MDP Playground: A Design and Debug Testbed for Reinforcement Learning

Raghu Rajan[1], **Jessica Lizeth Borja Diaz**[1], **Suresh Guttikonda**[1], **Fabio Ferreira**[1], **André Biedenkapp**[1], **Jan Ole von Hartz**[1] **& Frank Hutter**[1,2]
[1] University of Freiburg [2] Bosch Center for Artificial Intelligence
rajanr@cs.uni-freiburg.de

Abstract

We present *MDP Playground*, an efficient testbed for Reinforcement Learning (RL) agents with *orthogonal* dimensions that can be controlled independently to challenge agents in different ways and obtain varying degrees of hardness in generated environments. We consider and allow control over a wide variety of dimensions, including *delayed rewards*, *rewardable sequences*, *density of rewards*, *stochasticity*, *image representations*, *irrelevant features*, *time unit*, *action range* and more. We define a parameterised collection of fast-to-run toy environments in *OpenAI Gym* by varying these dimensions and propose to use these for the initial design and development of agents. We also provide wrappers that inject these dimensions into complex environments from *Atari* and *Mujoco* to allow for evaluating agent robustness. We further provide various example use-cases and instructions on how to use *MDP Playground* to design and debug agents. We believe that *MDP Playground* is a valuable testbed for researchers designing new, adaptive and intelligent RL agents and those wanting to unit test their agents.

1 Introduction

RL has succeeded at many disparate tasks, such as helicopter aerobatics, game-playing and continuous control [2, 38, 49, 10, 14, 17]. However, a lot of the insights obtained are on very complex and in many instances *blackbox* environments.

There are many different types of standard environments, as many as there are different kinds of tasks in RL [e.g. 57, 6, 11]. They specialise in *specific* kinds of tasks. The underlying assumptions in many of these environments are that of a Markov Decision Process (MDP) [see, e.g., 44, 52] or a Partially Observable MDP (POMDP) [see, e.g., 22, 25]. However, there is a lack of simple and *general* MDPs which capture common difficulties seen in RL and let researchers experiment with them in a fine-grained manner. Many researchers design their own toy problems which capture the key aspect of their problem and then try to gain *whitebox* insights because the standard complex environments, such as *Atari* and *Mujoco*, are too expensive or too opaque for the initial design and development of their agent. To standardise this initial design and debug phase of the development pipeline, we propose a platform which *distils* difficulties for MDPs that can be generalised across RL problems and allows to *independently* inject these difficulties.

Disadvantages of *complex* environments when considered from a point of view of a design and debug testbed include: **1)** They are very expensive to evaluate. For example, a DQN [38] run on *Atari* [6] took us 4 CPU days and 64GB of memory to run. **2)** The environment structure itself is so complex that it leads to "lucky" agents performing better (e.g., in [18]). Furthermore, different implementations even using the same libraries can lead to very different results [18]. **3)** Many difficulties are concurrently present in the environments and do not allow us to independently test

Submitted to the 35th Conference on Neural Information Processing Systems (NeurIPS 2021) Track on Datasets and Benchmarks. Do not distribute.

their impact on agents' performance. During the design phase, we need environments to encapsulate, preferably orthogonally, the different difficulties present. For instance, MNIST [32] captured some key difficulties required for computer vision (CV) which made it a good testbed for designing and debugging CV algorithms, even though it cannot be used to directly learn models for much more specific CV applications such as classification of plants or medical image analysis.

The main contributions of this paper are:

- We identify and discuss dimensions of MDPs that can have a significant effect on agent performance, both for discrete and continuous environments;
- We discuss how to use *MDP Playground* to design and debug agents with various experiments; toy experiments can be run in as few as 30 seconds on a single core of a laptop;
- We discuss insights that can be gained with the various considered dimensions; transferring insights from toy to complex environments for some under-studied dimensions led to significant improvements in performances on complex environments.

2 Dimensions of MDPs

We try to exhaustively identify orthogonal *dimensions* of hardness in RL by going over the many components of a (PO)MDP. By *orthogonal*, we mean that these dimensions are present independent of each other in environments. This was tried *exhaustively* to allow as many dimensions as possible for researchers to systematically study them and gain new insights.

We define an MDP as a 7-tuple $(S, A, P, R, \rho_o, \gamma, T)$, where S is the set of states, A is the set of actions, $P : S \times A \rightarrow S$ describes the transition dynamics, $R : S \times A \times S \rightarrow \mathbb{R}$ describes the reward dynamics, $\rho_o : S \rightarrow \mathbb{R}^+$ is the initial state distribution, γ is the discount factor and T is the set of terminal states. We define a POMDP with two additional components - O represents the set of observations and $\Omega : S \times A \times O \rightarrow \mathbb{R}^+$ describes the probability density function of an observation given a state and action. To clarify terminology, following [51] we will use *information state* to mean the state representation used by the agent and *belief state* as the posterior belief of the unobserved state given the full observation history. If the belief state were to be used as the information state by an agent, this would be sufficient to compute an optimal policy. However, since the full observation history is not tractable to store for many environments, agents in practice use the last few observations as their information state which renders it only partially observable. This is important because many of the motivated dimensions are actually due to the information state being non-Markov.

2.1 MDPs in MDP Playground

Toy Environments The toy environments are cheap and encapsulate all the identified dimensions. The components of the MDP can be automatically generated according to the dimensions or can be user-defined. Any dimension not specified is set to a vanilla default value. Further, the underlying MDP state is exposed in an *augmented_state* variable, which allows users to design agents that may try to identify the true underlying MDP state given the observations. We now briefly describe the auto-generated discrete and continuous environments, since we use these for the experiments section and expect that these will cover the majority of the use-cases. This is followed by implementation details of selected dimensions; details for all dimensions can be found in Algorithm 1 in Appendix C.

Discrete Environments In the discrete case, S and A contain *categorical* elements, and random instantiations of P and R are generated after the remaining dimensions have been set. The generated P and R are deterministic and held fixed for the environment. We keep ρ_o to be uniform over the non-terminal states, and T is fixed to be a subset of S based on a chosen *terminal state density*.

Continuous Environments In the continuous case, environments correspond to the simplest real world task we could find: moving a rigid body to a target point, similar to [16] and [28]. P is formulated such that each action dimension affects the corresponding space dimension - s is set to be equal to the action applied for *time unit* seconds on a rigid body. This is integrated over time to yield the next state. R is designed such that the reward for the current time step is the distance travelled towards the target since the last step.

Both, the discrete and continuous environments, in *MDP Playground* can be described as graphical POMDPs.

2.2 Motivations of Dimensions and Implementations

We now describe many of the dimensions from a general point of view and their implementations in *MDP Playground*. For clarity, we describe only the dimensions with experiments in the main paper here in greater detail and refer the reader to Appendix B and the documentation for more detailed descriptions of all the dimensions.

Reward Delay For many environments, in many situations, agents perform an action that is consequential to receiving a reward but the agent is only rewarded in a *delayed* manner [see e.g. 4] (see Figure 1d). For example, shooting at an enemy ship in *Space Invaders* leads to rewards much later than the action of shooting. Any action taken after that is inconsequential to obtaining the reward for destroying that enemy ship. In *MDP Playground*, the reward is artificially delayed by a non-negative integer number of timesteps, d.

Reward Density Environments can also be characterised by their *reward density*. When an environment has denser rewards (see Figure 1a), one is more likely to obtain a supervisory reward signal. In sparse reward settings [15], the reward is 0 more frequently, especially, for example, in continuous control environments where a long trajectory is followed and then a single non-zero reward is received at its end. In *MDP Playground*, for discrete environments, the *reward density*, rd, is defined as the fraction of possible sequences of length n that are actually rewarded by the environment, given that n is constant. If num_r sequences are rewarded, we define the reward density to be $rd = num_r / \frac{(|S|-|T|)!}{(|S|-|T|-n)!}$ and the sparsity as $1 - rd$. For continuous environments, density is controlled by having a sparse or dense environment using a *make_denser* configuration option.

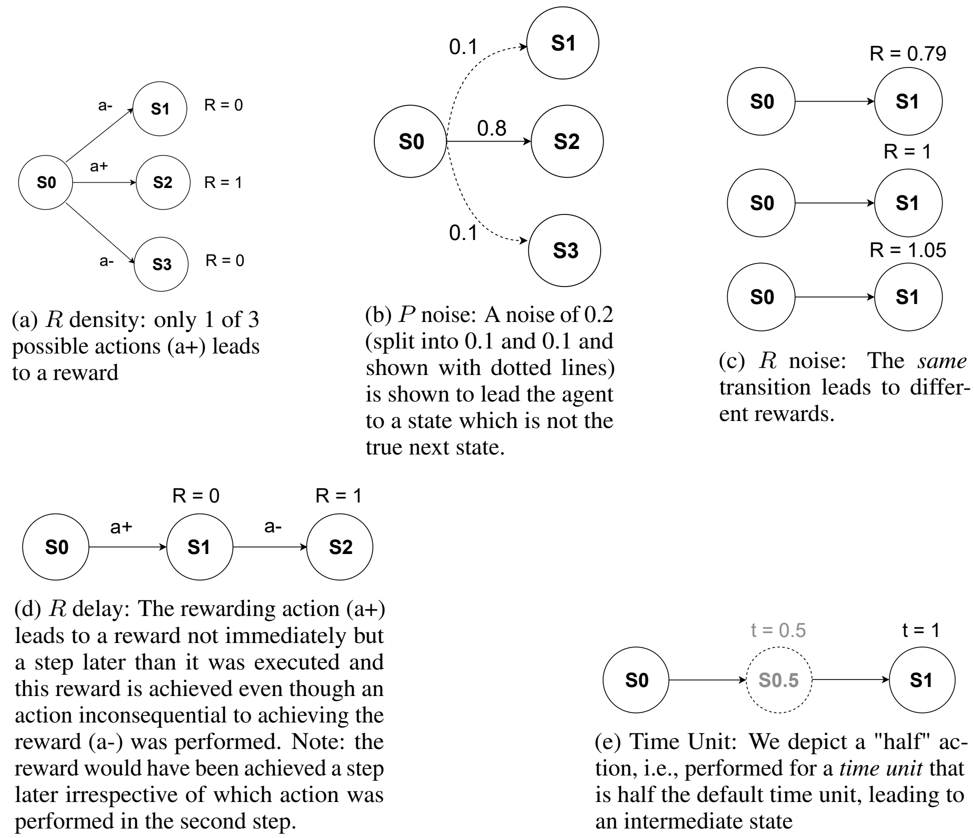
Stochasticity Another characteristic of environments that can significantly impact performance of agents is *stochasticity*. The environment, i.e., dynamics P and R, may be stochastic or may seem stochastic to the agent due to partial observability or sensor noise (see Figure 1b-1c). A robot equipped with a rangefinder, for example, has to deal with various sources of noise in its sensors [55]. In *MDP Playground*, for discrete environments, *transition noise* $t\_n \in [0, 1]$; with probability t_n, an environment transitions uniformly at random to a state that is not the *true* next state given by P. For discrete environments, *reward noise* $r\_n \in \mathbb{R}$; a normal random variable distributed according to $\mathcal{N}(0, {\sigma^2}_{r\_n})$ is added to the *true* reward. For continuous environments, both p_n and r_n are normally distributed and directly added to the states and rewards.

Irrelevant Features Environments also tend to have a lot of *irrelevant features* [45] that one need not focus on. This holds for both table-based learners and approximators like Neural Networks (NNs). NNs additionally can even fit random noise [64] and having irrelevant features is likely to degrade performance. For example, in certain racing car games, though the whole screen is visible, concentrating on only the road would be more efficient without loss in performance. In *MDP Playground*, for discrete environments, a new discrete dimension with its own transition function P_{irr} which is independent of P, is introduced. However, only the discrete dimension corresponding to P is *relevant* to calculate the reward function. Similarly, in continuous environments, dimensions of S and A are labelled as irrelevant and not considered in the reward calculation.

Representations Another aspect is that of *representations*. The same underlying state may have many different external representations/observations, e.g., *feature* space vs *pixel* space. Mujoco tasks may be learnt in feature space vs directly from pixels, and Atari games can use the underlying RAM state or images. For images, various image transformations [*shift*, *scale*, *rotate*, *flip* and others; 19] may manifest as observations of the same underlying state and can pose a challenge to learning. In *MDP Playground*, for discrete environments, when this aspect is enabled, each categorical state is associated with an image of a regular polygon which becomes the externally visible observation o to the agent. This image can further be transformed by *shifting*, *scaling*, *rotating* or *flipping*, which are applied at random to the polygon whenever an observation is generated. For continuous environments, image observations can be rendered for 2D environments. Examples of some generated states can be seen in Figures 10-11 in Appendix I.

Time Unit and Action Range For continuous control problems, we describe 2 additional dimensions here: *action range* [26], a weight penalising actions; and *time unit*, the discretisation of time (see Figure 1e).

We now summarise the dimensions identified above (with the (PO)MDP component they impact in brackets):



(a) R density: only 1 of 3 possible actions (a+) leads to a reward

(b) P noise: A noise of 0.2 (split into 0.1 and 0.1 and shown with dotted lines) is shown to lead the agent to a state which is not the true next state.

(c) R noise: The *same* transition leads to different rewards.

(d) R delay: The rewarding action (a+) leads to a reward not immediately but a step later than it was executed and this reward is achieved even though an action inconsequential to achieving the reward (a-) was performed. Note: the reward would have been achieved a step later irrespective of which action was performed in the second step.

(e) Time Unit: We depict a "half" action, i.e., performed for a *time unit* that is half the default time unit, leading to an intermediate state

Figure 1: We depict some of the dimensions visually following [59]. Not all states and actions are depicted to focus on the dimension of interest. Rewarding actions are shown as a+ while actions shown as a- are not rewarding. Reward is shown as R and time unit as t.

- Reward Delay (R)
- Reward Density (R)
- Transition Noise (P)
- Reward Noise (R)
- Irrelevant Features (O)
- Representations (O)
- Action Range (A)
- Time Unit (P)

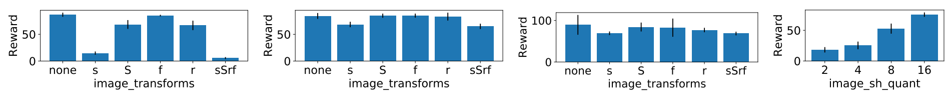
Only selected dimensions are included here, to aid in understanding and to show use-cases for *MDP playground*. Trying to exhaustively identify dimensions has led to a very flexible platform and Appendix B lists all the dimensions of MDP Playground. We would like to point out that it largely depends on the domain which dimensions are important. For instance, in a video game domain, a practitioner may not want to inject any kind of noise into the environment, if their only aim is to obtain high scores, whereas in a domain like robotics adding such noise to a deterministic simulator could be crucial in order to obtain generalisable policies [56].

3 MDP Playground

Code samples An environment instance is created as easily as passing a Python `dict`:

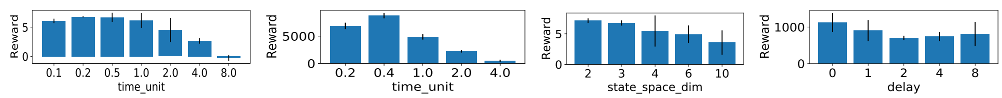
```
from mdp_playground.envs import RLToyEnv
config = {
    'state_space_type': 'discrete',
    'action_space_size': 8,
    'delay': 1,
    'sequence_length': 3,
    'reward_density': 0.25,
    }
env = RLToyEnv(**config)
```

Very low-cost execution Experiments with *MDP Playground* are cheap, allowing academics without special hardware to perform insightful experiments. Wall-clock times depend a lot on the agent, network size (in case of NNs) and the dimensions used. Nevertheless, to give the reader an idea of the runtimes involved, DQN experiments (with a network with 2 hidden layers of 256 units each) took on average 35s for a *complete* run of DQN



(a) DQN (b) Rainbow (c) A3C (d) DQN shift

Figure 2: AUC of episodic reward at the end of training for the different agents **when varying representation**. 's' denotes *shift* (quantisation of 1), 'S' *scale*, 'f' *flip* and 'r' *rotate* in the labels in the first three subfigures and *image_sh_quant* represents quantisation of the *shift*s in the DQN experiment for this. Error bars represent 1 standard deviation. Note the different reward scales.



(a) time unit toy (b) time unit complex (c) irr. dims. rew. (d) DQN qbert

Figure 3: **a** and **b**: DDPG with **time unit** on toy and complex (HalfCheetah) environment at the end of training (*time unit* is relative to the defaults). **c**: DDPG with **irrelevant dimensions** injected on the toy environment. **d**: DQN on qbert. Error bars represent 1 standard deviation. Note the different y-axis scales.

for 20 000 environment steps. In this setting, we restricted Ray RLLib [33] and the underlying Tensorflow [1] to run on *one core of a laptop* (core-i7-8850H CPU – the full CPU specifications for a single core can be found in Appendix R). This equates to roughly 30 minutes for the *entire* delay experiment shown in Figure 12a which was plotted using 50 runs (10 seeds $\times$ 5 settings for *delay*; these 50 runs could also be run in an embarrassingly parallel manner on a cluster). Even when using the more expensive continuous or representation learning environments, runs were only about 3-5 times slower.

Complex Environment Wrappers We further provide wrappers for *Atari* and *Mujoco* which can be used to inject some of the dimensions also into complex environments.

Design decisions While many dimensions can seem challenging at first, it is also the nature of RL that different dimensions tend to be important in different specific applications. The video game domain was provided as an example of this in Section 2.2. Another example is of *reward scale*. The agents we tested here re-scale or clip rewards already and the effects of this dimension are not as important as they would be otherwise. To maintain the flexibility of having as many dimensions as possible and yet keep the platform easy to use, **default** values are set for dimensions that are not configured. This effectively turns off those dimensions. Thus, as in the code example, users only need to provide dimensions they are interested in.

Further design decisions are discussed in detail in Appendix G.

4 Using MDP Playground

We discuss in detail various experiments along with how they may be used to design new agents and to debug existing agents. For the experiments, we set $|S|$ and $|A|$ to 8 and the *terminal state density* to 0.25. The *reward scale* is set to 1.0 whenever a reward is given by the environment. We evaluated *Rllib* implementations [33] of DQN [38], Rainbow DQN [20], A3C [37] on discrete environments and DDPG [34], TD3 [14] and SAC [17] on continuous environments over grids of values for the dimensions. Hyperparameters and the tuning procedure used are available in Appendix O. We used fully connected networks except for pixel-based representations where we used Convolutional Neural Networks (CNNs) [31].

4.1 Designing New Agents

We hope our toy environments will help identify inductive biases needed for designing new RL agents without getting confounded by other sources of "noise" in the evaluation. What is important for doing this is to be able to identify if the trends seen on the toy environments would also occur for more complex environments. We now provide empirical support for this with several experiments.

We tested the trends of the dimensions on more complex Atari and Mujoco tasks. For Atari, we ran the agents on *beam_rider*, *breakout*, *qbert* and *space_invaders* when varying the dimensions *delay* and *transition noise*. For Mujoco, we ran the agents on *HalfCheetah*, *Pusher* and *Reacher* using *mujoco-py* when varying the dimensions *time unit* and *action range*. We evaluated 5 seeds for $500k$ steps for *Pusher* and *Reacher*, $3M$ for *HalfCheetah* and $10M$ ($40M$ frames) for Atari. The values shown for *action range* and *time unit* are relative to the ones used in Mujoco.

Varying *representations* We turned on image representations for discrete environments and applied various transforms (*shift*, *scale*, *rotate* and *flip*) one at a time and also all at once. We observed that the more transforms are applied to the images, the harder it is for agents to learn, as can be seen in Figures 2a-c. This was to be expected since there are many more combinations to generalise over for the agent.

It is important to note, from the point of view of a design platform, that our platform allows us to identify the inductive bias of CNNs being good for image observations without having to conduct such experiments on complex and expensive environments. This is because the toy environments capture many key features of image representations and thus the image classification capabilities of CNNs can help identify the underlying MDP state. In a similar manner, we have captured key features of other dimensions. If one were to design a new inductive bias which helps the agent identify the underlying MDP state in the presence of the other dimensions, this could be tested in a coarse and quick manner on our platform.

Varying *time unit* We observed that the *time unit* has an optimal value which has significant impact on performance in the toy continuous environment (Figure 3a), i.e., that it can be neither too small nor too large. We decided to tune the *time unit* also for complex environments (Figures 3b, 8 and 9). The insight from the toy environment transferred to the complex case and there were gains of even 100% in some cases over the default value of the time units used in the "expert-tuned" environments. A further insight to be had is that for simpler environments like the toy, *Pusher* and *Reacher*, the effect of the selection of the *time unit* was not as pronounced as for a more complex environment like *HalfCheetah*. This makes intuitive sense as one can expect a narrower range of values to work for more complex environments. This shows that it is even more important to tune such dimensions for more complex environments.

The *basic* agent design we showed above does this once and sets its optimal *time unit* statically. An ideal adaptive agent design would even set the *time unit* in an *online* manner. Since the trends from the toy environment coarsely transfer to the complex environments, coarse and quick insights can be gained on the toy environments.

Varying *action range* We observed similar trends as for *time unit*, in that there was an optimal value of *action range*, i.e., that it can be neither too small nor too large. Figure 9 shows this for all considered agents on HalfCheetah (for SAC and DDPG, runs for *action range* values $>= 2$ and $>= 4$ crashed and are absent from the plot). This supports the insight gained on our simpler environment that tuning this value may lead to significant gains for an agent. For already tuned environments, such as the ones in *Gym*, this dimension is easily overlooked but when faced with new environments setting it appropriately can lead to substantial gains. In fact, even in the tuned environment setting of *Gym*, we found that all three algorithms performed best for an *action range* 0.25 times the value found in *Gym* for *Reacher* (Figures 8c, 8k, 8g in Appendix H). Moreover, the learning curves in Appendix N further show that for increasing *action range* the training gets more variant. The difference in performances across the different values of *action range* is much greater in the complex environments. We believe this is due to correlations within the multiple degrees of freedom as opposed to a rigid object in the toy environment.

To the best of our knowledge, the impact of *time unit* and *action range* is under-researched while developing agents because the standard environments have been pre-configured by experts. However, it's clear from Figure 3b, that pre-configured values were not optimal and even basic tuning improves performance significantly in even *known* environments. In a completely *unknown* environment, if we want agents to perform optimally, these dimensions would need to be taken into account even more when designing agents.

Varying *transition noise* We observe similar trends for injecting transition noise into Atari environments for all three agents as for the toy environments. We also observe that for some of the environments, transition noise actually helps improve performance. This has also been observed in prior work [61]. This happens when the exploration policy was not tuned optimally since inserting transition noise is almost equivalent to ϵ-greedy exploration for low values of noise. We also observed a similar effect for the toy environments in Figure 18 in Appendix J. However, we also observe that performance drop is different for different environments. This is to be expected as there are other dimensions of hardness which we cannot control or measure for these environments.

Varying *reward delay* We see that on average performance drops for the delay experiments when more delay is inserted , as was the case for the toy environments. For *qbert* (Figure 3d), these drops are greater on average across the agents. However, for *breakout* (Figure 6b), in many instances, we don't even see performance drops. In *beam_rider* (Figure 6a) and *space_invaders* (Figure 6d), the magnitude of these effects are intermediate to *breakout* and *qbert*. This trend becomes clearer when we also look at Figures 7b-p in Appendix H. We believe this is because large delays from played action to reward are already present in *breakout*, which means that inserting more delays does not have as large an effect as in *qbert* (Figures 3d). Agents are strongest affected in qbert which, upon looking at gameplay, we believe has the least delays from rewarding action to reward compared to the other games. The trends for delay were noisier than for transition noise, even though on average the trends transferred from MDP Playground to the complex environments. Many considered environments tend to also have repetitive sequences which would dilute the effect of injecting delays. Many of the learning curves in Appendix N, with delays inserted, are indistinguishable from normal learning curves. We believe that, in addition to the motivating examples, this is empirical evidence that delays are already present in these environments and so inserting them does not cause the curves to look vastly different. In contrast, when we see learning curves for transition noise, we observe that, as we inject more and more noise, training tends to a smoother curve as the agent tends towards becoming a completely random agent.

Additionally, we also have experiments with similar trends also for another dimension - *reward noise*. The average rank correlation over 12 experiments (3 agents x 4 Atari environments) was 0.867 for *transition noise*, 0.617 for *reward delay*, and 0.733 for *reward noise*. Tables 1, 2 and 3 list the individual rank correlation for each experiment, i.e. agent, environment and dimension.

To analyse transfer of dimensions between toy and complex benchmarks, for the Atari experiments, we use the Spearman rank correlation coefficient between corresponding toy and complex experiments for performance across different values of the dimension of hardness. The Spearman correlation was $>= 0.7$ for 19 out of 24 experiments and a positive correlation for four of the remaining five. DQN with delays added on breakout was the only experiment with correlation 0.

Varying *irrelevant features* We observed that introducing *irrelevant dimensions* to the control problem, while keeping the number of relevant dimensions fixed to 2, decreased an agent's performance (see Figures 3c & 17f). This gives us the insight that having irrelevant features interferes with the learning process. An inductive bias that learns to focus only on the relevant dimensions could be unit-tested to gain coarse insights on the toy environments.

We have shown similar trends for SAC on HalfCheetah in Figure 9a in Appendix H.

Varying Multiple Dimensions In *MDP Playground*, it is possible to vary multiple dimensions at the same time in the same base environment. For instance, Figure 4d shows the interaction effect (an inversely proportional relationship) between the *action range* and the *time unit* in the continuous toy environment with DDPG. This insight allows us to design an adaptive agent which sets its *action range* depending on the *time unit* and vice versa. Since many real-world systems can be described in terms of a simple rigid body moving towards a target point, the toy continuous environment is a useful testbed for this.

More such experiments can be found in Appendix L, including varying both P and R *noise*s together in discrete environments and more. Further design ideas for new agents can be found in Appendix E.

