RL-Scope: Cross-Stack Profiling for Deep Reinforcement Learning Workloads

James Gleeson [1] Srivatsan Krishnan [2] Moshe Gabel [1] Vijay Janapa Reddi [2] Eyal de Lara [1] Gennady Pekhimenko [1]

Abstract

In recent years, deep reinforcement learning (RL) has demonstrated groundbreaking results in robotics, datacenter management, and many other applications. Despite its increasing popularity, there has been little work in understanding system-level bottlenecks in RL workloads. Instead, the common implicit assumption is that RL workloads are similar to classic supervised learning (SL) workloads. Our analysis contradicts this assumption and shows operations considered GPU-heavy in SL spend at most 12.9% of time GPU-bound in RL workloads, with the rest CPU-bound in different layers of the software stack running high-level language code and non-compute code such as ML backend and CUDA API calls.

To explain where training time is spent in RL workloads, we propose RL-Scope: an accurate cross-stack profiler that supports multiple ML backends and simulators. In contrast to existing profilers that are limited to a single layer of the software and hardware stack, RL-Scope collects profiling information across the entire stack and scopes it to high-level operations, providing developers and researchers with a complete picture of RL training time.

We demonstrate RL-Scope's utility through several in-depth case studies. First, we compare RL frameworks to quantify the effects of fundamental design choices behind ML backends. For example, we use RL-Scope to measure and explain a $2.3\times$ difference in runtime between equivalent PyTorch and TensorFlow algorithm implementations, and to identify a bottleneck rooted in overly-abstracted algorithm implementations. Next, we survey how training bottlenecks change as we consider different simulators and RL algorithms, and show that on-policy algorithms are at least $3.5\times$ more simulation-bound than off-policy algorithms. Finally, we profile a scale-up workload and demonstrate that GPU utilization metrics reported by commonly-used tools dramatically inflate GPU usage, whereas RL-Scope reports true GPU-bound time. RL-Scope is an open-source tool available at `https://github.com/UofT-EcoSystem/rlscope`.

1 Introduction

Deep Reinforcement Learning (RL) has made many algorithmic advancements in the past decade such as learning to play Atari games from raw pixels (Mnih et al., 2015), surpassing human performance on games with intractably large state spaces (Silver et al., 2017), as well as in diverse industrial applications including robotics (Brockman et al., 2016), data center management (Mirhoseini et al., 2017), autonomous driving (Dosovitskiy et al., 2017), and drone tasks (Krishnan et al., 2019).

Despite their promise, RL models are notoriously slow to train with AlphaGoZero taking 40 days to train (Silver et al., 2017); to make matters worse, most benchmark studies of RL training focus only on model accuracy as a function of training steps (Duan et al., 2016; Fujimoto et al., 2019) not training runtime leading the community to *believe* they behave similarly to supervised learning (SL) workloads. In SL training workloads like image recognition, neural networks are large (e.g., 152 layers (He et al., 2016)) and runtime is largely accelerator-bound spending at least 53.7% of total training time executing on the GPU (Li et al., 2020). In contrast to SL workloads, we observe that RL workloads are fundamentally different in structure, spending a large portion of their training time on the CPU collecting training data through simulation, running high-level language code (e.g., Python) inside the training loop, and executing relatively small neural networks (e.g., AlphaGoZero (Silver et al., 2017) has only 39 layers).

Identifying the precise reasons for poor RL training runtime remains a challenge since existing profiling tools are

[1]Department of Computer Science, University of Toronto, Toronto, Ontario, Canada [2]School of Engineering and Applied Sciences, Harvard, Cambridge, Massachusetts, USA. Correspondence to: James Gleeson <jgleeson@cs.toronto.edu>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

designed for GPU-bound SL workloads and are not suitable for RL. General purpose GPU profiling tools (NVIDIA, 2020b) are unsuitable for two reasons. First, they mainly focus on GPU kernel metrics and low-level system calls with little context about which operation in high-level code they originate from. Second, they make no effort to correct for CPU profiling overhead that can inflate RL workloads significantly; we observe up to a $1.9\times$ inflation of total training time when profiling RL workloads. Even specialized machine learning (ML) profiling tools only target GPU-bound SL workloads and limit their analysis to total training time spent in neural network layers and bottleneck kernels within each layer (NVIDIA, 2020a; Li et al., 2020). In contrast, deep RL training workloads execute a diverse software stack that includes algorithms, simulators, ML backends, accelerator APIs, and GPU kernels, that are written in a mix of high- and low-level languages, and are executed asynchronously across single and even multiple processes resulting in CPU/GPU overlap. Hence, identifying bottlenecks in RL workloads requires a *cross-stack* profiler that can accurately separate both GPU and CPU resource usage at each different level, and correlate low-level CPU/GPU execution time with RL algorithmic operations.

To address this challenge, we present *RL-Scope* – an open-source, accurate, cross-stack, cross-backend tool for profiling RL workloads. RL-Scope provides ML researchers and developers with a simple Python API for annotating their high-level code with operations. RL-Scope collects cross-stack profiling information by scoping time spent in low-level GPU kernels, CUDA API calls, simulators, and ML backends to user annotations in high-level code. To ensure accurate insights during offline analysis, RL-Scope calibrates and corrects for CPU time inflation introduced by book-keeping code, correcting within 16% of profiling overhead. RL-Scope easily supports multiple ML backends (TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019)) and simulators, as it requires no recompilation of ML backends or simulators.

In contrast to prior studies that have been limited to GPU-bound SL training workloads (Li et al., 2020) and RL model accuracy as a function of training steps (Duan et al., 2016; Fujimoto et al., 2019), we use RL-Scope to perform the first cross-stack study of RL training runtime. In our first case study, we use RL-Scope to compare RL frameworks and analyze bottlenecks. RL-Scope can help practitioners decide between a myriad of potential RL frameworks spanning multiple ML backends with diverse execution models ranging from developer-friendly Eager execution popularized by PyTorch to the more sophisticated Autograph execution in TensorFlow that converts high-level code to in-graph operators. RL-Scope collects fine-grained metrics that quantify the intuition of why different execution models like Eager perform up to $4.8\times$ worse than Graph/Autograph, and uses these metrics to further explain a $2.3\times$ difference in runtime between a TensorFlow implementation and PyTorch implementation of the Eager execution model. RL-Scope can detect and analyze bottlenecks in RL algorithm implementations; RL-Scope's metrics detect a $3.7\times$ inflation in backpropagation rooted in an overly abstracted MPI-friendly but very GPU-unfriendly Adam optimizer. Finally, RL-Scope illustrates that GPU usage is poor across *all* RL frameworks regardless of ML backend or execution model.

In our next case study, we survey RL workloads across different RL algorithms and simulators, which can help researchers decide where to devote their efforts in optimizing RL workloads by surveying how training bottlenecks change. We observe that simulation time is non-negligible, taking up at least 38.1% and at most 74.2% of training runtime, with simulation time being higher for on-policy RL algorithms typically used in robotics tasks. Operations considered GPU-heavy in SL workloads (i.e., inference, backpropagation) only spend at most 12.9% of their time executing GPU kernels with the rest spent on the CPU in ML backend and CUDA API calls.

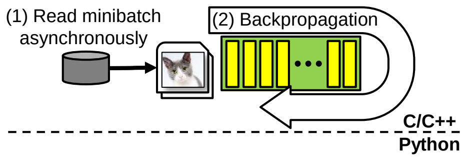
Our final case study examines a scale-up RL workload that increases GPU utilization by parallelizing inference operations. We find that coarse-grained GPU utilization metrics collected using common tools (e.g, `nvidia-smi`) can be a poor indicator of actual time spent executing GPU kernels on the GPU in the RL context, whereas RL-Scope is able to identify the true GPU-bound time.

To summarize, our contributions are:

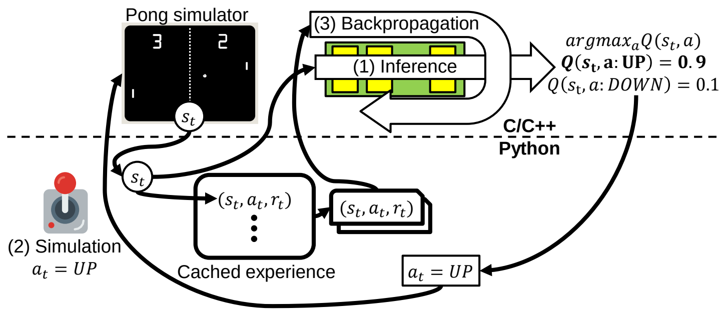
- We observe fundamental structural differences between RL and SL training workloads that leads to more time spent CPU-bound in RL and makes them poorly suited to existing profiling tools that target GPU-bound SL workloads (Section 2).
- We propose RL-Scope: the first profiler designed specifically for RL workloads, featuring: (i) scoping of high- and low-level GPU and CPU time across the stack to algorithmic operations, and (ii) low CPU profiling overhead, and (iii) support for multiple ML backends and simulators (Section 3). To help the community with measuring their own RL workloads, we have open-sourced RL-Scope at `https://github.com/UofT-EcoSystem/rlscope`.
- The first cross-stack study of RL training runtime (i) across multiple RL frameworks and ML backends, (ii) across multiple RL algorithms and simulators, (iii) in scale-up RL workloads. Prior studies have been limited to SL training workloads, and RL model accuracy as a function of training steps (Section 4).

