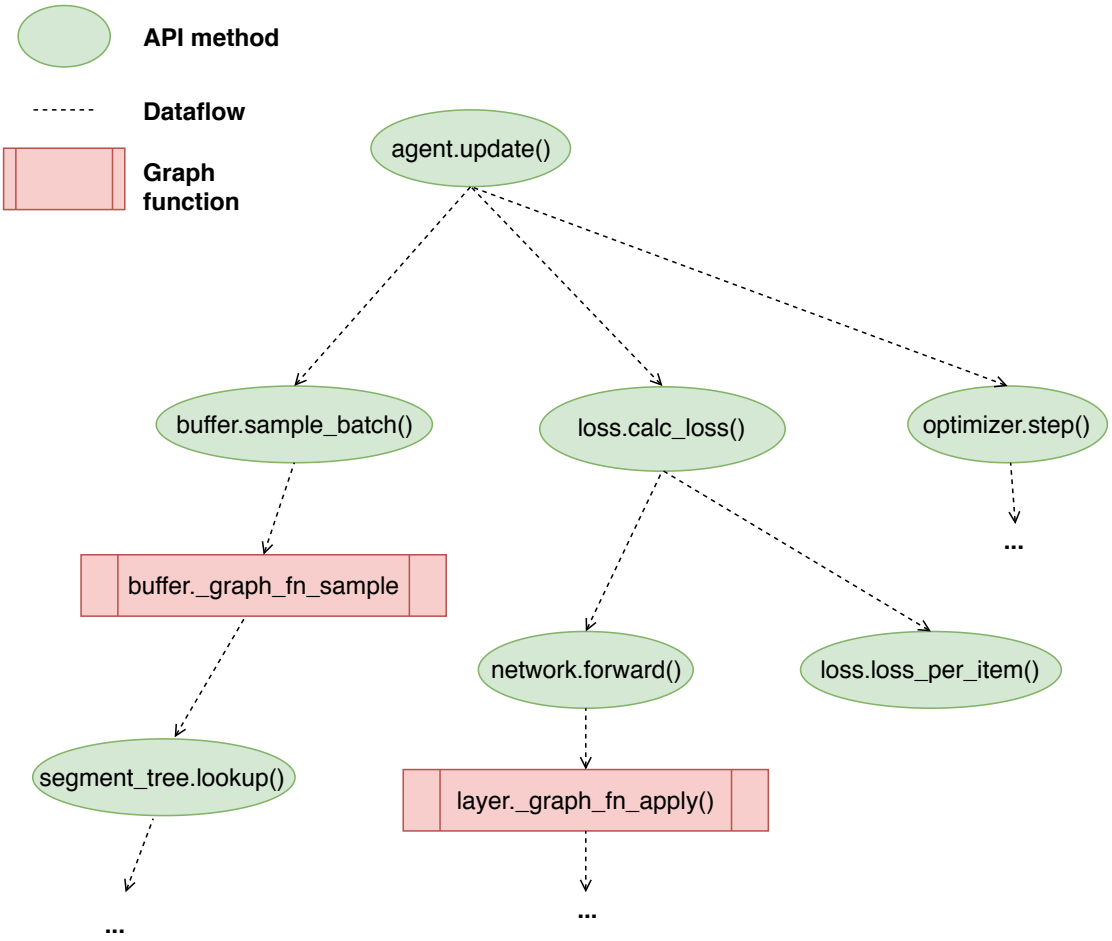


Figure 3.5: Simplified dataflow between API methods and selected graph functions for a training update method. The update-API method of an agent calls no graph functions but API methods of sub-components, which in turn call further sub-components. Where necessary, API methods are resolved by calling graph functions implementing backend-specific computations.

- Receive as input and return symbolic graph nodes in static graph operation.
- Receive and return tensor values in define-by-run mode.
- Can invoke other graph functions and API methods to coordinate computations.

This dataflow between components (and thus algorithm logic) is constructed in a backend-independent manner, as illustrated in Figure 3.5. The purpose of encapsulating computations within components is to flexibly combine stateful computations in static graphs.

The API decorator used to tag API methods and graph functions also modulates input and output dataflow to ensure components can be connected seamlessly. Dataflow between RL components often involves collections of high-dimensional matrices which may be arbitrarily nested or arranged. For example, experience trajectories may be sequences of state-transition tuples (s_t, a_t, r_t, s_{t+1}). Arbitrary composition of components hence requires a *compatible dataflow*. In RLgraph, compatibility is ensured via *space objects* which define type and layout (i.e. dimensionality) of dataflow.

Each space object can be viewed as a prototype of the dataflow expected to arrive at a graph function. Hence, space objects allow to construct dataflow graphs which may later

use backend-specific dataflow implementations in place of spaces. For example, records inserted into the priority buffer component may be unstructured collections of matrices. To access contents in a record, developers normally need to perform manual splitting and merging operations to manipulate individual entries. The layout of nested records (e.g. names of keys in a map) is hard-coded either in the function accessing the record, or via named function parameters. Numerical operations offered by deep learning frameworks do not operate on host-language container abstractions (e.g. Python lists or dictionaries, or C++ containers) but require framework-specific tensor objects.

Listing 3.3 illustrates this aspect for the *insert_records* functionality. Developers can either use native containers which require manual un-nesting (1.), or can hard-code tensor arguments in inflexible interfaces (2.). RLgraph API methods can request to flatten nested operations (and re-nest results based on key-structure) (3.). They can also request to split nested arguments and to unflatten and merge split results (4.).

API decorators hence allow developers to modify dataflow layouts to avoid hard-coding them into computations. While the prototype has been implemented in Python due to its popularity, the same principles could be implemented in any other host language to construct computation graphs. In summary, RLgraph components:

- offer an objected oriented API to construct backend-independent end-to-end computation graphs,
- allow control of execution behaviour and dataflow layout through API methods,
- can be composed and nested,
- and enable fine-grained control of devices for executing computations and managing internal state locations.

Next, I explain how components are used to construct the component graph.

3.3.2 Building the component graph

Algorithm logic is implemented by beginning with a root component as a graph container. Sub-components are then added to the root, and API methods are implemented to create dataflow between components. RLgraph's high level agent API (§3.4.1) is one example of a set of API methods exposing domain-specific functionality. Another example is a testing API used to test individual API methods when building single components (§3.5). To build the graph, the build algorithm traces all API methods from the root component its input spaces.

The component graph is a control-flow-graph (CFG) made up of API methods as vertices V and edges indicating dataflow between two API methods a_1, a_2. The CFG has the following properties:

- The root component's API methods are the entrance blocks, and there can be arbitrarily many exit blocks.
- Edges between API methods are restricted by component membership. An edge between a_1 and a_2 can exist under the following conditions:
  - $a_1 \in C_1$ and $a_2 \in C_1$, i.e. both API methods belong to the same component C_1.

```
1
2 # 1. "records" is a Python dictionary containing
3 # TensorFlow tensors
4 def insert_records(records):
5   # Method must assume unnested structure or flatten dict.
6   for name, tensor_op in records.items():
7     # Do something with tensor_op to create new graph nodes
8     assign_op = tf.assign(self.variables[name], tensor_op)
9     ...
10
11 # 2. Separate hardcoded arguments.
12 def insert_records(states, actions, rewards, terminals):
13   # Fixed arg handling reduces re-usability.
14   s_assign_op = tf.assign(self.states, states)
15   a_assign_op = tf.assign(self.actions, actions)
16   ..
17
18  # 3. RLgraph: Decorator can flatten, split, merge nested dataflow.
19 @rlgraph.api_method(flatten=True)
20 def _graph_fn_insert_records(records):
21   # Assign op, generic container operations.
22   self.buffer.assign(records)
23   ..
24
25 # 4. Record container is flattened (un-nested) and split,
26 # insert function called once per key,
27 # results are re-nested, auto-merged.
28 @rlgraph.api_method(flatten=True,
29    split=True, add_auto_key_as_first_param=True)
30 def _graph_fn_insert_records(key, records):
31   # Records is now a single tensor-op
32   # or tensor-value in define-by-run mode
33   return self.assign_variable(self.records[key], records)
34   ..
```

Listing 3.3: RLgraph decorators modulate dataflow to increase re-usability and flexibility.

```
# Dataflow type definitions necessary to build this API method:
spaces = dict(
  states=FloatBox((210, 160, 3), add_batch_rank=True)
  time_step=int,
  use_exploration=bool
)

# Defined within agent class, attached to root.
@rlgraph_api(component=self.root_component)
def get_preprocessed_state_and_action(root, states,
        time_step=0, use_exploration=True):
  # Preprocessed states is the result of an API method call,
  # either graph-op or tensor.
  preprocessed_states = self.preprocessor.preprocess(states)
  # Call another API method.
  return self.policy.sample(preprocessed_states)
```

Listing 3.4: API methods can be composed by calling own or other components' API methods.

  - $a_1 \in C_1$ and $a_2 \in C_2$ and C_2 is a (direct or indirect) sub-component of C_1.

  Components cannot call API methods of parent components to avoid implicit dependencies through cyclic call-chains.

- API method vertices can create edges using bounded loops and conditionals. Dynamic unbounded loops and recursion are not allowed in API methods but in graph functions which can implement arbitrary computations.

API methods can be combined within API functions and graph functions.

For example, Listing 3.4 illustrates an API method which preprocesses a state and retrieves an action by calling API methods on other components. At component build time, calling an API method does not create any backend operation (in static graph mode), and no values are passed through (in define-by-run mode).

The component graph is built dependent on the input spaces to the root component, or more precisely to all input arguments. In the example, input spaces for *states*, *time_step*, and *use_exploration* must be provided. The shape of *states* is an input argument usually provided e.g. via the gym interface. Only spaces to the root component need to be provided as input spaces to sub-components can be inferred from computations. For example, the *preprocessor* component may down-sample input states to a different shape, and the sub-sequent API call is dependent on that updated shape. The second phase of the build mechanism, where backend specific operations are defined, uses these shapes to ensure input placeholder, internal state variables, and results of API calls conform to the expected layouts and types.

The component algorithm builds a directed acyclic graph (DAG) of records for API methods via depth first search. Calls to API methods of root components are sources, and their final results (or rather the operations that produce them) are sinks.

A simplified graph assembly procedure is shown in Algorithm 2. The root component exposing the external interface and the input spaces for the external API are passed to the

Algorithm 2 Simplified component graph build procedure

```
Input: component root, input_spaces spaces
api = dict()
# Call all api methods once, check input dataflow types.
for method, record in root.api do
    input_op_records = list()
    # Create one input record per API input param.
    for param in record.input_args do
        # Check if space defined for param, fail otherwise.
        if param_name in input_spaces then
            input_op_records.append(Op(param.space))
        end if
    end for

    # Traverse graph from root for this method.
    out_ops_records = method(in_ops_records)
    # Register method with graph inputs and output ops.
    api[method] = [input_op_records, out_ops_records]

    # Tag last out-op-records.
    for op_record in record.out_op_columns[−1] do
        op_record.is_terminal_op = True
    end for
end for
return ComponentGraph(root, api)
```

component graph builder. This builder generates the backend-independent dataflow graph and the API by iterating over all API methods defined in the root component. For each method, a component graph operation is created for each of its parameters and looked up in the input graph (type checks omitted). The component graph is traversed by calling API method decorators which infer parameters and return values for each API called through the graph, and these are stored in records. Finally, the API method is registered in a registry which contains the input spaces and final output operations (nodes in the graph).

The component graph identifies the API of the graph and is used to identify missing input definitions or dependencies by failing as early as possible. It also enables graph-level device strategies as the graph can be rewritten before building backend operations (§3.4).

The component graph enables both push based (define-by-run) and pull based execution. Pull-based refers to the mechanisms whereby in frameworks like TensorFlow, evaluating a specific node in the graph is achieved by passing the node reference to the graph runtime. The runtime will then execute all dependent operations in the graph, which may require user inputs to provide values for input placeholder graph nodes. RLgraph hence needs to map API methods to graph nodes, i.e. which graph operations to call when users request execution of a specific API method. Session and operation handling are automated via a graph executor, and users only need to specify API methods they want to execute (§3.4). In push-based operation, input arguments are simply marshalled through API methods by

```
1  with tf.Graph().as_default(), \
2       tf.device(local_job_device + '/cpu'), \
3       pin_global_variables(global_variable_device):
4    tf.set_random_seed(FLAGS.seed)
5
6    # Create Queue and Agent on the learner.
7    with tf.device(shared_job_device):
8      queue = tf.FIFOQueue(1, dtypes, shapes, shared_name='buffer')
9      agent = Agent(len(action_set))
```

Listing 3.5: Example device management fragment from DeepMind open source implementation. Nested assignments create complicated dependencies.

function execution.

Execution of operations is restricted by design. Users cannot execute arbitrary operations within the underlying computation graph. To call a specific computation, it must be exposed as an API method. If arbitrary operations could be accessed from outside a graph function, they could also be used to extend the graph and create undesirable read-write dependencies between device boundaries of different components. Placeholders for graph extensions would also need to be defined manually based on the intermediate layout (i.e. the shape of the inputs to the operation within the graph function), which is generally not know in advance. Existing open source implementations particularly suffer from difficult-to-debug ad-hoc graph construction and re-wiring of operations (§3.1).

3.3.3 Building for static graphs

Next, I explain how end-to-end static computation graphs are created from the component graph representation. This section covers mechanisms to combine component subgraphs into larger graphs corresponding to complete algorithms. Conceptually, this is achieved by mapping input space definitions to the root component to dataflow input nodes in the corresponding graph backend. Input nodes with spaces and types derived from input spaces are then marshalled to individual components through the operation records in the intermediate representation.

The aim of this is to create connections between operations in graphs by invoking graph functions, with inputs traced from input nodes, and outputs being routed to the graph functions requiring them. Each graph function represents a subgraph and the build process iteratively connects them according to the dataflow dependencies given via the intermediate representations.

The difficulty lies in managing the corresponding internal state, local and global devices (for distributed execution (§3.4)) for each sub-task in the computation. While different programmatic approaches exist, TensorFlow for example utilises context managers to assign devices to states and computations. Listing 3.5 is taken from DeepMind's open source implementation of a distributed policy optimisation algorithm[2].

The example illustrates how implementations create I/O dependencies via ad-hoc assignments. In the code, lines 1-2 indicate that all following code is first assigned under a "local job device" CPU and a default graph. In line 3, another context manager is used to

[2] https://github.com/deepmind/scalable_agent/blob/master/experiment.py#L496

assign all global variables created under this context to a global device. Global variables are variables in the context of a distributed computation graph which are accessed by all nodes, and which are hosted on a specific device (in this case the learner). In line 7, another context manager is used to create a queue and agent under the shared device.

Algorithm logic is intertwined at every level with distributed execution and task-device assignment. As every operation is created under multiple context managers, a single misplaced operation can cause program execution to slow down an order of magnitude e.g. because local state is accidentally read from a remote device. Such problems are difficult to identify because they do not constitute bugs from the perspective of the runtime.

The purpose of RLgraph's build process is to avoid interspersing job and device assignments with dataflow definitions. The backend build mechanism incrementally creates a computation graph by combining component subgraphs. Each graph function in a component constitutes a sub-graph that can be built, device-assigned, and executed individually (e.g. for testing §3.5). Each component further can create in-graph state via variable creation which again can be separately device-assigned. I explain device strategies in §3.4.4. The build algorithm contains these phases:

1. Graph input placeholders are generated for the input arguments to all API methods at the root function, based on input space definitions.

2. Placeholders which represent nodes in a symbolic TensorFlow computation graph are stored as operations in the operation records created when building the component graph.

3. From this initial set of records containing operations, the algorithm iteratively builds the backend graph by marshalling operations through the component graph until no operations are left to process:

    3.1. Operations are processed once the components they connect to are input-complete. A component is input complete if none of its API methods are waiting on graph inputs.

    3.2. If a component is complete, its internal state is created by calling the generic state creation method with its dependent input spaces. Optionally a device assignment and variable sharing context is activated, otherwise the global default device is explicitly assigned.

    3.3. Graph functions as logical units of computations are called to create graph operations when all input operations are available. Graph functions are assigned to a default device or a device provided in a device map which enables manual or automatically optimised assignments of components to devices. Individual graph functions are called multiple times if the corresponding inputs are (potentially nested) container operations.

    3.4. Outputs of graph functions are again stored in records and, if required, re-nested or merged if the graph function is called multiple times with different arguments, each time creating a new static graph fragment.

Constructing static graphs in this manner avoids implicit device and variable sharing. Each iteration (i.e. processing a single operation record), the build algorithm asserts dataflow compatibility when connecting operations across components. This is achieved

by comparing the input and output spaces (i.e. dataflow prototypes) of the corresponding operations.

The resulting static graph is not logically different from ad-hoc implementations. Building the component graph and from it the static backend graph could be merged into a single procedure where graph operations are immediately created and marshalled through the components. This would create two disadvantages.

First, knowledge of component graph enables device strategies that require making copies of specific sub-graphs to split data across devices. The component graph can be modified before building backend operations to accommodate such strategies. Second, building the component graph is about an order of magnitude faster than building the backend graph since no backend operations are created. Separately building the component graph and creation of backend operations enables more fine-grained debugging and accelerates development. To summarise, this section used the example of TensorFlow to illustrate how RLgraph composes high-level RL components into end-to-end differentiable dataflow graphs.

3.3.4 Define-by-run component graphs

This section covers the alternative paradigm of dynamic graphs which implicitly arise from executing an imperative program. I discuss the tradeoffs of dynamic and static graphs and explain how RLgraph supports both through its component graph. In define-by-run or eager mode, no static graph is built ahead of execution (a static representation may be inferred *from* execution, §3.6.1).

Define-by-run frameworks such as PyTorch enable fast development cycles because they support imperative evaluation of numerical operations with built-in automatic differentiation. This allows developers to utilise mature debugging tool-chains from host languages. They also facilitate dynamic network architectures where the layer structure can be modified between each call.

In supervised learning scenarios with large batch sizes, training time is dominated by computing updates on accelerators [ARR+16]. CPUs manage I/O by loading training data from input sources (e.g. distributed file systems) and transferring it to device memory. RL workloads in contrast are characterised by CPU-intensive environment evaluation, processing of sample trajectories, and communication between workers and learners. RL implementations in design-by-run frameworks is hence complicated by the fact that there is no natural boundary between data and computation logic. In static graphs, each operation and its inputs must be intentionally placed within or outside the graph, thus allowing specific tradeoffs. For example, preprocessing a short state trajectory may be faster in a host-language than in a graph runtime like TensorFlow due to invocation overhead.

This fragmented dataflow (where fragmentation refers to execution context switches) does not allow for the application of end-to-end optimisations such as device placement [MGP+18] or automated graph rewrites for hardware-specific compilation [CMJ+18] (§3.6). Moreover, distributed execution and scheduling become more difficult as schedulers have no access to fine-granular task details (as in an end-to-end graph) but must execute programs as black-box tasks (e.g. Ray, [MNW+17], §3.4.5).

Define-by-run frameworks nonetheless enjoy increasingly popularity among developers especially in exploratory implementations where performance is secondary. TensorFlow similarly has introduced an eager execution mode due to the difficulties around designing

and debugging static graphs. Framework providers must identify abstractions which combine the fast prototyping and debugging of define-by-run mode with scalable execution mechanisms and optimisations. To resolve this tension, framework developers are exploring new mechanisms to trace imperative control flow for automated graph generation (§3.6.1).

RLgraph can provide define-by-run execution with the following considerations:

- The backend build process generates static graph operations from a set of input spaces and connects them through the component graph. However, from the perspective of the build process, there is no difference between static operations and define-by-run operations, which are simply tensors objects holding numerical values.
- Instead of generating input placeholders for a static graph, the build process samples from the input space. Graph functions and API methods do not return static tensor operations but immediate computation results.
- Any static component graph can also be executed as a define-by-run graph. This follows because allowed control flow in static graphs is a restricted sub-set of the control flow available in imperative languages [YAB+18].
- Not all define-by-run graphs can be built as static graphs if they use language features not supported by static runtimes.

Prior RL frameworks or high level deep learning APIs do not support both define-by-run and static graph frameworks through a single abstraction. They are designed for one framework or paradigm, or only support different paradigms on an entire algorithm level. For example, Ray RLlib provides parallelisation mechanisms on the Ray execution engine which can parallelise both e.g. TensorFlow and PyTorch algorithms. Implementations are separate so algorithmic logic must be entirely reimplemented per backend.

RLgraph supports end-to-end static graphs and define-by-run mode on a component level. Dataflow between algorithms is shared through all API methods. Only graph functions performing backend-specific computations need to implement framework-specific logic. In the future, I expect automatic graph generation to be able to reduce backend-specific code further.

I implemented a define-by-run graph building and execution mode for RLgraph supporting PyTorch as a popular define-by-run framework for three reasons. First, in define-by-run model, users can realise complex control flow which may be difficult to address in static graphs (e.g. recursion). Second, they can also choose define-by-run mode to implement features like dynamic neural networks. While the component graph is always static, graph functions can express dynamic behaviour. Third, even with tools for automatic graph generation from imperative code, algorithms need to be structured in a way that allows to leverage graph generation to connect computation fragments into end-to-end graphs.

Next, I explain how RLgraph executes algorithms across deep learning frameworks and execution engines.

3.4 Execution

The prior sections have covered the generation of differentiable dataflow graphs from high-level components. This section discusses how different execution scenarios, from

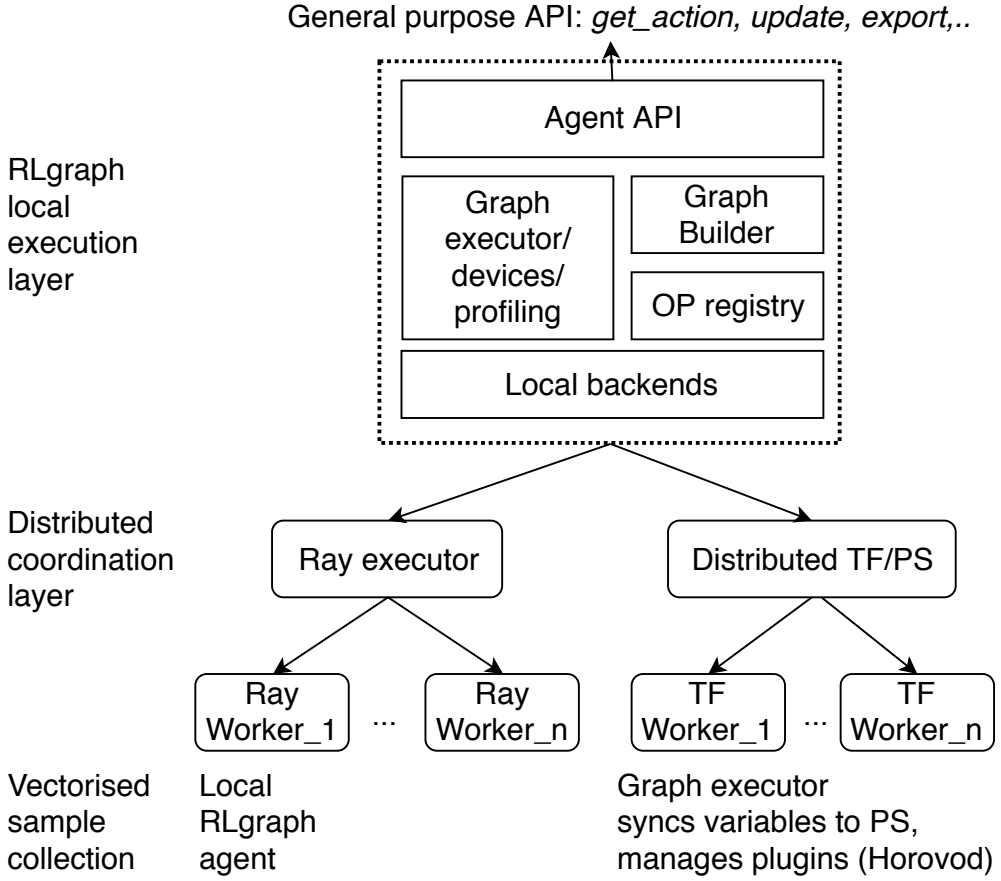


Figure 3.6: RLgraph execution architecture overview.

single-threaded simulators to large scale distributed engines, can be accommodated from the execution-agnostic representation.

RLgraph's execution abstraction is a *graph executor*. It serves as a bridge between the API defined in an agent and the computation graph/operations generated by the build process. Figure 3.6 provides an overview of RLgraph's execution stack. Users of pre-existing implementations interact with the high-level API (§3.4.1). The API passes requests to the graph executor which builds up inputs for the corresponding operations from the op registry, executes operations, and returns results (§3.4.2). The graph executor also manages scaffolding for each backend and can interleave build stages with device management (§3.4.4) and optional optimisations (§3.6). Execution on distributed execution engines is handled via dedicated executors which coordinate workers (§3.4.5).

3.4.1 Agent API

This section covers the agent API which I use for backend-independent distributed coordination. I first reiterate the main challenges for algorithm execution.

- When designing algorithms, the number of parallel environments, their step-times, and evaluation costs in later deployments are not known. For example, an algorithm could collect individual episode trajectories within a single thread, or produce a large set of parallel trajectories with different termination state and trajectory length at

unknown intervals. Pre- and post-processing tasks might be performed by the same process, or could be distributed.

- Execution-agnostic implementation hence means that dataflow logic is modularised so that all steps (i) can be invoked individually, (ii) can accommodate all possible trajectory layouts, (iii) and can be executed from internal stored graph state or purely functional by passing in all necessary information. This is needed to accommodate different synchronisation schemes which store state in-graph, or in external caches or databases.

The agent API describes a set of dataflow definitions which all algorithms must implement to utilise execution coordination. The API must serve the following purposes:

1. Provide a simple to use high-level RL interface for practitioners,
2. Allow rapid design exploration through component combination,
3. Support heterogeneous single-node and distributed execution with customised concurrency, parallelism, and control flow.

Listing 3.6 shows the main methods in the API. Agents are instantiated by providing the agent object with a component configuration containing a (nested) list of components and their configurations. The internal state of each component is fully described by configuration and input space definitions to avoids reproducibility problems.

```
abstract class rlgraph.agent:
  # Builds graph.
  def build(options=None)

  # Get predictions for states, control behaviour with flags.
  def get_actions(states, explore=True, apply_preprocessing=True)

  # Update from buffer or external data.
  def update(batch=None, sequence_indices=None, apply_postprocessing=True)

  # Write trajectories to buffer.
  def observe(states, actions, rewards, terminals, env_id)

  # Read component/model state.
  def get_weights(components=None)

  # Write model state.
  def set_weights(weights)

  # Reads internal state of the graph.
  def read_variables(variables)

  # Import model parameters.
  def import_model(path)

  # Serialise model parameters.
```

```
27  def export_model(path)
```

Listing 3.6: High level agent API.

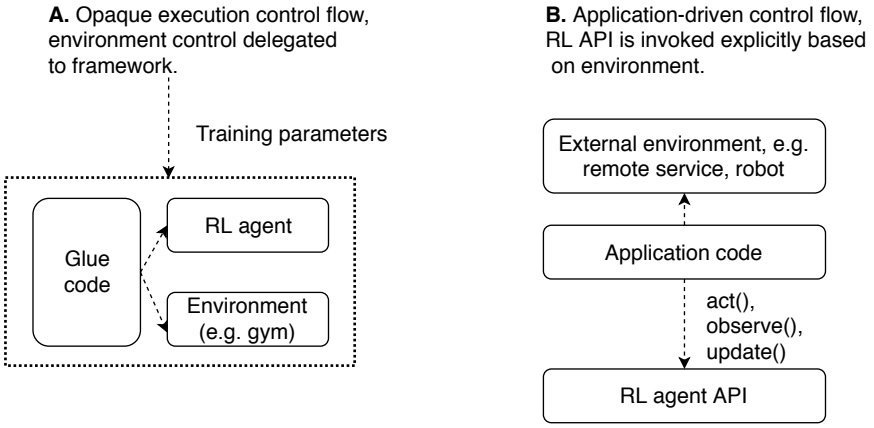


Figure 3.7: Agent-driven and environment driven execution modes.

A usability limitation of many open source RL libraries is their control flow assumption. The popularity of OpenAI gym has led implementations to assume a reactive environment driven by a single-threaded training loop (Figure 3.7). This agent-driven (environment waits on agent updates) design has been adopted for instance by OpenAI baselines [DHK+17], Nervana Coach [CLN17], and Ray RLlib [LLN+18].

Agent-driven execution is undesirable for domains where the environment cannot be provided via a convenient simulation handle. In environment-driven execution, an application invokes the agent API to retrieve actions and perform updates according to unknown application requirements. For example, one research user of RLgraph utilises RL for an interactive learning application where hundreds of users interact concurrently at a random schedule. The application needs to perform updates when a specified number of samples has been reached across all users. The update must process an unordered collection of trajectory fragments from terminal and non-terminal episodes. Such semantics are not supported by existing libraries which assume a specific episode and update schedule. For example, none of the libraries I surveyed support updating from concurrent episode trajectories without requiring the user to perform parts of the algorithm's post-processing in their application.

I designed the RLgraph API to be agnostic with regard to execution semantics. This means the API can both be consumed by common gym simulators executors, distributed, parallel and asynchronous executors for large scale simulations (§3.4.5), and external applications with custom semantics. This requires the API to include the following functionality:

- Pre-processing of states (e.g. down-sampling of images) and post-processing of trajectories (e.g. computing value estimates) must be available separately. In distributed execution, it is often convenient to distribute parts of pre- and post-processing to environment worker threads, so learner tasks are not burdened by them. In contrast, users from external applications may not be aware of pre- and post-processing workflow. They must be able to rely on them being performed correctly as part of the update.

- All pre- and post-processing must be able to operate on batches of trajectories invariant to the current state of each trajectory (terminal or non-terminal). This is achieved by using an additional array of boolean indices when updating from a batch of trajectories. A "true" value indicates the end of an episode fragment in an environment, and for the case of a single environment, this index is equivalent to the terminals of the episode. Without such an index, non-terminal episode fragments of multiple environments could not be distinguished.

- The internal state of component subgraphs and model parameters must be transparently readable and writable. Distributed execution engines employ a multitude of coordination schemes ranging from requiring users to implement manual synchronisation to engine-specific global state sharing (e.g. distributed TensorFlow [AIM17]).

3.4.2 Local execution

Local execution refers to the way requests to the agent API are resolved within a single process by invoking a graph executor. The graph executor evaluates if input arguments correspond to the prototype dataflow definitions given via input spaces. Supporting different machine learning frameworks requires implementing a graph executor with two key methods to build and execute graphs. A separate graph executor per framework is necessary due to the differences in how machine learning frameworks provide functionality such as device assignment, state management, and operation execution.

I exemplify this via the TensorFlow and PyTorch executors I implemented in my prototype. Building component graphs to the TensorFlow backend requires initialising a graph, a session, optional scaffolding for profiling, checkpointing and result statistics, and additional build arguments in case parts of the graph are meant to be declared global shared state. Executing an API method requires translating its output to the corresponding TensorFlow graph operations before invoking a session.

The PyTorch executor does not interfere with the build process as there is no session or graph scaffolding. Data can be moved across devices with ad-hoc calls on a tensor object during execution with corresponding runtime penalties for memory allocation and data transfer. The main consideration in define-by-run execution is the automated tracking of gradients when converting input arguments to tensor objects which are subject to gradient accumulation. The PyTorch executor executes API methods by retrieving not graph operations but the function object which is then called with tensor-converted input arguments. Gradients are attached if the method requires computing an update. Sub-sequent calls to dependent API methods and graph functions across components are subject to define-by-run versions of dataflow manipulation.

Evaluating decorators between API methods introduces some over-head in define-by-run mode. Recall that components can also be flexibly combined due to dataflow modifications when connecting them, realised via API methods. This means in define-by-run mode, every graph function evaluation begins and ends with evaluating potential layout transformations. Otherwise, this functionality would need to be hard-coded into components which hinders reusability. As I show in the evaluation, execution time for non-trivial model sizes is dominated by neural network operations.