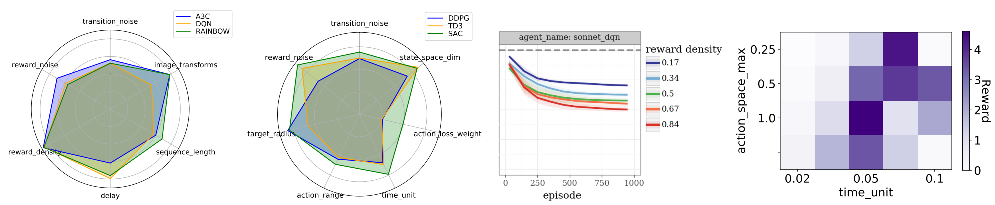
4.2 Insights into Existing Agents

Apart from the insights gained for designing agents above, we discuss more insights for existing agents explicitly here.

The experiment for varying representations on toy environments discussed above (Figures 2a-c) further showed that the degradation in performance is much stronger for DQN compared to Rainbow and A3C which are known to perform better than DQN in complex environments.

This led us to another interesting insight regarding the inductive bias of CNNs. It was unexpected for us that the most problematic transform for the agents to deal with was *shift*. Despite the spatial invariance learned in CNNs [30], our results imply that that seems to be the hardest one to adapt to. As these trends were strongest in DQN, we evaluated further ranges for the individual transforms for DQN. Here, *shifts* had the most possible different combinations that could be applied to the images. Therefore, we quantised the *shift*s to have fewer possible values. Figure 2d shows that DQN's performance improved with increasing quantisation (i.e., fewer possible values) of *shift*. We noticed similar trends for the other transforms as well, although not as strong as they do not have as many different values as *shift* (see Figures 29b-c in Appendix J). We emphasize that in a more complex setting, we would have easily attributed some of these results to luck but in the setting where we had individual control over the dimensions, our platform allowed us to dig deeper in a controlled manner.

Another insight we gain is from the *time unit* experiment (see Figures 3a and 3b), which indicates *time unit* should not be infinitesimally small to achieve too fine-grained control since there is an optimal *time unit* for which we should repeat the same action [7].



(a) discrete envs. (b) cont. envs. (c) reward density (d) action range + time unit

Figure 4: Analysing and Debugging

In Figure 3d, where we varied *delay* on *qbert*, we show how a dimension induces hardness in an environment. This result is representative of the experiments on toy and complex environments which are included in Appendix H and H with the difference that results are noisier in complex environments since the dimensions are already present there in varying degrees. We, thus, studied what kinds of failure modes can occur when an agent is faced with such dimensions and even obtained noisy learning curves typically associated with RL on the *toy* environments as can be seen in Appendix M.

At the same time, the experiment in Figure 3d also shows how the complex environment wrappers allow researchers, who are curious, to study the robustness of their agents to these dimensions on complex environments, without having to fiddle with lower-level code. This is a typical use-case further down the agent development pipeline, i.e., close to deployment.

Design and Analyse Experiments We allow the user the power to inject dimensions into toy or complex environments in a fine-grained manner. This can be used to define custom experiments with the dimensions. The results can be analysed in an accompanying *Jupyter notebook* using the 1D plots. There are also radar plots inspired by bsuite [42], but with more flexibility in choosing the dimensions, and these can even be applied to complex environment experiments. Since, different users might be interested in different dimensions, these are loaded dynamically from the data. For instance, radar plots for the dimensions we varied in our toy experiments can be seen as in Figures 4a and 4b.

4.3 Debugging Agents

Analysing how an agent performs under the effect of various dimensions can reveal unexpected aspects of an agent. For instance, when using bsuite agents, we noticed that when we varied our environment's *reward density*, the performance of the bsuite Sonnet DQN agent would go up in proportion to the density (see Figure 4c). This did not occur for other bsuite agents. This seemed to suggest something different for the DQN agent and when we looked at DQN's hyperparameters we realised that it had a fixed ϵ schedule while the other agents had decaying schedules. Such insights can easily go unnoticed if the environments used are too complex. The high bias nature of our toy environments helps debug such cases.

In another example, in one of the Ray versions we used, we observed that DQN was performing well on the *varying representations* environment while Rainbow was performing poorly. We were quickly able to ablate additional Rainbow hyperparameters on the toy environments and found that their noisy nets [13] implementation was broken (see Figure 5 in Appendix). We then tested and observed the same on more complex environments. This shows how easily and quickly agents can be debugged to see if something major is broken. This, in combination with their low computational cost, also makes a case to use the toy environments in Continuous Integration (CI) tests on repositories.

Further, we believe the same structured nature of *MDP Playground* also makes it a valuable tool for theoretical research. We evaluated tabular baselines Q-learning [52], Double Q-learning [60] and SARSA [52] on the discrete non-image based environments with similar qualitative results to those for deep agents. These can be found in Appendix K. This makes our platform a bridge between theory and practice where both kinds of agents can be tested.

The experiments here are only a glimpse into the power and flexibility of MDP Playground. Users can even upload custom Ps and Rs and custom images for representations O and our platform takes care of injecting the other dimensions for them (wherever possible). This allows users to control different dimensions in the same base environment and gain further insights.

5 Discussion and Related Work

The *Behaviour Suite for RL* [bsuite; 42] is the closest related work to MDP Playground. [42] collect known (toy) environments from the literature and use these to characterise agents based on their performance on these environments. Most environments in *bsuite* can be seen as an intermediate step between our MDPs and more complex environments. This is because *bsuite*'s environments are already more specific and complex than the toy environments in *MDP Playground*. This makes *bsuite*'s dimensions not orthogonal and *atomic* like ours and thus not individually controllable. Fine-grained control is a feature that sets our platform apart. *bsuite* has a collection of *presets* chosen by experts which work well but would be much harder to play around with. While *MDP Playground* also has good presets through default values defined for experiments, it is much easier to configure. Further, it also means that *bsuite* experiments are much more expensive than ours. While *bsuite* itself is quite cheap to run, *MDP Playground* experiments are an order of magnitude cheaper. In contrast to *bsuite*, we demonstrate how the identified trends on the toy and complex environments can be used to design and debug agents. Further, *bsuite* currently has no toy environment for Hierarchical RL (HRL) agents while *MDP Playground*'s rewardable sequences fits very well with HRL. Finally, bsuite offers **no continuous control environments**, whereas MDP Playground provides both discrete and continuous environments. This is important because several agents like DDPG, TD3, SAC are designed for continuous control. A more detailed comparison with bsuite and other related work can be found in Appendix D.

Toybox [58] and Minatar [62] are also cheap platforms like ours with similar goals of gaining deeper insights into RL agents. However, their games target the specific *Atari* domain and are, like *bsuite*, more specific and complementary to our approach.

We found [3] the most similar work to ours in spirit. They propose that current deep RL research has been increasing the complexity of the dynamics P but has not paid much attention to the state distributions and reward distribution over which RL policies work and that this has made RL agents brittle. This also raises concerns about the narrow scope of these so-called "complex" environments and we aim to remedy that with our dimensions. We agree with them in this regard. However, they only target continuous environments. We capture their dimensions in a different manner and offer many more dimensions with fine-grained control. Furthermore, their code is not open-source.

Further research includes *Procgen* [11], *Obstacle Tower* [24] and *Atari* [6]. Procgen adds various heterogeneous environments and tries to quantify generalisation in RL. In a similar vein, Obstacle Tower provides a generalization challenge for problems in vision, control, and planning. These benchmarks do not capture orthogonal dimensions of difficulty and as a result, they do not have the same type of fine-grained control over their environments' difficulty and neither can each dimension be controlled independently. We view this as a crucial aspect when testing new agents. [12] provides some overlapping dimensions with our platform but it consists of only continuous environments, and doesn't target the toy domain.

6 Limitations of the Approach and its Ethical and Societal Implications

The toy environments are meant to be design and debug testbeds and not for engineering/tuning the final agent HPs. As such, they are extremely cheap compared to complex environments and (as one would expect), they can only be used to draw high-level insights that transfer and are likely not as discriminating as complex environments for many of the finer changes between RL agents. They also cannot be used directly to determine the values of hyperparameters (HPs) to use on complex environments. For example, just as complex environments require bigger NNs, they would need correspondingly different HPs, such as bigger replay buffers. Even the performance of agents in bsuite (which has more complex environments than our benchmark) do not transfer to the more complex environments (`https://github.com/deepmind/bsuite/issues/14`). In a similar vein, to the best of our knowledge, MNIST hyperparameters do not transfer to ImageNet and it is only used for testing out initial design ideas.

Further, high-dimensional control problems where there are interaction effects between degrees of freedom are not captured in the toy rigid body control problem as this is the domain of complex benchmarks and beyond the scope of this platform. (The platform does provide complex environment wrappers, though, which inject some of the mentioned dimensions. We couldn't find such wrappers in the literature/on the Internet.)

Finally, Multi-Agent RL, Multi Objective RL, Time Varying MDPs (and probably some more research areas) are beyond the scope of the current work.

In terms of the broader impact on society and ethical considerations, we foresee no direct impact, only indirect consequences through RL since our work promotes standardisation and reproducibility which should accelerate RL research. An additional environmental impact would be that, at least, prototyping and testing of agents could be done cheaply, reducing carbon emissions.

7 Conclusion and Future Work

We introduced a low-cost platform to design and debug RL agents and provided instructions on how to use it with supporting experiments. The platform allows us to disentangle various factors that make RL environments hard by providing fine-grained control over various dimensions. This also lends itself to easily achievable insights and helps debug agents. We further demonstrated how the performance of the studied agents is adversely affected by the dimensions. To the best of our knowledge, we are the first to perform a principled study of how significant aspects such as non-Markov information states, irrelevant features, representations and low-level dimensions, like time discretisation, affect agent performance.

We want *MDP Playground* to be a community-driven effort and it is open-source for the benefit of the RL community at `https://github.com/automl/mdp-playground`. While we tried to exhaustively identify dimensions of hardness, it is unlikely that we have captured *all* orthogonal dimensions in RL. We welcome more dimensions that readers think will help us encapsulate further challenges in RL and will add them based on the community's thoughts.

Future work can tackle not only theoretical development of such dimensions but also additional analysis of such dimensions in complex domains such as *Mujoco* and dexterous manipulation [46].

Given the current brittleness of RL agents [18], and many claims that have been challenged [5, 58], we believe RL agents need to be tested on a lower and more basic level to gain insights into their inner workings. *MDP Playground* is like a programming language for regularly structured MDPs which allows delving deeper into the inner workings of RL agents.

Acknowledgements

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, by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG. They would like to thank their group, especially Joerg, Steven, Samuel, for helpful feedback and discussions. Raghu would like to additionally thank Michael Littman for his feedback and encouragement and the RLSS 2019, Lille organisers and participants who he had interesting discussions with.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

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

[3] Olov Andersson and Patrick Doherty. Toward robust deep rl via better benchmarks: Identifying neglected problem dimensions. In *2nd Reproducibility in Machine Learning Workshop at ICML 2018, Stockholm, Sweden*, 2018.

[4] J. A. Arjona-Medina, M. Gillhofer, M. Widrich, T. Unterthiner, J. Brandstetter, and S. Hochreiter. RUDDER: return decomposition for delayed rewards. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Proceedings of the 32nd International Conference on Advances in Neural Information Processing Systems (NeurIPS'19)*, pages 13544–13555, 2019.

[5] Akanksha Atrey, Kaleigh Clary, and David D. Jensen. Exploratory not explanatory: Counterfactual analysis of saliency maps for deep reinforcement learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013.

[7] A. Biedenkapp, R. Rajan, F. Hutter, and M. Lindauer. Towards TempoRL: Learning when to act. In *Workshop on Inductive Biases, Invariances and Generalization in RL (BIG@ICML'20)*, July 2020.

[8] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI gym. *arXiv:1606.01540 [cs.LG]*, June 2016.

[9] P. Chrabaszcz, I. Loshchilov, and F. Hutter. Back to basics: Benchmarking canonical evolution strategies for playing atari. In J. Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, (IJCAI'18)*, pages 1419–1426. ijcai.org, 2018.

[10] K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Proceedings of the 31st International Conference on Advances in Neural Information Processing Systems (NeurIPS'18)*, pages 4754–4765, 2018.

[11] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman. Leveraging Procedural Generation to Benchmark Reinforcement Learning. *arXiv:1912.01588 [cs.LG]*, Dec 2019.

[12] Gabriel Dulac-Arnold, Nir Levine, Daniel J. Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. An empirical investigation of the challenges of real-world reinforcement learning. *CoRR*, abs/2003.11881, 2020.

[13] M. Fortunato, M. G. Azar, B. Piot, J. Menick, M. Hessel, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg. Noisy networks for exploration. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*, 2018. Published online: `iclr.cc`.

[14] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, pages 1582–1591. PMLR, 2018.

[15] R. D. Gaina, S. M. Lucas, and D. Pérez-Liébana. Tackling sparse rewards in real-time games with statistical forward planning methods. In *Proceedings of the 33rd Conference on Artificial Intelligence (AAAI'19)*, pages 1691–1698. AAAI Press, 2019.

[16] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, (ICML'17)*, pages 1352–1361. PMLR, 2017.

[17] T. Haarnoja, A. Zhou, P. Abbeel, and Sergey Levine. Soft Actor-Critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In J. G. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, pages 1856–1865. PMLR, 2018.

[18] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In S. A. McIlraith and K. Q. Weinberger, editors, *Proceedings of the Conference on Artificial Intelligence (AAAI'18)*, pages 3207–3214. AAAI Press, 2018.

[19] D. Hendrycks and T. G. Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*, 2019. Published online: `iclr.cc`.

[20] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In S. A. McIlraith and K. Q. Weinberger, editors, *Proceedings of the Conference on Artificial Intelligence (AAAI'18)*, pages 3215–3222. AAAI Press, 2018.

[21] Alex Irpan. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[22] T. Jaakkola, S. P. Singh, and M. I. Jordan. Reinforcement learning algorithm for partially observable markov decision problems. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Proceedings of the 7th International Conference on Advances in Neural Information Processing Systems (NeurIPS'95)*, pages 345–352, 1995.

[23] Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *J. Mach. Learn. Res.*, 11:1563–1600, 2010.

[24] A. Juliani, A. Khalifa, V.P. Berges, J. Harper, E. Teng, H. Henry, A. Crespi, J. Togelius, and D. Lange. Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning. In S. Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2684–2691. ijcai.org, Feb 2019.

[25] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

[26] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning. In *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020*, pages 479–486. IEEE, 2020.

[27] Geir Kirkebøen and Gro HH Nordbye. Intuitive choices lead to intensified positive emotions: An overlooked reason for "intuition bias"? *Frontiers in Psychology*, 8:1942, 2017.

[28] P. Klink, H. Abdulsamad, B. Belousov, and J. Peters. Self-paced contextual reinforcement learning. In L. P. Kaelbling, D. Kragic, and K. Sugiura, editors, *3rd Annual Conference on Robot Learning, (CoRL'19)*, pages 513–529. PMLR, 2019.

[29] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[30] Y. LeCun. Learning invariant feature hierarchies. In A. Fusiello, V. Murino, and R. Cucchiara, editors, *Computer Vision - ECCV 2012*, pages 496–505. Springer, 2012.

[31] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Back-propagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, 1989.

[32] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[33] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica. RLlib: Abstractions for distributed reinforcement learning. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, volume 80, pages 3059–3068. Proceedings of Machine Learning Research, 2018.

[34] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In Y. Bengio and Y. LeCun, editors, *Proceedings of the International Conference on Learning Representations (ICLR'16)*, 2016. Published online: `iclr.cc`.

[35] Michael L. Littman, Ufuk Topcu, Jie Fu, Charles Lee Isbell Jr., Min Wen, and James MacGlashan. Environment-independent task specifications via GLTL. *CoRR*, abs/1704.04341, 2017.

[36] Odalric-Ambrym Maillard, Timothy A. Mann, and Shie Mannor. How hard is my mdp?" the distribution-norm to the rescue". In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 1835–1843, 2014.

[37] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In M. Balcan and K. Weinberger, editors, *Proceedings of the 33rd International Conference on Machine Learning (ICML'16)*, volume 48, pages 1928–1937, 2016.

[38] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[39] Norman Mu and Justin Gilmer. MNIST-C: A robustness benchmark for computer vision. *CoRR*, abs/1906.02337, 2019.

[40] D. S. Nau. Pathology on game trees revisited, and an alternative to minimaxing. *Artif. Intell.*, 21(1-2):221–244, 1983.

[41] Ronald Ortner, Pratik Gajane, and Peter Auer. Variational regret bounds for reinforcement learning. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, volume 115 of *Proceedings of Machine Learning Research*, pages 81–90. AUAI Press, 2019.

[42] I. Osband, Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepezvari, S. Singh, B. Van Roy, R. Sutton, D. Silver, and H. Van Hasselt. Behaviour suite for reinforcement learning. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*, 2019. Published online: `iclr.cc`.

[43] J. Pearl. Theoretical impediments to machine learning with seven sparks from the causal revolution. In Y. Chang, C. Zhai, Y. Liu, and Y. Maarek, editors, *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, (WSDM'18)*, page 3. ACM, February 2018.

[44] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.

[45] J. Rajendran, J. Ganhotra, S. Singh, and L. Polymenakos. Learning end-to-end goal-oriented dialog with multiple answers. In E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'19)*, pages 3834–3843. Association for Computational Linguistics, 2018.

[46] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. In *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018.

[47] R. Ramanujan, A. Sabharwal, and B. Selman. On adversarial search spaces and sampling-based planning. In R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling, (ICAPS'10)*, pages 242–245. AAAI, 2010.

[48] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347 [cs.LG]*, 2017.

[49] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[50] Elizabeth S Spelke and Katherine D Kinzler. Core knowledge. *Developmental science*, 10(1):89–96, 2007.

[51] Jayakumar Subramanian, Amit Sinha, Raihan Seraj, and Aditya Mahajan. Approximate information state for approximate planning and reinforcement learning in partially observed systems. *CoRR*, abs/2010.08843, 2020.

[52] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. The MIT Press, second edition, 2018.

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

[54] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In I. Dhillon, Y. Koren, R. Ghani, T. Senator, P. Bradley, R. Parekh, J. He, R. Grossman, and R. Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*, pages 847–855. ACM Press, 2013.

[55] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics*. Intelligent robotics and autonomous agents. MIT Press, 2005.

[56] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*, pages 23–30. IEEE, 2017.

[57] E. Todorov, T. Erez, and Y. Tassa. MuJoCo: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems (IROS'12)*, pages 5026–5033. IEEE, 2012.

[58] Emma Tosch, Kaleigh Clary, John Foley, and David D. Jensen. Toybox: A suite of environments for experimental evaluation of deep reinforcement learning. *CoRR*, abs/1905.02825, 2019.

[59] Alexander Matt Turner. Optimal farsighted agents tend to seek power. *CoRR*, abs/1912.01683, 2019.

[60] H. van Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Proceedings of the 24th International Conference on Advances in Neural Information Processing Systems (NeurIPS'10)*, pages 2613–2621, 2010.

[61] T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba. Benchmarking model-based reinforcement learning. *arXiv:1907.02057 [cs.LG]*, 2019.

[62] Kenny Young and Tian Tian. Minatar: An atari-inspired testbed for more efficient reinforcement learning experiments. *CoRR*, abs/1903.03176, 2019.

[63] Baohe Zhang, Raghu Rajan, Luis Pineda, Nathan Lambert, André Biedenkapp, Kurtland Chua, Frank Hutter, and Roberto Calandra. On the Importance of Hyperparameter Optimization for Model-based Reinforcement Learning. In *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS)'21*, April 2021.

[64] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, (ICLR'17)*. OpenReview.net, 2017.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] The orthogonal dimension that influence RL agents performances are presented and their role in the implemented MDPs is discussed in Section 2. We showed that varying these dimensions can provide new insights or confirm existing insights (on the toy environments that also hold on more complex ones) in Section 4.2. We discussed how our proposed benchmark can aid in designing new agents by taking the proposed dimensions into account during the design (see Section 4.1). Finally, we discuss how the benchmark can help in debugging agents and could be used for continuous integration (see Section 4.3).
   (b) Did you describe the limitations of your work? [Yes] See Section 6.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See `https://github.com/automl/mdp-playground` and the link is also given in Section 7.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix P
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] In Section 3 we discussed the low-cost execution of experiments on MDP Playground and we provide further details along with hardware specifications in the Appendix R.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Benchmark Track Checklist

1. Submission introducing new datasets must include the following in the supplementary materials:
   (a) Dataset documentation and intended uses. Recommended documentation frameworks include datasheets for datasets, dataset nutrition labels, data statements for NLP, and accountability frameworks. [Yes] Available in the platform's repository: `https://github.com/automl/mdp-playground`
   (b) URL to website/platform where the dataset/benchmark can be viewed and downloaded by the reviewers. [Yes] Available in the platform's repository: `https://github.com/automl/mdp-playground`
   (c) Author statement that they bear all responsibility in case of violation of rights, etc., and confirmation of the data license. [Yes] The license is an Apache license, available in the platform's repository: `https://github.com/automl/mdp-playground`
   (d) Hosting, licensing, and maintenance plan. The choice of hosting platform is yours, as long as you ensure access to the data (possibly through a curated interface) and will provide the necessary maintenance. [Yes] The platform's repository is publicly hosted on GitHub and we will actively continue to maintain and develop MDP Playground further. We also welcome and support community-driven efforts such as pull request, reported issues and forum discussions.
2. To ensure accessibility, the supplementary materials for datasets must include the following:
   (a) Links to access the dataset and its metadata. This can be hidden upon submission if the dataset is not yet publicly available but must be added in the camera-ready version. In select cases, e.g when the data can only be released at a later date, this can be added afterward. Simulation environments should link to (open source) code repositories. [Yes] Available in the platform's repository: `https://github.com/automl/mdp-playground`
   (b) The dataset itself should ideally use an open and widely used data format. Provide a detailed explanation on how the dataset can be read. For simulation environments, use existing frameworks or explain how they can be used. [Yes] The documentation is available in the platform's repository: `https://github.com/automl/mdp-playground`
   (c) Long-term preservation: It must be clear that the dataset will be available for a long time, either by uploading to a data repository or by explaining how the authors themselves will ensure this. [Yes] The platform will be actively maintained with input from the community
   (d) Explicit license: Authors must choose a license, ideally a CC license for datasets, or an open source license for code (e.g. RL environments). [Yes] The license is an Apache license, available in the platform's repository: `https://github.com/automl/mdp-playground`
   (e) Add structured metadata to a dataset's meta-data page using Web standards (like schema.org and DCAT): This allows it to be discovered and organised by anyone. If you use an existing data repository, this is often done automatically. [N/A]
   (f) Highly recommended: a persistent dereferenceable identifier (e.g. a DOI minted by a data repository or a prefix on identifiers.org) for datasets, or a code repository (e.g. GitHub, GitLab,...) for code. If this is not possible or useful, please explain why. [Yes] GitHub repository: `https://github.com/automl/mdp-playground`
3. For benchmarks, the supplementary materials must ensure that all results are easily reproducible. Where possible, use a reproducibility framework such as the ML reproducibility checklist, or otherwise guarantee that all results can be easily reproduced, i.e. all necessary datasets, code, and evaluation procedures must be accessible and documented. [Yes] The experiments adhere to the ML reproducibility checklist at: `https://arxiv.org/abs/2003.12206`
4. For papers introducing best practices in creating or curating datasets and benchmarks, the above supplementary materials are not required. [N/A]

B Dimensions in MDP Playground

We list here the dimensions for *MDP Playground*. Details on each dimension can be found in the documentation for the class `mdp_playground.envs.RLToyEnv` in the accompanying code.

- Reward Delay
- Rewardable Sequence Length
- Reward Sparsity
- P Noise
- R Noise
- Irrelevant Features
- Transforms for Representation Learning
- Reward Shift
- Reward Scale
- State space size/dimensionality
- Action space size/dimensionality
- Terminal State Density
- Terminal State Reward
- Relevant Dimensions (for both state and action spaces)

$*$ Only for discrete environments:
  - Diameter
  - Reward Distribution
  - Image Representations

  $*$ Only for Image Representations:
    - Shift Quantisation
    - Scale Range
    - Rotation Quantisation

$*$ Only for continuous environments:
  - Target Point
  - Target Radius
  - Time Unit
  - Inertia
  - State Space Max
  - Action Space Max
  - Transition Dynamics Order
  - Reward Function

$*$ Currently fixed dimensions:
  - Initial State Distribution

B.1 More exposition on the dimensions in MDP Playground

We also mention here the Q^*-value [38] and use it as an example to argue how violations of assumptions may lead to degradation in performance. For a state s and action a, a policy π and r_t the reward a timestep t, Q^* is defined as: $Q^*(s, a) = \max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_t = s, a_t = a, \pi\right]$.

For many environments, in many situations, agents perform an action that is consequential to receiving a reward but the agent is only rewarded in a *delayed* manner [see e.g. 4]. For example, shooting at an enemy ship in *Space Invaders* leads to rewards much later than the action of shooting. Any action taken after that is inconsequential to obtaining the reward for destroying that enemy ship. Regarding the Q^* value, this means that if an incorrect information state is used, then updates performed for approximating Q^* will tend to assign partial credit also to inconsequential actions.

In many environments, a reward is obtained for a *sequence* of actions taken and not just the information state and action. A simple example is executing a tennis serve, where one needs a sequence of actions which results in a point, e.g., if an ace was served. In contrast to *delayed rewards*, rewarding a sequence of actions addresses the actions taken which are consequential to obtaining a reward. [53] present a framework for temporal abstraction in RL to deal with such sequences. Regarding the Q^* value, this means that if an incorrect information state is used, then updates performed for approximating Q^* will tend to assign partial credit also to incomplete sequences. The agent may not realise that a whole sequence of actions is needed to be taken and not just some of them. While agents can converge asymptotically in the face of both delays and sequences, using the correct information state would lead to much better sample efficiency and more stable learning. In *MDP Playground*, for discrete environments, only specific sequences of states of positive integer length n are rewardable. Sequences consist of non-repeating states allowing for $\frac{(|S|-|T|)!}{(|S|-|T|-n)!}$ sequences. For the continuous environment of moving to a target, n is variable.