2 Background and Motivation

Using a simplified version of DQN (Mnih et al., 2015) on the Atari Pong simulator as an example RL workload, we



(a) Supervised learning (SL) training loop (e.g., image recognition)



(b) Reinforcement learning (RL) training loop (e.g., Atari Pong trained with DQN)

Figure 1. Training loop comparison: the SL training loop is more GPU-bound whereas RL computation is spread across the software and hardware stack. Supervised learning workloads (1a) spend most of their time GPU-bound: they stream training data into large scale neural networks on the GPU, while high-level language is kept out of the training loop. Conversely, RL workloads frequently transition between high-level $\leftrightarrow$ native code in their training loop, collect training data at runtime from a simulator, and use small neural networks.

outline the typical training loop of an RL workload, and showcase key differences between RL and supervised learning (SL) workloads. These differences inform our design of the RL-Scope profiling toolkit for RL workloads.

2.1 DQN – an Example RL Workload

The goal of the DQN algorithm is to learn to estimate the Q-value function $Q(s, a)$: the expected *reward* if at state s an agent takes action a, and repeats this until the simulation terminates. A reward for Pong would be 1 if the agent won the game, or 0 if it lost. DQN learns $Q(s, a)$ by constructing training minibatches sampled from a cache of *experience tuples* – past states, actions taken, estimated rewards, and resulting rewards – and applying the standard backpropagation training algorithm. Once learned, a deployed model runs inference to obtain $Q(s, a)$, and greedily selects the action that maximizes expected reward.

Code for training RL algorithms is often implemented in a high-level language such as Python, with calls to a native simulator (e.g., Atari emulator) and ML backend library (e.g., TensorFlow). Hence, several steps require *transitioning* from the high-level language to a native library and marshaling data between them. Other RL algorithms have a similar overall structure and can be grouped into the same components as DQN. Figure 1b illustrates the *high-level algorithmic operations* of the DQN training loop:

1. **Inference:** Given the latest state s_t as input (e.g., rendered pixels from the Atari emulator), perform an inference operation to predict $Q(s_t, a)$ for each possible action a. To ensure adequate exploration and convergence of $Q(s, a)$, we occasionally select a random action with probability ϵ, or the greedy action with the highest reward $Q(s, a)$ with probability $1 - \epsilon$.
2. **Simulation:** Call the simulator (e.g., Atari emulator C++ library) using the selected action (e.g., $a_t = UP$) as input, and receive the next state s_{t+1}.
3. **Backpropagation:** Form a minibatch of $Q(s, a)$ samples with predicted and actual rewards from cached experience tuples, and perform the forward, background, and gradient updates of the Q-value network by calling into the C++ ML backend.

2.2 Comparing SL and RL Training

Comparing the training loop of supervised learning (SL) (Figure 1a) to the training loop of RL (Figure 1b), clear differences in CPU/GPU execution become apparent.

First, RL workloads collect training data at runtime by running a simulator inside the training loop, dramatically increasing CPU runtime. In contrast, SL workloads are more GPU-bound since they use large pre-labeled training datasets that can be streamed asynchronously onto the GPU while forward/backward GPU compute passes are in progress.

Second, RL workloads run high-level language code inside the training loop. RL workloads use a high-level language (e.g., Python) to orchestrate collecting training data from simulators and store them in a data structure (*replay buffer*), to be later sampled from by high-level code at each backpropagation training step. In contrast, SL workloads statically define a computational graph ahead of time, then offload the execution of forward/backward passes to the ML backend without involving high-level code.

Finally, the deep neural networks in most RL workloads are smaller than typically used for SL workloads like image recognition. SL workloads consist of an extremely large number of layers, making their forward/backward training loop heavily GPU-bound. For instance, ResNet-152 (He et al., 2016), a popular image recognition model, has 152 layers, whereas the model used in AlphaGoZero (Silver et al., 2017) only contains 39 layers. Hence, while the sizes of networks in RL workloads are increasing, they have not reached near the capacity of SL workloads, and as a result, they are not usually bottlenecked by GPU runtime.

Due to these characteristics, the RL training loop is less GPU-bound and time spent on the CPU plays a crucial role on execution time. The inevitable question that arises is how much of an RL workload is CPU-bound, and how much is GPU-bound? How does this breakdown depend on the choice of algorithm and simulator? Which particular high-level algorithmic operation is responsible for the most CPU latency, or the most GPU latency? Do excessive language *transitions* contribute to increased training time? RL-Scope is designed to answer these types of questions.

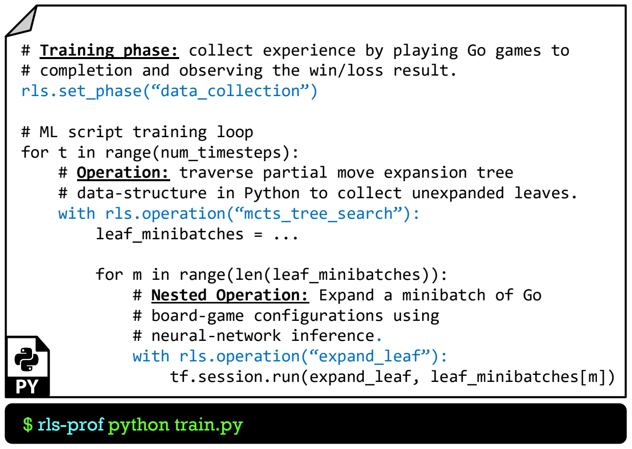
3 RL-SCOPE CROSS-STACK PROFILER

The RL-Scope profiler satisfies three high-level design goals. First, RL-Scope provides a cross-stack view of CPU/GPU resource usage by collecting information at each layer of the RL software stack and scoping fine-grained latency information to high-level algorithmic operations. Second, to ease adoption, RL-Scope does not require recompilation or modification of ML backends or simulators, and as a result, can be ported to new simulators and future ML backends. Finally, RL-Scope corrects for profiling overhead to ensure accurate critical path latency measurements.

To achieve these goals, RL-Scope adopts several key implementation choices:

1. **High-level algorithmic annotations:** RL-Scope provides the developer with a simple API for annotating their code with high-level algorithmic operations.
2. **Transparent event interception:** RL-Scope uses transparent hooks to intercept profiling events across all other levels of the stack, requiring no additional effort from the user.
3. **Cross-stack event overlap:** RL-Scope sums up regions of overlap between cross-stack events, which simultaneously measures overlap between CPU and GPU resources, and scopes training time to high-level algorithmic operations.
4. **Profiling calibration and overhead correction:** RL-Scope runs the RL training workload multiple times to calibrate for the average time spent in profiler book-keeping code, which it uses during offline analysis to correct for CPU overhead caused by profiling. This calibration need only be done once per workload, and can be reused in future profiling runs.

Our current implementation targets Python as the high-level language, and supports both TensorFlow and PyTorch as ML backends, since these are the most common tools used by RL developers. However, our design ensures that porting RL-Scope to other high-level languages and ML backends is straightforward. Additional implementation details such as avoiding sampling profilers and storing trace files asynchronously are provided in Appendix A.



```
# Training phase: collect experience by playing Go games to
# completion and observing the win/loss result.
rls.set_phase("data_collection")

# ML script training loop
for t in range(num_timesteps):
    # Operation: traverse partial move expansion tree
    # data-structure in Python to collect unexpanded leaves.
    with rls.operation("mcts_tree_search"):
        leaf_minibatches = ...

        for m in range(len(leaf_minibatches)):
            # Nested Operation: Expand a minibatch of Go
            # board-game configurations using
            # neural-network inference.
            with rls.operation("expand_leaf"):
                tf.session.run(expand_leaf, leaf_minibatches[m])
```

```
$ rls-prof python train.py
```

Figure 2. High-level algorithmic annotations: RL-Scope provides developers with a simple API for annotating their code with training phases and operations; these annotations are used to scope CPU/GPU critical path analysis to high-level algorithmic operations.

3.1 High-level Algorithmic Annotations

RL-Scope exposes a simple high-level API to the developer for annotating their code with *training phases* comprised of individual *operations*. We opt for manual annotations over automatic annotations derived from symbols since we have anecdotally observed that RL developers tend to become overwhelmed by results from language-level profilers (e.g., the Python profiler), but are quick to recognize high-level algorithmic operations.

Figure 2 shows part of a simplified Python script for training an agent to play the game Go (Silver et al., 2017). RL-Scope leverages the Python `with` language construct to allow the developer to annotate regions of their code with high-level operation labels. Executing a `with` block results in a start/end timestamps being recorded. Operations can be arbitrarily nested. For example, `mcts_tree_search` uses pure high-level Python code to traverse a tree data-structure of possible board-game states randomly and collects minibatches of unexpanded Go board-game states. Upon forming a minibatch, it executes `expand_leaf`, which passes the minibatch of board-game states through a neural network to decide how to traverse the tree next. This nesting of operations attributes all neural network inferences (in this example, CPU/GPU TensorFlow time) to `expand_leaf`, and all data-structure traversal (Python CPU time) to `mcts_tree_search`.