The component build mechanism facilitates optimisations. Consider any number of n evaluations of dataflow structure evaluations $g : \mathbb{R}^n \to \mathbb{R}^n$ of a method $f_i : \mathbb{R}^n \to \mathbb{R}^n$ with

$x \in R^n$:

$$y = g(f_1(g(f_2(g(f_n(x))))). \tag{3.1}$$

The structure of this call-chain is known ahead of execution due the intermediate component graph. This means that if any sub-subsequence $f_i, f_{i+1} \cdots, f_n$ does not require dataflow manipulations, the call-chain can be contracted to a single API evaluation by removing all intermediate edges between API function evaluation and methods within the sub-sequence:

$$y = g(f_1(f_2(f_n(x))). \tag{3.2}$$

A typical example of this is any stack-like structure (e.g. neural network layers). If inference performance is critical to the application, a tracing mechanism similar to the build mechanism can be used to extract underlying graph structure (§3.6.1).

3.4.3 Implementing algorithms

When implementing algorithms, agent API, executor and graph are utilised in the following structure:

1. Developers implement the agent API, create components such as neural networks, optimisers and probability distributions, and define dataflow between components through API methods attached to a root component (Figure 3.8). Users can also implement other ad-hoc APIs, RLgraph for example also has a light-weight testing interface used for generating sub-graph tests (§3.5).

2. The root component is passed to a graph executor which is instantiated based on the backend framework configured. The graph executor creates a graph builder and invokes the different build phases.

3. Calls to the agent API are routed to the graph executor which in turn requests operations or methods from the API registry created during graph building.

4. There is no other interaction between users and the model than through registered API methods. All backend-specific functionality and dataflow modifications are centralised into executors and API method/graph function decorators.

Crucially, the dataflow created by users exploring new algorithms is independent of the specific graph executor/backend used.

3.4.4 Device management

In the context of graph construction (§3.3), I discussed how device semantics in deep learning frameworks can cause implementations to tightly couple execution instructions with algorithmic logic.

Deep learning frameworks frequently provide built-in mechanisms for device strategies. These abstractions simplify distributed training for supervised learning with a static data-set, but alone are insufficient for reinforcement learning. Hierarchical pre- and post-processing of sample trajectories and sample transfer between workers and learners (as opposed to importing a dataset once from a distributed file system) necessitate fine-grained device assignments to optimise throughput.

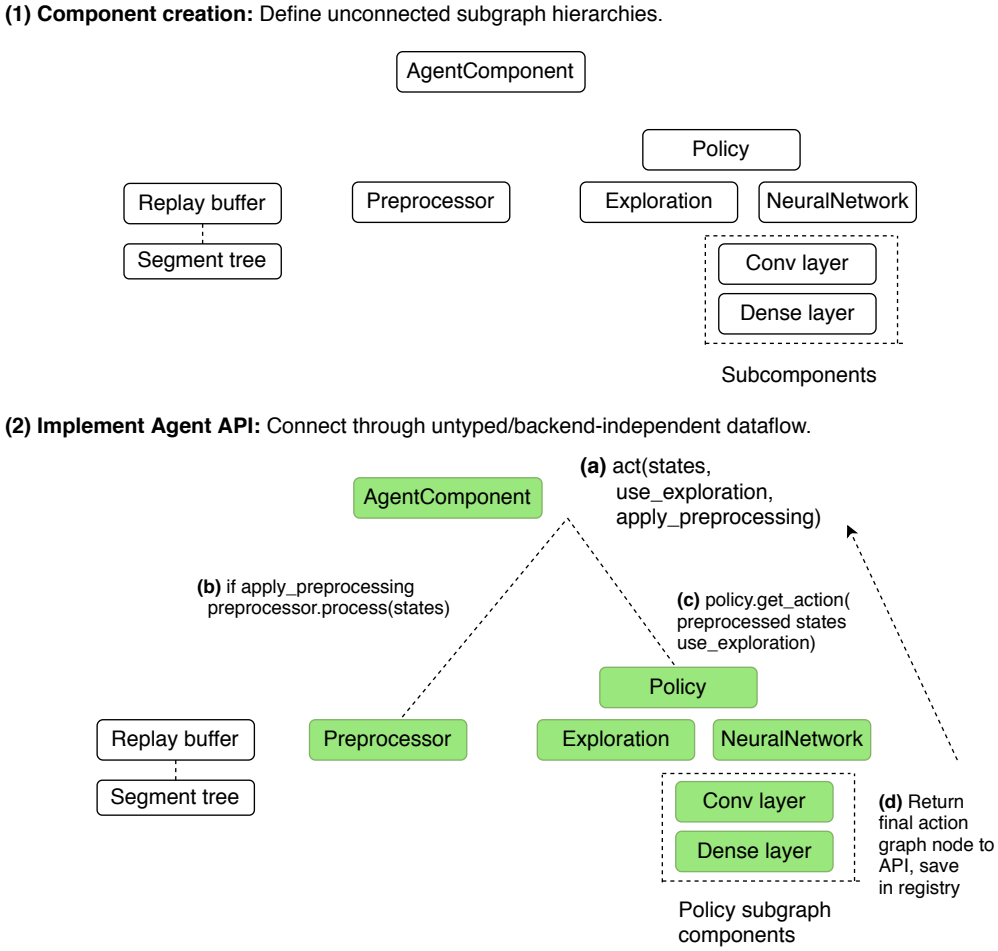


Figure 3.8: Implementation example. (1) Users create subcomponents of an agent component. (2) The agent API is implemented by connecting components. (3) Input type definitions restrict allowed dataflow for the static graph constructed during the build.

Device management in RL has not found much attention in the literature beyond show-casing how specific algorithm implementations can achieve high GPU utilisation [CMC17, ESM+18, LLN+18, HQB+18]. Prior research has focused on how to structure distributed dataflow to improve device utilisation, but not how to provide building blocks that enable new device strategies.

In consequence, existing RL frameworks do not offer modular device strategies. In RLgraph, devices can be managed both on graph level and component level:

1. **Component-level assignments.** Component level assignment allows to create internal state and computations for each component subgraph under a device context.

2. **Graph-level assignments.** Graph-level strategies leverage the fact that the component meta-graph can be modified before building. For example, to distribute sample trajectories across devices, component-sub-graphs can be copied and sample data

split across graph copies. Further, it can be beneficial to declare some sub-graphs as globally (i.e. cluster-wide) shared state in parameter-server architectures [LAP+14].

The need for both component and graph-level device management can be exemplified through distributed policy optimisation algorithms such as IMPALA [ESM+18]. IMPALA performs learning by letting a set of actors perform environment trajectory rollouts. Their samples are inserted into a globally shared blocking queue. The learner dequeues rollouts from the queue, post-processes them, and moves them to a staging-area. In parallel, a previous batch is removed from the staging area to compute the update. Some value estimation heuristics cannot be effectively parallelised on accelerators because they require a linear scan of sample trajectories.

Figure 3.9 visualises dataflow and device assignments in RLgraph's IMPALA implementation using TensorFlow's TensorBoard tool. When RLgraph builds its component graph to a TensorFlow graph, component names are used to generate *scoped* operation and device assignments. Here, device assignments are highlighted in color where green components are assigned to a GPU and blue components to CPUs. Blocks with both colours indicate that the component has sub-components on another device, e.g. the loss function computing importance weights on the CPU and back-propagation on the GPU.

Visualisations can be used as an interactive dataflow debugging tool if the graph is organised into logically meaningful components. This is not the case for ad-hoc implementation styles. Figures 3.10 and 3.11 are visualisations generated for DeepMind's open source implementation of IMPALA by the algorithm authors. Computations and device assignments are not systematically grouped in the graph. Specifically, the logical grouping in the implementation (where code is organised into functions and classes) does not automatically induce logical grouping in graph construction.

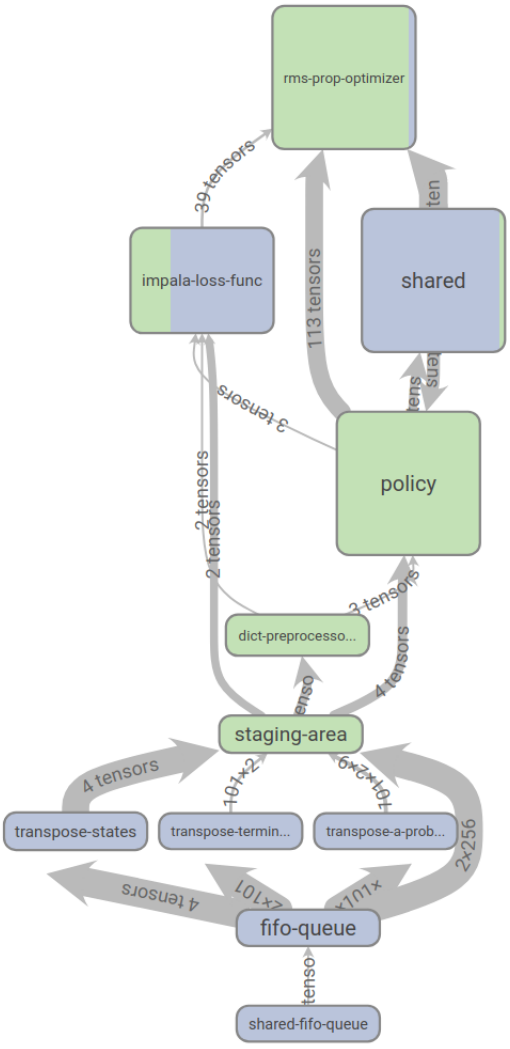


Figure 3.9: TensorBoard visualisation of RLgraph's IMPALA learner. All operations and state variables are organised logically in component subgraphs.

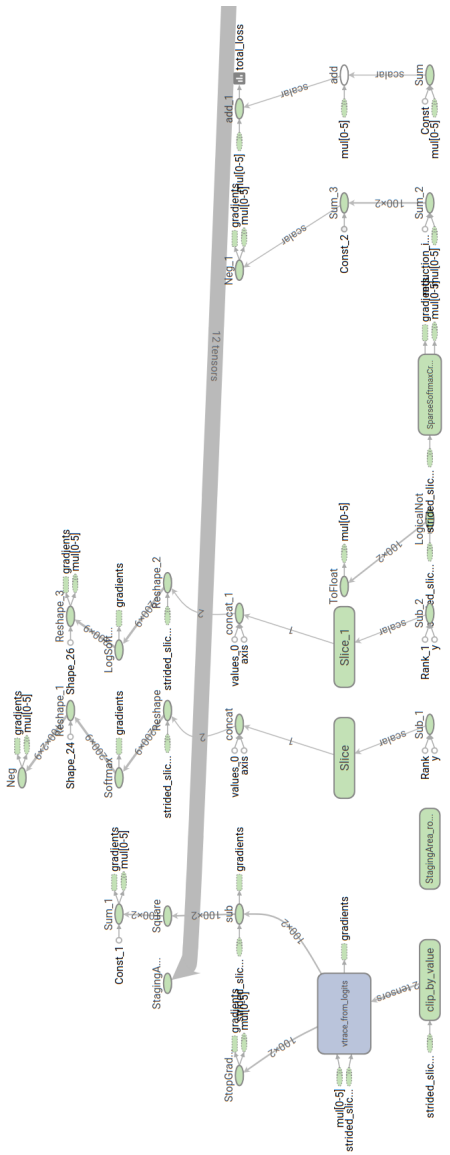


Figure 3.10: TensorBoard visualisation of DeepMind's IMPALA learner (left half of the graph).

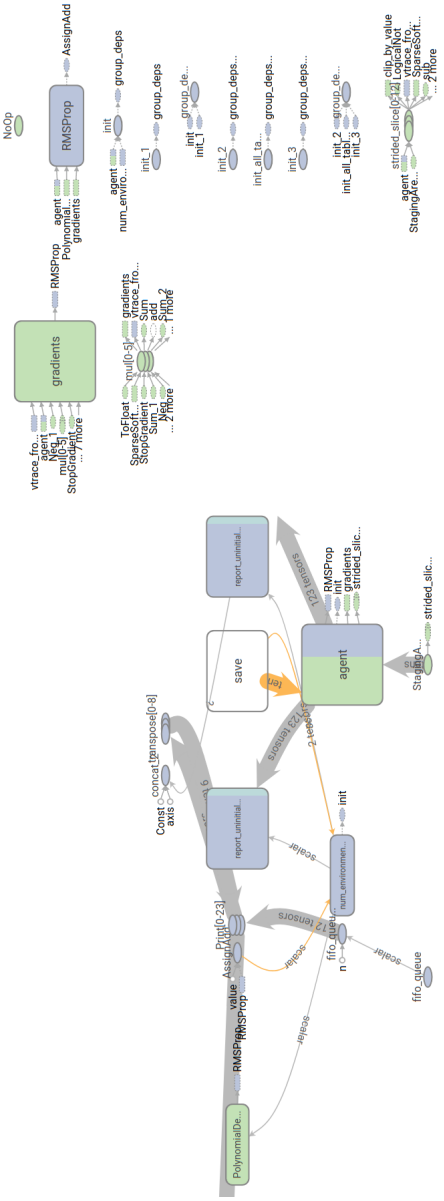


Figure 3.11: TensorBoard visualisation of DeepMind's IMPALA learner (right half of the graph).

3.4.5 Distributed execution engines

Closely related to device management is distributed execution. The emergence of deep learning workloads has given rise to a multitude of distributed learning schemes for synchronous and asynchronous data and model-parallel training [DCM+12, LAP+14, RRWN11, PLT+16]. Frameworks increasingly provide higher level distributed abstractions which do not require developers to explicitly implement synchronisation. These abstractions however only support well-defined workflows such as supervised training where practitioners have converged to a standard set of methods, e.g. data-parallel synchronous stochastic gradient descent.

RL workloads are not usually supported by such higher level abstractions due to their heterogeneous processing and communication patterns (§3.1). This poses two challenges for RL frameworks:

- **Choice of execution engine.** Novel machine learning workloads have given rise to a new generation of execution engines. Improving distributed or decentralised parameter exchange is a growing area of research. As these workloads (e.g. generative adversarial models [GPM+14], multi-agent communication [FAdFW16, REH+18, FdWF+18]) require new communication semantics, there is no one-size-fits-all solution to accommodate changing requirements.

- **Separating algorithm logic from execution.** As execution semantics may vary from algorithm to algorithm, distributed coordination should be independent from algorithm logic.

Existing implementations opt to reduce complexity by narrowing flexibility. Reference implementation collections often outright do not include generic distribution mechanisms. Consider OpenAI baselines [DHK+17] as a widely implementation collection provided by the original authors of some algorithms (e.g. PPO [SWD+17]). Some algorithms implicitly utilise a custom MPI-based distributed synchronous mechanism where sample workers each interact with an environment, and a single learner collects their samples, updates, and synchronises new weights to the workers. This mechanism is hard-coded and not configurable. Other frameworks such as Ray RLlib [LLN+18] provide distributed abstractions which can be combined freely with algorithm logic at the cost of locking users into executing training on the Ray platform [MNW+17].

RLgraph isolates distributed coordination from component graph logic to support different distributed execution paradigms. From RLgraph's perspective, there are two types of distribution mechanisms:

1. **External API consumers.** External consumers realise distributed mechanisms purely by interacting with RLgraph's API to read and write agent state, insert into buffers, or call updates from internal buffers or external trajectory data.

2. **Internal graph-aware mechanisms.** Internal mechanisms may be framework-specific distributed engines which rely on implementations using special operators or program design.

Examples of the first mechanism are custom communication strategies implemented via e.g. MPI or distributed execution engines such as Ray [MNW+17]. The second class of distributed execution method concerns mechanisms such as the distributed TensorFlow

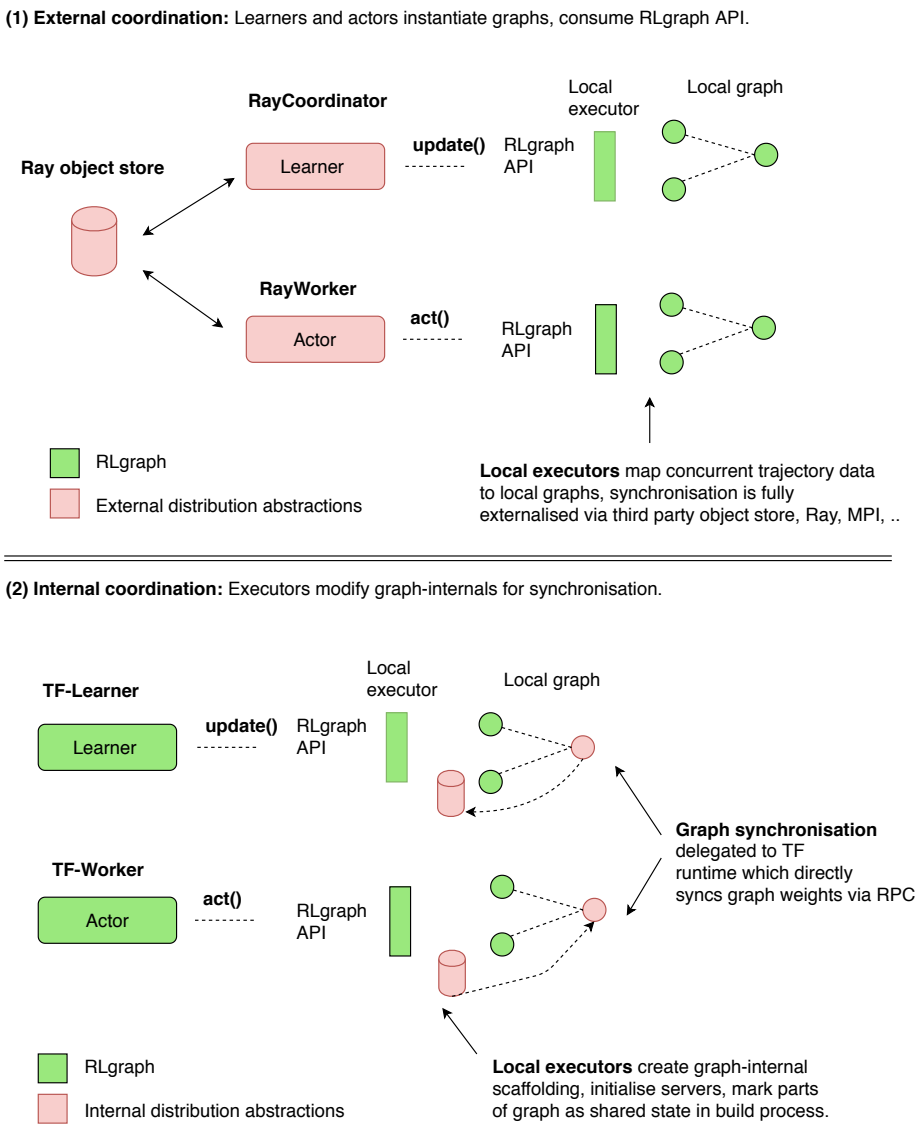


Figure 3.12: Graph-internal and graph-external distributed coordination.

runtime [ABC+16, YAB+18]. Distributed TF can schedule an *individual* graph across device and host boundaries. End-to-end graph scheduling is theoretically appealing because it enables both compile-time optimisations (§3.6.2) and fine-grained scheduling decisions at the level of individual graph operations.

This comes at the cost of a more complicated programming model. Distributed TensorFlow is most commonly used with a between-graph replication model. Instead of centralising control via an executor invoking tasks of remote workers, users must manually create client processes for different tasks and explicitly create in-graph synchronisation mechanisms. For example, the authors of IMPALA [ESM+18] describe workers communicating with a learner via a globally shared queue. The queue is part of both the learner's and worker's graph.

Figure 3.12 illustrates the difference between internal and external coordination. The main consideration for enabling flexible distribution strategies is to separate these abstractions from graph construction through the executor. Since the graph executor knows each

high-level component as a named subgraph, it can straightforwardly interpret high-level build instructions such as "mark the policy network as globally shared". This is achieved by interleaving the build process with execution-initialisation stages which ensure the necessary modifications (job assignment, state synchronisation) are applied to the relevant component subgraph. Viewing and building algorithm components through logically organised and named subgraphs is hence the central abstraction for flexible execution. Again, this is in contrast to making such modifications directly in the code which defines the graph, thus prohibiting modifications at build time.

To exemplify this approach, I implemented a Ray execution package interacting with RLgraph's external API. The Ray package consists of two abstractions, *Ray executors* and *Ray workers*. An executor implements a distributed coordination loop which schedules actor tasks and collects their results. Ray workers are actor classes which can execute any desired task but most commonly are used to interact with one ore more environment simulators. For example, a synchronous batch executor trains according to the following steps:

- The executor instantiates a local agent on the master node by building a component graph towards the desired backend. It also creates workers as actor objects via Ray.
- Until the specified number of training steps is completed, a step function is called:
  - The executor fetches weights from the local agent via the agent API.
  - Weights are synced to workers by calling a Ray task fetching weights from the distributed shared Ray memory to write the worker graphs, again using the agent API.
  - Next, the executor schedules sampling tasks for each worker. Workers interact with one or more environment copies to collect sample trajectories and post-process data.
  - The executor waits on task completion, merges trajectories from sample workers into a single batch, and calls the agent API to perform updates.

I evaluate this design in §4.3 against Ray's native reinforcement learning library RLlib. Executors and workers closely mirror RLlib's optimiser abstraction with the crucial difference that the agent API and build process are entirely separate from Ray. In contrast, RLlib's learning, sample collection and processing mechanisms are tightly coupled with Ray task code. Next, I explain how graphs can be incrementally built and tested using the test-generation interface.

3.5 Incremental building and sub-graph testing

Brittle RL algorithms necessitate new approaches to testing. This endeavour is complicated by multiple sources of non-determinism (recently investigated by Nagarajan [NWS18b, NWS18a]) :

- **Weight initialisation.** Neural network weights representing policies or value function parameters are commonly initialised from Normal distributions [GB10].

- **Stochastic policies.** Policy gradient algorithms act by sampling from a policy distribution. Value-based algorithms induce random exploration via heuristics like ϵ-exploration.
- **Stochastic environments.** Problem environments may exhibit random state transitions.
- **Stochastic gradient estimation.** Mini-batch stochastic gradient descent updates decorrelate trajectories by sampling random batches from buffers.
- **Non-deterministic devices.** Accelerators may not default to deterministic execution. For example, NVIDIA hardware does not guarantee the same bit-wise results for some neural network kernel implementations[3].

The difficulties of analysing non-deterministic behaviour are exacerbated by unclear sources of algorithmic performance (§3.1). Novel results can also be expensive or otherwise impractical (e.g. requiring specific hardware) to reproduce. Consider IMPALA, a recent distributed policy optimisation algorithm [ESM+18]. Reproducing learning capabilities on the benchmark tasks in the related publication requires running the algorithm for up to 10 billion frames for multi-task learning. This necessitates a cluster of worker nodes and costly accelerators to be run for hours to days. On public cloud services, this can translate to several thousand dollars in cost to run a single experiment across different random initialisations. This does not account for hyper-parameter tuning and debugging of new implementations (§6.4).

Resource requirements can cause insurmountable difficulties when implementing and benchmarking algorithms. Researchers may not have access to the resources needed to perform in-depth ablations on new techniques. Published research in turn is then difficult to evaluate due to this missing analysis, creating a vicious cycle of uncertainty around algorithms. In Chapter 6, I illustrate the true cost of reproducing and evaluating published results to assess learning capabilities. This "implementation risk" means practitioners often have to rely on open source implementations.

Systems research has not caught up with these developments. Existing RL libraries (§3.1) do not provide systematic incremental testing facilities. Accompanying benchmarks results can give evidence of performance at a specific point in time. The brittleness of implementations however means small modifications can lead to drastic loss in training performance going unnoticed (§4.3).

I argue that the lack of systematic testing is largely due to lack of systematic algorithm construction. Coarse-grained implementations mix and combine neural network construction, update logic, state management, and execution semantics. Instantiating specific modules is not possible without code modifications. Manually determining correct inputs for intermediate operations for testing purposes is tedious. RLgraph's abstractions lend themselves to modular, incremental testing:

- There is no conceptual difference between building a single component, a composition of components, or an entire algorithm. Execution scaffolding is always generated via graph builder and graph executor.

[3] `https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/%index.html#reproducibility`, accessed 25.08.2019

- Input and internal state of each component are based on well-defined spaces. Inputs for subcomponents can be trivially generated by sampling from the spaces used to build the component.

Tests are generated via a testing utility which builds any given component and its subcomponent.

```
# State has 64 floats with a batch and time dimension.
state_space = FloatBox(shape=(64,), add_batch_rank=True, add_time_rank=True)

# Action is container space: 1 discrete, 1 continuous action.
action_space = Dict(discrete=IntBox(), cont=FloatBox(), add_batch_rank=True)

# Create recurrent neural network from layer specification.
network = NeuralNetwork.from_spec([
  {"type": "dense", "units": 128, "activation": "linear"},
  {"type": "lstm", "units": 64, "activation": "linear"}
])

# Network is a sub-component of policy.
policy = Policy(network, action_space)

# Construct sub graph from spaces, generate API and scaffolding.
test = ComponentTest(policy, dict(nn_input=state_space),
                     action_space=action_space))

# Execute with sampled inputs from the input space.
action = test.test(policy.get_action, state_space.sample())
```

Listing 3.7: Testing sub-graphs from arbitrary spaces.

Listing 3.7 illustrates how a policy object, i.e. an object representing $\pi(s|\theta)$ via a neural network with parameters θ and action distributions, is constructed from input spaces specifications and sub-components. A state space with a batch and time rank is defined (indicating data along the trajectory timestep dimension and the batch dimension), and an action space container containing one discrete and one continuous output. Next, a neural network is specified from a list of layers and passed to a policy object which includes it as a sub-component. The policy component generates a set of output distributions, e.g. a Categorical distribution for the discrete action, and a Normal distribution for the continuous action. The policy component and its subcomponents are built as a component graph, and its API methods can now be tested with inputs generated from the defined spaces. Listing 3.7 exemplifies a sub-graph integration test where several features (recurrent policies, container action spaces) are tested together.

RLgraph implementations are hence tested along the following dimensions:

- **Component unit tests.** Each component's API methods are tested separately.
- **Dataflow integration tests.** The build process for the specific combination of components in a given algorithm is tested to detect problems when changing dataflow.
- **Incremental learning tests.** Training performance is evaluated on a series of small test environments (e.g. grid-worlds, controlling a cart-pole). This allows to incrementally evaluate sensitivity to learning heuristics.

Some learning failures cannot be detected through this scheme. Consider the popular Deep Q-Networks (DQN [MKS+15]). While many variants exist, a set of commonly used heuristics comprises loss-prioritised replay sampling [SQAS15], double Q-learning [HGS16],

n-step learning, and dueling network architectures to decompose $Q(s, a)$ into $V(s)$ and $A(s, a)$ [WSH+16].

Notions of environment difficulty in relation to a specific heuristic are not well-defined. A heuristic causing learning to fail (in terms of reaching pre-defined rewards thresholds) can be a sign of a faulty implementation if the base algorithm works. Conversely, an agent solving a task with and without a heuristic such as prioritised importance sampling is not evidence of the heuristic being implemented correctly. I illustrate this by running the

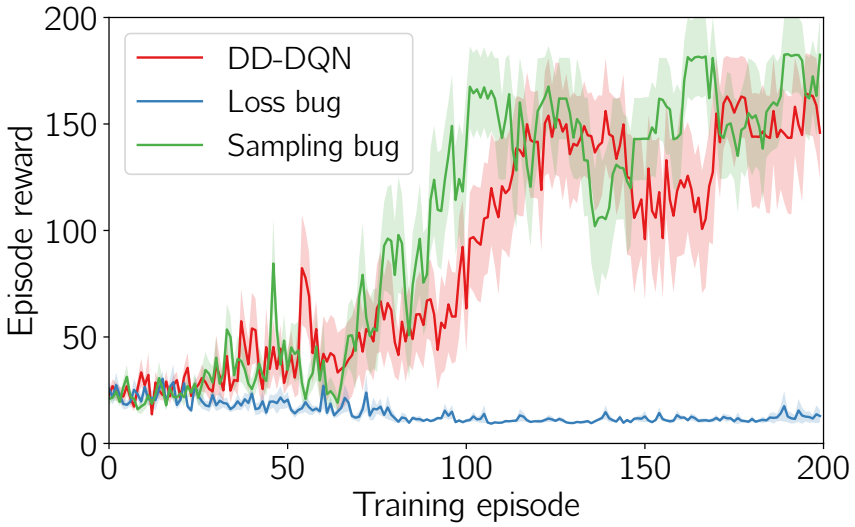


Figure 3.13: DQN training results with bug-free implementation, loss-fault injection, and sampling fault-injection.

cart-pole balancing problem environment using a DQN architecture and RLgraph's gym execution package. Figure 3.13 shows a double-dueling DQN (*DD-DQN*) with prioritised replay (i.e. canonical heuristics), and two faulty variants. In *Loss bug*, I injected an error where value estimation of subsequent states s_{T+1} from terminal states s_T is not set to zero, thus resulting in critical overestimation of Q-values in the update. In *Sampling bug*, I introduced a separate sign-bug in selecting samples by priority resulting in using the least relevant samples for updates instead of the most relevant.

I ran each variant with 10 different random seeds and plot the mean episode reward with a 1 standard deviation confidence interval (shaded area). The loss injection bug causes the agent to fail to improve its reward. The sampling bug's learning curve is not meaningfully distinguishable from the correct (to the best of my knowledge) one.

This behaviour is exemplary for the difficulties in evaluating RL algorithms in new environments. The impact of any learning heuristic is only empirically demonstrable. Consequently, practitioners cannot rely on selective benchmark results as indicators for correct implementations. In §4.3.3, I show how learning on existing frameworks can degrade between versions. In summary, RL implementations require fine-grained incremental testing mechanism which RLgraph facilitates through its subgraph instantiation from prototype dataflow.

3.6 Graph optimisations

In the prior sections, I have explained how RLgraph's modularisation helps design re-usable and testable implementations. Beyond programmability, algorithm implementations may benefit from a range of optimisations. In this section, I discuss how some emerging methods could be implemented into RLgraph.

3.6.1 Automated graph generation

To address difficulties in implementing end-to-end static graphs in imperative programming languages, the automated generation of graph fragments from imperative code has gained increasing attention. Specifically, graph functions in RLgraph using a static graph backend and which require control flow need to use special operators for branching and loops. Using custom control flow operators with restricted semantics creates large and unintuitive implementations.

RLgraph simplifies building differentiable graphs through its modularisation. Graph functions must always return an operation as the build process needs to connect inputs and outputs of graph function to an end-to-end graph. Within graph functions, backend code cannot be mixed, i.e. there can be no host-language control flow interspersed with tensor operations since this would partition the graph.

To resolve the tension between fast prototyping in imperative control-flow and the performance benefits enjoyed by static graphs, framework providers are considering new approaches towards automated graph generation. For example, in TensorFlow, a novel framework called AutoGraph [MDW+18] converts host-language control flow to static graphs by converting Python code into an intermediate representation and analysing both control flow and stateful dependencies (writing and reading graph state). Similar to device and job assignment, graph conversion is provided via context managers.

Graph generation could be integrated into the build process so graph functions with Python control flow are auto-converted to static-graph fragments. RLgraph could also exploit graph generation to unify backend-dependent code in graph functions.

```
# (1) Python-control flow  at graph construction time
# Control flow is not in graph, only one graph fragment constructed.
@rlgraph.graph_fn
def _graph_fn_sum(x, norm=True):
  # norm is a Python bool
  if norm is True:
    return tf.abs(x)
  else:
    return x

# (2) Condition depends on boolean tensor norm.
# Use special control flow operator.
# Both graph fragments are created.
@rlgraph.graph_fn
def _graph_fn_sum(x, norm):
  # norm is a tensor
   return tf.cond(
```

```
18     pred=norm,
19     true_fn: lambda: tf.abs(x),
20     false_fn: lambda: x
21    )
22
23 # (3) Graph function with autograph conversion.
24 # Graph functions invoke auto-graph,
25 @rlgraph.graph_fn(graph_generation=True)
26 def _graph_fn_sum(x, norm):
27    # norm is a tensor used in a Python conditional via AutoGraph
28    if norm is True:
29     return abs(x)
30   else:
31     return x
```

Listing 3.8: Control flow paradigms in graph functions.

