Journal of Artificial Intelligence Research 77 (2023) 821-890 Submitted 10/2022; published 07/2023

MDP Playground: An Analysis and Debug Testbed for Reinforcement Learning

Raghu Rajan RAJANR@CS.UNI-FREIBURG.DE
University of Freiburg

Jessica Lizeth Borja Diaz JESSICALIZETHBORJA@GMAIL.COM
University of Freiburg

Suresh Guttikonda SURESH.GUTTIKONDA@STUDENTS.UNI-FREIBURG.DE
University of Freiburg

Fabio Ferreira FERREIRA@CS.UNI-FREIBURG.DE
University of Freiburg

André Biedenkapp BIEDENKA@CS.UNI-FREIBURG.DE
University of Freiburg

Jan Ole von Hartz HARTZJ@CS.UNI-FREIBURG.DE
University of Freiburg

Frank Hutter FH@CS.UNI-FREIBURG.DE
University of Freiburg

Abstract

We present *MDP Playground*, a testbed for Reinforcement Learning (RL) agents with *dimensions of hardness* that can be controlled independently to challenge agents in different ways and obtain varying degrees of hardness in toy and complex RL environments. We consider and allow control over a wide variety of dimensions, including *delayed rewards*, *sequence lengths*, *reward density*, *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 to understand agents better. We then show how to design experiments using *MDP Playground* to gain insights on the toy environments. We also provide wrappers that can inject many of these dimensions into any *Gym* environment. We experiment with these wrappers on *Atari* and *Mujoco* to allow for understanding the effects of these dimensions on environments that are more complex than the toy environments. We also compare the effect of the dimensions on the toy and complex environments. Finally, we show how to use *MDP Playground* to debug agents, to study the interaction of multiple dimensions and describe further use-cases.

1. Introduction

We begin our discussion by motivating why we need dimensions of hardness and toy environments in Reinforcement Learning.

1.1 Need for Dimensions of Hardness

Reinforcement Learning (RL) algorithms can solve many disparate tasks, such as helicopter aerobatics, game-playing and continuous control (Kim et al., 2004; Mnih et al., 2015; Haarnoja et al., 2018). However, multiple recent papers (e.g., Henderson et al., 2018; Engstrom et al., 2020; Andrychowicz et al., 2021) have shown that RL agents tend to be brittle. It has been argued that current deep RL research has been increasing the complexity of the dynamics 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 (Andersson & Doherty, 2018). We believe as a step towards understanding and designing better RL agents, research analysing the effects of common challenges faced by RL agents is of growing importance (Osband et al., 2019; Dulac-Arnold et al., 2020). To this end, we have designed a framework, *MDP Playground*, which allows such challenges to be added, in a controlled manner, to RL environments. We describe a set of such challenges in this paper which we term *dimensions of hardness* as they can be controlled independently of each other in *MDP Playground*.

1.2 Need for Toy Environments

Furthermore, a lot of the insights obtained on the *standard* environments are on very complex and in many instances *blackbox* environments. There are many different types of such environments, as many as there are different kinds of tasks in RL (e.g. Todorov et al., 2012; Bellemare et al., 2013; Cobbe et al., 2020). They are each tied to *specific* kinds of tasks. The underlying assumptions in many of these environments are that of a Markov Decision Process (MDP) (Puterman, 1994) or a Partially Observable MDP (POMDP) (see e.g., Sutton & Barto, 1998). However, there is a lack of simple and accessible problems which capture common challenges seen in RL and let researchers experiment with them in a fine-grained manner. Many researchers design their own toy problems which capture key aspects of their problems and then try to use them to gain insights on *whitebox* environments[1] because the standard *complex* environments, such as *Atari* and *Mujoco*, are too expensive or too opaque.

To sum up the disadvantages of complex environments: **1)** They are very expensive to evaluate. For example, a DQN (Mnih et al., 2015) run on an *Atari* (Bellemare et al., 2013) environment took us 4 CPU days and 64GB of memory to run. **2)** The environment structure itself is so complex that it can lead to "lucky" agents performing better (e.g., in Henderson et al., 2018). Furthermore, different implementations even using the same libraries can lead to very different results (Henderson et al., 2018). **3)** Many dimensions of hardness are concurrently present in the environments and do not allow us to independently test their impact on agents' performances.

Motivated by the above discussion on dimensions of hardness and toy environments, we introduce *MDP Playground*, a platform whose intent is to: **1)** help us to understand RL agents better on toy and complex environments. This is achieved by injecting dimensions of hardness into the environments in a fine-grained manner. This is similar in spirit to the work of Hendrycks and Dietterich (2019). **2)** "unit test" characteristics of agents on toy MDPs. The unit tests are aimed at overcoming the disadvantages of complex environments

1. We use the term *whitebox* in analogy with *blackbox*, i.e., whitebox environments are those which are accessible and where we know all the internal details of how they work.

mentioned above - the evaluation is cheap, the structure of the environments is simple and accessible and one can *independently* inject desired dimensions of hardness.

We would like to emphasise here that the toy environments in *MDP Playground* are intended as a *complement* to complex environments, and not to replace them, because they trade off compute, accessibility and control at the expense of being further removed from the real world. They may be useful to try out new ideas quickly but may not lead to immediate results on the complex environments which may be the long-term goal of research. As an example of how this may be useful, we mention here that Q-learning (Sutton & Barto, 1998) and Double Q-learning (van Hasselt, 2010) were shown as proofs of concept on tabular or small environments several years before the advent of deep RL and complex environments. Neither could be shown transfer to complex environments at the time of publication. As such, toy environments in *MDP Playground* are meant as accessible environments where the ground truth is known and where one might quickly test out whether a new idea helps against a dimension of hardness and not, e.g., to tune hyperparameters for complex environments.

The main contributions of this paper are:

- We identify and discuss dimensions of hardness of MDPs that can have a significant effect on agent performance, both for discrete and continuous environments;
- We discuss experimental designs for toy and complex environments that: 1) show insights on toy environments alone, 2) compare the effect of dimensions on toy and complex environments;
- The toy environments in *MDP Playground* provide unit testing and debugging capabilities for RL agents with the ground truth being available; a single seed of an experiment can be run in as few as 30 seconds on a single core of a laptop;
- Many of the studied dimensions of hardness stem from partial observability and the related experiments provide an analysis of how partial observability can degrade performance; to the best of our knowledge, this is the first such systematic analysis.

The paper is structured as follows. We define MDPs, NMRDPs (Bacchus et al., 1996) and POMDPs needed to understand the dimensions of hardness in *MDP Playground* and then describe the motivations for the dimensions of hardness in *MDP Playground* from a high-level and general point of view in Section 2. We describe *MDP Playground* and its components in Section 3. We provide a low-level description with details of the specific implementation of the dimensions of hardness in *MDP Playground* in Section 4. We discuss design decisions involved in implementing *MDP Playground* in Section 5. We perform experiments and analyse them in Section 6. We discuss related work in Section 7. We discuss limitations of our approach and its ethical and societal implications in Section 8. Finally, we provide conclusions and discuss future work in Section 9.

2. Dimensions of Hardness in MDPs

To identify dimensions of hardness and to distinguish between different kinds of *state* that we will encounter, we need to begin with some definitions.

2.1 MDPs, NMRDPs and POMDPs

We define an MDP as a 7-tuple $(MS, A, P, R, \rho_o, \gamma, T)$, where MS is the set of states, A is the set of actions, $P : MS \times A \times MS \to \mathbb{R}^+$ describes the transition dynamics as a probability density of reaching a state given the current state and action[2], $R : MS \times A \times MS \times \mathbb{R} \to \mathbb{R}^+$ describes the reward dynamics as a probability density of obtaining a reward given the current state, action and next state, $\rho_o : MS \to \mathbb{R}^+$ is the initial state distribution, γ is the discount factor and T is the set of terminal states. We will consider systems which also have non-Markovian rewards because in the case of some dimensions of hardness, we have rewards which depend on the past history of the system. So, we additionally define an NMRDP, a *Non-Markovian Reward Decision Process* following (Bacchus et al., 1996) which corresponds to our MDP of the system but has a different state space, NS. It will be worthwhile to distinguish between the current state of the system which is non-Markovian and its corresponding Markovian state which contains the history, to be able to fully understand the details of *MDP Playground*. A state $ns \in NS$ will represent the current state of the system and a state $ms \in MS$ will represent the Markovian state. As such, any state $ms \in MS$ would correspond to a tuple of states $(ns_0, ns_1, ...)$ from NS. We will refer to the states from NS as N-states and the states from MS as M-states. The NMRDP is defined using NS instead of MS in its definitions of P and R[3]

Finally, we define a POMDP with two additional components - O represents the set of observations and $\Omega : MS \times A \times O \to \mathbb{R}^+$ describes the probability density function of an observation given an M-state and action. Following Subramanian et al. (2020), we will use *information state* to mean the state representation used by the agent and *belief state* (Cassandra et al., 1994) as the posterior belief of the unobserved Markov state given the full observation history. POMDPs can be modelled as fully observable MDPs by considering the belief state (i.e., the posterior belief of the unobserved state given all the observations made by the agent) as an information state (Subramanian et al., 2020). As a result, the theory and algorithms for exact and approximate planning for MDPs become applicable to POMDPs and using the belief state as the information state by an agent would be sufficient to compute an optimal policy. However, since the full observation history is not tractable to store for many environments, the last few observations are used as the information state which renders it only partially observable. This is an important point to keep in mind because some of the motivated dimensions of hardness are actually due to the information state being non-Markov.[4]

We also mention here the Q^*-value (Mnih et al., 2015) and use it as an example to argue how violations of assumptions may lead to degradation in performance. For a Markov state $ms \in MS$ and action a, a policy π and r_t the reward a timestep t, Q^* is defined as: $Q^*(ms, a) = \max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \| ms_t = ms, a_t = a, \pi\right]$.

2. The default MDPs in *MDP Playground* are deterministic. In such cases, P can also be represented as $P : MS \times A \to MS$. When describing such deterministic transition functions, we will assume this to be the case and talk about P as such.
3. As a result, P and R would be overloaded symbols. Which P or R we are referring to should be clear from the context. If unclear, we will mention it in the text.
4. We have introduced MDPs, NMRDPs and POMDPs, because all the concepts - the historical state of the system (which is Markov), the current state of the system (which is non-Markov) and the observations are distinct and aid an unambiguous explanation of the details of the dimensions of hardness.

2.2 Motivations of Dimensions and Implementations

We try to identify dimensions of hardness that can be injected independently of each other by going over the many components of an MDP and motivated by the literature. We try to motivate as many as possible, to allow researchers to systematically study and gain new insights. To make the understanding of these dimensions of hardness more concrete, we briefly also describe the implementation of the dimensions of hardness in *MDP Playground* in this section. To be able do so, we briefly also describe the toy discrete and continuous environments in MDP Playground. The text in this section is more of a *top-down / dimensions of hardness*-oriented presentation. For a more detailed and lower-level description of these environments and their dimensions of hardness, we refer the reader to Section 4, where additional dimensions which are mostly trivial to understand are described as well.

In the discrete case, the N-state and action spaces of the system, NS and A, contain *categorical* elements that are labelled from 0 to $\|NS\| - 1$ and 0 to $\|A\| - 1$ respectively, however, there is no ordering. As such, elements of both NS and A are 1-dimensional integers. P is formulated as a graph such that each N-state is a vertex and elements of NS are connected to other elements of NS through elements of A which form the edges. All the vertices in the graph are set to have the same degree which lends P a uniform structure: this avoids having "lucky" regions in the environment. In the continuous case, environments correspond to the simplest real world task we could find: moving a rigid body to a target point (Klink et al., 2019). P is formulated such that each action dimension affects the corresponding space dimension: for a given N-state, ns, the derivative of N-state, $\dot{ns}$, is set to be equal to the action applied for *time unit* seconds on the body, in the simplest version of the environment. This is integrated over time to yield the next state.

We now describe many of the dimensions of hardness and briefly their implementations in *MDP Playground*. Please see Figure 1 for guiding visualisatons.

2.2.1 Reward Delay

In many situations, agents perform an action that leads to a rewarding trajectory but the agent is only rewarded in a *delayed* manner (see e.g., Arjona-Medina et al., 2019) (see Figure 1e). For example, shooting at an enemy ship in *Space Invaders* leads to rewards much later than the action of shooting. Any action taken and the resulting trajectory afterwards 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 information state features and actions. In *MDP Playground*, the reward can be artificially delayed by a non-negative integer number of timesteps, d.

2.2.2 Sequence Length

In many environments, a reward is obtained for a *sequence* of N-states resulting from a sequence of actions taken and not just the current N-state and action (see Figure 1d). A simple example is executing a tennis serve, where one needs a sequence of actions which leads to a rewarding N-state trajectory that results in a point, e.g., if an ace was served. In contrast to *delayed rewards*, rewarding a sequence of N-states resulting from a sequence of actions addresses the trajectory followed as being consequential to obtaining a reward.

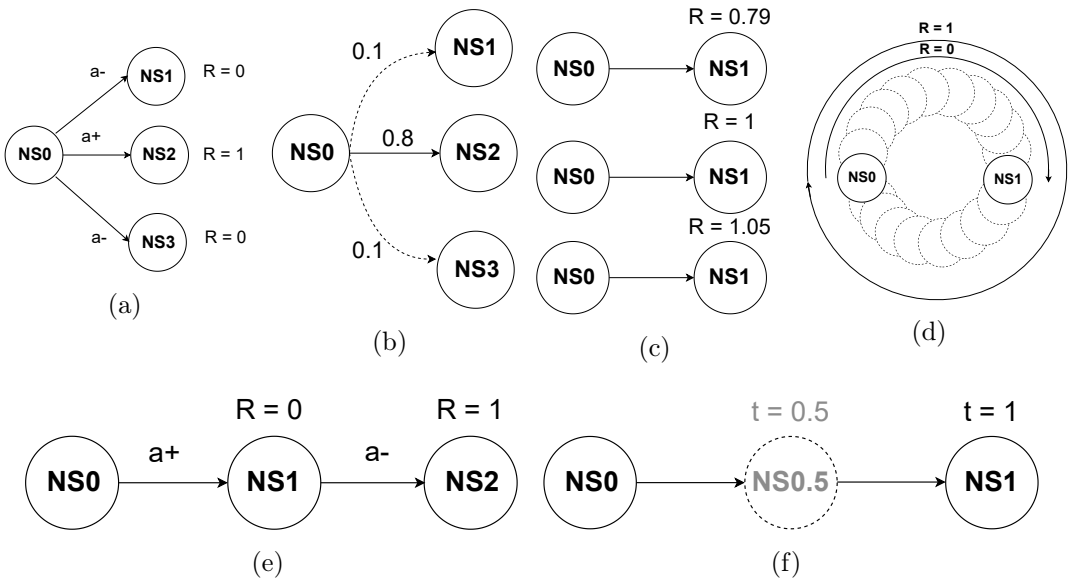


Figure 1: Some of the dimensions of hardness depicted visually to aid understanding. Please note that the sub-figures do not use accurate graphical notation, but are rather meant to be guiding visualisations about how a particular dimension of hardness works. We depict states of the system under consideration, i.e. the N-states as defined in the text. Not all N-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". (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 an N-state which is not the true next N-state. (c) R **noise**: The *same* transition leads to different rewards. (d) **Sequence Length**: Assume the task is to move in a circle, then executing a sequence of actions starting from N-state NS0 leading to N-state NS1 executes only a semi-circle and leads to 0 reward (intermediate N-states along the sequence shown in dashed lines). Executing the full circle and returning to N-state NS0 leads to a reward of 1. *Note*: there may be multiple such sequences corresponding to the task, but the emphasis here is on needing more than 1 action, i.e., a sequence. (e) 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 in between. *Note*: the reward would have been achieved a step later irrespective of which action was performed in the second step. (f) **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 N-state.

Learning sequences of actions as macro-actions or options is important, e.g., in Hierarchical Reinforcement Learning (HRL). Sutton et al. (1999) present a framework for temporal

abstraction in RL to deal with such sequences. Regarding the Q^* value, this means that if an incomplete 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 reach a rewarding N-state trajectory and not just a part of the sequence. While agents can perform well asymptotically in the face of both delays and sequences, using the correct information state would very likely lead to much better sample efficiency and more stable learning. In *MDP Playground*, for discrete environments, only specific sequences of N-states of positive integer length n are rewardable. Sequences consist of non-repeating N-states allowing for $\frac{(\|NS\|-\|T\|)!}{(\|NS\|-\|T\|-n)!}$ sequences. For the continuous environment of moving to a target, n is variable.

2.2.3 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 (Gaina et al., 2019), the reward is 0 more frequently, e.g., in continuous control environments where a long trajectory is followed and then a single non-zero reward is received at its end. This also holds true for the example of the tennis serve above. In *MDP Playground*, for discrete environments, the *reward density*, *rd*, is defined as the fraction of possible N-state sequences that are actually rewarded by the environment. For continuous environments, density is controlled by having a sparse or dense environment using a *make_ denser* configuration option. When the reward is dense, the reward for the current time step is the distance travelled towards the target since the last step. When the reward is sparse, a reward is only handed out when the agent reaches the target.

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, Todorov et al., 2012), are usually dense (such as the negative squared distance from the target). This lets the algorithm obtain a dense signal in space to guide learning, and it is well known (Sutton & Barto, 1998) 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* for discrete toy environments, the environment gives a fractional reward if a fraction of a rewardable sequence is achieved and it does so for all the possible rewardable sequences an agent is currently on.

2.2.4 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 (Thrun et al., 2005). In *MDP Playground*, for discrete environments, we define *transition noise* $t\_n \in [0, 1]$; with probability t_n, an environment transitions uniformly at random to an N-state that is not the *true* next N-state given by P. We define *reward*

noise as being normally distributed: $r\_n \sim \mathcal{N}(0, {\sigma^2}_{r\_n})$, and is added to the *true* reward. For continuous environments, both t_n and r_n are normally distributed and are directly added to the N-states and rewards.[5] [6]

2.2.5 Irrelevant Features

Environments also tend to have a lot of *irrelevant features* (Rajendran et al., 2018) that one need not focus on. This holds for all kinds of learners including Neural Networks (NNs). However, NNs can even fit random noise (Zhang et al., 2017) and having irrelevant features is likely to degrade performance. For example, in certain racing car games, even 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 is concatenated and appended to the N-state and action spaces each. This dimension causes the dimensionality of P, associated with the NMRDP, to increase to 2. The first dimension corresponds to dynamics relevant to the reward, P_{rel}, while the second dimension has dynamics P_{irr}. P_{rel} and P_{irr} are independent of each other and when concatenated together form P, the transition function visible to the agent. However, only the discrete dimension corresponding to P_{rel} is *relevant* to calculate the reward function. Similarly, in continuous environments, new irrelevant dimensions are added to NS and A that are not considered relevant to the reward. These irrelevant dimensions are visible in the observations which come from O.