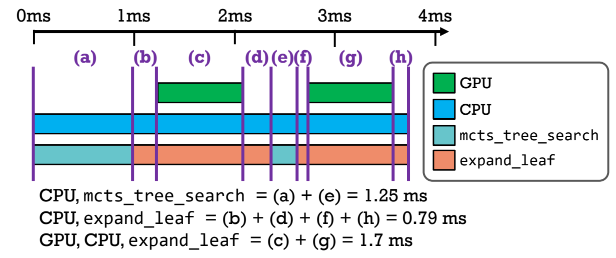

0ms
1ms
2ms
3ms
4ms
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
GPU
CPU
mcts_tree_search
expand_leaf
CPU, mcts_tree_search = (a) + (e) = 1.25 ms
CPU, expand_leaf = (b) + (d) + (f) + (h) = 0.79 ms
GPU, CPU, expand_leaf = (c) + (g) = 1.7 ms

Figure 3. Cross-stack event overlap: RL-Scope accounts for the overlap between cross-stack events, which simultaneously measures overlap between CPU and GPU resources, and scopes training time to high-level algorithmic operations.

3.2 Transparent Event Interception

To satisfy RL-Scope's goals of collecting cross-stack information and avoiding both simulator and ML library recompilation, RL-Scope uses transparent hooks to intercept and record the start/end timestamps of profiling events. RL-Scope uses two techniques to collect events across the stack transparently: (i) NVIDIA CUPTI profiling library and (ii) High-level language $\leftrightarrow$ C interception:

NVIDIA CUPTI Profiling Library We use the NVIDIA CUPTI profiling library (NVIDIA, 2019a) to transparently register hooks at program startup. When the user launches RL-Scope (bottom of Figure 2), we transparently prepend `librlscope.so` to the `LD_PRELOAD` environment variable to force hooks to be registered prior to loading any ML libraries. Using these hooks, we can collect *CUDA API time*, CPU time spent in CUDA API calls such as `cudaLaunchKernel` that asynchronously queue kernels for execution on the GPU, and *GPU kernel time*: time spent executing CUDA kernel code on the GPU.

High-level language $\leftrightarrow$ C interception: To collect time spent in high-level language (e.g., Python) and C libraries (e.g. ML backends and simulators) we intercept calls to (high-level $\rightarrow$ C) and returns from C libraries (C $\rightarrow$ high-level). To perform this interception while avoiding recompilation, we use Python to dynamically generate function wrappers around native library bindings for both simulator and ML libraries. Using timestamps at these interception points, we can collect time spent in Python, ML backends, and simulators.

3.3 Cross-Stack Event Overlap

Unfortunately, visualizations showing raw event traces tend to overwhelm RL users, since they do not provide a clear summary of the percentage of CPU and GPU time on the critical path of a program scoped to high-level algorithmic operations. RL-Scope solves this issue by using the raw event trace to compute CPU/GPU overlap scoped to high-level user operations. Figure 3 illustrates how CPU/GPU usage is computed. Offline, RL-Scope linearly walks left-to-right through the events in the trace sorted by start-time and identifies regions of overlap between CPU events (e.g., Python, simulator, and TensorFlow C), GPU events (i.e., kernel execution times, memory copies), and high-level operations (e.g., `mcts_tree_search` and `expand_leaf`). Resource utilization in these overlapping regions is then summed to obtain the total CPU/GPU critical path latency, scoped to the user's operations. For example, in Figure 3 `expand_leaf` spends $0.79ms$ of its time purely CPU-bound, while $1.7ms$ is spent executing on both the CPU and the GPU. For more fine-grained CPU information, CPU time can be further divided into ML backend and CUDA API time.

3.4 Profiling Calibration and Overhead Correction

Profilers inflate CPU-side time due to additional book-keeping code in the critical path. This overhead can reach up to 90.2% of runtime in our experiments (Appendix C.3). RL-Scope achieves accurate cross-stack critical path measurements by correcting for any CPU time inflation during offline analysis. The full details of how we calibrate and verify the overhead correction are available in Appendix C, and below, we provide a high-level summary.

To correct profiling overhead, RL-Scope calibrates for the average duration of book-keeping code paths and subtracts this time at the precise point when it occurs. Since RL-Scope already knows when book-keeping occurs from the information it collects for critical path analysis (Sections 3.1 and 3.2), the challenge lies in accurately calibrating for the average duration of book-keeping code.

We identify two types of profiler overhead that RL-Scope must calibrate. First, the overhead of most RL-Scope book-keeping code only depends on the type of intercepted event. For example, the extra CPU time incurred by RL-Scope intercepting Python $\leftrightarrow$ C/C++ transitions is the same regardless of which part of the code it occurs in. Similarly, the overhead of our interception of CUDA API calls does not depend on which CUDA API was used, and tracking algorithmic annotations does not depend on which operation was annotated. For these cases, we find RL-Scope's overhead by dividing the increase in total runtime when enabling profiling by the number of times the book-keeping code was called; see Appendix C.1 for details. The second type of overhead is incurred by internal closed-source profiling code inside the CUDA library. This code inflates runtime by different amounts, depending on which CUDA API is called. Since profiling cannot be enabled separately for different APIs, accounting for it requires tracking the number and duration of each individual API call separately; see Appendix C.2 for details.

We validate overhead correction accuracy on a diverse set of RL training workloads with different combinations of environment and RL algorithms. We run each workload without profiling, and again with full RL-Scope with all book-keeping code enabled. Across all algorithms and simulators, RL-Scope's error is within $\pm 16\%$ of the actual uninstrumented training time (see Appendix C.3 for details).

4 RL-Scope Case Studies

We illustrate the usefulness of RL-Scope's accurate cross-stack profiling metrics through in-depth case studies. The hardware configuration we used throughout all experiments is an AMD EPYC 7371 CPU running at 3.1 GHz with 128 GB of RAM and an NVIDIA 2080Ti GPU. For software configuration, we used Ubuntu 18.04, TensorFlow v2.2.0, PyTorch v1.6.0, and Python 3.6.9. The exact RL framework(s), simulator(s), RL algorithm(s), and ML backend(s) used varies by case study, where it is stated explicitly.

4.1 Case Study: Selecting an RL Framework

One of the first major problems faced by RL developers is which RL framework to use, of which there are many, spanning both TensorFlow and PyTorch ML backends (Winder, 2019). In the absence of apples-to-apples comparisons amongst available RL frameworks, users fall back on rule-of-thumb approaches (Winder, 2019; Simonini, 2019) to selecting an RL framework: PyTorch for ease-of-use, frameworks with a particular algorithm, frameworks built on a specific ML backend, and so on. This suggests developers often do not understand the exact training time trade-off between using PyTorch- and TensorFlow-based RL frameworks, and how this trade-off is influenced by the underlying *execution model* of the ML backend.

TensorFlow 1.0 initially only supported a Graph execution model, where symbolic computations are declared during program initialization, then run all at once using initial graph inputs. PyTorch, and more recently TensorFlow 2.0, supports an Eager execution model where operators are run as they are defined, making it easy to use with language level debuggers. With TensorFlow 2.0, developers can use the Autograph execution model to automatically convert their Eager program to an optimized TensorFlow graph, so long as developers adhere to coding practices (TensorFlow, 2020). While developers generally prefer to only use the Eager execution API for productivity, RL-Scope reveals that the execution model used for a given ML backend can subtly impact training performance.

To ensure an apples-to-apples comparison across RL frameworks, we compare the same underlying RL algorithms (TD3/DDPG), the same simulator (Walker2D), and identical hyperparameters (e.g., batch size, network architecture, learning rate, etc.) which were pre-tuned in the stable-baselines RL framework (Hill et al., 2018; Raffin, 2018). This allows us to attribute differences in training time breakdowns to an RL framework's ML backend and execution model. We consider four representative RL frameworks that span popular $\langle$*execution model, ML backend*$\rangle$ combinations[1], shown in Table 1.

Table 1. RL frameworks: we consider popular $\langle$*execution model, ML backend*$\rangle$ combinations.

RL framework	**Execution model**	**ML backend**
stable-baselines	Graph	TensorFlow 2.2.0
tf-agents	Autograph	TensorFlow 2.2.0
tf-agents	Eager	TensorFlow 2.2.0
ReAgent	Eager	PyTorch 1.6.0

Figure 4a shows the training time breakdown across the RL frameworks for TD3. The top black bars show the total training time for each workload. The bottom bars show the same training time split between the different operations of the RL training loop (i.e., backpropagation, inference, simulation). Colors indicate whether the time was spent in GPU, CPU, or overlapping both CPU + GPU, while patterns show the level of the software stack (*Simulator*, *Python*, ML *Backend*, or *CUDA* API calls). Figure 4c below shows the number of native library transitions from: Python$\rightarrow$ML *Backend*, Python$\rightarrow$*Simulator*, and ML Backend$\rightarrow$*CUDA* API calls. As we will see, these transitions are a major contributor to differences in execution time breakdowns. Similarly, Figures 4b and 4d shows the training time breakdown and transitions for DDPG. We will frequently refer to these figures to support our findings.

