SLM Lab: A Comprehensive Benchmark and Modular Software Framework for Reproducible Deep Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

We introduce SLM Lab, a software framework for reproducible reinforcement learning (RL) research. SLM Lab implements a number of popular RL algorithms, provides synchronous and asynchronous parallel experiment execution, hyperparameter search, and result analysis. RL algorithms in SLM Lab are implemented in a modular way such that differences in algorithm performance can be confidently ascribed to differences between algorithms, not between implementations. In this work we present the design choices behind SLM Lab and use it to produce a comprehensive single-codebase RL algorithm benchmark. In addition, as a consequence of SLM Lab's modular design, we introduce and evaluate a discrete-action variant of the Soft Actor-Critic algorithm (Haarnoja et al., 2018) and a hybrid synchronous/asynchronous training method for RL agents.

1 Introduction

Progress in reinforcement learning (RL) research proceeds only as quickly as researchers can implement new algorithms and publish reproducible empirical results. But it is no secret that modern RL algorithms are hard to implement correctly [Tucker et al., 2018], and many empirical results are challenging to reproduce [Henderson et al., 2017] [Islam et al., 2017] [Machado et al., 2017]. Addressing these problems is aided by providing better software tools to the RL research community.

In this work we introduce SLM Lab[1], a software framework for reinforcement learning research designed for reproducibility and extensibility. SLM Lab is the first open source library that includes algorithm implementations, parallelization, hyperparameter search, and experiment analysis in one framework.

After presenting the design and organization of SLM Lab, we demonstrate the correctness of its implementations by producing a comprehensive performance benchmark of RL algorithms across 77 environments. To our knowledge this is the largest single-codebase RL algorithm comparison in the literature. In addition, we leverage the modular design of SLM Lab to introduce a variant of the SAC algorithm for use in discrete-action-space environments and a hybrid synchronous/asynchronous training scheme for RL algorithms.

2 SLM Lab

2.1 Library Organization

Modularity is the central design choice in SLM Lab, as depicted in Figure[1] Reinforcement learning algorithms in SLM Lab are built around three base classes:

- **Algorithm**: Handles interaction with the environment, implements an action policy, computes the algorithm-specific loss functions, and runs the training step.
- **Net**: Implements the deep networks that serve as the function approximators for an Algorithm.

[1] https://github.com/kengz/SLM-Lab
• **Memory**: Provides the data storage and retrieval necessary for training.

The **Net** and **Memory** classes abstract data storage, data retrieval and network training details, simplifying algorithm implementations. Furthermore, many **Algorithm** classes are natural extensions of each other.

![Figure 1: SLM Lab classes and their inheritance structure.](image)

Modular code is critical for deep RL research because many RL algorithms are extensions of other RL algorithms. If two RL algorithms differ in only a small way, but a researcher compares their performance by running a standalone implementation of each algorithm, they cannot know whether differences in algorithm performance are due to meaningful differences between the algorithms or merely due to quirks in the two implementations. Henderson et al. (2017) showcase this, demonstrating significant performance differences between different implementations of the same algorithm.

Modular code is also important for research progress. It makes it as simple as possible for a researcher to implement — and reliably evaluate — new RL algorithms. And for the student of RL, modular code is easier to read and learn from due to its brevity and organization into digestible components.

Proximal Policy Optimization (PPO) (Schulman et al., 2017) is a good example. When considered as a stand alone algorithm, PPO has a number of different components. However, it differs from the Actor-Critic algorithm only in how it computes the policy loss, runs the training loop, and by needing to maintain an additional actor network during training. Figure 2 shows how this similarity is reflected in the SLM Lab implementation of PPO.

The result is that the PPO class in SLM Lab has five overridden methods and contains only about 140 lines of code. Implementing it was straightforward once the ActorCritic class was implemented and thoroughly tested. More importantly, we can be sure that the performance difference between Actor-Critic and PPO observed in experiments using SLM Lab, shown below in Section 3, are due to something in the 140 lines of code that differentiate ActorCritic and PPO, and not to other implementation differences.

Another example of modularity in SLM Lab is that, thanks to the consistent API shared between Algorithm, Net, and Memory subclasses, synchronous parallelization using vector environments (Dhariwal et al., 2017) can be combined with asynchronous parallelization of an individual RL agent’s learning algorithm. This multi-level parallelization is further discussed in Section 3.3, where we demonstrate its performance benefits.
2.2 **Experiment Organization**

Reinforcement learning algorithms vary greatly in their performance across different environments, hyperparameter settings, and even within a single environment due to inherent randomness. SLM Lab is designed to easily allow users to study all these types of variability.

SLM Lab organizes experiments in the following hierarchy, shown in Figure 3:

1. **Session** The lowest level of the SLM Lab experiment framework: a single training run of a one agent on one environment with one set of hyperparameters, all with a fixed random seed.