Environments can also be characterised by their *reward density*. In sparse reward settings [15], the supervisory reward signal is 0 throughout the trajectory and then a single non-zero reward is received at its end. This also holds true for the example of the tennis serve above.

The *diameter* of an MDP, i.e., the maximum distance between 2 states, is another significant dimension affecting performance and reachability of states [23, 41]. If rewarding states are very far apart, then an agent would get less reward on average. In *MDP Playground*, for discrete environments, for $diameter = d$, the set of states is set to be a d-partite graph, where, if we order the d sets as $1, 2, .., d$, states from set n will have actions leading to states in set $n+1$, with the final set d having actions leading to states in set 1. The number of actions for each state will, thus, be $(\textit{number of states})/(d)$. This gives the discrete environments a grid-world like structure. For continuous environments, setting the dimension *state space max* sets the bounds of the environment to ±*state space max* and the $diameter = 2\sqrt{2}$ *state space max*.

Further, an additional dimensions for continuous control problems we implement is *target radius* for the task of reaching a target: we have *target radius* [see, e.g., 28], a measure of the distance from the target within which we consider the target to have been successfully reached.

We now mathematically highlight some of our dimensions of hardness to aid understanding. The information state of an agent to compute an optimal policy would need to stack the previous $n + d$ observation and action pairs from the environment where n denotes a *sequence length* and d denotes a *delay*, i.e., a sequence of actions needs to be followed to obtain a reward which may be delayed by a certain number of steps. Reward density controls the fraction of elements in S^n that are rewardable.

Additionally, the continuous control dimensions can mathematically be described as follows. The *target radius* sets $T = \{s \mid \|s - s_t\|_2 < \textit{target radius}\}$, where s_t is the target point. The *action range* sets $A \subset \mathbb{R}^a$ where a is the action space dimensionality. The *time unit*, t, sets $P(s, a) = s + \int_t P_{cont}(s, a)\ dt$ where P_{cont} is the underlying continuous dynamics function. The *transition dynamics order*, n, sets P to be in C^n, the set of functions differentiable n times.

B.2 Additional density option for sequences

With regard to density, 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, 57], 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 [52] 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 observe the effects on algorithms. To achieve this, when $make\_denser$ is $True$, the environment gives a fractional reward if a fraction of a rewardable sequence is achieved in discrete environments. For continuous environments, for the move to a target point reward function, this option toggles between giving a dense reward as described in the main paper and giving a sparse reward when the agent is within the *target radius*.

C Algorithm for generating MDPs

Algorithm 1 Automatically Generated MDPs with MDP Playground

Input:
reward delay d,
rewardable sequence length n,
transition noise t_n or σ_{t_n},
reward noise σ_{r_n},
`reward_scale`,
`reward_shift`,
`term_state_reward`,
`make_denser`,
`relevant_dimensions`
▷ Dimensions specific to discrete environments
number of states $|S|$,
`diameter`,
reward density rd,
`terminal_state_density`,
reward distribution `reward_dist`
▷ Dimensions specific to continuous environments
`target_point`,
`target_radius`,
`transition_dynamics_order`,
`time_unit`,
`inertia`

function INIT_TRANSITION_FUNCTION():
  if discrete environment **then**
    Set $|A| = |S|/diameter$
    Divide S into independent sets S_i with $|A|$ elements in each with $i = 1, 2, ..., diameter$
    for each independent set S_i **do**
      for each state s in S_i **do**
        Set possible successor states: $S' = S_{i+1}$
        for each action a **do**
          Set $P(s, a) = s'$ sampled uniformly from S' and remove s' from S'
    if irrelevant features **then**
      Generate dynamics P_{irr} of irrelevant part of state space as was done for P
  else
    Do nothing as continuous environments have a fixed parameterisation

function INIT_REWARD_FUNCTION(n):
  if discrete environment **then**
    Randomly sample $rd * \frac{(|S|-|T|)!}{(|S|-|T|-n)!}$ and store in `rewardable_sequences` with corresponding reward sampled according to `reward_dist` if enabled
    ▷ The actual formula is more complicated because of the diameter
    ▷ Only those sequences are sampled which are *legal* according to P
  else
    Do nothing as continuous environments have fixed options for the reward function

function TRANSITION_FUNCTION(s, a):
  if discrete environment **then**
    $s' = P(s, a)$
    if $\mathcal{U}(0, 1) < t\_n$ **then**
      s' = a random state in $S \setminus \{P(s, a)\}$ ▷ Inject noise
    Observation $o = s'$
    if irrelevant features **then**
      Execute dynamics P_{irr} of irrelevant part of state space and concatenate with s' to get observation o
    if representation learning **then**
      o = image of corresponding polygon(s) with applied selected transforms
  else
    Set n = `transition_dynamics_order`
    Set $a^n = a$ ▷ Superscript n represents n^{th} derivative
    Set $s^n = a^n / inertia$ ▷ Each state dimension is controlled by each action dimension
    for i in reversed(range(n)) **do**
      Set $s^i_{t+1} = \sum_{j=0}^{n-i} s^{i+j}_t \cdot \frac{1}{j!} \cdot time\_unit^j$ ▷ t is current time step.
    $s_{t+1} \mathrel{+}= \mathcal{N}(0, {\sigma^2}_{t\_n})$
    $o = s_{t+1}$
  return o

function REWARD_FUNCTION(s, a):
  $r = 0$
  if irrelevant features **then**
    $s = s$[`relevant_dimensions`] ▷ Select the part of state space relevant to reward
  if discrete environment **then**
    if not `make_denser` **then**
      if state sequence ss of n states ending d steps in the past is in `rewardable_sequences` **then**
        r = `rewardable_sequences`[ss]
    else
      for i in range(n) **do**
        if sequence of i states ending d steps in the past is a prefix sub-sequence of a sequence in `rewardable_sequences` **then**
          $r \mathrel{+}=$ i/n
  else
    r = Distance moved towards the `target_point`
  $r \mathrel{+}= \mathcal{N}(0,\ {\sigma^2}_{r\_n})$
  $r \mathrel{*}=$ `reward_scale`
  $r \mathrel{+}=$ `reward_shift`
  if reached terminal state **then**
    $r \mathrel{+}=$ `term_state_reward`
  return r

function MAIN():
  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()

D More on Related Work

Many of the other benchmarks mentioned in the main paper are largely vision-based, which means that a large part of their problem solving receives benefits from advances in the vision community while our benchmarks try to tackle pure RL problems in their most toy form. This also means that our experiments are extremely cheap, making them a good platform to test out new algorithms' robustness to different challenges in RL.

A parallel and independent work along similar lines as the MDP Playground, which was released a month before ours on arXiv, is the Behaviour Suite for RL (bsuite, [42]). In contrast to our *generated* benchmarks, 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. Unlike their framework, where currently there is no toy environment for Hierarchical RL (HRL) algorithms, the rewardable sequences that we describe also fits very well with HRL. Additionally, we also have toy continuous environments whereas bsuite currently only has discrete environments. They also do not generate completely random P and R for their environments like we do, which would help avoid algorithms overfitting to certain benchmarks. 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.

[36] defines a novel theoretical metric for defining hardness of MDPs. It captures difficulties within MDPs when the true state of the MDP is known. However, a large part of the hardness in our MDPs comes from the agent not knowing the optimal information state to use. It'd be interesting to design a metric which captures this aspect of hardness as well.

Our platform allows formulating problems in terms of the identified dimensions and we feel this is a very human-understandable way of defining problems or specifying tasks. [35] defines a Geometric Linear Temporal Logic (GLTL) specification language to formally specify tasks for MDPs and RL environments. They also share our motivation in making it easier and more natural to specify tasks.

For some readers, it might feel obvious that injecting many of these dimensions causes difficulties for agents. However, to the best of our knowledge, no other work has tried to collect all *orthogonal* dimensions in one place and study them comprehensively and what aspects transfer from toy to more complex environments.

The nature of the toy environments is one of high bias. We believe that the *transfer* of the hardness dimensions from toy to complex environments occurs because the algorithms we have tested are environment agnostic and usually do not take aspects of the environment into account. Q-learning for instance is based on TD-errors and the Bellman equation. The equation is agnostic to the environment and while adding deep learning may help agents learn representations better, it does not remove the problems inherent in deep learning. While it's nice to have general algorithms that may be applied in a black box fashion, by studying the dimensions we have listed and their effects on environments, we gain deeper insights into what is needed to design better agents.

An additional comment can be made about comparing the continuous and discrete complex environments comparisons to the toy benchmarks. The "noise" in comparing the toy and complex discrete environments was higher compared to the continuous toy and complex environments and we believe this is due to the discrete environments being much more sparse and having many more *lucky areas* that can be exploited as with the *qbert* bug and *breakout* strategy mentioned. In comparison, continuous environments usually employ a dense reward formulation in which case the value functions are likely to be continuous.

Algorithms like DQN [38] have been 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 some of these environments, e.g. Montezuma's revenge, we need a very specific sequence of actions to get a reward. For others, there are different delays in rewards. A problem with evaluating on these environments is that we have either no control over their difficulty or little control such as having different difficulty levels. But even these difficulty levels, do not isolate the confounding factors that are present at the same time and do not allow us to control the confounding factors *individually*. We make that possible with our dimensions.

MDP Playground in relation to MNIST MNIST [32] captured some key difficulties required for computer vision (CV) which made it a good testbed for designing and debugging CV algorithms - even the webpage for the dataset mentions some distortions to inject hardness for MNIST: *distortions are random combinations of shifts, scaling, skewing, and compression*. [39] captures 15 such distortions to benchmark out-of-distribution robustness in computer vision. However, being a good testbed does not mean that MNIST can be used to directly learn models for much more specific CV applications such as classification of plants or medical image analysis. It captures many aspects that are general to CV problems but not specific ones.

When designing the platform, we went over the components of an MDP and tried to exhaustively add as many parameterisable dimensions as possible with the condition that they are all *orthogonal* and can be applied independently of each other. In a sense, this is an attempt to capture fundamental dimensions of hardness in the same way that human cognition is founded, in part, on four different systems and endow humans with abstract reasoning abilities [50]. We don't try to capture, say credit assignment or generalisation as dimensions. These are to be dealt with at a higher level the same way that intelligent behaviour and reasoning arise from the interplay of different underlying cognitive systems which process objects or space at a lower sensory level.

E More on Designing New agents

Varying *action range* Since the insights into the environments for this dimension were similar to the insights for the *time unit*, the design ideas for an agent robust to this dimension follow a similar vein as for the *time unit*. An ideal adaptive agent design would set the *action range* in an online manner. A new basic agent design could do this once at the beginning of training and set its optimal *action range*.

Varying *reward delay* Since it's clear (see Figure 12 and discussion in Appendix J) that not having the Markov state as the information state can lead to a significant drop in performance, a simple tabular agent design could incorporate delays into its formulation. For instance, one could formulate the Q-value as being over multiple possible previous states and actions and then take the estimate of delay to be the value for which the Q-values are maximised for different state-action pairs.

Varying *transition noise* Noise also has an adverse effect on the performance of agents (see Figure 13 and discussion in Appendix J). A simple model-based RL agent design that learns a probabilistic model could adaptively estimate the noise in the transitions by repeatedly measuring the same state-action pair's transitions. This would give it an estimate of the aleatoric uncertainty. The agent could then choose to stop learning its dynamics model once the uncertainty in its model is close to the estimated aleatoric uncertainty. This would save it from further computational expenses.

F More on Debugging Agents

We discuss here further the 2 examples of how the toy environments helped us debug RL algorithms in practice.

When merging some of our environments into bsuite, we noticed that when we varied sparsity, the performance of their DQN agent would go down in proportion to the environment's sparsity. This did not occur for other agents. This seemed to suggest something different for the DQN agent and when we looked at DQN's hyperparameters we realised that it had a fixed ϵ schedule. While that may be desirable in some situations, we felt it hurt DQN's performance because it was not allowed to explore enough early on nor exploit what it learnt fully later. When we use regular structured environments, the agent performances are freed of the "noise" that is present due to irregular transition functions and this makes it easy to see high-level trends.

When we were performing the complex environment experiments for Atari (using Ray 0.9.0 as explained in Appendix P), we noticed that there was no learning for Rainbow even though DQN learned. We debugged this by ablating the various additions to Rainbow when compared to DQN. We ran these ablations on the *image representations* toy environment of *MDP Playground* and observed (see Figure 5) that all ablations, apart from turning noisy nets off, performed poorly. This let us quickly debug that noisy nets was broken in Ray 0.9.0.

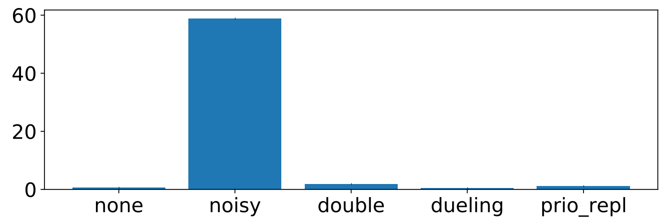


Figure 5: Ablations of Rainbow on *image representations* environments. Note the different y-axis scales.

Another example of how hard it can be to debug RL agents can be found in this GitHub issue for bsuite: `https://github.com/deepmind/bsuite/issues/20`

G Design Decisions

Discrete environment generation Once the values for the dimensions are set, for the case of auto-generated discrete environments, P is generated by selecting for each state s in independent set i, for each action a, a random successor state s' from independent set $i + 1$. This results in a *regular* grid-world like structure for P. For R, we select the num_r rewardable sequences randomly for a given reward density rd based on all the sequences possible under the generated P. The main reason for unit testing in this manner is that all the RL agents we are aware of do not themselves take the structure of the environment into account and are designed for general Ps and Rs. Because of this, once the toy environment's dimensions are set, the structure of the environment is set and the agents should show similar behaviour on all such environments and this is exactly what we observed in our experiments when run with different seeds for the environment generation.

A second reason a regular structure is imposed on P is that it is always possible to design *adversarial* Ps [40, 47] which can be made arbitrarily hard to solve. Suppose there is an environment where a large reward is placed in an unknown and deliberately unexpected location. Then, evaluating an agent on such an environment clearly does not give us a proper measure of the agent's performance. This is, in some cases, also a problem with many complex environments, e.g., *HalfCheetah* has a bug that allows the agent to reach infinite speed and obtain enormous rewards [63]. *qbert* has a bug which allows the agent to achieve a very large number of points [9]. *breakout* has a scenario where, if an agent creates a hole through the bricks, it can achieve a very large number of points. Even though the latter *can* be a sign of desired behaviour, it skews the distribution of rewards and introduces variance in the evaluation. There is the additional danger that the blackbox nature of complex environments can lead researchers to draw inferences that may be biased by their intuition [27]. For example, the agent strategy of creating a tunnel to target bricks in the top for *breakout* has been challenged multiple times [5, 58]. As [21] sums it up: *If my reinforcement learning code does no better than random, I have no idea if it's a bug, if my hyperparameters are bad, or if I simply got unlucky.* Thus, having a very complex P or R itself can introduce "noise" into the evaluation of agents and require many iterations of training before we can see the agent learning. We leave this for complex benchmarks to capture as they are closer to real world use cases. For unit testing, especially, one needs quick insights on vanilla environments and thus, it is beneficial to have what we term high *bias* environments to test whether agents are learning.

The third reason a *more* regular structure is imposed as opposed to the usual gridworld is that, though, semantically meaningful, such gridworlds have small irregularities around edges which makes them hard to keep consistent with all the other dimensions and begins to introduce the kind of "noise" that was discussed for more complex environments above.

Dimensions While the aim is to be as objective as possible while selecting the dimensions, a few subjective ones were included such as *action loss weight*, which penalises action magnitudes in continuous environments and is a very common use-case [8].

As some subjective design decisions were imposed on the auto-generated environments, users can also define their own Ps and Rs, e.g., as transition matrices or Python functions. However, it is important to note here that the user does *not* need to take care of injecting the dimensions in this case, as these are handled by *MDP Playground*, wherever possible.

H Effect of dimensions on more complex benchmarks

Table 1: Spearman Rank Correlations for performance on toy and complex environments across different amounts of the dimension injected: **transition noise**

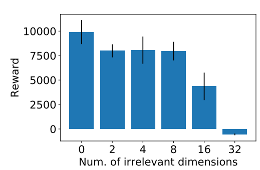
Environment/Agent	DQN	Rainbow	A3C
beam_rider	r=0.9, pvalue=0.037	r=0.7, pvalue=0.188	r=0.9, pvalue=0.037
breakout	r=1.0, pvalue=1e-24	r=1.0, pvalue=1e-24	r=0.9, pvalue=0.037
qbert	r=0.9, pvalue=0.037	r=0.9, pvalue=0.037	r=0.7, pvalue=0.188
space_invaders	r=0.9, pvalue=0.037	r=0.7, pvalue=0.188	r=0.9, pvalue=0.037

Table 2: Spearman Rank Correlations for performance on toy and complex environments across different amounts of the dimension injected: **reward delay**

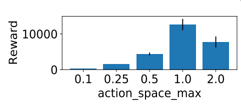
Environment/Agent	DQN	Rainbow	A3C
beam_rider	r=0.8, pvalue=0.104	r=0.4, pvalue=0.504	r=0.6, pvalue=0.284
breakout	r=0.0, pvalue=1.0	r=0.3, pvalue=0.623	r=0.8, pvalue=0.104
qbert	r=0.6, pvalue=0.284	r=0.4, pvalue=0.504	r=0.9, pvalue=0.037
space_invaders	r=0.9, pvalue=0.037	r=0.9, pvalue=0.037	r=0.8, pvalue=0.104

Table 3: Spearman Rank Correlations for performance on toy and complex environments across different amounts of the dimension injected: **reward noise**

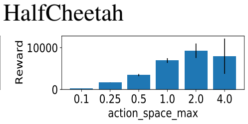
Environment/Agent	DQN	Rainbow	A3C
beam_rider	r=0.9, pvalue=0.037	r=0.9, pvalue=0.037	r=0.7, pvalue=0.188
breakout	r=0.8, pvalue=0.104	r=0.9, pvalue=0.037	r=0.9, pvalue=0.037
qbert	r=0.4, pvalue=0.504	r=0.5, pvalue=0.391	r=0.5, pvalue=0.391
space_invaders	r=0.9, pvalue=0.037	r=0.7, pvalue=0.188	r=0.7, pvalue=0.188



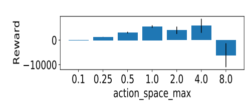
(a) SAC irr. dims. HalfCheetah



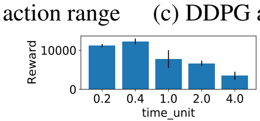
(b) SAC action range



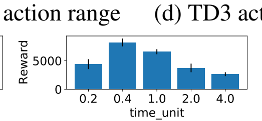
(c) DDPG action range



(d) TD3 action range

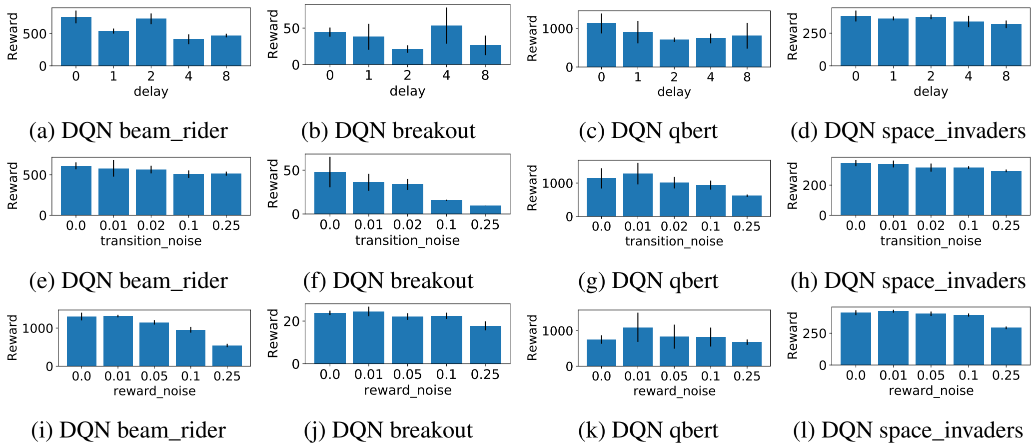


(e) SAC time unit



(f) TD3 time unit

Figure 9: AUC of episodic reward at the end of training on HalfCheetah **varying action range or time unit**. Error bars represent 1 standard deviation. Note the different y-axis scales.

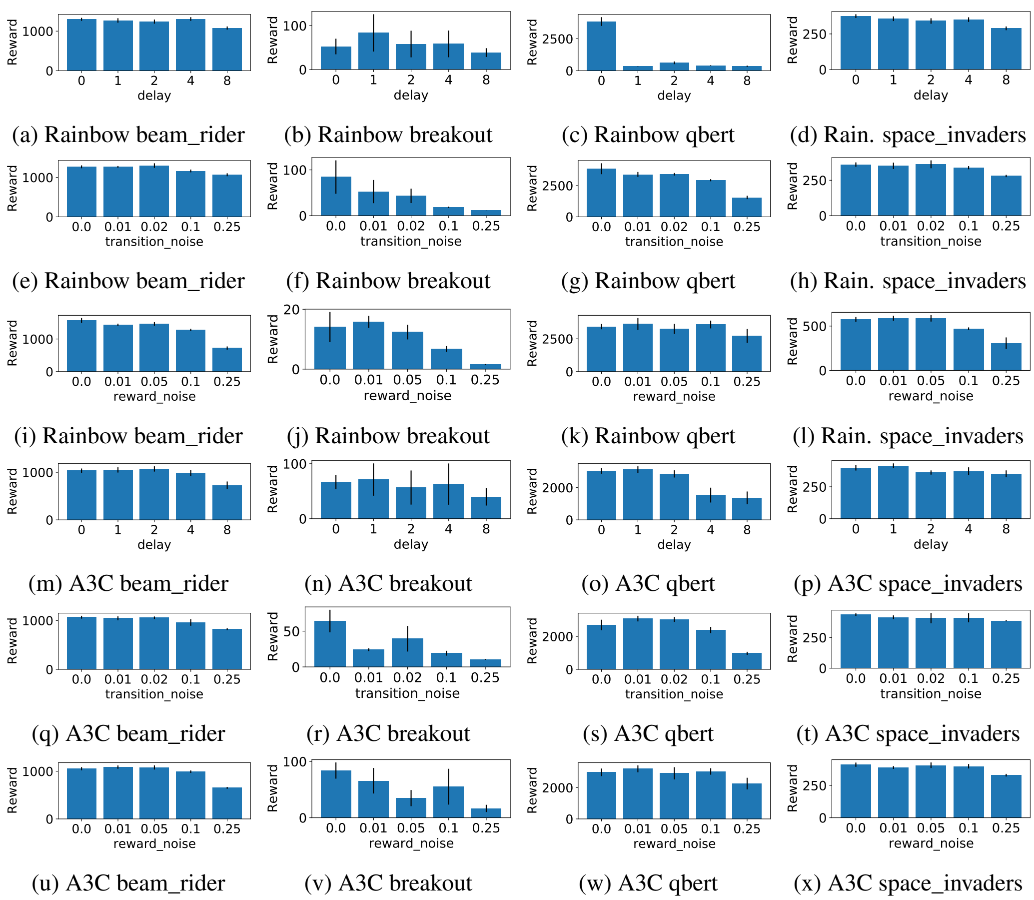


(a) DQN beam_rider (b) DQN breakout (c) DQN qbert (d) DQN space_invaders

(e) DQN beam_rider (f) DQN breakout (g) DQN qbert (h) DQN space_invaders

(i) DQN beam_rider (j) DQN breakout (k) DQN qbert (l) DQN space_invaders

Figure 6: AUC of episodic reward for DQN on various environments at the end of training. Error bars represent 1 standard deviation. Note the different y-axis scales.



(a) Rainbow beam_rider (b) Rainbow breakout (c) Rainbow qbert (d) Rain. space_invaders

(e) Rainbow beam_rider (f) Rainbow breakout (g) Rainbow qbert (h) Rain. space_invaders

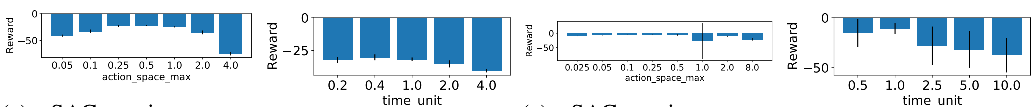
(i) Rainbow beam_rider (j) Rainbow breakout (k) Rainbow qbert (l) Rain. space_invaders

(m) A3C beam_rider (n) A3C breakout (o) A3C qbert (p) A3C space_invaders

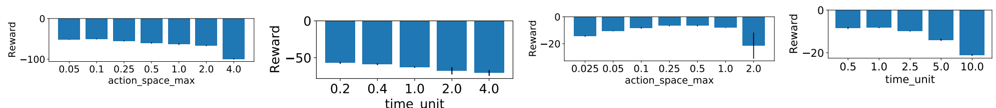
(q) A3C beam_rider (r) A3C breakout (s) A3C qbert (t) A3C space_invaders

(u) A3C beam_rider (v) A3C breakout (w) A3C qbert (x) A3C space_invaders

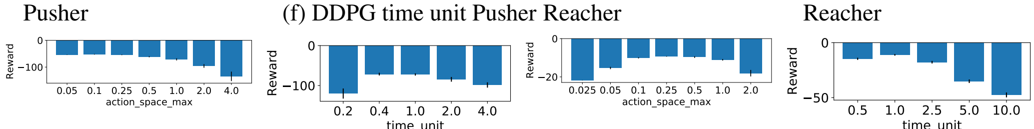
Figure 7: AUC of episodic reward for agents at the end of training for A3C and Rainbow. Error bars represent 1 standard deviation. Note the different y-axis scales.