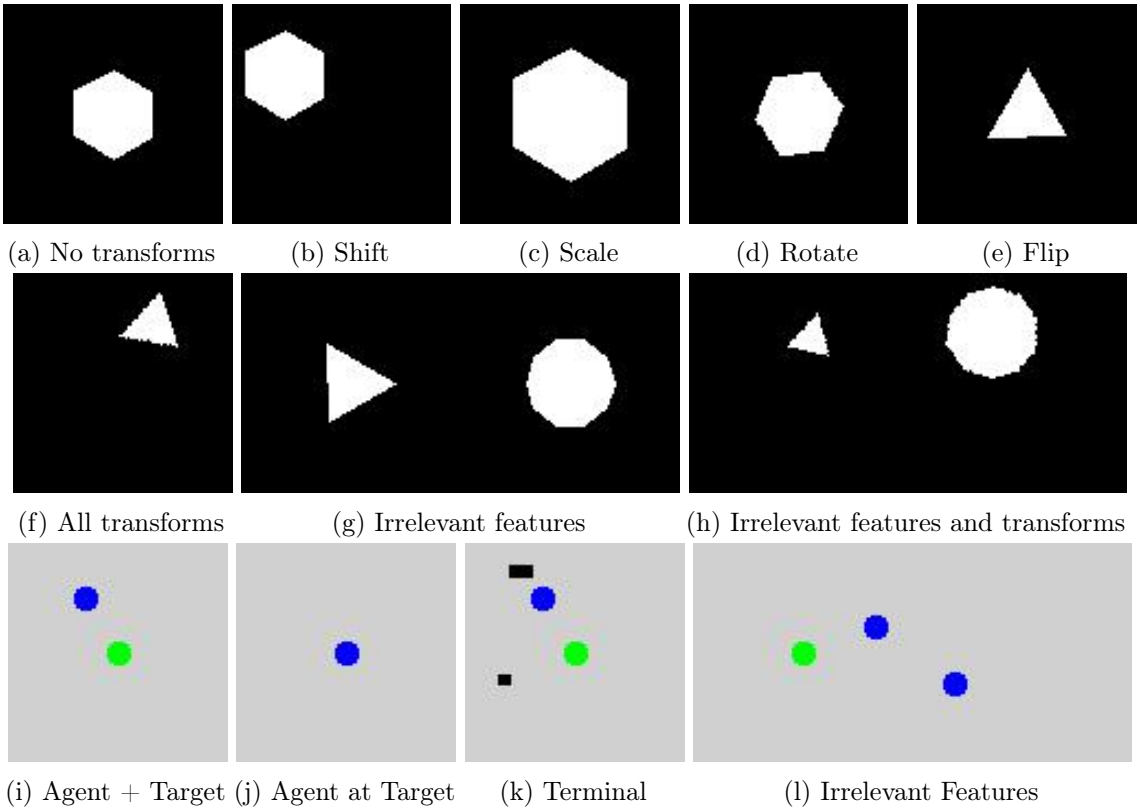
2.2.6 Representations

Another aspect is that of *representations*. The same underlying N-state may have many different external representations/observations from O, e.g., *feature* space would be the N-space while *pixel* space could be O in Mujoco. Similarly, Atari games can use the underlying RAM state or images. For images, various image transformations (*shift*, *scale*, *rotate*, *flip* and others; Hendrycks & Dietterich, 2019) may manifest as observations of the same underlying N-state and can pose a challenge to learning. In *MDP Playground*, for discrete environments, when image representations are enabled, each categorical N-state is associated with an image of a regular polygon which becomes the externally visible observation $o \in O$ to the agent (see Figure 2a-2h). This image can be further 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 (see Figure 2i-2l).

2.2.7 Diameter

The *diameter* of an MDP, i.e., the maximum distance between 2 states, is another significant dimension of hardness affecting performance and reachability of states (Jaksch et al., 2010). In our case, we define the diameter for the states of the NMRDP, i.e., the N-states. In *MDP*

5. The noise functions can actually be passed as Python lambda functions and the described normally distributed noise is just the default setting.
6. An important detail to note is that in case the actions are noisy, the effects of these noisy actions are seen in the resulting N-state trajectory and this is why in *MDP Playground*, we reward sequences of N-states and not sequences of N-states *and* actions.



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

(f) All transforms (g) Irrelevant features (h) Irrelevant features and transforms

(i) Agent + Target (j) Agent at Target (k) Terminal (l) Irrelevant Features

Figure 2: (a)-(f): **Discrete environments**: When using the dimension of hardness *representations* with image representations, each categorical N-state corresponds to an image of a polygon (if the states were numbered beginning from 0, each N-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 N-states 3 and 0. (g)-(h): **Discrete environments with irrelevant features**: When irrelevant features are enabled alongside image representations, an additional "irrelevant" image is stitched to the right of the relevant part and the same transforms are applied to each part. (i)-(k): **Continuous environments**: When using the dimension of hardness *representations* with image representations in continuous environments, the agent is shown as a blue circle, the target point as a green circle and terminal N-states are black. The agent is rendered after the target as seen in the second sub-figure. (l): **Continuous environments with irrelevant features**: As for the discrete environments, the relevant part with 2 dimensions is the left half of the image and the irrelevant is the right half. The agent exists in all 4 dimensions and is visible in both halves of the image while the target exists in only the 2 relevant dimensions and is visible only in the left half.

Playground, for discrete environments, for *diameter* $= d$, the set of N-states is set to be a d-partite graph, where, if we order the d sets as $1, 2, .., d$, N-states from set i will have actions

leading to N-states in set $i+1$, with the final set d having actions leading to N-states in set 1. The number of actions for each N-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 of hardness *state space max* sets the bounds of the N-state space to $\pm$*state space max* and the $\textit{diameter} = 2\sqrt{2}$ *state space max*.

2.2.8 Time Unit and Action Range

For continuous control problems, we now describe two dimensions of hardness together. Firstly, *action range* (Kanervisto et al., 2020), or the action space, is the range of actions that the agent may take. For example, in the *Gym* environment *Pendulum*, the action applied is a scalar representing the torque applied to the free end of the pendulum and its range is $(-2.0, 2.0)$. The action range can be expected to have a significant impact on an agent's performance. For example, if the action range is too large, the exploration may take too long to find the optimal actions. On the other hand, if, for example, one is designing an MDP to solve a problem, if the action range is set to be too small, it may not contain the desired actions one is hoping to optimise for. Similar to the action range, the *time unit*, the discretisation of time (see Figure 1f) can be expected to have a significant impact on an agent's performance. For example, Biedenkapp et al. (2021) shows that there is an optimal number of times an action needs to be repeated. Similar to the action range, having too small a time unit would not capture such optima and having too large a time unit would lead to a much harder exploration problem. The *time unit*, t, sets $P(ns, a) = ns + \int_t P_{cont}(ns, a)\, dt$ where P_{cont} is the time-derivative of the dynamics function. The *action range* sets $A \subset \mathbb{R}^a$ where a is the action space dimensionality. In order to avoid any confusion, we must clarify here that it is not our intention to suggest that standardised or benchmark problem definitions be changed to show new state of the art performances because changing the problem would invalidate the benchmark. What we want to say is that: 1) if the problem definition is given, e.g., a fixed action range, one might want to search in a subset of this range to learn quicker; the RL algorithm could decide to do so on a meta-level as is done with, e.g., *reward shaping* on a meta-level (Zheng et al., 2018; Hu et al., 2020; Zou et al., 2021); one could call it *action space shaping* when doing it with the action space; 2) in some cases, the problem at hand may not have been defined in a desirable manner, leading to solutions which the people trying to solve the problem may not desire, and in such cases one might want to, e.g., in the case of a fixed action range, search outside this range if physically possible. This second case is more likely to occur for completely unknown or new environments where the problem definition itself may be unclear. We believe there is a difference between the problem people want to solve and the MDP formulation they use to write their problem in a way for standard RL agents to work. For example, the discount factor, γ, is frequently treated as a hyperparameter even though it is part of the MDP formulation and changing it changes the MDP being solved (Xu et al., 2018; Tessler & Mannor, 2020; François-Lavet et al., 2015). As such, Xu et al. (2018) tuned γ with meta-gradients to try and solve their target problem. Another example is that of Runge et al. (2019). They aimed to learn to design RNA structures, and formulated their MDP encoding such that it had hyperparameters that modified the state space and shaped the reward function. These hyperparameters of the MDP could be tuned alongside the RL algorithm's hyperparameters in order to find a combination of MDP

encoding and RL algorithm that solved the problem as well as possible; 3) finally, one might just want to play around with problem definitions to observe the effects on their agents, we do not propose that researchers release such results claiming that they improved the performance on the original benchmark problem, but such studies can still yield insights into the RL agent's behaviour.

2.2.9 Target Radius

Finally, an additional dimension of hardness for continuous control problems we describe is *target radius* (Klink et al., 2019) which is a measure of the distance from the target within which we consider the target to have been successfully reached. This dimension of hardness is a fairly common use-case in continuous control environments, e.g., in *OpenAI Gym*. The *target radius* sets $T = \{ns \mid \|ns - ns_t\|_2 < \textit{target radius}\}$, where ns_t is the target point.

We now summarise the dimensions of hardness identified above (with the (PO)MDP component they directly impact in *MDP Playground* in brackets):

- Reward Delay (R, MS)
- Sequence Length (R, MS)
- Reward Density (R)
- Stochasticity (P, R)
- Irrelevant Features (O)
- Representations (O)
- Diameter (P, NS)
- Time Unit (P)
- Action Range (A)
- Target Radius (T)

Only selected dimensions of hardness are included here, to aid in understanding and to show use-cases for *MDP playground*. Trying to identify as many dimensions as possible has led to a very flexible platform and Table 1 lists all the dimensions of hardness and important configuration options of *MDP Playground*. We would like to point out that what dimensions of hardness are important largely depends on the domain. 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 (Tobin et al., 2017). To maintain the flexibility of having as many dimensions of hardness 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 of hardness.

3. MDP Playground

MDP Playground consists of 4 components: 1) Toy Environments 2) Complex Environment Wrappers 3) Experiments 4) Analysis. We now briefly describe the first two parts and describe how to experiment and analyse with *MDP Playground* in detail in Section 6.

3.1 Toy Environments

The toy environments are *Gym environments*. They are cheap and encapsulate all the identified dimensions of hardness. The components of the MDP can be automatically generated according to the configured dimensions of hardness or they can be user-defined. We discuss how the automatically-generated MDPs function in detail in Section 4 and the corresponding algorithm in Algorithms 1 and 2 in Appendix B.3. Further, the underlying Markovian M-state is exposed in an *augmented_state* variable, which allows users to design

agents that may try to identify the true underlying MDP M-state given the observations. Design decisions regarding the MDPs are discussed in detail in Section 5.

3.2 Complex Environment Wrappers

We further provide wrappers for *Gym* environments which can be used to inject many of the dimensions of hardness into complex environments such as *Atari*, *Mujoco*, *ProcGen* (Cobbe et al., 2020) or any other *OpenAI Gym* environment. These can be used to analyse agent behaviour in response to such dimensions of hardness.