Listing 3.8 illustrates different modes of conditional evaluation. First, native control flow can be used to statically decide which graph fragment to construct depending on a boolean flag (1). The flag is not evaluated at runtime since only one of the fragments is included in the static graph. Next, the boolean flag is a tensor evaluated at every call to decide which branch to execute, and a special conditional operator must be used to create graph-branching (or loops). This means control-flow cannot be shared between backends. In (3), graph generation could be used which converts the native Python control flow evaluating a tensor-value in the "if" condition to the graph representation in (2). PyTorch as a define-by-run framework can evaluate native Python control flow but needs similar tracing mechanisms to export programs into deployable serialisation formats.

3.6.2 Relationship to compilers

The proliferation of hardware accelerators and new deployment scenarios (e.g. mobile, embedded, edge devices) has also given rise to new static compilation and runtime optimisation tooling. They address two questions:

1. How can a given symbolic graph definition be optimised for a target hardware backend?

2. For a compiled graph, how should workload dependent execution properties be modified at runtime to improve scheduling?

For example, TVM [CMJ+18] is a compiler architecture for deep learning which auto-generates hardware-optimised kernels from high-level graph definitions. Runtime optimisations include both hyper-parameter tuning during training (e.g. population-based training [JDO+17]) and automated device placements [MGP+18, JZA18]. While theoretically appealing, compile-time optimisations have received sparse attention in the RL literature due to the multitude of other concerns. The lack of systematic graph construction in existing frameworks also makes integration of these techniques difficult.

Finally, mobile and edge (e.g. on IoT sensors) deployments of neural network models in domains like computer vision often rely on a combination of pruning, compression, and graph rewriting techniques [LBM+17, ZGMX18]. It is unclear if these techniques are

readily applicable to shrink policy networks in RL. As algorithms are known to be sensitive to small changes in the input distribution [KSM+17], further investigation is needed.

RLgraph could naturally integrate novel optimisations in its build mechanism. The prototype implementation contains one example of graph rewriting for device strategies, i.e. automatically splitting workloads across multiple accelerators.

3.7 Limitations

Decoupling execution semantics from logical dataflow decomposition increases flexibility for applications built on top of RLgraph. This comes at the cost of enforcing an implementation style which may be unintuitive for researchers used to implementing an algorithm and its execution logic in a single module. RLgraph is best suited for users who need to maintain and incrementally build up implementations for longer-term research or real-world deployments. While algorithmic researchers may prefer ad-hoc implementations, they benefit from RLgraph's incremental testing and graph assembly to obtain performance insights. Here, I discuss a number of current limitations and how they can be addressed in future work.

3.7.1 Multi-agent communication

Current multi-agent scenarios are biased towards centralised learners and two-layered task hierarchies. In Chapter 5, I describe how a task-graph architecture on top of RLgraph facilitates task decomposition.

Emerging multi-agent scenarios include competitive and cooperative learners which may exchange not only weights but also instructions, questions, rewards, or gradient to enable rich communication semantics. A number of existing multi-agent implementations (e.g. in RLlib or PyMARL [SRdW+19]) assume nested parallelism but do not consider these new scenarios. Multiple policies or agents are organised into unordered sets of individual models.

To provide declarative generalised multi-agent semantics, I propose to implement an agent graph architecture wherein each node refers to a single RLgraph graph. Specifically, a single RLgraph architecture can theoretically include many independent optimisations for unrelated or hierarchically arranged tasks. The disadvantage of a large shared graph for all tasks is that individual tasks cannot be independently scheduled, and that changing communication between tasks requires modifying implementation of that graph.

A more flexible design for multi-agent execution restricts each RLgraph graph to a single differentiable graph with explicit interfaces for communication. Developers create multi-agent scenarios by first declaring individual agents. They can then specify communication channels and task hierarchies by creating edges in the agent graph.

Existing execution engines are not well suited to decentralised multi-agent scenarios where agents create ad-hoc communication channels and asynchronously interact with shared or separate problem environments. Distributed TensorFlow is best suited for single-end-to-end graphs while Ray focuses on centralised hierarchical parallelism.

3.7.2 Graph flexibility

RLgraph implements all-or-nothing semantics with regard to building static graphs. Since RLgraph encapsulates graph construction, device assignment, and session management, it does not allow static graph designs whereby parts within an algorithm are outside of RLgraph, other than preprocessing of inputs and post-processing of outputs. This restriction does not apply to define-by-run mode. If a new design requires semantics not supported by the graph building mechanism, substantial development may be needed to add a feature throughout the framework. For some research use cases, a stand-alone implementation can be preferable.

3.7.3 Gradient-free optimisation

A key purpose of RLgraph's build process is to facilitate end-to-end differentiable graphs with dynamic control-flow. Gradient-based optimisation is the most popular optimisation paradigm in training deep neural networks. However, gradient-free approaches (e.g. evolutionary strategies [SHCS17, SMC+17] , augmented random search [MGR18]) may in the future serve as alternatives. They are simple to parallelise and can achieve competitive results. The overhead of the build procedure to ensure an end-to-end static computation graph may not be justified if automatic differentation is not required.

3.8 Summary

In this chapter, I argued for the separation of algorithm logic from execution when implementing RL approaches. I based this argument on an analysis of workload characteristics and design challenges encountered in fast-changing algorithms (§3.1). I then gave an overview of RLgraph and described how it decouples logical component composition from physical execution plans (§3.2). I subsequently described RLgraph's key abstractions, graph building mechanisms from a component intermediate representation, and high level agent API (§3.3). Finally, I explained how different local and distributed backends engines and execution semantics can be integrated (§3.4).

In summary, RLgraph is a programming model to design reinforcement learning algorithm and execute them in different application contexts and execution paradigms. It offers a high-level plug-and-play frontend API for practitioners, and extensive dataflow composition tools for researchers. The main contribution lies in mapping high-level subgraph abstractions to end-to-end differentiable dynamic control flow without requiring users to explicitly construct this dataflow graph. RLgraph's abstractions can also be used outside of RL, but other domains have already established stronger standardised workflows.

Using RLgraph, researchers and practitioners benefit from being able to rapidly explore new designs without tedious manual tensor manipulation due to space-independent components. In Chapter 4, I show that RLgraph generates high-performance execution plans across backends while incurring negligible overhead from its abstractions.

Chapter 4

RLgraph evaluation

4.1 Evaluation aims

In this chapter, I evaluate my prototype of the decoupled component graph architecture. RLgraph must be able to generate high-performing execution plans for different execution engines. The aim of the evaluation is not to benchmark these frameworks or reproduce published benchmark results. Instead, I analyse how RLgraph helps address RL-specific design challenges. I focus on answering these questions:

1. What overhead does RLgraph's build process incur for different backends? (§4.2)
2. How does RLgraph perform using third-party execution engines compared to native libraries? (§4.3.2)
3. Are there benefits in robustness in large evolving codebases when using RLgraph? (§4.3.3)
4. How does RLgraph compare to tuned one-off implementations?
5. How can algorithm designers use RLgraph to compose new methods (§4.6)?

4.2 Build overhead and backends

I first evaluate RLgraph's graph construction mechanism. There are two sources of overhead. First, the intermediate representation is created by tracing dataflow from root component API definitions. Second, the backend-build mechanism creates an executable computation graph and operation definitions.

Figure 4.1 shows the runtimes of constructing three component graphs. Tracing a single component's API (an experience buffer for sample trajectories) takes less than 50 milliseconds. A more involved architecture such as double-dueling DQN with a number of image preprocessors and multiple optimisers traces for approximately 500-600 ms with 61 components. More important than the number of components is the number of operations to trace between components. A third architecture, Proximal Policy Optimisation (PPO), uses 51 components but requires the component graph procedure to build only 381 operation records (versus 947 for DQN). As expected, runtime for component tracing does

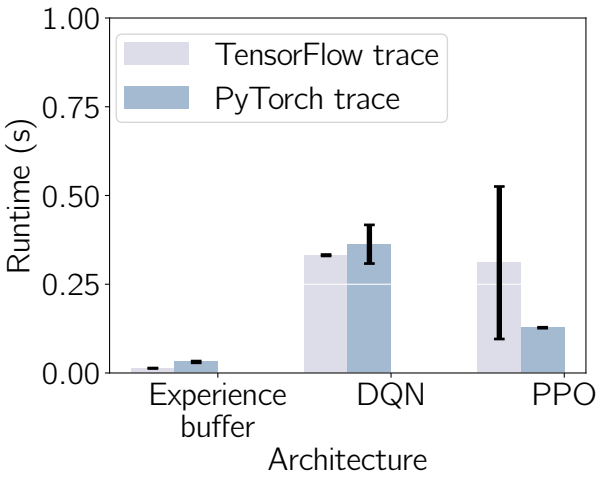


Figure 4.1: Component graph trace overhead for a single component and two common agent architectures.

not vary (beyond hardware variation) for the static and define-by-run backends, as no graph functions are built. Both DQN and PPO are built for the same environment layout (ALE Pong from image inputs), and are configured with the same network architecture.

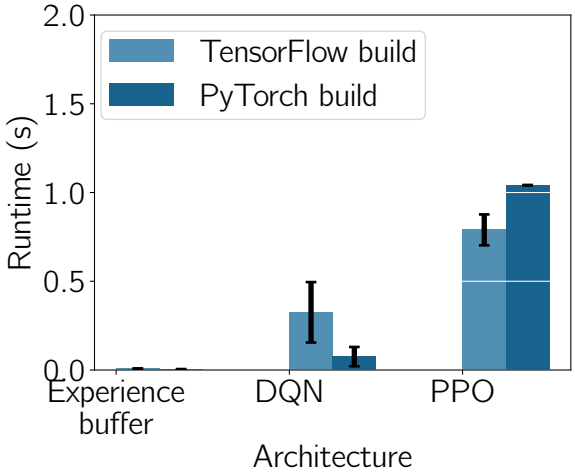


Figure 4.2: Corresponding backend build overheads induced by the modular graph function and variable creation.

Figure 4.2 depicts the corresponding overhead from the second build phase of building/testing backend operations. Note that build overhead here only refers to the extra time required to iteratively create graph fragments. It does not include the time required to create operations or variables which would have to be done irrespectively of using RLgraph (i.e. in an ad-hoc implementation). The build overhead only weakly correlates to the number of components or records, as PPO is much more expensive to build than DQN. The reason lies in the fact that graph functions (i.e. operations) can internally call other API methods and operations. PPO uses an in-graph iterative update procedure which uses dynamic control flow to invoke several nested graph functions. Building the graph functions from within the nested dataflow requires additional build iterations.

PyTorch builds DQN substantially faster than TensorFlow as it does not need to

connect individual graph fragments, but executes each API method once by marshalling prototype tensors through the graph. This reverses when building PPO, as the iterative optimisation is fully executed in define-by-run mode. As the build includes multiple mini-batch gradient updates to test functionality, its runtime exceeds static graph mode.

The main take-away is that building individual components creates less than 50 ms in combined overhead irrespective of graph semantics. This makes per-component building practical to incrementally debug and test dataflow. Building entire architectures can add 1-2 seconds overhead to test-run all dataflow in define-by-run mode. This is nonetheless negligible when compared to typical runtimes of RL experiments (hours or even days).

The experiments in this section were run on a commodity server class machine using the Google Cloud Platform (GCP) with 8 CPUs.

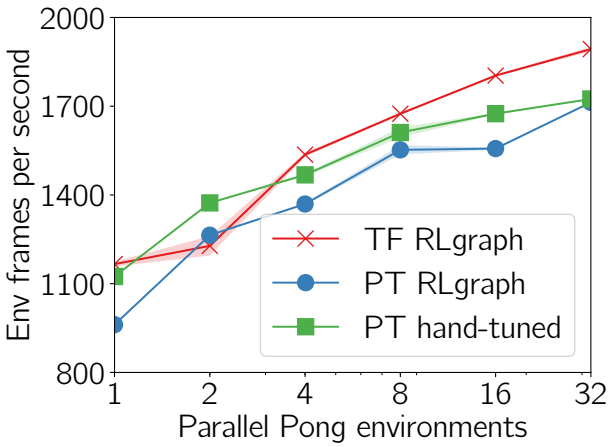


Figure 4.3: Act throughput on ALE Pong for a varying number of simulator copies.

Next, I analyse run-time overhead caused by RLgraph's abstractions. Using a static graph backend does not incur overhead because RLgraph's intermediate representation is discarded after the build. Executing API requests only requires a lookup for the operation to invoke via e.g. TensorFlow's session interface. The define-by-run backend incurs overhead caused by intermediate evaluations of API decorators.

In Figure 4.3, I evaluate act (i.e. forward passes through the Q-network) performance on different backends. I use a standard 3-layer convolutional architecture followed by a fully connected layer corresponding to the widely used ALE benchmark. I also implemented a hand-tuned bare-bones PyTorch implementation of the same architecture and evaluated all variants on the ALE Pong environment. When using few environments, the hand-tuned PyTorch implementation outperforms RLgraph. For TensorFlow, the overhead of invoking a session is not amortised by potentially (statically optimised) execution. As batch size (i.e. number of environments providing new states) increases, TensorFlow gains favourably and outperforms both PyTorch variants.

Regarding the use of different backends, I conclude that:

- The build overhead of RLgraph's component architecture is negligible compared to the duration of RL workloads. The relative difference in overhead between define-by-run and static backends is a secondary consideration for running workloads. Building individual components for testing incurs a negligible delay for interactive debugging workflows.

- For runtime performance, RLgraph's define-by-run mode incurs overhead compared to a bare-bones implementation. When batching forward passes through environment vectorisation, runtime is dominated by neural network operations while overhead remains fixed.
- TensorFlow's static graph backend outperformed defined-by-run mode for larger networks when using vectorised action selection. For all following experiments, I report results using the TensorFlow backend.

4.3 Execution on Ray

4.3.1 Setup

Next, I evaluate RLgraph on the distributed execution engine Ray in comparison to Ray's native library, RLlib (v0.5.2) [LLN+18]. First, I evaluate if RLgraph's design creates performance advantages or introduces overhead when compared to a native library tightly integrated with an execution engine. Second, I demonstrate how developers can rapidly explore new Ray execution semantics via RLgraph's execution-agnostic API.

The relevant performance dimensions are (i) experience collection throughput as a measure for implementation efficiency and (ii) learning success measured by reaching a score in a known benchmark.

For this comparison, I implemented distributed prioritised experience replay (Ape-X [HQB+18]), a recent Q-learning algorithm on RLgraph's Ray executor. I configured both RLlib and RLgraph with the same training hyper-parameters (using RLlib's own optimised configuration). Experiments were performed on Google Cloud with the centralised learner being hosted on a GPU instance with 1 active V100 GPU, 24 vCPUs and 104 GiB RAM. Sample collection nodes had 64 vCPUs and 256 GiB RAM. Ape-X utilises asynchronous sampling tasks. New samples are initially inserted into one ore more sharded buffers. The learner schedules sampling tasks which retrieve batches for updating.

4.3.2 Results

Figure 4.4 shows sampling performance on the Pong environment. The x-axis represents the number of policy-evaluators/Ray-workers respectively (RLlib, RLgraph), each initialized with a single CPU, and y-axis shows environment frames per second (including frame skips). Each worker executed 4 environments, and I used 4 instances of replay memories to feed the learner. All settings were run with 8 sample nodes except 256 workers (16 sample nodes) to ensure sufficient memory. RLgraph outperforms RLlib by a large margin (185% on 16, 60% on 256 workers) despite implementing the same algorithm with equivalent hyper-parameters and model architecture. Performance for 16 workers is highest due to better resource utilisation. Improved performance is not caused by one single insight but rather compound effects:

- RLgraph's environment interaction processing was implemented from scratch. Existing implementations, including RLlib's, can be traced back to example code in open source which was not optimised for performance. For example, RLgraph micro-batches worker-side post-processing, and further samples exploration actions without graph invocations.

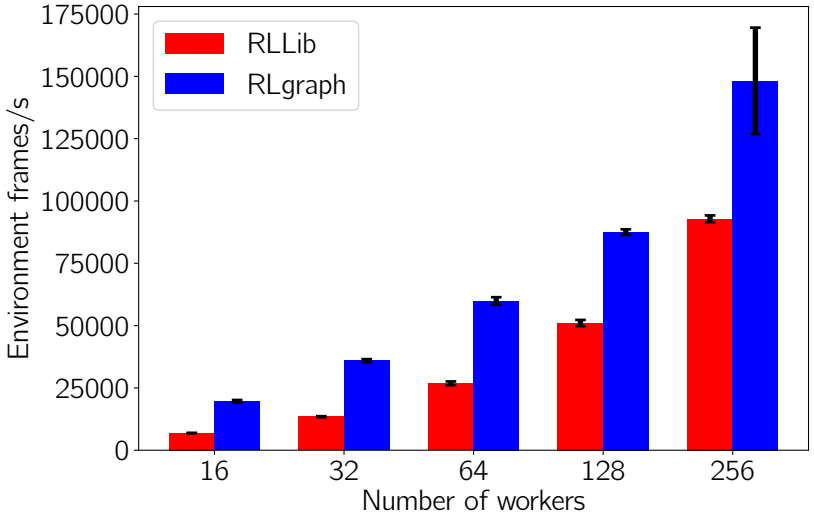


Figure 4.4: Distributed sample throughput on ALE *Pong*.

- I also observed memory leaks when interacting with Ray's object store. This caused RLgraph's implementation to undergo several rewrites during debugging (and thus led to uncovering various improvements related to intermediate result materialisation).
- RLgraph also completes its sampling tasks faster due to improved implementations in low-level data structures. For example, RLlib used a recursive implementation for its priority management segment tree. An iterative implementation yielded 30-40% improvements for insertion and replay sampling tasks.

Since RLgraph's publication, RLlib has also undergone changes to adopt a more functional approach similar to RLgraph, and the memory leaks in its object store have since been resolved. RLgraph's sample throughput is not at the cost of learning performance (Fig.

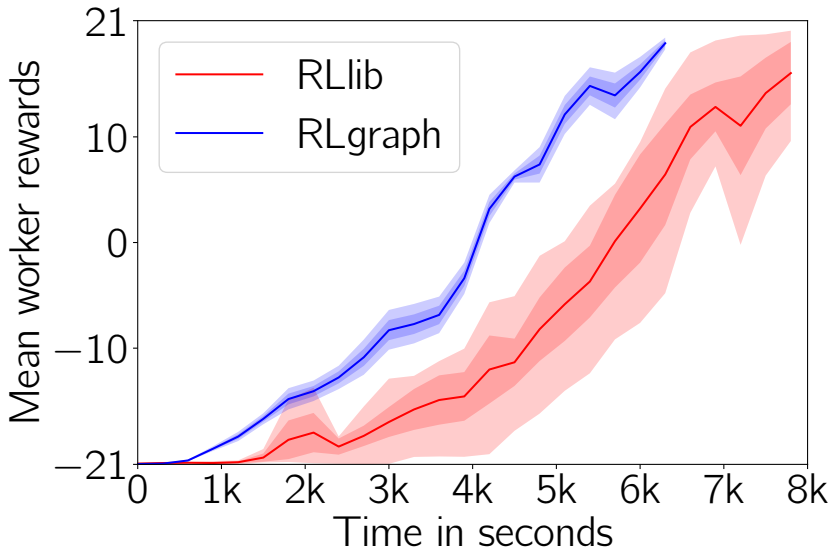


Figure 4.5: Learning performance on ALE *Pong*.

4.5). To compare learning, I adopted RLlib's provided tuned *Pong* configuration (32 workers). In RL, the same code using the same hyper-parameters can vary drastically across runs, so reliably comparing learning is difficult [HIB+17]. I ran 10 random seeds

and average across the 3 best runs (both libraries did not learn anything for some seeds as expected). In line with throughput, RLgraph learns to solve (i.e. achieving the maximum episode reward of 21) Pong substantially faster than RLlib.

RLlib's published results on Ape-X throughput do not include updating without stating this explicitly, and later reported results including updates[1] are up to 130 k frames per second on 256 workers (versus 170 k max for RLgraph). Some performance differences may be attributed to different hardware setup (fewer CPUs on head node, GCP GPU latency). Experimental versions of a new Ray backend include improved garbage collection of which RLgraph would benefit to the same extent as RLlib.

My results show RLgraph's execution-agnostic design can integrate with an external execution engine and perform competitively. While RLlib could adopt more efficient implementations, my main insight is that RLgraph can be used on Ray via few wrapper classes. Implementing other distributed semantics on Ray with RLgraph only requires extending the generic Ray executor to implement a coordination loop (§4.3.4).

4.3.3 Robustness

I argue that RL programming models without fine-grained modularisation and testing are unsuitable for maintaining larger code-bases due to the brittle nature of algorithmic performance in RL workloads. I repeated the learning experiment with a subsequent version of RLlib (v0.6.0) and compared again to RLgraph. Both libraries had undergone substantial changes between experiments (4 months).

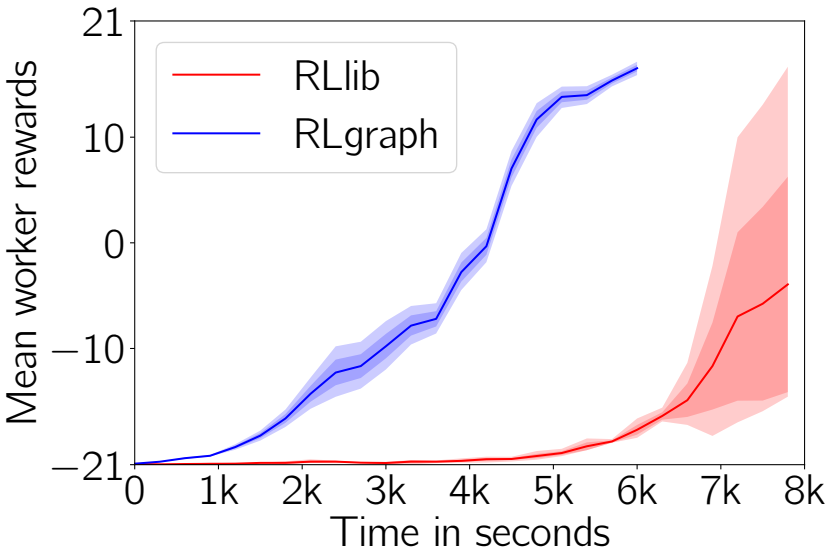


Figure 4.6: Replicated learning performance on ALE *Pong*.

Figure 4.6 shows results from the repeated experiment (again running 10 random seeds). While RLgraph closely mirrors prior training results, RLlib performs significantly worse due to undocumented changes. In a cursory analysis, the authors attempted to improve sample throughput by discarding batches whenever the learner blocks on queued training samples. Update throughput decreased, thus slowing down learning.

Table 4.1 gives a detailed breakdown of random seed performance across initial and replicated experiments. An experiment is considered a success if a maximum reward r_{max}

[1]Source: RLlib authors, `https://github.com/ray-project/ray/issues/2466`, accessed 25.08.2019

```
def _step(self, samples):
  # 1. Sync local learners weights to remote workers.
  weights = ray.put(RayWeight(self.local_agent.get_weights()))
  for ray_worker in self.ray_env_sample_workers:
    ray_worker.set_weights.remote(weights)

  # 2. Schedule samples and fetch results from RayWorkers.
  sample_batches = []
  num_samples = 0
  while num_samples < self.update_batch_size:
    batches = ray.get([worker.execute_and_get_timesteps.remote(samples)
                       for worker in self.ray_env_sample_workers])
    # Each batch has exactly worker_sample_size length.
    num_samples += len(batches) * self.worker_sample_size
    sample_batches.extend(batches)

  env_steps += num_samples
  # 3. Merge samples.
  batch = merge_samples(sample_batches, decompress=self.compress_states)

  # 4. Update from merged batch.
  self.local_agent.update(batch, apply_postprocessing=False)
  return env_steps
```

Listing 4.1: Implementation of distributed coordination on RLgraph. Here, a centralised learner consumes the agent API to facilitate distributed synchronous policy optimisation.

of 20 is reached and a failure if r_{max} is below -10 (untrained initial value is -21). While Ray RLLib can solve Pong, it does so significantly less often than RLgraph as subtle implementation problems affect training.

Framework :	$r_{max} \geq 20$	$r_{max} \leq -10$ (fail)	$-10 \geq r_{max} < 20$ (straggler)
RLgraph	15	3	2
RLlib	6	11	3

Table 4.1: Breaking down random seed performance for Atari Pong.

RLgraph's modularisation helps visualising and understanding dataflow in computation graphs when compared to fragmented programming models such as RLlib's. When executing on Ray, computation graphs are fragmented into separate tasks represented by Ray actors, as opposed to end-to-end differentiable graphs such as in distributed TensorFlow [AIM17, YAB+18]. In RLlib, task code is scattered across agent classes, policy graphs, and backend-specific utilities. Debugging dataflow between components is difficult due to a lack of consistent modularisation across imperative function calls.

4.3.4 Implementing new coordination semantics

Executing RLgraph under different coordination semantics on Ray only requires changing the centralised coordination loop responsible for task scheduling and result processing. In Listing 4.1, I implement distributed synchronous coordination which simply schedules sampling tasks, merges samples, and passes environment trajectories to a local learner via the agent API. Weights are synchronised to workers after every update. The example further illustrates how local updates (i.e. line 22) are decoupled from the Ray executor which only consumes the agent API.

4.4 Multi-GPU mode

Effective use of hardware accelerators (e.g. GPUs) depends on large training batches available at high frequency to utilise accelerator memory bandwidth. This is at odds with interacting with an environment where even a simulator may take seconds to build up a large sample trajectory. High GPU utilisation is hence primarily achieved in off-policy scenarios where sample batches can be retrieved from an asynchronous buffer without waiting on environment interaction.

A key design choice in RLgraph is that distributed processing is decoupled from local device management. For example, Ray executors, irrespective of their distributed coordination logic, request an update from the agent API which can transparently choose to schedule update operations across multiple devices. I evaluate RLgraph's multi-gpu mode using the Ape-X executioner as it can generate high sample throughput to make effective use of multiple GPUs. I repeat the learning experiments from the prior sections and compare impact.

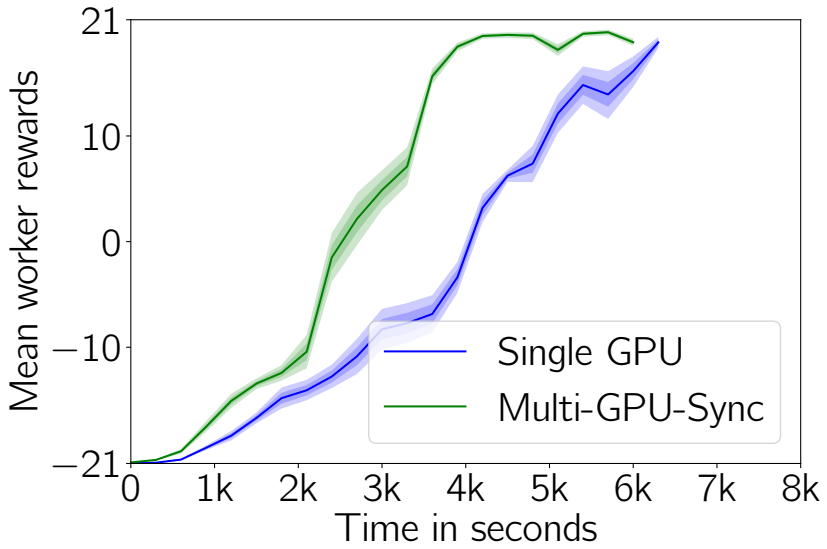


Figure 4.7: Single versus 2-GPU learning performance on ALE *Pong*.

Figure 4.7 shows how multi-GPU acceleration provides significant speed-up in training. Importantly, this speed-up is achieved without changing a single line of code in RLgraph's Ray execution package.

4.5 Distributed TensorFlow

I also evaluate RLgraph using the distributed TensorFlow backend on DeepMind's (DM) importance-weighted actor-learner architecture (IMPALA) [ESM+18].

The authors have open-sourced an optimised implementation[2]. IMPALA perhaps best represents the end-to-end computation graph paradigm, where even environment interaction is fused into the TF graph. RLgraph provides generic execution components for graph-fused environment stepping based on DeepMind's implementation (adapted by Sven Mika). IMPALA executes updates by letting each actor perform a rollout step and input its samples into a globally shared blocking queue. The learner dequeues rollouts and uses a staging area to hide GPU latency.

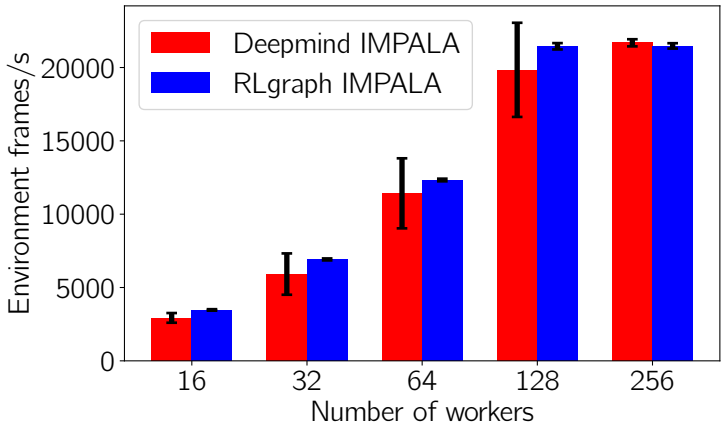


Figure 4.8: Distributed sampling throughput on DeepMind lab tasks.

Figure 4.8 compares throughput using the large network described in the paper on a DM lab 3D task (which are more expensive to render than Atari tasks). I use a single V100 GPU for the learner and let 4 workers each share a 8 vCPU instance. RLgraph achieves about 10-15% higher mean throughput (5 runs) for fewer workers until both implementations are limited by updates. DM's implementation exhibited higher variance due to subtle differences in preprocessing tensors after unstaging. DM's code also carried out unneeded variable assignments in the actor. Removing these yielded 20% improvement in a single-worker. I did not reproduce learning experiments for the published IMPALA benchmark tasks due to prohibitive cost. Tests on toy environments and individual IMPALA components however indicate expected training behaviour.

The purpose of the IMPALA comparison is to demonstrate RLgraph's abstractions can be used even when delegating distributed execution and environment management entirely to a backend-specific execution engine. This is in contrast to using a third-party engine as a consumer of RLgraph's API. Performance differences are a consequence of minor inefficiencies. The main take-away is that end-to-end graphs, while being more difficult to implement without a build-system, are not prone to the problems encountered in mixed-backends. Using Ray, implementations were highly sensitive to data-type and copy errors where wrongly referenced objects could drastically slow down code.

[2]Code at: `https://github.com/deepmind/scalable_agent`, accessed 25.08.2019

4.6 Exploratory workflows for algorithm design

Next, I discuss how RLgraph can be used to compose new algorithmic variants. Many new algorithms do not introduce theoretical frameworks but recombine existing abstractions. Consider the distributed Q-learning variant Ape-X [HQB+18] I used as a benchmark against RLlib. I decompose Ape-X in algorithmic elements and execution elements.

At its core, Ape-X utilises a double-dueling DQN loss-function [WSH+16] to train a Q-Network. To accelerate distributed training, the following execution modifications to the original DQN [MKS+15] workflow were introduced:

- Acting and learning are decoupled so the learner is not bottlenecked by sample processing.
- Distributed actors collect sample in local buffers. Each actor has a different exploration policy, e.g. by sampling a different initial value for ϵ-greedy exploration. Moreover, they compute sampling priorities for collected trajectories by computing the temporal-difference loss, without applying updates. They also pre-process sample trajectories by computing n-step discounts.
- Periodically, actors send preprocessed samples to a global shared buffer. The learner samples from this buffer according to precomputed priorities in one thread and passes them (e.g. via a queue) to a separate learner thread. The learner thread computes Q-learning updates which yields new loss values to be used as sample priorities which are written to the shared buffer.
- The learner periodically transmits Q-function weights θ to actors.

The execution semantics of Ape-X rely on Q-learning to be off-policy but are otherwise orthogonal to the loss function. To illustrate how RLgraph facilitates algorithmic composition, I create an algorithmic variant based on combining the Soft Actor Critic (SAC) [HZAL18, HZH+18] and Ape-X.

SAC describes a collection of algorithms combining off-policy updates with a stochastic actor-critic policy based on a maximum entropy framework. In the maximum entropy regime, agents do not only maximise the discounted reward but also the expected entropy of the policy to control stochasticity [ZMBD08]. Harnooja et al. evaluate several SAC variants which significantly outperform prior model-free methods in various continuous control benchmark tasks. As the primary objective of SAC is to combine off-policy learning with continuous stochastic policies, no evaluation was executed on discrete tasks.

