SLM Lab

A Modular Deep Reinforcement Learning Framework in PyTorch

Laura Graesser, Wah Loon Keng, Siraj Raval

October 4, 2018

Abstract

SLM Lab[1] is a modular deep reinforcement learning framework built using PyTorch. It implements a number of canonical algorithms using modular and reusable components, provides an automated experiment and analytics framework focusing on reproducibility, introduces a multi-dimensional fitness metric, and integrates with OpenAI Gym and Unity ML-Agents. This whitepaper discusses the features and design principles of SLM Lab.

1 Introduction

Reinforcement learning (RL) is a subfield of artificial intelligence that grew out of optimal control theory and Markov decision processes (MDP). RL is concerned with sequential decision making in the context of an unknown objective function. Unlike supervised learning, algorithms which try to solve RL problems do not have access to the "correct answer" during the learning phase. Instead, an RL system consists of an agent and an environment, subject to alternating interactions in succession. The environment provides a signal, known as the state, to the agent, who then observes it, deliberates, and performs an action. The environment receives the action, transitions into the next state, and provides feedback to the agent in the form of a reward and the next state. The cycle then repeats until the environment terminates. The objective or goal for the agent in this system is to maximize the discounted sum of rewards received over time. For this reason, RL has been referred to as "learning with a critic" (the rewards received) in contrast to the supervised learning setup of "learning with a teacher" (the correct action to take in every state) [34].

Deep learning is a method for approximating highly complex non-linear functions by learning from data. Recently, deep learning has been combined with RL to great effect. It is being applied in a number of areas including robotics [1, 8, 25], navigation [26], datacenter cooling [10], and natural language understanding [21]. It has also led to expert level or superhuman play in a number of games: Backgammon [32], Atari video games [14, 13], Go [30, 31], Dota [17, 18, 19], and Chess [29]. However a number of challenges remain.

It is well known that deep RL (DRL) algorithms are often sample inefficient[6], struggle to generalize beyond the specific task they were trained on[6], can be unstable [5, 6], and results are hard to reproduce [23, 5, 6]. The combination of three factors makes learning difficult: complex non-linear function approximation with neural networks; non-stationary, potentially stochastic, environments with unknown dynamics; and inherent randomness in agent policies[5]. Consequently, for DRL to work effectively, many tricks are needed along with extensive hyperparameter tuning. This contributes to the challenge of reproducibility and also makes DRL hard to learn.

[1] https://github.com/kengz/SLM-Lab

SLM Lab[9] is a modular DRL framework built using PyTorch[22] created for DRL research and applications. It emerged from the authors facing these problems and wanting to make a contribution towards solving them. The design was guided by four principles: **modularity**, **simplicity**, **analytical clarity**, and **reproducibility**.

Modularity: component design in SLM Lab is modular. For example, an agent consists of an algorithm, network(s), policy, and memory, which interact through a common API. Consequently, many components are re-used by different agents. Algorithms also share many functions. All algorithms are derived from one of two base classes, SARSA[24] or REINFORCE[35], and only the necessary incremental code is implemented for each new algorithm. This structure makes research easier and more accessible by enabling users to focus only on the relevant work when implementing new algorithms. It also makes it easier to learn DRL by breaking down the complex DRL algorithms into more manageable, digestible components. Moreover, when components are maximally reused, there is less code, fewer bugs, and more test coverage.

Simplicity: the lab components are designed to closely correspond to the way papers or books discuss RL. This reduces the burden of understanding ideas, or translating ideas to implementations . We understand that a modular library is not necessarily simple. Simplicity serves to balance modularity to prevent overly complex abstractions that are difficult to understand and use.

Analytical clarity: experiment results are automatically analyzed and presented hierarchically in increasingly granular detail. A big experiment with multiple trials contains a lot of data. With hierarchical results, one can quickly look at the experiment graph (Figure 2) to determine if it was a success overall, then pick the best trials and drill down to their trial (Figure 1b) and session (Figure 1a) graphs, which contains lower level details such as reward and loss.

Reproducibility: a long term aspiration in DRL is that all research is reproducible. To do so, one needs the source code, hyperparameters, and result data. Using SLM Lab, an experiment can be fully reproduced using a spec file, which exposes all the hyperparameters, and a git SHA, which allows one to checkout the version of the code used to run it. These and the experiment results are submitted to the lab via a Pull Request. The complete experiment data is uploaded to a public cloud storage for open access, and the result is tracked in the lab's benchmark page.

2 Features

The main contributions of SLM Lab can be grouped into three parts. First, it implements most of the canonical algorithms in DRL. This is enabled by a standardized modular design with well-tested, reusable components. Second, SLM Lab provides an automated experiment and analytics framework. The framework is organized into three tiers: experiment, trial, and session. It incorporates hyperparameter search, distributed training, replication with random seeds, automated analysis, and introduces a fitness metric. Third, it integrates with OpenAI gym and Unity ML-Agents[2], and has an API for adding environments from other sources.