2. **Trial** A trial consists of multiple Sessions, with the Sessions varying only in the random seed.

3. **Experiment** Generates different sets of hyperparameters (according to a spec file, see below) and runs a Trial for each one. It can be thought of as a study, e.g. “What values of gamma and learning rate provide the fastest, most stable solution, if the other variables are held constant?”

SLM Lab automatically produces plots for Sessions, Trials, and Experiments for any combination of environments and algorithms. It also logs and tracks metrics during training such as rewards, loss, exploration and entropy variables, frames-per-second and wall-clock time. Hyperparameter search results are automatically analyzed and presented hierarchically in increasingly granular detail.
2.3 Reproducibility

The complexity of RL algorithms makes reproducing RL results challenging (Henderson et al., 2017; Machado et al., 2017). Every RL researcher knows the difficulty of trying to reproduce an algorithm from its description in a paper alone. Even if code is published along with RL research results, the key algorithm design choices (or mistakes (Tucker et al., 2018)) are often buried in the algorithm implementation and not exposed naturally to the user.

In SLM Lab, every configurable hyperparameter for an algorithm is specified in a spec file. The spec file is a JSON file containing a git SHA and all the information required to reproduce a Session, Trial, or Experiment as per Section 2.2. Reproducing the entirety of an RL experiment merely requires checking out the code at the git SHA and running the saved spec file.

The main entries in a spec file are given below. Examples of spec files are given in full in Supplementary Section A.4.

1. **agent** - A list (to allow for multi-agent environments), each element of which contains the spec for a single agent. Each agent spec contains the details for its components as described in Section 2.1:
   
   (a) **algorithm**. The main parameters specific to the algorithm, such as the policy type, algorithm coefficients, rate decays, and training schedules.
   
   (b) **memory**. Specifies which memory to use as appropriate to the algorithm along with any specific memory hyperparameters such as the batch size and the memory size.
   
   (c) **net**. The type of neural network, its hidden layer architecture, activations, gradient clipping, loss function, optimizer, rate decays, update method, and CUDA usage.

2. **env** - A list (to allow for multi-environment agents), each element of which specifies an environment. Each environment spec includes an optional maximum time step per episode, the total time steps (frames) in a Session, the state and reward preprocessing methods, and the number of environments in a vector environment.

3. **body** - Specifies how (multi-)agents connect to (multi-)environments.

4. **meta** - The high-level configuration of how the experiment is to be run. It gives the number of Trials and Sessions to run, the evaluation and logging frequency, and a toggle to activate asynchronous training.

5. **search** - The hyperparameters to search over and the methods used to sample them. Any variables in the spec file can be searched over, including environment variables.

3 Results

Reporting benchmark results is essential for validating algorithm implementations. SLM Lab maintains a benchmark page and a public directory containing all of the experiment data, models, plots, and spec files associated with the reported results. We welcome contributions to this benchmark page via a pull request.

We tested the algorithms implemented in SLM Lab on 77 environments: 62 Atari games and 11 Roboschool environments available through OpenAI gym (Brockman et al., 2016) and 4 Unity environments (Juliani et al., 2018). These environments span discrete (Table 1) and continuous (Table 2) control problems with high- and low-dimensional state spaces.

The results we report in each of the tables are the score per episode at the end of training averaged over the previous 100 training checkpoints. Agents were checkpointed every 10k (Atari) or 1k (Roboschool, Unity) training frames. This measure is less sensitive to rapid increases or decreases in performance, instead reflecting average performance over a substantial number of training frames.

To our knowledge, the results we present below are a more comprehensive performance comparison than has been previously published for a single codebase. A full set of learning curves as well as a full table of results for the Atari environments are provided in the supplementary materials.
3.1 EXPERIMENT DETAILS

A complete set of spec files listing all hyperparameters for all algorithms and experiments are included in the `slm_lab/spec/benchmark` directory of SLM Lab as well as in the experiment data released along with the results. Example spec files listing all of the hyperparameters for PPO and DDQN + PER on the Atari and Roboschool environments are included in Supplementary Section A.4.

For the Atari environments, agents were trained for 10M frames (40M accounting for skipped frames). For the Roboschool environments, agents were trained for 2M frames except for RoboschoolHumanoid (50M frames), RoboschoolHumanoidFlagrun (100M) and RoboschoolHumanoidFlagrunHarder (100M). For the Unity environments, agents were trained for 2M frames. Training was parallelized either synchronously or with a hybrid of synchronous and asynchronous methods.

Our results for PPO and A2C on Atari games are comparable the results published by Schulman et al. (2017). The results on DQN and DDQN + PER on Atari games are mixed: at the same number of training frames we sometimes match or exceed the reported results and sometimes perform worse. This is likely due to three hyperparameter differences. We used a replay memory of size 200,000 compared to 1M in Mnih et al. (2015), van Hasselt et al. (2015), and Schaul et al. (2015). The final output layer of the network is smaller fully-connected layer with 256 instead of 512 units. Finally, our results for SAC confirm the strength of this algorithm compared to PPO. However the absolute performance is typically worse than the published results from Haarnoja et al. (2018b). Due to computational constraints, SAC was trained with a replay buffer of 0.2M elements and combined experience replay (Zhang & Sutton, 2017) compared with 1M elements in Haarnoja et al. (2018b) and this is potentially a significant difference.