(a) SAC action range Pusher (b) SAC time unit Pusher (c) SAC action range Reacher (d) SAC time unit Reacher



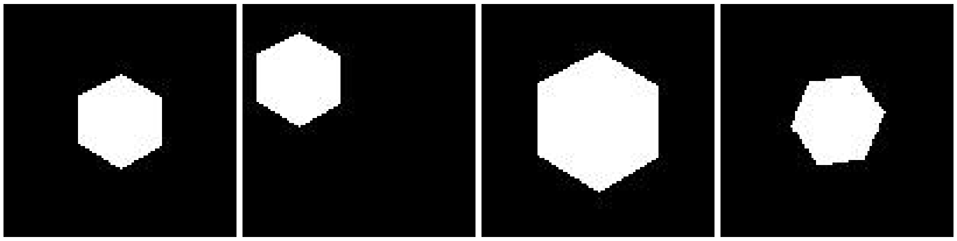
(e) DDPG action range Pusher (f) DDPG time unit Pusher (g) DDPG action range Reacher (h) DDPG time unit Reacher



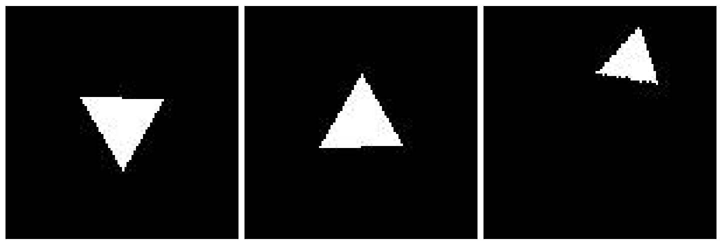
(i) TD3 action range Pusher (j) TD3 time unit Pusher (k) TD3 action range Reacher (l) TD3 time unit Reacher

Figure 8: AUC of episodic reward at the end of training on Pusher and Reacher environments **varying action max** and **time unit**. Error bars represent 1 standard deviation. Note the different y-axis scales.

I Sample states used for Representation Learning



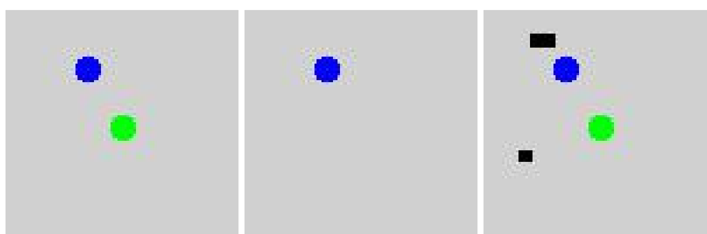
(a) No transforms (b) Shift (c) Scale (d) Rotate



(e) No transforms (f) Flip (g) All transforms

Figure 10: When using the dimension *representation learning* in discrete environments, each categorical state corresponds to an image of a polygon (if the states were numbered beginning from 0, each state n corresponds to a polygon with $n + 3$ sides). Various transforms can be applied to the polygons randomly at each time step. Samples shown correspond to states 3 and 0

.



(a) Cont. env. (b) Cont. env. (c) Cont. env.

Figure 11: When using the dimension *representation learning* in continuous environments, the agent is shown as a blue circle, the target point as a green circle and terminal states are black

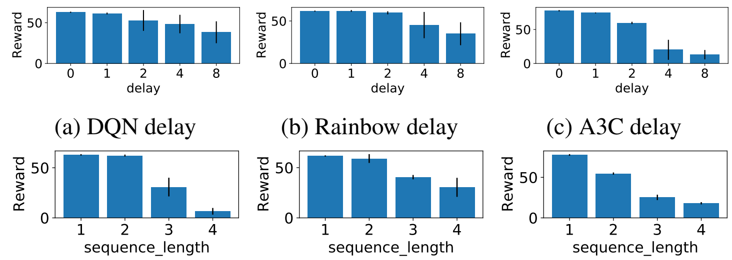
.

J More Experiments and Additional Reward Plots

We continue with the experiments and results from the main paper here.

J.1 Discrete environments

Varying *reward delay* Figures 12a-c, depict the mean and standard deviation over 10 runs for various *delays*. One run consists of 10 random seeds for the algorithm but uses a fixed seed for the environment. We plot the Area Under the Curve (AUC) which takes the mean over previous training rewards. As can be seen from the figure, all algorithms perform very well in the *vanilla environment* where the MDP is fully observable as the information state of the agent is equal to the MDP's state. For all algorithms, performance degrades in environments where the information state is non-Markov. Performance clearly degrades more as the information state needed to compute the optimal policy requires more observations to be stacked. It is interesting (and expected) that Rainbow DQN is somewhat more robust than DQN. The plots also show that DQN variants are more robust to *delay* as compared to A3C variants.



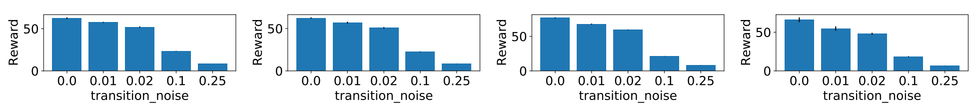
(a) DQN delay (b) Rainbow delay (c) A3C delay

(d) DQN seq len (e) Rainbow seq len (f) A3C seq len

Figure 12: AUC of episodic reward at the end of training for different agents for varying **delays** (top) and **sequence lengths** (bottom). Error bars represent 1 standard deviation. Note the reward scales.

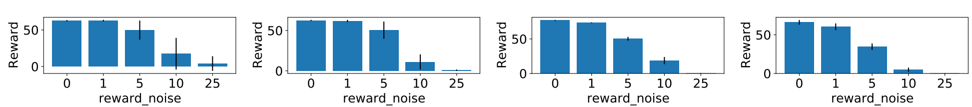
Varying *rewardable sequence length* Results here are qualitatively similar to the ones for delay. However, we observe in Figures 12d-f that sequence length has a more drastic effect in terms of degradation of performance. The improvements of Rainbow DQN over DQN are also more pronounced for these harder problems.

Results for varying transition and reward noises



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

Figure 13: Mean episodic reward at the end of training for the different algorithms **when varying transition noise**. Error bars represent 1 standard deviation. Note the different reward scales.

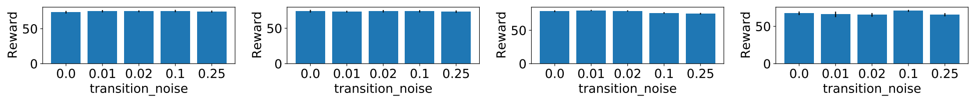


(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 14: Mean episodic reward at the end of training for the different algorithms **when varying reward noise**. Error bars represent 1 standard deviation. Note the different reward scales.

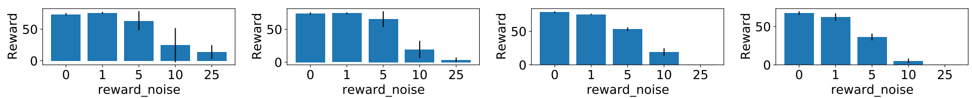
We see a similar trend during *training*, as for delays and sequences, when we vary the *transition noise* in Figure 13 and the *reward noise* in Figure 14. Performance degrades gradually as more and more noise is injected. It is interesting that, during training, all the algorithms seem to be more sensitive 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 scale* being 1.0) reward noise standard deviation of $\sigma_{r\_n} = 1$ still resulted in learning progress.



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

Figure 15: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying transition noise**. Error bars represent 1 standard deviation.



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

Figure 16: Mean episodic reward for evaluation rollouts (max 100 timesteps) at the end of training for the different algorithms **when varying reward noise**. Error bars represent 1 standard deviation. Note the different reward scales.

Interestingly, when we plot the *evaluation* performances[1] in Figures 15 and 16, we see, on comparing with the training plots, that the training performance of the algorithms is more sensitive to noise in the transition dynamics (Figure 13) than the eventual evaluation performance is (Figure 15). While it is obvious that the mean episodic reward during training would be perturbed when noise is injected into the *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 variants especially (and DQN to a small extent), suggest that it performs *better* in the presence of transition noise. This might indicate that A3C in the presence of no transition noise does not explore enough (as was also conjectured in the unexpected results for varying the sparsity meta-feature) and is actually helped when transition noise is present during training.

J.2 Continuous Environments

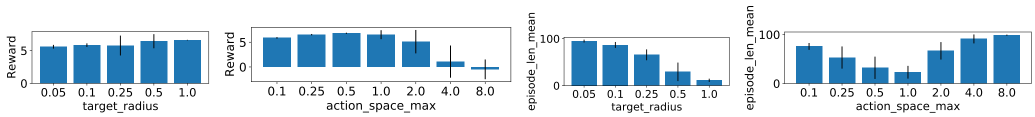
We set the state and action space dimensionalities to 2. The state space range for each dimension was $[-10, 10]$ while the default action space range was $[-1, 1]$. The task would terminate when an algorithm would reach the *target point*, or after at most 100 timesteps. We focus on results for DDPG as results for TD3 and SAC are qualitatively similar (see Appendix J).

Varying *action range* We observed that the total reward gets worse for action *max* > 1. Up until the value of 1, the episode lengths decreased as we would desire (see Figures 17b & 17d). This can be attributed to the fact that the exploration schedules for the studied agents take the max range available and explore based on that. But, as can be seen from these results, tuning these ranges or adapting exploration mechanisms can produce substantially better results.

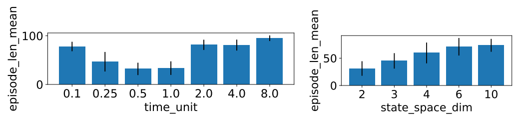
Varying *target radius* The *target radius* is a value which is generally set to a small enough value to be able to say that the algorithm has reached the target. However, we noticed that, for small values, all the continuous control agents oscillated around the target to reach it exactly. This can be observed in Figure 17a and 17c, where we note that even though the task *was* learnt for different *target radii*, the episode lengths were shorter for larger radii as the agents kept oscillating outside the radius. Even for such a simple task all evaluated algorithms failed to adapt to performing fine-grained control near the target. We hypothesise that the agents did not learn to slow down close to the goal. Given more experience close to the goal, we expect the agents to be able to learn this behaviour.

As we mentioned earlier, one of the advantages of our platform is that it allows us to introduce all the hardness dimensions on the same base environment at the same time. This is helpful to understand

[1]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.



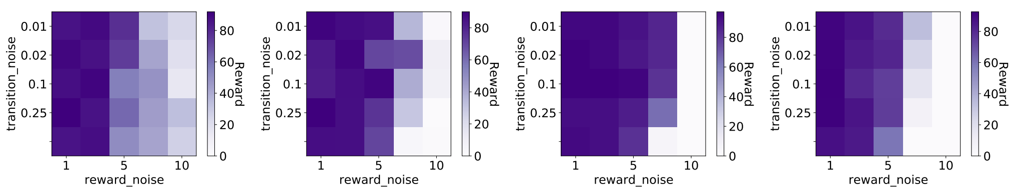
(a) target radius rew. (b) action range rew. (c) target radius len. (d) action range len.



(e) time unit len. (f) irr. dims. len.

Figure 17: AUC of episodic reward (top) and lengths (bottom) for DDPG at the end of training. Error bars represent 1 standard deviation. Note the different y-axis scales.

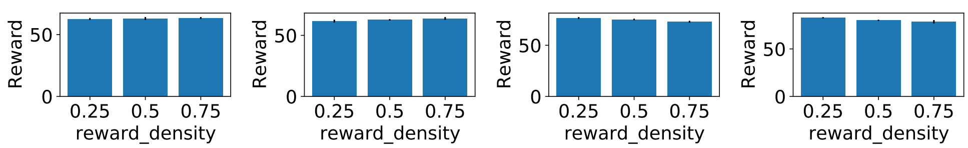
interaction effects between them. We plot the most interesting interaction effects in Figure 18 where we varied both transition and reward noise over respective grid values. This plot shows that our observation, that transition noise helps A3C out during *evaluation*, is only clearly valid when the reward noise is not so high ($\sigma_{r\_n} <= 1$) as to disrupt training. The corresponding heatmap plot for training when varying the noises and additional ones for jointly varying delay and rewardable sequence length are present in the Appendix (Figures 37 - 39).



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

Figure 18: Mean episodic reward for evaluation rollouts (max 100 timesteps) at the end of training for the different algorithms **when varying transition and reward noise**.

J.3 Results for varying reward sparsity



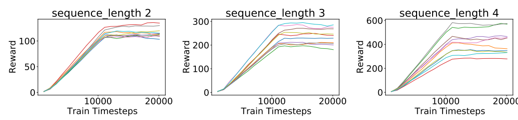
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 19: Mean episodic reward at the end of training for the algorithms **when varying reward sparsity**. Error bars represent 1 standard deviation. Note the different reward scales.

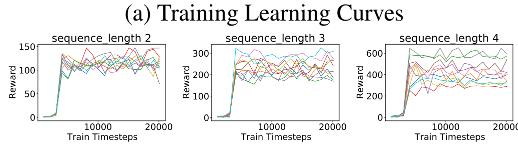
Figure 19 shows the results of controlling the meta-feature *sparsity* in the environment. The DQN variants were able to learn the important rewarding states in the environment even when these were sparse while the behaviour of A3C was unexpected. One explanation could be that A3C's exploration was not very good, in which case increasing reward density would help as in Figure 19c. But adding in an LSTM to the A3C agent seems to show the opposite trend (Figure 19d) as increasing reward density leads to worsening performance. This could indicate 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.

We have observed A3C is more variant in general than its DQN counterparts and this should be expected as it launches and collects data from several instances of the same environment which induces more variance.

The $make\_denser$ configuration option, makes learning smoother and less variant across different runs of an algorithm, as can be seen in Figure 20a for DQN when compared to Figure 22 for corresponding sequence lengths. To evaluate the *true* learning of algorithms, we turn off the $make\_denser$ option in the evaluation rollouts. The learning curves for these can be seen for DQN in Figure 20b. The agent still does not perform as well as might be expected when making the reward signal denser during training. This is probably due to the sequence lengths still *violating* the complete observability assumption made by the algorithm. The plots for learning curves for the remaining algorithms are present in Figures 70-75. The plots for final mean reward during training and evaluation are given in Figures 19 and 26.



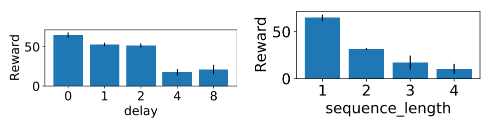
(a) Training Learning Curves



(b) Evaluation Learning Curves

Figure 20: Learning curves for DQN **when *make_denser* is *True* for rewardable sequences**. Please note the different Y-axis scales and the fact that with longer rewardable sequences, a greater number of seeds do not learn anything for the evaluation rollouts (reward ≈ 0).

J.4 Further results for varying reward delays and sequences



(a) A3C + LSTM delay (b) A3C + LSTM seq len

Figure 21: Mean episodic reward at the end of training for different agents for varying **delays** (top) and **sequence lengths** (bottom). Error bars represent 1 standard deviation. Note the reward scales.

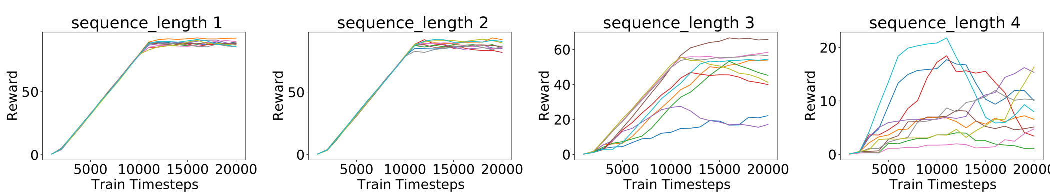


Figure 22: Train Learning Curves for 10 runs with different seeds for DQN **when varying sequence lengths**. Please note that each different colour corresponds to one of 10 seeds in each subplot.

Note that we varied delay on a logarithmic scale and sequence length on a linear one which means that this effect is more pronounced than may first appear when looking at the figures. We additionally also plot the *learning curves*, when varying sequence lengths, in Figure 22. We see how training proceeds much more smoothly and is less variant across different seeds for the vanilla environment (where the sequence length is 1) and that the variance across seeds is very large for sequence length 3.

J.5 Selecting Total Timesteps for Runs

We ran the experiments and plot the results for DQN variants up to $20\,000$ environment timesteps and the ones for A3C variants up to $150\,000$ time steps since A3C took longer[2] to learn as can be

[2]In terms of environment steps. Wallclock time used was still similar.

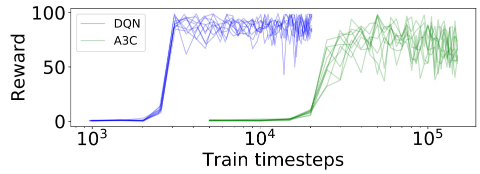
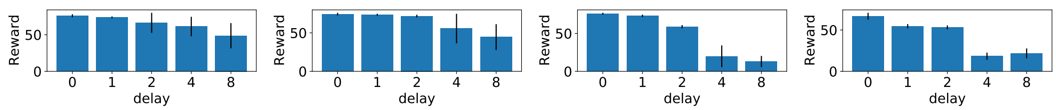


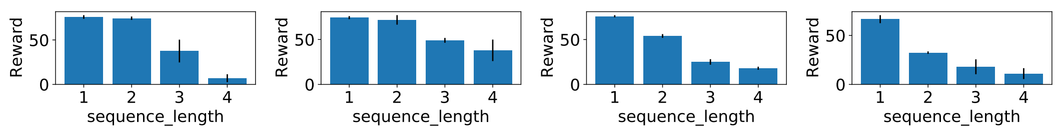
Figure 23: Evaluation rollouts (limited to 100 timesteps per episode) for DQN and A3C in the vanilla environment which shows that DQN learns faster than A3C in terms of the number of timesteps.

seen in Figure 23. We refrain from fixing a single number of timesteps for our environments (as, e.g., bsuite does), since the study of different trends for different families of algorithms will require different numbers of timesteps. Policy gradient methods such as A3C are slower in general compared to value-based approaches such as DQN. Throughout, we always run 10 seeds of all algorithms to obtain reliable results. We repeated many of our experiments with an independent set of 10 seeds and obtained the same qualitative results.



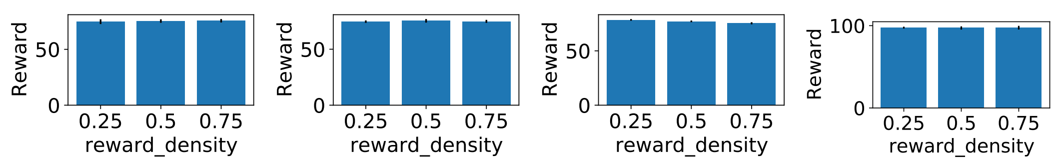
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 24: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying delay**. Error bars represent 1 standard deviation.



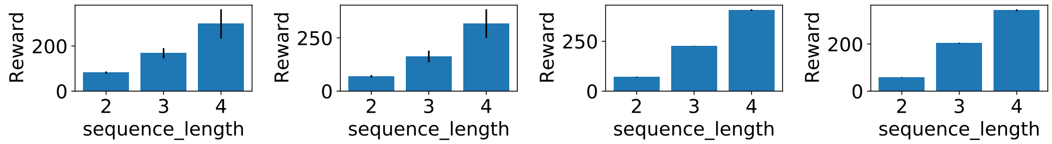
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 25: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying sequence lengths**. Error bars represent 1 standard deviation.



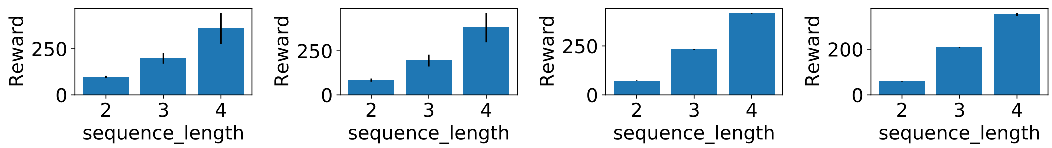
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 26: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying reward sparsity**. Error bars represent 1 standard deviation.



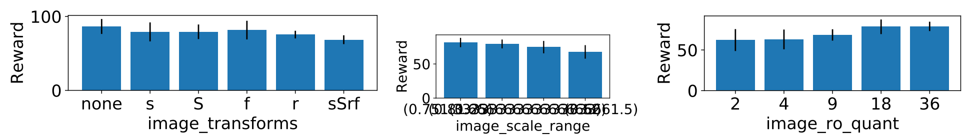
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 27: Mean episodic reward at the end of training for the different algorithms **when *make_denser* is *True* for rewardable sequences**. Error bars represent 1 standard deviation.



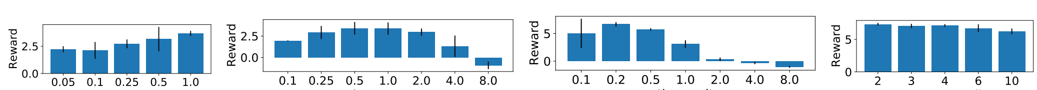
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 28: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when *make_denser* is *True* for rewardable sequences**. Error bars represent 1 standard deviation.

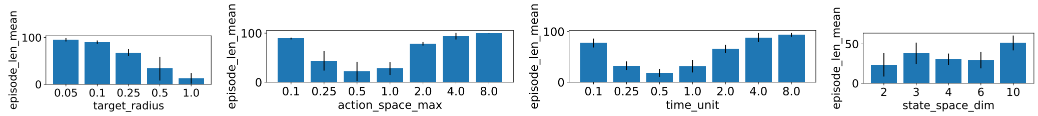


(a) A3C + LSTM (b) DQN scale (c) DQN rotate

Figure 29: Mean episodic reward at the end of training for the different algorithms **when varying representation learning**. ’s’ represents *shift*, ’S’ represents *scale*, ’f’ represents *flip* and ’r’ represents *rotate* in the labels in the first subfigure. *scale_range* represents *scaling* ranges in the second subfigure. *image_ro_quant* is represents quantisation of the *rotation*s in the third subfigure. Error bars represent 1 standard deviation.

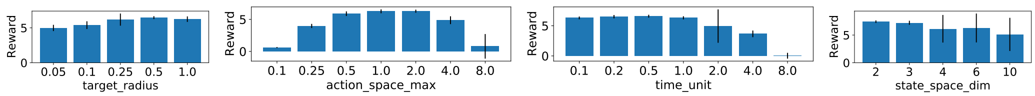


(a) target radius rew. (b) action max rew. (c) time unit rew. (d) irr. dims. rew.

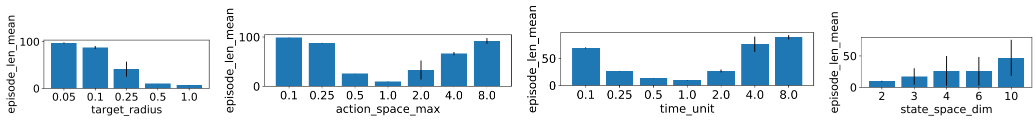


(e) target radius len. (f) action max len. (g) time unit len. (h) irr. dims. len.

Figure 30: Mean episodic reward (above) and lengths (below) for TD3 at the end of training. Error bars represent 1 standard deviation.



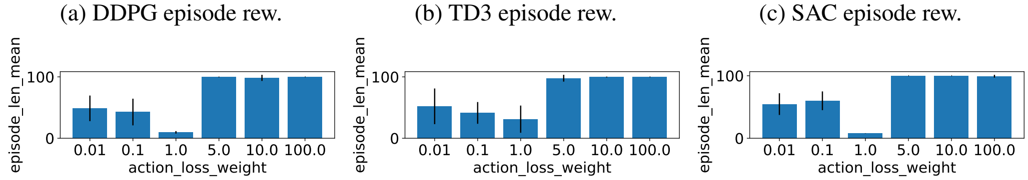
(a) target radius rew. (b) action max rew. (c) time unit rew. (d) irr. dims. rew.



(e) target radius len. (f) action max len. (g) time unit len. (h) irr. dims. len.

Figure 31: Mean episodic reward (above) and lengths (below) for SAC at the end of training. Error bars represent 1 standard deviation.

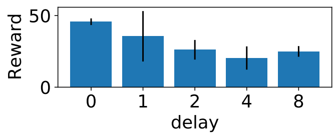
(a) DDPG episode rew. (b) TD3 episode rew. (c) SAC episode rew.



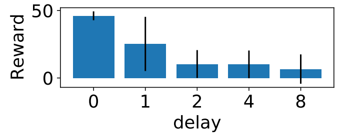
(d) DDPG episode len. (e) TD3 episode len. (f) SAC episode len.

Figure 32: Mean episodic reward (above) and lengths (below) at the end of training for evaluation rollouts for DDPG, TD3 and SAC when varying **action_loss_weight**. Error bars represent 1 standard deviation.

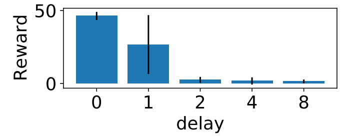
K Plots for tabular baselines



(a) Q-Learning

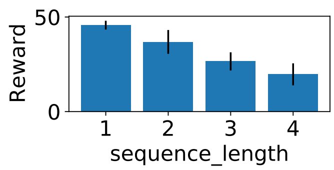


(b) SARSA

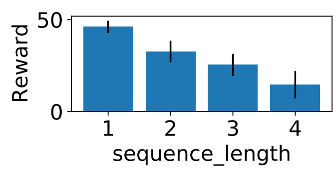


(c) Double Q-Learning

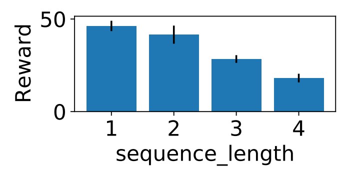
Figure 33: Mean episodic reward (limited to 100 timesteps) at the end of training for three different tabular baseline algorithms **when varying reward delay**. Error bars represent 1 standard deviation.



(a) Q-Learning

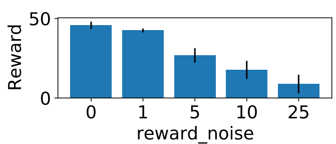


(b) SARSA

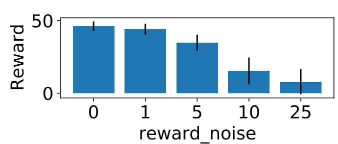


(c) Double Q-Learning

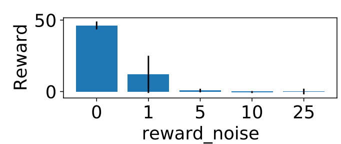
Figure 34: Mean episodic reward (limited to 100 timesteps) at the end of training for three different tabular baseline algorithms **when varying sequence length**. Error bars represent 1 standard deviation.