3.3 Code Sample

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

ae = gym.make("QbertNoFrameskip-v4")
env = GymEnvWrapper(ae, **config)
```

An environment instance is created as easily as passing a Python `dict`. Users can set the values of dimensions in the `dict` with one line of code for each dimension. As mentioned before, users only need to provide dimensions they are interested in. The rest are set to *vanilla* values. By *vanilla*, we mean the dimension being set to its default value which turns it off in the case of most dimensions, e.g., setting delay, $d = 0$ or sequence length, $n = 1$. Please see Table 1 for a list of default values of the dimensions. In the code sample below, we create a toy *and* a complex environment using the wrapper with the same dimensions of hardness.

3.4 Experiments and Analysis

The GitHub repository[7] describes how to run experiments and how to analyse them with plots in a Jupyter Notebook. Section 6 describes some experiments and analyses of this kind.

3.5 Very Low-Cost Execution

Toy 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 of hardness used. Nevertheless, to give the reader an idea of the runtimes involved, DQN delay experiments from Section 6 performed using Ray RLLib (Liang et al., 2018) with a network with 2 hidden layers of 256 units each) took on average 35s for a *complete* run of DQN for 20 000 environment steps. In this setting, we restricted Ray RLLib and the underlying Tensorflow to run on *one core of a laptop* (core-i7-8850H CPU – the full CPU specifications can be found in Appendix H). This equates to roughly 300 minutes for the *entire* delay experiment shown in Figure 9a which was plotted

7. `https://github.com/automl/mdp-playground`

using 500 runs (100 seeds $\times$ 5 settings for *delay*; these 500 runs could also be run in an embarrassingly parallel manner on a cluster e.g. in less than 2 minutes with 500 CPU-cores in total with one core for each seed). Even when using the more expensive continuous or image representation environments, runs were only about 3-5 times slower.

4. MDPs in MDP Playground

In this section, we provide a more *bottom-up/implementation*-oriented presentation of *MDP Playground* and its dimensions of hardness. We begin with describing the discrete toy environments (corresponding algorithm in Algorithm 1 in Appendix B.3). We then describe the continuous toy environments (corresponding algorithm in Algorithm 2 in Appendix B.3). Finally, we describe the complex environment wrappers.

4.1 Discrete Toy Environments

In the discrete toy environments, NS consists of the set of states of the system, i.e., it contains no history. For $diameter = d$, NS is set to be a d-partite graph, where, if we order the d sets as $1, 2, .., d$, N-states from set i will have actions leading to all N-states in set $i+1$, with the final set d having actions leading to all N-states in set 1. The number of actions for each N-state is, thus, $(number\ of\ states)/(d)$. In setting the configuration, the user sets *action space size* (which is equal to the size of each independent set) and the *diameter*, which indirectly sets the size of NS. The N-states are labelled from 0 to $\|NS\| - 1$ which we term *global labelling*. The N-states in each independent set are labelled from 0 to $\|NS\|/d - 1$ within that set (this is obtained by modulo division of the global label with $\|A\|$ which we term *local labelling*. They are *categorical* in that they are not ordered. As such, elements of both NS and A are 1-dimensional integers. In the case of the vanilla environment, N-states, M-states and observations coincide. The terminal N-state density can also be set by the user. The initial state distribution, ρ_0, is currently fixed to be uniform over the non-terminal N-states. The environments' episodes are restricted to a maximum of 100 timesteps by default. As an example, Figure 3 shows a discrete toy environment with 6 N-states in total, partitioned into sets of 2 each. Please see the text in the figure caption for details on the example.

Based on the value the *reward density* rd and the *sequence length* n are set to, automatic generation of the MDP leads to randomly selecting $\lfloor rd * number\ of\ possible\ sequences \rfloor$ (where $\lfloor\ \rfloor$ denotes the floor function) of non-repeating N-state sequences of length n to be rewardable. While there are transitions possible to terminal N-states, the reward density and sequence lengths only consider non-terminal states to determine rewarding sequences, allowing for $\frac{(\|NS\|-\|T\|)!}{(\|NS\|-\|T\|-n)!}$ sequences. The reward handed out for following a rewardable sequence is 1. Based on the value the reward delay d is set to, we delay the reward at any timestep to be handed out d steps later. In the case of reward delay and sequence lengths, the N-states remain the same as for the vanilla environment, but the M-states consist of N-state sequences of length n from d timesteps in the past. The observations remain the same as the N-states. Finally, when $make\_denser$ is $True$ for discrete toy environments, the environment gives a fractional reward if a fraction of a rewardable sequence is achieved and it does so for all the possible rewardable sequences an agent is currently on.

An additional point to note about episodic rewards in the presence of *delay* and *sequence length* in the toy environments is that the optimal episodic reward changes in the presence of these dimensions of hardness. In the case of delays, an agent cannot get the delayed reward at the end of an episode if the episode terminates before the delayed reward is handed out. In the case of sequence lengths, it is a bit tricky to handle rewardable sequences because the agent might be on multiple rewardable sequences at the same time and it is a very hard problem to determine the optimal episodic reward for larger sequence lengths in such cases. Since we are usually concerned only with agents trying to execute non-overlapping rewardable sequences, we provide the option *reward_every_n_steps* which is turned on by default. With this option, we hand out rewards for executing a rewardable sequence only if it finishes executing in a timestep of the environment that is divisible by n if the sequence length is n and there is no reward handed out if a rewardable sequence is completed at other timesteps (this does not affect other dimensions, e.g., there may be independent reward noise after every step or a terminal state reward at the end of the episode as described later). This means that even if the agent is on multiple rewardable sequences at once, they only get rewarded for one of them and the optimal episodic reward for the toy environment with 100 steps is $100/n$. For example, suppose the sequence length were 4. If an agent in N-state ns_0 at timestep 0 started executing a rewarding sequence and then reached N-states ns_1, ns_2, ns_3 which are all also the starting points of other rewarding sequence, it could then be on 4 rewarding sequences at once for the given sequence length of 4. If the agent successfully executes all 4 sequences, it could be rewarded at timesteps $4, 5, 6$ and 7 (if *reward_every_n_steps* were *False*). This complicates the optimal reward calculation because given all the generated rewarding sequences, it would be hard to calculate the optimal sequence of overlapping rewarding sequences the agent needs to be on. However, if *reward_every_n_steps* were *True*, the agent would only be rewarded after every n^{th} step, so only after step 4 in the above example. As a result, the optimal reward calculation in this case is fairly easy. It would be the episode length, i.e. 100, divided by n. This holds even when the diameter is > 1, because an equal number of rewarding sequences per independent set of N-states is selected. However, this does change the reward scale of the optimal episodic reward. To combat these changing episodic reward scales, we normalise the plots to have episodic reward scales of 100. This normalisation will be described in Section 6.

Transition noise t_n sets the fraction of times we transition to a connected N-state that is not the true next N-state as defined by the transition function P for the current N-state, ns and action a. In the presence of *diameter*, this means that the next N-state belongs to the next independent set. *Reward noise* σ_{r_n} adds a normally distributed reward with standard deviation σ_{r_n} to the obtained reward at each time step. Stochasticity, thus, affects all of N-states, M-states and the observations.[8].

When *irrelevant features* are turned on, the user sets the *action space size* with a list of length 2. The first number in the list is used as the normal/relevant action space size as specified above. The second number is used in the same way to set the irrelevant action space size and the size of the irrelevant N-state space size using the same diameter as for the relevant

8. Since the noise generation process is based on a pseudo-random number generator (PRNG), one could in principle claim that the M-state should, in the case of stochasticity have, additionally, the state of the PRNG as well because the next M-state depends on it. However, our intent is to treat the noise as *true* noise and, as such, we ignore this technicality.

spaces. The N-states and observations have the additional irrelevant dimension appended to the original N-states and observations, respectively. The M-states in this case are unchanged from when irrelevant features would be turned off. The dynamics function associated with the sub-space relevant to the reward, P_{rel} and the dynamics function associated with the sub-space irrelevant to the reward, P_{irr}, are independent of each other. They are concatenated together to form the dynamics function, P associated with the NMRDP. The dynamics associated with the MDP, however, are only P_{rel}[9]. The irrelevant sub-space also does not have any terminal states. Finally, the same transition noise is applied to the irrelevant sub-space as the relevant sub-space.

When turning on *image representations*, the observations become 2-dimensional greyscale images. The integer representation, n, of the underlying N-state space is associated with a polygon of $n + 3$ sides as shown in Figure 2 and this is used as the observation[10]. This polygon is displayed in the centre of a 100×100 pixel image with the radius of the circle which circumscribes the polygon as 20 pixels. Turning on *image representations* changes only the observations. All of the dimensions of hardness discussed so far remain largely the same, otherwise. For example, the transition noise is only applied when transitioning within the underlying N-state space. There is no *additional* noise in the image space, when transition noise is turned on, due to image representations also being turned on. In the case of irrelevant features also being turned on at same time as image representations, however, it may be ambiguous as to how image representations of the relevant and irrelevant sub-spaces are concatenated. In this case, we do so along axis 0, i.e., the "X-axis" of the image (in other words, width-wise) with the relevant part always being on the left side of the image. Additionally, in the case when the *diameter* is > 1, there is more than one independent set and it may be ambiguous as to whether the global or the local label is used to create the polygons in the default image representations. We do so using the global label to be able to distinguish different N-states visually.

As image observations are associated with different kinds of transforms such as *shift*, *scale*, *rotate* and *flip*, we allow these transforms to be applied *when image representations are turned on*. While the default observation is of a polygon centred in the image with a fixed size, further turning on these transforms results in the observations associated with the *same* underlying N-state to vary as shown in sub-figures 2b-2f.[11] This allows the representation learning aspect of the environment to potentially also be made harder. We also allow the quantisation of the *shift* and *rotate* transforms and the range of the *scale* transform to be adjusted to have more fine-grained control over these. Finally, in the case of irrelevant features also being turned on at the same time as transforms for image representations, we apply the same transforms to the irrelevant part of the image as to the relevant part and then concatenate along the "X-axis" of the image with the relevant part always being on the left side of the image.

9. One could, naturally, also change the M-states and MDP to include the irrelevant features and still remain Markovian. However, in our MDP, we have decided to retain the minimal amount of information needed to obtain rewards.
10. Since the above image representations might only be useful for vertex detectors of polygons, we allow users to provide a directory with custom images that can be used as the representations for the N-states in case they have better ideas to test the representation learning aspect of their algorithm.
11. We take care that the polygon does not translate out of the image and also print warnings in case the polygon size ends up being "too small" or "too large".

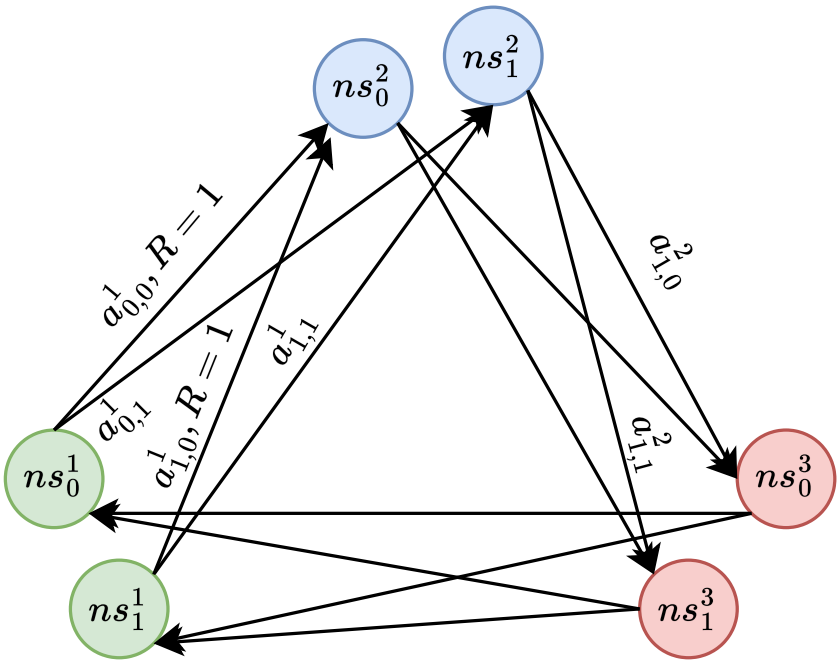


Figure 3: An example discrete toy environment with 6 N-states with diameter = 3[12], divided into 3 independent sets with 2 N-states each. Each N-state has 2 possible actions in its action space. The N-states have the set index for the set they belong to as superscript and the N-state index *within* the set (the *local label* as described in the text) as subscript. The actions have, additionally, the action index for the N-state they belong to as a second subscript. From each N-state in set 1, one can take an action to enter each N-state in set 2. The reward density is set to 1/6 and the sequence length to 1. Based on these settings, N-state sequences of length 1 would be rewardable, i.e., one needs to reach a rewardable N-state to obtain a reward of 1. Automatic generation of the MDP leads to randomly selecting 1 out of the 6 N-state sequences of length 1 being picked as rewardable. In this example, this leads to the only rewarding transitions being from ns_0^1 to ns_0^2 and ns_1^1 to ns_0^2 with the rewardable sequence of N-states being ns_0^2. The edges to it are labelled with reward $R = 1$, all other rewards are 0. Not all edges are labelled to avoid clutter. Not all dimensions of hardness and no terminal N-states are included here for simplicity.

Finally, the *reward scale* scales the (possibly noisy) reward at each time step. After this, the *reward shift* is added to the reward at each time step. The *terminal state reward* scaled by the *reward scale* is added to the reward at the end of the episode.

12. The diameter is 3, and not 2 as may seem at first sight, because the route from a starting vertex to the other vertices in the same set are of minimum length 3.

The *Gym* environment returns observations as is usual. We, additionally, also return the whole N-state sequence starting from $d + n + 1$ time steps in the past in *Gym*'s `info` variable, so that the agent might use these to validate its learning in different ways, e.g., checking if it can create the underlying M-state given the observation.

4.2 Continuous Toy Environments

We now describe the continuous toy environments in *MDP Playground*.

The continuous toy environments consist of a point mass which is controllable using the agent's actions. The user can set the mass of the system using the *inertia* configuration of the system. Similar to the discrete environments, NS consists of the (non-Markovian) state of the system. In the case of the continuous toy environment, the elements of NS and A belong to $\mathbb{R}^{n_{dim}}$ where n_{dim} is the dimensionality of the system, which is set by the user. The user also sets the bounds of the system by setting the *state space max* and *action space max* options. The dimensionality of NS and A is equal and each action dimension controls each N-state space dimension. The user can set the *transition dynamics order*, n_P, of the system. This sets an action a, when applied, to set the n_P^{th} order derivative of the position of the mass divided by the inertia for *time unit* seconds. In the simplest case of $n_P = 1$ and $inertia = 1$, this sets the velocity of the point mass. Or, for example, in the case of $n_P = 2$, this sets the acceleration of the point mass. The *time unit*, t, then sets $P(ns, a) = ns + \int_t P_{cont}(ns, a)\, dt$ where P_{cont} is the time-derivative of the dynamics function. In this case where a is constant for a fixed timestep, for current N-state, ns, the equations of motion can be analytically derived in closed form to be $ns^i_{t+1} = \sum_{j=0}^{n_P - i} ns_t^{i+j} \cdot \frac{1}{j!} \cdot time\_unit^j$ where an integer in the superscript represents the order of the derivative and not exponentiation. For example, if $n_P = 1$, the equation is $ns = ns + ns^1 * t$, i.e., the next N-state is equal to the current N-state plus ns^1 where $ns^1 = a/inertia$, is set equal to the action, a. As in the case of discrete toy environments, the continuous toy environments' episodes are restricted to a maximum of 100 timesteps by default.