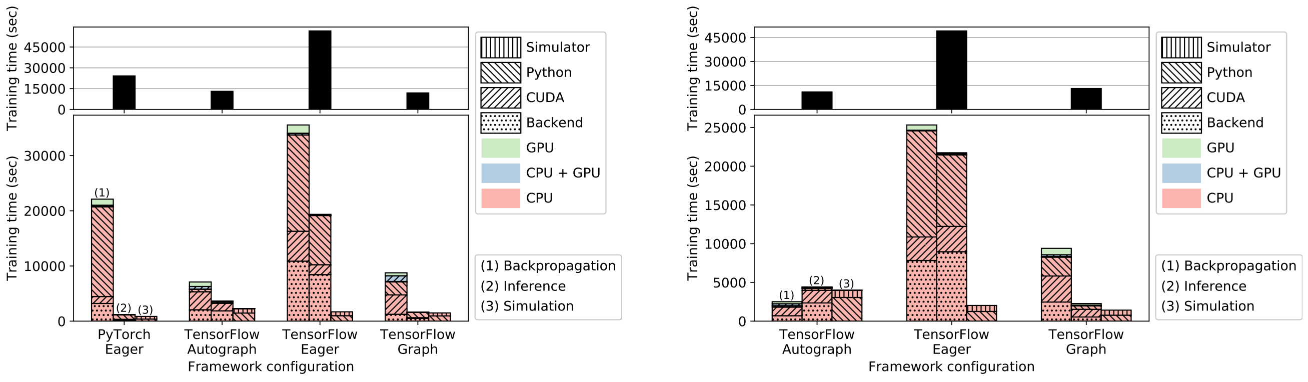
4.1.1 Execution model

RL-Scope's metrics are able to quantify our intuition of how execution models affect training time by correlating it with reduced Python$\rightarrow$Backend transitions. This correlation further explains a $2.3\times$ slowdown between the same algorithm implemented in two different ML backends.

$\boxed{F.1}$ **Eager execution is $1.9\times$ to $4.8\times$ slower than both Autograph and Graph execution, whereas neither Autograph nor Graph always outperform the other and are within 19.7% of one another.**

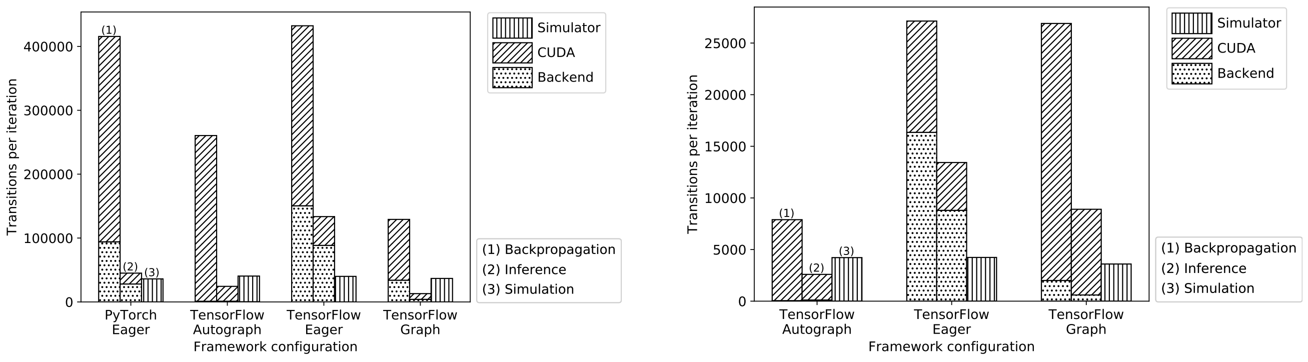
The total training time of Figure 4a shows that Eager execution implementations generally perform worse than Graph and Autograph. TensorFlow's older Graph API is still a more efficient API than the newer Eager API. Hence, developers should be careful to ensure their code-base can run with Autograph by adhering to coding practices (TensorFlow, 2020) and not rely on Eager performance alone. Moreover, even Autograph does not always outperform the

[1] ReAgent does not yet make use of PyTorch's Autograph analog TorchScript for training, only for model serving.



(a) (TD3, Walker2D) - time breakdown

(b) (DDPG, Walker2D) - time breakdown



(c) (TD3, Walker2D) - language transitions

(d) (DDPG, Walker2D) - language transitions

Figure 4. RL framework comparison: We used identical RL algorithm (left: TD3, right: DDPG), simulator (Walker2D), and tuned hyperparameters; differences in execution between RL frameworks are strictly due to RL algorithm implementation and ML backend. Differences in *time breakdown* (top) across RL frameworks can be explained by higher number of *language transitions* (bottom) between the Python and the ML backend (*Backend*), and between the ML backend and the accelerator API calls (*CUDA*).

older Graph API. In particular, for TD3 (Figure 4a) Graph is 9.7% faster than Autograph; conversely, for DDPG (Figure 4b) Autograph is 19.7% faster than Graph.

$\boxed{F.2}$ By reducing Python→Backend transitions, Autograph substantially reduces Python time from 59.2% in Graph to at most 7.2% for inference/backpropagation.

Comparing TensorFlow Autograph to TensorFlow Graph for TD3 in Figure 4a demonstrates substantial reductions in Python time both for backpropagation ($5.1\times$) and for inference ($4.4\times$); similar results are observed for backpropagation ($13.5\times$) and inference ($1.8\times$) for DDPG in Figure 4b. These reductions in Python time are consistent with Autograph's design, which automatically converts high-level Python code (e.g., conditionals, while loops) into in-graph TensorFlow operators. Figure 4c/4d explains this reduction in Python time for TD3/DDPG since the number of Backend transitions are close to zero in Autograph compared to other execution models.

$\boxed{F.3}$ For TD3, PyTorch Eager is $2.3\times$ faster than TensorFlow Eager since it minimizes Python→Backend transitions more effectively. The Graph/Autograph implementation is $2.0\times$ faster than PyTorch Eager.

For TD3 (Figure 4a), the precise difference in the Eager execution model's performance is markedly different across PyTorch and TensorFlow ML backends. PyTorch Eager performs up to $2.0\times$ slower than Graph/Autograph, whereas TensorFlow Eager is up to $4.8\times$ slower than Graph/Autograph. This difference in Eager performance can be explained by Figure 4c, where we observe a larger number of transitions to the Backend from Python in TensorFlow Eager than in PyTorch Eager during backpropagation ($1.6\times$) and inference ($3.2\times$).

4.1.2 Bottleneck analysis

RL-Scope's fine-grained time breakdown is useful for explaining subtle performance bottlenecks that are a result of an overly abstracted RL algorithm implementation, sensitivity to particular hyperparameter settings, and even the result of anomalous behaviour in the underlying implementation of an ML backend.

$\boxed{F.4}$ **RL-Scope's metrics identify inefficient abstractions in high-level code as responsible for a $3.7\times$ inflation in backpropagation in Graph compared to Autograph for DDPG.**

Backpropagation in Autograph is $3.7\times$ faster than Graph for DDPG (Figure 4b), whereas backpropagation in Autograph is only $1.2\times$ faster than Graph for TD3 (Figure 4a); these inefficiencies in backpropagation for DDPG Graph are correlated with high CUDA API time inflation ($2.6\times$) and high Python inflation ($13.5\times$) relative to DDPG Autograph. The inflation in CUDA API suggests a greater number of kernel executions, and greater Python time suggests a larger number of calls to the ML backend. Inspecting the DDPG and TD3 implementations reveals that these inefficiencies in DDPG backpropagation are caused by two factors. First, DDPG uses inefficient abstractions leading to more GPU kernel launches; DDPG uses a MPI-friendly but GPU-unfriendly Python implementation of the Adam optimizer that copies GPU weights to the CPU then writes back results to the GPU even during single-node training. Second, DDPG performs frequent backend transitions leading to more Python time; several operations (e.g., copying network weights to a "target" network, applying gradients to actor and critic networks) execute in separate Backend calls that could be bundled into a single call.

$\boxed{F.5}$ **Autograph can inflate Python time by as much as $2.4\times$ during simulation; training time (not just model accuracy) is highly sensitive to small differences in hyperparameter choices.**

RL-Scope is able to identify a performance anomaly in the DDPG Autograph implementation in Figure 4b; interestingly, the Python time of simulation in Autograph is inflated $2.4\times$ when compared to executing in Eager mode. Even more surprisingly, both DDPG and TD3 share the same data collection code, yet TD3 does not suffer from this inflation (Figure 4a). Hence, the only difference that would explain the anomaly in DDPG is a difference in hyperparameter settings from TD3. Inspecting the algorithm implementations reveals that TD3 performs 1000 consecutive simulator steps before performing a gradient update, whereas DDPG performs only 100 consecutive simulator steps. Hence, TD3 is better able to amortize overheads associated will calling into Autograph's in-graph data collection loop. To confirm our hypothesis, we adjusted DDPG's hyperparameter to 1000 to match TD3 and the inflation dropped down to $1.1\times$, similar to what we see for TD3 ($1.5\times$).