(a) Q-Learning

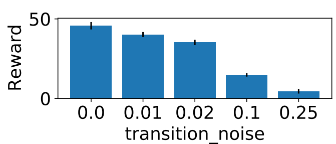


(b) SARSA

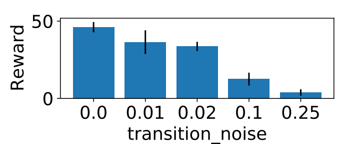


(c) Double Q-Learning

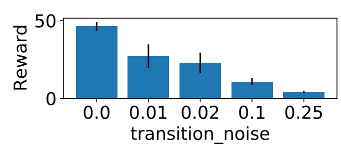
Figure 35: Mean episodic reward (limited to 100 timesteps) at the end of training for three different tabular baseline algorithms **when varying reward noise**. Error bars represent 1 standard deviation.



(a) Q-Learning



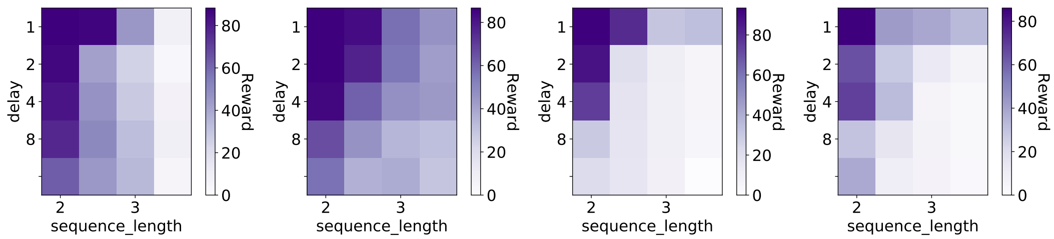
(b) SARSA



(c) Double Q-Learning

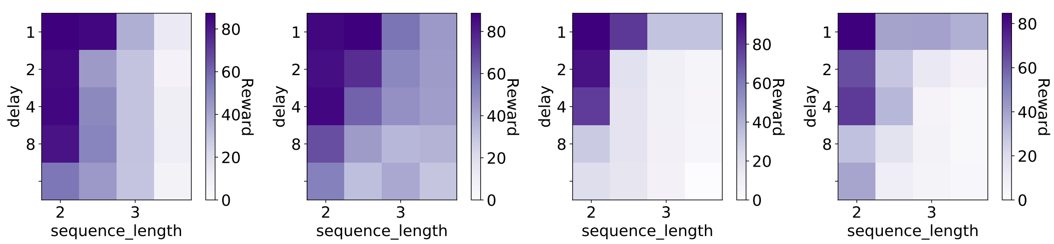
Figure 36: Mean episodic reward (limited to 100 timesteps) at the end of training for three different tabular baseline algorithms **when varying transition noise**. Error bars represent 1 standard deviation.

L Plots for varying 2 hardness dimensions together



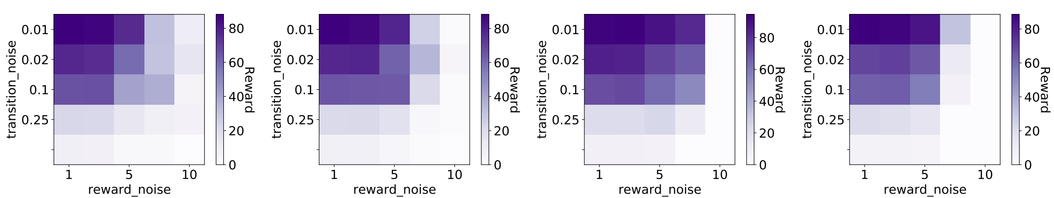
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 37: Mean episodic reward at the end of training for the different algorithms **when varying delay and sequence lengths**. Please note the different colorbar scales.



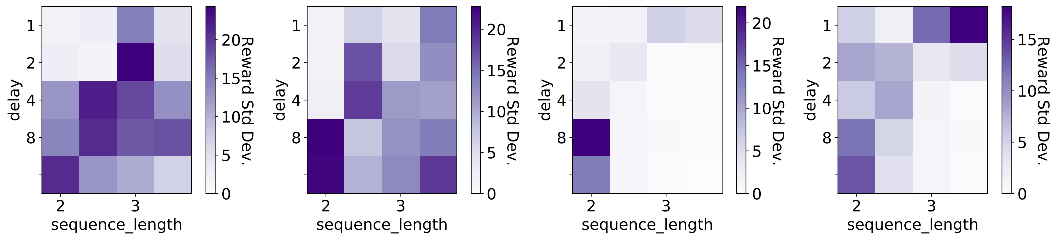
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 38: Mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying delay and rewardable sequence lengths**. Please note the different colorbar scales.



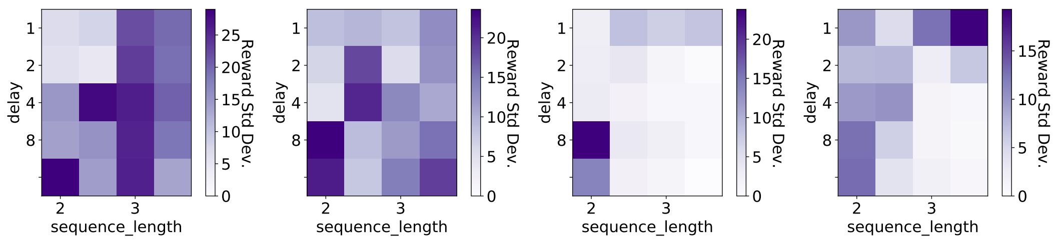
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 39: 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.



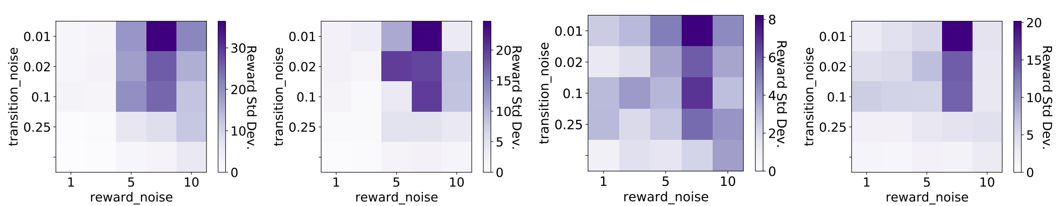
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 40: Standard deviation of mean episodic reward at the end of training for the different algorithms **when varying delay and sequence lengths**. Please note the different colorbar scales.



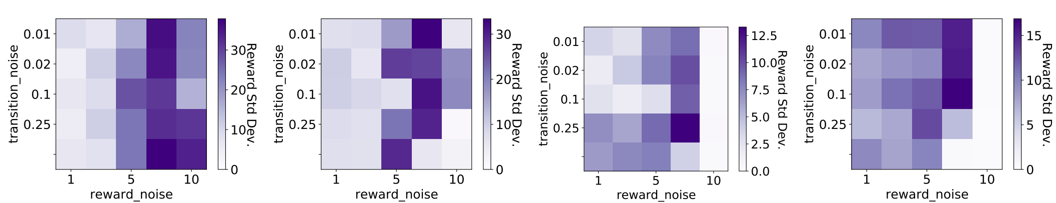
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 41: Standard deviation of mean episodic reward for evaluation rollouts (limited to 100 timesteps) at the end of training for the different algorithms **when varying delay and rewardable sequence lengths**. Please note the different colorbar scales.



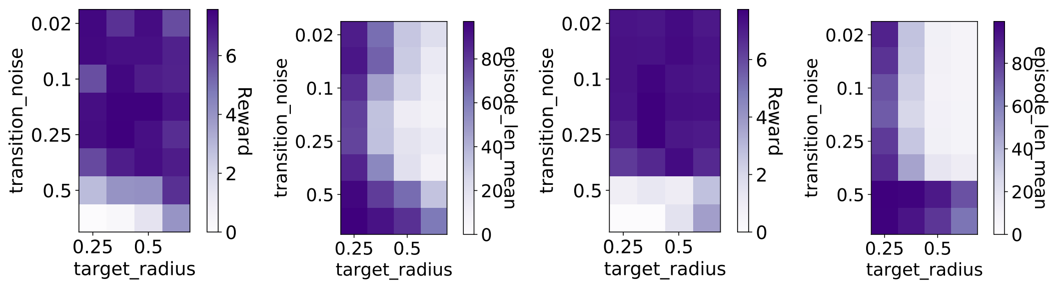
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 42: Standard deviation of 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.



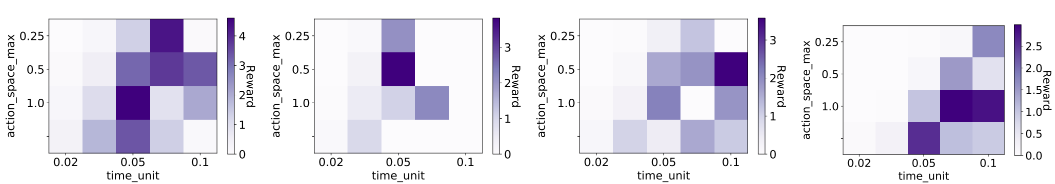
(a) DQN (b) Rainbow (c) A3C (d) A3C + LSTM

Figure 43: Standard deviation of mean episodic reward at the end of training 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.



(a) TD3 episode rew. (b) TD3 episode len. (c) SAC episode rew. (d) SAC episode len.

Figure 44: Mean episodic reward and lengths at the end of training for the different algorithms **when varying** P **noise and *target radius***.



(a) DDPG P Order 2 (b) DDPG P Order 3 (c) TD3 P Order 2 (d) SAC P Order 2

Figure 45: Mean episodic reward at the end of training for the different algorithms **when varying action space max and time unit for a given** P **order**.

M Additional Learning Curves

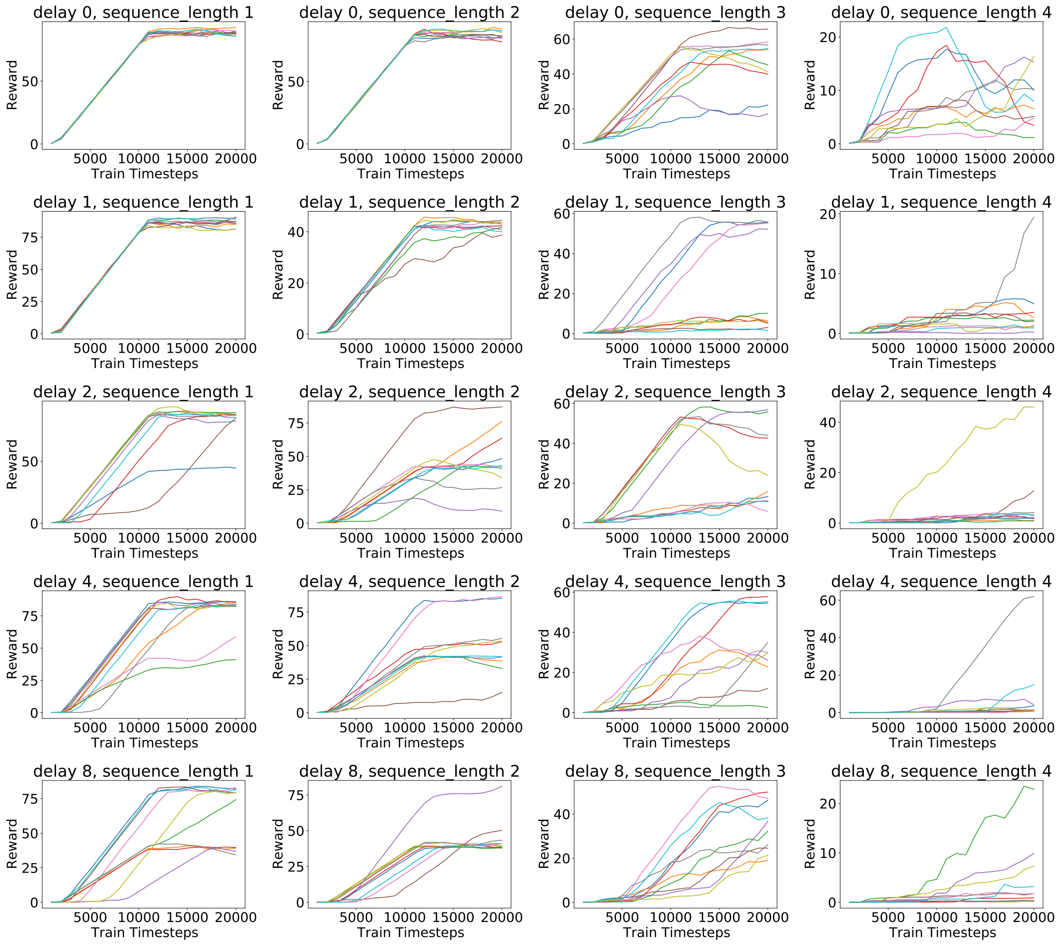


Figure 46: Training Learning Curves for DQN **when varying delay and sequence lengths**. Please note the different colorbar scales.

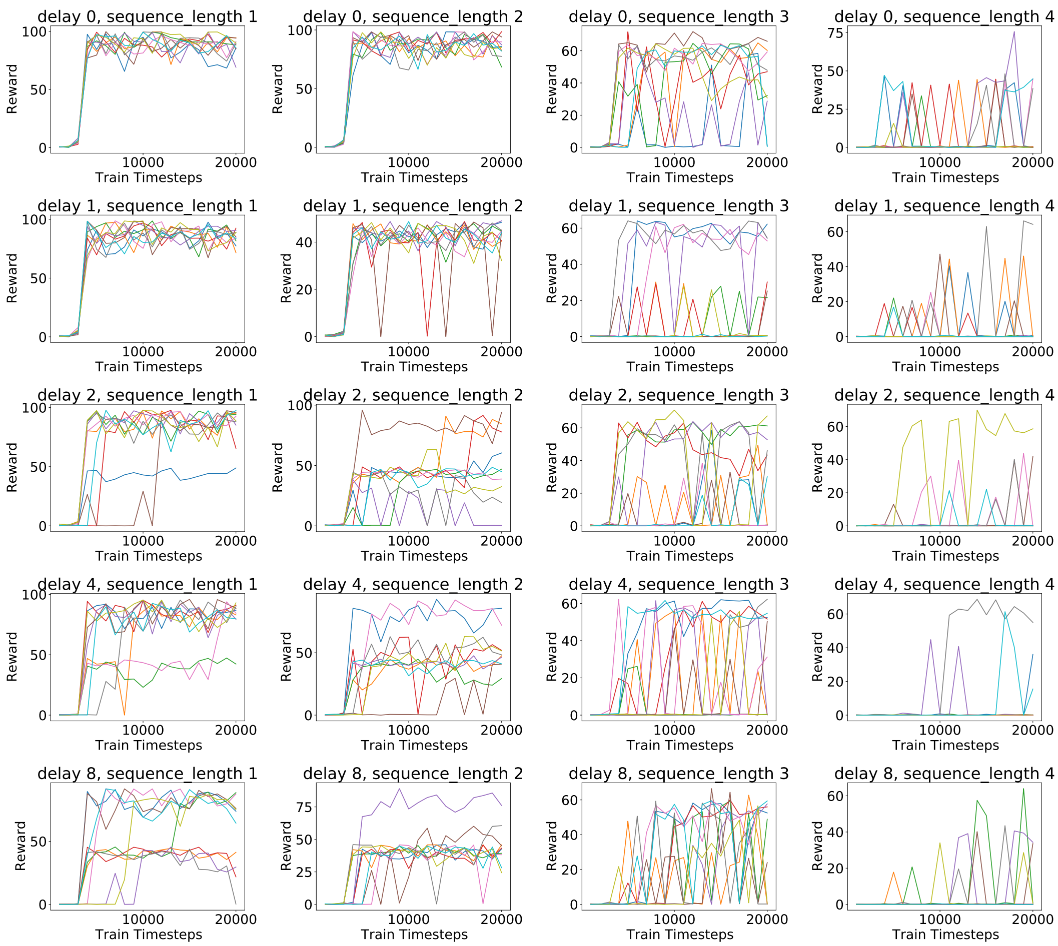


Figure 47: Evaluation Learning Curves for DQN **when varying delay and sequence lengths**. Please note the different colorbar scales.

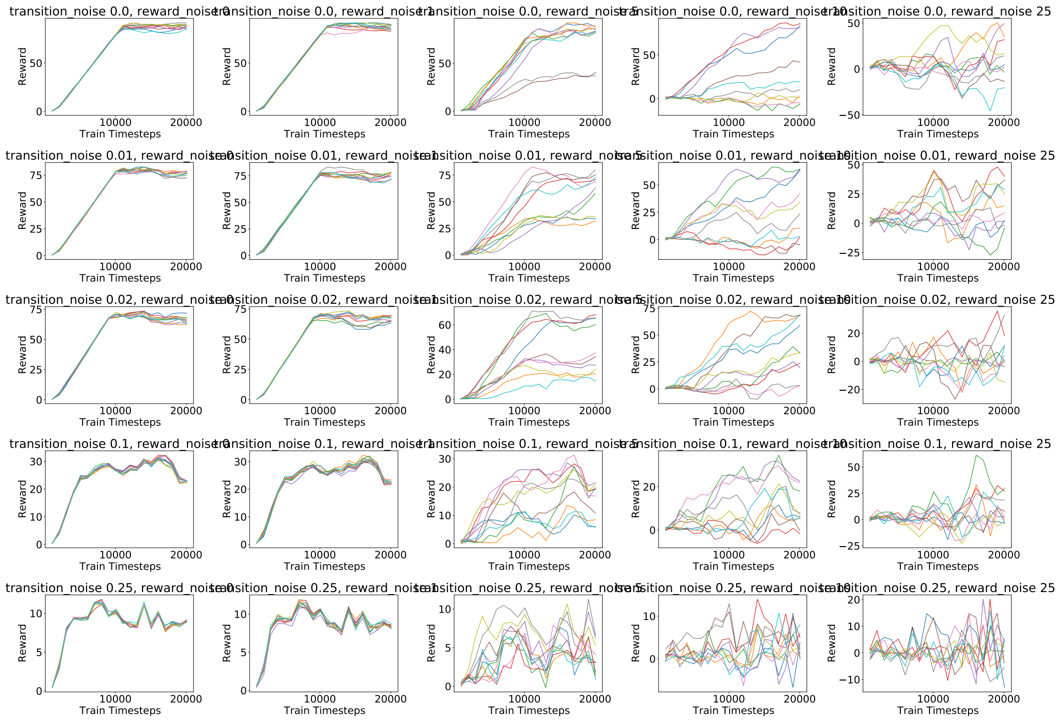


Figure 48: Training Learning Curves for DQN **when varying transition noise and reward noise**.

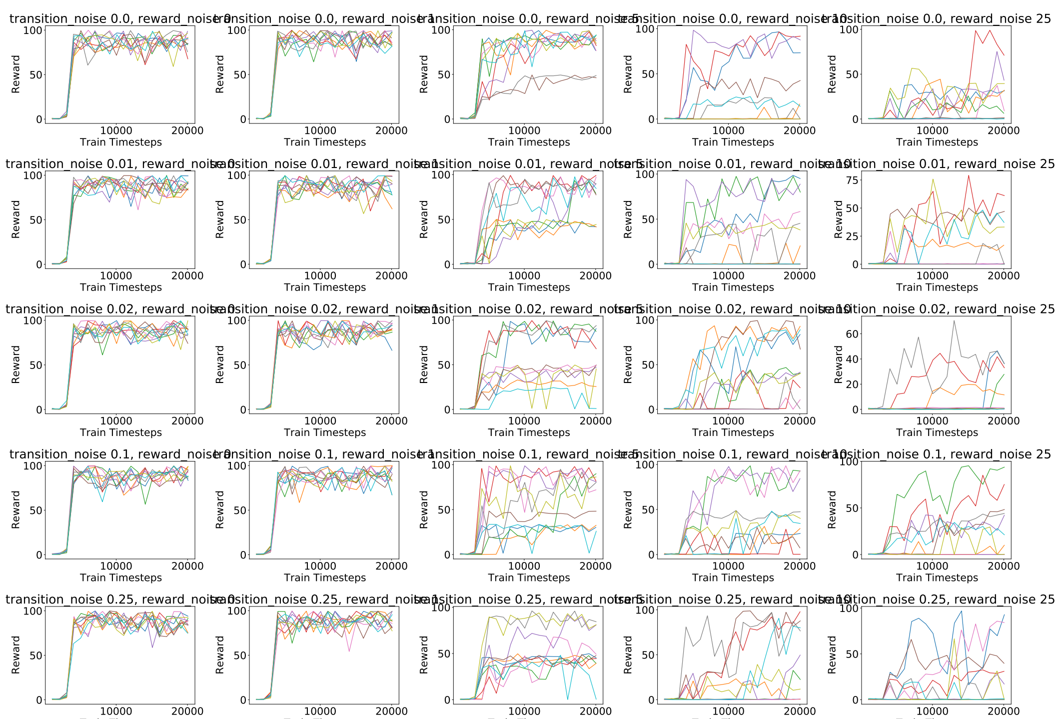


Figure 49: Evaluation Learning Curves for DQN **when varying transition noise and reward noise**. Please note the different Y-axis scales.

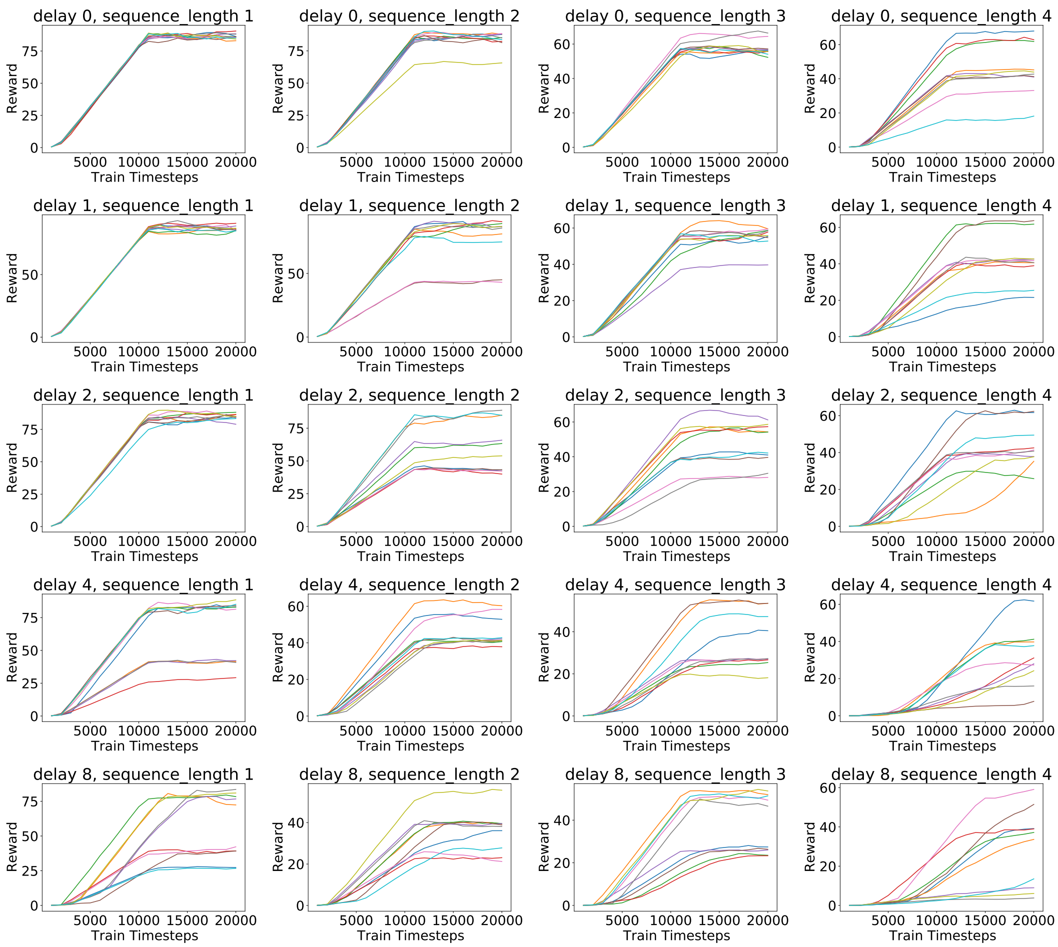


Figure 50: Training Learning Curves for Rainbow **when varying delay and sequence lengths**. Please note the different Y-axis scales.