The reward function of the continuous environments is such that there is a *target point* that needs to be reached. The difference between the distance of the current N-state from the target point and the previous N-state from the target point is handed out as the reward at each time step.[13] Terminal N-state regions can be set by the user by providing *terminal_states* which are n_{dim}-dimensional points that specify centres of hypercube sub-spaces. Entering these hypercubes ends the episode.[14] The length of the side of the hypercube is set by *term_state_edge*. The initial N-state of the system is set to be sampled uniformly from non-terminal regions of the space.

13. There is an additional reward function, *moving along a line* as opposed to moving to a (target) point but this is still in the experimental phase. This is an attempt at having a *sequential* task in the continuous environment and is described in the experimental dimensions section in Appendix A.
14. The reason sub-spaces are used to implement terminal states in the continuous toy environments, as opposed to randomly sampling them from the whole N-state space as is the case with the discrete toy environments, is that, in principle, randomly sampling terminal states in continuous spaces would mean they are virtually impossible for the agent to reach. In practice, though, due to the finite precision of floating point numbers, one could contemplate doing the latter. However, we believe it would be a lot of overhead for little gain and so decided to not implement it in this manner.

The *reward delay* in the continuous toy environments works as in the discrete ones. The *sequence length* is variable for the task of reaching a target point and depends on the actions taken, so it cannot be set by the user. The *reward density* can be controlled in the sense of whether the reward is handed out after each time step or only once the target point is reached. This is done by turning the configuration option *make_denser* on or off. The *transition noise* differs from the discrete environments. The *transition noise*, σ_{t_n}, and *reward noise*, σ_{r_n}, both add normally distributed noise with standard deviation σ_{t_n} or σ_{r_n} to the N-state or reward, respectively, at each time step. Stochasticity, thus, affects all of N-states, M-states and the observations.

Irrelevant features can be specified by using the *relevant indices* option and specifying a dimensionality of the system, n_{dim}, which is greater than the number of relevant indices. So, the continuous environments can have more than 1 relevant dimension and more than 1 irrelevant dimension. In contrast to the discrete toy environment, the relevant and irrelevant parts have the same transition dynamics. The only difference between the relevant and irrelevant parts in the continuous case is that the former are considered for calculating the reward, i.e., the dimensionality of the target point is equal to the number of relevant dimensions and the agent needs to try and reach the target point in the relevant sub-space of the system. The irrelevant sub-space also does not have any terminal regions. Irrelevant features, thus, affects the N-state and observations but the M-states are unchanged from when irrelevant features would be turned off.

Image representations are only available for 2-dimensional spaces. They represent the point mass as a blue circle with radius 5 pixels and the target point as a green circle with radius 5 pixels. Terminal regions are represented as black and the background is grey as shown in Figures 2i-2l. They only change the observations of the system and do not affect the N-states or the M-states. In case irrelevant features are also turned on, as in the case of the discrete environments, there are relevant and irrelevant parts to the image that are stitched together along the "X-axis" with the relevant part on the left (each part is fixed to be 2-dimensional). We do not allow transforms for image representations for continuous environments as there would be no way to distinguish between an action and a *shift* transform.

The *target radius* is the distance from the *target point* where it is considered to have been reached. In addition to the user set terminal regions, reaching the target point ends the episode. The *action space max* option mentioned above sets the *action range* of the system. The user can additionally also opt to add an *action loss weight* to penalise the L2-norm of the actions taken during the control task. This subtracts the *action loss weight* multiplied by the L2-norm of the action from the reward at the current time step.

Finally, the *reward scale, reward shift* and *terminal state reward* work in the same way as for the discrete toy environments.

4.3 Complex Environment Wrappers

We now describe the complex environment wrappers which allow a subset of the dimensions of hardness of *MDP Playground* to be controlled in complex environments such as *Atari, Mujoco* and *Procgen*. They function in largely the same way as the toy environments, however, they only implement a subset of the dimensions of hardness. An overview of which of these are implemented for complex environments can be found in Table 1.

The *reward delay* works the same way as in the toy environments, except towards the end of the episode, when any delayed reward that was not yet handed out is handed out at the end of the episode. This is to maintain the same episodic reward scales for the environment when comparing across delays. For the toy environments we decided to normalise the episodic rewards post hoc (i.e., only in the plots) and not hand out all the delayed reward at the end of an episode because 1) we know the optimal rewards for the toy environments whereas we do not know the optimal rewards for complex environments; 2) rewards in the toy environment experiments we performed were much denser than the complex environments and handing out all the delayed reward at once in the toy environment would bias the terminal samples stored in the replay buffer a lot compared to the complex environments.

The *transition noise* for discrete environments is applied by choosing an action that is not the selected action from the action space.[15] The *transition noise* for continuous environments and the *reward noise* for both discrete and continuous environments are applied in the same way as for the toy environments. The *irrelevant features* for continuous environments creates a toy continuous environment (with the same dynamics as the ones described above and with its own dimensions of hardness which are set to their defaults if not set) whose N-states are appended to the N-states of the continuous environment for every transition and whose reward is ignored. The *reward scale, reward shift* and *terminal state reward* work in the same way as for the toy environments.

The *action range* and *time unit* wrappers are specific to *Mujoco*. For both of these dimensions of hardness, the scalar value passed in the configuration is used to multiply the base environments' default values for that dimension of hardness. With respect to the *action space max* (which controls the *action range*), the new action range is achieved by multiplying the base *Gym Mujoco* environments' `action_max` and `action_min` by the *action space max* passed in the configuration for the wrapper. This works with any *Mujoco* environment.

With respect to the new *time unit*, it is achieved by multiplying the base *Gym Mujoco* environments' `frame_skip` and not *Mujoco*'s internal timestep. This means that if the base environments' `frame_skip` is 5, for example, the smallest new *time unit* achievable using our wrapper would be 1/5 times the default time unit of the base environment and that larger time units would be integral multiples of this smallest achievable time unit. We decided not to achieve the new *time unit* by modifying *Mujoco*'s internal timestep, although this would have technically been possible, because that would change the numerical integration done by *Mujoco* and thus would introduce more noise into the dynamics of the environment. Further, while this wrapper also works with any *Mujoco* environment in the way described above, we would advise the reader to be careful that this might interfere with the reward function of the base environment. For the environments *HalfCheetahV3, Pusher* and *Reacher*, since changing the *time unit* to be 5 times smaller, for example, means that we get 5 rewards in the original amount of *real* time per time step and these rewards are of the same scale, we multiply each of the 5 rewards by 0.2 to achieve *episodic* reward on the same scale as the original environment. For *HalfCheetahV3*, for example, the rewards are of the same scale in the new time unit because its reward consists of 2 parts: the velocity and the control cost. So, for a given velocity and given applied control having rewards 5 times more frequently

15. The transition noise for the discrete complex environments wrappers is modelled in the action space and not in the state space (as was the case for the toy environments), as it is not possible to know all the possible next discrete states in advance for the complex environments.

Table 1: Dimensions of hardness and additional configuration options in MDP Playground with default values. For the wrappers, + next to a ✓signifies that the dimension only works on the *continuous* case, ∗ that it only works on *continuous Mujoco* environments. (*Note*: Formatting and colour of the text: **bold** dimensions signify applicability on both continuous and discrete toy environments, plain text: only on discrete toy, *italic*: only on continuous toy, coloured ones have wrappers implemented: blue: the wrappers can be used any environment, red: only with *continuous* environments, green: only with *Mujoco* environments.)

Dimension of Hardness	Discrete	Continuous	Wrapper	Default
Reward Delay	✓	✓	✓	0
Sequence Length	✓			1
Reward Density	✓			0.25
Transition Noise	✓	✓	✓	0
Reward Noise	✓	✓	✓	0
Irrelevant Features	✓	✓	✓+	False
Image Representations	✓	✓		False
Image Transforms	✓			None
Reward Scale	✓	✓	✓	1
Reward Shift	✓	✓	✓	0
Make denser	✓	✓		False
N-State space size, $\|NS\|$	✓			8
Action space size, $\|A\|$	✓			8
Reward every n steps	✓			True
Diameter	✓			1
Terminal state density	✓			0.25
Terminal state reward	✓	✓	✓	0
Target Radius		✓		0.05
Target Point		✓		Origin
N-State Space Range		✓		[-10, 10]
Action Range		✓	✓*	[-1, 1]
Time Unit		✓	✓*	1
Action Loss Weight		✓		0
Initial State Distribution	✓	✓		Uniform over non-terminal states
Transition Dynamics Order		✓		1
Inertia		✓		1
Terminal regions		✓		Circle of radius *target radius* around target point

in the original *real* time step would mean a reward scale that is 5 times greater and so the need for the correction. *Pusher* and *Reacher* have similar reasoning and, therefore, we would caution the user to be careful when setting the *time unit* dimension of hardness in the complex *Mujoco* wrapper[16].

5. Design Decisions for Toy Environments

We now discuss some of the design decisions underlying the toy environments to shed more light on their design.

5.1 Discrete Environment Generation

Once the values for the dimensions of hardness are set, for the case of auto-generated discrete toy environments, as mentioned above, P is generated by selecting for each N-state ns in independent set i, for each action a, a random successor state ns' 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. A motivating reason for unit testing in such a regularly structured environment 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.

5.2 Regular Structure for Unit Tests

In principle, it is always possible to design *adversarial* Ps (Nau, 1983; Ramanujan et al., 2010) 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. For many use cases, evaluating an agent on such an environment may give us a misleading measure of the type of agent intelligence we are hoping to measure. 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 (Zhang et al., 2021). *qbert* has a bug which allows the agent to achieve a very large number of points (Chrabaszcz et al., 2018). *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 a *lot of* 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 (Kirkebøen & Nordbye, 2017). For example, the agent strategy of creating a tunnel to target bricks in the top for *breakout* has been challenged multiple times (Atrey et al., 2020; Tosch et al., 2019). As Irpan (2018)

16. A similar concern does *not* exist for the toy continuous environments because in that case, the reward is the amount of distance moved from the current position to the target point. So even if the time step is made smaller, the reward scale becomes proportionally smaller because the agent can only travel proportionally smaller distances towards the goal on average in smaller timesteps.

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. For unit testing, especially, such complexity and variance in evaluation is likely undesirable. One needs quick insights on whitebox environments and thus, we believe it is beneficial to have a simple and regular structure.

Finally, a *more* regular structure is imposed as opposed to the usual gridworld because, 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.

5.3 Independently Controllable Dimensions

Algorithms like DQN (Mnih et al., 2015) 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 with 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. We do not try to capture, say credit assignment or generalisation as dimensions because these are not independently controllable. They are to be dealt with at a higher level, something that is like an *integration* test, rather than a *unit* test.

6. Experimenting with MDP Playground

We now discuss in detail how to experiment on *MDP Playground* to understand and debug agents.

6.1 Experimental Setup

We used the following experimental setup for this section. **Agents**: We evaluated *Rllib* implementations (Liang et al., 2018) of DQN (Mnih et al., 2015), Rainbow DQN (Hessel et al., 2018), and A3C (Mnih et al., 2016) on discrete environments and DDPG (Lillicrap et al., 2016), TD3 (Fujimoto et al., 2018) and SAC (Haarnoja et al., 2018) on continuous environments over grids of values for the dimensions of hardness. The information state used by the agent was always the observation given out by the environment. Hyperparameters (including neural network architectures used) and the tuning procedure used are specified in Appendices F and G. We used fully connected networks except for image representations where we used Convolutional Neural Networks (CNNs).

6.1.1 Environments

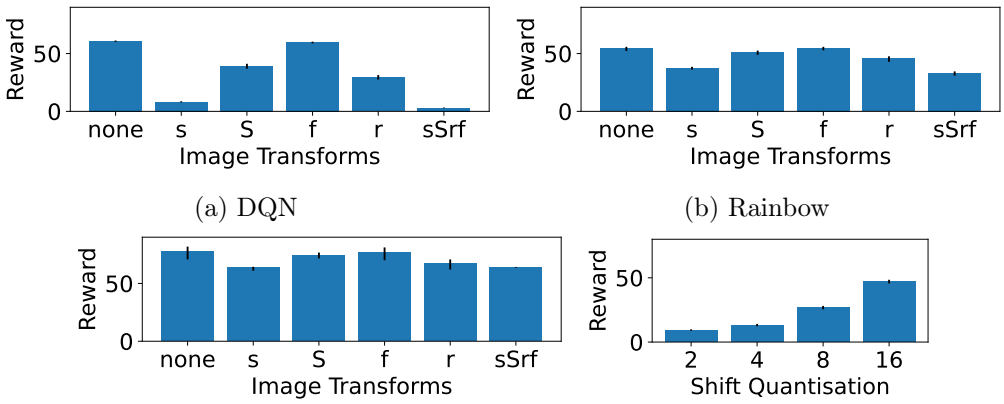
For the **discrete toy** experiments, we created a simple toy task with $\|S\|$ and $\|A\|$ set to 8. The *diameter* was set to 1. Thus, an agent could transition from an N-state to every other N-state. The reward density was such that when the sequence length was 1, only one sequence of a single N-state was rewarding, so that the task in the vanilla environment was as easy as choosing the right action to reach the rewarding N-state at every time step. For the **continuous toy** experiments, 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 episodes would terminate when an algorithm would reach the *target point*, or after at most 100 timesteps. The agent had to reach the same target point $[0, 0]$, from random starting positions. For these toy experiments, the dimension under consideration had a grid of values, each of which is shown in the figures for the corresponding experiment below. All other dimensions of hardness were set to their default values mentioned in Table 1. We evaluated 100 seeds for each of these experiments. We ran the DQN variants for $20k$ timesteps and A3C for $150k$ timesteps. The reason for selecting a different number of timesteps was dependent on how long it took for the different agents to converge and is discussed further in Appendix B.1. We ran all the continuous control task agents for $20k$ timesteps. For the **complex** experiments, we used *Atari* and *Mujoco*. For *Atari*, we ran the agents on *Beam Rider*, *Breakout*, *Qbert* and *Space Invaders*. For *Mujoco*, we ran the agents on *HalfCheetahV3*, *Pusher* and *Reacher* using *mujoco-py*. We evaluated 5 seeds for $10M$ steps ($40M$ frames) for Atari, $500k$ steps for *Pusher* and *Reacher* and $3M$ for *HalfCheetah*.

6.1.2 Analysis