2.1 Algorithms

SLM Lab implements many reinforcement learning algorithms, listed below:

[2]Pre-built binaries for a number of Unity ML-Agents environments are packaged with SLM Lab.

- SARSA[24], DQN[14], DDQN[3], DuelingDQN[33], and their asynchronous versions[12]
- Prioritized Experience Replay[27], Combined Experience Replay[36]
- REINFORCE[35], A2C, PPO[28], and their asynchronous versions including A3C[12] and DPPO[4]
- (A2C)SIL and PPOSIL[16]

Implementations are organized into standardized modular components that closely correspond to how they are discussed in DRL papers and books: algorithm, network, replay memory, and policy. The modular design does not compromise simplicity, making it efficient to translate between the code and theory.

These components are grouped under an `Agent` class, which handles interaction with the environments. Primarily, the `Algorithm` class controls the other components and their interactions, computes the algorithm-specific loss functions, and runs the training step. Under it, `Net` class serves as the function approximators for the algorithm, while the `Memory` class provides the necessary data storage and retrieval for training. `Policy` is a generalized abstraction which takes a network output layer to construct a probability distribution used for sampling actions and calculating log probabilities and entropies.

The implemented `Net` classes are `MLPNet, HydraMLPNet, DuelingMLPNet, RecurrentNet, ConvNet, DuelingConvNet`. There are two types of `Memory` classes: `Replay, SeqReplay, SILReplay, SILSeqReplay, ConcatReplay, AtariReplay, PrioritizedReplay, AtariPrioritizedReplay` for off-policy replay, and `OnPolicyReplay, OnPolicySeqReplay, OnPolicyBatchReplay, OnPolicySeqBatchReplay, OnPolicyAtariReplay` for on-policy replay. The `policy` module handles discrete and continuous controls, with exploration methods like epsilon-greedy and Boltzmann.

Most of the above are implemented concisely using class inheritance. Furthermore, the lab provides a standard API for them to interact and integrate. These together make it easy to implement a new component and have it immediately apply to all relevant algorithms. As a result, components are maximally reused and enjoy the benefits of shorter code, fewer bugs, and more test coverage. With this, the workflow becomes lightweight and fast with the lab, as one can use the existing components and focus only on those under research or development.

To illustrate, the hogwild asynchronous training of A3C[12] is implemented by constructing and updating the global networks using the `Net` API, so it applies to all the algorithms in the lab. For example, one can turn A2C into A3C with by specifying `"distributed": true` in the spec file, as shown in appendix listing 2. PPO + SIL is implemented using multi-inheritance by symbolically calling the parent classes `SIL` and `PPO` without any newly written code. Likewise, listing 1 shows how `AtariPrioritizedReplay` inherits from the existing `PrioritizedReplay` and `AtariReplay`, again without new code.

Listing 1 AtariPrioritizedReplay is implemented without new code from the existing PrioritizedReplay and AtariReplay classes.

```
# source: slm_lab/agent/memory/prioritized.py
from slm_lab.agent.memory.replay import Replay, AtariReplay


class AtariPrioritizedReplay(PrioritizedReplay, AtariReplay):
    '''Make a Prioritized AtariReplay via nice multi-inheritance'''
    pass
```

Such implementation techniques have saved significant time and effort when developing with SLM Lab,

and have maximized the benefits of existing reliable components. This has been the main enabling factor for expanding SLM Lab to include many algorithms.

2.2 Experimentation and analysis

SLM Lab provides an automated experimentation and analytics framework. The main features include:

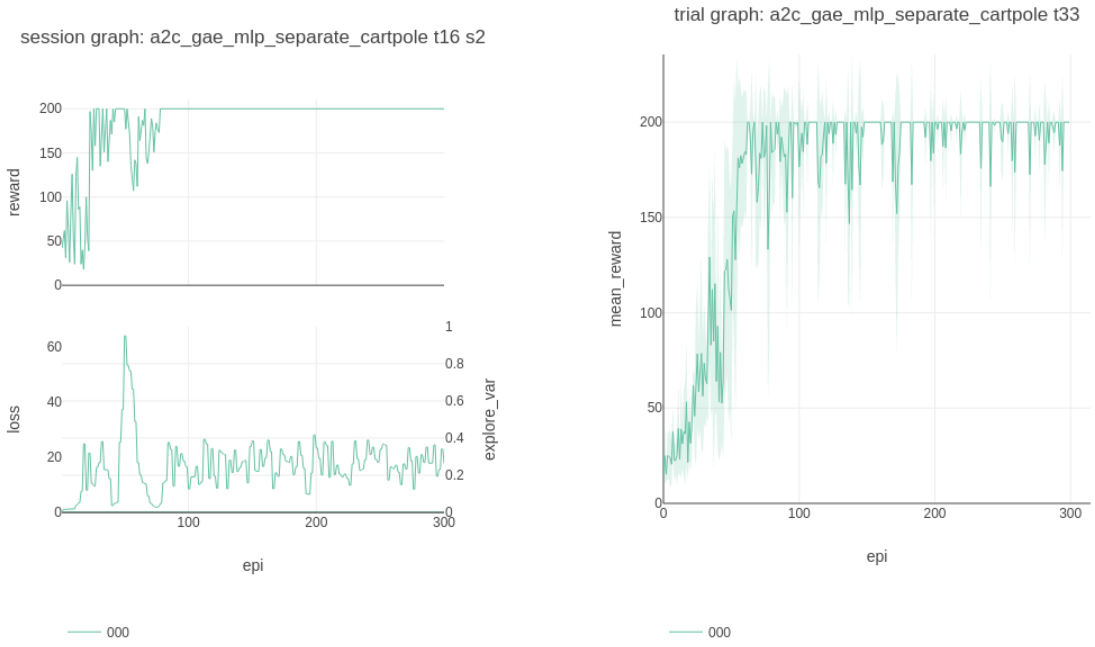
- hyperparameter search using Ray Tune[15, 11]
- distributed training which automatically scales to the available CPU and GPU resources
- automatic replication of training with different random seeds for the same hyperparameter settings
- automatic analysis and plotting of results. These are presented at multiple levels of granularity. Summarizing results in this way makes it easy and fast to get an overview of an experiment and drill down into detailed results if desired
- agent fitness metric. Instead of considering just the total reward, the lab also measures the convergence speed, stability and consistency to provide a richer set of metrics

