MDP Playground: Meta-Features in Reinforcement Learning

Raghu Rajan
Department of Computer Science
University of Freiburg
Freiburg, Germany
rajanr@cs.uni-freiburg.de

Frank Hutter
Department of Computer Science
University of Freiburg
Freiburg, Germany &
Bosch Center for Artificial Intelligence
fh@cs.uni-freiburg.de

Abstract

Reinforcement Learning (RL) algorithms usually do not try to identify specific features of environments which could help them perform better. Here, we present a few key *meta-features* of environments: delayed rewards, specific reward sequences, sparsity of rewards, and stochasticity of environments, adapting to which should help RL agents perform better. While it is very time consuming to run RL algorithms on standard benchmarks, we define a parameterised collection of fast-to-run toy benchmarks in OpenAI Gym by varying these meta-features. Despite their toy nature and low compute requirements, we show that these benchmarks present substantial difficulties to current RL algorithms. Furthermore, since we can generate environments with a desired value for each of the meta-features, we have fine-grained control over the environments' *difficulty* and also have the ground truth available for evaluating algorithms. We believe that devising algorithms that can detect such meta-features of environments and adapt to them will be key to creating robust RL algorithms that work in a variety of different real-world problems.

1 Introduction

Like humans, a true Artificial General Intelligence (AGI) would generalise to all kinds of environments by adapting to the task at hand. Despite the success of RL algorithms at many tasks (Abbeel et al., 2010; Mnih et al., 2013; Silver et al., 2016; Chua et al., 2018), we are still far away from AGI. RL algorithms can solve many tasks, such as the game of Go, video game playing in Atari, and locomotion in Mujoco, but when faced with a completely new environment, they can't really adapt like humans do. We need to understand the environments and their interactions with RL algorithms better if we are to progress to more intelligent algorithms.

RL algorithms usually assume the environment to be an MDP (or POMDP). However, the state formulation used may not be Markovian.[1] In the POMDP case, the fact that the true state needed to have Markovianness may not be accessible can have a significant impact on the performance of algorithms. In this work, we attempt to identify some key *meta-features* of POMDP environments which help characterise environments and we provide a platform with different instantiations of these meta-features in order to understand the workings of different RL algorithms better. The platform is implemented as a Python package, *MDP Playground* that allows us complete control over these meta-features to be able to benchmark RL algorithms.

[1] This depends on the state formulation in general. But assuming an infinitely differentiable world and dynamics of arbitrary order, we would need an infinite "stack" of the current "state" and its higher order derivatives to be able to predict the next state based on the current one and thus have Markovianness.

Submitted to 33rd Conference on Neural Information Processing Systems Deep RL Workshop (NeurIPS 2019). Do not distribute.

Usual environments that RL algorithms are tested on also tend to take a long time to run. As a remedy, our platform is meant to be a low cost proxy for identifying how RL algorithms would work on real world problems.

The main contributions of this paper are:

- Identifying meta-features to characterise POMDP environments
- Implementation of low-cost benchmarks with fine-grained control over the meta-features
- Experiments on baseline RL algorithms to highlight the impact of the meta-features

2 Key Meta-Features

We define an MDP as a 6-tuple (S, A, P, R, ρ_o, T), where S is the set of states, A is the set of actions, $P : S \times A \rightarrow S$ is the transitions dynamics, $R : S \times A \rightarrow \mathbb{R}$ is the reward dynamics, $\rho_o : S \rightarrow \mathbb{R}$ is the initial state distribution, and T is the set of terminal states.[2] For a POMDP, the state would not be completely observable to an agent.

In RL algorithms, usually the assumption is that we receive immediate reward depending on only the previous state and action. However, in general, this is not true for even many simple environments

Algorithms like DQN (Mnih et al., 2013) were applied to many varied environments and produce very variable performance across these. In some simple environments, DQN's performance exceeds human performance by large amounts, but in other environments, such as Montezuma's revenge, performance is very poor. For such environments, we need a very specific sequence of actions to get a reward. Another simple real world example is executing a tennis serve, where we need a very specific sequence of actions which would result in a point if we served an ace.

Environments where specific sequences of states need to be followed to get a reward also tend to have sparsity. This can significantly impact performance of algorithms.

Another characteristic of environments that can significantly impact performance of algorithms is stochasticity. The environment itself, i.e., dynamics P and R, may be stochastic or may seem stochastic to the agent due to partial observability or sensor noise.

Key meta-features of environments that we identify from the above discussion are

- Delayed Rewards
- Specific Sequences
- Sparsity of Rewards
- Stochasticity

These are usually not observable to the agents directly, making the environment a POMDP for the agent. We now describe the benchmark environment where we can control these meta-features to study their impact on the performance of various algorithms.

3 MDP Playground

We believe that the meta-features discussed above are key features which can be controlled in small environments and if an algorithm can master all of these in simple environments, it has gone some way towards being able to perform at scale as well on vastly different environments. In order to be able to benchmark how algorithms would perform in such environments, we implement a package, *MDP Playground*, which generates randomly configured OpenAI Gym environments to allow us to benchmark algorithms across a variety of meta-feature configurations. It is also manually configurable so we can control in a fine-grained manner how exactly we intend the environment to be. We now briefly describe how the environment and these meta-features are implemented.

[2] We don't include the discount factor γ as we feel it is more an intrinsic feature of the learning algorithm than of the MDP.

The environment is implemented as an MDP with an *augmented state*. For the current discussion, we restrict ourselves to discrete state and action spaces.[3] Some additional meta-features that we allow the user to fully control are the *terminal state density* and the *reward unit* which is the reward given whenever the environment hands out one (this is intended to help test algorithms with different reward scales in an environment). We also allow the user to fully specify an MDP they may have in mind and use it for their experiments.

Algorithm 1 Generating random MDPs with MDP Playground

Input: number of states $|S|$, number of actions $|A|$, reward delay d, length of reward sequences n, density of reward sequences rd, transition noise t_n, reward noise σ_{r_n}, $reward\_unit$, $make\_denser$, $terminal\_state\_density$

function INIT_TRANSITION_FUNCTION() ▷ For generating a completely connected P
    for each state s **do**
        Set possible successor states: $S' = S$
        for each action a **do**
            Set $P(s, a) = s'$ sampled uniformly from S' and remove s' from S'

function INIT_REWARD_FUNCTION(n)
    Randomly sample $rd * \frac{|S|!}{(|S|-n)!}$ sequences and store in $rewardable\_sequences$

function REWARD_FUNCTION(s, a)
    $r = 0$
    if not $make\_denser$ **then**
        if state sequence of n states ending d steps in the past is in $rewardable\_sequences$ **then**
            $r = reward\_unit$
    else
        for i in range(n) **do**
            if sequence of i states ending d steps in the past is in sub-sequences of length i in $rewardable\_sequences$ **then**
                $r+ = reward\_unit$ * i/n
    $r+ = \mathcal{N}(0, \sigma^2{}_{r\_n})$
    return r

function TRANSITION_FUNCTION(s, a)
    $s' = P(s, a)$
    if $\mathcal{U}(0, 1) < t\_n$ **then**
        $s' =$ a random state in $S \setminus \{P(s, a)\}$
    return s'

INIT_TERMINAL_STATES() ▷ Set T according to $terminal\_state\_density$
INIT_INIT_STATE_DIST() ▷ Set ρ_o to uniform distribution over non-terminal states
INIT_TRANSITION_FUNCTION()
INIT_REWARD_FUNCTION()

For a user chosen $|S|$ and $|A|$, S and A contain categorical elements and we generate random instantiations of P and R for each instantiation of an environment. The generated P and R are deterministic unless we deliberately inject stochasticity through the respective meta-features we have created for them. We currently keep ρ_o to be uniform over the non-terminal states and T is fixed to be a subset of S based on the user chosen terminal state density.

Delayed Rewards We delay the reward for a state-action pair by a non-negative integer number of timesteps, which we call the *delay length*, d. In general, d will not be a constant in real world environments and would be a function of the state and action (sequence), but for our simple experiments here, we use a fixed d.

[3]The playground also supports some basic continuous environments. Expanded support is under development.

Specific Sequences We reward only specific sequence of states of positive integer length n. Like d, n would not be a constant in real world environments, but for our simple experiments here, we use a fixed n. For our experiments, we consider that specific sequences of *states* would be rewardable, though, in general, specific sequences of states *and* actions should be considered for rewards.[4]

Sparsity of Rewards We define the *reward density* rd of sequences in terms of the fraction of possible sequences of length n that are actually rewarded by the environment, for the specific case when the sequence length n is constant. There are $\frac{|S|!}{(|S|-n)!}$ possible specific sequences (when no state repeats along the sequence) for an environment with sequence length n and state space size $|S|$ and if num_r of them are rewarded we define the reward density to be $rd = num_r * \frac{(|S|-n)!}{|S|!}$ and sparsity as $1 - rd$.[5]

Stochasticity Stochastic environments are implemented by making P and R noisy. For discrete environments, for the randomly generated P we do this by taking a *transition noise* $t\_n \in [0, 1]$, and with probability t_n, we let the environment transition to a state that is not the *true* next state given by the generated P. For R, we take a *reward_noise* $\sigma_{r\_n} \in \mathbb{R}$ and add a normal random variable distributed according to $\mathcal{N}(0, {\sigma^2}_{r\_n})$ to the reward that would have been given out by the environment had there been no reward noise.

With regard to sparsity, recall the tennis serve again. The point received by serving an ace would be a sparse reward. We as humans know to reward ourselves for executing only a part of the sequence correctly. Rewards in continuous control tasks to reach a target point (e.g., in Mujoco (Todorov et al., 2012)), are usually dense (such as the negative squared distance from the target). This lets the algorithm obtain a dense signal in space to guide learning, and it is well known that it would be much harder for the algorithm to learn if it only received a single reward at the target point. The environments in MDP Playground have a configuration option, $make\_denser$, to allow this kind of reward shaping to make the reward denser and enable algorithms to learn faster. To achieve this, when $make\_denser$ is $True$, the environment gives a fractional reward if a fraction of a specific sequence is achieved. (Please refer Algorithm 1.)