For the experiments with toy environments, in the presence of *delay* and *sequence length*, we normalise the plots so that the optimal episodic reward is 100. For delay, we do so by multiplying the episodic reward by $100/(100 - \textit{delay})$. For sequence length, we do so by multiplying the episodic reward by the sequence length, n. In the bar plots, we plot the Area Under the Curve (AUC) of the training curves. As a result, the reward scales between the learning curves and bar plots would differ. For all bar plots, we apply Bonferroni corrections (Colas et al., 2018) for a significance level of 0.05 (Bonferroni corrections adjust the significance level of 0.05 as 0.05 / (*number of comparisons being performed*), e.g. $0.05/\binom{5}{2}$ if there are 5 bars in a plot).

6.2 Understanding Existing Agents

We hope our platform can help analyse and understand RL agents much better, especially due to the ease of performing experiments with fine-grained control over the injected dimensions of hardness. We first describe an experiment where experiments on toy environments alone led us to interesting insights. We then describe some experiments where we compare and contrast the performances on the toy and complex environments for further interesting insights.



(a) DQN (b) Rainbow

(c) A3C (d) DQN shift

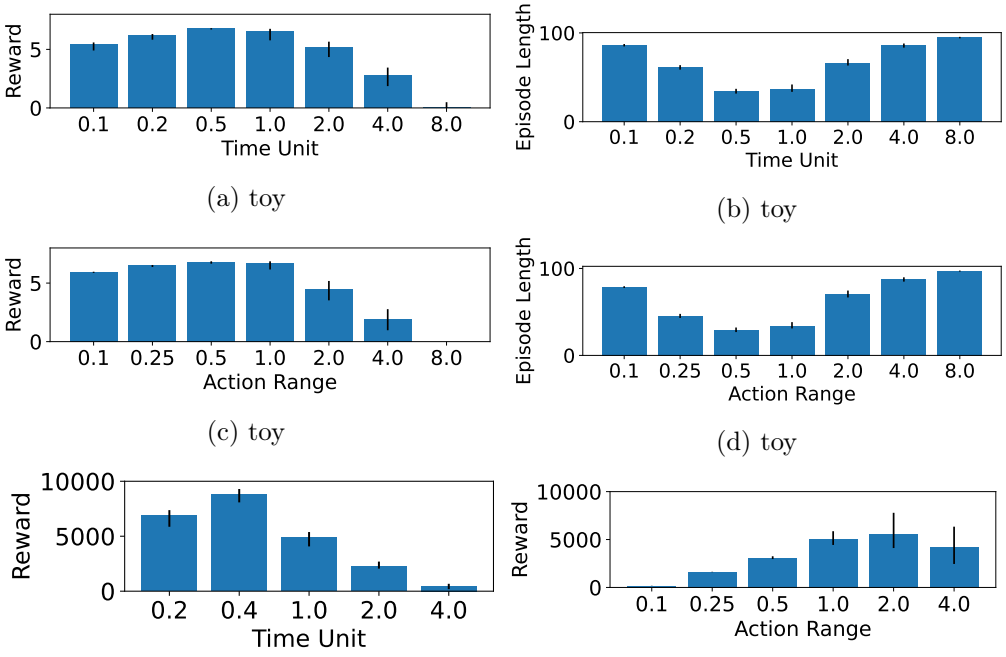
Figure 4: AUC of episodic reward at the end of training for the different agents when varying **image representation transforms** on the discrete toy environments. 's' denotes *shift*, 'S' denotes *scale*, 'f' denotes *flip*, 'r' denotes *rotate* and 'sSrf' denotes all 4 of these transforms in the X-axis ticks in the first three sub-figures and *Shift Quantisation* in the fourth sub-figure represents quantisation (in pixels) of the *shift*s in the DQN experiment for this. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

6.2.1 Image representations

As an example of what a researcher might want to experiment with, *MDP Playground* allows turning on image representations for the categorical states in the discrete toy environments. It further allows applying transforms to these images to have multiple possible representations for the same state. We applied various transforms (*shift*, *scale*, *rotate* and *flip*) one at a time and also all at once to study the effects of these. We observed that the more transforms that are applied to the images at once, the harder it is for agents to learn, as can be seen in Figures 4a-4c. This was to be expected since there are many more combinations to generalise over for the agent.

However, it was unexpected to us that the most difficult transform for the agents to deal with was *shift*. Despite the spatial invariance learned in CNNs (LeCun, 2012), our results imply that this variation seems to be the hardest one to adapt to. The results seemed to suggest that the more the total different values that are possible for a transform, the harder it is to adapt to. *Shift* had the most possible different values that could be applied to the images as compared to the other transforms. This fact seems to suggest that at least some amount of memorisation is taking place in the NNs.

As these trends were strongest in DQN, we evaluated further ranges for the individual transforms for DQN. Therefore, we quantised the *shift*s to have fewer possible values. Figure 4d 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 18a-18b in Appendix



(a) toy (b) toy

(c) toy (d) toy

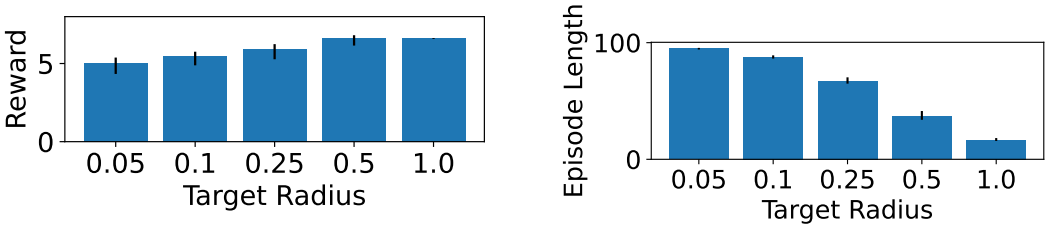
(e) HalfCheetah (f) HalfCheetah

Figure 5: AUC of selected metric at the end of training for DDPG **(a)**: episodic reward with **time unit** on the toy environment **(b)**: episode length with **time unit** on the toy environment. **(c)**: episodic reward with **action range** on the toy environment **(d)**: episode length with **action range** on the toy environment. **(e)**: episodic reward with **time unit** on a complex (HalfCheetah) environment (*time unit* values are relative to the defaults). **(f)**: episodic reward with **action range** on a complex (HalfCheetah) environment (*action range* values are relative to the defaults). Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

B). We emphasise that in a more complex setting, we would have easily attributed some of these results to chance but in the setting where we had individual control over the dimensions of hardness, our platform allowed us to dig deeper in a controlled manner.

6.2.2 Time unit and Action Range

As another example of a dimension of hardness a researcher might be interested in controlling, *MDP Playground* allows varying the *time unit* of how often the agent acts in a continuous environment. We varied this dimension of hardness in the toy continuous environment and observed that the *time unit* has an optimal value which has a significant impact on performance (Figure 5a for DDPG; and Figures 28c and 28b in the appendix for SAC and TD3). The *time unit* can be neither too small nor too large since there is an optimal *time unit* for which we should repeat the same action (Biedenkapp et al., 2021). Based on this finding, we also analysed the *time unit* for complex environments (Figure 5e, Figures 34d-34f, 35d-35f



(a) Episodic reward

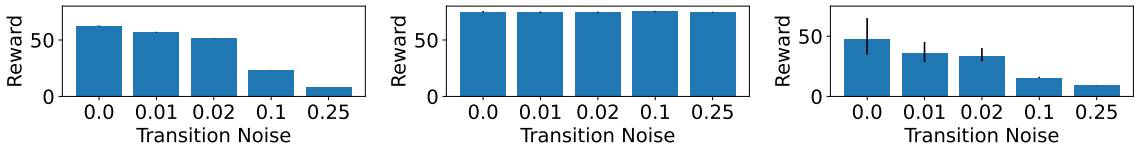
(b) Episode length

Figure 6: AUC of selected metric at the end of training for DDPG when varying **target radius** on the toy environment **(a)**: episodic reward **(b)**: episodic length. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

and 36d-36f in the appendix), finding similar results on all environments. In fact, for all three agents, (Figures 5e, Figures 34f and 34e in the appendix), we see peak performance on *HalfCheetah* for a *time unit* 0.4 smaller than that of the *expert*-designed vanilla environment. There were gains of even 100% in performance in some cases. We emphasise again that we do not advocate modifying the problem definition of such benchmark problems to claim new state of the art results, but rather for analysing problems. 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 may be even more important to study the effect of such dimensions of hardness for more complex environments, especially when defining the problem for a completely new task.

We observed similar trends for *action range* 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. For the toy environment, please see Figure 5c for DDPG and Figures 27c and 27b in the appendix for SAC and TD3. Similarly as for *time unit*, more pronounced trends for *action range* can be seen for HalfCheetah in Figure 5f for DDPG and in Figures 34b-34c for TD3 and SAC in the appendix. Figures 35a-35c and 36a-36c in the appendix show trends largely similar to the toy environment for all three agents on *Pusher* and *Reacher*. This supports the insight gained on our toy environment that setting this value differently may lead to significant gains for an agent. Similar to the *time unit*, in the expert-designed environment setting of *Reacher* in *Gym*, we found that all three algorithms performed best for an *action range* 0.25 times the value found in *Gym* (Figures 36a, 36b, 36c in the appendix). 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* while developing agents and defining problems is under-researched because the standard benchmark environments have been pre-configured by experts. From the above discussion, it is clear that playing around with pre-configured values can lead to new insights even in *known* environments and



(a) toy train (b) toy noise-free (c) Breakout

Figure 7: AUC of episodic reward for DQN at the end of training when varying **transition noise (a)**: on the toy environment while training. **(b)**: on the toy environment when rolling out the policy that was learned on the noisy environment on a noise-free setting **(c)** on *Breakout*. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

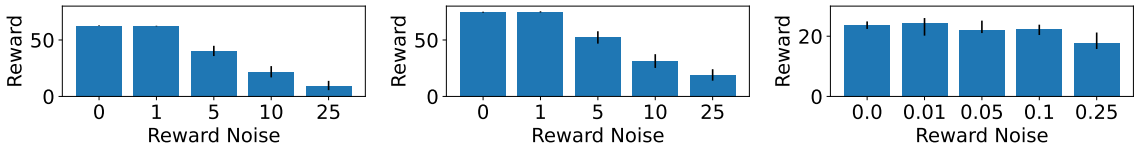
that we need to pay attention to these dimensions of hardness in a completely *unknown* environment when defining the MDP formulation of a problem, if we want agents to perform as desired. And even when the problem definition is set, one might still want their agent to search within a subset of the action range or want their agent to repeat actions over multiple timesteps (Biedenkapp et al., 2021).

6.2.3 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. We noticed in the continuous toy environment that, for small values of the target radius, all the continuous control agents oscillated around the target while trying to reach it exactly. This can be observed in Figure 6a and 6b for DDPG (and in Figures 26b and 26e for TD3 and Figures 26c and 26f for SAC in the appendix), where we note that even though the task *was* learnt for different *target radii*, the episode lengths were much longer for shorter target radii. This was because the agents kept oscillating outside the target radius. Even for such a simple task all evaluated algorithms failed to adapt to performing fine-grained control near the target. We believe that the agents did not learn to perform fine-grained control close to the target because the agents need to see data points closer to the goal to train their NNs, which highlights how data intensive model-free algorithms can be.

6.2.4 Transition and Reward Noise

Experimenting with *transition noise* and *reward noise* allowed us to observe that performance dropped on the toy environments for all agents as one might expect (Figure 7a and 8a; Figures 21-22 in the appendix). 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.

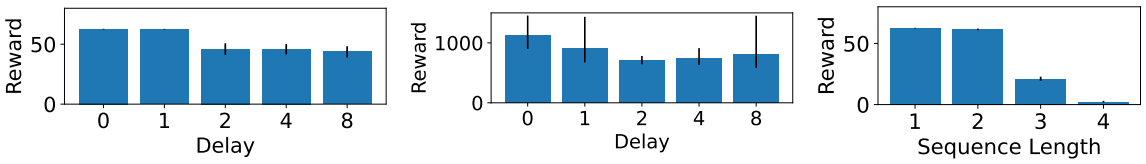


(a) toy train (b) toy noise-free (c) Breakout

Figure 8: AUC of episodic reward for DQN at the end of training when varying **reward noise (a)**: on the toy environment while training. **(b)**: on the toy environment when rolling out the policy that was learned on the noisy environment on a noise-free setting **(c)**: on *Breakout*. Please note that this run on *Breakout* had reward clamping turned off, which is on by default in Ray and thus had a very different episodic reward range compared to runs with other dimensions of hardness on *Breakout*. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

Interestingly, when we look at the performances on the noisy environment versus when rolling out the policy that was learned on the noisy environment on a noise-free setting for *transition* noise in Figures 7a and 7b (Figures 21 and 23 for all agents in the appendix), we see that the training performance of the algorithms is more sensitive to noise during training than the eventual performance of the same policy on the noise-free setting. It is a non-trivial insight that injecting noise into the *transition* function still leads to good learning for the noise-free version of the transition noise environment (as displayed in the noise-free rollout plots). A second interesting insight is when comparing the above pairs of figures from the transition noise environments to the corresponding ones for the *reward* noise versions of the environments Figures 8a and 8b (Figures 22 and 24 for all agents in the appendix). We observe that injecting noise into the *reward* function does *not* lead to good learning for the noise-free version of the reward noise environment. This is in contrast to the insight from the transition noise versions of the environments. We conjecture that this could be due to the noise from different noisy transitions in the replay buffer/training data cancelling each other out in the gradient-based updates while learning for the transition noise but not for the reward noise. It could also be that transition noise helps the agent *explore* more while reward noise does not.

We observed also for the Atari environments for all three agents that performance dropped on average when injecting *transition noise* (Figure 7c for DQN on *Breakout*; Figures 30d-30f, 31d-31f, 32d-32f and 33d-33f for all agents and environments in the appendix). However, we also observe that performance drop is different for different environments. This is to be expected as there are other dimensions of hardness that we cannot control or measure for the complex environments. We further observed that for some of the environments, transition noise actually even helped performance (e.g. Figure 33e for Rainbow on *Space Invaders*). This has also been observed in prior work (Wang et al., 2019). This can happen when the exploration policy was not tuned optimally since inserting transition noise is almost equivalent to ϵ-greedy exploration for low values of noise. Finally, we observed similar trends



(a) toy (b) Qbert (c) Sequence Length

Figure 9: AUC of episodic reward for DQN at the end of training when varying **(a)**: **delay** on the toy environment. **(b)**: **delay** on *Qbert*. **(c)**: **sequence length** on the toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

for *reward noise* (Figure 8c for DQN on *Breakout*; Figures 30g-30i, 31g-31i, 32g-32i and 33g-33i for all agents and environments in the appendix).

Another interesting insight is that if one compares the X-axis ticks for the toy and complex environments for *reward noise* (e.g. Figures 8a and 8c), the scale of the *reward noise* is very different in the toy and complex environments. This, we believe, is because the complex environments are much more sparse - if one rolls out a random agent, or even a trained one, on the complex environments, the non-zero rewards are far less frequent than in the toy environments we experimented with. While it is naturally possible to design toy environments to be much more representative in terms of trying to match the hardness dimensions of complex environments, it tends to happen that this leads to them becoming much harder to learn on as well and, as a result, losing their ability to be quick testbeds. Training the agents on them with the same scale of *reward noise* as the toy environments led to no learning at all, so we had to select far smaller values to show the trends. Another important detail to note about the reward noise experiments on Atari environments is that Atari environments in Ray have what they term *reward clipping* turned on by default. This sets the reward at every time step to 0, -1 or +1 depending on the sign of the reward obtained[17]. This interferes with the reward noise experiments because even very small noisy rewards are clamped according to their sign, e.g., +0.01 would be set to +1 and then the meaning of reward noise is different to what we intended to inject. As such, we had to change reward clipping to instead actually have "clip" behaviour, i.e. to clip rewards greater than +1 to +1 and clip rewards less than −1 to −1, to perform meaningful experiments with reward noise. This led to very different episodic reward ranges compared to runs with other dimensions of hardness, e.g. on *Breakout* (compare reward scales in Figures 7c and 8c). This difference seemed to occur only for DQN and Rainbow but not for A3C (please compare the reward scales in the bottom row of Figure 31 in the appendix with the reward scales in its upper rows) which is an interesting avenue for future research regarding behaviour of these agents in the presence or absence of reward clipping. Ideally, needless to say, we want our agents to be robust to different scales of reward and to not have to resort to something like reward clipping.

17. It is called reward *clipping* in there, but *clamping* might be a better term

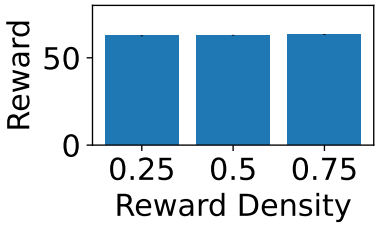


Figure 10: AUC of episodic reward for DQN at the end of training when varying **(a)**: **reward density** on the toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

6.2.5 Reward Delay and Sequence Length

Experimenting with *delay* also led to similar high-level trends as for *transition noise*: performance dropped on the toy environments for all agents (Figure 9a for DQN; Figure 19 for all agents in the appendix). We additionally also plot the *learning curves*, when varying delay, in Figure 13. We see how training proceeds much more smoothly and is less variant across different seeds for the vanilla environment (where the delay is 0) and that the variance across seeds is very large for delay = 2. Such noisy learning curves are typically associated with RL and clearly partial observability, as is the case here, is a contributing factor even for a toy environment. Sometimes, the agents still managed to perform well, which emphasises that agents *can* perform well even when having a non-Markov information state and shows how *tuning* seeds can lead to good results (see e.g., Figure 5 in Henderson et al., 2018).

We also note that, on average, performance dropped for the Atari environments (Figure 9b for DQN on *Qbert*; Figures 30a-30c, 31a-31c, 32a-32c and 33a-33c for all agents and environments in the appendix). However, the trends for *delay* were noisier than for *transition noise*. We believe this is because the situation for delays is more nuanced in the case of complex environments: many considered environments tend to have repetitive sequences which would make the effect of injecting delays noisier. Interestingly, many of the learning curves with delays inserted, are indistinguishable from normal learning curves (e.g., DQN on Atari in Figure 11). We believe that, in addition to the motivating examples from Section 2.2, 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 look at learning curves for transition noise (e.g., DQN on Atari in Figure 12), 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. Pointers to more such learning curves can be found in Appendix J.