$\boxed{F.6}$ **Inference time in Autograph is inflated due to a $4.1\times$ inflation in Backend time compared to Graph. This inflation is *not* due to extra Python→Backend transitions, and is instead a performance anomaly within the ML backend itself.**

RL-Scope's scoping capabilities can identify performance anomalies in RL framework implementations within specific high-level algorithmic operations, at precisely the level of the stack they originate from. In TD3 (Figure 4a), there is a $3.8\times$ increase in inference Backend time in Autograph compared to Graph; similarly in DDPG (Figure 4b), there is a $4.4\times$ increase. However, this inflation is not due to increased Python→Backend transitions; in fact, Autograph clearly minimizes Backend transitions (Figures 4c and 4d). Hence, this consistent anomalous behaviour is indicative of a performance issue within the Autograph implementation of inference, which is not present in the Graph API implementation.

4.1.3 GPU usage

RL-Scope confirms that GPU usage is poor across *all* RL frameworks, showing this issue is not isolated to any one ML backend. In all ML backends, poor GPU usage is rooted in greater CPU time spent in CUDA API calls than spent in GPU kernels.

$\boxed{F.7}$ **Total GPU time is similar across all RL frameworks regardless of ML backend, and consistently low across all RL frameworks, making up at most 14.1% of total training time.**

In all RL frameworks the total GPU execution time makes up only a very small portion of the total training time of both TD3 (Figure 4a) and DDPG (Figure 4b), which is immediately apparent by the small total GPU time (i.e., $\boxed{\text{GPU}}$ + $\boxed{\text{CPU + GPU}}$). This suggests that poor GPU usage is not an implementation quirk in any one ML backend, it is instead a widespread problem affecting *all* ML backends.

$\boxed{F.8}$ **In all RL frameworks, CPU-side CUDA API time dominates total GPU kernel execution time, taking up on average $3.6\times$ as much time as GPU kernel execution.**

The surprising insight from RL-Scope that CUDA API time dominates GPU kernel execution time emphasizes the drastic differences between supervised learning workloads and RL workloads established in Section 2.2: (1) *Training data collection:* data must be collected at run-time by repeatedly alternating between selecting an action (inference) and taking an action (simulation) – the training loop includes more than just backpropagation, (2) *Smaller neural networks:* RL algorithms make use of smaller neural networks than state-of-art models from supervised learning tasks, leading to shorter GPU kernel execution times.

4.2 Case Study: RL Algorithm and Simulator Survey

Practitioners must evaluate a large number of RL algorithms for each application. For example, OpenAI's repository has implemented 9 RL algorithms (Dhariwal et al., 2017). We investigate how algorithm choice affects the workload

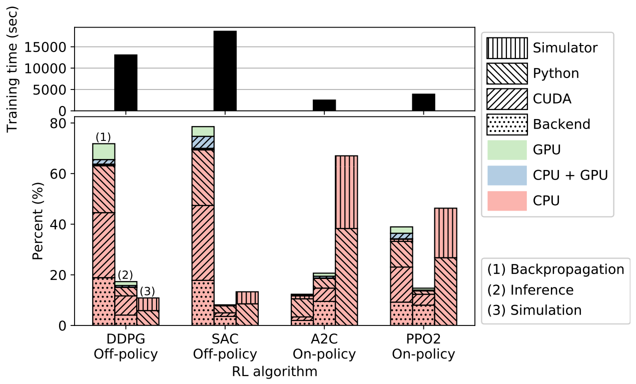


Figure 5. Algorithm choice: We used a popular RL environment (Walker2D: robotics task of a walking humanoid) measured how the stages (backpropagation, inference, simulation) of each measured algorithm change with respect to algorithm choice. All tested RL workloads spend about 90% of their runtime purely in the CPU.

profile (Figure 5). We chose the popular Walker2D "walking bipedal humanoid" simulated task (Erez et al., 2012) where the agent must learn to walk forward.

$\boxed{F.9}$ Most of the training time across RL algorithms is spent CPU-bound executing the software stack – CUDA API calls, backend code, high-level code – suggesting the entire hardware/software stack is poorly optimized for the RL use case. Even inference and backpropagation, which are GPU-heavy in SL workloads, spend at most 12.9% executing GPU kernels.

Of the surveyed RL algorithms, DDPG has the largest portion of time spent GPU-bound, with 10.6% of total training time spent executing GPU kernels (i.e., GPU + CPU + GPU); conversely, A2C spends the least time on the GPU with only 2.6% total training time spent executing GPU kernels. If we consider backpropagation only, at most 12.5% percent of a backpropagation operation is spent executing GPU kernels (for PPO2). Similarly, at most 12.9% of an inference operation is spent executing GPU kernels (for DDPG). For the most GPU-heavy operation (backpropagation) of the most GPU-heavy algorithm (DDPG), 6.0% of CPU time is spent in simulation and 31.1% in Python while, with a non-negligible amount of CPU time spent in CUDA API calls (37.2%) and the Backend (25.7%); a similar spread of CPU time across the RL software stack is seen in all RL algorithms.

$\boxed{F.10}$ On-policy RL algorithms are at least $3.5\times$ more simulation-bound than off-policy RL algorithms.

A2C and PPO2 spend a majority of their execution in simulation, with A2C spending 67.0% and PPO2 spending 46.3%. Conversely, DDPG and SAC spend only 10.9% and 13.3% of their execution in simulation respectively, which are instead dominated by backpropagation. *Off-policy algorithms* (e.g., DDPG, SAC) are said to be more "sample-efficient" since they are learning $Q(s, a)$ or $V(s)$ by sampling rewards obtained from the simulator; as a result, they can re-use data collected from the simulator when forming a minibatch to learn from, allowing them to spend a smaller percentage of their training time collecting data from the simulator. *On-policy algorithms* (e.g., A2C, PPO2) must learn and update a policy network π directly. As a result, the minibatches for updating the policy network must consist of experience collected using the same policy. This means entire episodes of experience must be collected from the simulator before a minibatch and subsequent gradient update can be performed; hence, the on-policy algorithm is sample-*in*efficient. Interestingly, this disparity has been qualitatively explained as the "sample-efficiency" problem in the RL research community (Wang et al., 2017), but it has never been systematically quantified.

Due to space constraints, our survey of how simulators affect total RL training time can be found in Appendix B.1.

4.3 Case Study: Scale-up RL Workload

As shown above in our RL algorithm and simulator survey, an RL workload running on one worker is unable to fully saturate a modern GPU. One way practitioners try to increase GPU usage is by taking advantage of CPU and GPU hardware parallelism. RL training workloads feature a data collection phase, where the worker repeatedly switches between inference to decide on an action to take and running the simulator on the chosen action. Since the model is not being updated during data collection, one common way to increase GPU usage is to have multiple workers collecting data at the same time with the hope that inference minibatches will be concurrently processed by the GPU to keep it occupied.

To decide whether a given number of parallel workers fully occupies the GPU, developers resort to overly coarse-grained tools (e.g., `nvidia-smi`) that report a "GPU utilization" metric, and increase the number of parallel workers until it reaches 100%.

RL-Scope reveals that coarse-grained tools misleadingly report 100% utilization even though the total GPU usage is, in fact, extremely low. To illustrate this problem in practice, we use RL-Scope to profile the Minigo workload, a popular re-implementation of AlphaGoZero (Silver et al., 2017) adopted by the MLPerf training benchmark suite (Mattson et al., 2020). Minigo parallelizes data collection by having 16 "self-play" worker processes play games of Go in parallel, thereby overlapping inference minibatches when evaluating board game states. For brevity, we summarize the finding and supporting metrics provided by RL-Scope; figures containing these metrics and additional details of the Minigo training loop can be found in Appendix B.2.

$\boxed{F.11}$ The GPU utilization reported by `nvidia-smi` for short inference tasks, common in RL workloads, gives a drastically inflated measure of GPU use. In reality, the workload is hardly using the GPU. Scaling-up by running more workers can exacerbate this issue.

RL-Scope's time breakdown metrics reveals that self-play workers take at most 5080 seconds in total, and only a meager 20 seconds of each worker is spent executing GPU kernels. Conversely, `nvidia-smi` misleadingly reports 100% GPU utilization during parallel data collection. RL-Scope's metrics reveal the danger of using coarse-grained metrics like GPU utilization for measuring GPU usage. The official documentation for `nvidia-smi` (NVIDIA, 2020d) states that GPU utilization is measured by looking to see if one or more kernels are executing over the sample period, and that the sample period is between $1/6$ seconds and 1 second. Hence, if an extremely small kernel runs that does not fully occupy the sample period, that entire sample period will still have 100% GPU utilization. Given that RL inference operations are short kernels, but numerous since 16 processes generate them in parallel, we suspect that all sample periods include at least one such short GPU kernel.

5 Related work

We begin with a high-level comparison of RL-Scope to groups of related works; detailed comparisons of RL-Scope to each work are given below. Existing *ML profiling tools* target GPU-bound supervised learning (SL) workloads and as a result limit their analysis to bottleneck layers and kernels. In contrast, with RL-Scope we observe the majority of the time is spent CPU-bound throughout the software stack, even for operations considered GPU-heavy (e.g., backpropagation). *GPU profiling tools* collect GPU kernel execution time with some user-level control over scoping, but unlike RL-Scope they are not capable of correcting for profiling overhead. Finally, *distributed RL training and frameworks* attempt to increase GPU usage by parallelizing inference during data collection, but we observe this to be ineffective with Minigo (Section 4.3).