Framework components The experiment framework is organized hierarchically into three components:

1. **Session**: A session is the lowest level of SLM Lab. It initializes the agents and environments, and runs the control loop in a single process. Each session runs for some defined maximum episodes; each episode runs for some maximum number of timesteps before resetting the environments. A session is associated with an algorithm plus a particular set of hyperparameter values *and* a distinct random seed. The output of a session is an instance of a trained agent, with a particular set of learned parameters.
2. **Trial**: A trial holds a fixed configuration (a spec) which is an algorithm plus a particular set of hyperparameter values. A trial uses its spec to run multiple sessions with different random seeds to measure reproducibility. It then analyzes the sessions and takes the average. If a trial's configuration is a stable solution, then all the sessions should solve the environment with similar performance. This is the basic idea behind the "consistency" aspect of the fitness score.
3. **Experiment**: An experiment is the highest level of SLM Lab's experiment framework. It can be thought of as a study, e.g. "what values of gamma and learning rate provide the fastest, most stable solution, while the other variables are held constant?" The input variables are encoded as the hyperparameters, and the outcome is measured by the fitness vector. An experiment runs a trial for every set of parameters generated by a hyperparameter search algorithm (random search for example), and a trial runs multiple replicated, seeded sessions to average the results. Then, the search is to find the hyperparameters that yield maximum fitness for a given experiment. The hyperparameters to search over, their range, and sampling method, are specified in a spec file.

The organization above, when combined with the standard modular component design, helps expose all the hyperparameters of an algorithm using a JSON spec file. This allows for wider search options, as well as reducing the number of hidden hyperparameters, which has contributed to the problem of reproducibility. As a result, an experiment can be fully reproduced in SLM Lab using just the JSON spec file and the correct git commit of the code repository, which is specified with the git SHA that recorded automatically in the generated spec during an experiment.

Experiment data Experiments generate data which is automatically saved to the `data/<experiment_id>` folder. Each session generates a plot of the rewards and loss per episode, a table of metrics per episode (e.g, reward, loss, exploration variable value, episode length), a session's fitness, and checkpoints of the algorithm models. See Figure 1a for an example.



(a) Session graph

(b) Trial graph

Figure 1: Session and Trial graphs
A2C in Cartpole-v0[2] environment

Each trial generates a plot of the average reward per episode plus error bars, a spec file detailing the specific hyperparameter values, and a trial's fitness. See Figure 1b for an example.

Each experiment generates a summary plot of each trial's fitness for each of the different hyperparameters searched over. This plot makes it clear which parameters an algorithm is sensitive to, and which values work best. The multi-dimensional fitness metric also allows one to measure the effect on convergence speed, stability and consistency. It is immediately clear from this graph whether any trials succeeded. See Figure 2 for an example.

The `experiment_df` CSV file contains a summary of each trial sorted from highest to lowest fitness. It is intended to clearly show the hyperparameter value ranges and combinations from the most successful to the least. Finally, each experiment saves a spec file detailing the search specifications and other hyperparameter settings, along with the SHA of the git commit used to run it. They encapsulates all the information needed to reproduce an experiment.

All of these reproduction instructions and experiment data can be contributed back to the SLM Lab via a Github Pull Request, which follows the format of a scientific report to encourage reproducibility. The complete experiment data is uploaded to a public cloud storage for open access, and the result is tracked in the lab's benchmark page. Both of them are linked from the SLM Lab website.