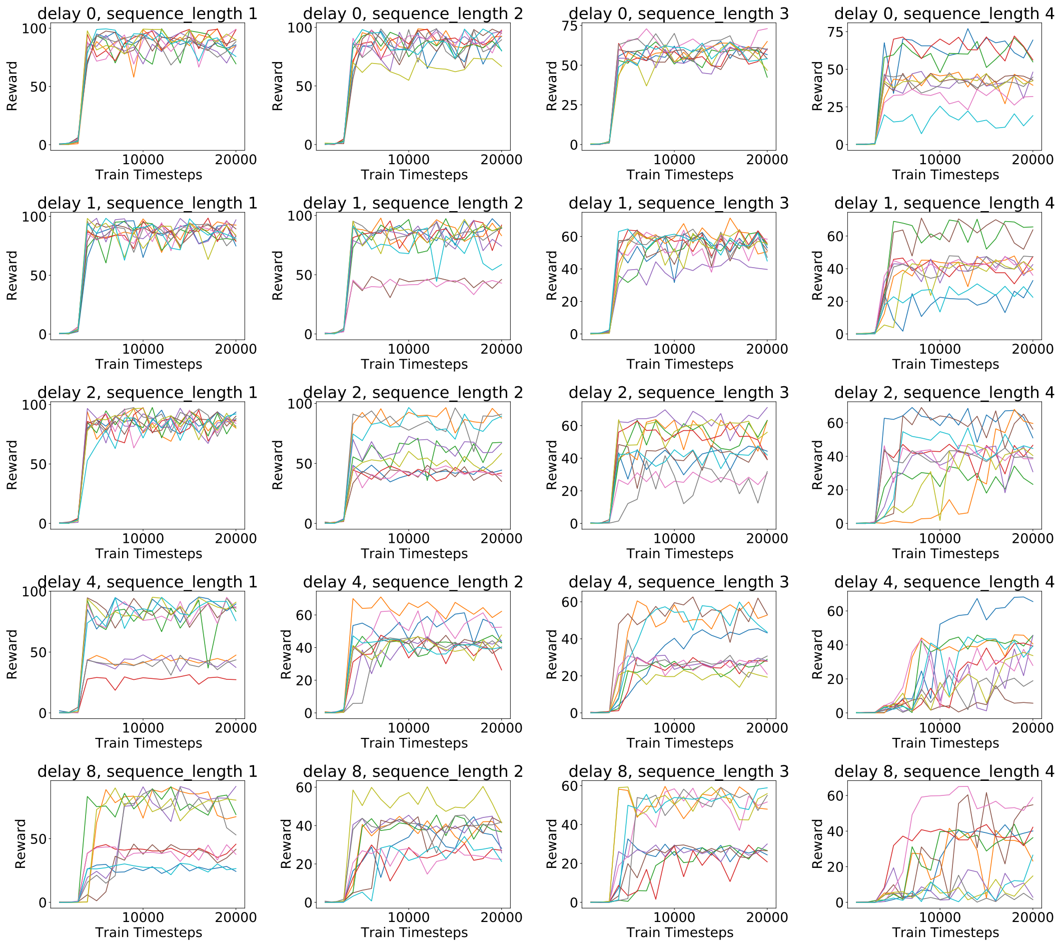


Figure 51: Evaluation Learning Curves for Rainbow **when varying delay and sequence lengths**. Please note the different Y-axis scales.

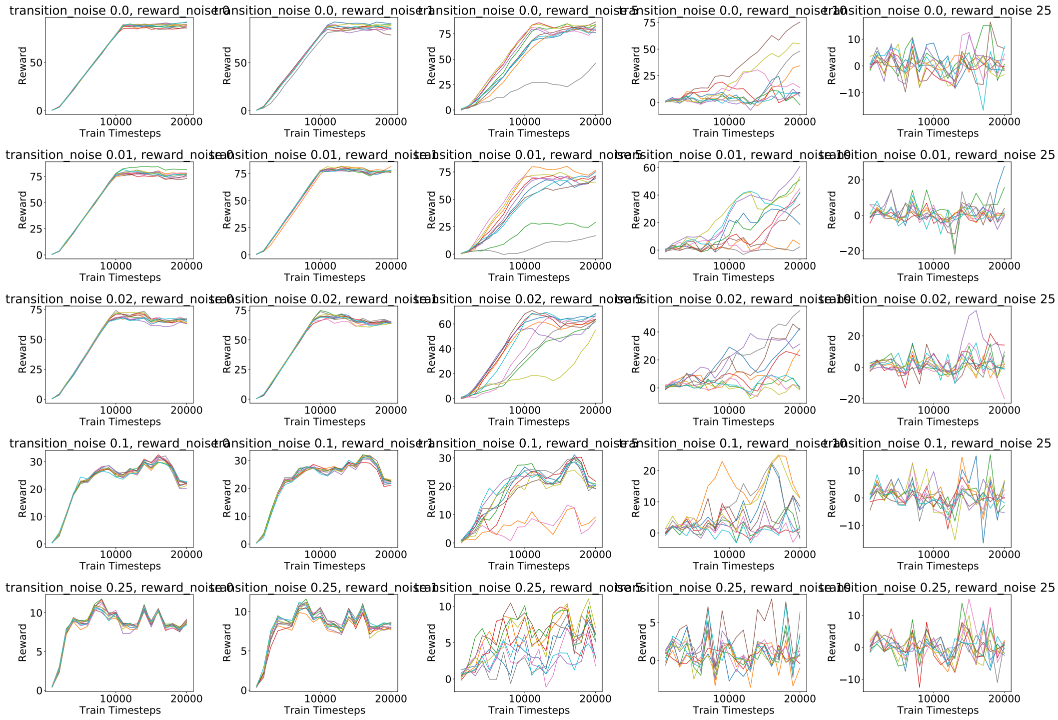


Figure 52: Training Learning Curves for Rainbow **when varying noises**. Please note the different Y-axis scales.

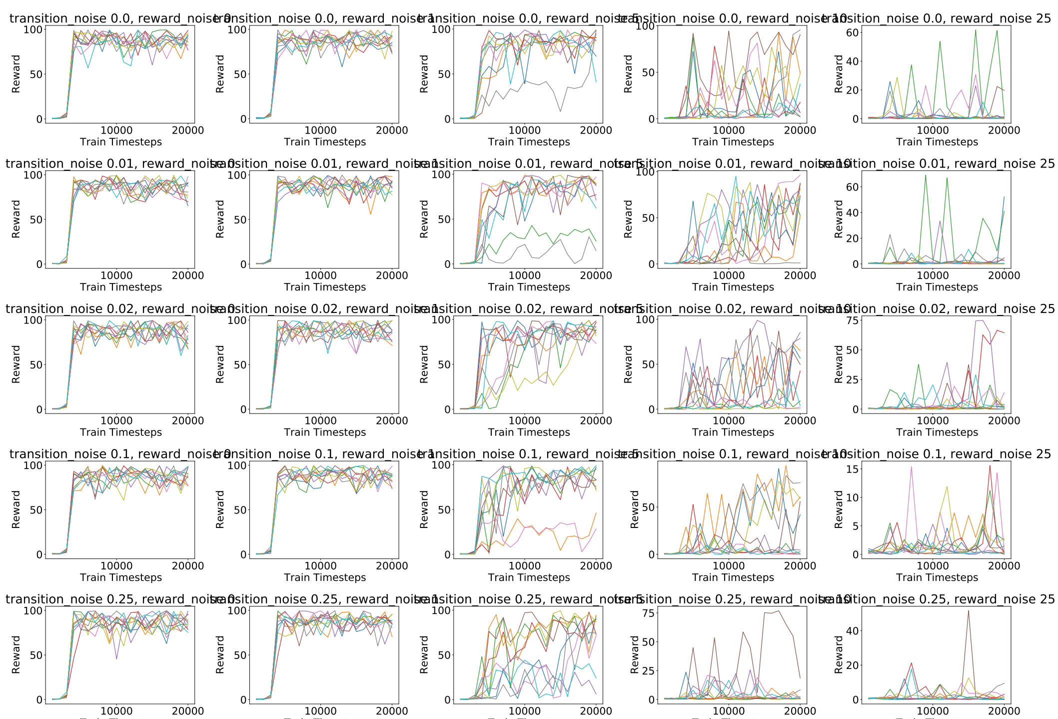


Figure 53: Evaluation Learning Curves for Rainbow **when varying noises**. Please note the different Y-axis scales.

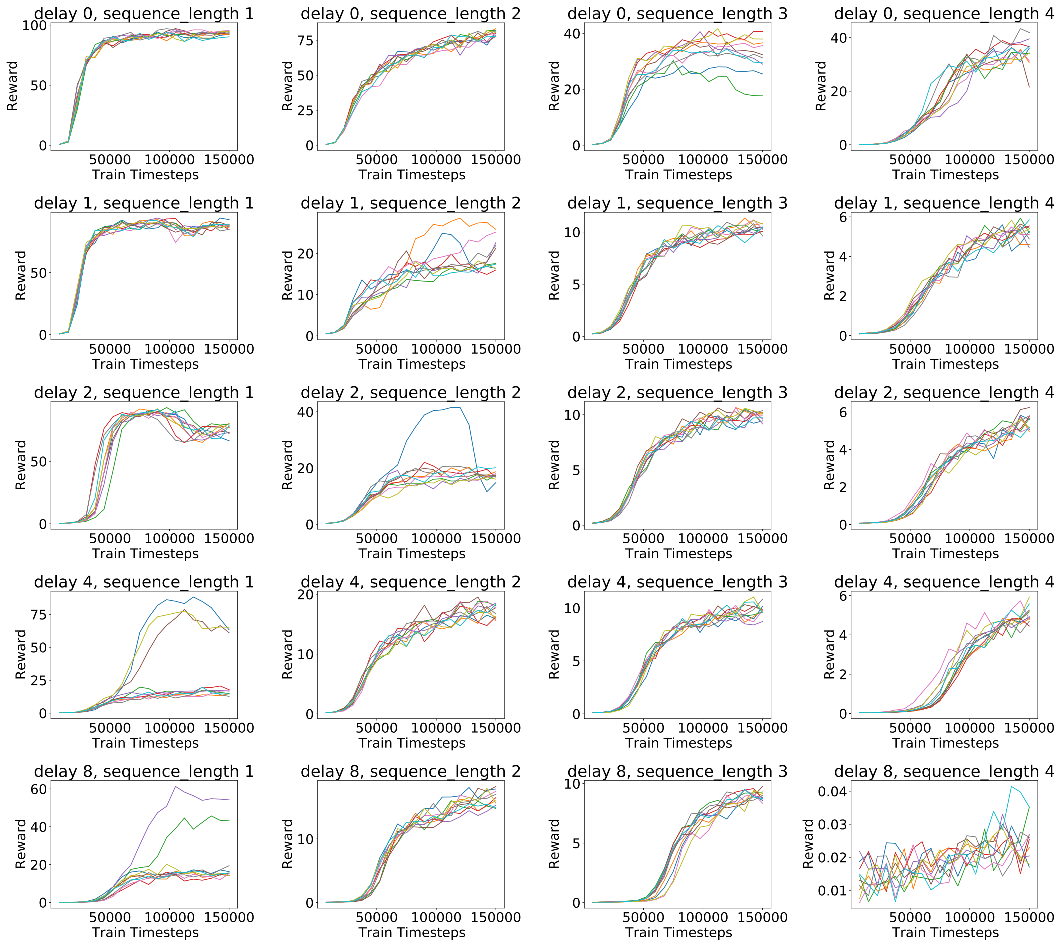


Figure 54: Training Learning Curves for A3C **when varying delay and sequence lengths**. Please note the different Y-axis scales.

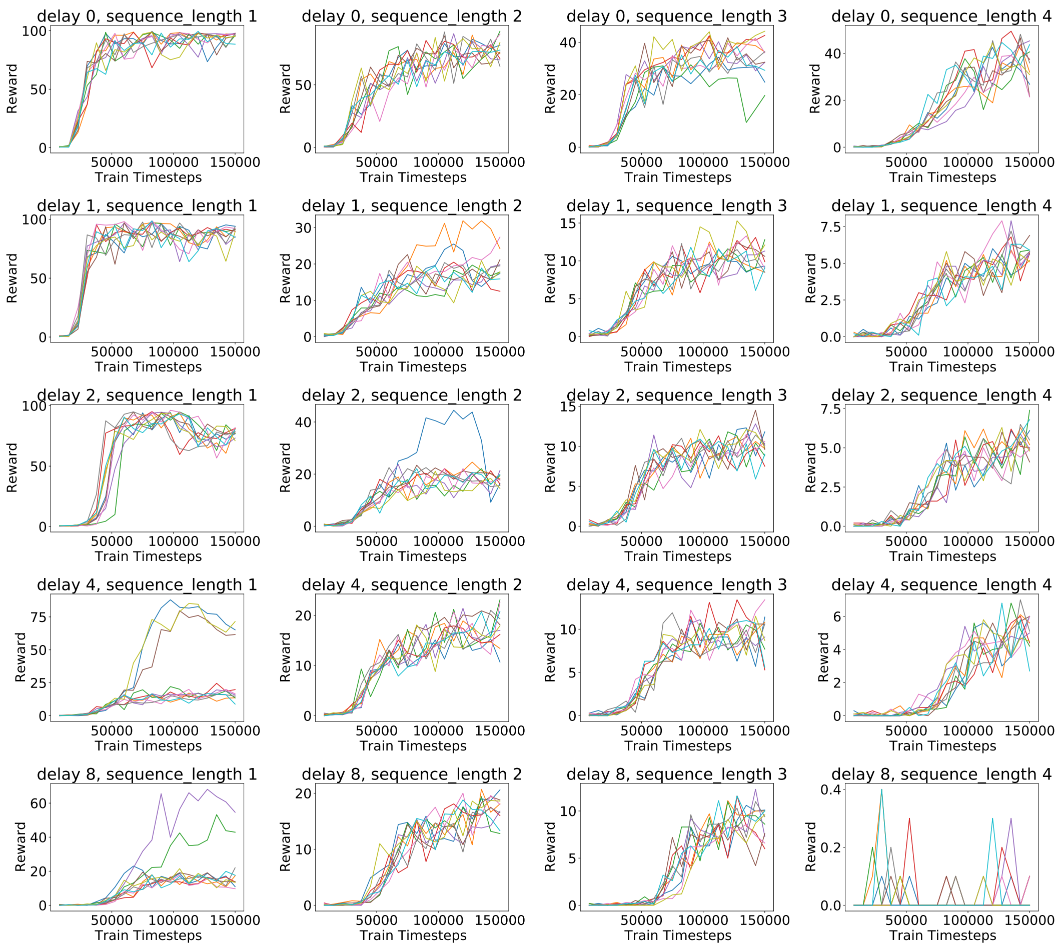


Figure 55: Evaluation Learning Curves for A3C **when varying delay and sequence lengths**. Please note the different Y-axis scales.

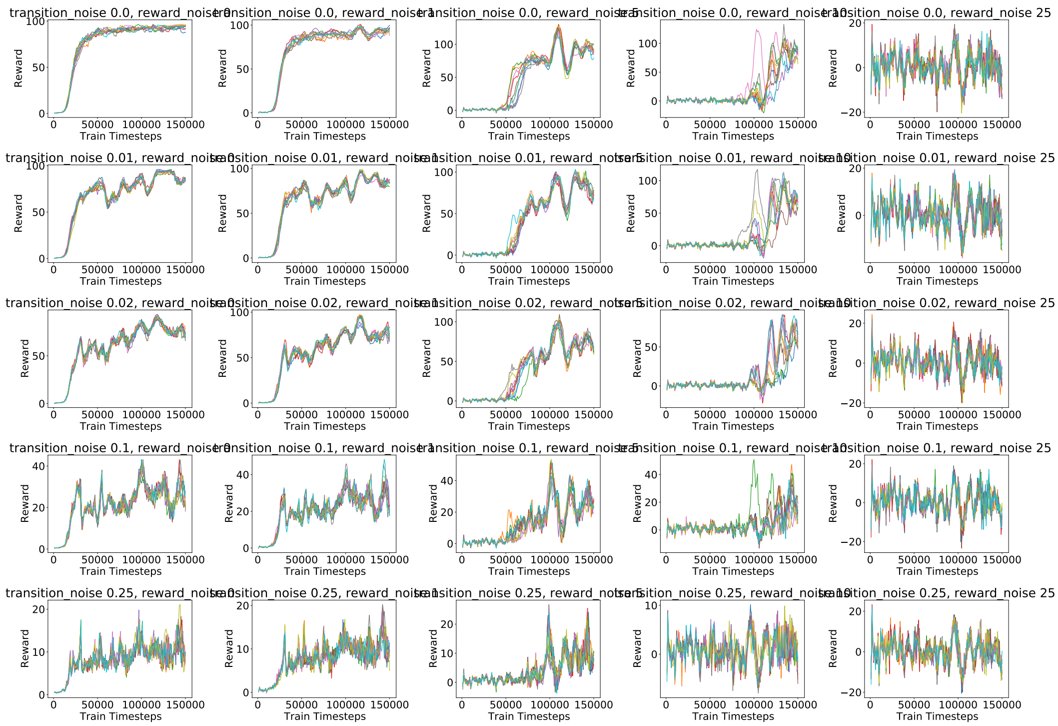


Figure 56: Training Learning Curves for A3C **when varying noises**. Please note the different Y-axis scales.

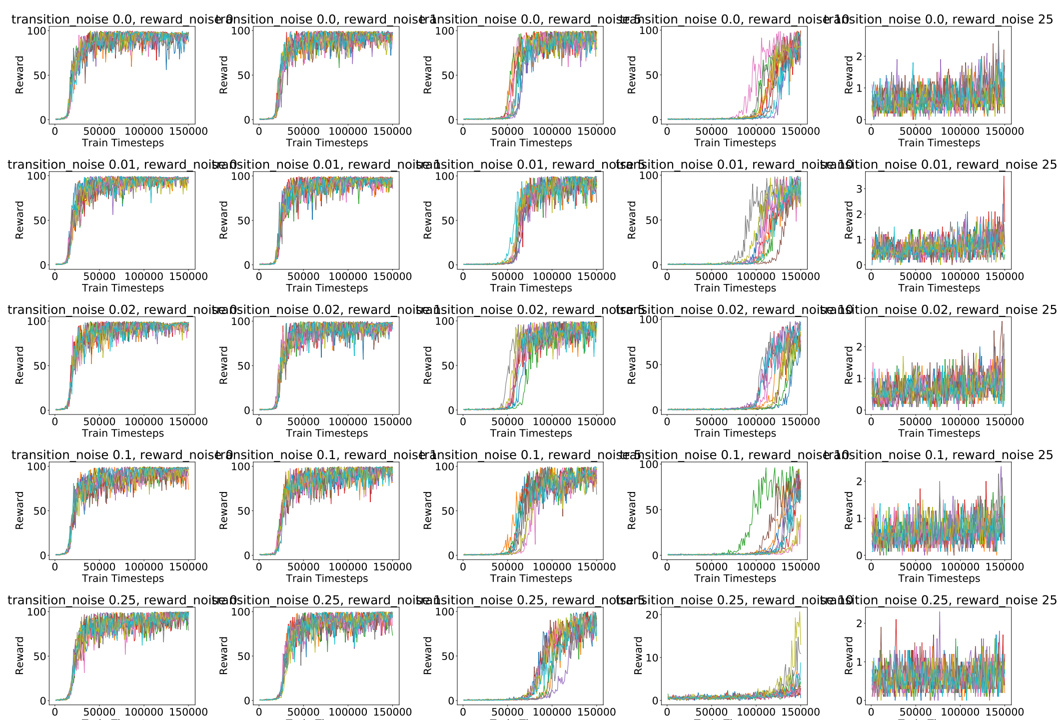


Figure 57: Evaluation Learning Curves for A3C **when varying noises**. Please note the different Y-axis scales.

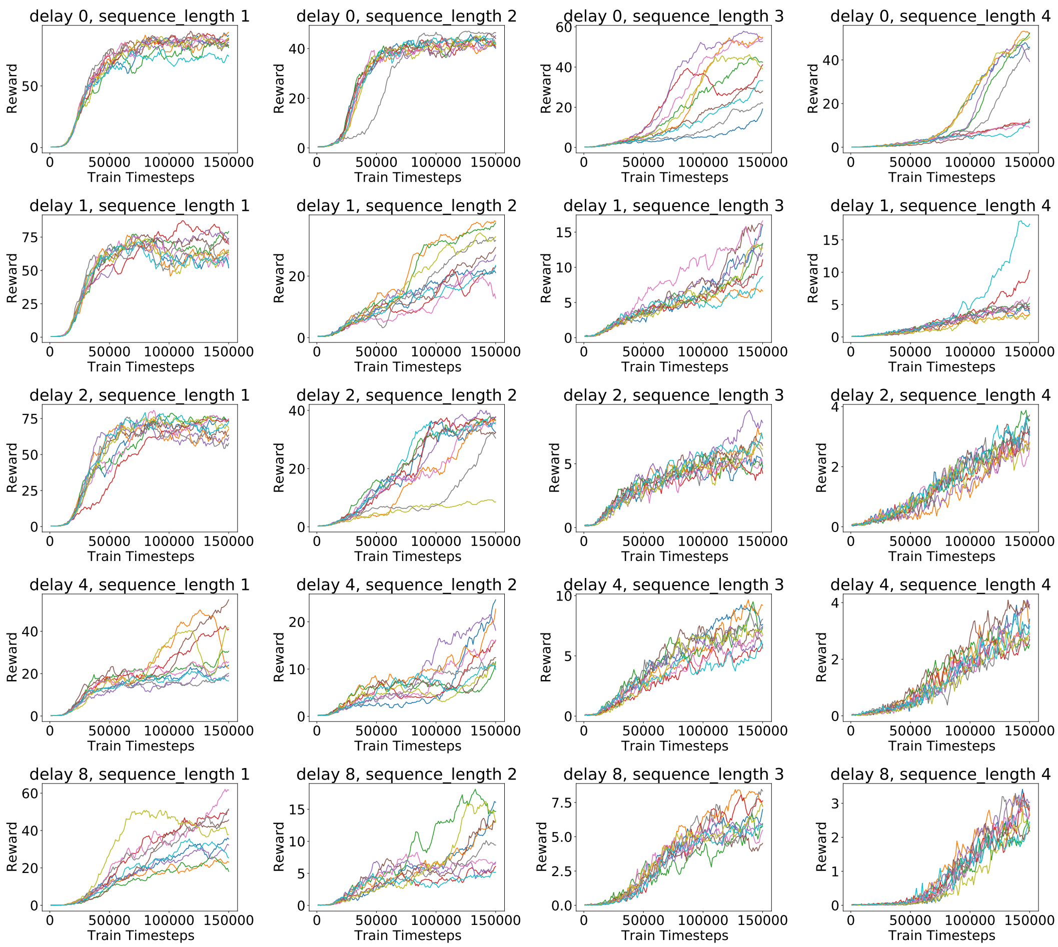


Figure 58: Training Learning Curves for A3C with LSTM **when varying delay and sequence lengths**. Please note the different Y-axis scales.

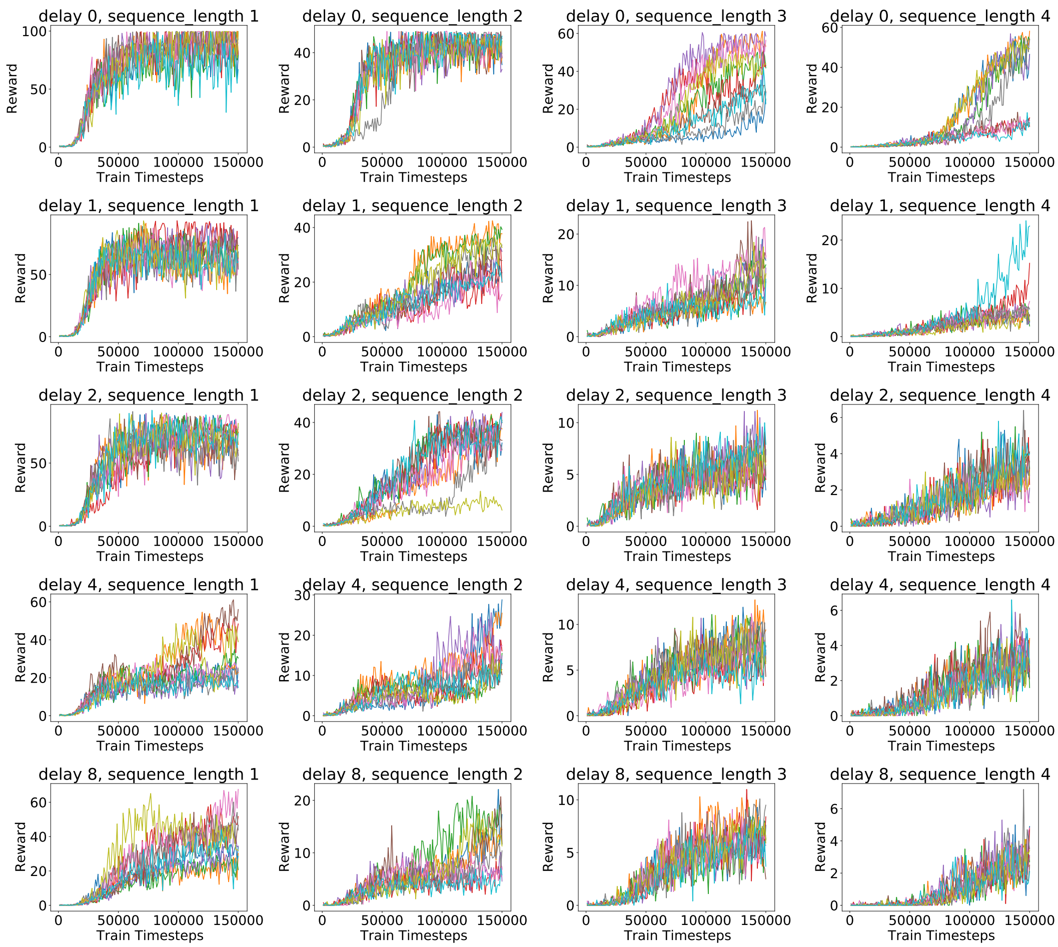


Figure 59: Evaluation Learning Curves for A3C with LSTM **when varying delay and sequence lengths**. Please note the different Y-axis scales.