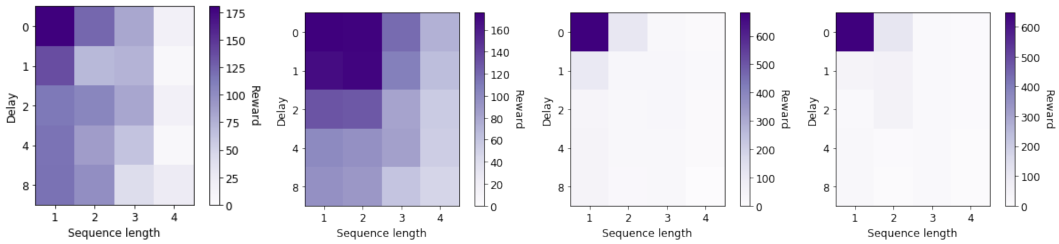
A parallel and independent work along similar lines as the MDP Playground, which was released last month, is the Behaviour Suite for RL (bsuite, Osband et al. (2019)). That suite collects simple RL benchmarks from the literature that are representative of various types of problems which occur in RL and tries to characterise RL algorithms. However, they do not employ orthogonal meta-features like we do and as a result, they do not have the same type of fine-grained control over their environments' difficulty, especially not along controllable dimensions. They also do not generate completely random P and R for their environments like we do, which would help avoid algorithms overfitting to benchmarks. Unlike their framework, where currently there's no toy environment for Hierarchical RL (HRL) algorithms, the specific sequences that we describe would also fit very well with HRL. An important distinction between the two platforms could be summed up by saying that they try to characterise algorithms while we try to characterise environments with the aim that new adaptable algorithms can be developed that can tackle environments of desired difficulty.

4 Experiments and Results

Experimental Setup We ran DQN (Mnih et al., 2013), Rainbow DQN (Hessel et al., 2017), A3C (Mnih et al., 2016), A3C with LSTM (all from the Ray RLLib (Liang et al., 2017) implementations) on grids of values for the meta-features discussed above. We fixed $|S|$ and $|A|$ to be 8, ρ_o to be uniformly random over non-terminal states, and the density of terminal states, equal to $|T|/|S|$, to be 0.25 for the experiments. The *reward unit* is fixed to be 1.0 whenever a reward is given by the environment. We generated random Ps that were *completely connected*, i.e., from each state there was a transition possible to every state in state space (including itself).

[4]Our implementation would be easily extendable to additionally consider actions for future experiments.

[5]For the general case where n is variable, it may be worthwhile to define reward density as the average reward a random agent receives in an environment, but for the fixed length case, it makes sense to implement and define it like we have done here.



(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 1: Mean episodic reward at the end of training for the different algorithms **when varying delay and sequence lengths**. Please note the different colorbar scales. We note that the intent of this figure is solely to show how the performance of an algorithm across meta-features gets worse for greater violations of Markovianness. It is not to compare DQN with A3C since the training procedures were different and we stopped training after different number of environment timesteps. What we tried to keep constant was the number of optimizer steps trained.

Results for varying reward delay and length of specific reward sequences We plot the average over 10 runs[6] of the final mean episodic reward[7] at the end of training for all the algorithms in Figure 1 for a grid of values over the *delay* and *specific sequences* meta-features. As can be seen from the figure, all algorithms perform very well in the vanilla environment where the MDP is completely observable because there is no delay and the sequence length is 1, but performance degrades in environments where the meta-features induce partial observability and hence make the state used by the algorithm non-Markovian. Performance clearly degrades more as we become more non-Markovian. It is interesting (and expected) that Rainbow DQN is more robust than DQN. However, it is unexpected that A3C with an LSTM does not improve over vanilla A3C (even though we set the LSTM max sequence length to the delay + sequence length which would let it *remember* the stack of states that would let the environment be modelled as a fully observable MDP); we plan to study this effect in more detail in the future.

We plot the standard deviation in the training of one of the algorithms (DQN) in Figure 2. The plot shows high variance in many of the environments with partial observability. Sometimes, DQN nevertheless managed to perform decently, which emphasizes that algorithms *can* sometimes perform well even when their assumption of complete observability is violated; this is one of the possible explanatory factors for the fact that *tuning* seeds can lead to good results.

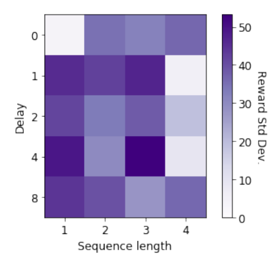


Figure 2: DQN training reward standard deviation across 10 runs.

We relegate plots of the evaluation reward at the end of the training[8] to the Appendix (Figure 9 in Appendix) since they are qualitatively similar to the training episodic rewards in Figure 1.

Results for varying transition and reward noise We see a similar trend, as for delays and sequences, when we vary the transition and reward noises in Figure 3. Performance degrades gradually as more and more noise is injected. It is interesting that DQN seems to be more sensitive to noise in the transition dynamics compared to the reward dynamics: transition noise values as low as 0.02 lead to a clear handicap in learning while for the reward dynamics (with the *reward unit* being 1.0) reward noise variances of $\sigma_r^2 = 1$ still resulted in decent learning.

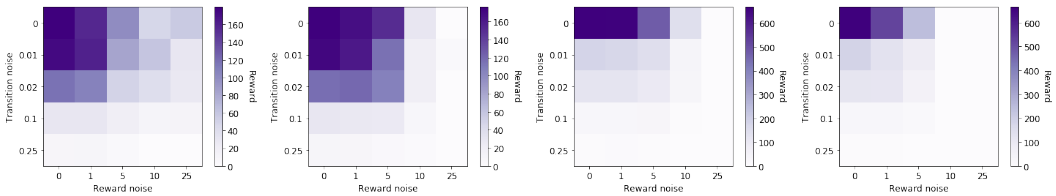
Next to the training performance in Figure 3, we also plot the evaluation performance in Figure 4.[9] Comparing the 2 figures shows that the training performance of the algorithms is more sensitive to noise in the transition dynamics than the eventual evaluation performance is. While it is obvious that the mean episodic reward during training would be perturbed when noise is injected into the

[6]over 10 random seeds for the algorithm but fixed seed for the environment

[7]over previous 100 episodes

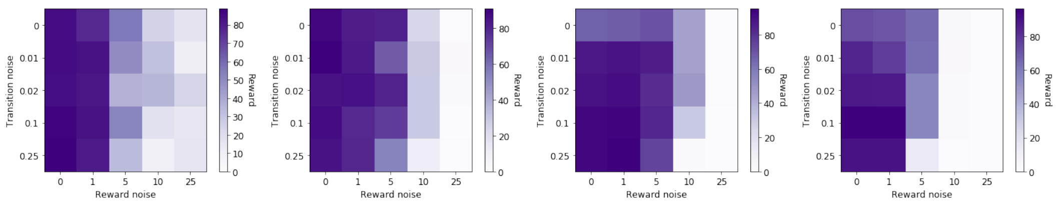
[8]rollout with the learnt policy, averaged over 10 episodes

[9]Here, for evaluation, and not for training because training is *in* the noisy environment, we evaluated in the corresponding environment without noise to assess how well the *true* learning is proceeding.



(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 3: Mean episodic reward at the end of training for the different algorithms **when varying transition noise and reward noise**. Please note the different colorbar scales.



(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 4: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying transition noise and reward noise**. Please note the different colorbar scales.

reward function, it is non-trivial that injecting noise into the *transition* function still leads to good learning (as displayed in the evaluation rollout plots). An additional seeming anomaly is that the evaluation rollouts for A3C suggest that it performs better in the presence of transition noise (when *reward noise* variance $\sigma^2_{r\_n}$ is 0 or 1); this might indicate that, with little reward noise which doesn't disrupt training too much, A3C in the presence of no transition noise does not explore enough and is actually helped when transition noise *is* present during training.

In addition to the plots for the rewards at the end of the training, we plot the complete learning curves for evaluation rollouts for DQN in the presence of injected transition and reward noises in Figure 5. (This means that each square in the heatmap in Figure 4a corresponds to the mean over the rightmost points in the corresponding evaluation learning curve plot in Figure 5). This underlines that throughout the trajectory, training is more robust to transition noise than to reward noise.

Results for sparsity The plots for controlling the meta-feature sparsity in the vanilla environment show that DQN variants are able to learn the important rewarding states in the vanilla environment even when these are sparse while the behaviour of A3C was once again somewhat unexpected (Figure 6). One explanation could be that A3C's exploration was not very good (as was also conjectured in the unexpected results for varying the noise meta-features), in which case increasing reward density would help as in 6c. But adding in an LSTM to the A3C agent seems to show the opposite trend as increasing reward density leads to worsening performance. This could mean that having a greater density of rewarding states makes it harder for the LSTM to remember one state to stick to. This behaviour of A3C warrants more investigation in the future.

The $make\_denser$ configuration option, where we make the environment give denser rewards by rewarding even when only a fraction of a rewardable sequence has been achieved, makes learning less variant across different runs of an algorithm although the algorithms still don't perform as well as they could (as would be expected when making rewards denser) in the evaluation rollouts where we turn off the option to evaluate *true* learning, probably due to the sequence lengths still *violating* the complete observability assumption made by the algorithm. The plots for learning curves for DQN are in Figure 7 and the rest are present in the Appendix (Figure 10 and Figures 26-32).