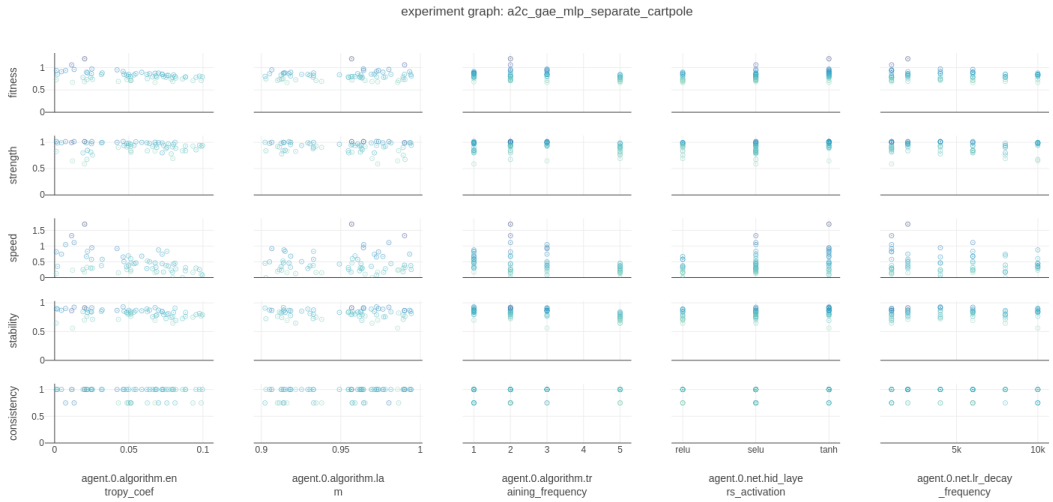


Figure 2: Experiment graph: A2C performance on Cartpole-v0[2], with a high level view of the effects of the searched hyperparameters on various fitness metrics.

2.3 Fitness metric

The lab introduces a vectorial fitness metric as a richer measurement of an agent's performance. It measures an agent's strength (in terms of rewards), speed (how fast it attains its strength), stability (once learned, is the solution stable), and consistency (how reproducible is the trial result). It serves a dual purpose in the lab:

1. provide a standard metric to compare algorithms and environments
2. provide a value for the hyperparameter optimizer to maximize

The fitness metric was motivated by the fact it is common to perform hundreds or thousands of hyperparameter searches in the course of DRL research, even when working on a single algorithm or environment[3]. However it is not always straightforward to conclude which are the best set of hyperparameter values. Which are "best" depends on what aspect of performance you are looking at. Does the agent achieve high reward? Does it learn fast? Is learning stable? Is learning reproducible? Reinforcement learning research typically reports on the first two metrics; the total rewards achieved, and the speed of learning (number of time steps to achieve a result). Increasingly the variance of the rewards over time is also reported. However, it is not common to measure the stability during training.

What then are the desirable properties of a performance, or fitness, metric? The authors propose the following properties:

- **multi-dimensional**: to measure multiple metrics
- **standardizable**: to allow comparison among different experiments

[3]SLM Lab's experiment framework is intended to make this easier to do

- **extensible, backward compatible**: when new metrics are added, the old vector becomes a special case, similar to how `x,y` is a restriction of `x,y,z` space. It is not necessary to re-run experiments when fitness is updated.
- **reducible to a single number**: fitness = L2 norm of the fitness vector.
- **magnification**: under L2 norm, higher fitness vectors have bigger differences than lower fitness vectors. e.g. $2^2 - 1.9^2 > 0.9^2 - 0.8^2$
- **easy to understand**: with familiar, simple definitions, and a nice geometric interpretation

A *fitness vector* satisfies all of the above. Given a fitness vector, we can also ensure that every dimension (and the resultant norm fitness) has these properties:

- the value is **0 for a random agent**
- the value is **1 for a standard, universal baseline**
- the value is **higher for better** fitness; the scale should not be too extreme

SLM Lab proposes a metric that satisfies these properties, the fitness metric. Appendix A details how SLM Lab's fitness vector and the resulting fitness metric are calculated.

2.4 Environments

Currently, SLM Lab integrates with environments from OpenAI gym[2] and Unity ML-Agents[7]. Even though the interface methods of these two are different, they are standardized into the environment API of the lab using a lightweight `Env` wrapper class. The same technique can also be applied to add new environments into the lab in the future.

The lab makes use of the environment properties to automatically inference the cardinality and shape of the state and action spaces, as well as the type of actions, such as discrete, continuous, or a hybrid from multi-environments. This information is further used by the `policy` module to construct the proper probability distribution for sampling actions.

Integrating multiple environments expands the testing ground for all the algorithms in the lab, allowing us to study their behaviors in greater variations. An interesting consequence from standardizing the environment APIs is that the lab can create composite environments, such as one from OpenAI gym and one from Unity ML-Agents, to let a multitask agent solve both simultaneously. Moreover, the lab has a `Body` class which handles the interface between agent and environment. This is especially useful for keeping track of the interfaces in multi-agent, multi-environment settings, which the lab too provides.