ML Profiling Tools: In TBD (Zhu et al., 2018), the authors survey ML training workloads (including one RL workload) by collecting high-level CPU/GPU utilization metrics; RL-Scope illustrates that GPU utilization can be a misleading indicator of GPU usage and empowers developers to explain why GPU utilization is a poor metric by providing an accurate CPU/GPU time breakdown. XSP (Li et al., 2020) is an "across-stack" profiler that combines profiling information from the model/layer/GPU-kernel levels to provide a drill-down interface for exploring neural network accelerator bottlenecks (e.g., a specific kernel in a specific layer) that dominate SL workloads. The authors avoid some sources of profiling overhead by collecting profiling information from different levels of the stack at different training iterations, and correlating them into a single view. The two key differences with our approach are that (1) XSP limits its timing analysis to bottleneck layers and kernels in GPU-bound SL workloads whereas RL-Scope targets RL workloads which are more CPU-bound due to time spent in high-level code, CUDA APIs, simulators, and from excessive *transitions* to ML backends, and (2) RL-Scope's calibration approach allows it to correct for additional sources of book-keeping overhead that XSP cannot correct such as inflation in closed-source CUDA API calls; we observe that RL workloads are particularly sensitive to profiling overhead and can experience training time inflation of up to $1.9\times$ (Figure 11).

GPU Profiling Tools: NVIDIA Nsight systems (NVIDIA, 2020b) visualizes a timeline of GPU kernels, CUDA API calls, and system calls, and allows users to annotate their code with high-level operations using NVTX (NVIDIA, 2020c). DLProf (NVIDIA, 2020a) focuses on deep neural networks by extending Nsight systems to provide layer-wise scoping and recognizing training iterations to enable summary statistics. RL-Scope avoids using both Nsight systems and DLProf since their analysis backend is closed-source, and as a result is unable to provide calibration correction of CPU profiling overhead which is needed for accurately attributing time spent throughout the RL software stack.

Distributed RL training and frameworks: Distributed RL frameworks (Moritz et al., 2018; Nair et al., 2015) have been proposed to scale exploring large hyperparameter search spaces to entire clusters of machines. These frameworks have been used to reduce training time of 49 Atari games from 14 days on a single-machine to 6 days on 100 machines by exploiting the embarrassingly parallel nature of some RL training jobs (Nair et al., 2015). Unfortunately, little attention has been paid to the performance of individual machines within a cluster. Today's distributed RL frameworks are built off of pre-existing ML backends (e.g., TensorFlow, PyTorch), and as a result, inherit all of the poor GPU usage characteristics more commonly observed in single-machine frameworks. In particular, scaling GPU usage up by parallelizing inference requests is still insufficient to fully occupy GPU resource usage (Section 4.3). In future work, we hope to extend RL-Scope to support distributed RL frameworks, though we expect many of the findings to remain consistent with single-machine frameworks.

6 Conclusions

While deep Reinforcement Learning (RL) models are notoriously slow to train, the exact reasons for this are not well-understood. We observe that RL workloads are fundamentally different from supervised learning workloads, meaning current GPU and ML profilers are a poor fit for this setting.

RL-Scope is the first profiler specifically designed for RL workloads: it explains where time is spent across the software stack by scoping CPU/GPU usage to high-level algorithmic annotations, corrects for profiling overhead, captures language transitions, and supports multiple ML backends and simulators. Using RL-Scope, we studied the effect of RL framework and ML backend on training time, identified bottlenecks caused by inefficient abstractions in RL frameworks, quantified how bottlenecks change as we vary the RL algorithm and simulator, and showed how coarse-grained GPU utilization metrics can be misleading in a popular scale-up RL workload. RL-Scope is an open-source tool available at `https://github.com/UofT-EcoSystem/rlscope`, and we look forward to enabling future researchers and practitioners to benefit from RL-Scope's insights in their own RL code-bases.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI gym. *arXiv preprint arXiv:1606.01540*, 2016.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. OpenAI baselines. `https://github.com/openai/baselines`, 2017.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. CARLA: An open urban driving simulator. *Conference on Robot Learning*, 2017.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pp. 1329–1338, 2016.

Epic Games. Unreal engine 4. `https://www.unrealengine.com/`, 2019.

Erez, T., Tassa, Y., and Todorov, E. Infinite-horizon model predictive control for periodic tasks with contacts. *Robotics: Science and systems VII*, pp. 73, 2012.

Fujimoto, S., Conti, E., Ghavamzadeh, M., and Pineau, J. Benchmarking batch deep reinforcement learning algorithms. *arXiv preprint arXiv:1910.01708*, 2019.

Google. Protocol Buffers — Google Developers. `https://developers.google.com/protocol-buffers`, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

Intel. Intel VTune Amplifier. `https://software.intel.com/en-us/vtune`, 2019.

Krishnan, S., Borojerdian, B., Fu, W., Faust, A., and Reddi, V. J. Air learning: An AI research platform for algorithm-hardware benchmarking of autonomous aerial robots. *arXiv preprint arXiv:1906.00421*, 2019.

Li, C., Dakkak, A., Xiong, J., Wei, W., Xu, L., and Hwu, W.-m. XSP: Across-stack profiling and analysis of machine learning models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 326–327. IEEE, 2020.

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., Brooks, D., Chen, D., Dutta, D., Gupta, U., Hazelwood, K., Hock, A., Huang, X., Kang, D., Kanter, D., Kumar, N., Liao, J., Narayanan, D., Oguntebi, T., Pekhimenko, G., Pentecost, L., Janapa Reddi, V., Robie, T., St John, T., Wu, C.-J., Xu, L., Young, C., and Zaharia, M. MLPerf training benchmark. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 336–349, 2020. URL `https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf`.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. *International Conference on Machine Learning*, 2017.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529, 2015.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 561–577, 2018.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Mnih, V., Kavukcuoglu, K., and Silver, D. Massively parallel methods for deep reinforcement learning. *Deep Learning Workshop, International Conference on Machine Learning*, 2015.

NVIDIA. CUPTI :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cupti/index.html, 2019a.

NVIDIA. NVIDIA Developer Forums - PC Sampling leads to large slow-downs in execution time? https://devtalk.nvidia.com/default/topic/1061180/cuda-profiler-tools-interface-cupti-/pc-sampling-leads-to-large-slow-downs-in-execution-time-, 2019b.

NVIDIA. CUPTI :: CUPTI Documentation - PC Sampling. https://docs.nvidia.com/cupti/Cupti/r_main.html#r_pc_sampling, 2019c.

NVIDIA. DLProf User Guide. https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide, 2020a.

NVIDIA. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems, 2020b.

NVIDIA. NVIDIA Tools Extension Library (NVTX). https://docs.nvidia.com/nsight-visual-studio-edition/2020.1/nvtx/index.html, 2020c.

NVIDIA. NVIDIA - nvidia-smi Documentation. http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, 2020d.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Python. multiprocessing — Process-based parallelism - Python. https://docs.python.org/3/library/multiprocessing.html, 2020.

Raffin, A. RL baselines zoo. https://github.com/araffin/rl-baselines-zoo, 2018.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

Simonini, T. On Choosing a Deep Reinforcement Learning Library - Medium. https://medium.com/data-from-the-trenches/choosing-a-deep-reinforcement-learning-library-890fb0307092, 2019.

TensorFlow. Better performance with tf.function. https://www.tensorflow.org/guide/function#python_side_effects, 2020.

Tromp, J. and Farnebäck, G. Combinatorics of Go. In *International Conference on Computers and Games*, pp. 84–99. Springer, 2006.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. Sample efficient actor-critic with experience replay. *International Conference on Learning Representations*, 2017.

Winder, P. A Comparison of Reinforcement Learning Frameworks: Dopamine, RLLib, Keras-RL, Coach, TRFL, Tensorforce, Coach and more. https://winderresearch.com/a-comparison-of-reinforcement-learning-frameworks-dopamine-rllib-keras-rl-coach-trfl-tensorforce-coach-and-more/, 2019.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Jayarajan, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 88–100. IEEE, 2018.

Summary of Appendices

In Section A, we provide information essential for developers looking to implement profiling tools analogous to RL-Scope. In Section B, we provide a simulator survey to show how RL training bottlenecks vary with simulator choice, and we provide supporting figures and additional training loop details for the Minigo RL scale-up workload. In Section C, we provide additional details on how RL-Scope corrects for profiling overhead from transparent interception hooks and closed-source CUDA profiling libraries, and show how RL-Scope is able to correct within $\pm 16\%$ of profiling overhead. In Section D, we link to detailed online artifact evaluation instructions for reproducing some of the figures found in this paper.

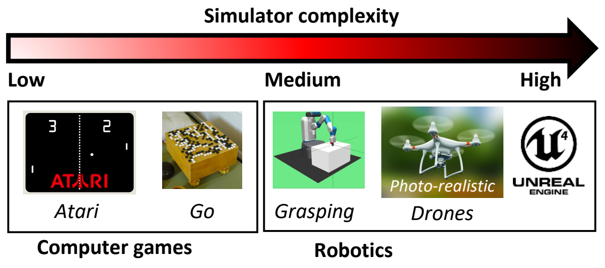


Figure 6. A representative sample of RL simulators organized by computational complexity. Computer games such as arcade games have simple graphics and hence have lower complexity. Emerging robotics applications often have photo-realistic simulators and have higher complexity.