Table 1: Episode score at the end of training attained by SLM Lab implementations on discrete-action control problems. Reported episode scores are the average over the last 100 checkpoints, and then averaged over 4 Sessions. Results marked with ‘*’ were trained using the hybrid synchronous/asynchronous version of SAC to parallelize and speed up training time.

<table>
<thead>
<tr>
<th>Environment</th>
<th>DQN</th>
<th>DDQN+PER</th>
<th>A2C (GAE)</th>
<th>A2C (n-step)</th>
<th>PPO</th>
<th>SAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakout</td>
<td>80.88</td>
<td>182</td>
<td>377</td>
<td>398</td>
<td>443</td>
<td>-</td>
</tr>
<tr>
<td>Pong</td>
<td>18.48</td>
<td>20.5</td>
<td>19.31</td>
<td>19.56</td>
<td>20.58</td>
<td>19.87*</td>
</tr>
<tr>
<td>Seaquest</td>
<td>1185</td>
<td>4405</td>
<td>1070</td>
<td>1684</td>
<td>1715</td>
<td>-</td>
</tr>
<tr>
<td>Qbert</td>
<td>5494</td>
<td>11426</td>
<td>12405</td>
<td>13590</td>
<td>13460</td>
<td>214*</td>
</tr>
<tr>
<td>LunarLander</td>
<td>192</td>
<td>233</td>
<td>25.21</td>
<td>68.23</td>
<td>214</td>
<td>276</td>
</tr>
<tr>
<td>UnityHallway</td>
<td>-0.32</td>
<td>0.27</td>
<td>0.08</td>
<td>-0.96</td>
<td>0.73</td>
<td>-</td>
</tr>
<tr>
<td>UnityPushBlock</td>
<td>4.88</td>
<td>4.93</td>
<td>4.68</td>
<td>4.93</td>
<td>4.97</td>
<td>-</td>
</tr>
</tbody>
</table>

3.2 SOFT ACTOR-CRITIC FOR DISCRETE ENVIRONMENTS

All published results for the Soft Actor-Critic (SAC) algorithm (Haarnoja et al., 2018a,b) are for continuous control environments. However, nothing in its algorithmic description makes it unsuitable in principle for use in discrete action-space environments. As a consequence of the modular structure of SLM Lab, it was straightforward to design and implement a discrete variant of SAC. We did so by using policy Nets that produced parameters of a Gumbel-Softmax distribution (Jang et al., 2016; Maddison et al., 2016) from which discrete actions were sampled. The results of this discrete variant of SAC are in Table 1.

However, even though SAC trained successfully on Pong and Lunar Lander, we were not able to successfully train it on all Atari environments. We also note that while SAC is sample efficient, it is more computationally expensive than other algorithms, which presents an obstacle for extensive performance tuning.

3 Estimated using the training curves provided in Figure 7 of Schaul et al. (2015)
Table 2: Episode score at the end of training attained by SLM Lab implementations on continuous control problems. Reported episode scores are the average over the last 100 checkpoints, and then averaged over 4 Sessions. Results marked with ‘*’ require 50M-100M frames, so we use the hybrid synchronous/asynchronous version of SAC to parallelize and speed up training.

<table>
<thead>
<tr>
<th>Environment</th>
<th>A2C (GAE)</th>
<th>A2C (n-step)</th>
<th>PPO</th>
<th>SAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoboschoolAnt</td>
<td>787</td>
<td>1396</td>
<td>1843</td>
<td>2915</td>
</tr>
<tr>
<td>RoboschoolAtlasForwardWalk</td>
<td>59.87</td>
<td>88.04</td>
<td>172</td>
<td>800</td>
</tr>
<tr>
<td>RoboschoolHalfCheetah</td>
<td>712</td>
<td>439</td>
<td>1960</td>
<td>2497</td>
</tr>
<tr>
<td>RoboschoolHopper</td>
<td>710</td>
<td>285</td>
<td>2042</td>
<td>2045</td>
</tr>
<tr>
<td>RoboschoolInvertedDoublePendulum</td>
<td>996</td>
<td>4410</td>
<td>8076</td>
<td>8085</td>
</tr>
<tr>
<td>RoboschoolInvertedPendulum</td>
<td>995</td>
<td>978</td>
<td>986</td>
<td>941</td>
</tr>
<tr>
<td>RoboschoolReacher</td>
<td>12.9</td>
<td>10.16</td>
<td>19.51</td>
<td>19.99</td>
</tr>
<tr>
<td>RoboschoolWalker2d</td>
<td>280</td>
<td>220</td>
<td>1660</td>
<td>1894</td>
</tr>
<tr>
<td>RoboschoolHumanoid</td>
<td>99.31</td>
<td>54.58</td>
<td>2388</td>
<td>2621*</td>
</tr>
<tr>
<td>RoboschoolHumanoidFlagrun</td>
<td>73.57</td>
<td>178</td>
<td>2014</td>
<td>2056*</td>
</tr>
<tr>
<td>RoboschoolHumanoidFlagrunHarder</td>
<td>-429</td>
<td>253</td>
<td>680</td>
<td>280*</td>
</tr>
<tr>
<td>Unity3DBall</td>
<td>33.48</td>
<td>53.46</td>
<td>78.24</td>
<td>98.44</td>
</tr>
<tr>
<td>Unity3DBallHard</td>
<td>62.92</td>
<td>71.92</td>
<td>91.41</td>
<td>97.06</td>
</tr>
</tbody>
</table>