2.5 Python package

SLM Lab is portable. Its algorithms and components can be imported and used outside of the framework. This makes it possible to develop with SLM Lab then deploy the agent elsewhere in a software application, with minimal effort. To do so, one simply needs to install SLM Lab as a pip module by doing a `pip install -e .` at the repository root, and require any of its components as a normal Python module. This is further detailed in the documentation of the lab.

3 Example Results

SLM-Lab aims to provide a set of results on well known DRL environments, including the Atari suite[2]. All data from the associated experiments are uploaded to the SLM Lab following the Pull Request procedure discussed earlier. One advantage of providing benchmarks using SLM-Lab is that if researchers implement algorithms using the same framework, it is easier to consistently compare the performance of algorithms.

Some example results are provided in the tables below. These examples indicate the type of information that is available from the experiments. It includes the more typical metrics; episode reward, 100-episode running mean reward, as well as the agent's fitness, overall and by component.

One interesting consequence of testing algorithms using SLM-Lab is that it is easy to evaluate the effect of a specific component; for example self imitation learning (SIL) for PPO, or combined experience replay (CER) for DQN. Two experiments can be run with identical setups, with and without the specific component of interest.

	Algorithm			
Environment	DQN	DDQN*	A2C**	A2C GAE***
Cartpole	200.0 / 126	200.0 / 219	200.0 / 110	200.0 / 176
Lunar Lander	226.5 / 296	226.6 / 406	-	191.0 / na
Mountain Car	-104.6 / 1067	-101.5 / 1063	-	-

Table 1: Best 100 episode average rewards during training / minimum episode to solve (first episode for which the 100 episode average $\geq$ solved reward)

Solved score for each of the environments: Cartpole: 195.0, Lunar Lander: 200.0, Mountain Car: -110.0
*Double DQN **A2C with nstep forward returns ***A2C with Generalized Advantage Estimation

	Algorithm			
Environment	DQN	DDQN	A2C	A2C GAE
Cartpole	3.52 (4.79*)	0.85 (5.65*)	7.10	1.20 (6.26**)
Lunar Lander	1.14	1.15	-	0.57***
Mountain Car	0.92	0.81	-	-

Table 2: Agent fitness

*+ Combined Experience Replay **+ Self Imitation Learning. *** Low score driven by lack of consistency, see Table 3.

	Algorithm			
Environment	DQN	DDQN	A2C	A2C GAE
Cartpole	0.76 / 9.84 / 0.67 / 0.75	0.89 / 0.55 / 0.90 / 1.0*	1.02 / 14.13 / 0.93 / 0.75	1.01 / 1.7 / 0.91 / 1.0
Lunar Lander	1.05 / 1.61 / 0.73 / 1.0	1.02 / 1.66 / 0.72 / 1.0	-	0.92 / 0.26 / 0.63 / 0.0**
Mountain Car	1.01 / 0.98 / 0.63 / 1.0	1.02 / 0.45 / 0.65 / 1.0	-	-

Table 3: Agent fitness components: strength / speed / stability / consistency

*Stronger solutions but less stable solutions existed. **The strongest solution found so far, but not consistent

4 Future work

We hope to continue work on SLM Lab in several directions:

1. *Expand the benchmark results.* All the algorithms will gradually be benchmarked on more environments, including the Atari suite and robotics tasks. Providing a set of results on the Atari suite is a priority for SLM Lab development going forward.
2. *Implement new algorithms and components.* For example, curiosity[20] and QT-Opt[8] are the next algorithms we plan to implement. For more details, see the Roadmap available on SLM Lab's GitHub repository.
3. *Research and applications.* We hope the Lab will support users to publish research and deploy DRL in industrial applications.

5 Acknowledgements

We thank the School of AI for their support. We also thank Milan Cvitkovic from Caltech for his help and contributions.

References

[1] Marcin Andrychowicz et al. "Learning Dexterous In-Hand Manipulation". In: (2018). URL: https://arxiv.org/abs/1808.00177.

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

[3] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[4] Nicolas Heess et al. "Emergence of Locomotion Behaviours in Rich Environments". In: *CoRR* abs/1707.02286 (2017). arXiv: 1707.02286. URL: http://arxiv.org/abs/1707.02286.

[5] Peter Henderson et al. "Deep Reinforcement Learning that Matters". In: *CoRR* abs/1709.06560 (2017). arXiv: 1709.06560. URL: http://arxiv.org/abs/1709.06560.

[6] Alex Irpan. *Deep Reinforcement Learning Doesn't Work Yet*. https://www.alexirpan.com/2018/02/14/rl-hard.html. 2018.

[7] Arthur Juliani et al. *Unity: A General Platform for Intelligent Agents*. 2018. eprint: arXiv:1809.02627.

[8] Dmitry Kalashnikov et al. "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". In: *CoRR* abs/1806.10293 (2018). arXiv: 1806.10293. URL: http://arxiv.org/abs/1806.10293.

[9] Wah Loon Keng and Laura Graesser. *SLM-Lab*. https://github.com/kengz/SLM-Lab. 2017.