A RL-Scope implementation details

We provide additional information for developers looking to implement profiling tools analogous to RL-Scope.

A.1 Storing Traces Asynchronously

To avoid adding profiling overhead as much as possible, RL-Scope stores trace files to disk asynchronously, off the critical path of training. Traces are aggregated in a stand-alone C++ library (`librlscope.so`), are saved to disk once they reach 20MB, and are stored using the Protobuf binary serialization library (Google, 2020). We explicitly avoid collecting and dumping trace files from Python since asynchronous dumping in python requires using the `multiprocessing` module (Python, 2020), which is implemented using a process fork. This would lead to undefined behaviour in multi-threaded libraries (e.g., TensorFlow), and would also require serialization of 20MB of data on the critical path. `librlscope.so` allows us to access shared memory multi-threading in C++ without overheads.

A.2 Avoiding Sampling Profilers

During development, we considered the possibility of collecting information using a sampling profiler, which aims to minimize profiling overhead by periodically sampling events at regular intervals. Examples of sampling profilers for CPU-only workloads include VTune (Intel, 2019) and the Linux `perf` utility. Analogously, NVIDIA's CUPTI library has a PC Sampling API (NVIDIA, 2019c) for periodically sampling the GPU-side program-counter of executing GPU kernels. However after experimenting with the CUPTI PC Sampling API, we chose to avoid sampling profilers due to several shortcomings. First, GPU PC sampling adds significant overhead: we observed large overheads when using the CUPTI PC Sampling API, which exceeded those of collecting GPU kernel start/end timestamps[2]. Additionally, given that overhead is inevitable, sampling profilers offer no convenient way to correct for their overhead during offline analysis. Lastly, sampling provides incomplete GPU event information, since some GPU kernels execute very quickly ($\leq 1ms$). A sampling profiler would need to account for loss of accuracy due to missed GPU events.

B Extended RL-Scope Case Studies

We provide an additional case study relevant to RL practitioners, showing how RL training time breakdowns vary significantly with different choices of simulator. We also provide more details on our case study of scale-up RL workloads (Section 4.3), with supporting figures and additional details on the Minigo training loop.

B.1 Case Study: Simulator Survey

To observe how RL training time varies with the choice of simulator, we focus on three application domains, organized by simulator complexity (Figure 6), described in detail below:

ML algorithm research: These applications use simple, quick-to-execute simulators such as games (i.e. Atari Pong, Go board game), where the notion of reward is well-defined (e.g., 1 or 0 if a game is won or lost). RL researchers often use such applications when exploring enhancements to core RL algorithms, since it is easier to understand the behaviour of the learned policy since rules of the game and common human strategies are known. Atari games are provided by the OpenAI gym repository (Brockman et al., 2016).

Robotics: These applications use physics simulators to emulate a robot interacting with its environment, in tasks

[2]We confirmed with NVIDIA employees on their developer forums that overheads above $2\times$ are to be expected with the CUPTI PC Sampling API (NVIDIA, 2019b).

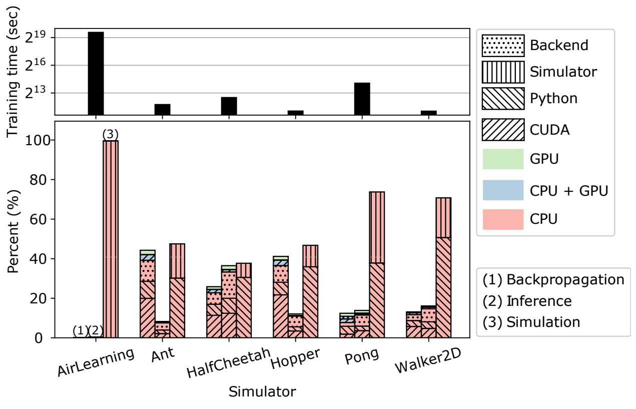


Figure 7. Simulator choice: We used a top-performing RL algorithm (PPO) and measured how each stage of the algorithm changes with respect to environment choice. GPU accounts for 5% or less of the runtime across all simulators.

such as locomotion and object grasping. The physics simulation is done on the CPU, whereas training is done on the GPU. These medium complexity robotics tasks are provided by the OpenAI gym repository (Brockman et al., 2016).

Drone applications: These applications use realistic physics simulators with photo-realistic rendering implemented using a popular video game engine (Epic Games, 2019). Sufficiently realistic simulators allow a researcher to determine how an RL algorithm will behave in a real-world deployment without the high cost overhead associated with actually deploying it (Dosovitskiy et al., 2017). These simulators are computationally intensive, and make use the GPU to perform graphics rendering. We use the point-to-point navigation task from the AirLearning toolkit for unmanned aerial vehicle (UAV) robotics tasks (Krishnan et al., 2019).

We chose a top-performing (i.e., highest final average episodic reward) algorithm from the OpenAI stable-baselines repository (Hill et al., 2018; Raffin, 2018) and ran it across a representative sampling of simulators ranging from low, medium, to high complexity; Figure 7 provides a breakdown of training time for each different simulator.

$\boxed{F.12}$ Simulation time varies, but is always a *large* training bottleneck, taking up at least 38.1% of training time.

Excluding AirLearning, the CPU/GPU breakdown is similar across all low/medium complexity simulators, with as little as 2.8% GPU kernel time for Walker2D and as high as 6.9% for Pong. For high-complexity simulators like AirLearning that simulate a drone in a realistic video game engine, simulation dominates total training time at 99.6%. In the presence of such large simulation times, GPU time spent in inference or backpropagation makes practically no contribution to total training time. Excluding high-complexity simulators (AirLearning), even in low and medium complexity environments, simulation is always the majority contributor to total training time compared to inference and backpropagation, accounting for anywhere between 38.1% in HalfCheetah, to 74.2% in the Pong simulator.

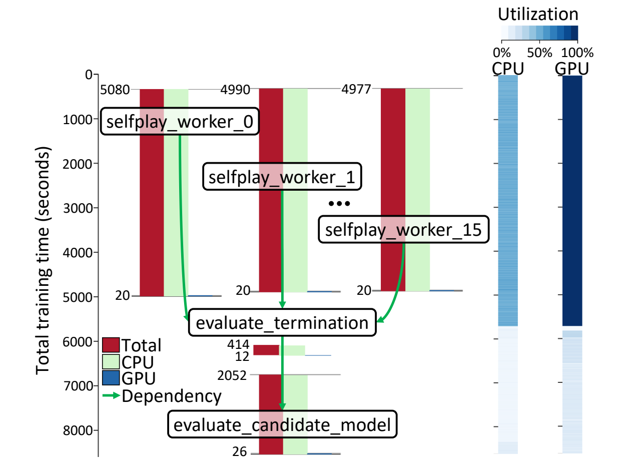


Figure 8. Minigo – multi-process view: `nvidia-smi` misleadingly reports 100% GPU utilization (rightmost bar), even though RL-Scope's GPU execution time metric reveals low GPU usage in each self-play worker (20 seconds).

Even though Pong is a low-complexity simulator, the tuned hyperparameter configuration we used for (PPO, Pong) performs a small number of gradient updates compared to the number of simulator invocations, which explains why it has a large simulation time; this is also the case for the Walker2D configuration. Further, even though backpropagation has a different relative contribution to training time depending on the choice of simulator, it always has a consistent breakdown that is mostly dominated by CUDA API call time, and a small fraction of GPU kernel execution time; the same can be said of inference.

B.2 Case Study: Scale-up RL Workload

First, we provide RL-Scope's multi-process analysis of Minigo which reveals that `nvidia-smi` reports 100% GPU utilization even though RL-Scope reveals that overall GPU usage is still poor. Finally, for interested readers we provide additional details of the full Minigo training loop.

B.2.1 *Minigo RL-Scope Analysis*

In Minigo, each parallel self-play worker plays a game of Go against itself (i.e., each player uses identical model weights), with each worker running an inference minibatch of potential board game states to estimate the probability of winning the match $Q(s, a)$ given the current board game state s and move selection a.

Figure 8 shows RL-Scope's multi-process view of training time for one round of the Minigo training loop: data collection through self-play across 16 parallel worker processes (`selfplay_worker_*`), followed by a model update and evaluation. Each parallel self-play worker process appears as a separate "node" (barplot) in the "computational graph", with dependencies generated from process fork/join relationships. RL-Scope's execution time metrics reveal that `selfplay_worker_0` contributes the most to training time, taking 5080 seconds in total; however, only a meager 20 seconds are spent executing GPU kernels, with the other workers having nearly identical breakdowns. Conversely, `nvidia-smi` misleadingly reports 100% GPU utilization during parallel data collection.