Even more nuanced in the case of *delay* is the comparison of effects on the different environments: the greatest drops in performance, on average, were on *Qbert*, followed by *Beam Rider*, *Space Invaders* and *Breakout* (see again Figures 30a-30c, 31a-31c, 32a-32c and 33a-33c in the appendix); for *Breakout*, in many instances, we do not even see any performance drops. We believe this is because large delays from rewarding trajectory to reward are already present in *Breakout*, which means that inserting more delays does not

have as large an effect as in *Qbert*. Agents are most strongly affected in *Qbert* which, upon looking at gameplay, we believe has the least delays from rewarding trajectory to reward. It is interesting that inserting *delays* for Rainbow on *Qbert* seems to really impact its performance, more so than any of the other Atari experiments.

To further analyse and compare the effects of the dimensions of hardness on the toy and complex environments 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 (Tables 2-4 in the appendix list the individual rank correlation for each experiment (for different agents, environments and dimensions of hardness). The Spearman correlation was $>= 0.7$ for 27 out of 36 experiments and a positive correlation for eight of the remaining nine. DQN with delays added on breakout was the only experiment with correlation 0, which further supports our hypothesis above that delays are already present in the greatest amount in breakout. Further, the average rank correlation over 12 experiments (3 agents x 4 Atari environments) was 0.867 for *transition noise*, 0.733 for *reward noise* and 0.617 for *reward delay* which supports the observation above that trends were noisier for delay.

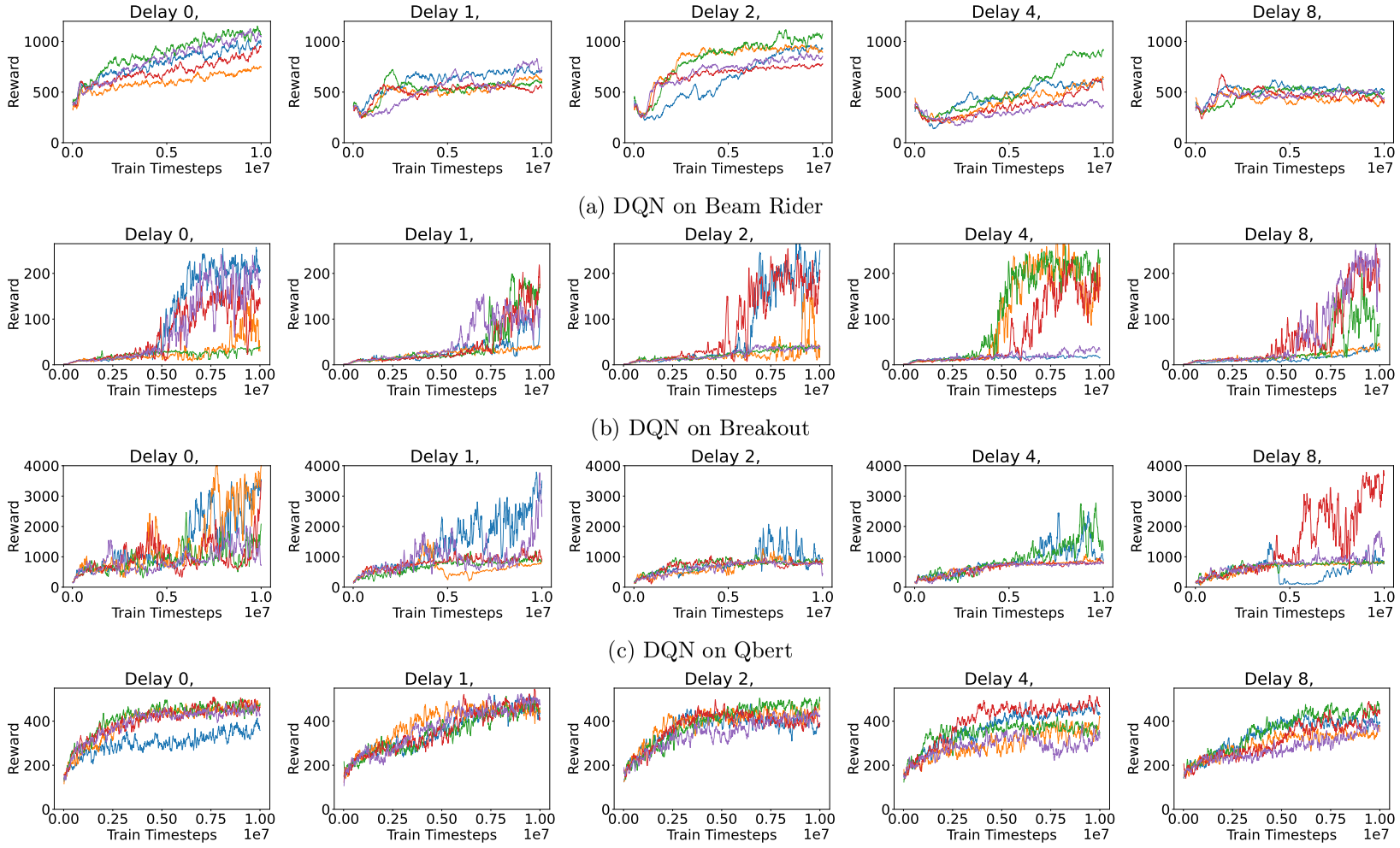
Results for experiments with different *sequence length*s on the toy environment are qualitatively similar to the ones for delay. However, we observe (see Figure 9c for DQN, Figure 20 for all agents in the appendix) that sequence length has a more drastic effect in terms of degradation of performance.

6.2.6 Irrelevant Features

Introducing *irrelevant dimensions* in to the toy continuous environments, while keeping the number of relevant dimensions fixed to 2, decreased agent performance (Figure 14a for SAC; Figure 29a-29c for all agents in the appendix). This gave us the insight that irrelevant features interfere with the learning process. It was not just the episodic reward that was worse but also the mean episodic length (Figure 14b; Figures 29d-29f for all agents in the appendix), which meant that even though the agents were reaching the target point within the episode, they were taking longer to do so. We performed similar experiments with SAC on a complex environment (*HalfCheetah*) and observed similar trends for performance (Figure 14c).

6.2.7 Reward Density

Figure 10 shows the results for DQN (Figure 25 for all agents in the appendix) when controlling the *reward density* on the toy environment. It is interesting that the density does not seem to impact the performance almost at all for any of the agents. We believe that is because once any of these agents finds a rewarding sequence (which is a single N-state in the case of a sequence length of 1), they do not really need to change their policy because the other rewarding sequences also result in the same reward. To allow for a more varied reward distribution to explore more interesting directions in such cases, we have an experimental dimension of hardness, the *reward distribution* (for more information on how this dimension works, please see Appendix A). While the above experiment might make it seem that the dimension of hardness reward density is not useful when the reward handed out is always 1, Section 6.3 shows an interesting use-case for it in the presence of ϵ-greedy exploration.



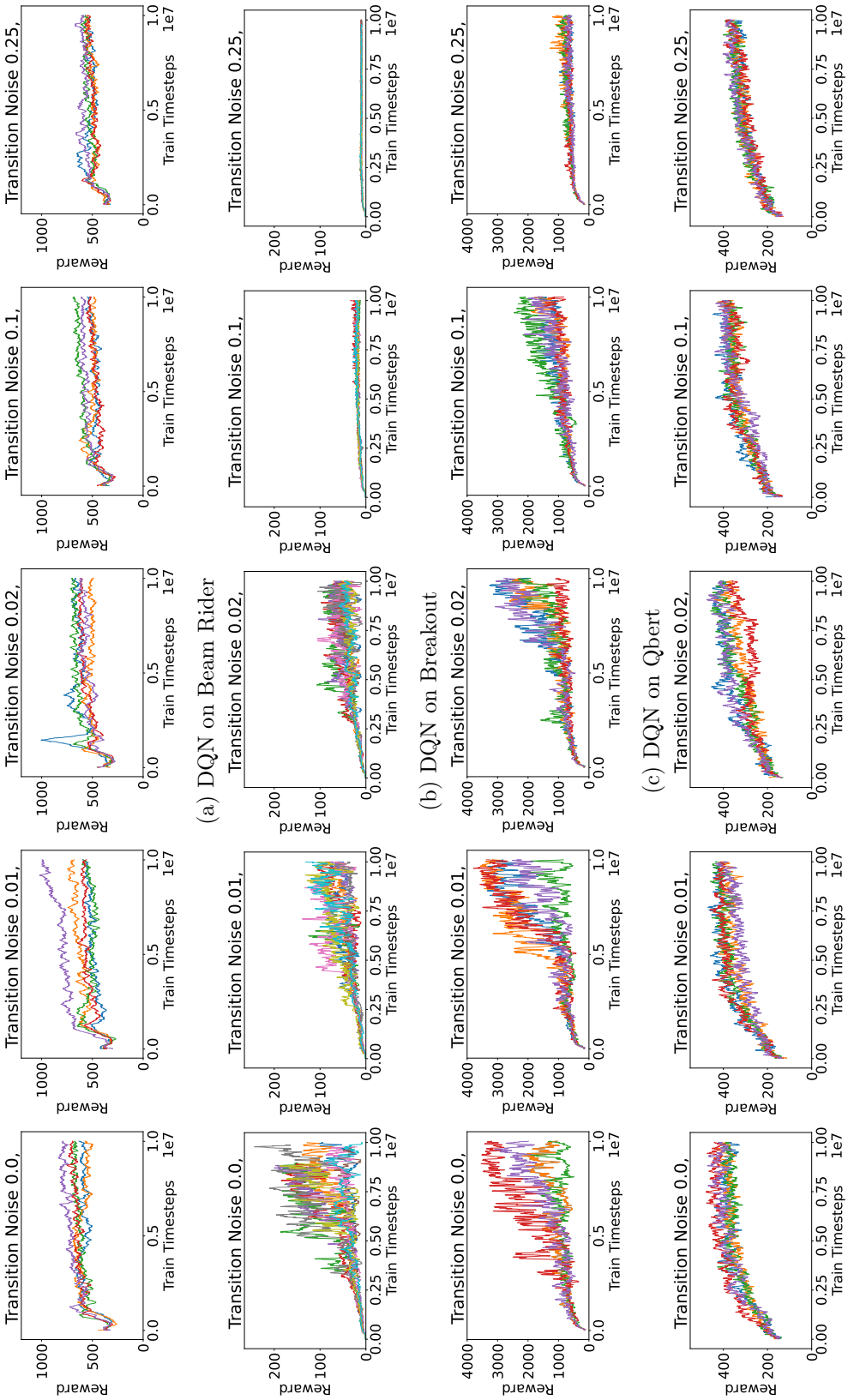
(a) DQN on Beam Rider

(b) DQN on Breakout

(c) DQN on Qbert

(d) DQN on Space Invaders

Figure 11: Training Learning Curves for DQN on Atari environments with *delay*. Please note that each different colour corresponds to one of 5 seeds in each subplot.



(a) DQN on Beam Rider

(b) DQN on Breakout

(c) DQN on Qbert

(d) DQN on Space Invaders

Figure 12: Training Learning Curves for DQN on Atari environments with *transition noise*. Please note that each different colour corresponds to one of 5 seeds in each subplot.

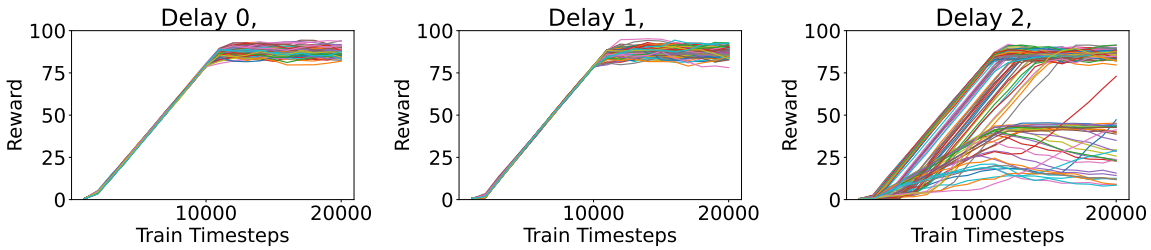
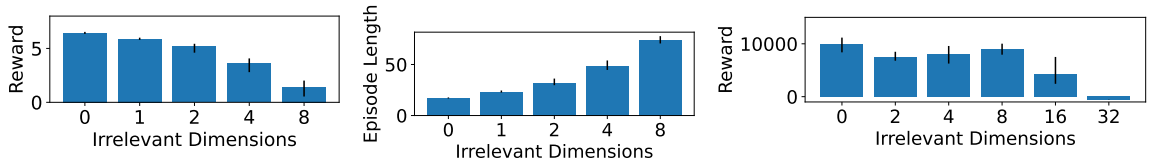


Figure 13: Train Learning Curves for 100 runs with different seeds for DQN when varying **delay**. Please note that each different colour corresponds to one of 100 seeds in each subplot. We do not show the curves for delay = 4 and delay = 8 which follow the same trends to improve readability.



(a) SAC toy (b) SAC toy (c) SAC HalfCheetah

Figure 14: AUC of selected metric for SAC at the end of training when varying **irrelevant dimensions** **(a)**: episodic reward on the toy environment. **(b)**: episode length on the toy environment. **(c)**: episodic reward on *HalfCheetah*. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

6.2.8 Further Discussion

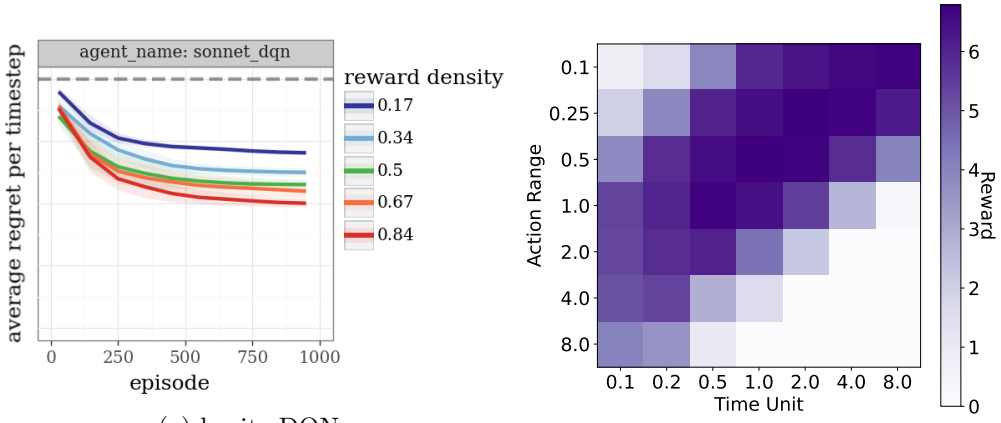
For some readers, it might feel self-evident that injecting many of these dimensions causes difficulties for agents. However, to the best of our knowledge, no other work has tried to collect so many dimensions of hardness in one place and study them comprehensively and what aspects transfer from toy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

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 is nice to have general algorithms that may be applied in a blackbox fashion, by studying the dimensions we have listed and their effects on environments, we may gain deeper insights into what is needed to design better agents.

An additional comment can be made about comparing the continuous complex to toy comparisons to the discrete complex to toy comparisons. The "noise" in comparing the toy to the complex discrete environments was higher compared to comparing the continuous toy to 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 leading to less noisy comparisons.

The experiments in this section also show how the complex environment wrappers allow researchers, who are curious, to study the robustness of their agents to these dimensions of hardness on complex environments, without having to fiddle with lower-level code.



(a) bsuite DQN

(b) 2 dimensions together

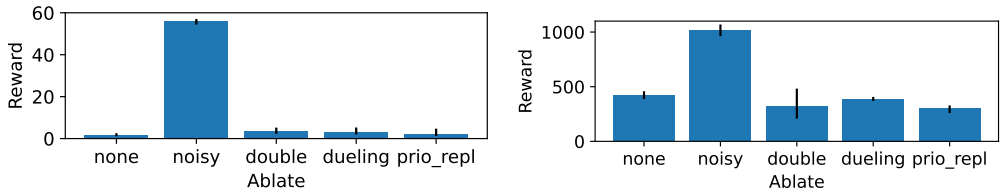
Figure 15: Analysing and Debugging. **(a)**: Varying *reward density* on the toy environment for the *bsuite* DQN agent. **(b)**: Varying 2 dimensions of hardness together - **action range** and **time unit**.

6.3 Debugging Agents

Analysing how an agent performs under the effect of various dimensions of hardness can reveal unexpected aspects of the agent. For instance, when using bsuite agents (Osband et al., 2019), we noticed that when we varied our environment's *reward density*, the performance of the bsuite DQN agent got worse as the reward got sparser (see Figure 15a). This did not occur for other bsuite agents. This behaviour with *MDP Playground* occurs because the DQN agent used ϵ-greedy while the other agents explored using other methods. ϵ-greedy causes the agent to perform a completely random a certain fraction of the time which then leads to entering a terminal state a proportional fraction of the time and consequently to rewards in proportion to the reward density. Such insights can easily go unnoticed if the environments used are too complex. The simplicity of our toy environments helps debug such cases.

In another example, in one of the Ray versions we used (`0.9.0.dev0`), we observed that DQN was performing well on the *image representations* environment while Rainbow was performing poorly. We were quickly able to *ablate* additional Rainbow hyperparameters on the toy environments and found that the noisy nets (Fortunato et al., 2018) implementation was broken (see Figure 16a). We then tested and observed the same on a more complex

environment, *Beam Rider* (see Figure 16b). This shows how easily and quickly agents can be debugged to see if something major is broken.[18] This, in combination with their low computational cost, also makes a case to use the toy environments in Continuous Integration (CI) tests on code repositories.



(a) toy (b) Beam Rider

Figure 16: Ablations of Rainbow on *image representation* environments. X-axis labels mention which component was ablated, i.e., turned off: *noisy* represents noisy nets, *double* represents Double DQN, *dueling* represents Dueling Networks and *prio_ repl* represents prioritised replay. Runs on *Beam Rider* were only till 3M timesteps in this experiment. Turning noisy nets **off** improved performances on both toy and complex environments significantly. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

6.4 Further Use Cases

We now discuss further interesting use cases for *MDP Playground*.

6.4.1 Multiple Dimensions

It is possible to vary multiple dimensions of hardness at the same time in the same base environment in *MDP Playground*. For instance, Figure 15b shows the interaction effect (an inversely proportional relationship) between the *action range* and the *time unit* in the continuous toy environment with DDPG. This is an interesting insight and shows that the agent's *action range* should depend on the *time unit* or vice versa. On thinking more about this, it makes sense because: 1) if actions that are too large are allowed for large time steps, the agent might go out of control, while if large actions are allowed only for shorter time steps, the agent can still remain stable because it has fine-grained control; 2) allowing a large space of actions for large time steps increases the search space for optimal actions and make it a harder control problem. Plots for more such experiments can be found in Appendix E, including varying both P and R *noise*s together and varying *delay* and *sequence length* together in the toy discrete environments.

18. An 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`.

6.4.2 Design and Analyse Experiments

We allow the user the ability to inject dimensions of hardness into toy or complex environments in a fine-grained manner. This can be used to define custom experiments with the dimensions of hardness. The results can be analysed in the accompanying *Jupyter notebook*[19].

6.4.3 Ground Truth

Since *MDP Playground* has the ground truth of the underlying MDPs available for the toy environments, it is a useful tool for designing agents that run on POMDPs and try to extract the underlying features of an MDP that make it fully observable. The ground truth could also be used, for instance, in model-based RL to test how well true aleatoric uncertainties are learned by a probabilistic model on a toy environment with stochasticity injected.

6.4.4 Tabular Agents

The fact that the toy environments are so simple also allows the platform to be used to compare tabular and deep agents. To this end, we also evaluated the tabular baselines Q-learning (Sutton & Barto, 1998), Double Q-learning (van Hasselt, 2010) and SARSA (Sutton & Barto, 1998) on the discrete non-image based environments with similar qualitative results as to those for deep agents. Results can be seen in Appendix D.

The experiments here are only a glimpse into the power and flexibility of *MDP Playground*. Users can also upload custom functions for P and R and custom image representations O (instead of the polygons from Figure 2) and our platform takes care of injecting the other dimensions of hardness for them (wherever possible).

7. Related Work

Many of the other environments 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 environments 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, Osband et al., 2019). In contrast to our *generated* environments, bsuite *collects* simple RL environments from the literature that are representative of various types of problems which occur in RL and tries to characterise RL algorithms. *Bsuite* can be seen as an intermediate step between our toy MDPs and more complex environments, because the toy environments that *bsuite* collects are more complex than ours and do not always have controllable dimensions of hardness that can be independently injected. This makes *bsuite*'s dimensions not individually controllable and *atomic* like ours. 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. *Bsuite*'s experiments are

19. `https://github.com/automl/mdp-playground/blob/master/plot_experiments.ipynb`

also more expensive than ours; while *bsuite* is already quite cheap to run, *MDP Playground* experiments are over an order of magnitude cheaper. Furthermore, while *bsuite* offers **no continuous control environments**, *MDP Playground* provides discrete and continuous environments. This is important because several agents like DDPG, TD3, SAC are designed for continuous control. 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. Finally, in contrast to *bsuite*, *MDP Playground* offers wrappers to inject many of the identified dimensions of hardness into complex environments. We use these wrappers to also compare the identified trends on toy and complex environments. 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 with specific challenges.

Toybox (Tosch et al., 2019) and Minatar (Young & Tian, 2019) are also cheap platforms with similar goals of gaining deeper insights into RL agents. However, their games target the specific *Atari* domain and, like *bsuite*, they are complementary to our approach as they have more specific environments and do not offer fine-grained control over dimensions of hardness.

We found the work of Andersson and Doherty (2018) the most similar work to ours in spirit. In contrast to our platform, they only target continuous environments. We capture their dimensions of hardness in a different manner and offer many more dimensions with fine-grained control. Furthermore, their code is not open-source, making their work harder to build on by the community.

Maillard et al. (2014) defines a novel theoretical metric for defining hardness of MDPs and captures this hardness 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 would 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. Littman et al. (2017) 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.

Further related work includes *Procgen* (Cobbe et al., 2020) and *Obstacle Tower* (Juliani et al., 2019). Procgen adds various heterogeneous environments and tries to quantify generalisation in RL. In a similar vein, Obstacle Tower provides a generalisation challenge for problems in vision, control, and planning. Unlike ours, the scope of these platforms is not being a testbed. As such, they do not capture dimensions of hardness that can be independently injected with fine-grained control over them, which we view as an important aspect when testing agents. Dulac-Arnold et al. (2020) provides some overlapping dimensions of hardness with our platform but only for continuous environments, and has no toy environments.

7.1 MDP Playground in Relation to MNIST

MNIST (LeCun & Cortes, 2010) captured some key aspects of hardness required for computer vision (CV) which made it a good testbed for designing and debugging CV algorithms - the webpage for the dataset (`http://yann.lecun.com/exdb/mnist/` mentions some distor-

tions for MNIST (similar to our image representation transforms): *distortions are random combinations of shifts, scaling, skewing, and compression.* Mu and Gilmer (2019) captures 15 such distortions to benchmark out-of-distribution robustness in CV. 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, similar to our toy environments for RL.

8. Limitations of the Approach and its Ethical and Societal Implications

While we do not see any limitations regarding the complex environment wrappers in *MDP Playground* beyond the limitations of complex environments themselves, we would like to caution against some limitations for the toy environments. Toy environments in *MDP Playground* are meant to be analysis and debug testbeds and not for tuning the final agent hyperparameters (HPs) for use on complex environments. They are extremely cheap compared to complex environments and (as one would expect), can only be used to draw high-level insights and are likely not as differentiating as complex environments for many of the finer changes between RL agents. As also mentioned in Dehghani et al. (2021), a lot of tricks may be involved in achieving SOTA scores on complex benchmarks and the toy environments cannot be expected to be predictive of performance on such benchmarks. Finally, Multi-Agent RL, Multi Objective RL, and Time Varying MDPs (amongst others) are beyond the scope of the currently implemented toy environments. We also note here that we currently consider partial observability only in the reward dynamics and not in the transition dynamics. Introducing partial observability also in the transition dynamics is an interesting avenue for future work.