Hyperparameter Tuning Hyperparameters were tuned for the vanilla environment; we did so manually in order to obtain good intuition about them before applying automated tools. We tuned the hyperparameters in sets, loosely in order of their significance and did 3 runs over each setting to get a

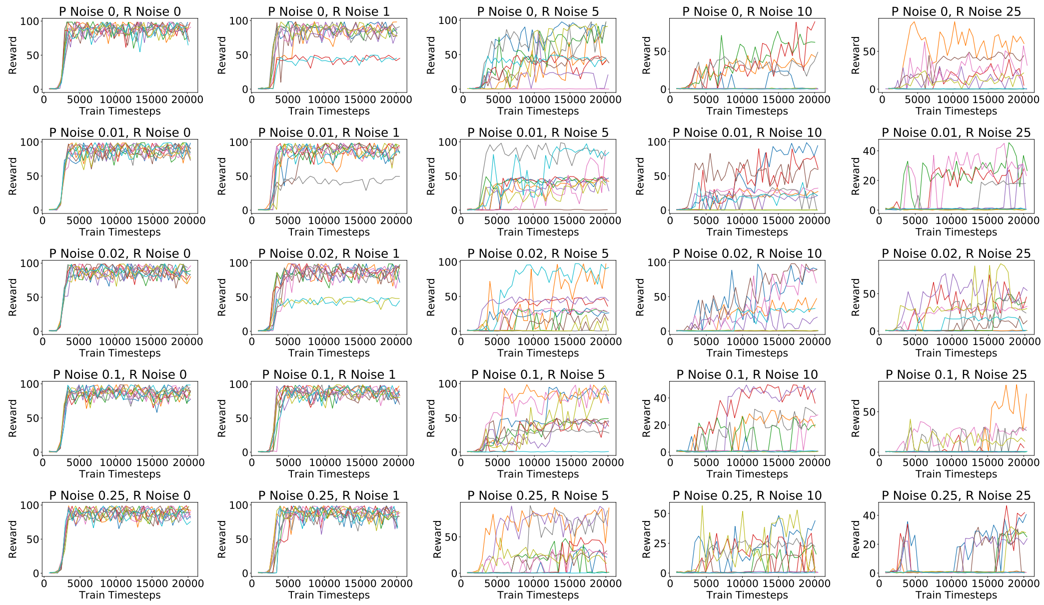
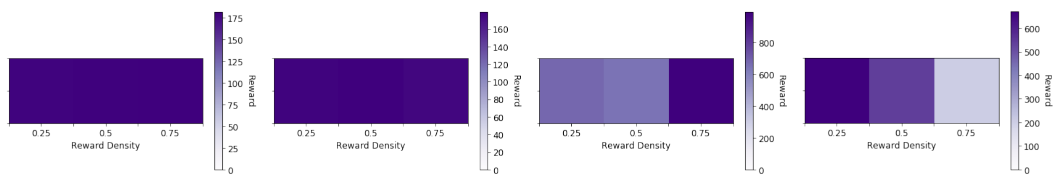


Figure 5: Evaluation Learning Curves for DQN **when varying transition noise and reward noise**. Please note the different Y-axis scales.



(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 6: Mean episodic reward at the end of training for the different algorithms **when varying reward sparsity**. Please note the different colorbar scales.

more robust performance estimate. Our prior was that for such toy environments we would not need much hyperparameter tuning, but it turned out that hyperparameter tuning was still very significant. Thus, our toy environments might in fact be good test beds for researching hyperparameters in RL, too. We describe a small part of our hyperparameter tuning for DQN next. All hyperparameter settings for tuned agents can be found in Appendix C.

We expected that quite small neural networks would already perform well for such toy environments and we initially grid searched over small network sizes (Figure 8a). However, the variance in performance was quite high (Figure 8b. When we tried to tune DQN hyperparameters *learning*

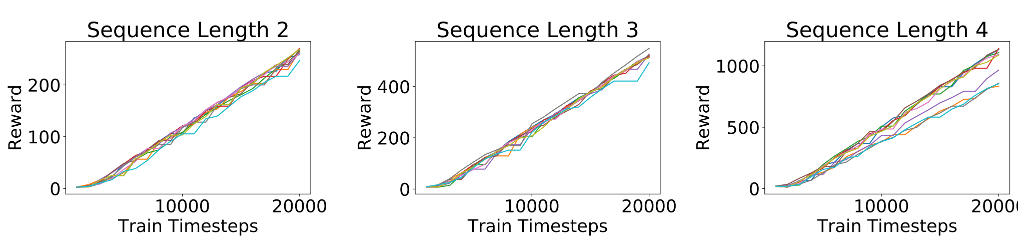
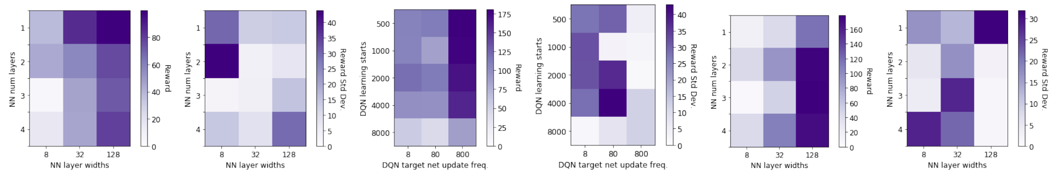


Figure 7: Training Learning Curves for DQN **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.



(a) Reward (b) Std dev. (c) Reward (d) Std. Dev. (e) Reward (f) Std dev.

Figure 8: Mean episodic reward at the end of training for different hyperparameter sets for DQN. Please note the different colorbar scales.

starts and *target network update frequency*, however, it became clear that the target network update frequency was very significant (Figure 8c and 8d) and when we repeated the grid search over network sizes with a better value of 800 for the target network update frequency (instead of the old 80) this led to both better performance and lower variance (Figure 8e and 8f).

5 Conclusion and Future Work

We introduced a low-cost[10] platform to test RL algorithms in environments with varying, controllable, key meta-features that we identified; we also evaluated some baseline RL algorithms using the platform. The platform allows us to disentangle various factors that make RL benchmarks hard by providing fine-grained control over various meta-feature dimensions. It therefore allows us to assess which characteristics make RL problems hard for different algorithms and would also allow us to evaluate future algorithms which may adapt to variations in these and other meta-features.

We will release all our code as Open Source to facilitate better, cheaper, more reproducible, and more directed benchmarking in the RL community.[11]

We will further implement plug and play model-based metrics to evaluate model-based algorithms, such as the Wasserstein metric (likely a sampled version because analytical calculation would be intractable in many cases) between the true dynamics models and the learnt one to keep track of how model learning is proceeding. Our Environments already allow using their transition and reward functions to perform *imaginary* rollouts without affecting the current state of the system.

We also have existing support for basic continuous environments and a toy task where we hand out greater rewards the closer a point object is to moving along a line. This is also a better task to test exploration than the completely random discrete environments. It was already giving some interesting results and further work will follow.[12]

Another significant meta-feature is *reachability* in the transition graph. We believe a lot of insights can be gained from graph theory to model toy environments which try to mimic specific real life situations at a very high level. Users can already specify their own transition graphs, but we plan to add random generation of specific types of graphs.

Even though we have a playground to generate environments where the meta-features such as sequence length are constant, being able to solve environments with variable delay and sequence lengths and identifying them (i.e., segmentation of events in the time domain) is another area we are currently working on with attention-based agents and various other ideas.

It would also be interesting to integrate our platform with the bsuite. Overall, we intend to promote more adaptivity in RL algorithms and we hope this platform is a first small step towards that.

[10] The runtimes depend a lot on the algorithm, network size and meta-features. But to give the reader an idea of the runtimes involved for our experiments, the DQN experiments (with a network with 2 hidden layers of 256 units each) in Figure 1a took about 3.5 hours, which equates to 1 minute for every single run of DQN for 20,000 environment steps.

[11] The code is currently available for anonymous review at `https://github.com/anonips/-MDP-Playground.git`.

[12] Since the platform has many new improvements planned, we expand further on some of the points mentioned here in some more detail in the Appendix (Section D).

Acknowledgments

The authors gratefully acknowledge support by BMBF grant DeToL, by the Bosch Center for Artificial Intelligence, and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant no. 716721. They would like to thank their group, especially André Biedenkapp, for helpful discussions. Raghu would like to additionally thank the RLSS 2019, Lille organizers and participants for a stimulating summer school.

References

Abbeel, P., Coates, A., and Ng, A. Y. (2010). Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639.

Arjona-Medina, J. A., Gillhofer, M., Widrich, M., Unterthiner, T., Brandstetter, J., and Hochreiter, S. (2018). Rudder: Return decomposition for delayed rewards.

Chua, K., Calandra, R., McAllister, R., and Levine, S. (2018). Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2017). Deep reinforcement learning that matters.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning.

Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I. (2017). Rllib: Abstractions for distributed reinforcement learning.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepezvari, C., Singh, S., Roy, B. V., Sutton, R., Silver, D., and Hasselt, H. V. (2019). Behaviour suite for reinforcement learning.

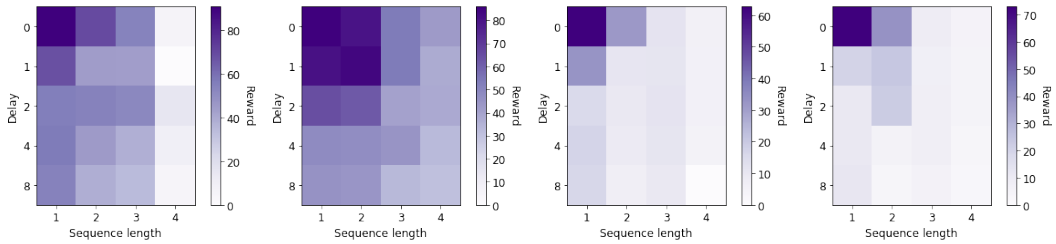
Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211.

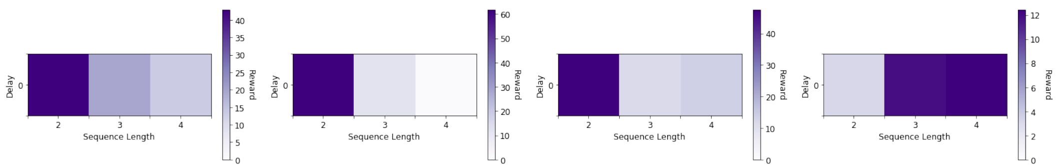
Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE.

A Additional Reward Plots



(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 9: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying delay and specific sequences**. Please note the different colorbar scales.