I modified SAC (contributed to RLgraph by open source contributor Janislav Jankov) to i) interface discrete problems and ii) execute using Ape-X semantics. This is based on the observation that Ape-X incurs unfavourable exploration characteristics due to manually defined epsilon-greedy strategies. A discrete stochastic policy naturally explores by sampling from the parametrised policy distribution. Further, in my distributed Ape-X experiments the learner was bottlenecked by sampling training batches from buffers, not update throughput. This was experimentally verified by creating a fixed size queue between controller and learner, and counting all instances where the controller was blocked on a full queue, thus indicating the learner not keeping up. Queue blocking did not occur regularly beyond warm-up. I hence hypothesise:

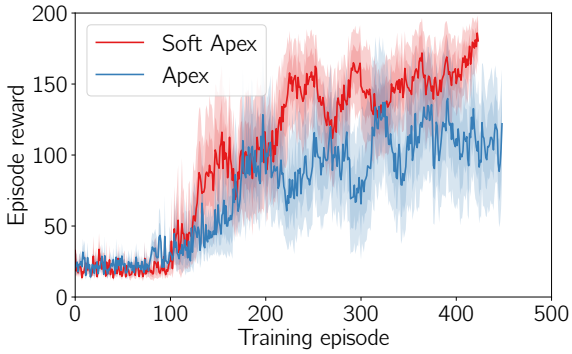


Figure 4.9: Ray worker 1 performance.

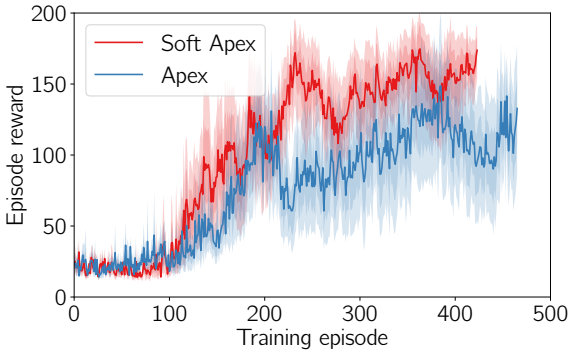


Figure 4.10: Ray worker 2 performance.

- Centralised distributed training scenarios on high-dimensional input states (e.g. images) are bottlenecked by sample collection and communication overhead when sending samples to a centralised learner. This allows more computationally expensive training mechanisms.

- SAC decouples value estimation from a stochastic policy actor. While the learner trains not only the policy but multiple separate value functions and the entropy temperature parameter, actors only need to synchronise policy parameters.

A Soft Ape-X (or distributed discrete Soft Actor Critic) learner can hence utilise underused learner resources. It further solves the exploration problem in Ape-X through on-policy exploration and the augmented entropy objective.

A discrete policy in standard policy optimisation methods is represented by a categorical distribution. However, SAC's training mechanism requires action samples to be differentiable with regard to distribution parameters which is not the case for the categorical distribution. A continuous approximation to the categorical distribution is available via the Gumbel-Softmax distribution [JGP16].

I implemented a Gumbel-Softmax distribution and acting policy as an RLgraph component which can be used by the SAC implementation. The only additional modification required is to convert Gumbel-Softmax samples to discrete actions in the output by computing the arg-max for the sampled probability vector. I then combined discrete SAC with Ape-X for a distributed discrete SAC or "soft Ape-X" which did not require any modifications on the Ape-X executor.

Figures 4.9 and 4.10 show training performance (10 random seeds) on the classic CartPole balancing benchmark as a minimal example task, comparing both algorithms. Each were run for 100k time steps, resulting in a variable number of episodes (episode length correlates to reward). Two Ray workers (left and right figures respectively) were used on a single node with a single off-policy buffer and a centralised learner. On the CartPole task, Soft Apex outperforms the naive Apex variant. Shaded regions indicate confidence intervals for rewards achieved in by the respective training episode (dark shaded area CI=65, light shaded area CI=95). Despite performing a more expensive update (as Soft Apex computes updates separately for value function and policies), both implementations achieved the same update throughput. As hypothesised, update throughput was not limited by the learner but sample I/O.

Benchmarks on larger discrete tasks such as ALE or DeepMind lab would require costly hyperparameter sweeps. Here, I highlight the exploratory workflow enabled by RLgraph.

Testing the initial viability only required to implement a Gumbel-Softmax distribution component and a corresponding action adapter component which interprets distribution samples. All other algorithmic and execution components could be combined as is.

4.7 Summary

In this chapter, I investigated RLgraph's abstractions as a means to implement and execute RL workloads across a number of different execution paradigms. My experiments demonstrate my thesis that the decoupled component graph architecture:

1. incurs negligible overhead through its build process. The static graph backend via TensorFlow adds no runtime overhead as build abstractions are discarded. The define-by-run backend introduces some overhead for intermediate evaluations.

2. generates high-performance execution plans on multiple distributed backends with highly customised device strategies.

3. produces modular, incrementally testable implementations which result in robust learning performance, significantly outperforming prior implementations.

4. is extensible to different device and execution paradigms due to its separation of concerns. Since components make no implicit dependency assumptions and transparently configured through spaces, they are fully re-usable.

5. helps composing new algorithmic variants through reusable building blocks.

To summarise, RLgraph's programming model results in high-performing implementations at the cost of low initial build and runtime overhead. In Chapter 6, I illustrate how RLgraph can be used to implement a highly customised application scenario by combining multiple agents.

The fast-moving nature of empirically driven RL research on the one hand combined with high individual customisation per algorithm on the other hand means framework designers cannot commit to a single execution architecture or learning paradigm. In Chapter 7, I discuss how RLgraph can support new emerging paradigms.

Chapter 5

Wield: Incremental task design with progressive randomisation

The emergence of deep neural networks as powerful function approximators has caused a surge of interest in systems applications of RL. The shared motivation across these new inquiries stems from two goals. First, learned controllers would relieve systems designers from manually engineering automation solutions in ever-larger systems. Second, learned controllers promise to adapt system behaviour to workload distribution, thus outperforming best-effort heuristics.

The notion of applying RL to systems problems is not novel (§2.4) but translating experimental successes to practical deployments remains difficult. In this chapter, I argue that research in this domain suffers from fragmentation and lack of experimental standardisation. In the absence of shared tasks, experiments are performed in customised environments with hand-picked workloads. State, action and reward representations further depend on difficult to reproduce manual featurisations. RL frameworks like RLgraph or RLlib assume pre-existing environments. They operate on state and action descriptions provided through standardised interfaces (gym). They are not concerned with how the problem representation itself is designed. In applied domains like computer systems, practitioners must manually explore effective problem state representations and actions based on domain knowledge.

I present Wield, an architecture to accelerate applied RL research. Wield provides the following mechanisms:

1. A set of common abstractions to model systems optimisation tasks as RL problems (§5.4). Wield's abstractions are shared by different online and offline workflows requiring to convert trajectory data between agent and system representations. Wield further introduces a task graph abstraction to decompose learning tasks into independent and hierarchically related sub-tasks as a means to handle large state and action spaces (§5.4.3).

2. It delineates randomisation in training, testing and workload generation by providing different blackbox and generalisation protocols, each at different levels of determinism. To this end, Wield introduces an experiment protocol and classification scheme called progressive randomisation (§5.5).

3. It introduces demonstration abstractions for training scenarios where pre-existing

trace data can be enriched using weak supervision to generate off-policy training data (§5.6).

Reinforcement learning applications in systems compete with a wealth of optimisation approaches. I begin by discussing common tuning approaches to determine (i) the trade-offs involved in using RL and (ii) set expectations with regard to practical utility for different use cases. Parts of this chapter have been published as a preprint and are under peer review [SFY19].

5.1 Optimisation in computer systems

5.1.1 Iterative optimisation

Optimising configurations to tune deployments is a central challenge in computer systems. In recent years, designers and practitioners have increasingly adopted automated tuning workflows using so-called auto-tuners.

Spearmint is an early framework to provide Bayesian optimisation (BO) as a generic tuning mechanism. Its high-level interface omitting operational details of Gaussian Processes sparked a wave of research into practical applications of BO [SLA12]. For example, CherryPick is a cloud-configuration tuning framework which interfaces Spearmint to select VM types for cloud workloads [ALC+17]. Arrow is a similar cloud-configuration system which includes low-level performance metrics to reduce search time [HNFM18]. OtterTune is a domain-specific framework to provide end-to-end BO for databases [VAPGZ17]. Instead of indiscriminately tuning (potentially many) parameters, OtterTune performs factor analysis and clustering to identify performance-critical settings which are then fine-tuned via BO.

OpenTuner [AKV+14] is an auto-tuner library combining several optimisation techniques such as random search and evolutionary algorithms in parallel. OpenTuner's key contribution is a meta optimiser which allocates more resources to methods which yield higher improvements across iterations. Golovin et al. describe an internal tool at Google, Vizier, which serves as an auto-tuning service similarly selecting from several search techniques [GSM+17]. To manage large parameter spaces, BOAT decomposes problems into semi-parametric sub-problems [DSY17]. This enables users to inject domain-knowledge by implementing parametric functions of sub-problem behaviour. BOAT significantly outperforms other auto-tuners in domains with expensive evaluations, e.g. larger systems experiments. Uncovering parametric relationships however requires extensively manual analysis.

The main difference between deep RL and these auto-tuners is that neural network-based models can outperform other approaches due to learning representational structure in high-dimensional problem inputs. This comes at significant additional design cost (§5.7).

5.1.2 Analytical performance models

Less automated approaches fit pre-designed parametric models by executing partial workloads. Ernest for instance [VYF+16] predicts resource requirements for analytics workloads by executing small input samples. The performance profile of the sample execution is used to fit a parametric model of resource costs and predict a configuration. A

similar approach was proposed by Paragon which classifies cluster scheduling workloads by comparing test runs with prior workloads to match new workloads to cluster server types [DK13]. Quasar extends Paragon to include resource constraints and scale-up/scale-out scenarios in large scale cluster management [DK14].

Optimus is a scheduler for deep learning workloads on top of Kubernetes which learns training speed for different resource configurations to dynamically allocate resources [PBC+18]. A model of the training loss curve is obtained by decomposing training into forward and backward pass cost and communication overhead and fitting respective coefficients via a least squares solver. Prior approaches also include offline feature analysis where workload traces are clustered to identify common workload scenarios and prepare per-scenario responses. Teabe et al. successfully employed this strategy to improve hypervisor scheduling [TTH16]. Kumar et al. presented a similar scheme to determine waiting durations on aggregation queries [KARS15].

These empirical performance models are similar to RL methods in that they also iteratively incorporate system feedback to update a model. Unlike the deep RL approaches I focus on in this dissertation, they do not utilise representational learning but hand-constructed parametric performance models. This creates a trade-off where representational learning can result in higher performance at the cost of requiring orders of magnitude more training data. Moreover, training neural networks incurs higher computational cost and introduces additional hyper-parameters. Finally, hand-designed feature models with closed-form analytical expressions or simple heuristics exhibit predictable performance.

The overhead of building deep-learning based approaches in systems can only be justified if performance improvements against other approaches result in large cost or resource savings.

5.2 Practical considerations and limitations

Systems research focuses on the study and design of software artefacts. New algorithms or system designs are often evaluated with emphasis on practical utility. However, a "deployment gap" exists between a fast growing body of research on the one hand, and limited practical systems utilising RL on the other hand. I discuss limitations which may have prevented experimental successes from practical use.

A key limitation is the use of task models which are only practical on simplified workloads and deployments. This goes hand-in-hand with missing discussions on model scalability. A canonical example of this issue are resource management problems where the number of actions is linear (or superlinear) in the number of resources, and a proof of concept on a small number of resources is proposed as indicative for a solution to the full scale problem. I discuss task models in §5.4.3.

Research in this domain also suffers from severe reproducibility issues. This is partially a consequence of missing common benchmarks for fine-grained decision making. For example, traditional database benchmarks like the TPC suite of tests [Tra10] are application-driven, i.e. the benchmark constitutes one representative use-case. Such fixed queries ensure fair comparisons on e.g. transaction throughput of query optimisers. However, training and evaluation deep learning methods on a handful of queries known in advance can cause non-transferable designs and overfitting. This leaves researchers implementing customised new workload variants which are not typically released. If generalisation is evaluated, test set design criteria remain opaque. I argue for randomisation and the need for new

evaluation protocols in Section §5.5. In Wield's evaluation, I show how slight variations of query sets and task difficulty drastically affect training results.

Moreover, in systems applications of RL, states, action and reward models often rely on processing system metrics or updating system configurations (§5.4.3). Models cannot be reproduced without full details on these preprocessing steps alongside model hyper-parameters. As a result, research is fragmented, and progress on similar problems difficult to assess.

Finally, the use of custom-tailored simulators is prevalent in applied RL. Recent work into new applications of RL has often relied on simulators which are not released, and which cannot be assessed. Unless strongly grounded in a prior problem model (such as protocol analysis, motion dynamics based on real world physics), reliance on custom simulators makes practical transfer in systems unlikely. A further consequence of missing shared benchmarks are missing shared baselines. RL models can unsurprisingly outperform baselines not tuned towards specific workloads. This does not answer the question on how good a specific representation is at finding effective representations.

If an RL model finds an effective set of device assignments or query plan, a natural baseline to compare against is variants of random search [MGR18]. Succinctly, researchers must distinguish in their baselines between adapting to a workload (as opposed to e.g. rule-based heuristics), and the quality of this adaptive mechanism. For example, Li and Talwalkar found that random search matched prior approaches in neural architecture search [LT19]. Architecture search, i.e. automated design of neural network architectures for a given task and data-set, was earlier highlighted as a potential RL application [ZL17]. The term random search here can refer to a variety of search strategies ranging from directly sampling a solution to augmented random searches which randomly mutate network parameters based on rewards. They resemble evolutionary methods.

In summary, value function approximation with ever-larger neural networks is unlikely to be sufficient to address the deployment gap. The lack of sample efficiency, algorithmic brittleness, and problematic scalability of action spaces in large-scale systems calls into question how model-free approaches can reach practical utility. This does not imply there are no practically viable applications. However, evaluating the utility of current approaches is obscured by the complications discussed in this section.

Wield makes two contributions towards these challenges. First, it helps design RL task representations through its abstractions which decouple system interface, representation, and data layouts. Second, it provides an instructive experimentation protocol to incrementally evaluate model capabilities.

5.3 Wield Overview

Delineating progress requires systematic assessment and comparison of approaches. The aim of Wield is to provide reusable abstractions to standardise task design for systems applications of reinforcement learning. The same abstractions can further be used to interface traditional auto-tuners as baselines. Wield also provides different optimisation and randomisation modes to evaluate sensitivity to workload and optimisation stochasticity (§5.5).

Figure 5.1 gives a conceptual overview of Wield. On a high level, Wield acts as an interface between a data processing system (e.g. database, distributed stream processing engine, scheduler) and RL frameworks such as RLgraph (or another auto-tuner or any

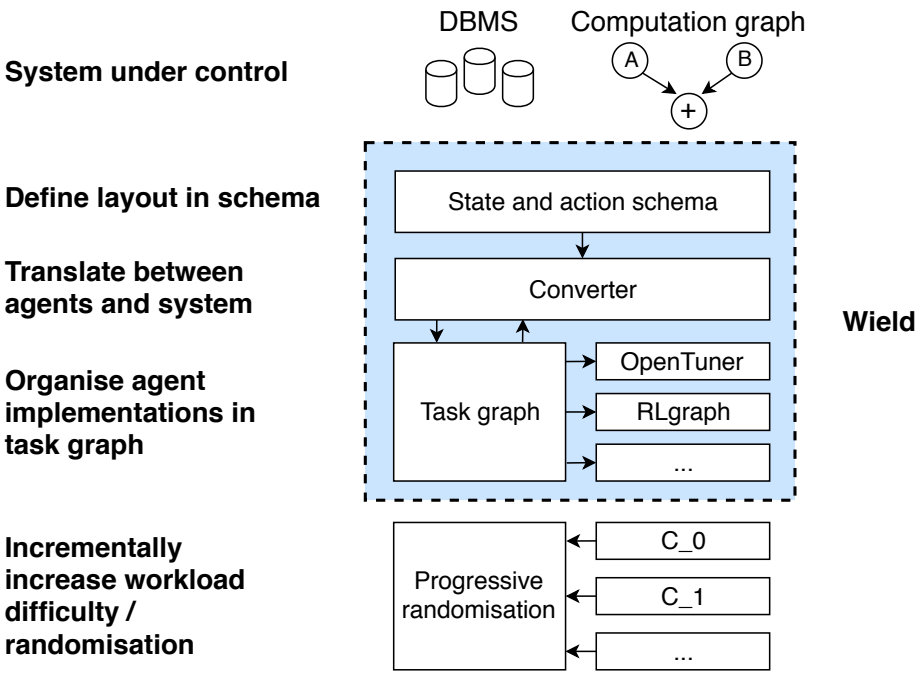


Figure 5.1: Conceptual overview of Wield in relation to existing auto-tuners, reinforcement learning frameworks like RLgraph, and systems.

implementation exposing a task interface). The highest level abstraction in Wield are *workflows* which coordinate execution of online (interacting with a system) or offline (with historical datasets) training, evaluation, and serialisation by invoking Wield's individual abstractions.

Models use task graphs to describe hierarchies of tasks wherein a single node may be a single RLgraph graph, or a blackbox optimisation, or a supervised learning task. Tasks use converters to map between agent and system view of data, and schemas to standardise programmatic layouts of inputs, reward, and actions. By separating task design and system-specifics, task architectures can be used across similar systems or problem structures which only differ in the system interface (e.g. different databases with distinct query languages but otherwise similar query processing). Finally, imperfect demonstrations and data sets can be used in data augmentation worfklows to manage or generate off-policy training data.

5.4 Task design abstractions

Wield's task abstractions unify common workflow streams via standardised physical layouts. Layout refers to identifying the Markov Decision Process representation of an optimisation problem, i.e. the concrete dimensions, data types and processing steps for all inputs and outputs of an RL agent. Specifically, Wield uses:

- **Schemas** as centralised task descriptions shared by all worfklows. Unlike simulators, decision processes extracted from systems often have programmatic data layouts which vary depending on deployment details.

- **Converters** which allow applications to convert between agent and system representations of a decision process.

- **Task graphs** which allow users to decompose tasks into collections of DAGs.

In this section, I motivate these abstractions and explain how they can be used to express systems tasks.

5.4.1 Designing states and actions with task schemas

Schemas are motivated by the observation that states for systems problems need to be designed. They help structure this design. In contrast, game simulators like ALE have fixed state dimensions across games. All methods can rely on a fixed base representation (i.e. the game frame) for reproducible and comparable experiments. Wield schemas encapsulate input-dependent state and action layout construction. Input-dependent here refers to states where layout may depend on the specific problem instance.

Consider an RL model for a query processing task for databases. States may contain tokenised versions of queries with tokens mapped to integer encodings, i.e. each query operator and column name maps to an integer. The input vocabulary depends on the database schema due to variable numbers of columns and tables, and on the query language used for its query operators. In combination with the current workload (e.g. a query instance), states contain relevant aspects of the current system configuration, e.g. capacity of task queues, resource usage statistics, previous decisions on dependent tasks. Different input types and layouts (i.e. dimensions of state arrays) are required per deployment. Wield schemas allow developers to express a state layout as a function of system parameters.

States can also encode bias towards decision horizons. For example, Tesauro et al. describe a choice of state encoding in the context of server resource allocation via a discretised mean request arrival rate [TJDB06]. Their state includes both the current mean arrival rate and the one from the prior observation interval to relate the impact of actions to arrival rate. In workload management tasks, the workload generating process is generally unknown, and future workloads (e.g. request rates or job size) may be independent or correlated to current decisions (c.f. Mao et al.'s discussion on this regarding value function estimates [MVSA18]). State features and preprocessing, e.g. temporal smoothing, must encode such assumptions.

In summary, state design for RL in is an iterative process which differs from feature design for supervised learning as the state must also capture transition dynamics. To help researchers explore, compare and version state designs, Wield standardises them through schemas. Environments implementing the gym interface normally include both problem dynamics and task layouts. In Wield, I argue that these should be separated.

Compact feature representations from high-dimensional inputs can also be generated by using e.g. a variational auto-encoder [KW14] which learns a latent representation in an unsupervised manner. Separating feature representation from prediction and control by training the latter on a learned compact representation can drastically improve training performance [HS18]. This separation of concerns does not relieve systems designers from selecting which system metrics to incorporate.

Similar to state design, action structure must be designed manually as agent outputs need to be translated to structured system calls, e.g. by generating a special query to update the state of a database. Simple action representations include single binary or categorical decisions where an action selects one of a small number of resources or task slots, e.g. which task to schedule next from a task queue, or which device to assign to an operation on a single node.

The term "action structure" refers to interpreting the outputs of a neural network. For example, in Q-learning, a neural network used to represent the Q-function is designed by creating a final action selection layer with one neuron per possible integer action. The outputs are interpreted as Q-values. To output multiple distinct actions, multiple of such action layers may be created. RL practitioners must explicitly consider how decision problems can be mapped to convenient (i.e. as few distinct actions as possible) action representations.

Small-scale single-task action structures are popular in experimental applications as they correspond to well known RL problems and implementations such as Atari games, where a single integer action is selected. However, such representations may not scale to larger problem instances if the number of actions directly corresponds to problem size. Consider a task in the database domain which selects single columns in a database table (e.g. for index creation, join order, ..). The number of columns in a normalised database design is typically a small integer ($n < 100$) which facilitates a direct integer-action mapping. Such designs are prevalent across prior work in systems-RL [SSD, MNA17]. Scaling this to compound indices spanning multiple columns exponentially increases number of actions (e.g. combining 3 of 15 columns results in 3375 discrete actions), and in turn the experience required to explore them.

Similarly, a cluster scheduler trained to schedule a handful of resources will not scale to data-center scale with tens of thousands of resources where each resource represents an action. In the absence of a prior, a policy would have to learn from scratch the similarity between resources (e.g. similar performance between the same hardware configuration of different nodes in a rack). This would require an impractical number of samples to explore action combinations in large discrete action spaces. Large discrete action spaces may require task decomposition. If similarity metrics between actions can be defined in advance, actions can also be selected in a multi-stage approach whereby first an action in a promising region of the action space is selected, and a nearest-neighbour lookup is subsequently performed to identify a fitting local action [DAEvH$^+$15]. In §5.7, I describe a combinatorial optimisation problem where such an approach is not possible as no covariance function can be given ahead of time.

Listing 5.1 illustrates a simplified single-task schema. The example illustrates how system-specific configurations (here database operators and database schema) are used to define layouts for states and actions. Observe that this schema describes a query processing task but is independent of the query language, as operators are configurable parameters.

In summary, task schemas conceptually define physical layouts of states and actions based on problem parameters. In practice, task schemas in Wield describe input spaces which can then be translated to RLgraph in order to generate a compatible computation graph.

5.4.2 Converters

Converters are adapters which express how system metrics, configuration parameters, and query languages or custom protocols correspond to numerical representations within an optimisation. A schema specifying a layout can be used by different converters, and a converter may work with different schemas. Schemas constrain how decision model is encoded structurally (layout), converters specify how this encoding is achieved from raw system information (content). Listing 5.2 shows the conversion API provided by Wield.

```
class CombinatorialSchema(Schema):
  def __init__(self, schema_spec):
     # SELECT, FROM, AND, LIKE, IN, ..
     self.selection_ops = schema_spec['selections']
     # SORT, GROUP_BY, MAX, ..
     self.aggregation_ops = schema_spec['aggregations']
     self.tables = schema_spec['tables']
     self.max_query_len = schema_spec['max_query_len']
     # Num of colums to select per step.
     self.max_action_colums = schema_spec['max_action_colums']
     self._num_cols = 0
     self._build_inputs()
     self._build_outputs()

  def _build_inputs():
    """Represents  tokenised queries as states."""
    # Build input vocabulary
    vocab = {}
    idx = 0
    for op in self.selection_ops:
      vocab[op] = idx
      idx +=1
    for op in self.aggregation_ops:
          vocab[op] = idx
          idx +=1
    for table in self.tables:
      for col in table['columns']
            vocab[col] = idx
            idx +=1
        self._num_cols +=1
     self.states_spec = IntBox(low=0, high=idx,
       shape=(self.max_query_len,))

  def _build_outputs():
    """Selects a set of columns per step."""
    num_actions = 1 + math.pow(self._num_cols,  self.max_action_colums)
    self.actions_spec = IntBox(low=0, high=num_actions)

  @property
  def actions_spec(self):
    return self._actions_spec

@property
  def states_spec(self):
    return self._states_spec
```

Listing 5.1: Simplified task-schema defining state and action layout based on deployment-specific parameters in a database task

```
def system_to_agent_state(system_state)

def system_to_agent_action(system_action)

def agent_to_system_action(agent_action)

def system_to_agent_reward(system_metrics)
```

Listing 5.2: Wield converter API to translate between agent and system views.

Unused conversions such as agent-to-system reward conversion are omitted. System-to-agent action conversion is used to map demonstrator actions to numerical representations (§5.6). Converters use layouts provided by schemas to implement the conversion API. I give an implementation example in §5.7 when discussing a concrete case study.

5.4.3 Task architectures

Schemas and converters help decouple system-specifics from task representation in RL for *individual* tasks. Next, I introduce Wield's task graph abstraction which helps break down problems into separate decisions. Task graphs organise tasks into task architectures with two categories:

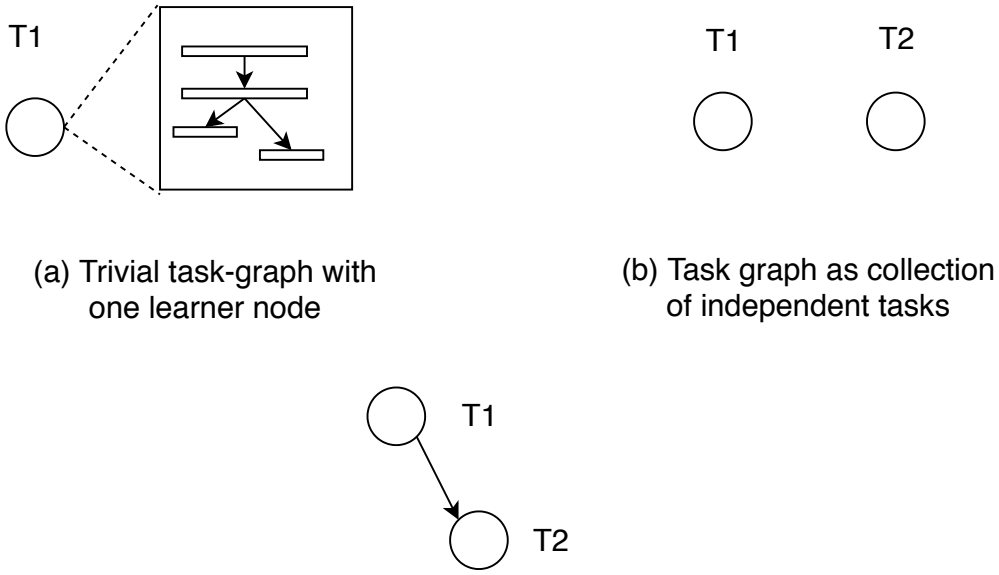
1. **Shared-parameter tasks** are multi-task architectures where a single end-to-end differentiable architecture has multiple task output networks which each emit separate actions per step. For example, in my indexing case study §5.7, a single neural network outputs multiple separate indexing keys to form a compound index.

2. **Independent tasks** are task architectures where separate learners focus on different sub-tasks, e.g. in the case of hierarchical decomposition or parallel independent tasks.

Shared-parameter tasks can be realised in Wield by making use of RLgraph's branching task mechanism. Branching here refers to a common practice of utilising multiple policy or Q-networks on top of a single shared neural network learning a problem representation. Users define a set of named actions and corresponding spaces, and RLgraph automatically creates the corresponding multi-task shared parameter architecture on top of a shared value function or policy. The corresponding task graph only has a single vertex, encapsulating a single learner.

Non-trivial task graphs occur through hierarchical and independent task decomposition (Figure 5.2). Hierarchical task decomposition refers to tasks organised as directed acyclic graphs where outputs from single tasks (vertices) are used as input states (edges) to other task vertices. Independent tasks refer to a scenario where multiple learners interact with an environment, possibly learning at different time scales and schedules.

Hierarchical reinforcement learning has been studied in a variety of contexts with the most well known approach being the options framework [SPS99]. There, a top-level policy chooses between different sub-policies (options) to execute over a time-frame (until the sub-task terminates). A large body of work in this domain exists on information sharing, automated goal discovery, and task transfer [BL14, VOS+17, BHP17, NGLL18]. In Wield,



(a) Trivial task-graph with one learner node

(b) Task graph as collection of independent tasks

(c) Task graph with dependent tasks, T2 act is invoked with T1 outputs

Figure 5.2: Basic task architectures. In (a), a single task node contains one differentiable multi-task architecture with a shared network. (b) refers to a task graph simply collecting multiple independent learner instances with no communication or interaction. In (c), a hierarchical arrangement uses two task nodes where task T2 can only act based on decisions made by T1.

I focus on workflows where a systems designer manually identifies sub-tasks as a means of encoding domain knowledge.

Hierarchical designs to organise resources at different granularities are also a core element of systems research (e.g. cache hierarchies, hierarchical scheduling). However, hierarchical RL has found limited attention in the systems community as a means to manage large state and action spaces. Mirhoseini et al. first utilised hierarchical RL for TF device placement [MGP+18]. In their work on device placement for computational graphs, the action space of available devices is a small as operations are placed on a single node, thus choosing between (an aggregate) CPU device and one or more GPUs. There can however be tens of thousands of operations in a single graph, and learning device interactions on the level of individual numerical operations creates a impractically large space of state combinations. The hierarchical scheme used by the authors first computes a grouping where each operation is classified into a placement group by a grouper task. A placer task learns to assign each group to a device.

One source for the limited utilisation of hierarchical approaches may be a lack of software tooling. In my survey of RL in systems (§2.4), a majority of authors utilised open source implementations such as OpenAI baselines [DHK+17]. As these popular libraries focus on single-agent scenarios with rigid control flow assumptions, hierarchical designs are not supported.

Task graphs in Wield simplify factorisation of tasks into different sub-tasks which may train and act jointly, or at different time-scales. Task objects primarily encapsulate distinct RLgraph agents or any other optimisation implementing gym-style interfaces. Hierarchical tasks often require to transform the output of one task before inputting it to a subsequent task, e.g. by enriching it with additional environment information or preparing a specific

input format. Nodes in a task graph hence further encapsulate pre-and post-processing for each sub-task. Edges in the graph are implicitly created by creating one task as a sub-task of another task in the same task-graph. When performing inference, task outputs are routed through the task graph based on user-defined directed edges between tasks, and the results of all tasks during execution are returned. In Chapter 6, I illustrate how task graphs can be used in practice to design hierarchical tasks.

5.5 Task evaluation protocols

5.5.1 The case for workload randomisation

A key obstacle when assessing model capabilities is the use of fixed workloads. As I discussed in §2.4, in domains like database management or query processing, benchmarks often focus on narrow application scenarios with small query sets (e.g. TPC-C [Tra10]). Other clients such as YCSB focus on read and write throughput for key values stores [CST+10] but only include high-level read and scan scenarios. Leis et al. proposed the Join Order Benchmark which contains 113 queries specifically designed to investigate join estimation capabilities in query optimisers [LGM+15]. While hand-designed workloads can highlight particular weaknesses or strengths of a system, they nevertheless are prone to over-fitting small test sets. Researchers may be misled to design feature representations sensitive to particular properties of hand-picked test-sets.

I argue that the design of RL applications can benefit from synthetic workload mechanisms with configurable task difficulty as a means to understand training and test-time behaviour. In both my evaluation of RLgraph and my motivation for incremental testing in RL, I described the impact of random seeds when evaluating training of a DQN variant on an ALE task (§4.3), a topic which has also gained attention in the RL community [HIB+17, MGR18].

To reason about non-determinism when evaluating stochastic optimisation mechanisms, researchers must delineate deterministic and non-deterministic elements in their workload and optimisation procedure. In Wield, I construct workloads from the perspective of changing between several evaluation and randomisation modes. I distinguish between blackbox and generalisation mode from the perspective of workload generation. In blackbox-mode, a single workload instance (e.g. a single set of queries or jobs) is generated, and a model is trained and evaluated on that same instance. In generalisation mode, training is executed on different instances than the ones used in the final evaluation.