3.3 Hybrid synchronous and asynchronous training

Synchronous and asynchronous parallelization can be combined in SLM Lab to accelerate training, as shown in Figure 4. SLM Lab implements synchronous parallelization within Sessions (Section 2.2) using vector environments (Dhariwal et al., 2017) and asynchronous parallelization within Trials (Section 2.2) using multiple workers, one per Session. There are two available methods for Trial level parallelization; Hogwild! (Recht et al., 2011) or a server-worker model in which workers periodically push gradients to a central network and pull copies of the updated parameters.

If training is constrained by data sampling from the environment, then increasing the number of vector environments (synchronous parallelization) speeds up training. But this speed-up saturates as the training step becomes a bottleneck and the environment waits for the agent to train. In the example shown in Figure 4 the frames per seconds (fps) increases from around 200 for a single worker and 1 environment to around 360 for 1 worker and 16 environments, and saturates thereafter. Once fps becomes constrained by the training step, it is beneficial to add workers (asynchronous parallelization) to effectively parallelize the parameter updates. A hybrid of 16 workers each with 4 environments resulted in the maximum fps of around 3800.

4 Related Work

4.1 Reproducibility in Reinforcement Learning

The instability of RL algorithms (Haarnoja et al. [2018a]), randomness in agent policies and the environment (Henderson et al. [2017]; Islam et al. [2017]), as well as differences in hyperparameter tuning (Islam et al. [2017]) and implementations (Henderson et al. [2017]; Tucker et al. [2018]) all contribute to the challenge of reproducing RL results.

Consequently, the importance of comprehensively documenting all hyperparameters along with published results and software is well recognized within the research community (Machado et al. [2017]; Castro et al. [2018]).

4.2 Software for Reinforcement Learning

To date more than twenty reinforcement-learning-themed open source software libraries have been released. They can be organized into two categories: those implementing RL algorithms and those
Figure 4: Average frames per second as number of vector environments and Hogwild! workers are varied. Each setting was trained using PPO with the same hyperparameters on the RoboschoolAnt environment.

SLM Lab is an algorithm-focused library with built-in integration with the OpenAI gym (Brockman et al., 2016), OpenAI Roboschool, VizDoom (Kempka et al., 2016), and Unity ML-Agents (Juliani et al., 2018) environment libraries.

Table 3 summarizes the algorithm-focused reinforcement learning software libraries.

Libraries such as ChainerRL (chainer, 2017; Tokui & Oono, 2015), coach (Caspi et al., 2017), DeepRL (Zhang, 2017), OpenAI baselines (Dhariwal et al., 2017), RLgraph (Scharschmidt et al., 2019), RLkit (Pong, 2018), rpyt (Stooke & Abbeel, 2017), RLLib (Liang et al., 2017), Stable Baselines (Hill et al., 2018), TensorForce (Kühnle et al., 2017), TF-Agents (Guadarrama et al., 2018), and vel (Tworek, 2018) implement a wide variety of algorithms and are intended to be applied to a variety of RL problems. Most of these libraries also provide some benchmark results for the implemented algorithms to validate their performance. These can be thought of as generalist RL libraries and are the most closely related to SLM Lab.

Other libraries focus on specific algorithms (Dopamine (Castro et al., 2018), Softlearning (Haarnoja et al., 2018a), a2c-ppo-acktr-gail (Kostrikov, 2018), Keras-RL (Plappert, 2016), problems (OpenSpiel (Lanctot et al., 2019), ELF (Tian et al., 2019), MAgent (Zheng et al., 2017), reaver (Ring, 2018)) or components such as loss functions (TRFL (DeepMind, 2018) or scaling training (Horizon (Gauci et al., 2018)).