Another limitation is that in the complex environment wrappers, it is not always possible to control the exact "amount" of hardness in complex environments. Using the wrappers, however, always increases this existing amount. For dimensions of hardness like *transition noise* and *reward noise*, one can use the wrappers to control the exact amount of hardness. However, for dimensions of hardness like *delay* and *sequence length* which are usually already present in a complex environment and variable across the state space, it is not possible to control their exact amounts of hardness with the wrappers as we do not know how much of these exist already in the environment. It is a very hard problem to determine how much of each of them exists already and being able to determine this could go a long way to solving the RL problem itself, so there is no trivial way to get around this limitation of the wrappers in *MDP Playground*.

Further, high-dimensional control problems where there are interaction effects between degrees of freedom are not captured in the toy rigid body continuous control problem as this is the domain of complex environments and beyond the scope of this platform.

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. An additional environmental impact would be that, at least, prototyping and testing of agents could be made cheaper, reducing carbon emissions but also hopefully accelerating RL research.

9. Conclusion and Future Work

We introduced a platform to analyse and debug RL agents with fine-grained control over various dimensions of hardness. The platform allows low-cost experiments on toy environments and analysing the effects of dimensions of hardness on complex environments. We studied the performance of various agents and gained insights on toy and complex environments by varying these dimensions. To the best of our knowledge, we are also the first to perform a principled study of how significant aspects such as non-Markov information states, irrelevant features, representations and low-level dimensions of hardness, like time discretisation, affect agent performance. Finally, we also described debugging capabilities of the platform.

We believe there is great potential to design agents that adapt the time unit, action range, etc. dynamically and test them on vanilla environments. To the best of our knowledge agents currently do not try to adapt most of these dynamically and yet as we know from observing humans, humans are adaptive to these dimensions of hardness. Previous theoretical breakthroughs initially came on very small toy examples while *deep* breakthroughs came, in some cases, decades later. If we are able to make agents more robust to some of these dimensions of hardness on toy examples, we may later be able to make it work on complex environments. Such agents could also improve explainability of RL agents because, as we know, humans can also provide rough estimates of the dimensions of hardness to explain their actions. For example, if a human buys a lottery ticket and wins the lottery a week later, they can say that it was the action of buying with a week's delay that got them their reward. This also has links to causality.

An interesting distinction can also be made between agents' robustness to perturbations to a single task and agents learning to generalise across a distribution of tasks. Having ground truth and fine-grained control over the dimensions of hardness allows *MDP Playground* to also be used as a testbed for MetaRL which learns over distributions of tasks/environments. For example, it could be studied how robust MAML (Finn et al., 2017) is to wider distributions of the same task, by simply varying the range of a dimension of hardness of *MDP Playground*. It could also be used to study how the number of gradient descent steps performed from the common initialisation found by MAML interacts with the distribution it performs well over. While the exact values from such analyses may not carry over to complex domains, it is interesting to analyse whether such effects exist on a simpler domain and to then possibly formalise them theoretically and come up with numerical quantifications of the effects and their transfer across environments.

Finally, we would like *MDP Playground* to be a community-driven effort. It is open-source for the benefit of the RL community at `https://github.com/automl/mdp-playground`. While we tried to identify as many dimensions of hardness as possible, it is unlikely that we have captured *all* such dimensions that can be independently injected in RL environments. We have some further dimensions of hardness currently in the experimental phase and we describe these in Appendix A. We welcome more dimensions of hardness that readers think will help us encapsulate further challenges in RL and will add them to *MDP Playground* based on the community's opinions. Given the current brittleness of RL agents (Henderson et al., 2018), and many claims that have been challenged (Atrey et al., 2020; Tosch et al., 2019), we believe RL agents need to be tested on a lower and more basic level to gain insights into their inner workings.

Acknowledgments

The authors gratefully acknowledge support by BMBF grant DeToL, by the Bosch Center for Artificial Intelligence, 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, by the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. The authors acknowledge funding through the research network "Responsive and Scalable Learning for Robots Assisting Humans" (ReScaLe) of the University of Freiburg. The ReScaLe project is funded by the Carl Zeiss Foundation. They would like to thank their group, especially Joerg, Steven, Samuel, for helpful feedback and discussions. The authors are also grateful to the reviewers for their many helpful insights and feedback that helped improve the paper. Raghu would like to additionally thank Michael Littman for his feedback and encou ragement and the RLSS 2019, Lille organisers and participants who he had interesting discussions with.

Appendix A. Experimental Dimensions in MDP Playground

We list here some dimensions that are still experimental in *MDP Playground*.

* Only for discrete toy environments:

  - **Reward Distribution**: When the reward distribution option is turned on, it can be specified as a list with 2 floats. These 2 values are interpreted as a closed interval and taken as the end support points of a categorical distribution with support points equally spaced along the interval. Each support point is assigned randomly to one of the automatically generated rewardable sequences, so that when a rewardable sequence is achieved, the reward handed out is the value of the support point. Further different kinds of reward distributions that could lead to interesting insights are an interesting avenue to develop this dimension of hardness further.

* Only for continuous toy environments:

  - **Reward Function**: The reward function is currently fixed to `move_to_a_point` which represents the task of moving the point mass to a *target point*. Additionally, we have a toy task of moving along a line which can be specified by setting the reward function to *move_along_a_line*. For this task, we hand out greater rewards the closer a point object is to moving linearly for n consecutive steps where n is the sequence length. The linear movement can be along *any* direction, the important thing is that n consecutive movements of the point mass should be in a straight line to achieve the reward. This allows evaluating agents on a sequential/navigation task of moving along a line. However, it is currently in the experimental phase because it turned out to be too hard for the agents and we probably need to find a better formulation for the task so that it allows us more valuable insights into agents.

Appendix B. Additional Details on Toy Experiments

We provide some additional detail on the experiments in the main paper here, including the remaining plots referenced in the discussion in the main paper.

B.1 Selecting Total Timesteps for Discrete Toy Runs

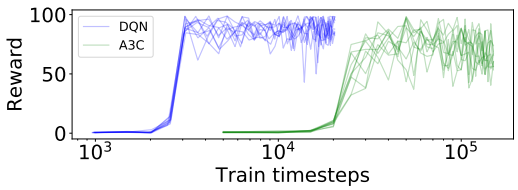


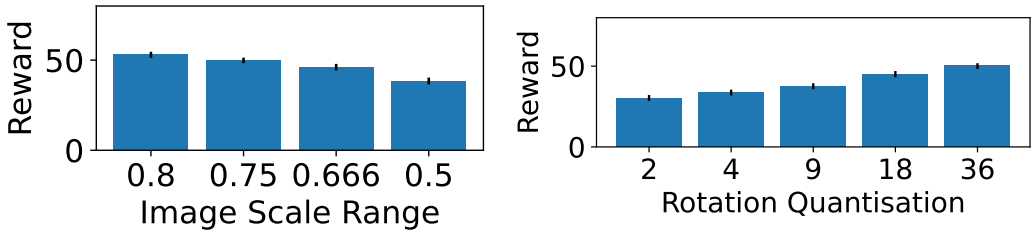
Figure 17: Evaluation rollouts for DQN and A3C in the vanilla environment which shows that DQN learns faster than A3C in terms of the number of timesteps.

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[20] to learn as can be seen in Figure 17. 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 tend to be slower in terms of time steps compared to value-based approaches such as DQN. Throughout, we always run 100 seeds of all algorithms to obtain reliable results. We repeated many of our experiments with an independent set of 100 seeds and obtained the same qualitative results.

20. In terms of environment steps. Wallclock time used was still similar.

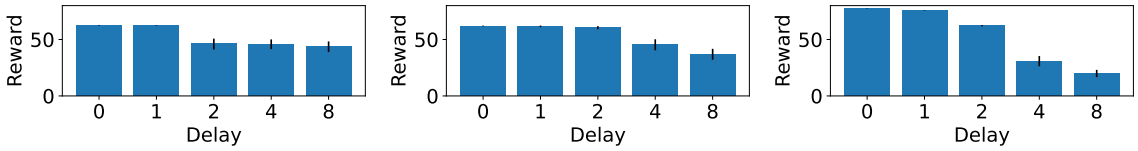
B.2 Additional Plots

We provide here additional plots for the various experiments we ran on *MDP Playground*.



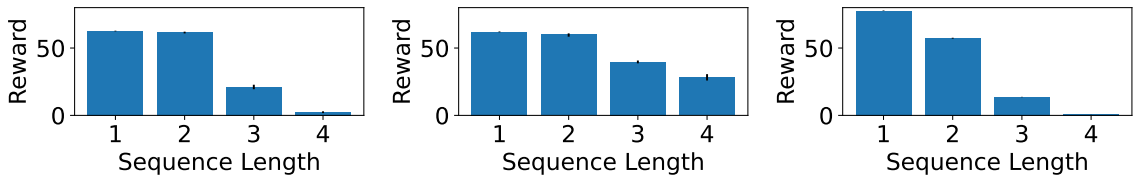
(a) Scale Range (b) Rotate

Figure 18: AUC of episodic reward at the end of training for DQN when varying **the quantisation/range of transforms for image representations**. **(a)**: *Image Scale Range* represents *scaling* ranges. The X-axis ticks are the lower end of the image scale range, the upper end is just the inverse of the tick value shown. **(b)**: *Rotation quantisation* represents quantisation of the *rotation*s. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



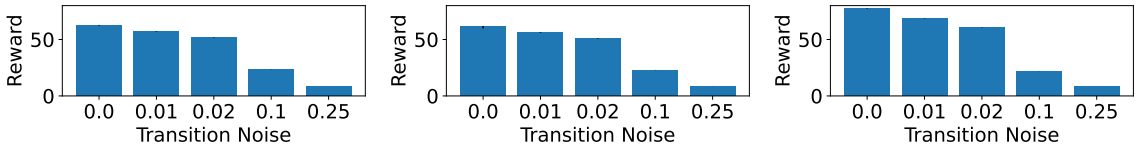
(a) DQN (b) Rainbow (c) A3C

Figure 19: AUC of episodic reward at the end of training for DQN, Rainbow, A3C when varying **delays** on the discrete toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



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

Figure 20: AUC of episodic reward at the end of training for DQN, Rainbow, A3C when varying **sequence lengths** on the discrete toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

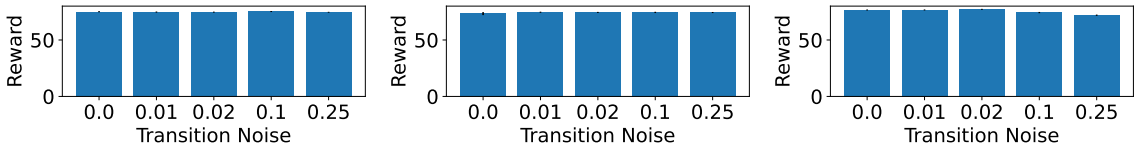


(a) DQN (b) Rainbow (c) A3C

Figure 21: AUC of episodic reward at the end of training for DQN, Rainbow, A3C when varying **transition noise** on the discrete toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

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

Figure 22: AUC of episodic reward at the end of training for DQN, Rainbow, A3C when varying **reward noise** on the discrete toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

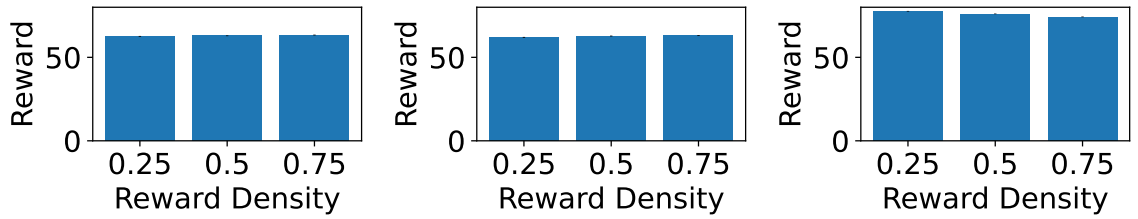


(a) DQN (b) Rainbow (c) A3C

Figure 23: AUC of episodic reward when rolling out the policy that was learned on the noisy environment on a noise-free setting at the end of training for DQN, Rainbow, A3C when varying **transition noise** on the discrete toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

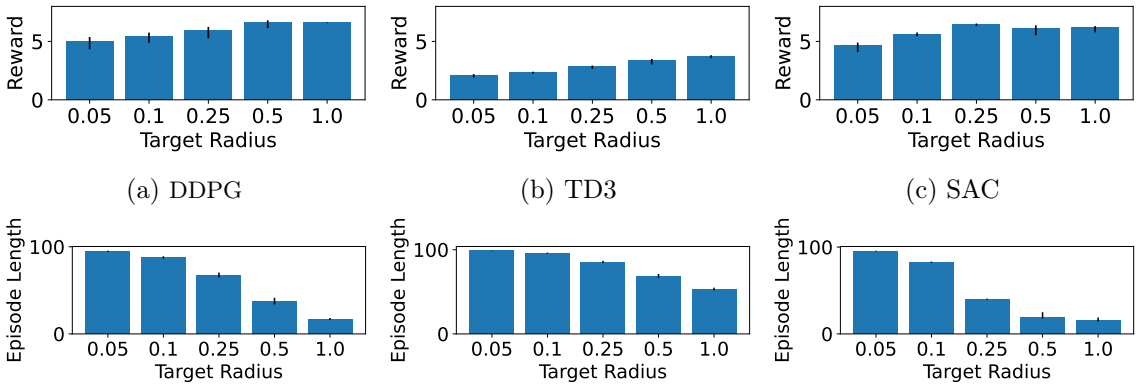
(a) DQN (b) Rainbow (c) A3C

Figure 24: AUC of episodic reward when rolling out the policy that was learned on the noisy environment on a noise-free setting at the end of training for DQN, Rainbow, A3C when varying **reward noise** on the discrete toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



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

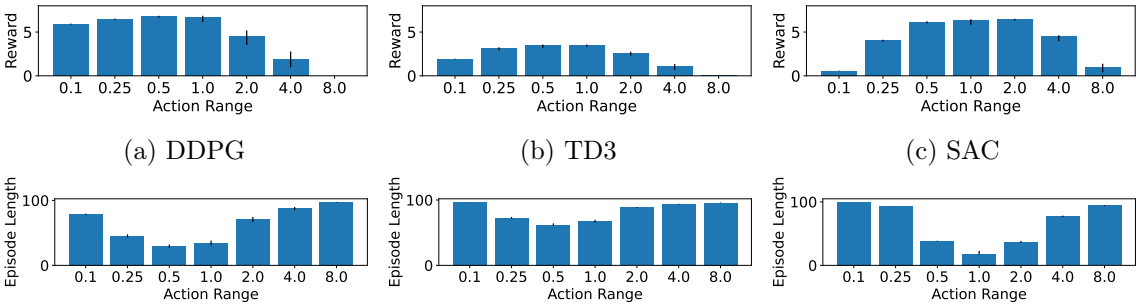
Figure 25: AUC of episodic reward at the end of training for DQN, Rainbow, A3C when varying **reward density** on the discrete toy environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



(a) DDPG (b) TD3 (c) SAC

(d) DDPG (e) TD3 (f) SAC

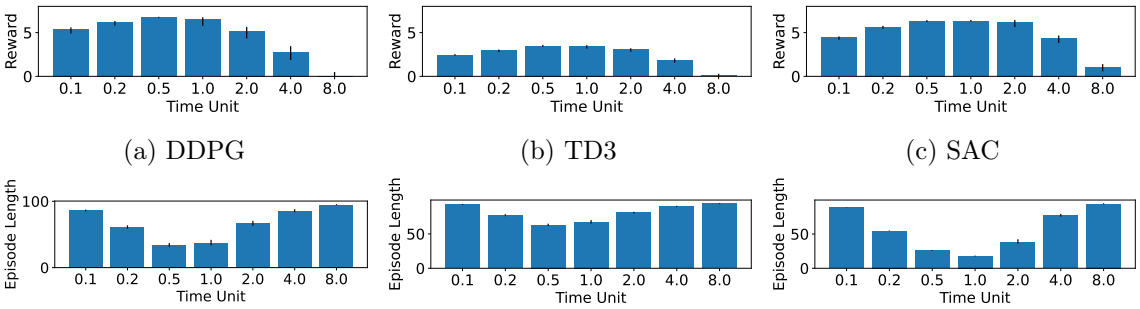
Figure 26: AUC of episodic reward (top) and lengths (bottom) for DDPG, TD3, SAC at the end of training when varying **target radius** on the discrete continuous environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



(a) DDPG (b) TD3 (c) SAC

(d) DDPG (e) TD3 (f) SAC

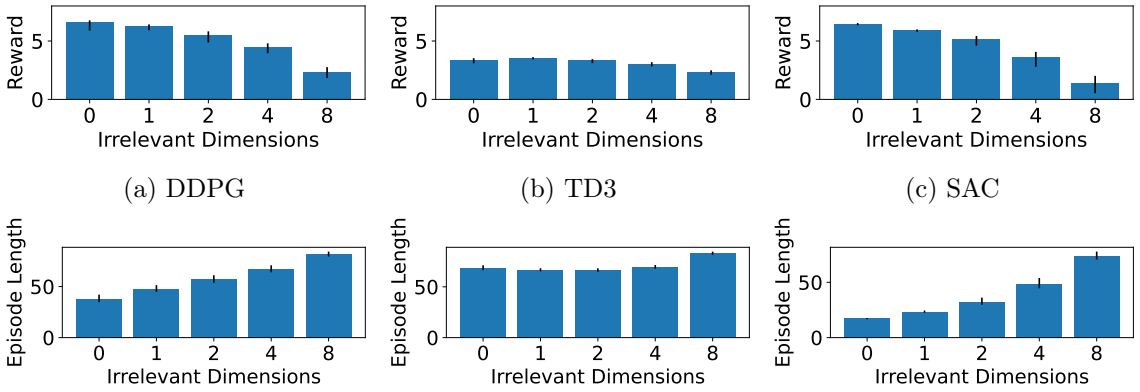
Figure 27: AUC of episodic reward (above) and lengths (below) for DDPG, TD3, SAC at the end of training when varying **action range** on the discrete continuous environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



(a) DDPG (b) TD3 (c) SAC

(d) DDPG (e) TD3 (f) SAC

Figure 28: AUC of episodic reward (above) and lengths (below) for DDPG, TD3, SAC at the end of training when varying **time unit** on the discrete continuous environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



(a) DDPG (b) TD3 (c) SAC

(d) DDPG (e) TD3 (f) SAC

Figure 29: AUC of episodic reward (above) and lengths (below) for DDPG, TD3, SAC at the end of training when varying **irrelevant dimensions** on the discrete continuous environment. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

B.3 Algorithms for Automatically Generated Toy MDPs with MDP Playground

Algorithm 1 Automatically Generated Discrete Toy MDPs with MDP Playground

Input:
number of actions $\|A\|$,
`diameter`,
reward density rd,
`terminal_state_density`,
reward delay d,
sequence length n,
transition noise t_n,
reward noise σ_{r_n},
`reward_scale`,
`reward_shift`,
`term_state_reward`,
`irrelevant_features`,
`image_representations`,
transforms for image representations `transforms`

function INIT_TRANSITION_FUNCTION():
    Set $\|NS\| = \|A\| * diameter$
    Divide NS into independent sets NS_i with $\|A\|$ elements in each with $i = 1, 2, ..., diameter$
    for each independent set NS_i **do**
        for each state ns in NS_i **do**
            Set possible successor states: $NS' = NS_{i+1}$
            for each action a **do**
                Set $P_{rel}(ns, a) = ns'$ sampled uniformly from NS' and remove ns' from NS'
    if irrelevant_features **then**
        Generate dynamics P_{irr} of irrelevant part of N-state space as was done for P_{rel}

function INIT_REWARD_FUNCTION():
    Randomly sample $rd * \frac{(\|NS\|-\|T\|)!}{(\|NS\|-\|T\|-n)!}$ and store in `rewardable_sequences`
    ▷ The actual formula is more complicated because of the diameter and independent sets. The formula shown here is valid for a diameter of 1.
            ▷ Only those sequences are sampled which are admissible according to P_{rel}

function TRANSITION_FUNCTION(ns, a):
    $ns' = P_{rel}(ns, a)$
    if $\mathcal{U}(0, 1) < t\_n$ **then**
        ns' = a random state in $NS \setminus \{P_{rel}(ns, a)\}$ ▷ Inject noise
    Observation $o = ns'$
    if `irrelevant_features` **then**
        Execute dynamics P_{irr} of irrelevant part of state space and append to ns' to get observation o
    if `image_representations` **then**
        Set $o =$ to image of corresponding polygon(s) with applied selected `transforms`
    return o

function REWARD_FUNCTION(ns, a):
    $r = 0$
    if `irrelevant_features` **then**
        $ns = ns[0]$ ▷ Select first dimension of the N-state which is relevant to the reward
    if not `make_denser` **then**
        if state sequence nss of n N-states ending d steps in the past is in `rewardable_sequences` **then**
            $r = 1$
    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**
                $r+ =$ i$/n$
    if *reward every n steps* and timesteps % n != 0 **then**
        $r = 0$
    $r \mathrel{+}= \mathcal{N}(0,\ {\sigma^2}_{r\_n})$
    $r \mathrel{*}=$ `reward_scale`
    $r \mathrel{+}=$ `reward_shift`
    if reached terminal state **then**
        $r \mathrel{+}=$ `term_state_reward` $*$ `reward_scale`
    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()

Algorithm 2 Automatically Generated Continuous Toy MDPs with MDP Playground

Input:
reward delay d,
transition noise σ_{t_n},
reward noise σ_{r_n},
`reward_scale`,
`reward_shift`,
`term_state_reward`,
`make_denser`,
`relevant_dimensions`,
`target_point`,
`target_radius`,
`transition_dynamics_order`,
`time_unit`,
`inertia`

function INIT_TRANSITION_FUNCTION():
    ▷ Do nothing as continuous environments have a fixed parameterisation
function INIT_REWARD_FUNCTION(n):
    ▷ Do nothing as continuous environments have a fixed parameterisation

function TRANSITION_FUNCTION(ns, a):
    Set $n =$ `transition_dynamics_order`
    Set $a^n = a$ ▷ Superscript n represents n^{th} derivative
    Set $ns^n = a^n / inertia$ ▷ Each state dimension is controlled by each action dimension
    for i in reversed(range(n)) **do**
        Set $ns^i_{t+1} = \sum_{j=0}^{n-i} ns_t^{i+j} \cdot \frac{1}{j!} \cdot time\_unit^j$ ▷ t is current time step.
    $ns_{t+1} += \mathcal{N}(0, \sigma^2{}_{t\_n})$
    $o = ns_{t+1}$
    if `image_representations` **then**
        Set left part of $o =$ to image of point mass, terminal regions and target point according to the first 2 dimensions of o
        if irrelevant features **then**
            Set right part of $o =$ to image of point mass according to the last 2 dimensions of o
    return o