[10] Will Knight. *Google just gave control over data center cooling to an AI*. https://www.technologyreview.com/s/611902/google-just-gave-control-over-data-center-cooling-to-an-ai/. 2018.

[11] Richard Liaw et al. "Tune: A Research Platform for Distributed Model Selection and Training". In: *arXiv preprint arXiv:1807.05118* (2018).

[12] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1928–1937. URL: http://dl.acm.org/citation.cfm?id=3045390.3045594.

[13] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

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

[15] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *CoRR* abs/1712.05889 (2017). arXiv: 1712.05889. URL: http://arxiv.org/abs/1712.05889.

[16] Junhyuk Oh et al. "Self-Imitation Learning". In: (2018). URL: https://arxiv.org/abs/1806.05635.

[17] OpenAI. *More on DotA 2*. 2017. URL: https://blog.openai.com/more-on-dota-2/.

[18] OpenAI. *OpenAI Five*. 2018. URL: https://blog.openai.com/openai-five/.

[19] OpenAI. *OpenAI Five Benchmark: Results*. 2018. URL: https://blog.openai.com/openai-five-benchmark-results/.

[20] Deepak Pathak et al. "Curiosity-driven Exploration by Self-supervised Prediction". In: *ICML*. 2017.

[21] Romain Paulus, Caiming Xiong, and Richard Socher. "A Deep Reinforced Model for Abstractive Summarization". In: *CoRR* abs/1705.04304 (2017). arXiv: 1705.04304. URL: http://arxiv.org/abs/1705.04304.

[22] *PyTorch*. 2018. URL: https://github.com/pytorch/pytorch.

[23] Matthew Rahtz. *Lessons Learned Reproducing a Deep Reinforcement Learning Paper*. http://amid.fish/reproducing-deep-rl. 2018.

[24] G. A. Rummery and M. Niranjan. *On-Line Q-Learning Using Connectionist Systems*. Tech. rep. 1994.

[25] Fereshteh Sadeghi. *Teaching Uncalibrated Robots to Visually Self-Adapt*. https://ai.googleblog.com/2018/06/teaching-uncalibrated-robots-to_22.html. 2018.

[26] Fereshteh Sadeghi and Sergey Levine. "CAD2RL: Real Single-Image Flight without a Single Real Image". In: *CoRR* abs/1611.04201 (2016). arXiv: 1611.04201. URL: http://arxiv.org/abs/1611.04201.

[27] Tom Schaul et al. "Prioritized Experience Replay". In: *CoRR* abs/1511.05952 (2015). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[28] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[29] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[30] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[31] David Silver et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.

[32] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon". In: *Commun. ACM* 38.3 (Mar. 1995), pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: http://doi.acm.org/10.1145/203330.203343.

[33] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.

[34] B. Widrow, N. K. Gupta, and S. Maitra. "Punish/Reward: Learning with a Critic in Adaptive Threshold Systems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-3.5 (1973), pp. 455–465. ISSN: 0018-9472. DOI: 10.1109/TSMC.1973.4309272.

[35] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3-4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[36] Shangtong Zhang and Richard S. Sutton. "A Deeper Look at Experience Replay". In: *CoRR* abs/1712.01275 (2017). arXiv: 1712.01275. URL: http://arxiv.org/abs/1712.01275.

A Fitness metric

Definition SLM Lab measures fitness with the following:

- **strength**: in terms of total rewards
- **speed**: how fast does it attain its strength?
- **stability**: once learned, is the solution stable?
- **consistency**: how reproducible is the trial result?

A.1 Strength

For each episode, use the total rewards to calculate the strength as

$$strength_{epi} = \frac{reward_{epi} - reward_{rand}}{reward_{std} - reward_{rand}}$$

which is the ratio of the current performance to the standard performance, with random offset.

Properties

- random agent has strength 0, standard agent has strength 1.
- if an agent achieve x2 rewards, the strength is x2, and so on.
- strength of learning agent always tends toward positive regardless of the sign of rewards (some environments use negative rewards)
- scale of strength is always standard at 1 and its multiplies, regardless of the scale of actual rewards. Strength stays invariant even as reward gets rescaled.

This allows for standard comparison between agents on the same problem using an intuitive measurement of strength. With proper scaling by a difficulty factor, we can compare across problems of different difficulties.

A.2 Speed

For each session, measure the moving average (MA) for strength with interval = 100 episodes.

$$strength\_ma_{epi} = MA(strength_{epi}, interval = 100)$$

The reason for using moving average is for stability: It smoothes out the noise in strength variations across episodes; so we do not falsely consider a "lucky shot" where the agent gains high strength in one episode and is weak for the rest.