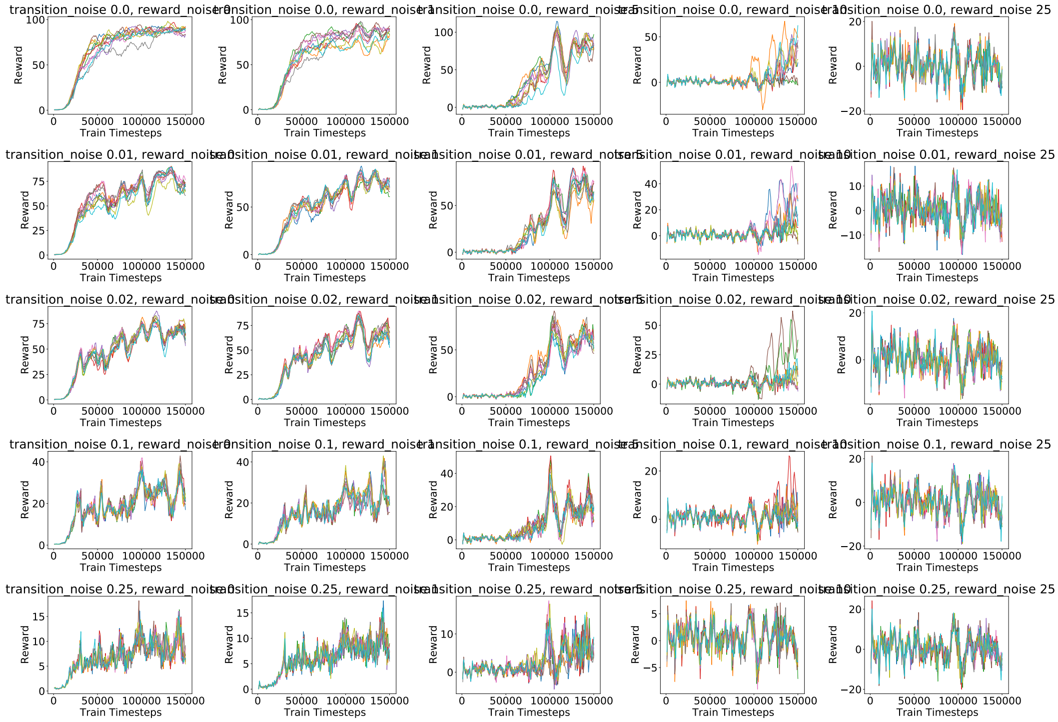


Figure 60: Training Learning Curves for A3C with LSTM **when varying noises**. Please note the different Y-axis scales.

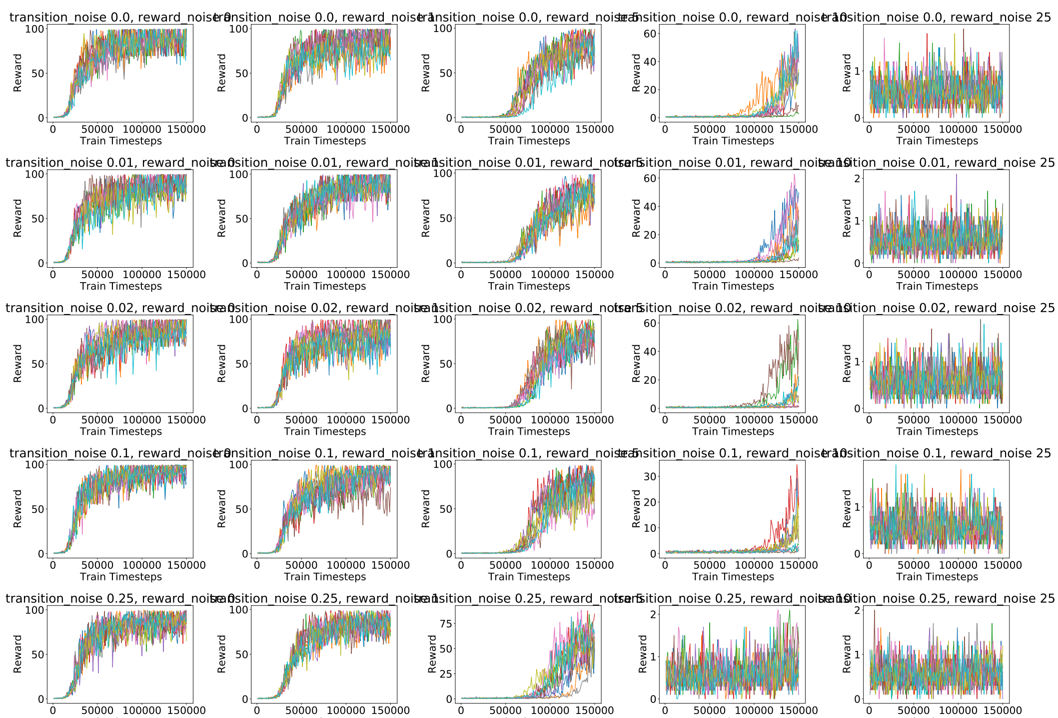


Figure 61: Evaluation Learning Curves for A3C with LSTM **when varying noises**. Please note the different Y-axis scales.

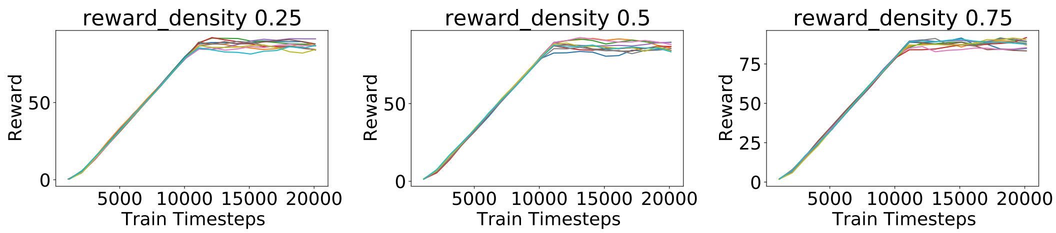


Figure 62: Training Learning Curves for DQN **when varying reward sparsity**. Please note the different Y-axis scales.

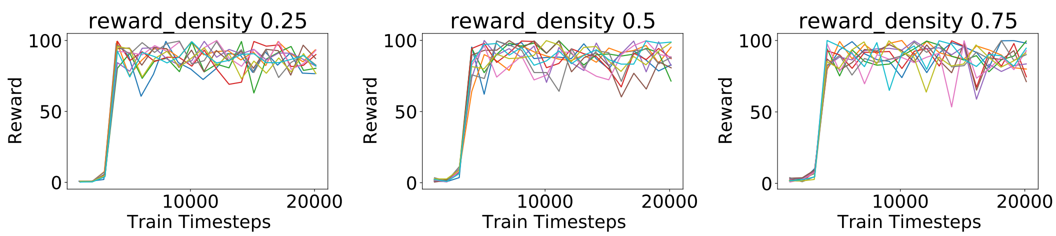


Figure 63: Evaluation Learning Curves for DQN **when varying reward sparsity**. Please note the different Y-axis scales.

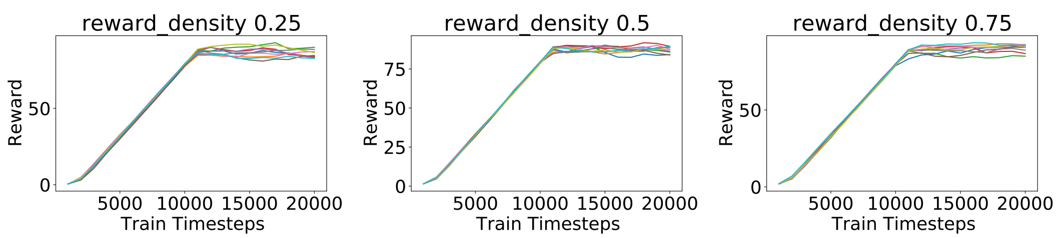


Figure 64: Training Learning Curves for Rainbow **when varying reward sparsity**. Please note the different Y-axis scales.

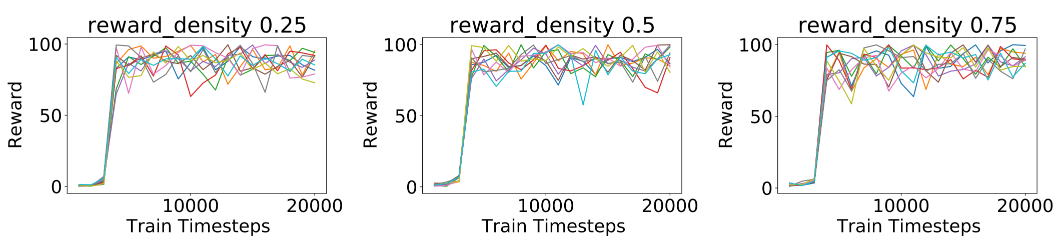


Figure 65: Evaluation Learning Curves for Rainbow **when varying reward sparsity**. Please note the different Y-axis scales.

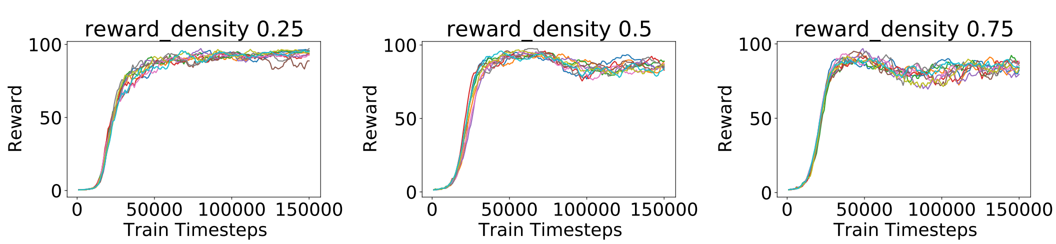


Figure 66: Training Learning Curves for A3C **when varying reward sparsity**. Please note the different Y-axis scales.

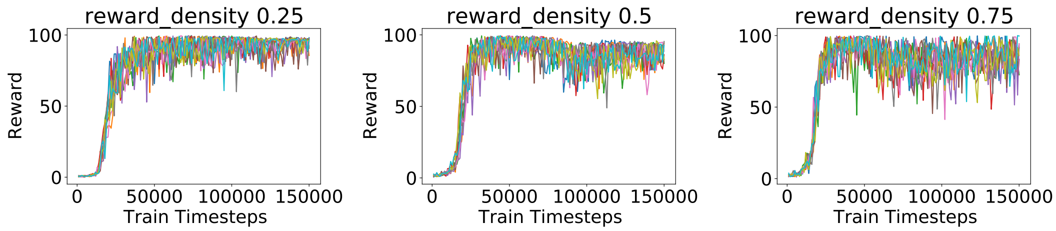


Figure 67: Evaluation Learning Curves for A3C **when varying reward sparsity**. Please note the different Y-axis scales.

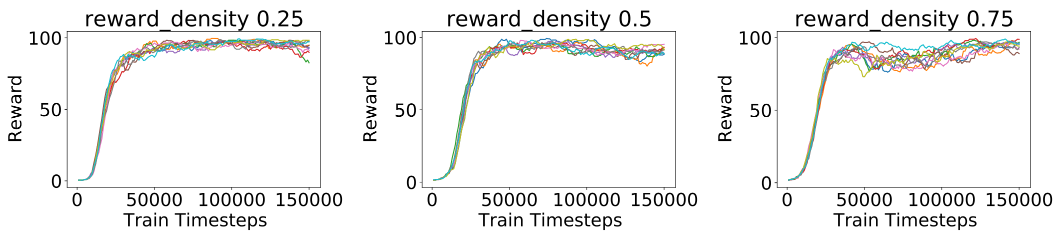


Figure 68: Training Learning Curves for A3C + LSTM **when varying reward sparsity**. Please note the different Y-axis scales.

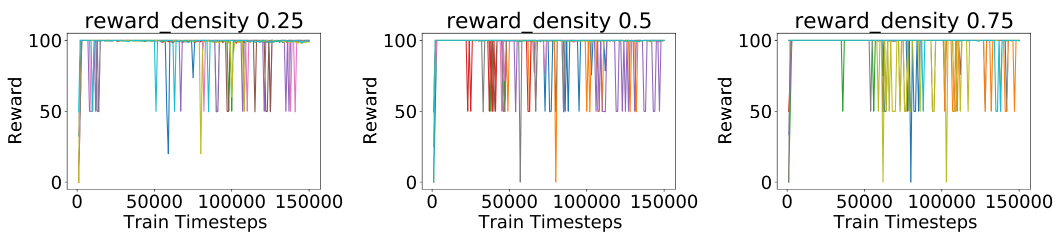


Figure 69: Evaluation Learning Curves for A3C + LSTM **when varying reward sparsity**. Please note the different Y-axis scales.

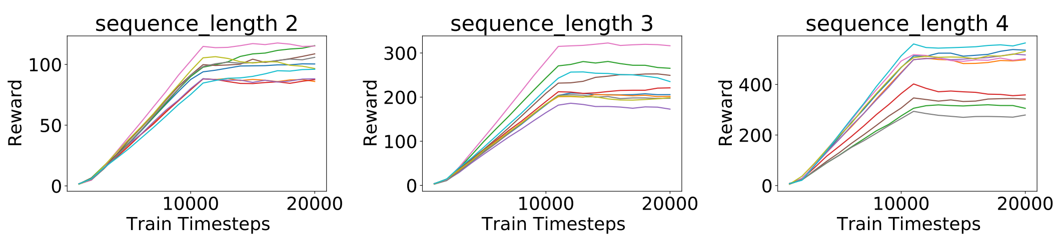


Figure 70: Training Learning Curves for Rainbow **when *make_denser* is *True* for rewardable sequences**. Please note the different Y-axis scales.

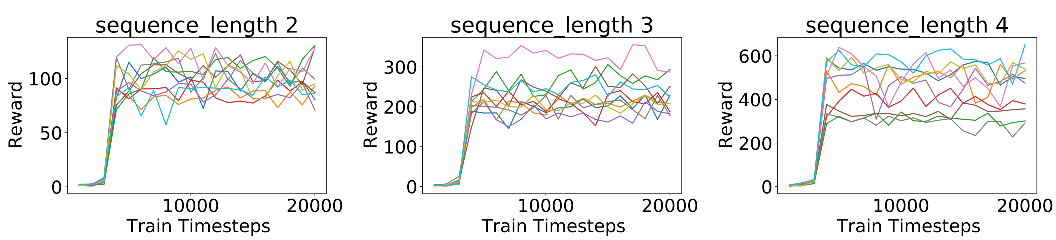


Figure 71: Evaluation Learning Curves for Rainbow **when *make_denser* is *True* for rewardable sequences**. Please note the different Y-axis scales.

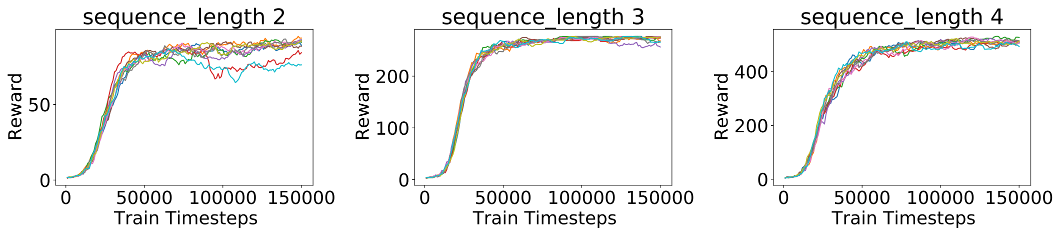


Figure 72: Training Learning Curves for A3C **when *make_denser* is *True* for rewardable sequences**. Please note the different Y-axis scales.

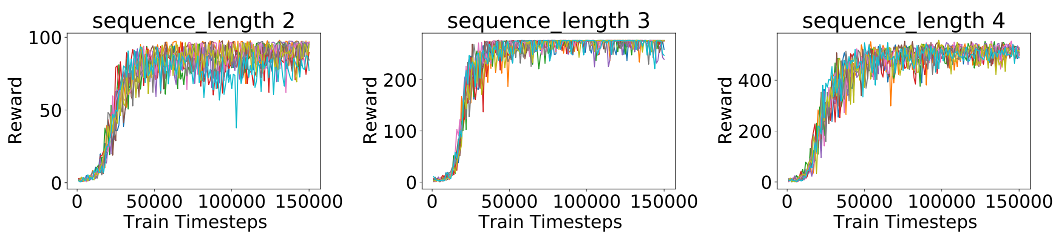


Figure 73: Evaluation Learning Curves for A3C **when *make_denser* is *True* for rewardable sequences**. Please note the different Y-axis scales.

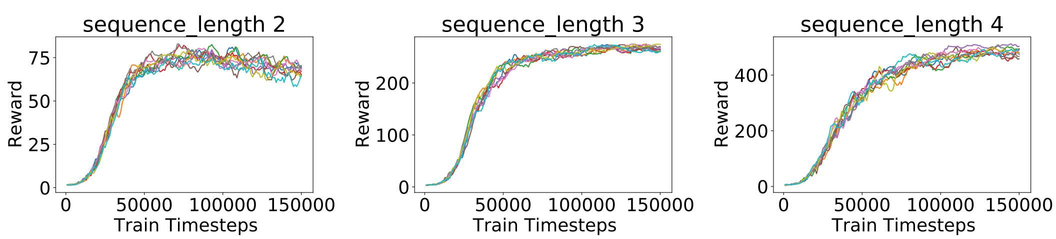


Figure 74: Training Learning Curves for A3C + LSTM **when *make_denser* is *True* for rewardable sequences**. Please note the different Y-axis scales.

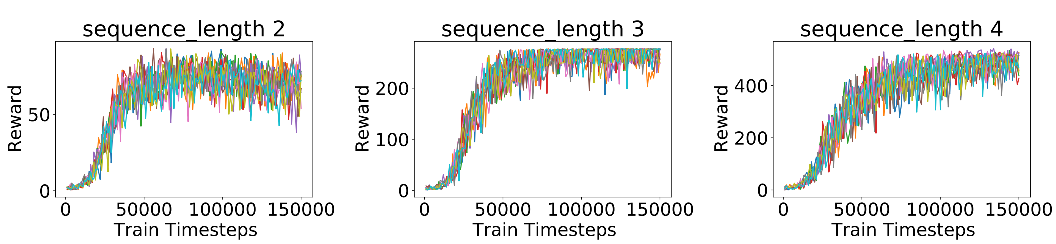


Figure 75: Evaluation Learning Curves for A3C + LSTM **when *make_denser* is *True* for rewardable sequences**. Please note the different Y-axis scales.

N Learning Curves for Complex environments

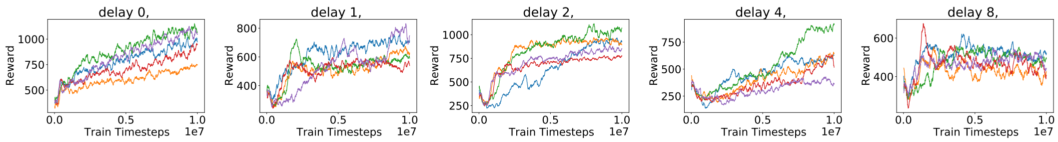


Figure 76: Training Learning Curves for DQN on beam_rider **when varying delay**. Please note the different Y-axis scales.

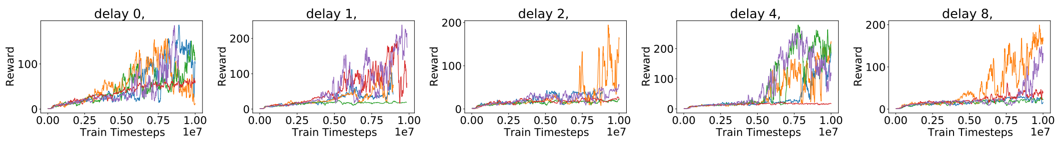


Figure 77: Training Learning Curves for DQN on breakout **when varying delay**. Please note the different Y-axis scales.

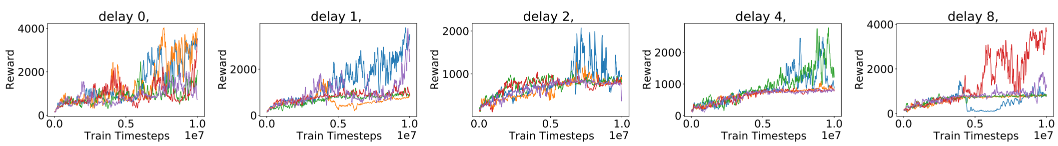


Figure 78: Training Learning Curves for DQN on qbert **when varying delay**. Please note the different Y-axis scales.

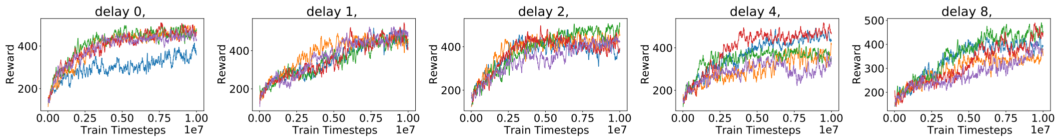


Figure 79: Training Learning Curves for DQN on space_invaders **when varying delay**. Please note the different Y-axis scales.

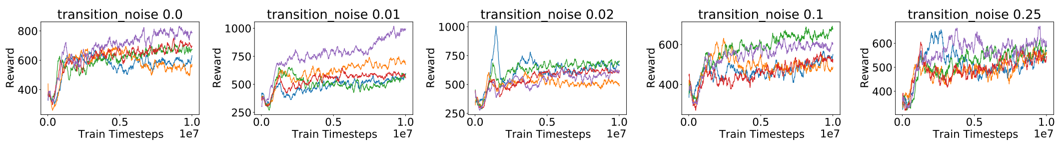


Figure 80: Training Learning Curves for DQN on beam_rider **when varying transition noise**. Please note the different Y-axis scales.

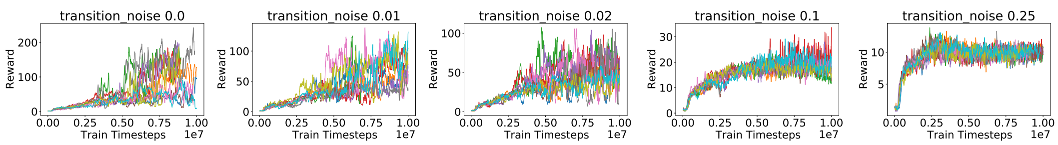


Figure 81: Training Learning Curves for DQN on breakout **when varying transition noise**. Please note the different Y-axis scales.

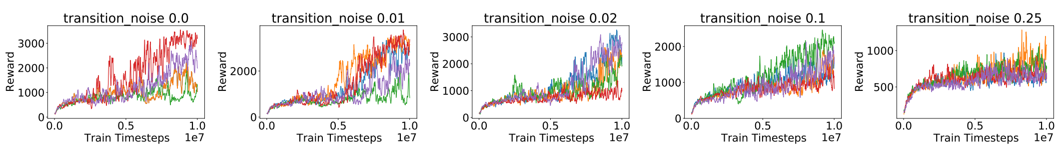


Figure 82: Training Learning Curves for DQN on qbert **when varying transition noise**. Please note the different Y-axis scales.

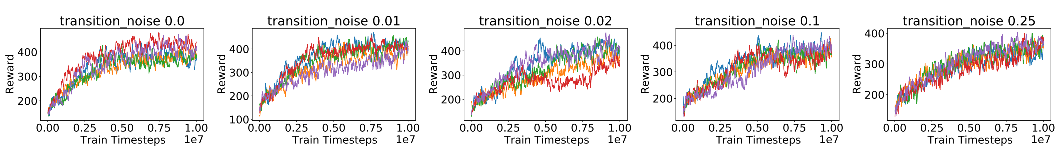


Figure 83: Training Learning Curves for DQN on space_invaders **when varying transition noise**. Please note the different Y-axis scales.

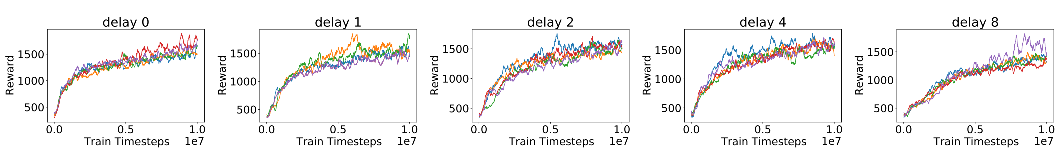


Figure 84: Training Learning Curves for Rainbow on beam_rider **when varying delay**. Please note the different Y-axis scales.

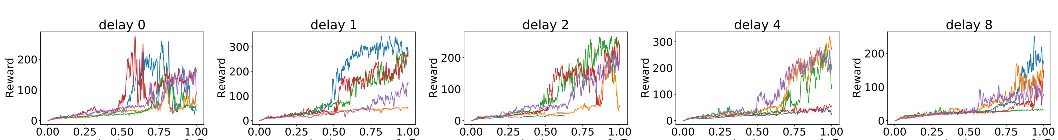


Figure 85: Training Learning Curves for Rainbow on breakout **when varying delay**. Please note the different Y-axis scales.

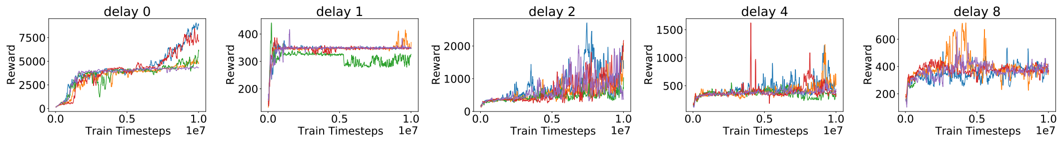


Figure 86: Training Learning Curves for Rainbow on qbert **when varying delay**. Please note the different Y-axis scales.

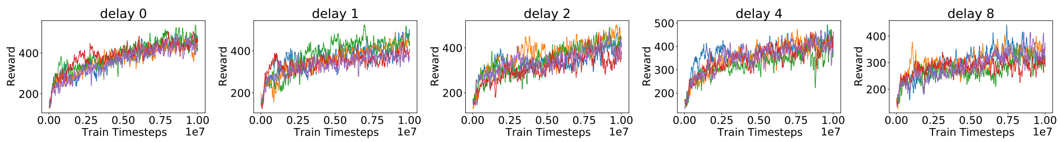


Figure 87: Training Learning Curves for Rainbow on space_invaders **when varying delay**. Please note the different Y-axis scales.

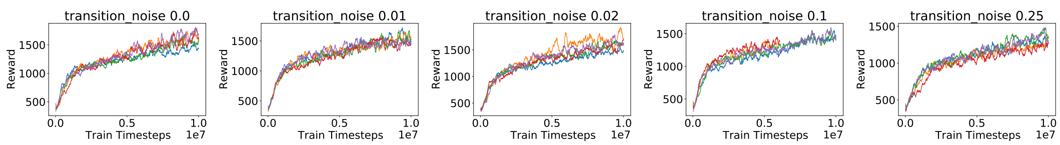


Figure 88: Training Learning Curves for Rainbow on beam_rider **when varying transition noise**. Please note the different Y-axis scales.