(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 10: Mean episodic evaluation rollout reward at the end of training for the different algorithms **when making reward for specific sequences denser**. Please note the different colorbar scales.

B Additional Learning Curves

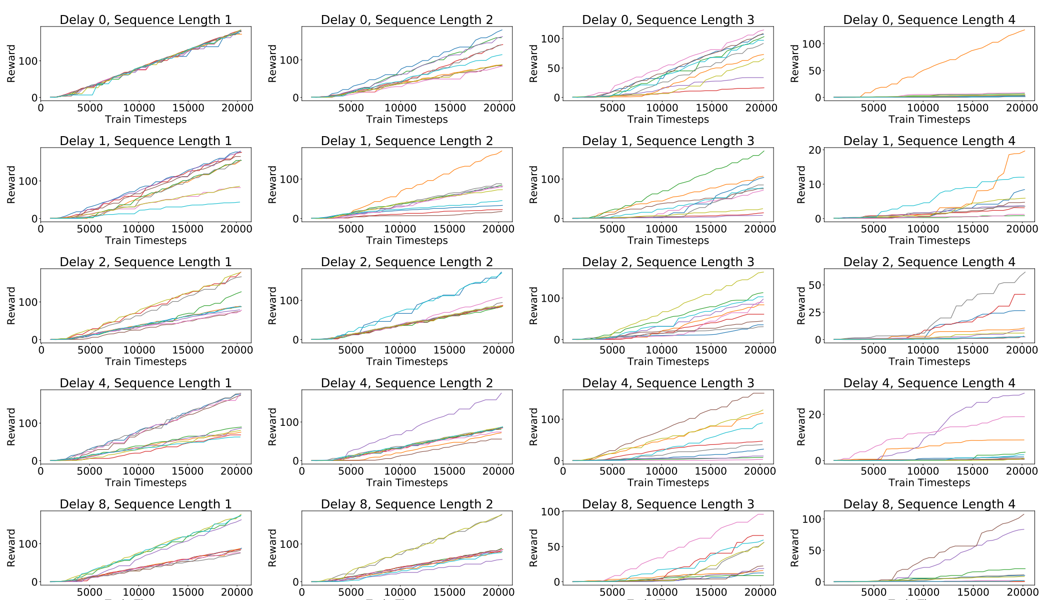


Figure 11: Training Learning Curves for DQN **when varying delay and specific sequences**. Please note the different colorbar scales.

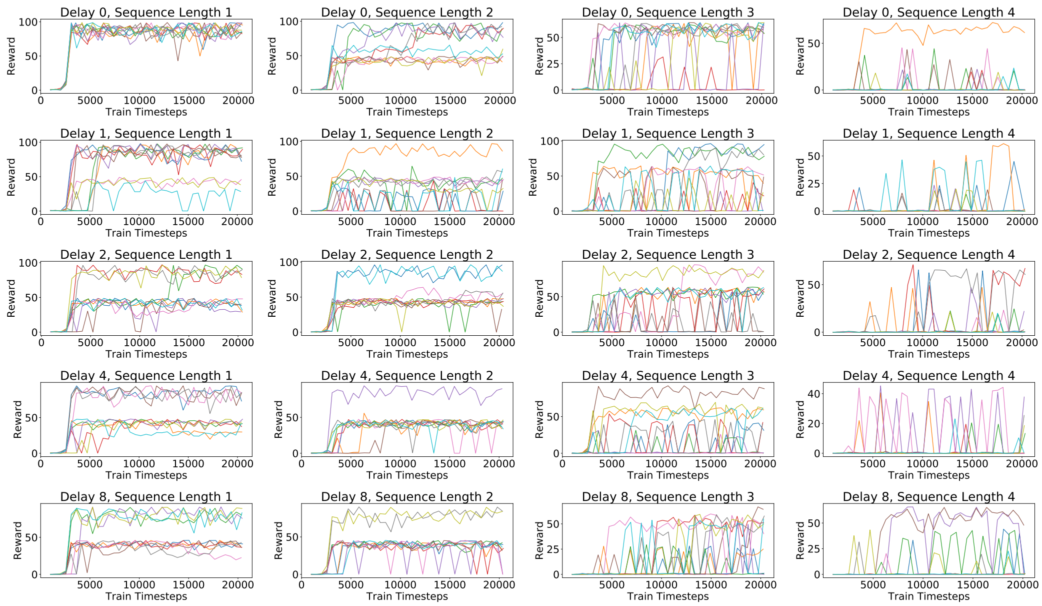


Figure 12: Evaluation Learning Curves for DQN **when varying delay and specific sequences**. Please note the different colorbar scales.

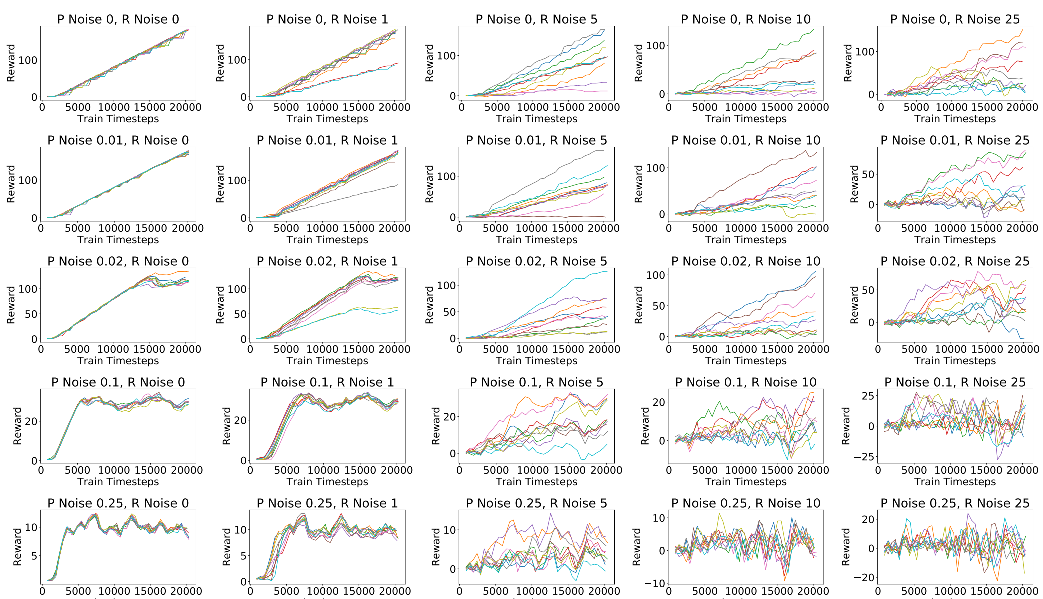


Figure 13: Training Learning Curves for DQN **when varying transition noise and reward noise**. Please note the different colorbar scales.

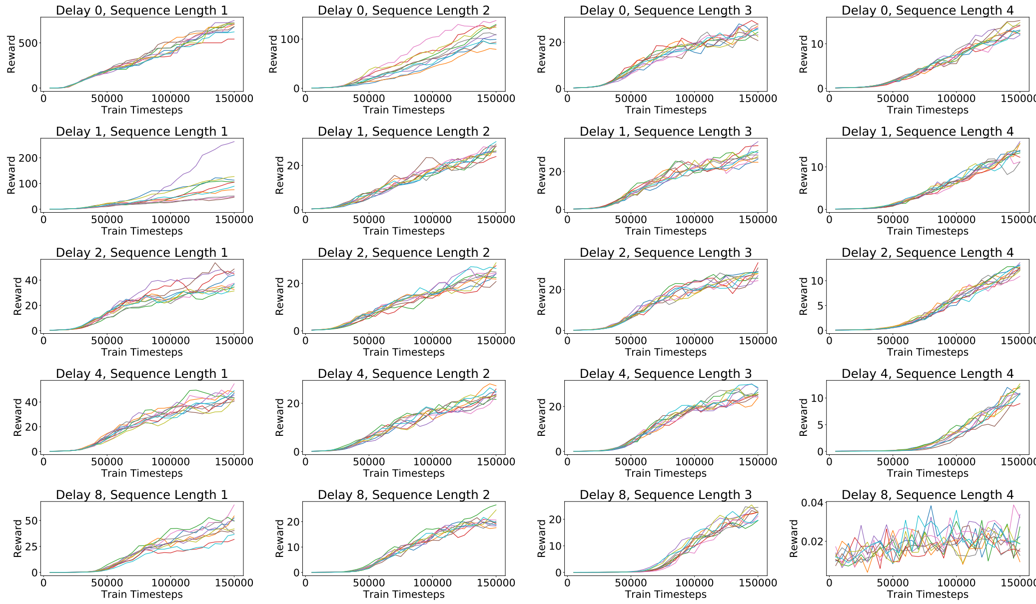


Figure 14: Training Learning Curves for A3C **when varying delay and specific sequences**. Please note the different Y-axis scales.

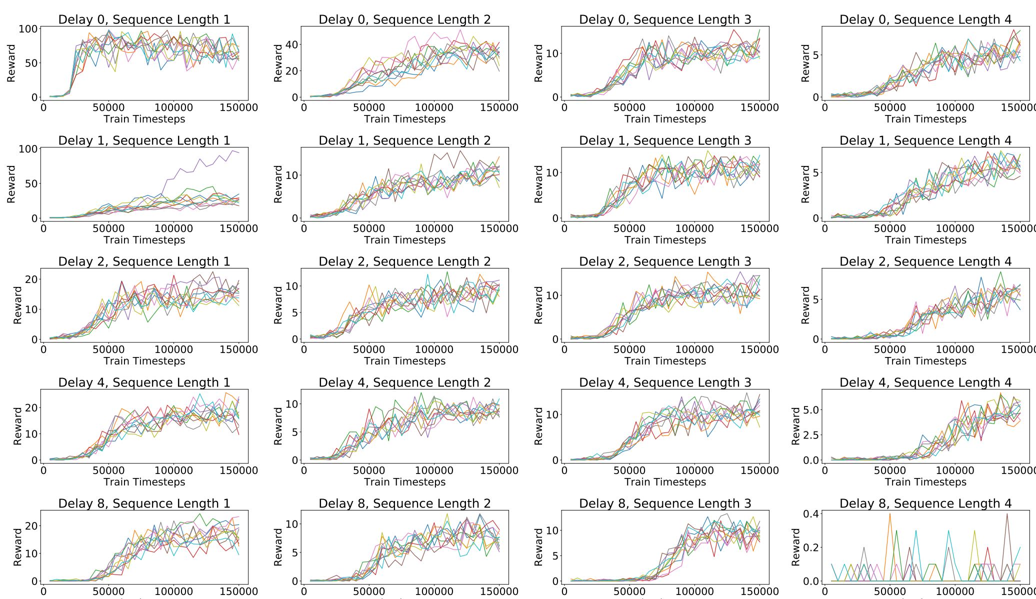


Figure 15: Evaluation Learning Curves for A3C **when varying delay and specific sequences**. Please note the different Y-axis scales.