Both modes can be executed with varying levels of randomisation. Workload determinism refers to deterministic behaviour of task instances. Training determinism refers to deterministic initialisation and sampling during training. For example, in black-box mode the generation of the single task instance and the training initialisation can both be deterministic. Similarly, in generalisation, both the instances used during training and the final test instances can be randomly generated or held fixed. This invites problematic practices such as cherry-picking and presenting only successful combinations of workloads and weight initialisation values.

In the RL literature, all combinations of blackbox and generalisation modes can be found. Comparing results is difficult if authors do not to report which workload elements are held fixed or are subject to randomisation. Researchers in computer systems have thus far not identified shared sets of evaluation protocols towards randomisation issues.

Class	Randomisation implementation	Example use cases
$\mathbf{C_0}$	Fixed blackbox task, Fixed optimisation parameters	Iterate representation until sufficient
$\mathbf{C_1}$	Fixed blackbox task, Random optimisation parameters	Model sensitivity to weight initialisation
$\mathbf{C_2}$	Randomised blackbox task, Random optimisation parameters	Model sensitivity to task parameters
$\mathbf{C_3}$	Fixed in-distribution generalisation, Random optimisation parameters	Understand sample requirements
$\mathbf{C_4}$	Randomised in-distribution generalisation, Randomised optimisation parameters	Production use in controlled environments
$\mathbf{C_5}$	Fixed out-of-distribution generalisation, Randomised optimisation parameters	Robustness against unforeseen inputs
$\mathbf{C_6}$	Randomised out-of-distribution generalisation, Randomised optimisation parameters	Production use without customisation

Table 5.1: Progressive randomisation protocol overview. Each class specifies a different level of non-determinism.

5.5.2 Progressive randomisation

I propose an evaluation protocol called *progressive randomisation*. Progressive randomisation incrementally and explicitly increases dimensions of non-determinism.

The protocol is based on the observation that different randomisation modes can serve different phases of design. For example, holding a workload fixed to study robustness against random initialisation is valuable when a designer is uncertain if a model design can solve a task at all. Conversely, using a fixed optimisation can be useful to study the impact of workload parameters on optimisation outcomes. Evaluation difficulties are not inherent to a specific mode of randomisation or evaluation. They arise when conflating sources of performance variation or misinterpreting model capabilities.

In supervised learning, projects such as DAWNBench [CKN+18] have suggested metrics like time-to-accuracy to compare model designs and hardware choices to understand trade-offs in deep learning systems. In contrast, shared deep RL tasks such as the Malmo Minecraft challenge [JHHB16] or Unity agents [JBV+18b] are focused on task performance in simulated worlds where randomisation is *incidental*. That is, tasks may include some degree of randomisation and generalisation but these are not varied to analyse their contribution to agent performance (or lack thereof). Task variation in these scenarios is further constrained by experimental cost. Bsuite [ODH+] is a novel benchmark for analysing agent behaviour which varies random seeds to score agent performance but which does not distinguish different generalisation modes or randomised tasks.

Table 5.1 lists the different evaluation modes in the protocol and their purpose. It also lists example applications. Fixed optimisation parameters in practice refer to the random weight initialisation strategies in neural networks, and further to the random seed used when sampling mini-batches for stochastic gradient descent as well as policy decisions.

Fixed blackbox refers to always training on the same task, while fixed generalisation refers to a fixed test task. Randomised generalisation implies that for each reported experiment result, a new test task was generated.

Intentionally, not all possible combinations of non-determinism are present in the protocol. Fixed optimisation parameters are initially useful to produce repeatable results and debug non-optimisation components of a task (C_0). For subsequent design concerns, they should be randomised as results from a fixed optimisation seed yield little insight about wider model usability (avoiding 'lucky' seeds).

Generalisation semantics are complicated by task-specific concerns. Consider fixed in-distribution generalisation (C_3). "In-distribution" refers to workload assumptions where the test task is taken from the same distribution training tasks were generated from. For example, a database task may be evaluated on different sets of queries while a scheduler task may receive held out jobs. A problem with fixed or randomised generalisation tasks is that there are typically no useful measures of how different test tasks are from training examples. More specifically, for randomised training tasks it is rarely reported if parts or the entirety of a test task could be seen during training.

Nonetheless, the description of a model to e.g. be in C_3 for a certain task gives useful indication of expected behaviour. Here, I refer to being in a class as to meeting application-specific performance objectives under the given randomisation assumptions. For example, a model in C_2 which meets randomised blackbox objectives can be used as a direct search tool in practice without requiring to retune hyper-parameters, whereas a model in C_1 tuned for a fixed blackbox objective is customised to a single deployment or task context. Distinguishing model classes helps set expectations and allows researchers to effectively communicate evaluation designs.

Generalisation concerns in deep reinforcement learning are poorly understood. They are not captured analytically but rather empirically per task. A model may be in different classes depending on the number the samples is trained on. Even for the same hyper-parameters, a substantial fraction of random weight initialisation and optimisation seeds may fail. I illustrate this by applying progressive randomisation when reproducing prior work in Chapter 6.

I propose to further describe models based on these empirical properties. Specifically, I propose to include

- the number of state transitions experienced during training $\mathbf{n}$,
- the number of random seeds used for weight initialisation and optimisations $\mathbf{s}$,
- and the observed frequency f where learning objectives were achieved.

For example, a model may be described as $C_1(n = 10^7, s = 10, f = 0.4)$ to communicate empirical success when experiencing 10 million state transitions and trying 10 different random seeds. In the following notation, I may omit s and f from notation when only discussing sample count or class membership. Communicating success rates is especially important when considering the cost of training a model against its claimed performance improvements.

As sample collection cost varies drastically between tasks, conditioning class membership on sample size is useful for estimates on model transfer on tasks with different sample collection cost. For example, the same model may be in $C_1(n = 10^4)$ but in $C_3(n = 10^7)$

as robustness to inputs increases with number of states observed, or it may increase its success frequency.

In summary, progressive randomisation encourages shared understanding of model capabilities across problem domains. Several dimensions regarding model scale, the cost of featurisation, and other hidden cost are not captured by the protocol. The protocol also does not replace standard considerations on experiment design or statistical analysis. Reaching a training objective may be an ill-defined term inviting moving of goal-posts for many applications. Optimising performance of a data processing system may be a best-effort consideration, unless a service level objective is defined ahead of experimentation. The classification system is intentionally simple to serve as a low-overhead summary of design assumptions.

While only a starting point, progressive randomisation constitutes the first explicit evaluation protocol for deep reinforcement learning focused on delineating workload randomisation. I provide an example of using it as an instructive protocol in Chapter 6.

5.5.3 Prior work viewed through progressive randomisation

An experimental classification protocol can be both instructive (guiding experimentation) and descriptive (helping compare and qualify prior work). To illustrate its utility as an analytical tool, I use progressive randomisation as a lens on selected prior work in research and applied RL. Table 5.2 classifies prior work. For example, the evaluation of RLgraph's implementations on ALE pong concerned fixed blackbox scenarios with randomised optimisation parameters. I discussed learning success against different random seeds in §4.3.3.

The classification immediately illustrates problem progress. For example, in the device placement problem, Mirhoseini et al.'s initial work [MPL+17] with manual operation grouping required orders of magnitudes more samples than their subsequent work using a hierarchical approach [MGP+18]. Both operated in a fixed blackbox setting. Addanki's et al.'s [AVG+19] and Paliwal et al's recent work [PGN+19] utilising graph neural networks then illustrates progress towards generalisation through permutation-invariant representations. I found subtle differences in evaluation randomisation which can be made make explicit through progressive randomisation. For example, Addanki et al. generate random variations of computation graphs for training and testing, but both sets are fixed (C_3).

A similar progress pattern can be observed in database tasks. In their first work on join order enumeration, Marcus et al. [MP18] used a policy optimisation method on a fixed set of training and test queries, the Join Order Benchmark (JOB) [LGM+15]. Training with randomised optimisation parameters yields $C_3(n \approx 10^5)$. In subsequent work, Marcus et al. proposed a learned query optimiser [MNM+19] which they evaluate on several tasks, including a fixed set of out-of-distribution queries ($C_5(n \approx 10^4 - 10^6, s = 50, f = 1.0)$, results averaged across seeds). Training workloads in database applications were often generated by augmenting fixed existing query sets (TPC-H, IMDB). It would be desirable for the systems-RL community to develop shared standards on training and test randomisation.

Many approaches do not report explicitly how workload randomisation and optimisation parameters were selected which makes classification difficult. If a fixed task is presented without reporting number of training trials, seeds, or randomisation assumptions (i.e. a potentially cherry picked single random seed), I assume C_0. Few works I surveyed explicitly report on failure modes, despite often using appendices to share training hyper-parameters.

This highlights the need for more explicit evaluation protocols. In Chapter 6, I also attempt to reproduce and classify a result listed here to assess published claims.

Work	Objective	Fixed objective	Highest class reported $C_i(n)$
Neural packet classification [LZJS19]	Classification time, memory	no	$C_0(n = 10^7, s =?, f =?)$
Deep reinforcement learning for for join order enumeration [MP18]	Query execution time	no	$C_1(n \approx 10^4, s =?, f =?)$
TF device placement [MPL+17]	Gradient descent iteration time	no	$C_1(n = 2 \times 10^5, s = 4, f = 1.0)^*$
Hierarchical TF device placement [MGP+18]	Gradient descent iteration time	no	$C_1(n = 10^3, s =?, f =?)$
RLgraph training evaluation	ALE Pong score threshold	yes	$C_1(n = 5 \times 10^7, s = 20, f = 0.75)$
Device placement [AVG+19]	Gradient descent iteration time	no	$C_3(n = 10^3 - 10^5, s =?, f =?)$
Language to program [GPLL17]	Program generation	yes	$C_3(n = 13,000, s = 5, f = 1.0)^*$
Cardinality predictions [OBGK18]	Prediction error	no	$C_3(n \approx 10^5 - 10^6, s =?, f =?)$
Learning scheduling algorithms for data processing clusters (Decima) [MSV+18b]	Spark job completion times	no	$C_4(n = 10^8, s =?, f =?)$
Congestion control [JRG+19]	Throughput, latency	no	$C_4(n \approx 10^6 - 10^7, s =?, f =?)$
Neo: A Learned Query Optimizer [MNM+19]	Query execution time	no	$C_5(n \approx 10^4 - 10^6, s = 50, f = 1.0)^*$
Computation graph rewriting [PGN+19]	Memory usage	no	$C_5(n = 400000, s = 20)$
AlphaZero [SHS+18]	Win game of chess	yes	$C_6(n \approx 10^9, s =?, f =?)$

Table 5.2: Prior work classified. A * indicates results being reported as the median or mean across random seeds. If a range is given without $\approx$, this refers to results being reported on multiple datasets at different sample counts.

Finally, applications reviewed here were not typically defined through a fixed objective such as winning a game or reaching a score threshold. Performance objectives were explorative, e.g. outperforming problem-specific baselines. This can obfuscate practical utility without cost-benefit analysis on implementation cost.

5.6 Data augmentation from demonstrations

Systems developers utilise their domain expertise to encode a state representation, action model, and reward structure. To interface RL algorithms, systems designers inherently must encode prior knowledge on workloads.

Another way of encoding domain knowledge is through supervision. In supervised machine learning, training labels for widely used benchmarks were created through hand-curation [DDS+09], or more recently tool-assisted through weak supervision [RBE+17]. Demonstrations are also a common tool in RL in scenarios where an expert can without difficulty identify and perform a correct action (§5.6.3). In Wield, I explore the potential of rule-based weak supervision to enrich trajectory data based on domain knowledge. This is motivated by human users generally not being able to identify optimal solutions by intuition (i.e. solutions to high-dimensional optimisation problems). I first discuss an algorithmic variant for reinforcement learning from demonstration.

5.6.1 Algorithms

I briefly summarise Deep-Q learning from demonstrations DQFD, a DQN variant developed by Hester et al. for real world applications of DQN [HVP+17]. They argue that in problems where a simulator may not be available, prior control data from expert configurations may be used to pretrain to a safe initial model. Their algorithm is based on the observation that training a Deep Q-network using an unmodified DQN loss function creates the problem of "ungrounded" Q-values. For many state action pairs, no demonstration data is available as the expert demonstration has no reason to choose actions believed to be sub-optimal. This in turn means when training the Q-function, no realistic value estimates for these state-action pairs exist, and expert updates would update towards the highest ungrounded value (due to choosing the highest Q-value). Non-demonstrated action values would have unspecified Q-values in relation to the expert action, leading to unpredictable behaviour when training online and updating Q-values from new observations.

Hester et al. hence suggest to combine DQN variants with a supervised loss term which specifies how expert actions and non-expert actions differ in Q-values. This is achieved by assigning an 'expert margin' to demonstration actions by extending double Q-learning [HGS16], a Q-learning variant which corrects biased Q- estimates in the original DQN by decoupling action selection from action evaluation. The double DQN loss

$$J_{DQ}(Q) = (R(s,a) + \gamma Q(s_{t+1}, a_{t+1}^{max}; \theta') - Q(s,a;\theta))^2 \tag{5.1}$$

where

$$a_{t+1}^{max} = argmax_a Q(s_{t+1}, a; \theta) \tag{5.2}$$

uses the target network (as explained in §2.2, parametrised by θ')) to evaluate the action selected using the training network (parametrised by θ). DQfD combines this loss with

another expert loss function J_E:

$$J_E(Q) = \max_{a \in A}[Q(s,a) + l(s, a_E, a, m_E)] - Q(s, a_E) \tag{5.3}$$

$l(s, a_E, a)$ is called a large-margin function which outputs 0 for the expert action m_E, and a margin value > 0 otherwise. By adding the expert margin m_E to the loss of Q-values of *incorrect actions*, the update biases the Q-function towards the expert actions. A difference in Q-values between expert actions and other actions of at least the margin is enforced [PGP14] for each state.

In DQfD, the large margin function evaluates to a single fixed positive margin (e.g. 1.0) or zero. The main advantage of DQfD its conceptual simplicity, as the margin value provides a single hyper-parameter to express expert confidence. This comes at the cost of assuming all expert actions are equally better than other actions. It also does not allow for negative examples to be combined with positive examples.

I propose to modify DQfD to not use a single margin value m_e but a margin matrix M_e. M_e is constructed from a vector of margin values per sample, i.e. each state-action pair (s_i, a_i) can have a unique margin value m_{E_i}. Second, I observe that in this variant, the margin functions can also be used for negative examples. More generally, multiple demonstrations can be given for the same pair (s_i, a_i). This requires to modify the expert loss J_E to compute element-wise losses using minimisation or maximisation for positive and negative margins respectively for each element i in a mini-batch:

$$J_E(Q) = \begin{cases} \max_{a \in A}[Q(s,a) + l(s, a_E, a, m_{E_i})] - Q(s, a_E) & m_{E_i} \geq 0 \\ \min_{a \in A}[Q(s,a) + l(s, a_E, a, m_{E_i})] - Q(s, a_E) & m_{E_i} < 0 \end{cases} \tag{5.4}$$

This distinction is needed to ensure the Q-value of non-expert actions is directionally adjusted by taking the maximum or minimum estimate including the margin respectively as the expert loss (i.e. forcing all other actions to be worse or better than the demonstration).

Requiring a demonstrator to assign individual margins is not a practical strategy for the sample sizes required in reinforcement learning. In Wield, I provide demonstration abstractions to facilitate both positive and negative demonstrations. Instead of providing unique margins per sample, I propose to partition demonstrations into equivalence classes via labelling functions, which I introduce in the next section.

5.6.2 Demonstration abstractions

Demonstration abstractions are used to enrich incomplete trajectories. This is based on the observation that system trace data and logs cannot generally be assumed to contain complete trajectory information which could directly be used for off-policy training. For data augmentation, I distinguish the following cases:

1. **Full trajectories** Full trajectories implies that states, actions, rewards can be directly extracted from traces. They can be available when a system is profiled for debugging and analysis purposes, or when a deployment is explicitly instrumented for later optimisation. For example, databases may write full information on all queries where execution time crosses a pre-defined threshold (i.e. a slow query log), but not for all executed queries due to storage requirements.

2. **States and actions** Workload and configuration information is available to construct a state, and system decisions to infer actions. No reward is available because the relevant system metrics were not collected or cannot be matched to states and actions effectively.

3. **States only** Only workload information is available but no results of a system interacting with inputs. For example, in the context of databases, only example queries may be available, or input job descriptions for scheduling problems.

Other combinations are possible, e.g. logs only containing systems decisions (actions) but not workload inputs. Inverse reinforcement learning addresses the problem of recovering a reward function from states and actions to subsequently construct a policy [NR00]. Inverse reinforcement learning is beyond the scope of this dissertation because system performance objectives are typically known (e.g. throughput and latency service level objectives).

Due to a lack of available datasets, I focus on the third case where only workload information is available. As systems designers adopt more fine-grained decision making mechanisms, I expect wider adoption of trajectory data collection.

A labelling function $f : s \rightarrow (a, r)$ maps a state s to an action a and reward r. The state s must consider the context of prior decisions for labelling functions to be able to construct a state sequence., i.e. the state must satisfy the Markov property. This is in contrast to supervised labelling where individual inputs can be labelled without consideration for other data points (as supported by tools like Snorkel [RBE+17]), as they are evaluated independently.

Consider a workload trace consisting of a set of database queries, or a list of job descriptions to be processed by a scheduler. To relate system configuration to workload, the state representation should combine relevant configuration parameters and current workload information. In Wield, I iteratively construct demonstration trajectories from raw workload traces represented as a sequence of workload items $W = w_0, w_1, ..w_n$ by the following steps:

1. The system state is initialised with the initial configuration c_0.

2. A state s_0 is generated using a converter which combines w_0 and c_0 according to its conversion function.

3. A labelling function produces $a_0, r_0 = f(s_0)$.

4. An updated configuration representation c_1 is produced from a_0.

5. Repeat until all w_i are processed.

This procedure can be repeated for arbitrarily many labelling functions to generate multiple trajectories from a single workload sequence W. I give a practical example of creating demonstrations using labelling functions in Section 5.7.

5.6.3 Alternative approaches to learning from demonstrations

Learning from demonstration or expert supervision is a long-studied topic of interest in RL. For example, DAGGER (for Dataset Aggregation) is an approach for imitation learning which requires an expert to continuously provide new input [RGB11]. Interactive

approaches like DAGGER more closely mirror the workflow of interactive tuning tools used e.g. in databases. While Wield could be extended to support interactive demonstrations, I believe interactive approaches to be impractical for scalable systems tuning. First, additional computational resources are cheaper than expert time to guide training. Second, unlike many perceptual tasks in computer vision or natural language understanding, humans cannot easily give accurate demonstrations for increasingly complex systems. A human presented with an intermediate result may not be able to provide useful feedback.

Behavioural cloning fits expert trajectories as a supervised learning problem, and was more recently also studied in the context of generative adversarial models [WMR+17, HE16]. Behavioural cloning as a simple (i.e. no further data generation mechanisms) supervised problem is limited by requiring large amounts of expert trajectories and poor generalisation to out-of-sample inputs.

The demonstration methods used in Wield are also conceptually limited. Their optimisation mechanisms are brute-force overrides of value estimates. Their purpose is to demonstrate how expert knowledge can be systematically integrated into reinforcement learning design workflows. Emerging work is concerned with helping learners assign confidence into expert demonstrations [WCB+19].

5.7 Case study: database indexing

Next, I motivate an application example and discuss its representative problem properties.

5.7.1 The compound indexing problem

In database management, a key task for database administrators is to create secondary indices for stored data. Such index data structures can accelerate query execution times by orders of magnitudes. This is achieved by providing fast look-ups for specific query operators such as range comparisons (B-trees) or exist queries (Bloom filters). A single index can span multiple attributes, and query planners employ a wide range of heuristics to combine existing indices at runtime, e.g. by partial evaluation of a compound (multi-attribute) index. The compound indexing can thus be described as follows.

For a fixed set or distribution of database queries, one must identify the combination of indices which minimizes overall workload latency and memory usage of indices. Both objectives can be phrased as constraints, e.g. meeting a service level objective on a latency percentile, while not using more memory than is provisioned to the database server.

Determining optimal indices is an attractive optimisation problem due to non-intuitive, data-driven behaviour by the query planner. If an ineffective index is chosen, performance may degrade due to unnecessary index lookups, thus creating more indices does not equal faster queries. Instead, index performance depends on attribute cardinality and query distribution. This creates an interesting opportunity to use RL to adapt indices based on query execution feedback. Sharma et al. previously used RL on a simplified single-column indexing problem [SSD]. I study the full problem where indices may span multiple attributes.

In practice, indices are currently identified using various techniques ranging from offline tool-assisted analysis [ACK+04, DDD+04] to online and adaptive indexing strategies [GK10, IMKG11, HIKY12, PIM15]. Managed database-as-a-service (DBaaS) offerings

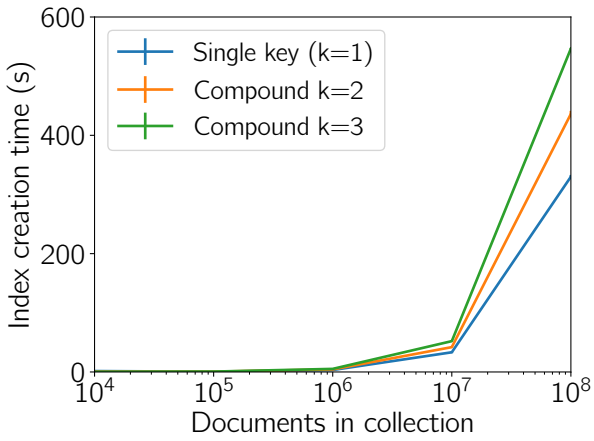


Figure 5.3: Index creation times as a function of document size and the number of attributes k spanned by an index.

sometimes offer a hybrid approach where indices for individual attributes are automatically created but users need to manually create compound indices.

I study MongoDB as a popular open source document database where data is organised as nested J/BSON documents. While a large body of work exists on adaptive indexing strategies for relational databases and columnar stores [PIM15], compound indexing in document databases has received less attention. Index selection is a representative problem for systems-RL because:

- Evaluation times scale unfavourably with problem scale. Index creation time increases with database size. I illustrate this by creating synthetic compound indices of different index and collection sizes in MongoDB (Figure 5.3).
- The problem is partially solvable as some indices may be correct while others are not. This is attractive for evaluating training progress, as opposed to all-or-nothing problems.
- Combinatorial selection where similar indices (e.g. covering similar fields) may not yield similar performance, i.e. a covariance function is not known and cannot easily be specified.

Combinatorial optimisation problems are common across a wide range of system optimisation problems such as compiler optimisation, device assignment, join order evaluation, or job scheduling.

Document databases are offered by all major cloud service providers, e.g. Microsoft's Azure CosmosDB offers native MongoDB support [Mic18], Amazon's AWS offers DynamoDB [Ama18], and Google Cloud provides Cloud Datastore [Goo18]. The document database services I surveyed offer varying specialised query operators, index design, and query planners using different indexing heuristics. The aim of automatic index selection is to omit this operational task from service users. I initially focus on common query operators available in most query dialects. In MongoDB, queries themselves are nested documents:

$$db.find(\$and : [\{"name" : \{"\$eq : "Jane"\}, \{"country" : \{"\$eq : "England"\}\}])$$

The MongoDB query planner uses a single index per query with the exception of $\$or$ expressions where each sub-expression can use a separate index. An index may span

between 1 and k document attributes. The index is specified via an ordered sequence of tuples $(f_1, s_1), .., (f_n, s_n)$ where each tuple consists of a field name f_i and a sort direction s_i (ascending or descending). At runtime, the optimiser employs a number of heuristics to determine the best index to use.

Via index intersection, the optimiser can partially utilise existing indices to resolve queries. For example, *prefix intersection* means that for any index sequence of length k, the optimiser can also use any ordered prefix of length $1..k-1$ to resolve queries which do not contain all k attributes in the full index. Consequently, while the tuple ordering of the index does not matter for individual queries, the number of indices for the entire query set can be significantly reduced if index creation considers potential prefix intersections with other queries. Similarly, sort-ordering in indices can be used to sort query results via sort intersection in case of matching sort patterns. For example, an index of the shape $[(f_1, ASC), (f_2, DESC)]$ can be used to sort ascending/descending and descending/ascending (i.e. inverted) sort patterns, but not ascending/ascending or descending/descending.

Creating the full compound index for every single query is not a viable solution because it can both impractically increase storage cost and also slow down execution as all indices need to be updated after each insert. Moreover, not every query may benefit from an index in the case of low selectivity attributes. If a query predicate requires to evaluate most documents in a collection (or rows in a table), sequentially reading can be faster as it requires fewer disk seeks than working through an index.

Here, I show how to model compound indexing as a reinforcement learning problem based on the idea that execution feedback from query and index creation can be used to find a minimal (w.r.t. index size) combination of indices meeting runtime objectives.

5.7.2 Designing a problem model with Wield

In this subsection, I discuss state, action and reward model for indexing. Identifying an effective index for a query requires knowledge of the query shape comprised of its operators and attributes. To leverage intersection, the state must also contain information on existing indices which could be used to (partially) evaluate a query. I model the state by combining

- a tokenised representation of the query and
- a matrix containing column selectivity estimates and a binary vector indicating 1 for query attributes present in the query and 0 for all others.

Selectivity estimates are obtained by querying distinct column values when beginning training. The representation is based on the observation that indexing decisions must consider query operator structure (e.g. "or" predicates can evaluated with separate indices) and also data distribution. Without encoding selectivity estimates, the learner would have to indirectly observe the effects of different cardinalities by memorising the effects of each operator on each specific column.

For each query, the agent must output an index (or none) spanning at most k attributes where k is a small integer. Indices covering more than 2-4 attributes are rare in practice. This is also because compound indices containing arrays, which require multi-key indices (each array element indexed separately), scale poorly and can slow down queries.

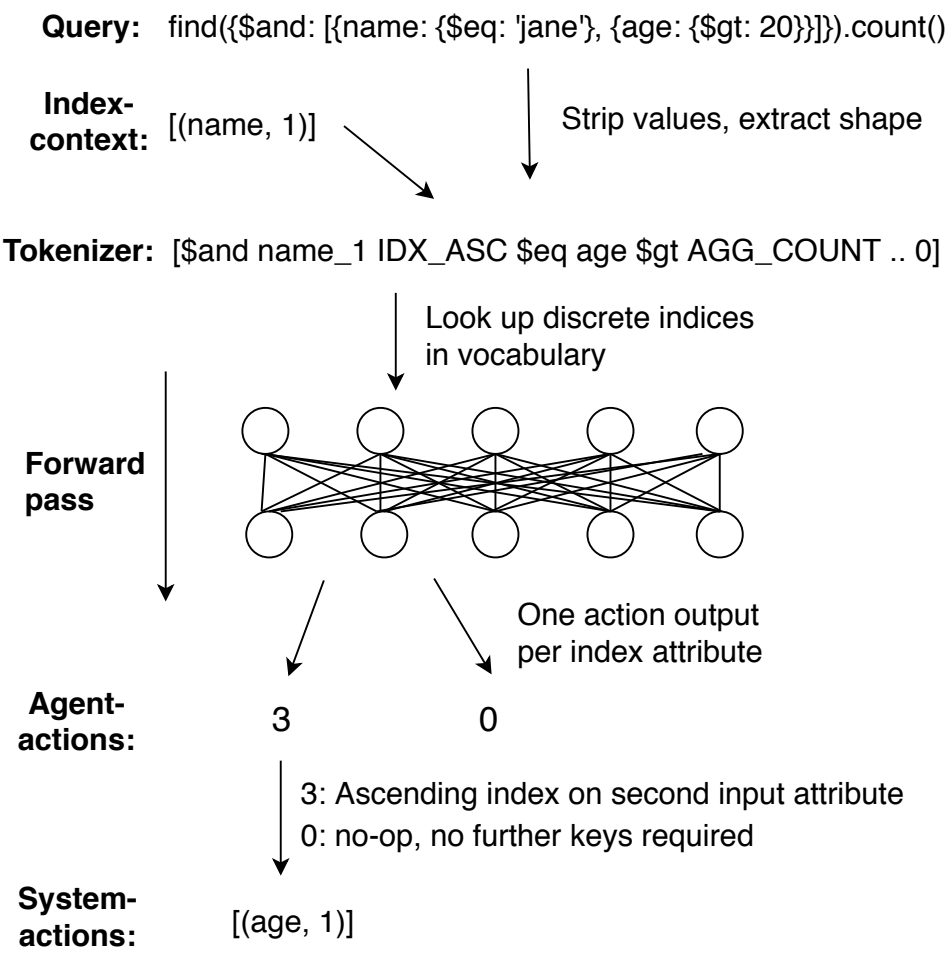


Figure 5.4: Action parsing scheme for MongoDB indexing case study.

Additionally, as discussed above, index intersection makes indices order- and sort-sensitive, thus requiring to also output a sort order per key.

The action representation should scale independently of the number of attributes in the document schema. Consider a combinatorial action model where the agent is modelled with one explicit action per attribute, and a separate action output per possible index-key. A 3-key index task on 10 attributes would already result in thousands of action options per step (5×10^3) when including extra actions for possible sort patterns (ascending/descending permutations).

This approach would not generalise to changing schemas or data sets. I propose a positional action model wherein the number of actions is linear in k. When receiving a query, I extract all query attributes and interpret an integer action as creating an index on the ith input attribute, thus allowing the agent to learn the importance of key-order for prefix intersection. To distinguish sort patterns, I create an extra action per key (one ascending, one descending with ascending default). This results in $1 + 2k$ actions for a k-key index (one output for no-op).

Figure 5.4 illustrates action parsing for $k = 2$ and a simple query on *name* and *age* attributes. In the example, the *name* field is already indexed so when the query is tokenised, a special index token (IDX_ASC) is inserted to indicate the existing index. The tokenised sequence is mapped to integers via the embedding layer and passed through the network, which outputs k integer actions. In the example, the agent decides to implement one additional single-key index by outputting 3 and 0, where 3 implies an ascending index on

the second input attribute, and 0 is used for no-op if fewer than k keys are required in the index.

For rewards, the optimal indexing strategy is the minimal set of indices $\mathcal{I}$ meeting performance level objectives such as mean latency or 99th latency percentiles for a set of queries $\mathcal{Q}$. Let $t(q)$ be the time to execute a query $q \in \mathcal{Q}$ under an index set $\mathcal{I}$ and let $m(\mathcal{I})$ be the memory usage of the current index set.

The choice of reward function is guided by cost of evaluation. Consider a sparse reward function which evaluates query runtimes once after an agent has made a decision on each query (*evaluate once*). When evaluating once per episode and providing a default zero reward for all intermediate steps, learning the impact of individual actions towards the global reward is more difficult, and the required sample size increases correspondingly. Such a reward can be necessary if intermediate evaluation is difficult to construct, i.e. when a game is either won or lost but the intermediate game state does not allow predicting a winner.

In indexing, the evaluation is always available as the entire query set could be theoretically executed after each indexing decision. The structure of the problem admits an incremental reward computation. I define the per-step reward as

$$r = -\omega_m m(\mathcal{I}) - \omega_t \sigma \left(\sum_i^N t(q_i) \right) \tag{5.5}$$

with σ being the desired aggregation function, e.g. the arithmetic mean (or 99th percentile) of query execution times. Both terms are weighted by ω_m / ω_t respectively to express preference of memory cost against execution times. Note that while each state transition (a transition corresponds to deciding on a single query) includes the entire query set runtime, this can be evaluated incrementally through caching:

1. Each step, the agent receives a query state and decides to create an index or not.
2. Only queries interacting with the columns in the index could theoretically prefix or intersect the index. They are executed and their runtime cached. Prior cached runtimes are invalidated.
3. All other query's runtimes are read from a cache as they are assumed to be unchanged.