Next, measure the total timesteps up to the first episode that first surpasses standard strength, allowing for noise of 0.05, i.e.

$$\text{Let } epi_{solved} = \text{first episode where } strength\_ma_{epi} \geq 1$$

and

$$timestep_{solved} = \begin{cases} \text{total timesteps through } epi_{solved} & \text{if } \exists epi_{solved} \\ \infty & \text{otherwise} \end{cases}$$

Finally, calculate speed as

$$speed = \frac{timestep_{std}}{timestep_{solved}}$$

Properties

- random agent has speed 0, standard agent has speed 1.
- if an agent takes x2 timesteps to exceed standard strength, we can say it is 2x slower.
- the speed of learning agent always tends toward positive regardless of the shape of the rewards curve
- the scale of speed is always standard at 1 and its multiplies, regardless of the absolute timesteps.

For agent failing to achieve standard strength 1, it is meaningless to measure speed or give false interpolation, so the speed is 0.

This allows an intuitive measurement of learning speed and the standard comparison between agents on the same problem.

Absolute timestep also measures the bits of new information given to the agent, which is a more grounded metric. With proper scaling of timescale (or bits scale), we can compare across problems of different difficulties. This is a topic for research.

A.3 Stability

Find a

$$baseline = \begin{cases} 0. + noise & \text{if solution very weak} \\ max(strength\_ma_{epi}) - noise & \text{if solution partial} \\ 1. - noise & \text{if solution strong} \end{cases}$$

So we get:

$$\begin{aligned} baseline_{weak} &= 0. + noise \\ baseline_{strong} &= min(max(strength\_ma_{epi}), 1.) - noise \\ baseline &= max(baseline_{weak}, baseline_{strong}) \end{aligned}$$

Let $epi_{baseline}$ be the episode where *baseline* is first attained. Consider the episodes starting from $epi_{baseline}$, let $\#epi_{+}$ be the number of episodes, and $\#epi_{\geq}$ the number of episodes where $strength\_ma_{epi}$ is monotonically increasing, allowing for noise of 0.05, i.e. 5% of standard strength.

Calculate stability as

$$stability = \begin{cases} \frac{\#epi_{\geq}}{\#epi_{+}} & \text{if } \exists epi_{solved} \\ 0 & \text{otherwise} \end{cases}$$

Properties:

- stable agent has value 1, unstable agent < 1, and non-solution $= 0$.
- allows for drops strength MA of 5% of standard strength, which is invariant to the scale of rewards
- if strength is monotonically increasing (with 5% noise), then it is stable
- sharp gain in strength is considered stable
- monotonically increasing implies strength can keep growing and as long as it does not fall much, it is considered stable

When an agent fails to achieve standard strength, it is meaningless to measure stability or give false interpolation, so stability is 0.

A.4 Consistency

Often in search, due to many interacting factors, the same setting can yield vastly different results. This motivates a trial-level measurement of consistency - to see how similar and reproducible are different runs of the same trial. Given multiple sessions of the same trial and their session-level fitness vectors $\{\mathbf{v_s}\}$, let $\{\mathbf{v_s}\}_{in}$ be the set of non-outlier vectors by MAD modified z-score, calculate consistency as the ratio of non-outliers:

$$consistency = \begin{cases} \frac{|\{\mathbf{v_s}\}_{in}|}{|\{\mathbf{v_s}\}|} & \text{if } \exists epi_{solved} \text{ for any session} \\ 0 & \text{otherwise} \end{cases}$$

Properties

- outliers are calculated using MAD modified z-score
- if all the fitness vectors are zero or all strength are zero, consistency = 0
- works for all sorts of session fitness vectors, with the standard scale
- When an agent fails to achieve standard strength, it is meaningless to measure consistency or give false interpolation, so consistency is 0.

A.5 Fitness as vector norm

With all the dimensions defined, we can define the fitness vector. For a session, the fitness vector and fitness respectively are:

$$\mathbf{v_s} := [strength, speed, stability]$$
$$\kappa_s := norm_{L2}(\mathbf{v_s})$$

For a trial of multiple sessions, obtain the mean vector $\overline{\mathbf{v_s}}$, and the final fitness vector and fitness respectively are:

$$\mathbf{v} := \overline{\mathbf{v_s}} \oplus [consistency]$$
$$\kappa := norm_{L2}(\mathbf{v})$$

B Experiment configuration: the spec file

A spec file is used by the lab Session to construct agent and environment. Its format is standardized in accordance with the standard modular component design. SLM Lab will read the lab config to find the appropriate spec file and use a spec in it to construct an experiment. The documentation page elaborates these in details. A quick overview of the spec file and two examples are presented below.

1. **agent**. The format is a list to accommodate multi-agents. An element in a list is an agent spec which further contains the specs for its components:

   (a) **algorithm**. The main parameters specific to the algorithm, such as the policy type, coefficients, rate decays, preprocessing.

   (b) **memory**. Specifies which memory to use as appropriate to the algorithm and neural network, its batch size, and 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**. The format is also a list to accommodate multi-environments. This specifies which environment to use, an optional maximum timestep, the max episode per session, and frequency for saving the session (agent models, etc.)

3. **body**. Specifies how multi-agent connects to multi-environments; leave at default for singleton case. Refer to lab documentation for more details.