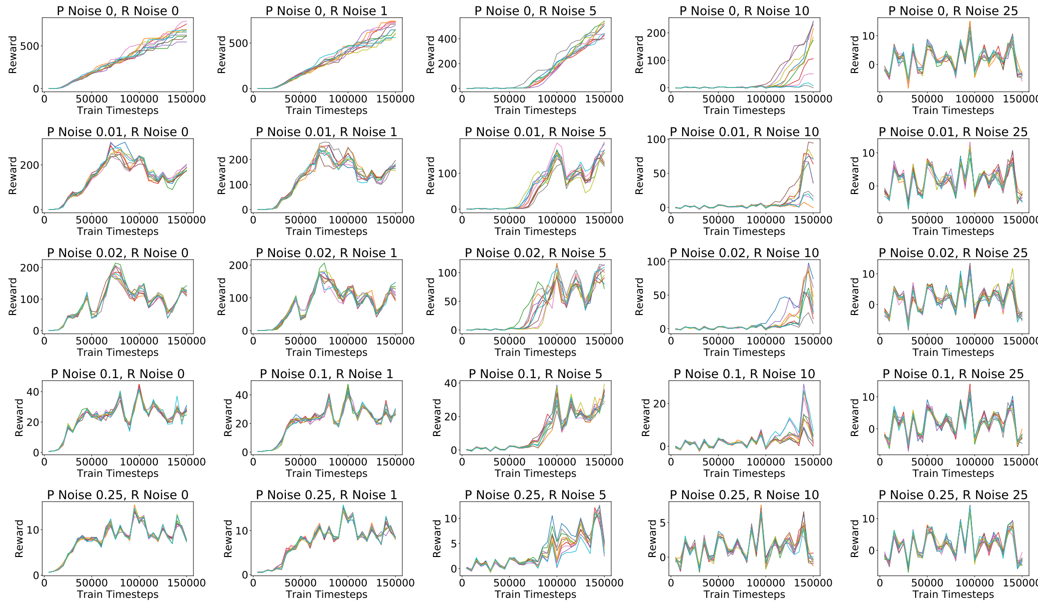


Figure 16: Training Learning Curves for A3C **when varying noises**. Please note the different Y-axis scales.

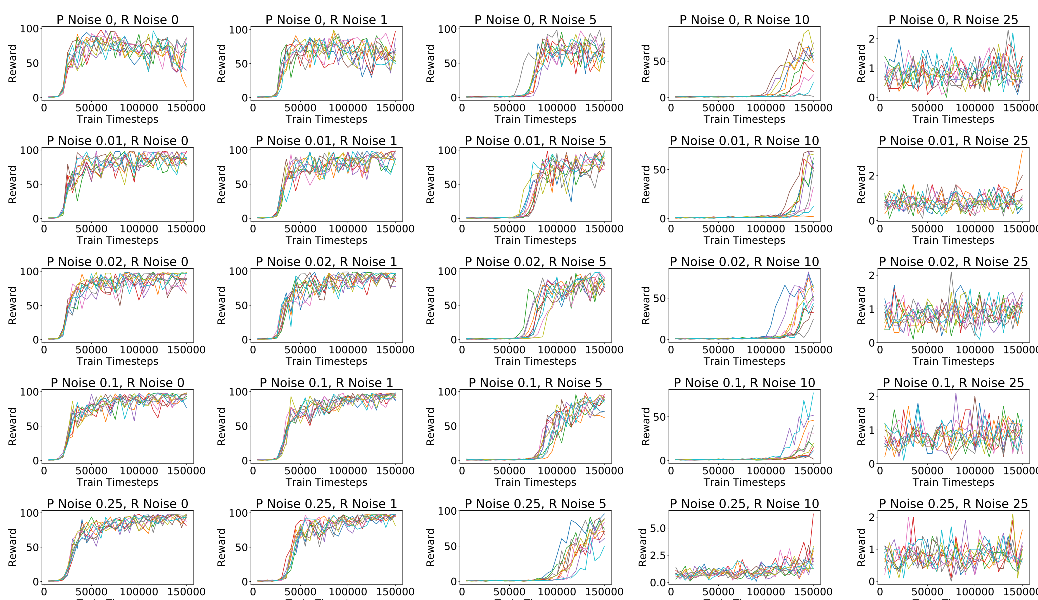


Figure 17: Evaluation Learning Curves for A3C **when varying noises**. Please note the different Y-axis scales.

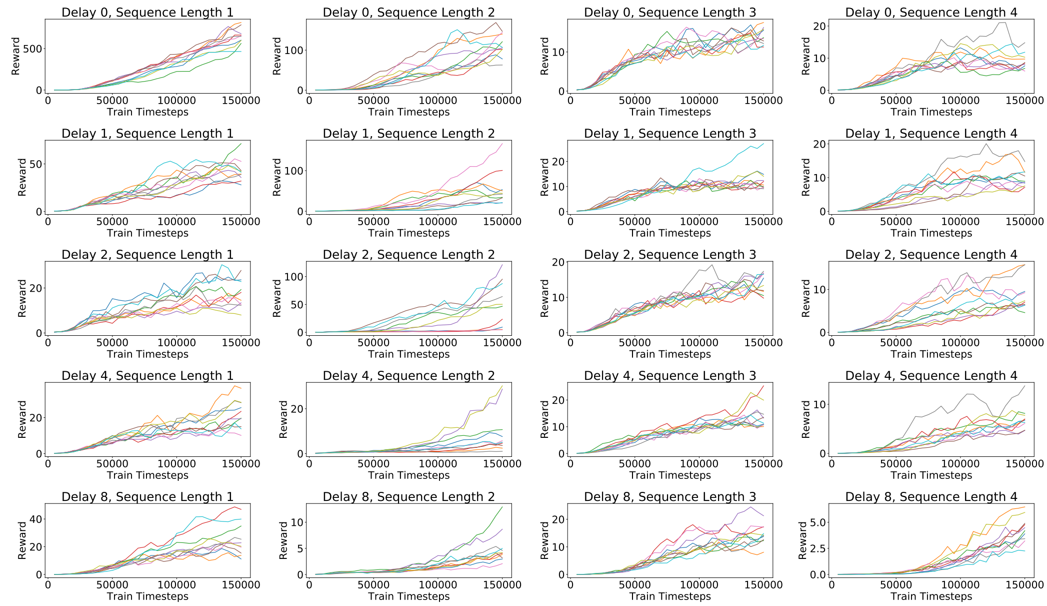


Figure 18: Training Learning Curves for A3C with LSTM **when varying delay and specific sequences**. Please note the different Y-axis scales.

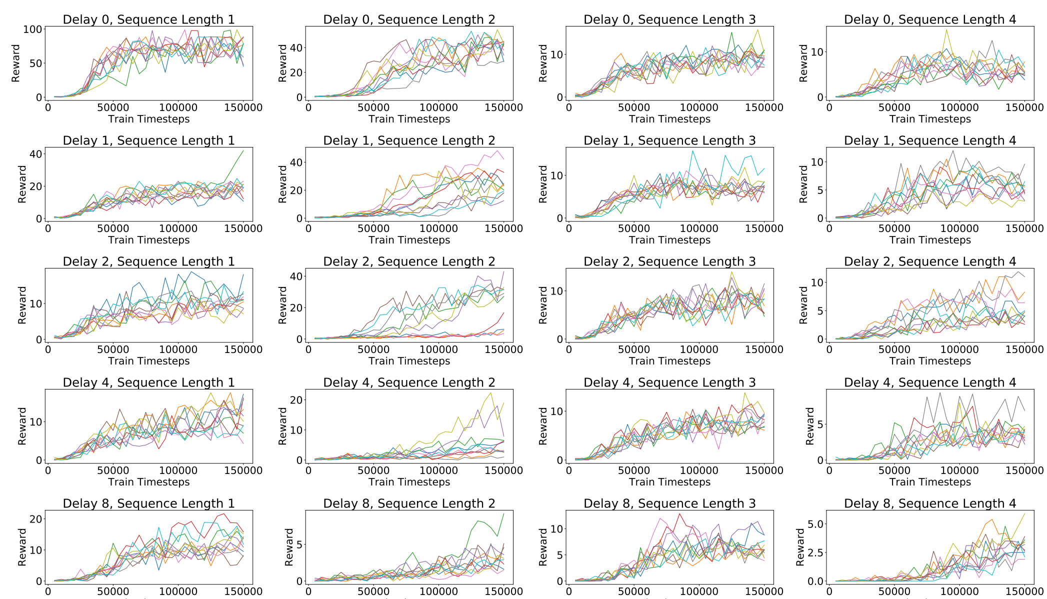


Figure 19: Evaluation Learning Curves for A3C with LSTM **when varying delay and specific sequences**. Please note the different Y-axis scales.