The use of configuration files to specify hyperparameters varies. coach, DeepRL, Dopamine, Horizon, reaver, Softlearning, RLgraph, RLLib, and vel use configuration files and provide a number of configured examples. In most cases the network architecture is excluded from the main configuration file and specified elsewhere. Horizon and RLgraph include all hyperparameters in a single configuration file and are the most similar to SLM Lab. However, RLgraph does not include environment information, and neither use the same file for configuring hyperparameter search.

Almost all libraries include some methods for parallelizing agent training, especially for on-policy methods. However most do not include hyperparameter optimization as a feature. The two exceptions are Stable Baselines (Hill et al., 2018) and Tensorforce (Kühnle et al., 2017).

A few libraries such as OpenSpiel (Lanctot et al., 2019) and ELF (Tian et al., 2019) implement both.
Table 3: Comparison of RL software libraries. Algorithm acronyms are explained in Supplementary Section A.1. REINFORCE is excluded as are less well-known algorithms. “Benchmark” indicates whether the library reports the performance of their implementations. “Config” indicates whether hyperparameters are specified separately from the implementation and run scripts; “split” indicates that the configuration is divided across multiple files; “partial” indicates that some but not all hyperparameters are included. “Parallel” denotes whether training for any algorithms can be parallelized. “HPO” is a hyperparameter optimization feature. “Plot” denotes whether the library provides any methods for visualizing results.

<table>
<thead>
<tr>
<th>Library</th>
<th>Algorithms</th>
<th>Benchmark</th>
<th>Config</th>
<th>Parallel</th>
<th>HPO</th>
<th>Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2c-ppo-acktr-gail</td>
<td>A2C, ACKTR, GAIL, PPO</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baselines</td>
<td>A2C, ACER, ACKTR, DDPG, DQN, GAIL,</td>
<td>✓</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HER, PPO, TRPO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChainerRL</td>
<td>A3C, ACER, C51, DDPG, DQN+, IQN,</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NSQ, PPO, Rainbow, TRPO, TD3, SAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coach</td>
<td>A3C, ACER, C51, DDPG, DQN+, QR-DQN,</td>
<td>✓</td>
<td>split</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NAF, NEC, NSQ, PPO, Rainbow, SAC,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TD3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepRL</td>
<td>A2C, C51, DDPG, DQN+, NSQ, OC, PPO,</td>
<td>✓</td>
<td>split</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PPO, QR-DQN, TD3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dopamine</td>
<td>DQN, C51, Rainbow</td>
<td>✓</td>
<td>split</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELF</td>
<td>A3C, DQN, MCTS, TRPO</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keras-RL</td>
<td>DDPG, DQN+, NAF, SARSA</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CEM, SAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Horizon</td>
<td>C51, DQN+, SAC, TD3</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAgent</td>
<td>A2C, DQN</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSpiel</td>
<td>A2C, DQN, MCTS</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pytorch-rl</td>
<td>A2C, DDPG, DQN+, HER</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reaver</td>
<td>A2C, PPO</td>
<td>✓</td>
<td>split</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RLgraph</td>
<td>A2C, Ape-X, DQN+, DQFD, IMPALA, PPO,</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RLkit</td>
<td>DQN+, HER, SAC, TDM, TD3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RLLib</td>
<td>A3C, Ape-X, ARS, DDPG, DQN, ES,</td>
<td>✓</td>
<td>partial</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IMPALA, PPO, SAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rlpy</td>
<td>A2C, DDPG, DQN+, CAT-DQN, PPO,</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TD3, SAC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Softlearning</td>
<td>SAC</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stable Baselines</td>
<td>A2C, ACER, ACKTR, DDPG, DQN, GAIL,</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HER, PPO, SAC, TD3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TensorForce</td>
<td>A3C, DQN+, DQFD, NAF, PPO, TRPO</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TF-Agents</td>
<td>DDPG, DQN+, PPO, SAC, TD3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRFL</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vel</td>
<td>A2C, ACER, DDPG, DQN+, Rainbow, PPO</td>
<td>✓</td>
<td>split</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRPO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLM Lab</td>
<td>A2C, A3C, CER, DQN+, PPO, SAC</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finally, many libraries include some tools for visualizing and plotting results. Notably coach (Caspi et al., 2017) provides an interactive dashboard for exploring a variety of metrics which are automatically tracked during training.
REFERENCES


A SUPPLEMENTARY MATERIALS

A.1 ALGORITHM ACRONYMS

The expanded acronyms for all of the algorithms listed in Table 3 are given below:

- A2C: Advantage Actor-Critic
- ACER: Actor-Critic with Experience Replay
- ACKTR: Actor-Critic using Kronecker-Factored Trust Region
- Ape-X: Distributed Prioritized Experience Replay
- ARS: Augmented Random Search
- C51: Categorical DQN
- CAT-DQN: Categorical DQN
- CEM: Cross Entropy Method
- CER: Combined Experience Replay
- DDPG: Deep Deterministic Policy Gradients
- DQN: Deep Q Networks
- DQN+: Deep Q Network modifications, including some or all of the following: Double DQN, Dueling DQN, Prioritized Experience Replay
- DQFD: Deep Q-Learning from Demonstrations
- ES: Evolutionary Strategies
- GAIL: Generative Adversarial Imitation learning
- HER: Hindsight Experience Replay
- IMPALA: Importance Weighted Actor-Learner Architecture
- IQN: Implicit Quantile Networks
- MCTS: Monte Carlo Tree Search
- NAF: Normalized Advantage Functions
- NEC: Neural Episodic Control
- NSQ: n-step Q-learning
- OC: Option-Critic
- PPO: Proximal Policy Optimization
- QR-DQN: Quantile Regression DQN
- Rainbow:
- TD3: Twin Delayed Deep Deterministic Policy Gradient
- TRPO: Trust Region Policy Optimization
- SAC: Soft Actor-Critic
- A.2 ATARI RESULTS

The table below presents results for 62 Atari games. All agents were trained for 10M frames (40M including skipped frames). Reported results are the episode score at the end of training, averaged over the previous 100 evaluation checkpoints with each checkpoint averaged over 4 Sessions. Agents were checkpointed every 10k training frames.

<table>
<thead>
<tr>
<th>Environment (Atari, Discrete)</th>
<th>DQN</th>
<th>DDQN+PER</th>
<th>Algorithm A2C (GAE)</th>
<th>A2C (n-step)</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adventure</td>
<td>-0.94</td>
<td>-0.92</td>
<td>-0.77</td>
<td>-0.85</td>
<td>-0.3</td>
</tr>
<tr>
<td>Adventure</td>
<td>-0.94</td>
<td>-0.92</td>
<td>-0.77</td>
<td>-0.85</td>
<td>-0.3</td>
</tr>
<tr>
<td>AirRaid</td>
<td>1876</td>
<td>3974</td>
<td>4202</td>
<td>3557</td>
<td>4028</td>
</tr>
<tr>
<td>Alien</td>
<td>822</td>
<td>1574</td>
<td>1519</td>
<td>1627</td>
<td>1413</td>
</tr>
<tr>
<td>Amidar</td>
<td>90.95</td>
<td>431</td>
<td>577</td>
<td>418</td>
<td>795</td>
</tr>
<tr>
<td>Assault</td>
<td>1392</td>
<td>2567</td>
<td>3366</td>
<td>3312</td>
<td>3619</td>
</tr>
<tr>
<td>Asterix</td>
<td>1253</td>
<td>6866</td>
<td>5559</td>
<td>5223</td>
<td>6132</td>
</tr>
<tr>
<td>Asteroids</td>
<td>439</td>
<td>426</td>
<td>2951</td>
<td>2147</td>
<td>2186</td>
</tr>
<tr>
<td>Atlantis</td>
<td>68679</td>
<td>644810</td>
<td>2747371</td>
<td>2259733</td>
<td>2148077</td>
</tr>
<tr>
<td>BankHeist</td>
<td>131</td>
<td>623</td>
<td>855</td>
<td>1170</td>
<td>1183</td>
</tr>
<tr>
<td>BattleZone</td>
<td>6564</td>
<td>6395</td>
<td>4363</td>
<td>4533</td>
<td>13649</td>
</tr>
<tr>
<td>BeamRider</td>
<td>2799</td>
<td>5870</td>
<td>2659</td>
<td>4139</td>
<td>4299</td>
</tr>
<tr>
<td>Berzerk</td>
<td>319</td>
<td>401</td>
<td>1073</td>
<td>763</td>
<td>860</td>
</tr>
<tr>
<td>Bowling</td>
<td>30.29</td>
<td>39.5</td>
<td>24.51</td>
<td>23.75</td>
<td>31.64</td>
</tr>
<tr>
<td>Boxing</td>
<td>72.11</td>
<td>90.98</td>
<td>1.57</td>
<td>1.26</td>
<td>96.53</td>
</tr>
<tr>
<td>Breakout</td>
<td>80.88</td>
<td>182</td>
<td>377</td>
<td>398</td>
<td>443</td>
</tr>
<tr>
<td>Carnival</td>
<td>4280</td>
<td>4773</td>
<td>2473</td>
<td>1827</td>
<td>4566</td>
</tr>
<tr>
<td>Centipede</td>
<td>1899</td>
<td>2153</td>
<td>3909</td>
<td>4202</td>
<td>5003</td>
</tr>
<tr>
<td>ChopperCommand</td>
<td>1083</td>
<td>4020</td>
<td>3043</td>
<td>1280</td>
<td>3357</td>
</tr>
<tr>
<td>CrazyClimber</td>
<td>46984</td>
<td>88814</td>
<td>106256</td>
<td>109998</td>
<td>116820</td>
</tr>
<tr>
<td>Defender</td>
<td>281999</td>
<td>313018</td>
<td>665609</td>
<td>657823</td>
<td>534639</td>
</tr>
<tr>
<td>DemonAttack</td>
<td>1705</td>
<td>19856</td>
<td>23779</td>
<td>19615</td>
<td>121172</td>
</tr>
<tr>
<td>DoubleDunk</td>
<td>-21.44</td>
<td>-22.38</td>
<td>-5.15</td>
<td>-13.3</td>
<td>-6.01</td>
</tr>
<tr>
<td>ElevatorAction</td>
<td>32.62</td>
<td>17.91</td>
<td>9966</td>
<td>8818</td>
<td>6471</td>
</tr>
<tr>
<td>Enduro</td>
<td>437</td>
<td>959</td>
<td>787</td>
<td>0.0</td>
<td>1926</td>
</tr>
<tr>
<td>FishingDerby</td>
<td>-88.14</td>
<td>-1.7</td>
<td>16.54</td>
<td>1.65</td>
<td>36.03</td>
</tr>
<tr>
<td>Freeway</td>
<td>24.46</td>
<td>30.49</td>
<td>30.97</td>
<td>0.0</td>
<td>32.11</td>
</tr>
<tr>
<td>Frostbite</td>
<td>98.8</td>
<td>2497</td>
<td>277</td>
<td>261</td>
<td>1062</td>
</tr>
<tr>
<td>Gopher</td>
<td>1095</td>
<td>7562</td>
<td>929</td>
<td>1545</td>
<td>2933</td>
</tr>
<tr>
<td>Gravitar</td>
<td>87.34</td>
<td>258</td>
<td>313</td>
<td>433</td>
<td>223</td>
</tr>
<tr>
<td>Hero</td>
<td>1051</td>
<td>12579</td>
<td>16502</td>
<td>19322</td>
<td>17412</td>
</tr>
<tr>
<td>IceHockey</td>
<td>-14.96</td>
<td>-14.24</td>
<td>-5.79</td>
<td>-6.06</td>
<td>-6.43</td>
</tr>
</tbody>
</table>
### A.3 LEARNING CURVES FOR ALL ALGORITHMS AND ENVIRONMENTS