4. **meta**. The high level spec specifying how the lab is ran. Includes distributed mode, the number of jobs, search algorithm, resource allocation.

5. **search**. The hyperparameter ranges to search over, and the distributions used to sample them. This uses the analogous spec format enumerated above, with the distributed method append like `__choice`, and the values are the sampling domain.


```
{
  "a3c_gae_mlp_shared_cartpole": {
    "agent": [{
      "name": "A3C",
      "algorithm": {
        "name": "ActorCritic",
        "action_pdtype": "default",
        "action_policy": "default",
        "action_policy_update": "no_update",
        "explore_var_start": null,
        "explore_var_end": null,
        "explore_anneal_epi": null,
        "gamma": 0.99,
        "use_gae": true,
        "lam": 1.0,
        "use_nstep": false,
        "num_step_returns": 1,
        "add_entropy": true,
        "entropy_coef": 0.1,
        "policy_loss_coef": 1.0,
        "val_loss_coef": 1.0,
        "training_frequency": 1,
        "training_epoch": 4,
        "normalize_state": false
      },
      "memory": {
        "name": "OnPolicyReplay"
      },
      "net": {
        "type": "MLPNet",
```

```
      "shared": true,
      "hid_layers": [32],
      "hid_layers_activation": "tanh",
      "clip_grad": false,
      "clip_grad_val": 1.0,
      "use_same_optim": false,
      "actor_optim_spec": {
        "name": "Adam",
        "lr": 0.02
      },
      "critic_optim_spec": {
        "name": "Adam",
        "lr": 0.02
      },
      "lr_decay": "linear_decay",
      "lr_decay_frequency": 2000,
      "lr_decay_min_timestep": 5000,
      "lr_anneal_timestep": 100000,
      "gpu": false
    }
  }],
  "env": [{
    "name": "CartPole-v0",
    "max_timestep": null,
    "max_episode": 400,
    "save_epi_frequency": 300
  }],
  "body": {
    "product": "outer",
    "num": 1
  },
  "meta": {
    "distributed": true,
    "max_session": 16,
    "max_trial": 64,
    "search": "RandomSearch",
    "resources": {
      "num_cpus": 16
    }
  },
  "search": {
    "agent": [{
      "algorithm": {
        "lam__uniform": [0.9, 1.0],
        "training_frequency__choice": [1, 2, 3, 5],
        "entropy_coef__uniform": [0.001, 0.1],
        "val_loss_coef__uniform": [0.01, 1.0]
      },
      "net": {
        "lr_decay_frequency__choice": [1000, 2000, 4000, 6000, 8000, 10000],
        "hid_layers_activation__choice": ["tanh", "relu", "selu"]
```

```
        }
      }]
    }
  }
}
```

Listing 2: A3C is realized simply as A2C with distributed training enabled by a single meta config, `"distributed": true`.

```
{
  "dqn_boltzmann_cartpole": {
    "agent": [{
      "name": "DQN",
      "algorithm": {
        "name": "DQN",
        "action_pdtype": "Argmax",
        "action_policy": "boltzmann",
        "action_policy_update": "linear_decay",
        "explore_var_start": 3.0,
        "explore_var_end": 1.0,
        "explore_anneal_epi": 20,
        "gamma": 0.99,
        "training_batch_epoch": 10,
        "training_epoch": 4,
        "training_frequency": 8,
        "training_min_timestep": 32,
        "normalize_state": true
      },
      "memory": {
        "name": "Replay",
        "batch_size": 32,
        "max_size": 10000,
        "use_cer": true
      },
      "net": {
        "type": "MLPNet",
        "hid_layers": [64],
        "hid_layers_activation": "relu",
        "clip_grad": false,
        "clip_grad_val": 1.0,
        "loss_spec": {
          "name": "MSELoss"
        },
        "optim_spec": {
          "name": "Adam",
          "lr": 0.001
        },
        "lr_decay": "linear_decay",
        "lr_decay_frequency": 2000,
        "lr_decay_min_timestep": 5000,
        "lr_anneal_timestep": 100000,
```

```
        "update_type": "polyak",
        "update_frequency": 1,
        "polyak_coef": 0,
        "gpu": false
      }
    }],
    "env": [{
      "name": "CartPole-v0",
      "max_timestep": null,
      "max_episode": 400,
      "save_epi_frequency": 300
    }],
    "body": {
      "product": "outer",
      "num": 1
    },
    "meta": {
      "distributed": false,
      "max_session": 4,
      "max_trial": 64,
      "search": "RandomSearch",
      "resources": {
        "num_cpus": 16
      }
    },
    "search": {
      "agent": [{
        "algorithm": {
          "explore_anneal_epi__choice": [10, 50, 100]
        },
        "net": {
          "hid_layers_activation__choice": ["tanh", "relu", "selu"],
          "optim_spec": {
            "lr__uniform": [0.00001, 0.01]
          }
        }
      }]
    }
  }
}
```

Listing 3: This DQN spec specifies a variant that uses the Boltzmann exploration policy, with Polyak network weight update.