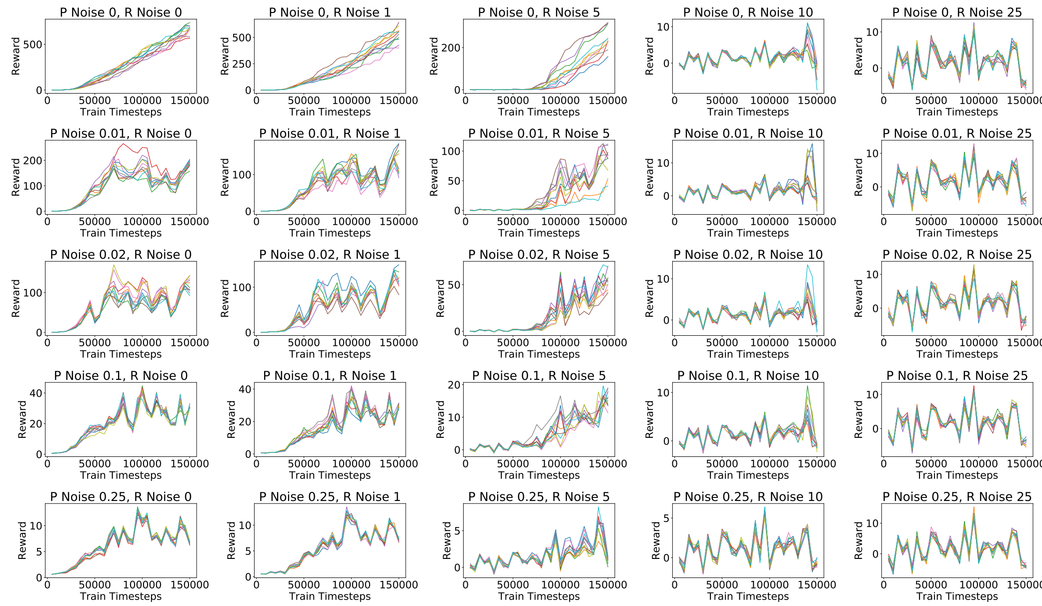


Figure 20: Training Learning Curves for A3C with LSTM **when varying noises**. Please note the different Y-axis scales.

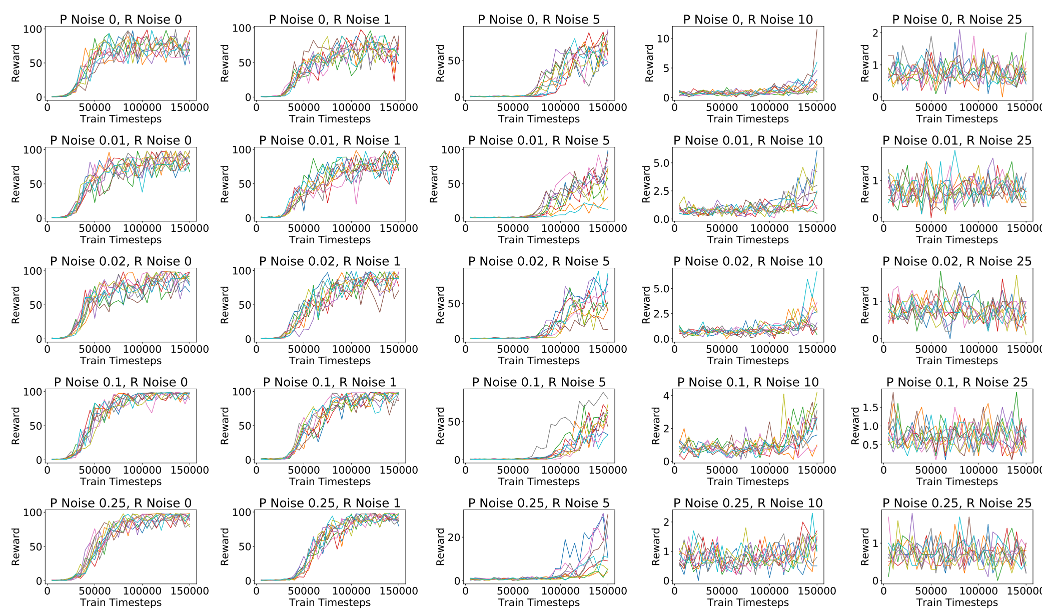


Figure 21: Evaluation Learning Curves for A3C with LSTM **when varying noises**. Please note the different Y-axis scales.

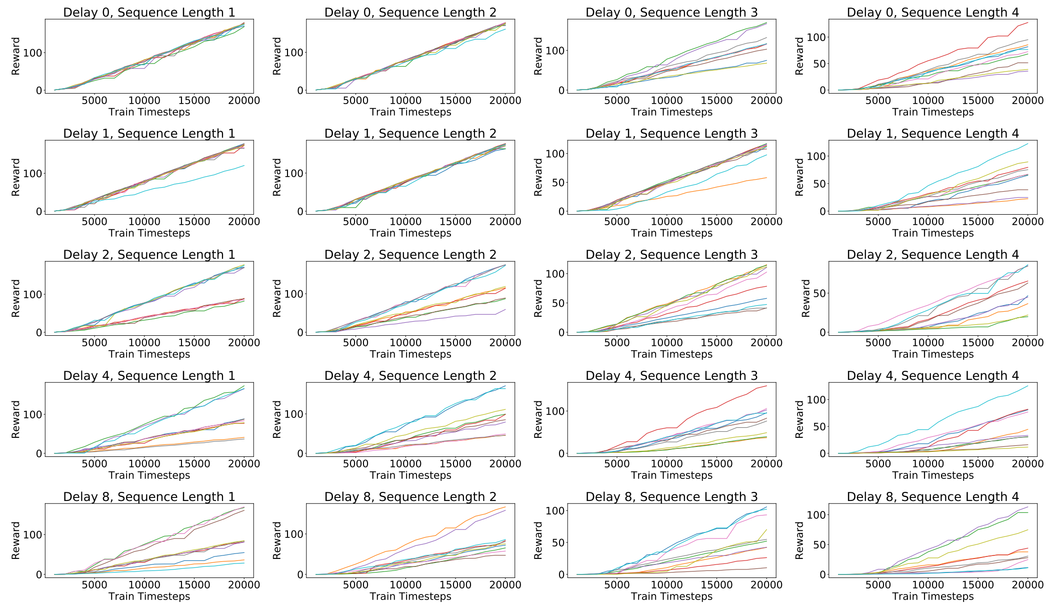


Figure 22: Training Learning Curves for Rainbow **when varying delay and specific sequences**. Please note the different Y-axis scales.

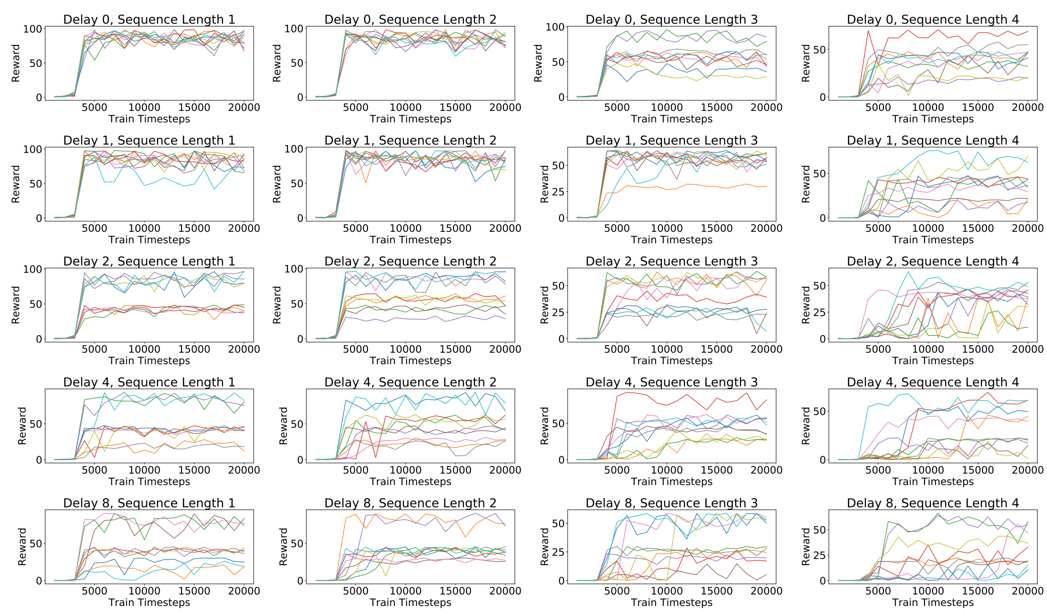


Figure 23: Evaluation Learning Curves for Rainbow **when varying delay and specific sequences**. Please note the different Y-axis scales.

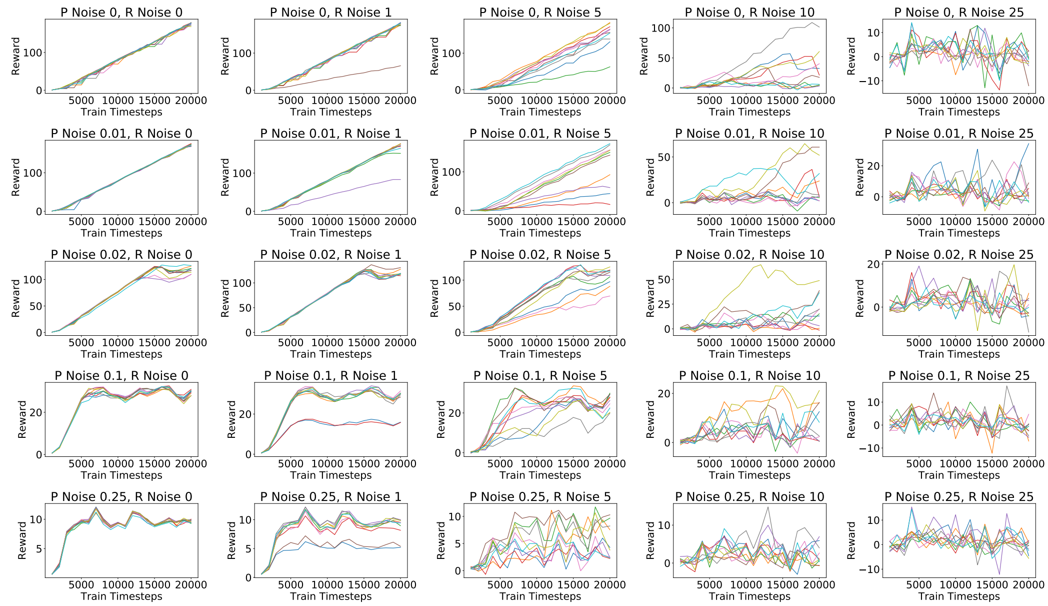


Figure 24: Training Learning Curves for Rainbow **when varying noises**. Please note the different Y-axis scales.