Each learning curve depicts the episode score, averaged over the previous 100 evaluation checkpoints, with each checkpoint averaged over 4 Sessions. The shaded area depicts +/- one standard deviation calculated over 4 Sessions.
A.4 Example Spec files

```json
{
  "ddqn_per_atari": {
    "agent": [{
      "name": "DoubleDQN",
      "algorithm": {
        "name": "DoubleDQN",
        "action_pdtype": "Argmax",
        "action_policy": "epsilon_greedy",
        "explore_var_spec": {
          "name": "linear_decay",
          "start_val": 1.0,
          "end_val": 0.01,
          "start_step": 10000,
          "end_step": 1000000
        },
        "gamma": 0.99,
        "training_batch_iter": 1,
        "training_iter": 4,
        "training_frequency": 4,
        "training_start_step": 10000
      },
      "memory": {
        "name": "PrioritizedReplay",
        "alpha": 0.6,
        "epsilon": 0.0001,
        "batch_size": 32,
        "max_size": 200000,
        "use_cer": false,
      },
      "net": {
        "type": "ConvNet",
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [64, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [256],
        "hid_layers_activation": "relu",
        "init_fn": null,
        "batch_norm": false,
        "clip_grad_val": 10.0,
        "loss_spec": {
          "name": "SmoothL1Loss"
        },
        "optim_spec": {
          "name": "Adam",
          "lr": 2.5e-5,
        },
        "lr_scheduler_spec": null,
        "update_type": "replace",
        "update_frequency": 1000,
        "gpu": true
      }
    }],
    "env": {
      "name": "${env}"
    }
  }]
}
```

Listing 1: The Double DQN + PER spec file for Atari games

```json
{
  "ppo_atari": {
    "agent": [{
      "name": "PPO",
      "algorithm": {
        "name": "PPO",
        "action_pdtype": "default",
        "action_policy": "default",
        "explore_var_spec": null,
        "gamma": 0.99,
        "lam": 0.70,
        "clip_eps_spec": {
          "name": "no_decay",
          "start_val": 0.10,
          "end_val": 0.10,
          "start_step": 0,
          "end_step": 0
        },
        "entropy_coef_spec": {
          "name": "no_decay",
          "start_val": 0.01,
          "end_val": 0.01,
          "start_step": 0,
          "end_step": 0
        },
        "val_loss_coef": 0.5,
        "time_horizon": 128,
        "minibatch_size": 256,
        "training_epoch": 4
      },
      "memory": {
        "name": "OnPolicyBatchReplay",
      },
      "net": {
        "type": "ConvNet",
        "shared": true,
        "conv_hid_layers": [
          [32, 8, 4, 0, 1],
          [64, 4, 2, 0, 1],
          [32, 3, 1, 0, 1]
        ],
        "fc_hid_layers": [512],
      }
    }
  }
}
```