Using this reward design, the agent observes the full query set performance against the cost of the index at each step without needing to perform a full evaluation. While undesirable from the theoretical view of an agent learning from raw feedback with minimal reward design, incremental evaluations are effective in sample-constrained environments. As algorithmic capabilities for credit assignment improve, expert-designed incremental evaluations may not be needed. In Wield, I hence implemented

- A schema mapping
  - query operators and schema information to integer indices to define state inputs,
  - and in the same schema, an output mapping based on the number of allowed keys in the index.
- A converter implementing:

- conversions between queries to tokenised input sequences, using the layout defined in the schema.
- conversion between integer actions and index columns to be combined in an index.

Listing 5.3 illustrates action and reward conversion for the indexing problem (state conversion omitted due to size of the tokeniser).

5.7.3 Indexing demonstrations

To facilitate learning from demonstrations for indexing, I give examples for intuitive demonstration rules grounded in the problem semantics.

1. A positive margin rule which simply suggests the full index with a small expert margin. Creating the full index, i.e. combine all attributes of each query into an index, improves latency but results in many unneeded indices. The labeling rule simply copies all columns in the query body as the action and specifies a positive expert margin (example in Listing 5.4).

2. An alternative positive rule which defaults to no-op. To avoid creating unneeded indices, this rule creates a bias for not creating any indices at all initially.

3. A negative margin rule to discourage reverse-order indexing which decreases prefixing, i.e. a query with column-order c_1, c_2, c_3 should not try to create an index in the reverse order c_3, c_2, c_1. The labeling rule provides the reverse order index as the action and specifies a negative expert margin.

These rules are neither exhaustive nor optimal. They illustrate how basic problem semantics can be encoded weak in supervision rules. If available, an index recommendation tool by a commercial database provider could be used to generate demonstrations. In Wield's evaluation, I discuss the performance improvements and tuning difficulties arising when using weak supervision.

5.7.4 Wield workflows: Putting it all together

Wield's abstractions are coordinated in *workflows* which consume the interfaces provided by schemas, converters, and demonstration rules. For example, the demonstration workflow:

1. Imports a data set using a converter to map system-representations to agent states, actions, rewards, and terminals.

2. If provided, applies demonstration labelling functions to generate actions and/or rewards.

3. Delegates offline training to an RL agent configured via the corresponding schema layouts for states and actions.

In contrast, an online workflow facilitates interaction with a live system through a system interface which executes converter outputs on the system, e.g. by sending a HTTP request to update a configuration parameter. Online workflows consist of these steps:

```
def system_to_agent_action(system_action, meta_data):
  index_columns = system_action["index"]
  action, attribute_order = {}, {}
  action_values = []
  i = 1
  for attribute in meta_data["query"].attributes:
    attribute_order[attribute] = i
    i += 1
  # Find position of index attribute in input attributes.
  for index_tuple in index_columns:
    input_position = attribute_order[index_tuple[0]]
    if index_tuple[1] == "ASC":
      action_values.append(1 + (input_position - 1) * 2)
    elif index_tuple[1] == "DESC":
      action_values.append(1 + (input_position - 1) * 2 + 1)
  # Map to action names.
  for i, name in enumerate(self.actions_spec.keys()):
    action[name] = action_values[i]
  return action

def agent_to_system_action(agent_action , meta_data):
  index_tuples = []
  query_attributes = meta_data["attributes"]
  for name in self.schema.actions_spec.keys():
    action_value = actions[name]
    if action_value != self.noop_index:
      action_input_field = int((action_value - 1) / 2) + 1
      if action_input_field <= len(query_attributes):
        attribute = query_attributes[action_input_field - 1]
       sort_direction = "ASC" if action_value % 2 == 1 else "DESC"
       index_tuples.append((attribute, sort_direction))
  # System command is list of attributes to combine in index.
  return index_tuples

def system_to_agent_reward(system_metrics):
  runtime = system_metrics["runtime"]
  index_size = system_metrics["index_size"]
  return - (self.runtime_weight * runtime)
         - (self.size_weight * index_size)
```

Listing 5.3: Indexing action converter based on positional action parsing.

```
# Maps system metrics to state inputs.
class DemonstrationRule(object):

  def __init__(self, converter, schema):
    # Layout, parsing.
    self.converter = converter
    self.schema = schema

  def generate_demonstration(self, state):
    #  State is a query.
    tokens = self.converter.tokenise(state)
    attributes = [c for c in tokens if c in self.schema.attributes)
    # Filter for ascending/descending sort  orders.
    sort_order = [s for s in tokens if c in self.schema.sort_tokens)
    # Full index copies all attributes and their sort order.
    return [(a, s) for a, s in zip(attributes, sort_order)]
```

Listing 5.4: Simplified demonstration rule for full indexing.

1. Based on schema definitions, a task graph for an RL model is created. The task graph in turns generates computational graphs for each task node, e.g. via RLgraph. Pre-trained model-weights can be restored from checkpoints if available.

2. A workload schedule is generated based on progressive randomisation assumptions and evaluation mode, i.e. blackbox mode or generalisation with fixed or randomised training and test sets.

3. To train a model, a system controller first initiates the workload schedule. In the case of query indexing, this means beginning to create indices from the current model by converting queries.

4. The system controller coordinating the workflow then keeps alternating between executing queries and creating indices by converting queries to agent states, agent actions to system actions, and system metrics to rewards via its converter.

5. Following completion of a workload schedule, the model is evaluated on a test schedule which in the case of black-box optimisation may be the same as the training schedule.

6. Final model parameters, serialised workload schedule, random seeds, and training parameters are exported to be fully reproducible.

All workflows rely on converters o transform representations when alternating between system and agent, demonstration rule, or other decision making mechanisms serving as baselines. Wield can hence be viewed as a design tool on top of RLgraph, where Wield's responsibility is to identify RL representations which RLgraph can execute.

Research into systems applications of reinforcement learning thus far has relied on one-off implementations which can lead to tight coupling of model, system-specific interfaces, and workload semantics. By separating systems semantics from workload coordination, and task layout from conversion between agent and system view, Wield provides re-usable

workflows which can be quickly adapted to new systems and task designs. For example, Jeremy Welborn in collaboration with me used Wield on top of RLgraph to explore structured action representations in PostgreSQL indexing [WSY19].

5.8 Future workflows and deployment

The indexing case study illustrates how problem implementation is assisted by Wield's software primitives. Here, I discuss near-time directions for model design.

First, the identification of effective state and action representations is an iterative manual process. Wield structures the implementation but currently does not provide means to help identifying states and actions. Several approaches may be used to support representation design:

- **Feature analysis.** While not specific to RL, statistical methods such as principal component analysis can be used to extract relevant system parameters for the state representation. This could be automated if users supply system configurations and profiling traces (e.g. in database tuning [VAPGZ17]).
- **Structure search.** More ambitious research could target automated hierarchical decomposition of state and action spaces. While feasible to implement as search problems, automated architecture designs require a large number of evaluations. By substantially expanding total hyperparameters, experimental approaches may be viable only in settings with cheap simulators.
- **Learning latent representations.** Another approach to dealing with large numbers of configuration parameters and performance metrics is to use e.g. a variational auto-encoder to learn a compact latent representation [HS18]. This in particular means a policy network can purely learn to act on the compact representation.

Their viability greatly varies based on the cost of evaluation and the availability of prior experience data, with feature analysis providing most near-time utility.

A befitting analogy may be a comparison to early database management systems. They primarily standardised access to data storage. Modern database engines include a plethora of utilities to help manage all aspects of data processing. Similarly, future versions of Wield or similar tools will both assist in the design phase, and in the training phase through monitoring, integrated tuning, and model lifecycle management.

5.9 Summary

In this chapter, I analysed reinforcement learning as a decision making and optimisation mechanism in computer systems. The starting point to my analysis is the observation that despite a wealth of experimental successes, real-world systems incorporating reinforcement learning approaches remain sparse. Successes of deep RL in games have revived interest in this line of research. Moreover, I argued that simply using neural networks as function approximators is not sufficient to address the practical gap, and that novel evaluation modes and software systems are required to support emerging learned system components.

To accelerate research, I introduced Wield, a framework for practical task design in RL. To study practical system behaviour, Wield focuses on standardising workflows when learning on real-world systems. To this end, Wield provides the following abstractions:

- Schemas and converters as programmatic task description mechanisms which can be translated to executable computation graphs by frameworks such as RLgraph.
- Progressive randomisation as a set of evaluation modes with explicit staged randomisation of both optimisation and workload aspects.
- Contextual labelling rules to create fine-grained imperfect demonstrations for workload data.
- Task graphs as a means to model multi-task and hierarchical learners to decompose large state and action spaces.

By focusing on incremental evaluation through randomisation, Wield is a first-of-its-kind tool in the emerging area of learned computer systems. Combined, Wield and RLgraph provide a comprehensive software stack for designing, evaluating, and executing deep RL algorithms. In the following chapter, I evaluate Wield's abstractions guided by the perspective of progressive randomisation.

Chapter 6

Wield evaluation

6.1 Evaluation aims

In this chapter, I evaluate my Wield prototype. The evaluation focuses on the following properties:

- The benefits of Wield's abstractions for implementing RL models in contrast to one-off implementations (§6.3).
- The utility of progressive randomisation as an instructive experimentation scheme (§6.4.3).
- Practical obstacles to deploying RL in systems tasks, considering the disparity between experimental successes but limited practical deployments (§6.4).

High-profile successes of RL (e.g. AlphaGo [SHS+18], OpenAI Five[Ope18]) have relied on orders of magnitudes more resources than most organisations can access. In what I consider resource-insensitive research, the aim is to demonstrate RL can solve a task without consideration for cost.

My goal in evaluating applications at smaller scale, both in the number of samples collected and deployment, is not to draw conclusions about these resource-insensitive experiments. Instead, my aim is to gather experimental evidence towards my initial thesis of limited real world deployments. What optimisation or generalisation capabilities do common algorithms exhibit at what cost? Is the gap between alleged capabilities and observed real-world utility in computer systems closing?

6.2 Learned indexing

6.2.1 Workload

In this section, I incrementally evaluate the compound indexing task. Using progressive randomisation requires configurable query workloads. To this end, I implemented a query workload generation mechanism which synthetically generates query shapes of specifiable difficulty (via the number of predicates and operator distribution). To configure a workload, allowed query operators, number and distributions of aggregation and selection operators

can be specified. For individual query generation, I use templated queries with sampling functions. They sample query predicate values from workload distributions based on real attribute values (e.g. Figure 6.1).

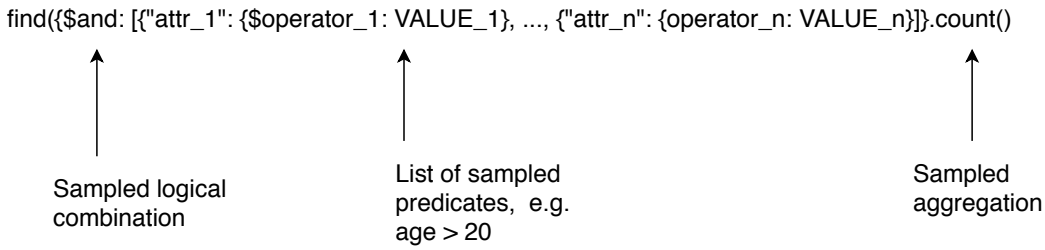


Figure 6.1: Randomised query generation.

Each training episode consists of a random or fixed set of queries, depending on randomisation mode. Importantly, query sets and schedules of training episodes (order of query sets) can be deterministically reproduced through random seeds. When training for generalisation, a different set of queries is sampled each episode (training covers a distribution of tasks), as well as in the final evaluation of the trained model. In black-box optimisation mode where the aim is to optimise a single fixed set of queries, I chose the index set associated with the highest reward during training. For the final evaluation, I recreate these indices and measure final index sizes and run times.

6.2.2 Experimental setup

I ran experiments on a real-world dataset, the Internet Movie Database (IMDB [imd18]). I imported datasets for titles and ratings (*title.akas*, *title.basics*, *title.ratings*) comprising ≈10 million documents into one collection (15 attribute fields). All experiments were run on a variety of commodity server class machines (e.g. 24 cores (Xeon E5-2600), 198 GB RAM), and using MongoDB v4.0.2.

I consider rule-based and auto-tuning baselines. One rule-based strategy I use to compare results is full indexing (*Full* in the following) wherein I simply create a compound index covering all fields in a query (respecting its sort order). I experimented with other rules based on only indexing unindexed fields but they failed to reliably improve latency. Full indexing improves latency but creates a large index-set.

I also compare my model to the OpenTuner [AKV+14] framework. OpenTuner exposes a generic tuning interface and uses an ensemble of search techniques ranging from evolution, hill-climbing, bandits, particle swarms to random search. Meta-techniques are used to allocate a larger proportion of trials to well-performing techniques. I chose OpenTuner in particular because defining a covariance function for indexing is difficult, and evolution-based methods permutating index variants appear promising. To use OpenTuner for indexing with Wield's abstractions, I created integer parameters corresponding to database attributes to combine into a compound index.

I test generalisation across with OpenTuner as follows. During training, where each episode samples a different set of queries, OpenTuner attempts to find the index configuration performing best across episodes. At test time, I evaluate OpenTuner's best found index set across the entire workload distribution. This helps answer the question if per-query decisions improve performance over less specific workload-wide indices.

Parameter	Value
Learning rate	0.0001
Batch size	16
Layer size	128
Prioritised replay size	1000
Initial exploration-ϵ	1.0
Final exploration-ϵ	0.05
Exploration-ϵ decay steps	500
Target network sync interval	24
Updates per transition	4
DQfD expert-margin	0.5
DQfD expert-reward	0.5
Size/runtime weights (equal)	1.0

Table 6.1: DQfD training parameters used in the indexing case study.

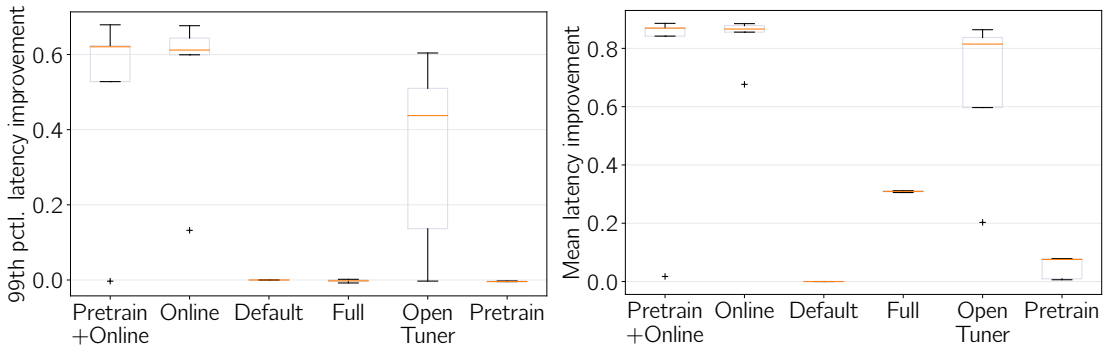


Figure 6.2: 99th percentile (left) and mean (right) relative latency improvements against unindexed configuration in the fixed blackbox task.

6.2.3 Fixed blackbox optimisation

I first evaluate the model described in §5.7 in a blackbox optimisation setting. Following the progressive randomisation scheme, I begin by sampling a workload to tune hyperparameters in a fixed workload with fixed optimisation parameters (C_0). Each workload comprises 20 templated queries sampling predicate values from actual IMDB entries. Each query comprised 1-3 selection predicates, a logical aggregation (e.g. "and", "or") of predicates, and a global aggregation (sort, count, limit). I sort queries by length to improve intersection (longest first).

A similar but smaller scenario was described by Sharma et al. [SSD] who used 15 fixed queries and single-key indices. In my experiments, the agent and OpenTuner were allowed to create indices combining up to 3 distinct attributes. Each experiment variant is executed for 100 episodes (five trials) where one episode corresponds to iterating across all queries in the task. Algorithm hyper-parameters were identified through iterative experimentation. I list parameters in table 6.1, and discuss tuning cost in §6.4.

Figure 6.2 shows 99th percentile and mean latency of the generated indices after training. For each run, I show the relative latency improvement against the default configuration (no index), and the relative index size improvement against the full indexing

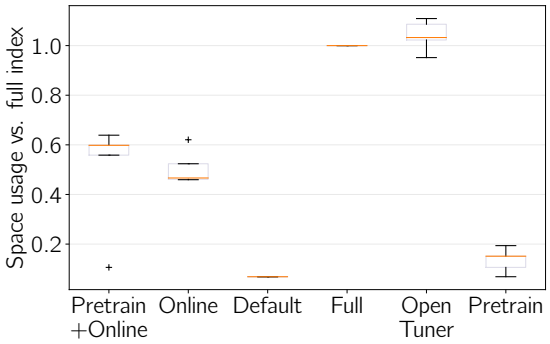


Figure 6.3: Index sizes of the created indices in the fixed blackbox setting with fixed weight initialisation.

	Total time	Pct.
Waiting on system	60698 s	95.1 %
Agent interaction/evaluation	3122 s	4.9 %
Mean episode duration	766 s	n/a
Min episode duration	578 s	n/a
Max episode duration	943 s	n/a

Table 6.2: Example wall clock times for training one model. One episode refers to creating the entire application index set.

strategy. The runtime component in the reward was optimised for mean latency. Results show large performance variation even with fixed network initialisation and workload as a consequence of noise in query execution times. This could likely be improved by re-executing queries at the cost of longer experiments.

In *this instance* of the fixed blackbox-task, both online training strategies (with and without pretraining) identified low latency index configurations. *Pretrain* refers to using a no-op labelling with positive margin, and a reverse-prefix function with negative margin to create demonstrations for each query. OpenTuner improved slightly against full indexing but did not identify strategies as effective as the learned ones. Figure 6.3 contains the corresponding index sizes (normalised against the full indexing strategy). OpenTuner created more compound indices than the full strategy. Queries have varying length, so it is possible to create an index spanning more columns than used by any query. Pretraining creates almost no indices as the blackbox task was pretrained with a slight preference for no-op. Both learned strategies used significantly less memory than full indexing. Differences between learned strategies are not significant. Both exhibit outliers failing to improve latency.

Table 6.2 breaks down wall clock times for training one model. Over 95% of time was spent waiting on index creation, with the remainder on updating the model and evaluating queries. Training does not require accelerator hardware and can be executed cheaply on a CPU due to small batch sizes and low sample throughputs. The distributed strategies discussed in the evaluation of RLgraph are not applicable, unless a simulator is available or many database servers accessible. The limited total number of experiences puts greater weight on hyperparameters and feature selection.

To summarise, the proposed model with a fixed blackbox task but random optimisation parameters is in $C_1(n = 2000, s = 5, f = 1.0)$. I define as success if the pretrain+online variant improves mean latency by at least 50% against the default configuration since

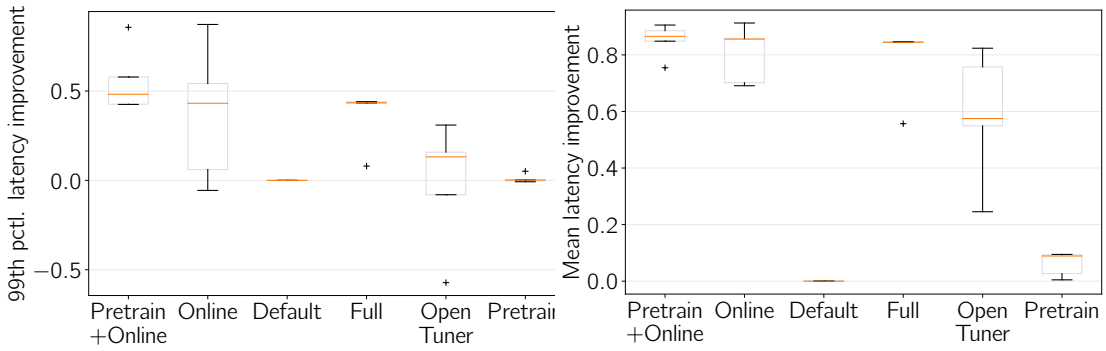


Figure 6.4: 99th percentile (left) and mean (right) relative latency improvements against unindexed configuration in the fixed blackbox task with random weight initialisation.

mean latency was the reward objective. Pretraining did not help performance in the fixed task. Neither method was able to learn to avoid any unneeded indices in the given number of samples. Random weight initialisation seeds did not significantly affect latency of the identified solutions but affected the size of the solution.

Next, I randomise the optimisation seed, i.e. network weights and mini-batch sampling strategies (C_1). Figure 6.4 shows latency results using random initialisation. While learned latencies achieve similar improvements as in the fixed configuration, a noticeable observation on this different fixed task is full index performance. In the prior query set used in C_0, the full index performs significantly worse than the learned strategies on query latency. Index sizes did not show meaningful difference to C_0..

Analysis of slow queries in the prior query set shows that count queries with an *or* logical operator (which can combine multiple indices) do not benefit from indexing all individual query predicates. In the task sampled for C_1, the full index achieves similar latency improvements as the learned strategies.

The relative performance against a baseline which may be used to make a case for a novel approach can vary drastically per workload-sample. While task variation seems to impact performance, the learned variants manage to improve latency while using less space than the baselines (full indexing, OpenTuner).

6.2.4 Randomised blackbox optimisation

I now expand the blackbox setting to sample a different query set per experiment to obtain further insight into sensitivity to workload variation, i.e. a randomised blackbox (C_2) scenario. Figures 6.5 and 6.6 show these relative effect sizes. Runtime improvements show substantial variation across different blackbox tasks $C_2(n = 2000, s = 5, f = 0.8)$. *Pretrain+Online* achieves better latency improvements on average than online-only training, and also improves against the full indexing strategy, while only using half the memory. OpenTuner similarly improved latency but at the cost of creating full indices.

Table 6.3 breaks down index statistics for different modes (standard deviations in parentheses). *Created* refers to the number of indices created, *usage* to the number of times each index was used, *Unused* to the indices not used at all, and *Unused size* to the size of all unused indices (all given as means across the different blackbox tasks).

First, $Pretrain + Online$ did not have the fewest indices but *Online* failed to improve 99th percentile latency. 99th percentile latency was highly sensitive to incorrect indices.

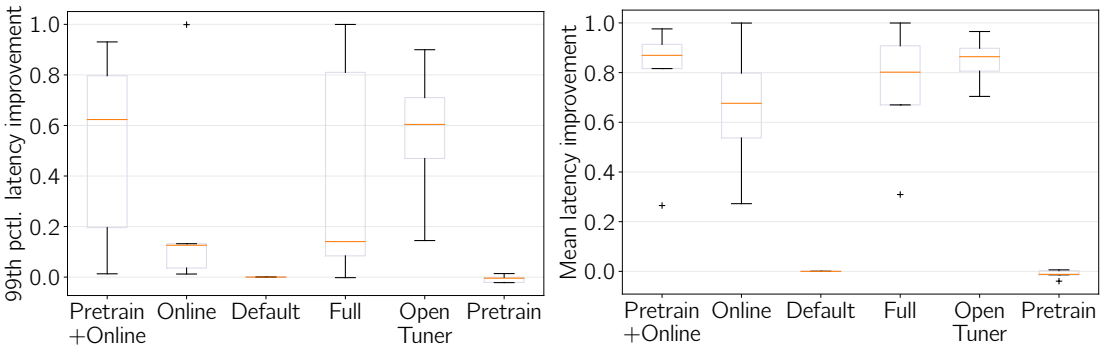


Figure 6.5: 99th percentile (left) and mean (right) relative latency improvements against unindexed configuration in the randomised blackbox task.

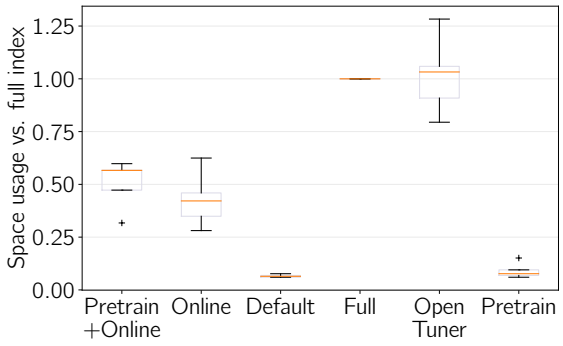


Figure 6.6: Index sizes against sizes of the full index in the randomised blackbox setting.

Queries with an *or* predicate may use multiple indices so usage counts do not necessarily sum-up to the number of queries. I measure the combined size of unused indices as I hypothesized that it may be difficult to learn to avoid very small unused indices due to small impact on reward. OpenTuner created the most indices but the average usage per index and the number of unused indices indicate that no effective strategy was found to leverage prefix intersection. Over half of its indices were not used. The main observation is that despite substantial real-world training time (up to 24 hours including all pre-training and orchestration for one training run), the trained strategies are evidently not optimal, as the unused indices could be removed. This does not take into account the true cost of calibrating this experiment for months (which I discuss in §6.4). Collecting a single set of five trials, which must be on the same hardware due to large variations in query execution times across CPU configurations, takes up to weeks.

I make no claim beyond the progressive randomisation classifications for the given number of samples. Implementing a simulator is another popular choice in applied RL but introduces significant design and transfer cost as the simulator may need retuning per database model. Here I focus on the feasibility of direct system evaluation which is closer to practical auto-tuning workflows.

While impractical for full experiments, I ran some randomised blackbox trials at double the number of episodes and show the reward curves in Figure 6.7. Importantly, rewards continue to improve, indicating that users can trade solution quality against experiment runtime. However, analysis of e.g. trial 3 with a sudden reward jump reveals that an undesirable solution was found whereby the agent could save substantial index size by allowing higher latency. This implies yet another tuning difficulty whereby tuning rewards

Mode	Created	Usage	Unused	Unused size (GB)
Pretrain + Online	10.2 (2.64)	1.01 (0.1)	3.6 (1.5)	0.64 (0.21)
Online	8.6 (3.01)	1.16 (0.23)	2.4 (1.36)	0.53 (0.24)
Full index	18.4 (1.62)	0.68 (0.11)	9.2 (1.6)	2.17 (0.35)
OpenTuner	19.8 (0.4)	0.53 (0.09)	11.8 (1.72)	2.2 (0.59)

Table 6.3: Index usage statistics for the randomised blackbox task. Numbers in parentheses indicate standard deviations.

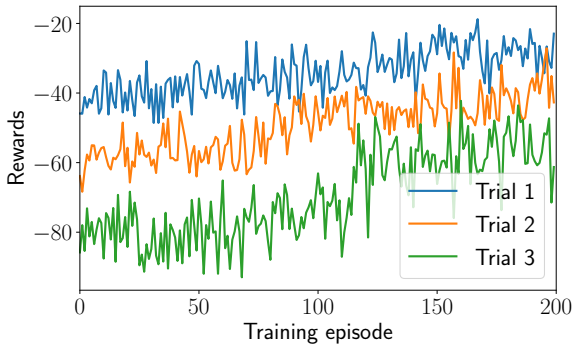


Figure 6.7: Continuing training yields further reward improvements.

for a shorter runtime may not be indicative of successful convergence later, again depending on the exact query configuration.

6.2.5 Generalisation

Next, I consider generalisation where all modes are evaluated on a randomly sampled final set of queries and trained across a distribution of query sets.

Figures 6.9 and 6.9 show latency and size metrics. I used the same hyper-parameters as in the blackbox evaluation. Recall that OpenTuner was used to find the best average index set across the query set distribution. Results illustrate the distinction between using a blackbox auto-tuner and a learned model. The index sets created have similar median latency improvements across all learned strategies and OpenTuner. The difference is that

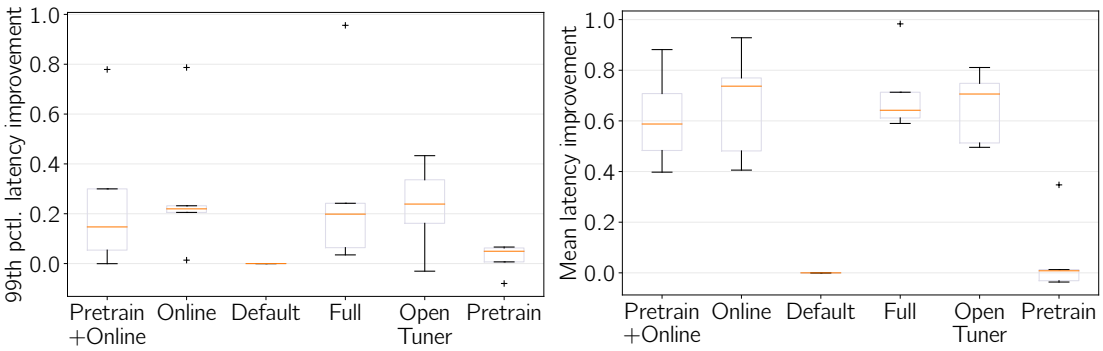


Figure 6.8: 99th percentile (left) and mean (right) relative latency improvements against unindexed configuration in the randomised generalisation task.

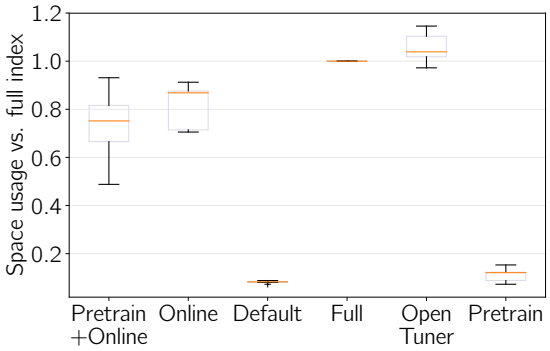


Figure 6.9: Relative index sizes in the randomised generalisation setting.

OpenTuner creates a significantly larger index set to cover the distribution of possible query sets while the learned models only create the indices for the particular test set (even if incorrect ones). How good are generalisation decisions and how do they compare to the indexing decisions made in a blackbox scenario? Median improvements of 99th percentile and mean latency were significantly higher in the randomised blackbox scenario (20-40%). Using the same latency score criterion, randomised generalisation achieved

$$C_4(n = 2000, s = 5, f = 0.6)$$

Index sizes against the full index were also higher. The number of unused indices for both learned strategies was much higher than in the blackbox setting, with up to one third of indices created not used by any query. Comparing latency improvements against OpenTuner, which produces a best effort across the entire workload distribution, shows no significant improvement from making query-specific generalisation decisions.

In-distribution generalisation also raises the question of test-task similarity. Since both test and training tasks are randomly generated sets of queries, training queries can appear in the test set. I hashed tokenised query shapes (after stripping concrete values in predicates) to quantify this similarity. Across experiments, I observed $30 - 55\%$ of queries in the test set being seen during training. However, there was no correlation between higher similarity and performance. Overall, the generalisation experiment confirms prior observations about limitations of Deep Q-learning. I hence did not test further generalisation modes.

6.2.6 Utility of weak demonstrations

Do imperfect demonstrations create useful inductive bias? I observed slight (but likely not significant) improvements in effect sizes when using a no-op labelling rule in randomised blackbox scenarios. In post-hoc analysis, this could be explained through the observation that bad indices can hurt performance. Thus, an additive model where the learner is biased towards no indices, and only adds good indices is preferable to a subtractive model where a learner is biased towards suboptimal indices.

However, my experiments also illustrate the additional tuning issues introduced by weak demonstrations in RL. The pretrain expert margin needs to be tuned in combination with online reward scales so that online training can override value estimates.

The notable difference between system tasks and other RL domains requiring human perception or control is that a demonstrator cannot easily assess demonstration quality. Manually obtaining a high-performing index set requires iterative analysis.

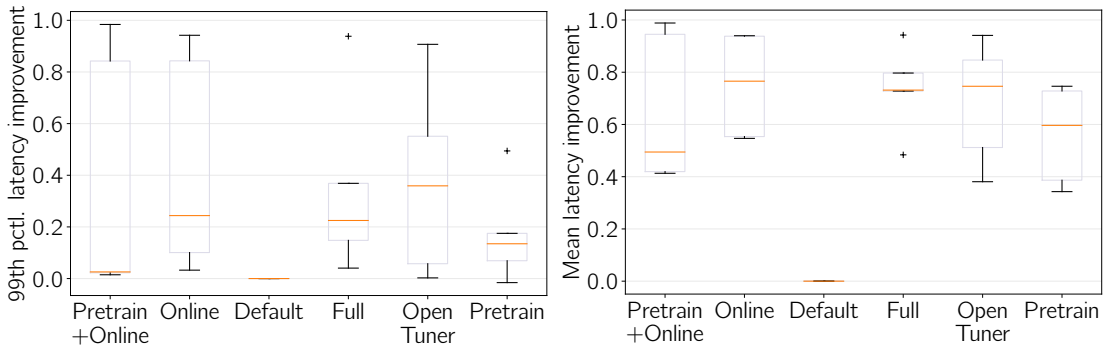


Figure 6.10: 99th percentile (left) and mean (right) relative latency improvements in the randomised generalisation task. Instead of no-op, a prefix rule was used.