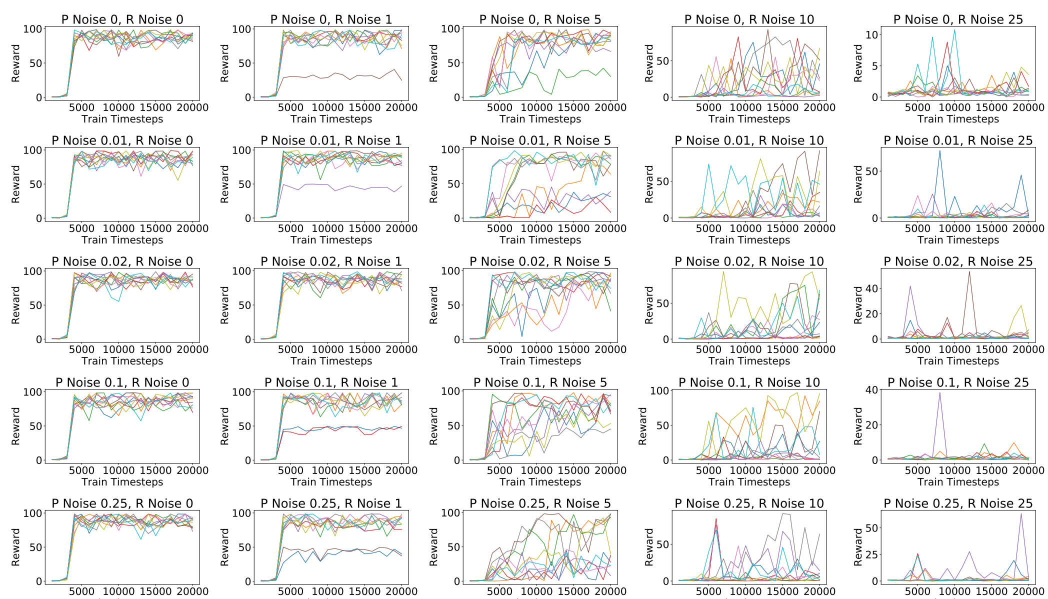


Figure 25: Evaluation Learning Curves for Rainbow **when varying noises**. Please note the different Y-axis scales.

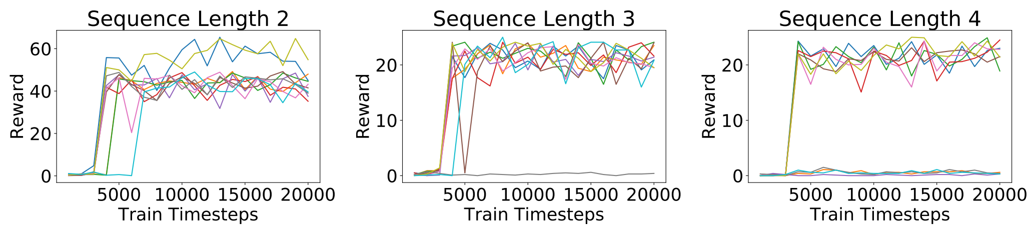


Figure 26: Evaluation Learning Curves for DQN **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.

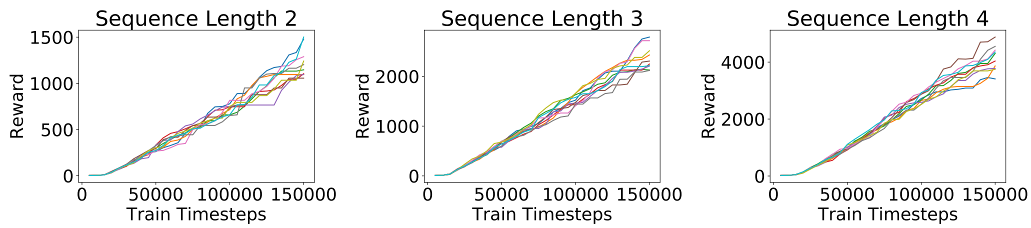


Figure 27: Training Learning Curves for A3C **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.

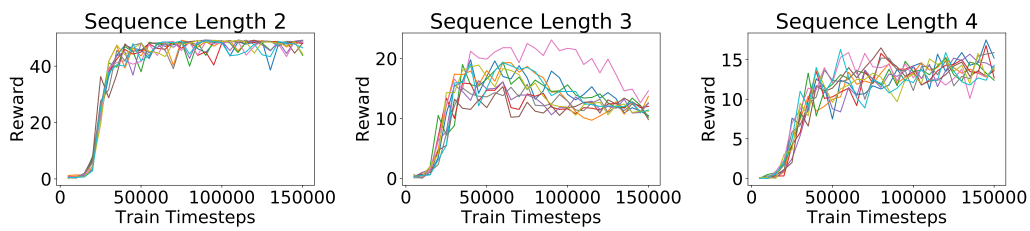


Figure 28: Evaluation Learning Curves for A3C **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.

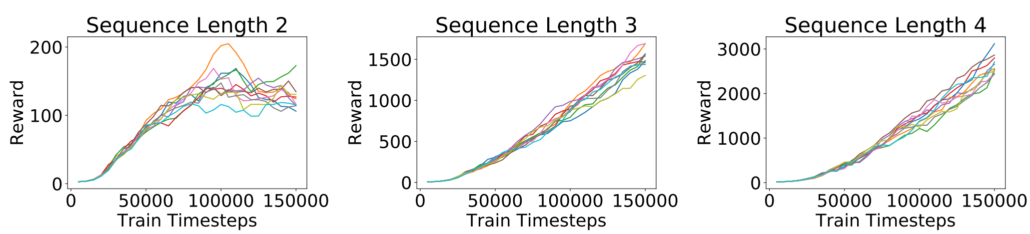


Figure 29: Training Learning Curves for A3C + LSTM **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.

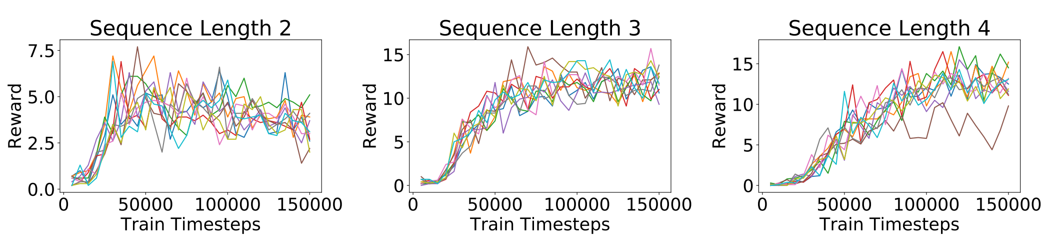


Figure 30: Evaluation Learning Curves for A3C + LSTM **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.

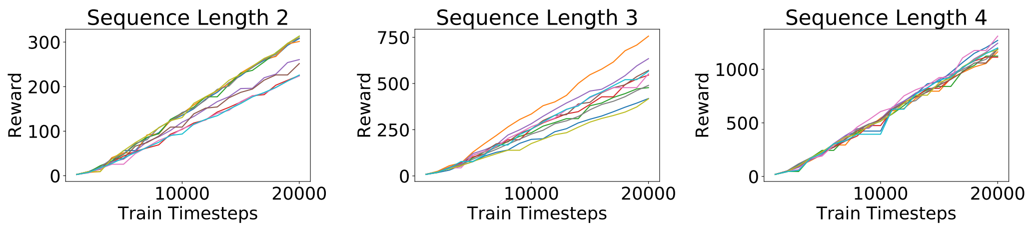


Figure 31: Training Learning Curves for Rainbow **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.

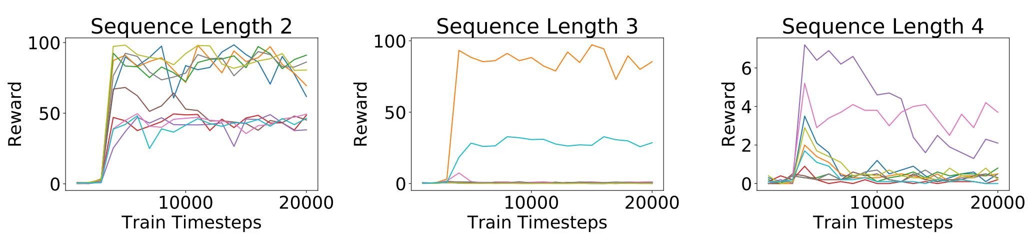


Figure 32: Evaluation Learning Curves for Rainbow **when *make_denser* is *True* for specific sequences**. Please note the different Y-axis scales.

C Tuned Hyperparameters

The following contain the hyperparameter and environment meta-feature settings that we ran **Ray tune** with. The names of the hyperparameters for the algorithms will match those used in Ray 0.7.3. These configurations can be inserted into `run_experiments.py` in the git repository to rerun the experiments from the paper.