Under review as a conference paper at ICLR 2020
"hid_layers_activation": "relu",
"init_fn": "orthogonal_",
"normalize": true,
"batch_norm": false,
"clip_grad_val": 0.5,
"use_same_optim": false,
"loss_spec": {
  "name": "MSELoss"
},
"actor_optim_spec": {
  "name": "Adam",
  "lr": 2.5e-4,
},
"critic_optim_spec": {
  "name": "Adam",
  "lr": 2.5e-4,
},
"lr_scheduler_spec": {
  "name": "LinearToZero",
  "frame": 1e7
},
"gpu": true
],
"env": [{
  "name": "${env}",
  "frame_op": "concat",
  "frame_op_len": 4,
  "reward_scale": "sign",
  "num_envs": 16,
  "max_t": null,
  "max_frame": 1e7
}],
"body": {
  "product": "outer",
  "num": 1
},
"meta": {
  "distributed": false,
  "eval_frequency": 10000,
  "log_frequency": 10000,
  "rigorous_eval": 0,
  "max_session": 4,
  "max_trial": 1
},
"spec_params": {
  "env": [
    "BreakoutNoFrameskip-v4", "PongNoFrameskip-v4", "QbertNoFrameskip-v4", "SeaquestNoFrameskip-v4"
  ]
}
}

Listing 2: The PPO spec file for Atari games
"gamma": 0.99,
"lam": 0.95,
"clip_eps_spec": {
  "name": "no_decay",
  "start_val": 0.20,
  "end_val": 0.20,
  "start_step": 0,
  "end_step": 0
},
"entropy_coef_spec": {
  "name": "no_decay",
  "start_val": 0.0,
  "end_val": 0.0,
  "start_step": 0,
  "end_step": 0
},
"val_loss_coef": 1.0,
"time_horizon": 2048,
"minibatch_size": 128,
"training_epoch": 10
},
"memory": {
  "name": "OnPolicyBatchReplay",
},
"net": {
  "type": "MLPNet",
  "shared": false,
  "hid_layers": [256, 256],
  "hid_layers_activation": "relu",
  "init_fn": "orthogonal_",
  "clip_grad_val": 0.5,
  "use_same_optim": false,
  "loss_spec": {
    "name": "MSELoss"
  },
  "actor_optim_spec": {
    "name": "Adam",
    "lr": 3e-4,
  },
  "critic_optim_spec": {
    "name": "Adam",
    "lr": 3e-4,
  },
  "lr_scheduler_spec": null,
  "gpu": false
},
"env": [{
  "name": "${env}",
  "num_envs": 8,
  "max_t": null,
  "max_frame": 2e6
}],
"body": {
  "product": "outer",
  "num": 1
},
"meta": {
  "distributed": false,
  "log_frequency": 1000,
  "eval_frequency": 1000,
  "rigorous_eval": 0,
  "max_session": 4,
  "max_trial": 1
},
Implementing a number of well-known deep reinforcement learning algorithms taught us many important lessons. Such lessons are typically relegated to blog posts and personal conversations, but we feel it is worth sharing such knowledge in the research literature.

- **Put everything** that varies into the spec file. e.g. when implementing A2C we hardcoded state normalization in some places but not others. This extended the debugging process by days.
- **Log policy entropy**, $Q$ or $V$ function outputs and check that they change. Looking at $Q$ function values for a random agent helped us to track down an error due to state mutation.
- **As with all Python code**, watch out for mutation. - e.g. We noticed that the state returned from OpenAI baselines vector environment was mutated by preprocessing for the next time steps. This was because the state was tracked as internal variables as returned directly. To remedy this, the state must be returned as a copy.
- **Start simple** - implement parent algorithms first. e.g. REINFORCE, then A2C, then PPO or SAC, and ensure that each of these work in turn. PPO and SAC were straightforward to debug once A2C was working.
- **Write tests**: Tests have been invaluable in debugging and growing SLM Lab. Especially useful are tests for tricky functions. For example, a test for the GAE calculation was essential for debugging A2C. Additionally, use Continuous Integration to automatically build and test every new code changes in a Pull Request, as is done in SLM Lab.
- **Check input and output tensor shapes** for important calculations. e.g. predicted and target values for $Q$ or $V$. Broadcasting errors may lead to incorrect tensor shapes which fail silently.
- **Check the size of the loss** at the beginning of training. Is this comparable to other implementations? Tracking this value was essential when debugging DQN. It led us to a bug in the image permutation and we also stopped normalizing the image values as a result.
- **Check that the main components of the computation graph** are connected as expected. This can be implemented as a decorator around the training step and run when debugging. Conversely, be careful to detach target values.
- **Visualize trained agents**. We forgot to evaluate agents with reward clipping turned off, which led to very significant underestimation of performance for all environments except Pong. We finally realized training was working only after deciding to look at some policies playing the game.