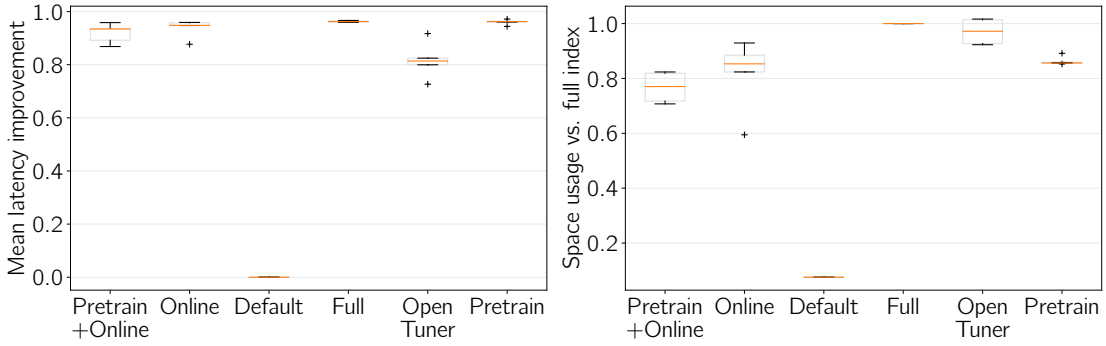


Figure 6.11: Mean latency improvement and relative index sizes using a human demonstrator in a fixed blackbox scenario.

I also evaluated a more sophisticated rule based on knowledge of query optimiser prefixing rules. The rule tests if query columns can be prefixed into any existing index, and otherwise creates a full index. It also tests reverse sort order combinations, i.e. if sorting is required, does an inverted index exists that can be used to execute the sort? Figure 6.10 depicts latency results in a randomised generalisation setting using the prefix rule. In generalisation experiments, I pretrained from 1000 synthetically generated and labelled episodes. The rule ('Pretrain') on its own improved latency, but combining it with online training yielded worse results than both online training or pretraining individually. Even semantically well-grounded rules may not create better inductive bias than random weight initialisation.

I also ran a separate experiment testing the utility of human intuition. This was done using a fixed query set for which I created manual demonstrations based on knowledge of schema and data distribution, but without iterative refinement. Figure 6.11 shows mean latency and index sizes (99th percentile latency showed both learned strategies close to full indexing). After retuning hyper-parameters (expert margins and training steps), the human demonstrator achieved the best index size with about 10% improvement against the from-scratch strategy.

When comparing to randomised blackbox results pre-trained by no-op, no-op provided a more useful inductive bias than expert human intuition. Overall, if the agent is not meant to exactly imitate demonstrator behaviour, weak demonstrations can be helpful if fine-tuned, but also detrimental to performance. Better confidence score schemes also introduce

additional hyperparameters. Neither indexing rule gives desirable default behaviour. In well-conditioned problems (i.e. smooth objective function), rule-based pretraining may be more useful.

6.2.7 Discussion

Results in this section indicate RL can be in principle be used to directly optimise database behaviour. Wield was used to generate demonstrations, interface OpenTuner, and explore different workload randomisation modes. Experimental results point towards some utility in a randomised blackbox scenario with $C_2(n = 2000, f = 0.8, s = 5)$. Generalisation performance was both observed with lower success rate and also with worse performance than black-box results. Strikingly, generalisation decisions did not produce better latency results than using an auto-tuner trained across the task distribution without query-set-specific decisions. They did however as expected produce smaller index sets.

Long experiment durations even for relatively small databases are the main hurdle to gain more robust insights and iteratively improve models. The results presented here required months of calibration. Similar to the observed behaviour when evaluating Atari Pong on RLgraph, training also fails for some workloads. These results call into question the widespread use of fixed tasks (e.g. Join Order Benchmark [LGM+15]) in prior applications of RL to databases. Learners may easily overfit to particular input features or task instance properties. Hyperparameters can accidentally be tuned to ensure a particular solution is uncovered during exploration.

6.3 Device placement

Next, I investigate the device placement problem of assigning parts of computation graphs onto different compute devices to minimise operation runtimes. If multiple accelerators are available, how should a large computation be distributed across them when considering the cost of allocating device memory and communication? The corresponding RL task selects one device for each operation.

Better placements yield graph runtime improvements which amortise tuning costs in sub-sequent experiments. I apply Wield and progressive randomisation to investigate capabilities of prior work, before implementing a new placer.

6.3.1 Setup

All experiments were run on TensorFlow 1.13 and using Google Cloud compute instances with four K80 GPUs. Results for the hierarchical placer in the original publication were reported on slower K40 GPUs, but these were no longer available for renting.

I first attempt to reproduce some of the results reported for the hierarchical device placement module of TensorFlow [MGP+18]. The implementation is available open source as part of the TensorFlow distribution (as part of a module called grappler)[1]. It contains high level utilities to evaluate a TF graph definition on given hardware configurations. I will in the following refer to the hierarchical placer as *Grappler* or *Grappler's placer*. Note that this module internally consists of two components, one for grouping operations into

[1] `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/grappler`, accessed 23.9.2019

device groups ('grouper'), and one for placing device groups ('placer'). The term 'placer' hence refers to both the module specifically responsible for placing groups, and the overall architecture.

Grappler's placer generates a placement configuration and instantiates a corresponding computation graph, evaluates its runtime, and runs optimisation with a fixed time budget.

6.3.2 Evaluating the hierarchical placer

My goal of rerunning Grappler's placer on some of the workloads reported is to classify it in the progressive randomisation scheme. Are the available set of hyperparameters and the sample counts in e.g. C_1 or C_2? The consequence of the hyperparameters being in C_1 would mean that the module needs to be re-tuned for every problem, thus significantly increasing deployment. In contrast, randomised blackbox success would imply that a large set of graphs could be optimised without calibration.

As a benchmark, I use the neural machine translation (NMT) architecture reported in the paper (and in prior work [MPL+17]). While some architectures were trained in a large distributed setting (i.e. 16 independent nodes with each 4 GPUs to evaluate placements), reported NMT results ([MGP+18], Figure 3) illustrate strong performance improvements within hours of training using a single node. NMT is an attractive benchmark task because the variants tested consist of an encoder-decoder architecture with multiple recurrent layers. When translating, input and output sequences are unrolled dynamically, and prior work on device placement shows that non-trivial placements split data across GPUs and time-step dimensions.

Grappler's placer was not able to run evaluations on the NMT implementation provided by Google[2] with its own graph evaluation utilities. Serialisation of a number of metagraph components failed, so Grappler could not initialise the corresponding TensorFlow graph. The results presented here were obtained by directly instantiating the TensorFlow graph and calling the training operations (implementation helped by Kai Fricke). This significantly increases set-up cost per measurement. Each measurement was given a single warm-up run, and the subsequent second runtime was used for the reward. The open source implementation decays its learning rate to 0 within 1000 updates, corresponding to the results reported in the paper. My results were run at least 1000 steps.

I begin with the fixed random seed, fixed workload setting (C_0). Figure 6.12 illustrates runtime of the training operation (one iteration of mini-batch stochastic gradient descent). I used the random seed supplied by the default configuration in the open source implementation, and repeated the experiment 10 times. Grappler identified improved placements in most runs with a mean final improvement (measured as the mean of the final 10 steps against the initial runtime) of 52%. One run failed to substantially improve (5%) final runtimes. Note that invalid placements were removed from the plots.

I break down both the *final* relative improvements and the best-seen improvements during training in 6.4. Results show that i) all trials identified significant improvements during training, and ii) some trials diverged so the final model underperformed. Again, divergence even occurs with a fixed random seed and a fixed workload, likely due to noise in the reward. The authors compared to expert and algorithmic graph partitioning baselines which are reasonable alternatives when considering the large number of operations in a graph which make direct application of auto-tuners difficult. As a sanity check, I

[2] `https://github.com/tensorflow/nmt`, accessed 23.9.2019

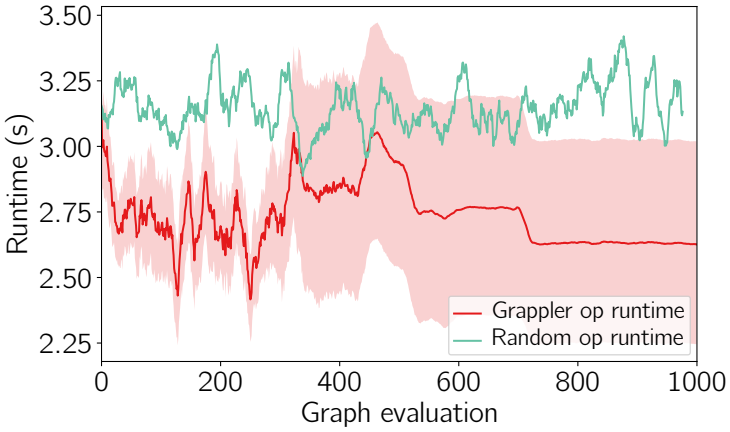


Figure 6.12: Fixed seed evaluation of Grappler's placer.

Trial	Final model	Best seen
Trial 1	57.0%	69.0%
Trial 2	67.0%	71.0%
Trial 3	25.0%	69.0%
Trial 4	51.0%	73.0%
Trial 5	5.0%	74.0%
Trial 6	70.0%	73.0%
Trial 7	69.0%	69.0%
Trial 8	47.0%	73.0%
Trial 9	68.0%	70.0%
Trial 10	66.0%	69.0%

Table 6.4: Improvements against initial execution time found by Grappler's placer.

contrast learned values against entirely random placements. I do not aim to reproduce their baselines but rather to assess model capabilities under randomisation assumptions.

In Figure 6.13, the same graph is evaluated on 10 randomly chosen seeds (fixed workload, randomised optimisation parameters, C_1). Mean improvement was 72% with all random seeds achieving over 70% improvement. A possible source of non-determinism is noise in the placement evaluation, with subtle differences in rewards leading to policy shift. While restricted to a single graph, reproduced results confirm the paper's claims of reliable improving runtimes in black-box settings by at least 30%, i.e. $C_1(n = 1000, s = 10, f = 1.0)$. This runtime object was chosen conservatively against the minimum reported improvement across graphs in the paper.

In Figure 6.14, I now modify task parameters by varying batch size and unroll lengths in the recurrent network to create a randomised blackbox setting (C_2) where both graph and optimisation seed are varied. Final runtimes are hence expected to differ due to different graph sizes. Three trials succeeded, one trial failed entirely (1), and one trial (5) diverged again from an effective configuration ($C_2(n = 1000, s = 6, f = 0.83)$).

I re-ran random search again on all graphs. Table 6.5 compares runtimes of best placements across trials. Random placements are significantly worse than learned placements for the same sample budget.

Did trial 2 fail due to posing a more difficult placement task, or due to random

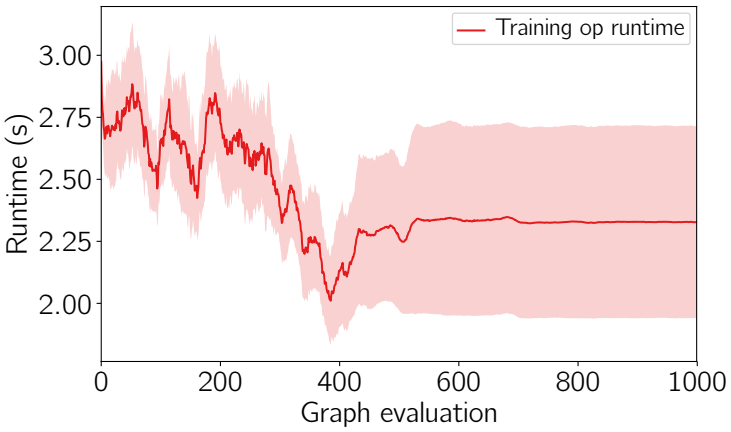


Figure 6.13: Random seed evaluation of the hierarchical placer in a fixed task.

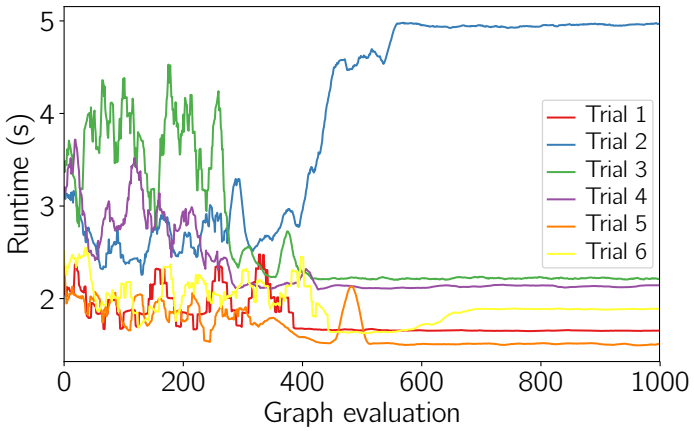


Figure 6.14: Randomised blackbox graph configuration trials.

initialisation parameters? The idea behind progressive randomisation in case of unknown failures is to decrease randomisation levels and re-evaluate the failed task. I hence re-ran the failed trial as a fixed blackbox task with randomised optimisation parameters. Figure 6.15 shows results of rerunning the same task nine more times for a total of ten trials. Results include more fails and diverged results but also succeeding runs. Three trials failed to improve placements significantly. Performance variation is further higher than in the published result which I initially evaluated in the fixed blackbox scenario. While cost-prohibitive, it would be desirable to evaluate a larger range of distinct blackbox tasks. Variations in task configuration seem to affect failure rate.

Finally, I consider generalisation capabilities. I first evaluate the final trained model in each trained randomised graph (Figure 6.14) against all other trials' learned models. That is, I measure the relative runtime overhead of model A on graph B against the model that was trained on B itself.

Figure 6.16 and 6.17 show two examples of this cross-graph comparison. Final models perform significantly worse against the best seen solutions when training specifically on the respective graph, with overheads against the best solution ranging between 20% - 50%. Analysis of the produced device placements shows:

- During training on a particular graph, non-trivial placements are identified during exploration. These use all devices.

Trial	Grappler	Random search
1	2.1 s	2.81 s
2	1.42 s	1.86 s
3	1.95 s	2.67 s
4	1.59 s	2.15 s
5	1.94 s	2.65 s
6	1.63 s	2.13 s

Table 6.5: Best run times found by Grappler against random search across trials. Each trial corresponds to one randomised graph problem.

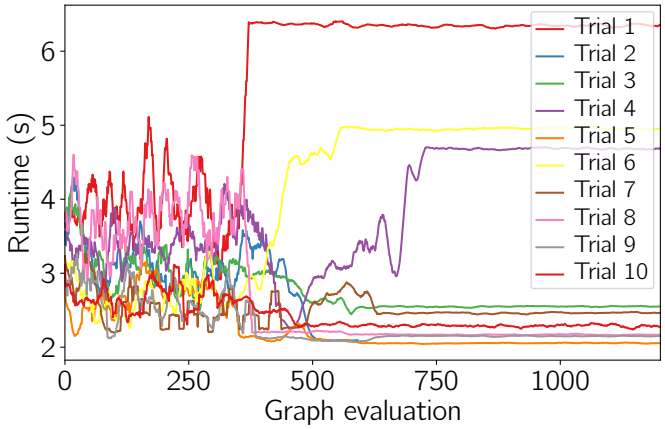


Figure 6.15: Repeated random seed trials for the workload parameters of failed trial 2.

- Diverged runs may identify effective placements through exploration but final placements default to single-device (CPU only) or single-GPU. When evaluating generalisation, placements for slightly varied graphs frequently defaulted to trivial single-GPU decisions.

The failure to identify non-trivial placements for slightly varied graphs is hence both a result of models diverging and limited model capability. In-distribution generalisation capabilities may be improved by training a placer across many different graph samples and using more training data. In summary, Grappler's placer is effective in blackbox tasks (C_2). It does not exhibit generalisation capabilities.

6.3.3 Implementing a placer with Wield

Reproducing Grappler's placer yielded several insights into placement behaviour. I use Wield and RLgraph to implement a novel hierarchical placer. The aim is to illustrate how Wield can be used to implement a hierarchical task architecture by combining RLgraph agents using Wield's abstractions. Instead of customising a hard-coded architecture, my goal is to implement a placer by transparent configuration and combination of components. The key design differences to the Grappler's placer are:

- Rewards are computed incrementally. Wield's placer places half of the device groups, leaves placements for the remainder of the graph unchanged, and evaluates placement

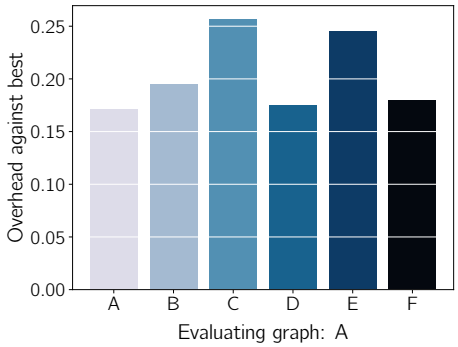


Figure 6.16: Evaluating graph A (corresponds to trial 1).

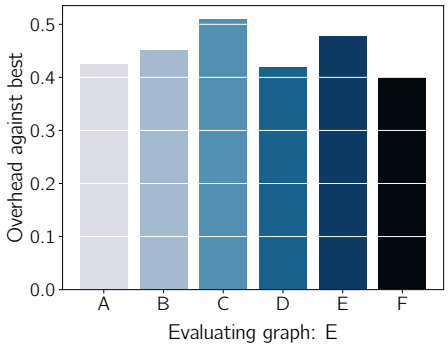


Figure 6.17: Evaluating graph E (corresponds to trial 5).

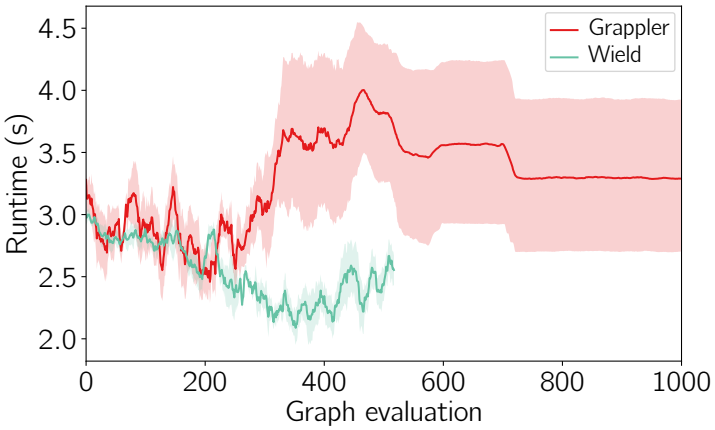


Figure 6.18: Open source hierarchical placer versus Wield placer

runtime. This is in contrast to Grappler's placer which evaluates placement decisions for the entire graph at once.

- Grouper and placer networks both use a proximal policy optimisation agent instead of a standard policy gradient. They are unmodified RLgraph implementations.
- Wield's placer uses a structured neural network representation described by Addanki et al. [AVG+19] (network implementation adapted by Kai Fricke). The neural network computes an embedding for each device group based on summing up the properties of neighbouring device groups to inform device transfer cost. Their work relies on hand-designed groups while Wield's placer learns grouping and placing through two independent learners. The neural network is implemented as an RLgraph component.

I reran Wield's placer on the neural machine translation architecture. To calibrate Wield hyper-parameters, I ran five trial experiments to identify effective learning rate schedules and incremental reward modes before final experiments. Figure 6.18 compares operation runtimes over the course of training. Due to experimental cost, I terminated some experiments early after no more improvement was observed. However, results show consistent improvement within 500 graph evaluations, and better placements were rarely observed in subsequent evaluations. Table 6.6 shows the best found runtimes for each trial in the fixed blackbox setting. Both approaches regularly found high-performing configurations with about 50% mean improvement against their initial runtimes.

Trial	Grappler	Wield
Trial 1	1.59 s	1.54 s
Trial 2	1.58 s	1.59 s
Trial 3	1.55 s	1.55 s
Trial 4	1.58 s	1.54 s
Trial 5	1.58 s	1.63 s
Trial 6	1.58 s	1.68 s
Trial 7	1.56 s	1.6 s
Trial 8	1.58 s	1.56 s
Trial 9	1.56 s	1.56 s
Trial 10	1.56 s	1.59 s

Table 6.6: Best runtimes found by Grappler and Wield's placer in the fixed graph setting.

Performance was highly sensitive to running the grouper and the placer agents at different learning rates and update schedules. The placer agent cannot begin identifying effective placements before operations are grouped. Hence, I executed the grouper which combines operations to device groups at a much higher initial learning rate than the placer. Flexibly executing tasks at different time schedules and update schedules through task factorisation is not possible in the rigid Grappler implementation.

Trial	Grappler	Wield
Trial 1	22.6%	37.4%
Trial 2	37.4%	38.2%
Trial 3	37.8%	32%
Trial 4	36.7%	37.4%
Trial 5	37.2%	41.5%
Trial 6	34.5%	48%
Mean	34.4% (5.4%)	39.1% (4.9%)

Table 6.7: Relative improvements of the respective best solution against the mean of the first twenty measurements for the randomised blackbox experiment.

The most noticeable observation when evaluating both placers was divergence after identifying the best-performing configurations. Grappler's provided default configuration updates for 1000 training steps after which the learning rate has decayed to zero. Due to cost, I could not investigate in more detail how other learning rate schedules might prevent divergence. Precisely timing decays for each specific task instance would at least double cost (e.g. sweeping 10 different learning rate schedules, then evaluating 10 different random seeds).

I also repeated the randomised blackbox experiment with 6 new graphs (C_2) to evaluate sensitivity to graph variations. Table 6.7 compares runtime improvements across trials between Grappler and Wield. Mean improvements are similar and differences not statistically significant. Finally, I also repeated the cross-graph generalisation experiment to investigate generalisation capabilities of the structured neural network representation. Since the network computes a permutation invariant embedding of operation neighbourhoods, higher robustness to input variants should be observed.

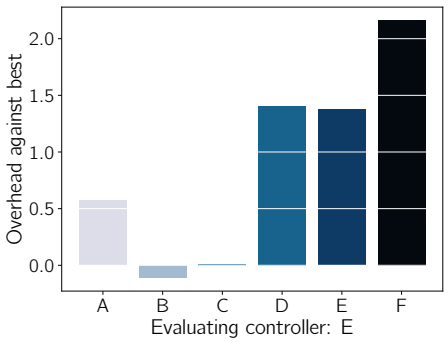


Figure 6.19: Evaluating graph E (corresponds to Wield trial 5).

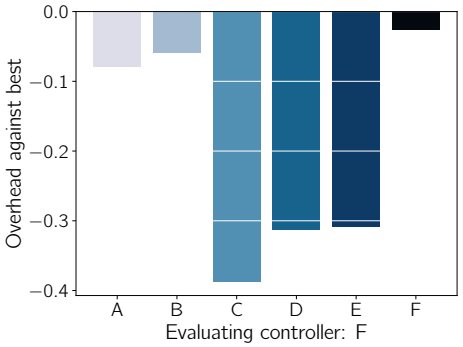


Figure 6.20: Evaluating graph F (corresponds to Wield trial 6).

Results (Figures 6.19 and 6.19) show a strong contrast across generalisation trials. When cross-evaluating models, both large overheads against the best blackbox solution (left) and significant improvements (right) were observed. I show a detailed breakdown of both approaches' generalisation capability by showing how the final model trained on a graph (rows) performs in terms of relative runtime improvement on another graph (columns) (Tables 6.8 and 6.9).

Tested graph	A	B	C	D	E	F
Model A	0.31	0.35	0.28	0.31	0.27	0.23
Model B	0.31	0.37	0.29	0.32	0.27	0.26
Model C	0.25	0.3	0.23	0.26	0.22	0.18
Model D	0.1	0.15	0.06	0.1	0.06	0.03
Model E	0.16	0.21	0.14	0.16	0.14	0.09
Model F	0.28	0.32	0.26	0.29	0.25	0.21

Table 6.8: Cross graph generalisation breakdown of Grappler models.

Tested graph	A	B	C	D	E	F
Model A	0.21	0.6	0.54	0.5	0.39	0.19
Model B	-0.3	0.33	0.24	0.09	-0.02	-0.22
Model C	-0.08	0.44	0.36	0.23	0.15	0.05
Model D	0.09	0.34	0.32	0.16	0.05	-0.08
Model E	-0.04	0.45	0.42	-0.49	-0.62	-0.83
Model F	0.39	0.42	0.65	0.58	0.53	0.44

Table 6.9: Cross graph generalisation breakdown of Wield models.

Results show Grappler's placer in the randomised generalisation setting only in few instances produces substantial improvements against the initial placement ($> 30\%$), with a classification of $C_4(n = 1000, s = 36, f = 0.22)$. Wield's placer achieves $C_4(n = 1000, s = 36, f = 0.47)$ and exhibits generalisation capabilities with model F. Overall, the Wield placer built through combining RLgraph and Wield abstractions performs like the custom-built tuned Grappler on blackbox tasks, and indicates potential on generalisation.

6.3.4 Discussion

In this section, I first incrementally evaluated capabilities of the open source Grappler placer. Reproducing Grappler results was instructive when contrasting them against published results on just a single blackbox task. While Grappler's placer improved results on NMT tasks with high empirical frequency, it failed entirely on some trials. Using progressive randomisation, my results point towards a classification of $C_2(n = 1000, s = 6, f = 0.83)$.

Grappler's placer also failed to exhibit generalisation capabilities even to slight input variations. In particular, models diverged after identifying effective placements, and learning rate schedules would need to be tuned to high precision to prevent this.

My experiments highlight the significant cost associated with merely evaluating a model for which a full set of pre-tuned parameters exist. When including all cost of calibration of the custom evaluation due to bugs in the open source code, it cost $5000 to assess the hierarchical placer on public cloud infrastructure. Identifying effective hyperparameters if none were available would have generated significant additional cost. Finally, I showed that using Wield and RLgraph, a competitive placer could be implemented by using RLgraph's algorithm implementations and Wield's design abstractions.

6.4 Progress and design costs

6.4.1 Hidden design costs

Using Wield and RLgraph, the overhead of developing RL models concentrates on state, action and reward design. Here, I discuss aspects of design cost which have received limited attention in the systems-RL literature.

Most applications I discussed in §2.4 executed training towards single objectives such as throughput or latency. They often report results for different individual objectives. Using weighted combinations of objectives may seem like an intuitive solution but this obfuscates the practical difficulty when balancing objectives. The reward contribution of each objective may change per problem instance. This results in solutions yielding different, potentially undesirable trade-offs unless weighing or normalisation schemes are tuned per problem.

For example, in the indexing case study I used a weighted combination of mean query execution time and memory usage of the corresponding indices. Neither individual objective is useful for optimisation, as training towards minimal runtime yields unneeded indices, and training towards minimal memory yields zero indices. Absolute runtimes and sizes depend on the query set. Even if each objective component is rescaled or normalised, training may result in different trade-offs being identified. This often resulted in undesirable solutions where some required indices were not created as the saved memory was identified to be yield higher reward gains than the improved latency. An alternative reward scheme may optimise only index size, subject to latency constraints.

Real world RL applications need to balance a multitude of objectives, and designing weights, normalisation schemes, or penalties to induce desired behaviour requires extensive analysis.

6.4.2 Hyperparameter tuning and customisation

Hyperparameter tuning is understood to be a critical part of deploying machine learning applications. Despite this 'common sense' knowledge, published work is often opaque regarding the full design cost. In RL, high sensitivity to hyperparameters and environment noise amplify the problem:

1. When iteratively designing an MDP representation, distinguishing ineffective features from ineffective hyperparameters is difficult without repeatedly sweeping parameter values between design changes. Moreover, during initial design, no detailed records of experiment trials may be kept.

2. Due to known difficulties in reproducing results, researchers may seek to avoid the impression of results being overly reliant on hyperparameters, as opposed to algorithmic innovation. Results are perceived to be more valuable if they are achieved with 'minimal tuning'.

Researchers are hence reluctant to provide implementation cost estimates. Based on the case studies I conducted, I conservatively estimate the experimental overhead for implementing novel RL applications to be at least 10-100 times the final experimentation cost. In supervised applications such as machine translation or image recognition, practitioners have moved to implementing custom applications by fine-tuning large pretrained models (e.g. transformers [VSP+17, DYY+19]).

While such standardisation has not taken place in RL, improved algorithmic robustness would enable similar transfer. Wield modularises task construction from a systems design perspective. Designing subsequent tasks in similar systems is accelerated by reusing decoupled task elements.

6.4.3 Evaluating progress and usability

A starting point to this dissertation has been the observed disparity between research interest and deployments in RL in systems. From that arose the question to what extent the success of deep neural networks has changed this, or has the potential to lead to more deployments.

The results presented in this chapter, using the lens of progressive randomisation, help categorise the utility of emerging RL techniques in computer systems. Specifically, I have observed:

- Deep RL mechanisms can successfully solve randomised blackbox optimisation (C_2) tasks directly interfacing real-world systems. This is achieved with varying reliability and requires extensive calibration.

- The recent emergence of structured representations [BHB+18] such as graph neural networks has led to applications reporting successful generalisation capabilities [MSV+18a, AVG+19]. These structured representations improve within-distribution generalisation through permutation invariance in the state.

- Generalisation capabilities remain difficult to reason about due to limited analysis into randomised workload generation mechanisms and the prevalent use of fixed task sets.

What do these observations imply for practical usage of RL in systems? A randomised blackbox model can be relied on as a 'plug and play' optimisation tool which needs to be trained per problem instance.

A within-distribution generalisable model can be used without requiring retraining if the workload distributions are known in advance. This is not typically the case for generic systems software components which organisations use to assemble their infrastructure (e.g. databases, message queues, stream processors, schedulers, application servers). A more realistic perspective may be pre-designed models with initial hyperparameters which require calibration per deployment.

Overall, it is likely that practical RL solutions in the nearer future will be restricted to large organisations which can afford the resources required to train and tune models. Real-world use-cases in domains such as finance or advertising where small optimisations can yield large financial gains should also be expected to increase. Examples of existing applications in advertising are Facebook's notification system [GCL+18] or Alibaba's ad-bidding system [JSL+18].

While RL applications may continue to capture the imagination of systems researchers, the difficulties discussed point towards near-time applications being more limited in scope than some may expect. Many of the prior works I discussed in Wield's progressive randomisation analysis relied on the same algorithms used here. They should be subject to similar limitations which need wider discussion.

6.5 Summary

In this chapter, I evaluated RL models designed with Wield on two combinatorial optimisation tasks. I demonstrated how Wield facilitates a number of essential design components to accelerate building and evaluating RL models. The evaluation has highlighted both the potential of RL in a systems automation workflow, and the difficulties arising when evaluating such models:

- The high design cost combined with algorithmic sensitivity continues to restrict use-cases to scenarios where improvements create substantial value. Users must carefully weigh off potential gains versus hand-crafted heuristics.
- Progressive randomisation helps understand where a model's capabilities may fit in compared to an existing heuristic. As a classification mechanism, It helps assess if prior work is suitable for a novel application.
- Wield enables encoding of pre-existing domain knowledge through labelling rules or direct demonstrations (per-instance). The practical utility of demonstrations is limited by brute-force demonstration algorithms. Experiments here were limited by lack of high-quality human demonstrations which are commonly used in high-value domains such as autonomous vehicles [BKO18].

Together, these results show the need stated in my thesis for identifying effective evaluation protocols and task design mechanisms. In the final chapter, I conclude the findings of my dissertation and discuss further future directions.

Chapter 7

Conclusion and future work

Rapid developments in reinforcement learning applications have created the need for software systems supporting these new workloads. For systems designers, they also pose the question how new learning techniques can be used to enhance or replace traditional algorithms. In Chapter 3, I discussed how RL workloads differ from supervised workloads, and analysed the design challenges arising from brittle, fast evolving algorithms. To address these challenges, this dissertation has introduced a modular programming model which enforces a strict separation of concerns:

- Decoupling execution patterns from modularised algorithmic components enables fast exploration of new agent designs.
- By enforcing a strict build mechanism with typed dataflow, RLgraph provides incremental testing of performance-critical heuristics.
- Results show significant improvements in training performance compared to unstructured implementations.