C.1 DQN

```
tune.run(
    "DQN",
    stop={
        "timesteps_total": 20000,
          },
    config={
      "adam_epsilon": 1e-4,
      "beta_annealing_fraction": 1.0,
      "buffer_size": 1000000,
      "double_q": False,
      "dueling": False,
      "exploration_final_eps": 0.01,
      "exploration_fraction": 0.1,
      "final_prioritized_replay_beta": 1.0,
      "hiddens": None,
      "learning_starts": 1000,
      "lr": 1e-4,
      "n_step": 1,
      "noisy": False,
      "num_atoms": 1,
      "prioritized_replay": False,
      "prioritized_replay_alpha": 0.5,
      "sample_batch_size": 4,
      "schedule_max_timesteps": 20000,
      "target_network_update_freq": 800,
      "timesteps_per_iteration": 100,
      "train_batch_size": 32,

      "env": "RLToy-v0",
      "env_config": {
        'dummy_seed': dummy_seed,
        'seed': 0,
        'state_space_type': 'discrete',
        'action_space_type': 'discrete',
        'state_space_size': state_space_size,
        'action_space_size': action_space_size,
        'generate_random_mdp': True,
        'delay': delay,
        'sequence_length': sequence_length,
        'reward_density': reward_density,
        'terminal_state_density': terminal_state_density,
        'repeats_in_sequences': False,
        'reward_unit': 1.0,
        'make_denser': False,
        'completely_connected': True
        },
    "model": {
        "fcnet_hiddens": [256, 256],
        "custom_preprocessor": "ohe",
        "custom_options": {},
```

```
        "fcnet_activation": "tanh",
        "use_lstm": False,
        "max_seq_len": 20,
        "lstm_cell_size": 256,
        "lstm_use_prev_action_reward": False,
        },

              "callbacks": {
                "on_episode_end": tune.function(on_episode_end),
                "on_train_result": tune.function(on_train_result),
            },
        "evaluation_interval": 1,
        "evaluation_config": {
        "exploration_fraction": 0,
        "exploration_final_eps": 0,
        "batch_mode": "complete_episodes",
        'horizon': 100,
          "env_config": {
            "dummy_eval": True,
            }
    },
    },
    )
```

C.2 Rainbow

```
tune.run(
    "DQN",
    stop={
        "timesteps_total": 20000,
          },
    config={
      "adam_epsilon": 1e-4,
      "buffer_size": 1000000,
      "double_q": True,
      "dueling": True,
      "lr": 1e-3,
      "exploration_final_eps": 0.01,
      "exploration_fraction": 0.1,
      "schedule_max_timesteps": 20000,
      "learning_starts": 500,
      "target_network_update_freq": 80,
      "n_step": 4,
      "noisy": True,
      "num_atoms": 10,
      "prioritized_replay": True,
      "prioritized_replay_alpha": 0.75,
      "prioritized_replay_beta": 0.4,
      "final_prioritized_replay_beta": 1.0,
      "beta_annealing_fraction": 1.0,

      "sample_batch_size": 4,
      "timesteps_per_iteration": 1000,
      "train_batch_size": 32,
      "min_iter_time_s": 1,

      "env": "RLToy-v0",
      "env_config": {
        'dummy_seed': dummy_seed,
```

```
        'seed': 0,
        'state_space_type': 'discrete',
        'action_space_type': 'discrete',
        'state_space_size': state_space_size,
        'action_space_size': action_space_size,
        'generate_random_mdp': True,
        'delay': delay,
        'sequence_length': sequence_length,
        'reward_density': reward_density,
        'terminal_state_density': terminal_state_density,
        'repeats_in_sequences': False,
        'reward_unit': 1.0,
        'make_denser': False,
        'completely_connected': True
        },
    "model": {
        "fcnet_hiddens": [256, 256],
        "custom_preprocessor": "ohe",
        "custom_options": {},
        "fcnet_activation": "tanh",
        "use_lstm": False,
        "max_seq_len": 20,
        "lstm_cell_size": 256,
        "lstm_use_prev_action_reward": False,
        },
              "callbacks": {
                "on_episode_end": tune.function(on_episode_end),
                "on_train_result": tune.function(on_train_result),
            },
            "evaluation_interval": 1,
            "evaluation_config": {
            "exploration_fraction": 0,
            "exploration_final_eps": 0,
            "batch_mode": "complete_episodes",
            'horizon': 100,
              "env_config": {
                "dummy_eval": True,
                }
            },
    },
 )
```

C.3 A3C

```
tune.run(
    "A3C",
    stop={
        "timesteps_total": 150000,
          },
    config={
            "sample_batch_size": 10,
            "train_batch_size": 100,
            "use_pytorch": False,
            "lambda": 0.0,
            "grad_clip": 10.0,
            "lr": 0.0001,
            "lr_schedule": None,
            "vf_loss_coeff": 0.5,
            "entropy_coeff": 0.1,
```

```
            "min_iter_time_s": 0,
            "sample_async": True,
            "timesteps_per_iteration": 5000,
            "num_workers": 3,
            "num_envs_per_worker": 5,

            "optimizer": {
                "grads_per_step": 10
            },

      "env": "RLToy-v0",
      "env_config": {
        'dummy_seed': dummy_seed,
        'seed': 0,
        'state_space_type': 'discrete',
        'action_space_type': 'discrete',
        'state_space_size': state_space_size,
        'action_space_size': action_space_size,
        'generate_random_mdp': True,
        'delay': delay,
        'sequence_length': sequence_length,
        'reward_density': reward_density,
        'terminal_state_density': terminal_state_density,
        'repeats_in_sequences': False,
        'reward_unit': 1.0,
        'make_denser': False,
        'completely_connected': True
        },
    "model": {
        "fcnet_hiddens": [128, 128, 128],
        "custom_preprocessor": "ohe",
        "custom_options": {},
        "fcnet_activation": "tanh",
        "use_lstm": False,
        "max_seq_len": 20,
        "lstm_cell_size": 256,
        "lstm_use_prev_action_reward": False,
        },

              "callbacks": {
                "on_episode_end": tune.function(on_episode_end),
                "on_train_result": tune.function(on_train_result),
            },
        "evaluation_interval": 1,
        "evaluation_config": {
        "exploration_fraction": 0,
        "exploration_final_eps": 0,
        "batch_mode": "complete_episodes",
        'horizon': 100,
          "env_config": {
            "dummy_eval": True,
            }
    },
    },
)
```

C.4 A3C + LSTM

```
tune.run(
```

```
"A3C",
stop={
    "timesteps_total": 150000,
      },
config={
        "sample_batch_size": 10,
        "train_batch_size": 100,
        "use_pytorch": False,
        "lambda": 0.0,
        "grad_clip": 10.0,
        "lr": 0.0001,
        "lr_schedule": None,
        "vf_loss_coeff": 0.1,
        "entropy_coeff": 0.1,
        "min_iter_time_s": 0,
        "sample_async": True,
        "timesteps_per_iteration": 5000,
        "num_workers": 3,
        "num_envs_per_worker": 5,

        "optimizer": {
            "grads_per_step": 10
        },


  "env": "RLToy-v0",
  "env_config": {
    'dummy_seed': dummy_seed,
    'seed': 0,
    'state_space_type': 'discrete',
    'action_space_type': 'discrete',
    'state_space_size': state_space_size,
    'action_space_size': action_space_size,
    'generate_random_mdp': True,
    'delay': delay,
    'sequence_length': sequence_length,
    'reward_density': reward_density,
    'terminal_state_density': terminal_state_density,
    'repeats_in_sequences': False,
    'reward_unit': 1.0,
    'make_denser': False,
    'completely_connected': True
    },
"model": {
    "fcnet_hiddens": [128, 128, 128],
    "custom_preprocessor": "ohe",
    "custom_options": {},
    "fcnet_activation": "tanh",
    "use_lstm": True,
    "max_seq_len": delay + sequence_length,
    "lstm_cell_size": 64,
    "lstm_use_prev_action_reward": True,
    },

          "callbacks": {
            "on_episode_end": tune.function(on_episode_end),
            "on_train_result": tune.function(on_train_result),
        },
    "evaluation_interval": 1,
```

```
        "evaluation_config": {
        "exploration_fraction": 0,
        "exploration_final_eps": 0,
        "batch_mode": "complete_episodes",
        'horizon': 100,
          "env_config": {
            "dummy_eval": True,
            }
    },
    },
)
```

D More on Conclusion and Future Work

Further interesting toy experiments which are already possible with our platform are varying the terminal state densities to have environments for benchmarking *safe RL*.

Another key meta-feature, or rather meta-feature related to specific sequences, is manifolds. Let's say every specific sequence that was rewarded had a length n and the reward density was, say, 0.1. If we had no prior knowledge of the environment and did completely random exploration, we would need O($n^{|S|}$) sequences to obtain a reward signal which is exponential in the state space size (if we ignore the action space). We need to have some prior knowledge about the manifolds that exist in the environment to explore in a directed manner. This we intend to infuse through having a model of the environment. We will initially create simple manifolds for different tasks, such as rewarding following a circle in a 3-D continuous space and test directed exploration strategies to see how these choose to explore.

The states and actions contained in a specific sequence could just be a single *compound* state and *compound* action if we discretised time in a suitable manner. This brings us to the idea of learning at multiple timescales. HRL algorithms with formulations like the options framework (Sutton et al., 1999), could try to identify these specific sequences at the higher level and then carry out "atomic" actions at the lower level.

We also hope to benchmark other algorithms like PPO[13] (Schulman et al., 2017), Rudder (Arjona-Medina et al., 2018), MCTS (Silver et al., 2016), DDPG[14] (Lillicrap et al., 2015) on continuous tasks and table-based algorithms and to show theoretical results match with practice on toy benchmarks.

We also aim to promote reproducibility in RL as in (Henderson et al., 2017) and hope our benchmark helps with that goal. To this end, we have already improved the Gym `Box` and `Discrete` `Spaces` to allow them to be seeded as well.

We need different RL algorithms for different environments. Aside from some basic heuristics such as applying DDPG (Lillicrap et al., 2015) to continuous environments and DQN to discrete environments, it is not very clear when to use which RL algorithms. We hope this will be a first step to being able to identify from the environment what sort of algorithm to use and to help build adaptive algorithms which adapt to the environment at hand. Additionally, aside from being a great benchmark for RL algorithms, it is also a great didactic tool for teaching how RL algorithms work in different environments.

[13] We tried PPO but could not get it to learn

[14] We tried DDPG also but there seemed to be a bug in the implementation and it crashed even on tuned examples from Ray