B.2.2 Minigo Training Loop

A naive computer program for playing Go decides the next move to make by exhaustively expanding all potential sequences of moves until game completion, and following the move that leads to a subtree that maximizes the occurrence of winning outcomes. Unfortunately, such an exhaustive expansion is intractable in the game of Go since there are an exponential number of board-game states to explore (2×10^{170} to be precise (Tromp & Farnebäck, 2006)). Monte-carlo-tree-search (MCTS) provides a strategy for approximating this approach of move-tree expansion; the move-tree is truncated using an approximation $Q(s, a)$ of the probability of winning starting from board position s and placing a game-piece at position a. In order for this MCTS strategy to be effective, we must learn a value function $Q(s, a)$. The RL approach to this problem is to learn $Q(s, a)$ by collecting moves from games of self-play, and labelling moves with the eventual outcome of each game.

The entire training process for Minigo is split into three distinct *training phases*. At the end of the three phases, a new *generation* i of model $Q(s, a)$ is obtained for evaluating Go board game moves. The phases are repeated a fixed number of times (default 1000) to obtain successive generations of models, where each generation of model performs as well as or better than the last. The training phases are as follows:

1. **Self-play - collect data:** The current generation of model plays against itself for 2000 games. Self-play games can be played in parallel by a pool of worker processes, where the size of the pool is chosen in advance to maximize CPU and GPU hardware utilization (we use 16 workers, which is the the number of processors on our AMD CPU).
2. **SGD-updates - propose candidate model:** data collected during self-play is used to update the $Q(s, a)$ network.
3. **Evaluation - choose best model for generation** $(i + 1)$**:** the current model and the candidate model play 100 games against each other, and the winner becomes the new model for the next generation[3].

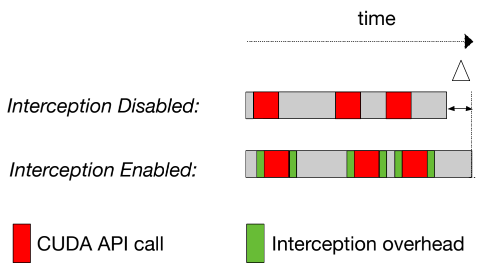


Figure 9. Delta calibration of profiling overhead: The difference in runtime, Δ, can be attributed to interception of CUDA API calls. In this example there are 3 CUDA API calls, so the average overhead for calibration is $\Delta/3$.

C Profiler Overhead Correction

As discussed in Section 3.4, profilers inflate CPU-side time due to additional book-keeping code. To achieve accurate cross-stack critical path measurements, RL-Scope corrects for any CPU time inflation during offline analysis by calibrating for the average duration of book-keeping code paths, and subtracting this time at the precise point when it occurs. Since RL-Scope already knows when book-keeping occurs from the information it collects for critical path analysis (Sections 3.1 and 3.2), the challenge lies in *accurately calibrating* for the average duration of book-keeping code.

C.1 Delta Calibration

To measure average book-keeping overhead, we run the training script twice: with book-keeping enabled and disabled. Assuming that additional runtime (Δ) is strictly due to the enabled book-keeping code[4], the average book-keeping time is Δ divided by the total number of times this book-keeping code executed. Figure 9 illustrates how RL-Scope uses this technique to measure the average book-keeping duration of intercepting CUDA API calls. Besides CUDA API interception overhead, RL-Scope uses delta calibration to correct for high-level algorithm annotations and Python $\leftrightarrow$ C transitions.

C.2 Difference of Average Calibration

As discussed in Section 3.4, not all sources of overhead can be handled by delta calibration. Besides CUDA API interception overhead, the CUDA library has internal closed-source profiling code paths that inflate both

[3]We use the Minigo implementation from MLPerf (Mattson et al., 2020). The Minigo implementation does not optimize evaluation by parallelizing this portion of training, but we believe there is nothing preventing this optimization in the future.

[4]ML code is designed to be deterministic given the same random seed.

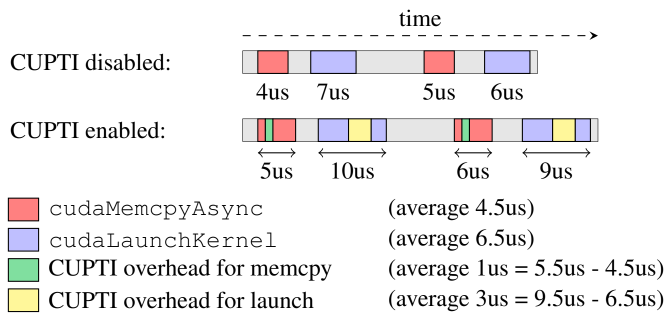

time
CUPTI disabled:
4us
7us
5us
6us
CUPTI enabled:
5us
10us
6us
9us
cudaMemcpyAsync (average 4.5us)
cudaLaunchKernel (average 6.5us)
CUPTI overhead for memcpy (average 1us = 5.5us - 4.5us)
CUPTI overhead for launch (average 3us = 9.5us - 6.5us)

Figure 10. Difference of average calibration of profiling overhead: `cudaLaunchKernel` and `cudaMemcpyAsync` experience different CPU time inflations when CUPTI (NVIDIA, 2019a) is enabled; we measure precise durations of CUDA API calls to compute book-keeping overheads for each CUDA API.

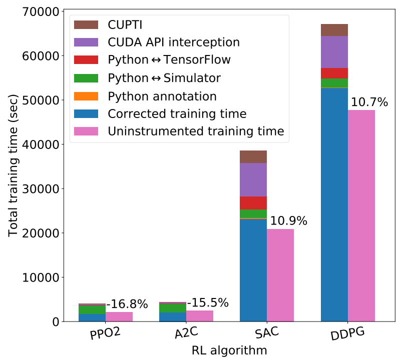
`cudaLaunchKernel` and `cudaMemcpyAsync` API calls by different amounts, and the inflation of each CUDA API cannot be enabled in isolation which is required for delta calibration. RL-Scope handles this using *difference of average calibration* which measures the average time of individual CUDA APIs with/without profiling enabled, and computes the overhead for each CUDA API using the difference of these averages. Figure 10 shows how RL-Scope calibrates for different profiling overheads in `cudaLaunchKernel` (3 us) `cudaMemcpyAsync` (1 us) induced by enabling CUDA's CUPTI profiling library.

C.3 Validating Accuracy of Overhead Correction

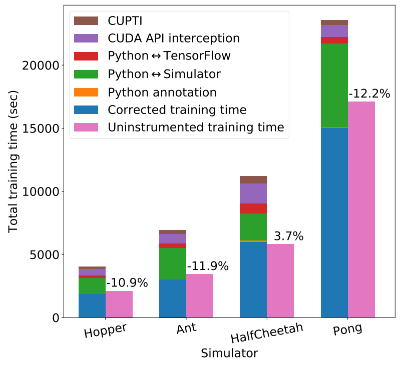
To evaluate the accuracy of overhead correction, we run each workload twice: once without RL-Scope (uninstrumented run), and once with full RL-Scope with all book-keeping code enabled. We compare the *corrected training time* reported by RL-Scope to the *uninstrumented training time*. If our overhead correction has low bias, then the corrected training time will be close to the uninstrumented training time. Figure 11 shows corrected (blue) and uninstrumented (pink) run times for different choices of environment (e.g., Walker2D, Ant, Pong) and RL algorithm (e.g., PPO2, A2C, DDPG). Regardless of the choice of RL algorithm and or simulator, RL-Scope is able to achieve accurate overhead correction of within $\pm 16\%$ of the actual uninstrumented training time, down from up to 90.2% without overhead correction.

C.4 Effect of Overhead Correction

To understand the impact of overhead correction on the case studies performed throughout this paper, we re-computed the RL framework case study (Section 4.1) without overhead correction, and observed how results of our analysis changed. First, we observed bottleneck shift with TensorFlow Eager DDPG (Figure 4b), where the largest contributor to CPU-bound DL backend time shifts from Inference *with* correction to Backpropagation *without* correction, which is due to overhead from a larger number of DL backend transitions in Backpropagation compared to Inference (Figure 4d). Second, findings related to GPU usage (Section 4.1.3) become incorrect since the total GPU-bound time becomes dwarfed by inflation in CPU time. For example, time spent in CUDA API calls exceeds GPU kernels by $5.7\times$ (up from $3.6\times$). Third, the total training time of the workloads we measured become inflated $1.6\times$ - $2.2\times$, and $1.8\times$ on average. Given that RL-Scope is a profiling tool for heterogeneous CPU/GPU RL workloads, it is important that RL-Scope provides accurate CPU-bound and GPU-bound metrics, otherwise developers will spend their time optimizing artificial bottlenecks.



(a) *Algorithm choice.* The same Walker2D simulator is used for different choices of RL algorithm.



(b) *Simulator choice.* The same PPO2 RL algorithm is used for different choices of simulators.

Figure 11. Low-bias overhead correction: The left stack of bars shows *Full RL-Scope*, and the right bar shows an *Uninstrumented* run. The bias (%) shows the deviation of the *Corrected training time* (blue, left) from the *Uninstrumented time* (pink, right). RL-Scope is able to achieve low-bias correction of critical path latency, regardless of simulator or algorithm.

D RL-SCOPE ARTIFACT EVALUATION

In our open-source documentation, we provide detailed instructions for reproducing some of the figures from this paper in a reproducible docker container. You can find these instructions here: `https://rl-scope.readthedocs.io/en/latest/artifacts.html`.