RLgraph's evaluation highlights the original motivation for this thesis regarding the need for novel systems abstractions which can accommodate RL workloads. As the field continues to progress at accelerating pace, it is likely that current systems will be replaced by more automated designs. I view RLgraph as a point in the design space built on principles which can inform future iterations. In Section 7.1, I discuss directions for extending RLgraph, and more broadly consider future systems design for RL.

In Chapter 5, I began from the observation of limited real-world systems use cases despite a wealth of research interest and experimental successes. I argued that successful RL applications were not solely a question of increased data processing capabilities. Instead, I hypothesised that there was both a lack of shared design understanding and missing clarity on the capabilities and use cases of RL in the systems community. In my analysis of prior work, I identified a lack of software abstractions for systematic task design and evaluation with consideration for varying levels of non-determinism.

With Wield, I introduced a software approach based on decoupling programmatic systems design aspects from reinforcement learning representation. As part of Wield, I further proposed a classification scheme to help delineate model capabilities under different task determinism assumptions. In Wield's evaluation, I illustrated its practical utility through two different case studies.

In conclusion, the contributions of this work serve to demonstrate the thesis stated in the introduction. First, that decoupling algorithm logic and execution facilitates robust scalable implementations by resolving tension between prototyping and distributed execution. Second, that novel evaluation protocols and implementation tools decoupling system-specific protocols from model representations are required to systematically evaluate RL applications in systems. While this dissertation has focused on systems design, the implementation and evaluation techniques I presented can serve as the foundation for novel algorithmic research. Without systematic algorithm construction, practitioners must rely on brittle one-off implementations. Without systematic randomisation, they can over-or underestimate performance improvements on fixed workloads.

7.1 Extending RLgraph

7.1.1 Programming models

RLgraph as a programming model has been designed around the limitations of first generation machine learning frameworks. TensorFlow as one of the most popular machine learning frameworks centred around manually constructing a static computation graph with complex asynchronous state manipulation in an imperative host language. While PyTorch's define by-run mechanism allows more flexible construction of dynamic neural network approaches, static graphs allow compile-time optimisations and simplify deployment. Even though RLgraph's separation of algorithm logic and backend implementation is backend-agnostic, much design efforts went towards mapping non-differentiable imperative code to an end-to-end differentiable static graph with dynamic in-graph control flow.

Recent approaches seek to simplify the generation of differentiable static graphs. With TensorFlow eager [AMP+19] in combination with AutoGraph [MDW+18] for automatic graph generation, framework designers propose to combine the flexibility of define-by-run approaches with automated graph generation. More radical approaches seek to integrate differentiability into language design itself [IEF+19].

For RL software design, the gap between prototyping approaches in define-by-run and more scalable static graph approaches may shrink. In RLgraph, fully embracing define-by-run or eager designs means the build phases could be simplified. The main build phase could be used to test functionality, and otherwise use JIT tracing or AutoGraph-like functionality to extract a graph definition. RLgraph's incremental subgraph instantiation mechanisms makes it exceptionally suitable to investigate automated graph rewriting, architecture search, and randomised test mechanisms.

7.1.2 Execution models and hardware

RLgraph focuses on single-learner execution models which remain the dominant paradigm in applications. In Wield, I proposed a simplified task graph for multi-learner tasks design which however assumes co-located execution. The purpose of Wield's task decomposition is sample efficiency in centralised scenarios (e.g. device placement).

Systems software supporting multi-learner models with arbitrary inter-agent dependencies will be required to support decentralised deep reinforcement learning applications. Emerging accelerator hardware for low-power neural network execution may give rise to 'RL on the edge' scenarios. Current RL frameworks often do not account for deployment

due to the limited real-world viability of many approaches. As algorithms improve, I expect RL frameworks to include considerations for custom hardware compilation and model minimisation which are prevalent in supervised training pipelines.

7.2 RL applications and Wield

7.2.1 Model-based planning

This dissertation has concentrated on model-free deep RL as the main driver of recent applications in systems research. In the wider RL domain, the ability to plan and reason about the world through an internal model is widely accepted as a core element of future intelligent agents [PLV+17, HS18, KBM+19].

In systems research, incorporating planning into deep RL applications has been scarcely utilised, despite workload forecasting being used in large scale resource management (§5.1). One reason for this may be an initial research focus on deterministic problems. For example, both in device placement and in the indexing case study, state transitions are deterministically computed via updated device state of the graph or updated query/index encoding respectively.

A model predicting future states is hence not required. A model predicting execution times based on state and action is valuable in the absence of a simulator but may much increase training data requirements. Thus far, researchers have opted to either directly experiment on real systems to evaluate models, or to construct high-fidelity simulators which may be calibrated through a small number of real executions [AVG+19]. Future work may seek to jointly train execution prediction models with policies, in particular for stochastic environments based on partially observable workload generation mechanisms. A counter-argument is that the substantial experience required to train a world model may be directly used to learn a controller.

7.2.2 Integrating domain expertise

In this dissertation, I have explored the use of rule-based demonstrations as a means of inductive bias. Lately, the notion of combinatorial generalisation through structured representation has come into focus [BHB+18, VCC+18]. A limitation of structured representations is that users must manually encode structured representations by defining entities and relations.

The difficulty in automating these approaches is the relative ease with which human experts can define important entities and relationships, as compared to the cost of discovering them from unstructured system data. Ample room exists for future systems software to enables experts to more effectively encode domain knowledge into structural representations or other form of imperfect demonstrations, e.g. to facilitate transfer learning between similar systems. The use of imitation learning in systems is hindered by the difficulty with which demonstrations are generated.

7.3 Lessons learned

Conducting research in an emerging discipline brings both opportunities and additional uncertainty. Deep reinforcement learning in particular is characterised by extremes. When beginning this thesis work, high profile successes in narrow domains had given rise to out-sized expectations on real world applications. I was not exempt from these expectations.

In computer systems, outlooks were also fuelled by new literature focusing on out-performing benchmarks while being short on experimental cost and algorithmic limitations. My own work thus started with a sense of cognitive dissonance. How could simple use cases be so sensitive to small variations and hyper-parameters, while using the same algorithms which had produced flagship successes? To resolve these observations, I first looked towards more complexity as a solution, and attempted to layer additional techniques upon already exquisitely complicated methods.

When this failed to provide clarity, my work eventually turned from applications towards building blocks for RL. I imagined tools would provide greater levers for others, while also de-risking myself against difficulties on application work. However, the fast-moving nature of machine learning research means any tool will eventually be obsoleted by novel programming models and frameworks, and the best hope is to provide guidance for these future iterations. In Wield, this lesson led me to focus more on transferable concepts (progressive randomisation) rather than software.

Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016. Cited on pages 28, 41, 45, and 66.

[ACK+04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. Very Large Data Bases Endowment Inc., August 2004. Cited on page 108.

[AHM+19] Ameer Haj Ali, Qijing Huang, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. AutoPhase: Compiler Phase-Ordering for High Level Synthesis with Deep Reinforcement Learning. *CoRR*, abs/1901.04615, 2019. Cited on page 33.

[AIM17] Martín Abadi, Michael Isard, and Derek G. Murray. A Computational Model for TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2017, pages 1–7, New York, NY, USA, 2017. ACM. Cited on pages 18, 41, 59, and 83.

[AKV+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM. Cited on pages 90 and 120.

[ALC+17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*, pages 469–482, 2017. Cited on page 90.

[Ama18] Amazon Inc. Amazon DynamoDB. `https://aws.amazon.com/dynamodb/`, Last accessed 8th February 2020, 2018. Cited on page 109.

[AMP+19] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. *CoRR*, abs/1903.01855, 2019. Cited on page 140.

[ARR+16] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference workloads for modern deep learning methods. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pages 1–10. IEEE, 2016. Cited on page 54.

[Aud14] Charles Audet. A survey on direct search methods for blackbox optimization and their applications. In *Mathematics without boundaries*, pages 31–56. Springer, 2014. Cited on page 31.

[AVG+19] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. *CoRR*, abs/1906.08879, 2019. Cited on pages 102, 104, 133, 137, and 141.

[BBBK11] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011. Cited on page 32.

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. Cited on pages 17 and 27.

[BCG+18] Marc G. Bellemare, Pablo Samuel C Castro, Carles Gelada, Saurabh Kumar, and Subhodeep Moitra. Dopamine. `https://github.com/google/dopamine`, 2018. Cited on page 41.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016. Cited on page 36.

[BDM17] Marc G. Bellemare, Will Dabney, and Rémi Munos. A Distributional Perspective on Reinforcement Learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 449–458. PMLR, 2017. Cited on page 30.

[BFT+17] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement Learning through Asynchronous Advantage Actor-Critic on a GPU. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. Cited on page 40.

[BGLL17] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*, 2017. Cited on page 27.

[BHB+18] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018. Cited on pages 137 and 141.

[BHP17] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The Option-Critic Architecture. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 1726–1734. AAAI Press, 2017. Cited on page 97.

[BKO18] Mayank Bansal, Alex Krizhevsky, and Abhijit S. Ogale. ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst. *CoRR*, abs/1812.03079, 2018. Cited on page 138.

[BL94] Justin A Boyan and Michael L Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems*, pages 671–671, 1994. Cited on page 32.

[BL14] Emma Brunskill and Lihong Li. PAC-inspired Option Discovery in Lifelong Reinforcement Learning. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 316–324. JMLR.org, 2014. Cited on page 97.

[BLT+16] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016. Cited on page 38.

[BNVB13] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013. Cited on page 36.

[C+15] François Chollet et al. Keras. `https://keras.io`, 2015. Cited on pages 41 and 46.

[CKH+18] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying Generalization in Reinforcement Learning. *arXiv preprint arXiv:1812.02341*, 2018. Cited on page 38.

[CKN+18] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Christopher Ré, and Matei Zaharia. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *CoRR*, abs/1806.01427, 2018. Cited on page 100.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015. Cited on pages 28 and 41.

[CLN17] Itai Caspi, Gal Leibovich, and Gal Novik. Reinforcement Learning Coach. https://github.com/NervanaSystems/coach, December 2017. Cited on pages 42 and 58.

[CMC17] Alfredo V. Clemente, Humberto Nicolás Castejón Martínez, and Arjun Chandra. Efficient Parallel Methods for Deep Reinforcement Learning. *CoRR*, abs/1705.04862, 2017. Cited on pages 40 and 61.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *http://arxiv.org/abs/1802.04799v2*, 2018. Cited on pages 54 and 73.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, pages 143–154, 2010. Cited on page 99.

[DAEvH+15] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. 2015. Cited on page 95.

[DCH+16] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking Deep Reinforcement Learning for Continuous Control. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1329–1338. JMLR.org, 2016. Cited on page 36.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012. Cited on page 65.

[DDD+04] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic SQL Tuning in Oracle 10G. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1098–1109. VLDB Endowment, 2004. Cited on page 108.

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009. Cited on page 105.

[Dee17] DeepMind. Sonnet: TensorFlow-based neural network library. `https://github.com/deepmind/sonnet`, 2017. Cited on page 46.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`, 2017. Cited on pages 41, 58, 65, and 98.

[DK13] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013. Cited on page 91.

[DK14] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGPLAN Notices*, volume 49, pages 127–144. ACM, 2014. Cited on page 91.

[DPP+18] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S. Sekeran, Fabián Rodriguez, and Laxmi Balami. GridFormation: Towards Self-Driven Online Data Partitioning using Reinforcement Learning. In Rajesh Bordawekar and Oded Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 1:1–1:7. ACM, 2018. Cited on page 33.

[DR11] Marc Peter Deisenroth and Carl Edward Rasmussen. PILCO: A Model-Based and Data-Efficient Approach to Policy Search. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 465–472. Omnipress, 2011. Cited on page 31.

[DSY16] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Tuning the Scheduling of Distributed Stochastic Gradient Descent with Bayesian Optimization. *CoRR*, abs/1612.00383, 2016. Cited on page 21.

[DSY17] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. BOAT: Building Auto-Tuners with Structured Bayesian Optimization. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 479–488, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee. Cited on pages 21 and 90.

[DYY+19] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. *CoRR*, abs/1901.02860, 2019. Cited on page 137.

[ESM+18] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. *CoRR*, abs/1802.01561, 2018. Cited on pages 40, 61, 62, 66, 68, and 85.

[FAdFW16] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to Communicate with Deep Multi-Agent Reinforcement Learning. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2137–2145, 2016. Cited on page 65.

[FdWF+18] Jakob N. Foerster, Christian A. Schröder de Witt, Gregory Farquhar, Philip H. S. Torr, Wendelin Boehmer, and Shimon Whiteson. Multi-Agent Common Knowledge Reinforcement Learning. *CoRR*, abs/1810.11702, 2018. Cited on page 65.

[FL17] Chelsea Finn and Sergey Levine. Deep visual foresight for planning robot motion. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2786–2793. IEEE, 2017. Cited on pages 17 and 27.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010. Cited on page 67.

[GCL+18] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, and Xiaohui Ye. Horizon: Facebook's Open Source Applied Reinforcement Learning Platform. *CoRR*, abs/1811.00260, 2018. Cited on pages 17, 42, and 138.

[GK10] Goetz Graefe and Harumi Kuno. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 371–381, New York, NY, USA, 2010. ACM. Cited on page 108.

[GLG+17] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E. Turner, and Sergey Levine. Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic. In *Proceedings International Conference on Learning Representations (ICLR)*. OpenReviews.net, April 2017. Cited on page 27.

[GLT+17] S. Gu, T. Lillicrap, R. E. Turner, Z. Ghahramani, B. Schölkopf, and S. Levine. Interpolated Policy Gradient: Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems 30*, pages 3849–3858. Curran Associates, Inc., December 2017. Cited on page 27.

[Goo18] Google Inc. Google Cloud Datastore. `https://cloud.google.com/datastore/`, Last accessed 8th February 2020, 2018. Cited on page 109.

[GPLL17] Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational*

Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, pages 1051–1062. Association for Computational Linguistics, 2017. Cited on page 104.

[GPM+14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative Adversarial Networks. *CoRR*, abs/1406.2661, 2014. Cited on page 65.

[GSM+17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017. Cited on page 90.

[GSW+17] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. Quaestor: Query Web Caching for Database-as-a-Service Providers. *PVLDB*, 10(12):1670–1681, 2017. Cited on page 21.

[GZ18] Piotr Gawlowicz and Anatolij Zubow. ns3-gym: Extending OpenAI Gym for Networking Research. *CoRR*, abs/1810.03943, 2018. Cited on page 38.

[HDT+17] Daniel Hein, Stefan Depeweg, Michel Tokic, Steffen Udluft, Alexander Hentschel, Thomas A Runkler, and Volkmar Sterzing. A Benchmark Environment Motivated by Industrial Control Problems. *arXiv preprint arXiv:1709.09480*, 2017. Cited on page 38.

[HDV17] Danijar Hafner, James Davidson, and Vincent Vanhoucke. TensorFlow Agents: Efficient Batched Reinforcement Learning in TensorFlow. *CoRR*, abs/1709.02878, 2017. Cited on page 42.

[HE16] Jonathan Ho and Stefano Ermon. Generative Adversarial Imitation Learning. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4565–4573, 2016. Cited on page 108.

[HGS16] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016. Cited on pages 29, 42, 70, and 105.

[HIB+17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. *CoRR*, abs/1709.06560, 2017. Cited on pages 38, 42, 81, and 99.

[HIKY12] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-memory Column-stores. *Proc. VLDB Endow.*, 5(6):502–513, February 2012. Cited on page 108.

[HMvH+18] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3215–3222. AAAI Press, 2018. Cited on page 30.

[HNFM18] Chin-Jung Hsu, Vivek Nair, Vincent W. Freeh, and Tim Menzies. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 660–670. IEEE Computer Society, 2018. Cited on page 90.

[HQB+18] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed Prioritized Experience Replay. *CoRR*, abs/1803.00933, 2018. Cited on pages 39, 42, 61, 80, and 86.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. Cited on page 27.

[HS18] David Ha and Jürgen Schmidhuber. Recurrent World Models Facilitate Policy Evolution. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 2455–2467, 2018. Cited on pages 31, 94, 116, and 141.

[HVP+17] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from Demonstrations for Real World Reinforcement Learning. *CoRR*, abs/1704.03732, 2017. Cited on page 105.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *CoRR*, abs/1801.01290, 2018. Cited on page 86.

[HZH+18] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications. *CoRR*, abs/1812.05905, 2018. Cited on page 86.

[IEF+19] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckus, Elliot Saba, Viral B Shah, and Will Tebbutt. Zygote: A Differentiable Programming System to Bridge Machine Learning and Scientific Computing. *arXiv preprint arXiv:1907.07587*, 2019. Cited on page 140.

[IES+18] Andrew Ilyas, Logan Engstrom, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Are Deep Policy Gradient Algorithms Truly Policy Gradient Algorithms? *CoRR*, abs/1811.02553, 2018. Cited on pages 38 and 42.

[imd18] imdb.com. Imdb datasets. website, 2018. Cited on page 120.

[IMKG11] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-memory Column-stores. *Proc. VLDB Endow.*, 4(9):586–597, June 2011. Cited on page 108.

[JBV+18a] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. *CoRR*, abs/1809.02627, 2018. Cited on page 38.

[JBV+18b] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018. Cited on page 100.

[JDO+17] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *CoRR*, abs/1711.09846, 2017. Cited on page 73.

[JGP16] Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax. *CoRR*, abs/1611.01144, 2016. Cited on page 87.

[JHHB16] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The Malmo Platform for Artificial Intelligence Experimentation. In *IJCAI*, pages 4246–4247, 2016. Cited on page 100.

[JRG+19] Nathan Jay, Noga H. Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 3050–3059. PMLR, 2019. Cited on page 104.

[JSL+18] Junqi Jin, Chengru Song, Han Li, Kun Gai, Jun Wang, and Weinan Zhang. Real-Time Bidding with Multi-Agent Reinforcement Learning in Display Advertising. In Alfredo Cuzzocrea, James Allan, Norman W. Paton, Divesh Srivastava, Rakesh Agrawal, Andrei Z. Broder, Mohammed J. Zaki, K. Selçuk Candan, Alexandros Labrinidis, Assaf Schuster, and Haixun Wang, editors, *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 2193–2201. ACM, 2018. Cited on page 138.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden,

Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12. ACM, 2017. Cited on page 40.

[JYPP18] Norman P. Jouppi, Cliff Young, Nishant Patil, and David A. Patterson. Motivation for and Evaluation of the First Tensor Processing Unit. *IEEE Micro*, 38(3):10–19, 2018. Cited on page 40.

[JZA18] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. *http://arxiv.org/pdf/1807.05358v1*, 2018. Cited on pages 33 and 73.

[KARS15] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold Them or Fold Them? Aggregation Queries under Performance Variations. Technical report, UC Berkeley TR UCB/EECS-2015-267, 2015. Cited on page 91.

[KBM+19] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for Atari. *arXiv preprint arXiv:1903.00374*, 2019. Cited on page 141.

[KBP13] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *I. J. Robotics Res.*, 32(11):1238–1274, 2013. Cited on page 31.

[KM97] Shailesh Kumar and Risto Miikkulainen. Dual reinforcement q-routing: An on-line adaptive routing algorithm. In *Artificial neural networks in engineering*, 1997. Cited on page 32.

[KM99] Shailesh Kumar and Risto Miikkulainen. Confidence based dual reinforcement q-routing: An adaptive online network routing algorithm. In *IJCAI*, volume 99, pages 758–763. Citeseer, 1999. Cited on page 32.

[KOQ+19] Steven Kapturowski, Georg Ostrovski, John Quan, Rémi Munos, and Will Dabney. Recurrent Experience Replay in Distributed Reinforcement Learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. Cited on page 39.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. Cited on pages 17 and 27.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, May 2017. Cited on page 27.

[KSM+17] Ken Kansky, Tom Silver, David A. Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott Phoenix, and Dileep George. Schema Networks: Zero-shot Transfer with a Generative Causal Model of Intuitive Physics. 2017. Cited on page 74.

[KW14] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. Cited on page 94.

[KYG+18] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR*, abs/1808.03196, 2018. Cited on page 33.

[LAP+14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, volume 14, pages 583–598, 2014. Cited on pages 62 and 65.

[LBM+17] Nicholas D. Lane, Sourav Bhattacharya, Akhil Mathur, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. Squeezing Deep Learning into Mobile and Embedded Devices. *IEEE Pervasive Computing*, 16(3):82–88, 2017. Cited on page 73.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015. Cited on pages 99, 102, and 128.

[LHP+16] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. Cited on page 42.

[Lin93] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993. Cited on page 28.

[LLN+18] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning*, pages 3059–3068, 2018. Cited on pages 42, 58, 61, 65, and 80.

[LMK+17] Jan Leike, Miljan Martic, Victoria Krakovna, Pedro A. Ortega, Tom Everitt, Andrew Lefrancq, Laurent Orseau, and Shane Legg. AI Safety Gridworlds. *CoRR*, abs/1711.09883, 2017. Cited on page 38.

[LPK+18] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018. Cited on page 27.

[LT19] Liam Li and Ameet Talwalkar. Random Search and Reproducibility for Neural Architecture Search. *CoRR*, abs/1902.07638, 2019. Cited on page 92.

[LXTW18] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free Control for Distributed Stream Data Processing using Deep Reinforcement Learning. *PVLDB*, 11(6):705–718, 2018. Cited on page 33.

[LZJS19] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural Packet Classification. *CoRR*, abs/1902.10319, 2019. Cited on pages 33 and 104.

[MAMK16] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016. Cited on page 32.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. February 2016. Cited on page 38.

[MDW+18] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. AutoGraph: Imperative-style Coding with Graph-based Performance. *CoRR*, abs/1810.08061, 2018. Cited on pages 72 and 140.

[MGP+18] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. Hierarchical Planning for Device Placement. 2018. Cited on pages 17, 33, 54, 73, 98, 102, 104, 128, and 129.

[MGR18] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *CoRR*, abs/1803.07055, 2018. Cited on pages 32, 38, 75, 92, and 99.

[Mic18] Microsoft. CosmosDB - A globally distributed database for low latency and massively scalable applications, with native support for NoSQL. website, 2018. Cited on page 109.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013. Cited on page 28.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. Cited on pages 17, 28, 70, and 86.

[MNA17] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 197–210, New York, NY, USA, 2017. ACM. Cited on pages 17, 33, and 95.

[MNM+19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. *arXiv e-prints*, page arXiv:1904.03711, Apr 2019. Cited on pages 17, 33, 102, and 104.

[MNW+17] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. *CoRR*, abs/1712.05889, 2017. Cited on pages 54 and 65.

[MP18] Ryan Marcus and Olga Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In Rajesh Bordawekar and Oded Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018. Cited on pages 17, 33, 102, and 104.

[MPL+17] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. *arXiv preprint arXiv:1706.04972*, 2017. Cited on pages 17, 33, 102, 104, and 129.

[MSV+18a] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. *CoRR*, abs/1810.01963, 2018. Cited on pages 33 and 137.

[MSV+18b] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963*, 2018. Cited on page 104.

[MSV+19] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In Jianping Wu and Wendy Hall, editors, *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 270–288. ACM, 2019. Cited on page 17.

[MVSA18] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance Reduction for Reinforcement Learning in Input-Driven Environments. *CoRR*, abs/1807.02264, 2018. Cited on page 94.

[NCD+06] AndrewY. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous Inverted Helicopter Flight via Reinforcement Learning. In Jr. Ang, MarceloH. and Oussama Khatib, editors, *Experimental Robotics IX*, volume 21 of *Springer Tracts in Advanced Robotics*, pages 363–372. Springer Berlin Heidelberg, 2006. Cited on page 17.

[NGLL18] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-Efficient Hierarchical Reinforcement Learning. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 3307–3317, 2018. Cited on page 97.

[NMW+] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I. Jordan, and Ion Stoica. Real-Time Machine Learning: The Missing Pieces. Cited on page 42.

[NR00] Andrew Y. Ng and Stuart J. Russell. Algorithms for Inverse Reinforcement Learning. In Pat Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, pages 663–670. Morgan Kaufmann, 2000. Cited on page 107.

[NSB+15] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively Parallel Methods for Deep Reinforcement Learning. *CoRR*, abs/1507.04296, 2015. Cited on page 39.

[NWS18a] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. Deterministic Implementations for Reproducibility in Deep Reinforcement Learning. *CoRR*, abs/1809.05676, 2018. Cited on page 67.

[NWS18b] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. The Impact of Nondeterminism on Reproducibility in Deep Reinforcement Learning. *https://arxiv.org/abs/1809.05676*, 2018. Cited on pages 44 and 67.

[OBGK18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In Sebastian Schelter, Stephan Seufert, and Arun Kumar, editors, *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 4:1–4:4. ACM, 2018. Cited on page 104.

[ODH+] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepezvari, Satinder Singh, Benjamin Van Roy, Richard Sutton, David Silver, and Hado Van Hasselt. Behaviour Suite for Reinforcement Learning. Cited on page 100.

[Ope18] OpenAI. OpenAI Five DOTA. `https://openai.com/blog/openai-five/`, Last accessed 8th February 2020, June 2018. Cited on page 119.

[PAA+17] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-Driving Database Management Systems. In *CIDR*, 2017. Cited on page 17.

[PBC+18] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 3:1–3:14. ACM, 2018. Cited on page 91.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017. Cited on pages 28 and 41.

[PGN+19] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. REGAL: Transfer Learning For Fast Optimization of Computation Graphs. *CoRR*, abs/1905.02494, 2019. Cited on pages 102 and 104.

[PGP14] Bilal Piot, Matthieu Geist, and Olivier Pietquin. Boosted Bellman residual minimization handling expert demonstrations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 549–564. Springer, 2014. Cited on page 106.

[PIM15] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1153–1166, New York, NY, USA, 2015. ACM. Cited on pages 108 and 109.

[Pla16] Matthias Plappert. keras-rl. `https://github.com/matthiasplappert/keras-rl`, 2016. Cited on page 42.

[PLT+16] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I Jordan, Kannan Ramchandran, and Christopher Ré.

Cyclades: Conflict-free asynchronous machine learning. In *Advances in Neural Information Processing Systems*, pages 2568–2576, 2016. Cited on page 65.

[PLV+17] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sebastien Racanière, David Reichert, Théophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*, 2017. Cited on page 141.

[PTS+17] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017. Cited on page 45.

[R+18] Steffen Rochel et al. Gluon - A clear, concise, simple yet powerful and efficient API for deep learning. `https://github.com/gluon-api/gluon-api`, 2018. Cited on page 46.

[RBE+17] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment*, 11(3):269–282, 2017. Cited on pages 105 and 107.

[REH+18] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. Pommerman: A Multi-Agent Playground. In Jichen Zhu, editor, *Joint Proceedings of the AIIDE 2018 Workshops co-located with 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2018), Edmonton, Canada, November 13-14, 2018.*, volume 2282 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. Cited on page 65.

[RGB11] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 627–635. JMLR.org, 2011. Cited on page 107.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011. Cited on page 65.

[RW06] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006. Cited on page 31.

[SA16] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International*

Conference on Knowledge Discovery and Data Mining, pages 2135–2135. ACM, 2016. Cited on pages 28 and 41.

[SB17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction, Second edition*, volume 1. MIT press Cambridge, 2017. Cited on pages 23, 25, 26, 27, and 30.

[Sch15] Juergen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Cited on page 28.

[SFY19] Michael Schaarschmidt, Kai Fricke, and Eiko Yoneki. Wield: Systematic Reinforcement Learning With Progressive Randomization. *arXiv e-prints*, 2019. Cited on pages 20 and 90.

[SGDY16] Michael Schaarschmidt, Felix Gessert, Valentin Dalibard, and Eiko Yoneki. Learning Runtime Parameters in Computer Systems with Delayed Experience Injection. *NIPS Deep Reinforcement Learning Workshop*, 2016. Cited on page 20.

[SHCS17] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017. Cited on pages 32 and 75.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. Cited on page 17.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. Cited on pages 104 and 119.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. Cited on pages 31 and 90.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015. Cited on page 30.

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014. Cited on page 26.

[SMC+17] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *CoRR*, abs/1712.06567, 2017. Cited on page 75.

[SMFY19] Michael Schaarschmidt, Sven Mika, Kai Fricke, and Eiko Yoneki. RLgraph: Modular Computation Graphs for Deep Reinforcement Learning. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML)*, April 2019. Cited on pages 20 and 35.

[SML+15] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *CoRR*, abs/1506.02438, 2015. Cited on page 27.

[SMS+99] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *NIPS*, volume 99, pages 1057–1063, 1999. Cited on page 27.

[SPS99] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artif. Intell.*, 112(1-2):181–211, 1999. Cited on page 97.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *ICLR 2016*, November 2015. Cited on pages 29, 46, and 70.

[SRdW+19] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft Multi-Agent Challenge. *CoRR*, abs/1902.04043, 2019. Cited on page 74.

[SSD] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The Case for Automatic Database Administration using Deep Reinforcement Learning. Cited on pages 33, 95, 108, and 121.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017. Cited on page 17.

[SSW+16] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016. Cited on page 31.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017. Cited on pages 30, 42, and 65.

[SZ14] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *http://arxiv.org/pdf/1409.1556v6*, 2014. Cited on page 27.

[SZ15] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. Cited on page 17.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*, pages 5026–5033. IEEE, 2012. Cited on page 36.

[TJDB06] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ICAC '06, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society. Cited on pages 32 and 94.

[Tra10] Transaction Processing Performance Council. TPC benchmark C (standard specification, revision 5.11), 2010. *URL: http://www. tpc. org/tpcc*, 2010. Cited on pages 91 and 99.

[TTH16] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 3. ACM, 2016. Cited on page 91.

[VAPGZ17] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017. Cited on pages 90 and 116.

[VCC+18] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. Cited on page 141.

[VOS+17] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. FeUdal Networks for Hierarchical Reinforcement Learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3540–3549. PMLR, 2017. Cited on page 97.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors,

Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA, pages 5998–6008, 2017. Cited on pages 17 and 137.

[VSST17] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. A Machine Learning Approach to Routing. *arXiv preprint arXiv:1708.03074*, 2017. Cited on page 33.

[VYF+16] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016. Cited on page 90.

[WCB+19] Yueh-Hua Wu, Nontawat Charoenphakdee, Han Bao, Voot Tangkaratt, and Masashi Sugiyama. Imitation Learning from Imperfect Demonstration. *arXiv preprint arXiv:1901.09387*, 2019. Cited on page 108.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992. Cited on pages 25 and 26.

[WMG+17] Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5279–5288, 2017. Cited on page 40.

[WMR+17] Ziyu Wang, Josh Merel, Scott E. Reed, Nando de Freitas, Gregory Wayne, and Nicolas Heess. Robust Imitation of Diverse Behaviors. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5320–5329, 2017. Cited on page 108.

[WRR+17] Theophane Weber, Sébastien Racanière, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, David Silver, and Daan Wierstra. Imagination-Augmented Agents for Deep Reinforcement Learning. *CoRR*, abs/1707.06203, 2017. Cited on page 31.

[WSC+16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016. Cited on page 27.

[WSH+16] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning,*

ICML 2016, New York City, NY, USA, June 19-24, 2016, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1995–2003. JMLR.org, 2016. Cited on pages 30, 71, and 86.

[WSY19] Jeremy Welborn, Michael Schaarschmidt, and Eiko Yoneki. Learning Index Selection with Structured Action Spaces. *CoRR*, abs/1909.07440, 2019. Cited on pages 20 and 116.

[YAB+18] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic Control Flow in Large-scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 18:1–18:15, New York, NY, USA, 2018. ACM. Cited on pages 45, 55, 66, and 83.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010. Cited on page 33.

[ZGMX18] Yiren Zhao, Xitong Gao, Robert Mullins, and Chengzhong Xu. Mayo: A Framework for Auto-generating Hardware Friendly Deep Neural Networks. In *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning, EMDL@MobiSys 2018, Munich, Germany, June 15, 2018*, pages 25–30. ACM, 2018. Cited on page 73.

[ZL17] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. Cited on page 92.

[ZMBD08] Brian D. Ziebart, Andrew L. Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum Entropy Inverse Reinforcement Learning. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1433–1438. AAAI Press, 2008. Cited on page 86.