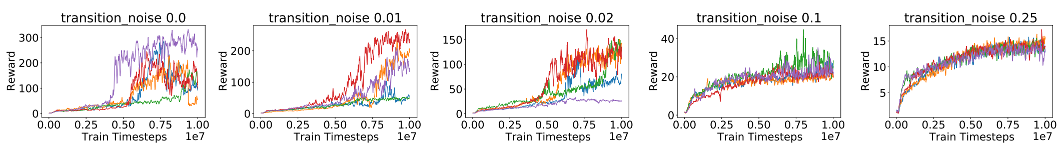


Figure 89: Training Learning Curves for Rainbow on breakout **when varying transition noise**. Please note the different Y-axis scales.

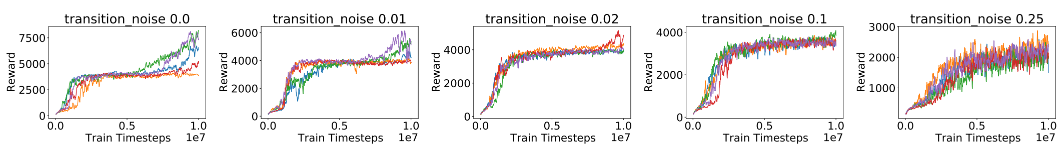


Figure 90: Training Learning Curves for Rainbow on qbert **when varying transition noise**. Please note the different Y-axis scales.

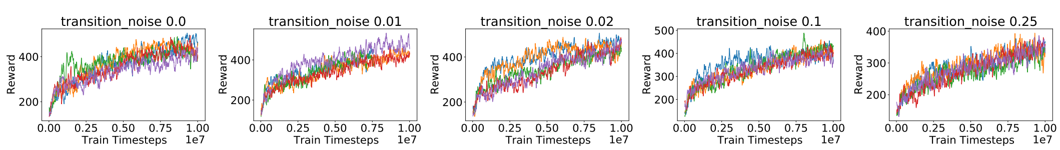


Figure 91: Training Learning Curves for Rainbow on space_invaders **when varying transition noise**. Please note the different Y-axis scales.

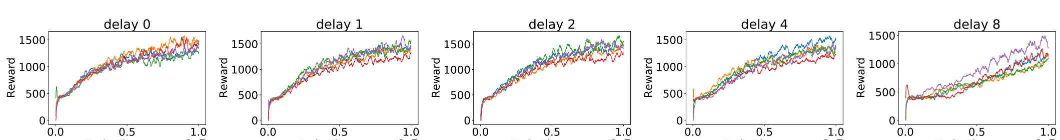


Figure 92: Training Learning Curves for A3C on beam_rider **when varying delay**. Please note the different Y-axis scales.

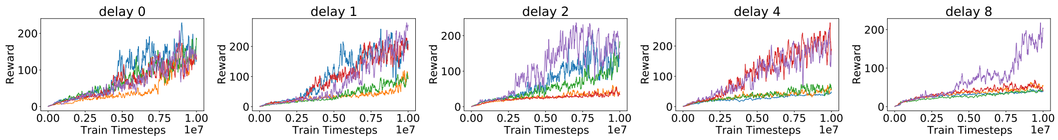


Figure 93: Training Learning Curves for A3C on breakout **when varying delay**. Please note the different Y-axis scales.

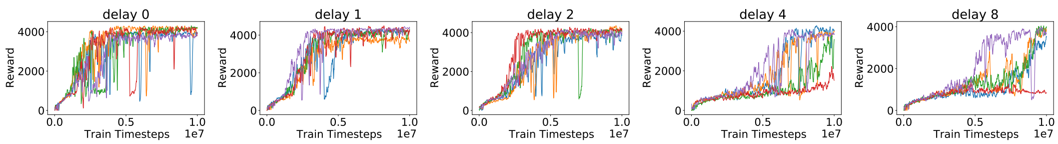


Figure 94: Training Learning Curves for A3C on qbert **when varying delay**. Please note the different Y-axis scales.

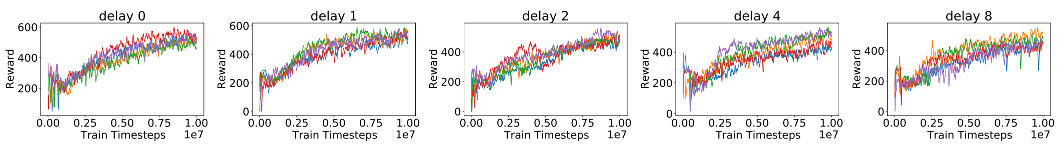


Figure 95: Training Learning Curves for A3C on space_invaders **when varying delay**. Please note the different Y-axis scales.

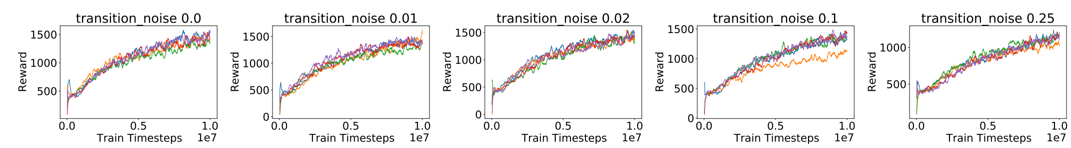


Figure 96: Training Learning Curves for A3C on beam_rider **when varying transition noise**. Please note the different Y-axis scales.

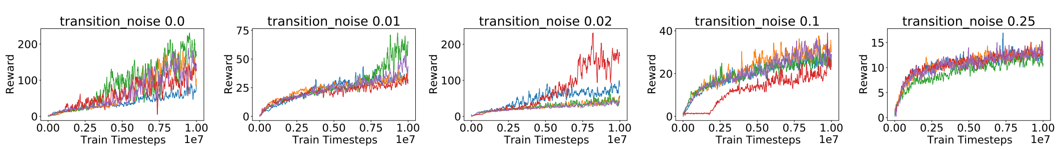


Figure 97: Training Learning Curves for A3C on breakout **when varying transition noise**. Please note the different Y-axis scales.

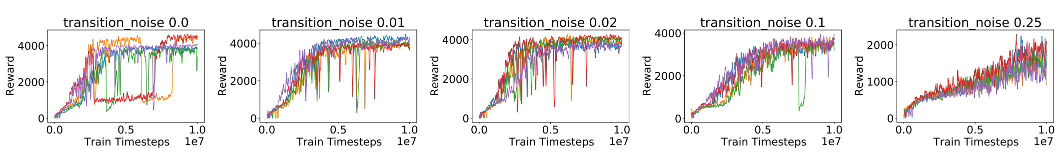


Figure 98: Training Learning Curves for A3C on qbert **when varying transition noise**. Please note the different Y-axis scales.

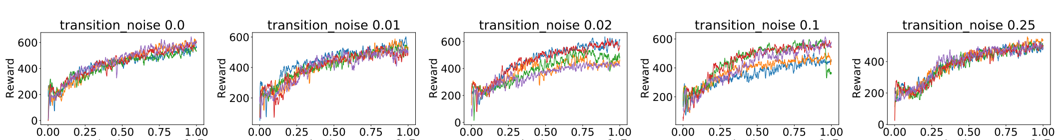


Figure 99: Training Learning Curves for A3C on space_invaders **when varying transition noise**. Please note the different Y-axis scales.

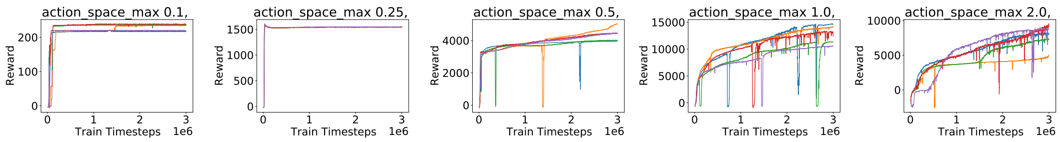


Figure 100: Training Learning Curves for SAC on HalfCheetah **when varying action max**. Please note the different Y-axis scales.

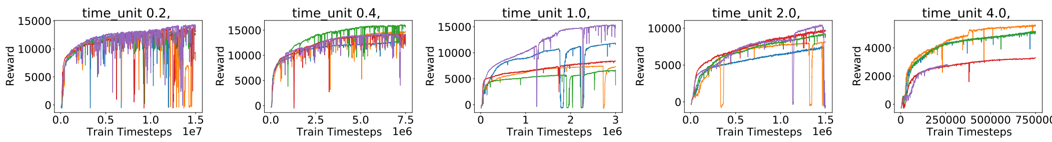


Figure 101: Training Learning Curves for SAC on HalfCheetah **when varying time unit**. Please note the different Y-axis scales.

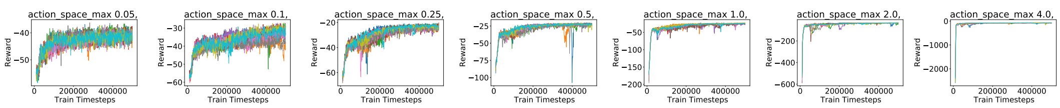


Figure 102: Training Learning Curves for SAC on Pusher **when varying action max**. Please note the different Y-axis scales.

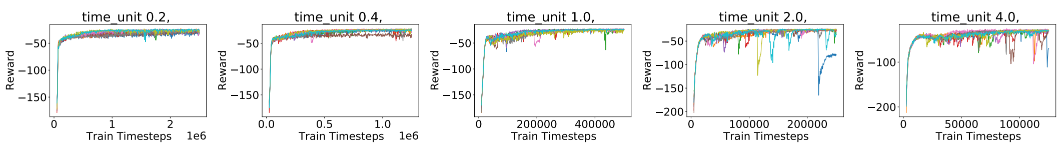


Figure 103: Training Learning Curves for SAC on Pusher **when varying time unit**. Please note the different Y-axis scales.

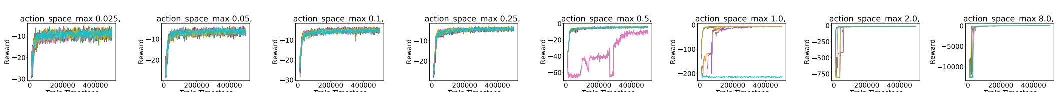


Figure 104: Training Learning Curves for SAC on Reacher **when varying action max**. Please note the different Y-axis scales.

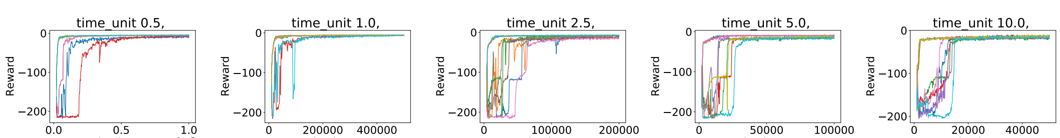


Figure 105: Training Learning Curves for SAC on Reacher **when varying time unit**. Please note the different Y-axis scales.

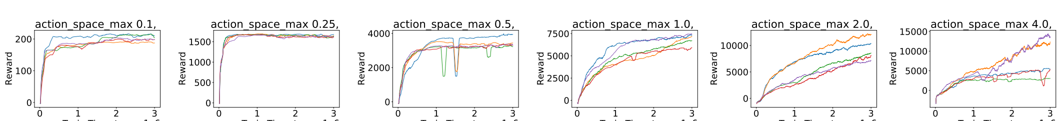


Figure 106: Training Learning Curves for DDPG **when varying action max**. Please note the different Y-axis scales.

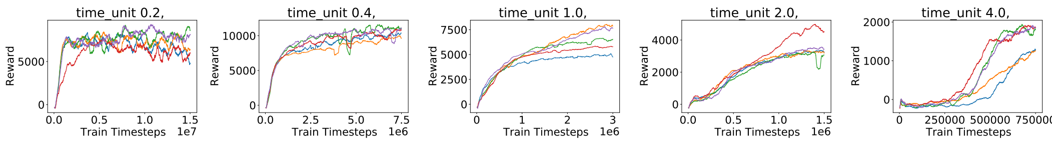


Figure 107: Training Learning Curves for DDPG **when varying time unit**. Please note the different Y-axis scales.

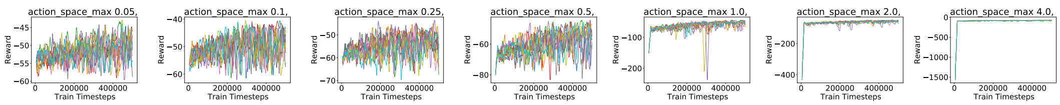


Figure 108: Training Learning Curves for DDPG **when varying action max**. Please note the different Y-axis scales.

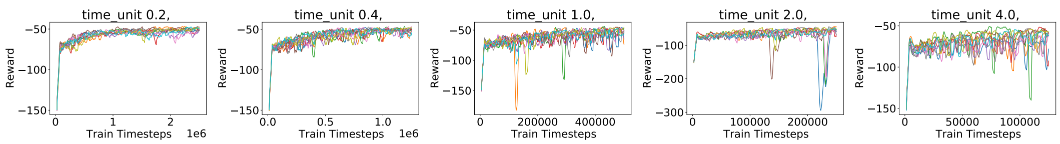


Figure 109: Training Learning Curves for DDPG **when varying time unit**. Please note the different Y-axis scales.

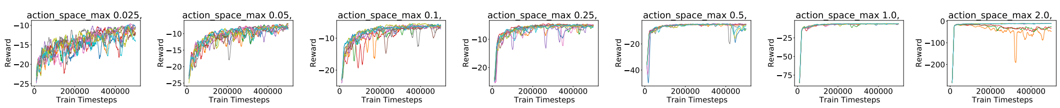


Figure 110: Training Learning Curves for DDPG **when varying action max**. Please note the different Y-axis scales.

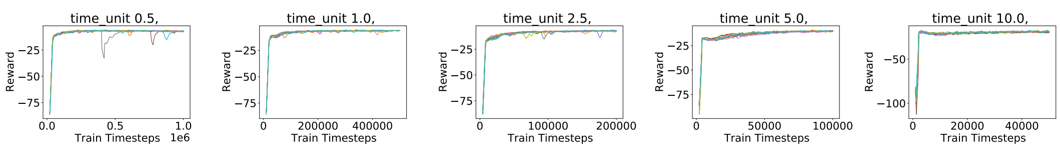


Figure 111: Training Learning Curves for DDPG **when varying time unit**. Please note the different Y-axis scales.

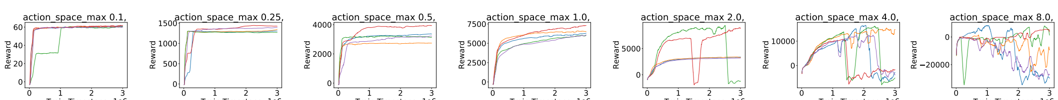


Figure 112: Training Learning Curves for TD3 **when varying action max**. Please note the different Y-axis scales.

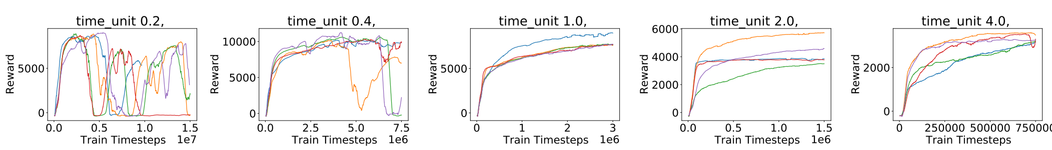


Figure 113: Training Learning Curves for TD3 **when varying time unit**. Please note the different Y-axis scales.

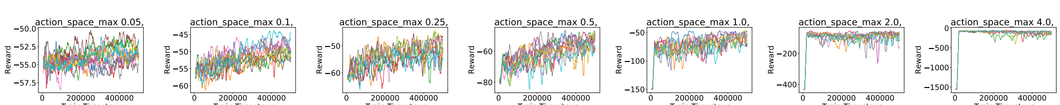


Figure 114: Training Learning Curves for TD3 **when varying action max**. Please note the different Y-axis scales.

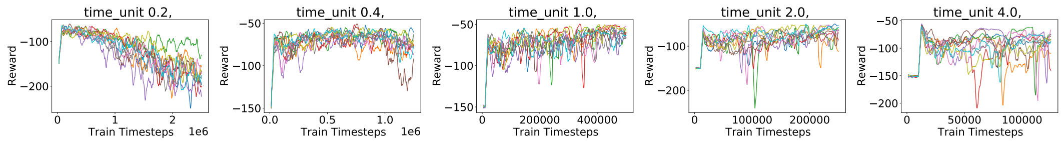


Figure 115: Training Learning Curves for TD3 **when varying time unit**. Please note the different Y-axis scales.

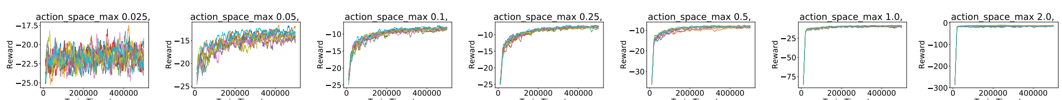


Figure 116: Training Learning Curves for TD3 **when varying action max**. Please note the different Y-axis scales.

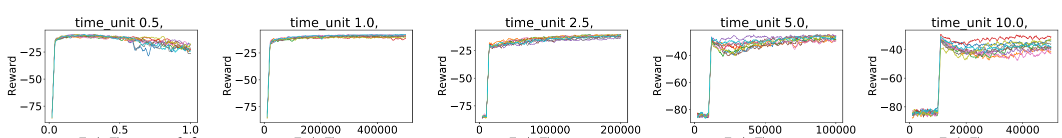


Figure 117: Training Learning Curves for TD3 **when varying time unit**. Please note the different Y-axis scales.

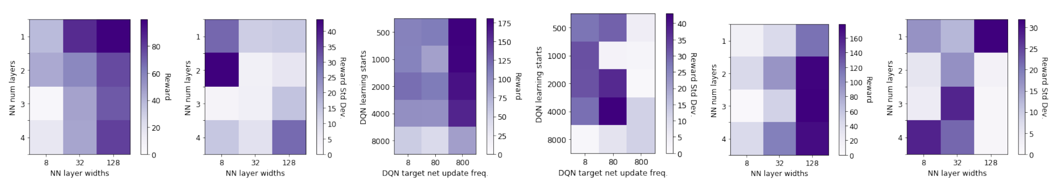
O Hyperparameter Tuning

We gained some interesting insights into the significance of certain hyperparameters while tuning them for the different algorithms. Thus, our toy environments might in fact be good test beds for researching hyperparameters in RL, too. For instance, *target network update frequency* turned out to be very significant for learning and sub-optimal values led to very noisy and unreliable training and unexpected results such as networks with greater capacity not performing well. Once we tuned it, however, training was much more reliable and, as expected, networks with greater capacity did well. We now describe the tuning process and an example insight in more detail.

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 more robust performance estimate. We describe a small part of our hyperparameter tuning for DQN next. All hyperparameter settings for tuned agents can be found in Appendix P.

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 118a). However, the variance in performance was quite high (Figure 118b). When we tried to tune DQN hyperparameters *learning starts* and *target network update frequency*, however, it became clear that the target network update frequency was very significant (Figure 118c and 118d) 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 118e and 118f).

We then changed the network number of neurons grid to [128, 256, 512] and changed target network update frequency grid to [80, 800, 8000] and continued with further tuning using the grid values specified in Appendix P.



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

Figure 118: Mean episodic reward at the end of training for different hyperparameter sets for DQN. Please note the different colorbar scales.

P Tuned Hyperparameters

The code for corresponding experiments for both discrete and continuous environments can be found in the accompanying code for the paper. The experiments with *_tune_hps* in the names contain the grid of HPs that were tuned over. In some instances (where *_tune_hps* experiments do not exist), in order to save costs, we used the default HPs in Ray. The README describes how to run the experiments using *config* files and which *config* files correspond to which experiments. Older experiments on the discrete toy environments were run with Ray 0.7.3, while for the newer continuous and complex environments, they were run with Ray 0.9.0. We had to use Ray 0.7.3 for the discrete toy environments and Ray 0.9.0 for the continuous toy ones because we had run the discrete cases for a previous version of the paper on 0.7.3. DDPG was not working and SAC was not implemented in Ray at that time. We tried to use Ray 0.9.0 also for the discrete version but found for the 1st algorithms we tested that, for the same hyperparameters, the results did not transfer even across implementations of the same library. This further makes our point about using our platform to unit test algorithms. For the complex environments, since we had to tune the environments again anyway, we decided to use the newer Ray version.

Since we did not save the hyperparameter grids for discrete toy environments in separate files, they are provided here. The names of the hyperparameters for the algorithms will match those used in Ray 0.7.3. The hyperparameters for the newer continuous and complex environment experiments can be found in the respective experiment config files in the experiments directory.

P.1 DQN

```
num_layerss = [1, 2, 3, 4]
layer_widths = [8, 32, 128] # at first
layer_widths = [128, 256, 512] # after setting target_net_update_freq = 800
     showed that 128 was the best number of the old 3, we changed search
    grid for number of neurons
fcnet_activations = ["tanh", "relu", "sigmoid"]
learning_startss = [500, 1000, 2000, 4000, 8000]
target_network_update_freqs = [8, 80, 800] # at first
target_network_update_freqs = [80, 800, 8000] # after seeing
    target_net_update_freq = 800 is much better than 80, changed the grid
    for it
double_dqn = [False, True]
learning_rates = [1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
adam_epsilons = [1e-3, 1e-4, 1e-5, 1e-6] # also tried [1e-1, 1e-4, 1e-7, 1e
    -10]

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,
              }
      },
      },
      )
```

P.2 Rainbow

```
num_layerss = [1, 2, 3, 4]
layer_widths = [128, 256, 512]
fcnet_activations = ["tanh", "relu", "sigmoid"]
```

```
learning_rates = [1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
learning_startss = [500, 1000, 2000, 4000, 8000]
target_network_update_freqs = [80, 800, 8000]
double_dqn = [False, True]

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,
                }
            },
    },
)
```

P.3 A3C

```
Grids of value for the hyperparameters over which they were tuned:

num_layerss = [1, 2, 3, 4]
layer_widths = [64, 128, 256]

learning_rates = [1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
fcnet_activations = ["tanh", "relu", "sigmoid"]

lambdas = [0, 0.5, 0.95, 1.0]
grad_clips = [10, 30, 100]

vf_loss_coeffs = [0.1, 0.5, 2.5]
entropy_coeffs = [0.001, 0.01, 0.1, 1]

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,
            }
    },
    },
)
```

P.4 A3C + LSTM

Grids of value for the hyperparameters over which they were tuned:

```
num_layerss = [1, 2, 3, 4]
layer_widths = [64, 128, 256]

learning_rates = [1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
fcnet_activations = ["tanh", "relu", "sigmoid"]
```

```
lambdas = [0, 0.5, 0.95, 1.0]
grad_clips = [10, 30, 100]

vf_loss_coeffs = [0.1, 0.5, 2.5]
entropy_coeffs = [0.001, 0.01, 0.1, 1]

lstm_cell_sizes = [64, 256, 512]
lstm_use_prev_action_rewards = [False, True]

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,
            }
    },
    },
)
```

Q More on Conclusion and Future Work

Our benchmark is also designed with long term AGI in mind. Dimensions like identifying delays and sequences may be essential to solving AGI because when these are identified we know the causal actions leading to a reward. Most current algorithms lack such capability because they cannot figure out the causality [43].

Among the continuous environments, we have a toy task of moving along a line. Here, 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 already gave some interesting results and further work will follow. We are in the process of implementing 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 plan to allow using their transition and reward functions to perform *imaginary* rollouts without affecting the current state of the system.

Another significant dimension is *reachability* in the transition graph. This is currently implemented using diameter. We believe a lot more insights can be gained from graph theory to model toy environments which try to mimic specific real life situations at a very high level. We plan to add random generation of specific types of transition graphs and not just the regularly structured one using the diameter.

The fine-grained control of dimensions allows relating these to good hyperparameter choices. So, our playground could also be used to learn a mapping from hardness dimensions to hyperparameters for different types of environments and even to warm-start hyperparameter optimisation for environments with similar hardness dimensions. This holds promise for future meta-learning algorithms. In a similar vein, it could also be used to perform Combined Algorithm Selection and Hyperparameter Optimisation [54], since it's clear that currently different RL algorithms do well in different kinds of environments.

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

The states and actions contained in a rewardable 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 [53], could try to identify these rewardable sequences at the higher level and then carry out *atomic* actions at the lower level.

We also hope to benchmark other algorithms like PPO[3] [48], Rudder [4], MCTS [49], DDPG[4] [34] on continuous tasks and table-based agents and to show theoretical results match with practice in toy environments.

We also aim to promote reproducibility in RL as in [18] and hope our platform helps with that goal. To this end, we have already improved the Gym `Box` and `Discrete` `Spaces` to allow their seeds to be controlled at initialization time as well.

We need different RL agents for different environments. Aside from some basic heuristics such as applying DDPG [34] to continuous environments and DQN to discrete environments, it is not very clear when to use which RL agents. 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 agents which adapt to the environment at hand. Additionally, aside from being a great platform for designing and debugging RL agents, *MDP Playground* is also a great didactic tool for teaching how RL agents work in different environments.

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

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

R CPU specifications

Cluster experiments were run on *Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz* cores for approximately 55000 CPU hours.

R.1 CO2 Emission Related to Experiments

Experiments were conducted using a private infrastructure, which has a carbon efficiency of 0.432 $kgCO_2eq/kWh$. A cumulative of 55000 hours of computation was performed on hardware of type Intel Xeon E5-2699 (TDP of 145W).

Total emissions are estimated to be 3445.2 $kgCO_2eq$ of which 0 percents were directly offset.

Estimations were conducted using the MachineLearning Impact calculator presented in [29].

The laptop specification were:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz
stepping        : 10
microcode       : 0xb4
cpu MHz         : 900.055
cache size      : 9216 KB
physical id     : 0
siblings        : 12
core id         : 0
cpu cores       : 6
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
    cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall
    nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good
    nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni
    pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16
    xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes
     xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb
    invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
    ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms
    invpcid rtm mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec
    xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window
    hwp_epp md_clear flush_l1d
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf
     mds swapgs
bogomips        : 5184.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```