```
function REWARD_FUNCTION(ns, a):
    r = 0
    if relevant_dimensions then
        ns = ns[relevant_dimensions]        ▷ Select the part of state space relevant to reward
    r = Distance moved towards the target_point
    r += N(0, sigma^2_r_n)
    r *= reward_scale
    r += reward_shift
    if reached terminal state then
        r += Sum of delayed rewards that were not handed out so far
        r += term_state_reward * reward_scale
    return r

function MAIN():
    INIT_TERMINAL_STATES() ▷ Set T to be states within target_radius of target_point
    INIT_INIT_STATE_DIST() ▷ Set rho_o to uniform distribution over non-terminal states
    INIT_TRANSITION_FUNCTION()
    INIT_REWARD_FUNCTION()
```

Appendix C. Effect of Dimensions on More Complex Environments

We include here the remaining plots and tables referenced in the discussion in the main paper that provide further detail on the experiments.

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

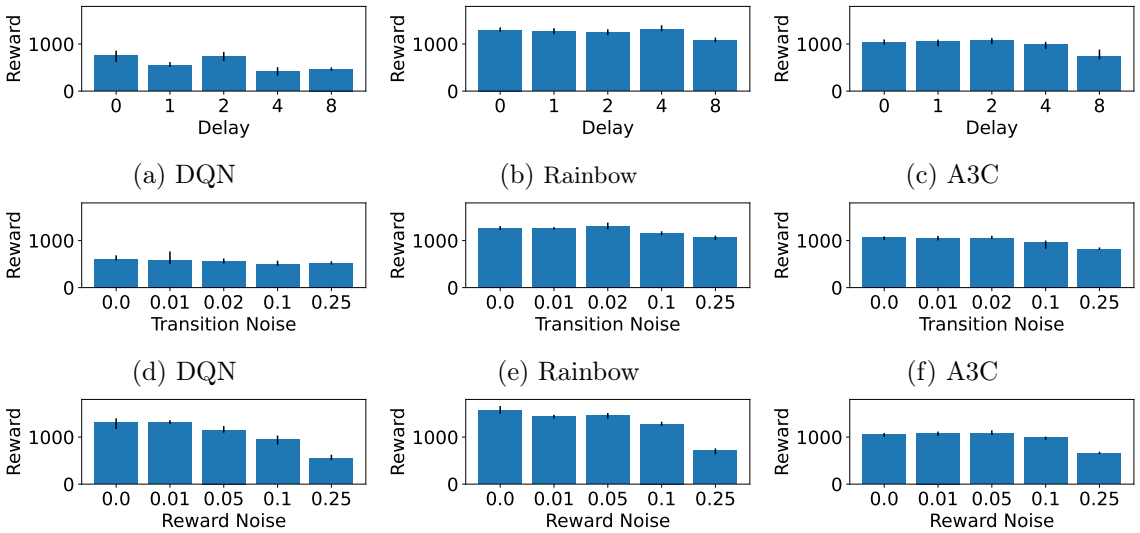
Environment/Agent	DQN	Rainbow	A3C
beam_rider	r=0.8, pvalue=0.104	r=0.4, pvalue=0.504	r=0.6, pvalue=0.284
breakout	r=0.0, pvalue=1.0	r=0.3, pvalue=0.623	r=0.8, pvalue=0.104
qbert	r=0.6, pvalue=0.284	r=0.4, pvalue=0.504	r=0.9, pvalue=0.037
space_invaders	r=0.9, pvalue=0.037	r=0.9, pvalue=0.037	r=0.8, pvalue=0.104

Table 3: Spearman Rank Correlations for performance on toy and complex environments across different amounts of the dimension injected: **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 4: 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

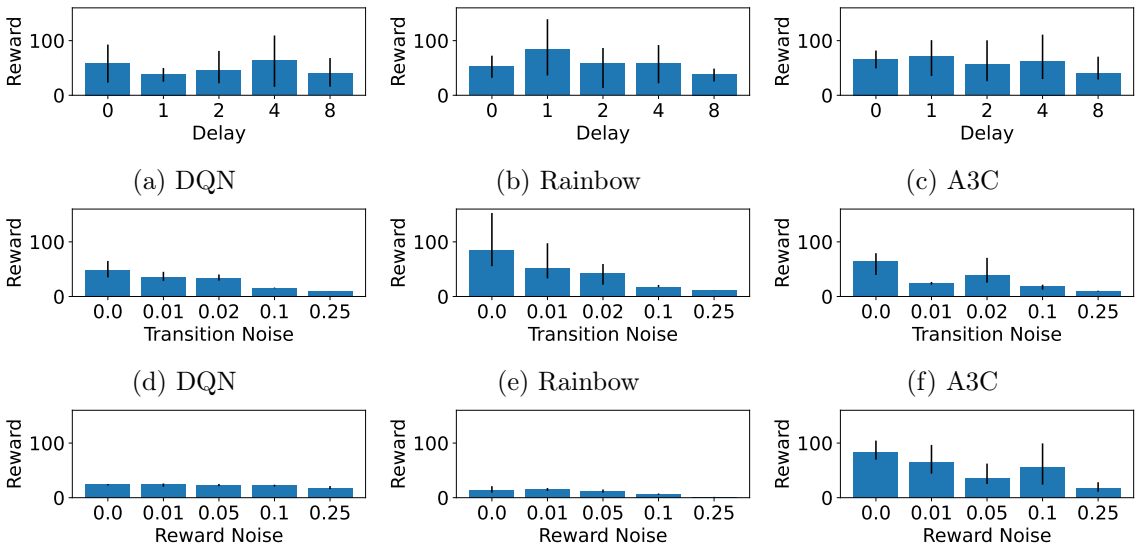


(a) DQN (b) Rainbow (c) A3C

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

(g) DQN (h) Rainbow (i) A3C

Figure 30: AUC of episodic reward for DQN, Rainbow, A3C at the end of training on *Beam Rider* for various dimensions of hardness. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

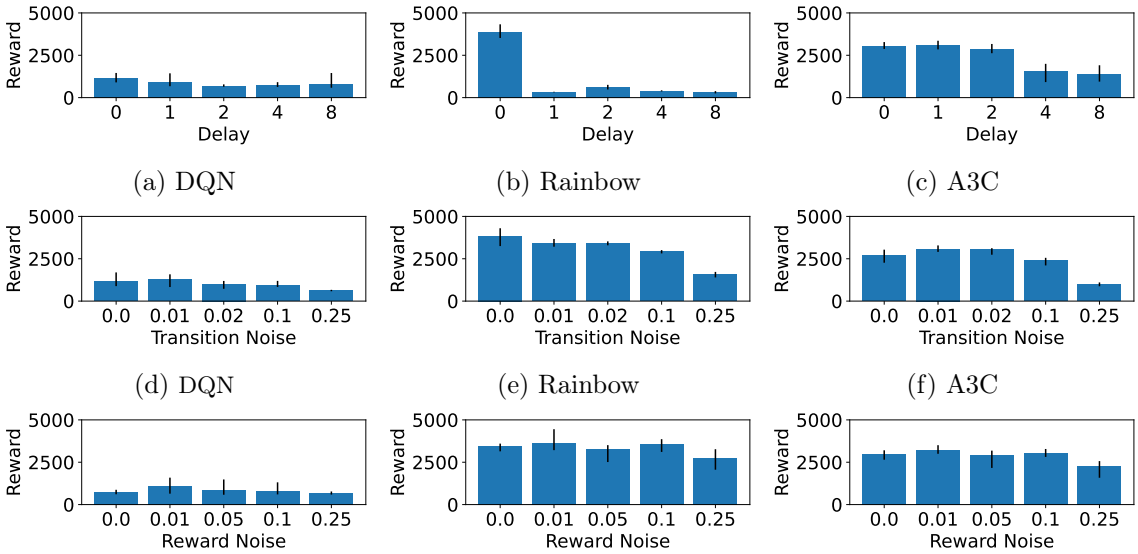


(a) DQN (b) Rainbow (c) A3C

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

(g) DQN (h) Rainbow (i) A3C

Figure 31: AUC of episodic reward for DQN, Rainbow, A3C at the end of training on *Breakout* for various dimensions of hardness. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

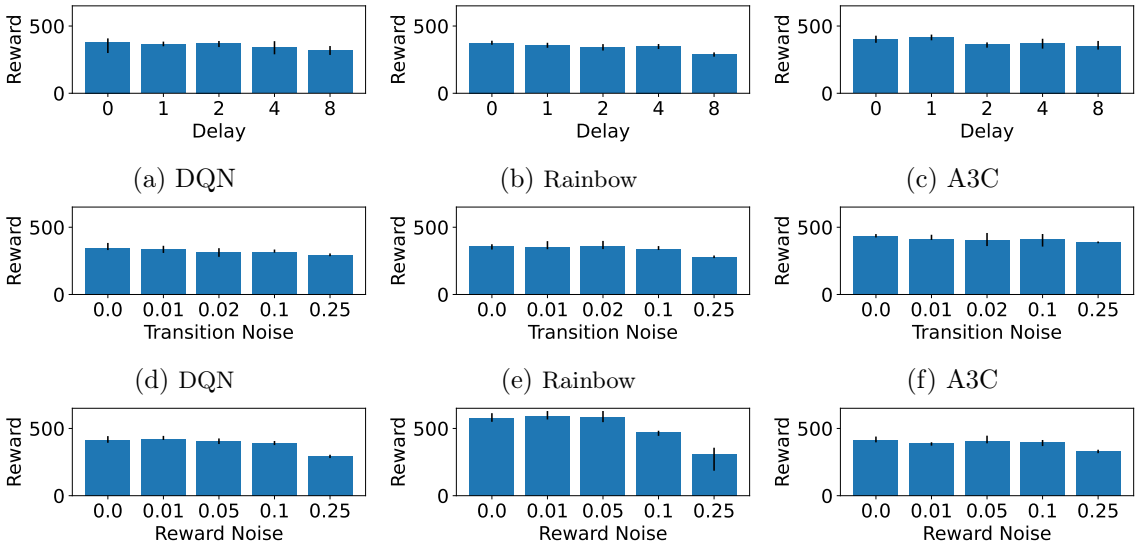


(a) DQN (b) Rainbow (c) A3C

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

(g) DQN (h) Rainbow (i) A3C

Figure 32: AUC of episodic reward for DQN, Rainbow, A3C at the end of training on *Qbert* for various dimensions of hardness. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

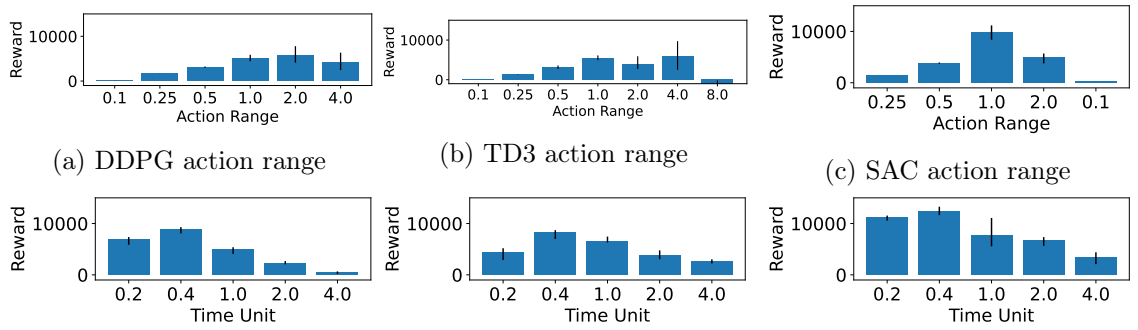


(a) DQN (b) Rainbow (c) A3C

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

(g) DQN (h) Rainbow (i) A3C

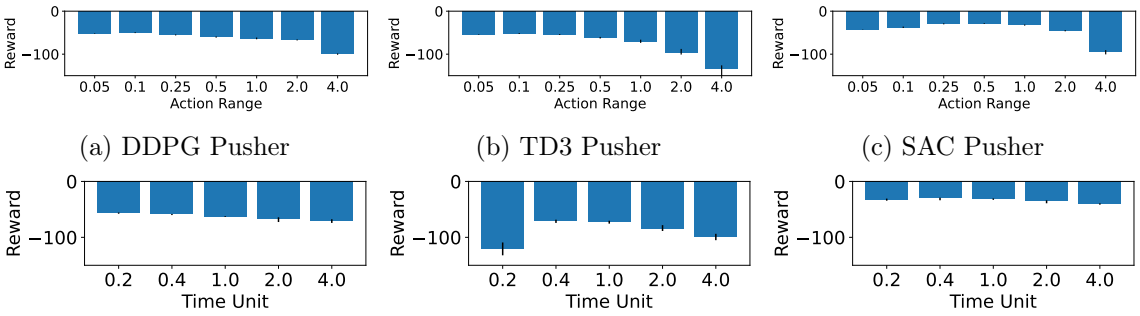
Figure 33: AUC of episodic reward for DQN, Rainbow, A3C at the end of training on *Space Invaders* for various dimensions of hardness. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



(a) DDPG action range (b) TD3 action range (c) SAC action range

(d) DDPG time unit (e) TD3 time unit (f) SAC time unit

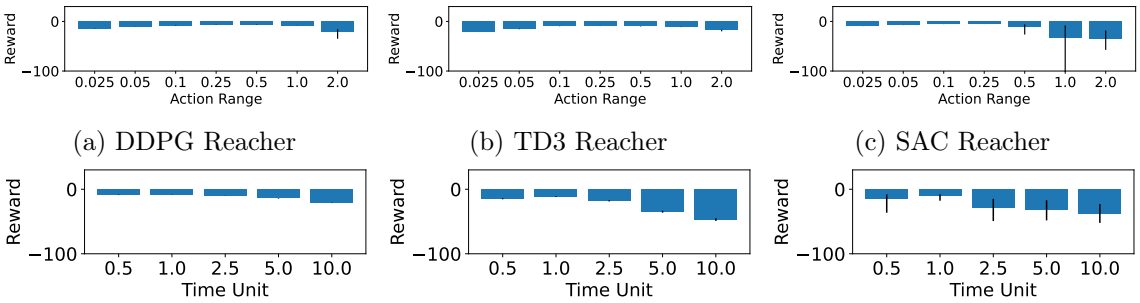
Figure 34: AUC of episodic reward for DDPG, TD3, SAC at the end of training on HalfCheetah when varying **action range** and **time unit**. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05. For SAC and DDPG, runs for *action range* values >= 2 and >= 4 crashed and are absent from the plot.



(a) DDPG Pusher (b) TD3 Pusher (c) SAC Pusher

(d) DDPG Pusher (e) TD3 Pusher (f) SAC Pusher

Figure 35: AUC of episodic reward for DDPG, TD3, SAC at the end of training on *Pusher* when varying **action range** and **time unit**. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



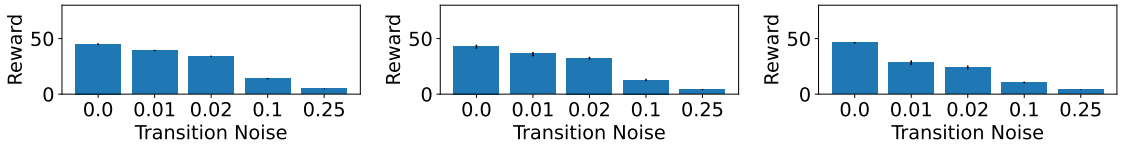
(a) DDPG Reacher (b) TD3 Reacher (c) SAC Reacher

(d) DDPG Reacher (e) TD3 Reacher (f) SAC Reacher

Figure 36: AUC of episodic reward for DDPG, TD3, SAC at the end of training on *Reacher* when varying **action range** and **time unit**. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

Appendix D. Plots for Tabular Baselines

We present here plots of experiments with the tabular baselines on the discrete toy environments with image representations turned off.

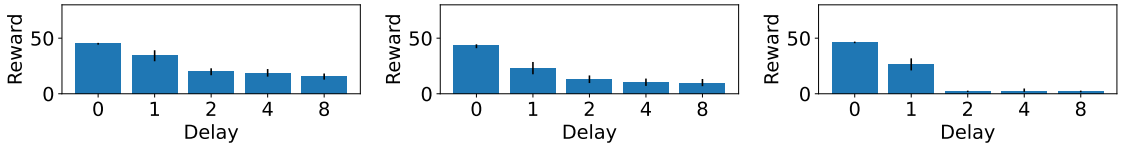


(a) Q-Learning (b) SARSA (c) Double Q-Learning

Figure 37: AUC of episodic reward at the end of training for three different tabular baseline algorithms when varying **transition noise**. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

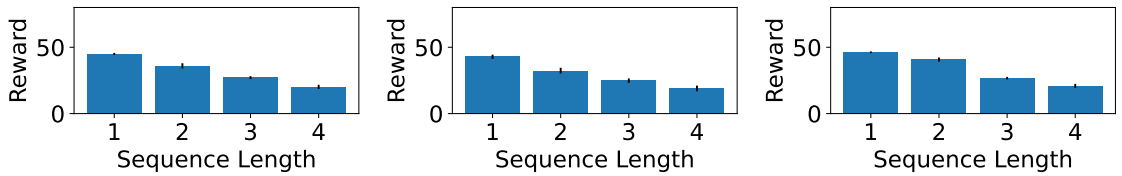
(a) Q-Learning (b) SARSA (c) Double Q-Learning

Figure 38: AUC of episodic reward at the end of training for three different tabular baseline algorithms when varying **reward noise**. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.



(a) Q-Learning (b) SARSA (c) Double Q-Learning

Figure 39: AUC of episodic reward at the end of training for three different tabular baseline algorithms when varying **reward delay**. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

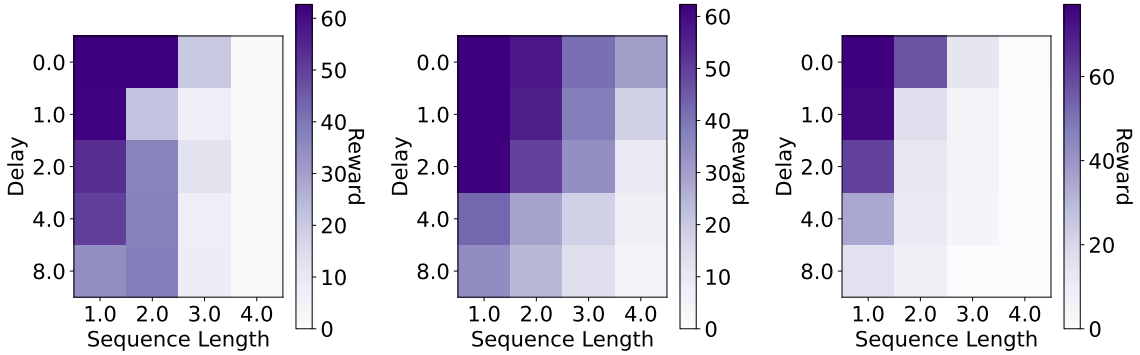


(a) Q-Learning (b) SARSA (c) Double Q-Learning

Figure 40: AUC of episodic reward at the end of training for three different tabular baseline algorithms when varying **sequence length**. Error bars represent bootstrapped confidence intervals with Bonferroni corrections for a significance level of 0.05.

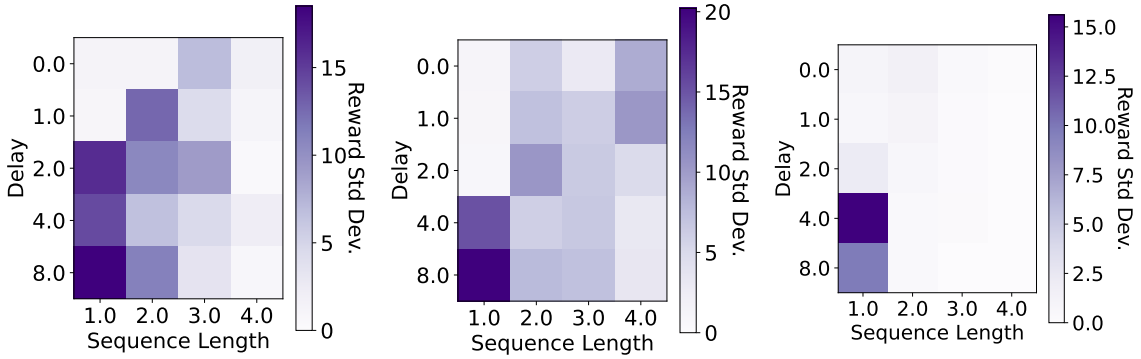
Appendix E. Plots for Varying 2 Hardness Dimensions Together

We present here some more plots of experiments with 2 dimensions of hardness varied together. The experiments in this section are only for 10 seeds as opposed to 100 seeds for the main paper.



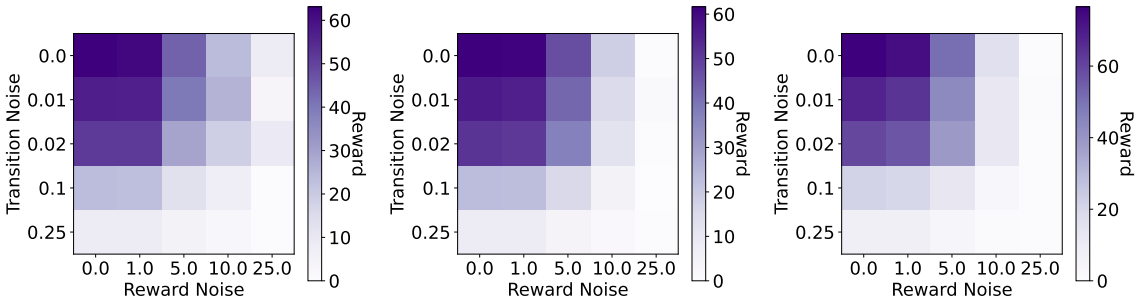
(a) DQN (b) Rainbow (c) A3C

Figure 41: AUC of episodic reward at the end of training for the different algorithms when varying **delay and sequence lengths**. Please note the different colour bar scales.



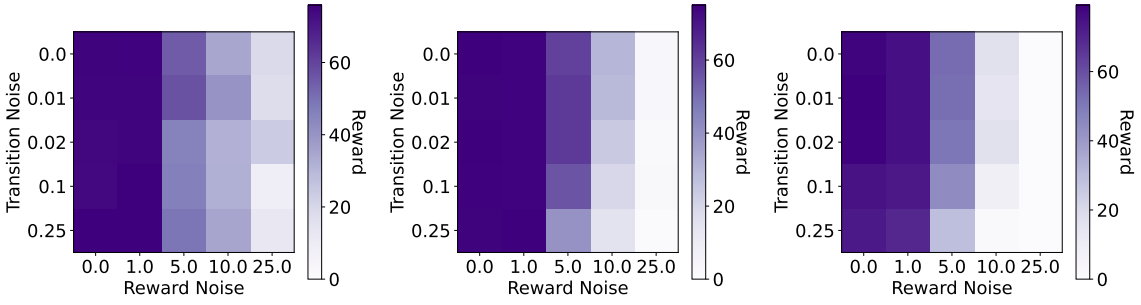
(a) DQN (b) Rainbow (c) A3C

Figure 42: Standard deviation of AUC of mean episodic reward at the end of training for the different algorithms when varying **delay and sequence lengths**. Please note the different colour bar scales.



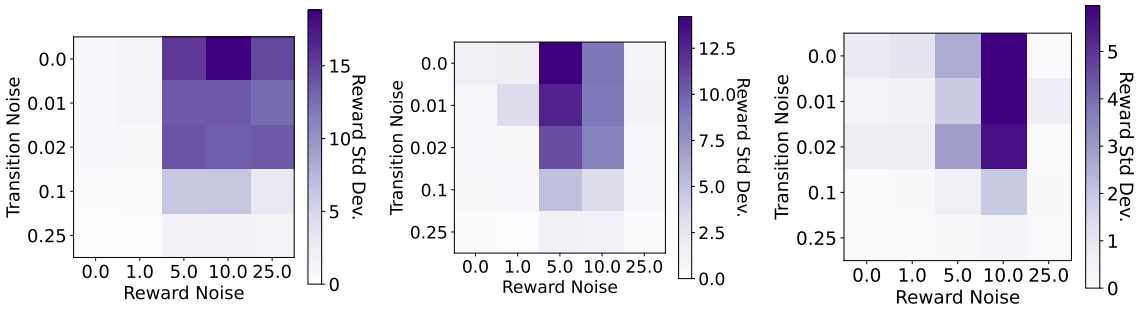
(a) DQN (b) Rainbow (c) A3C

Figure 43: AUC of episodic reward at the end of training for the different algorithms when varying **transition noise and reward noise**. Please note the different colour bar scales.



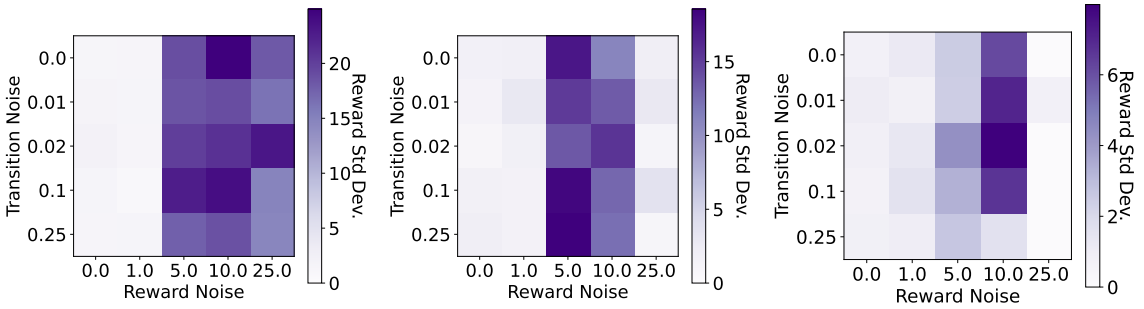
(a) DQN (b) Rainbow (c) A3C

Figure 44: AUC of episodic reward when varying **transition and reward noise** when rolling out the policy that was learned on the noisy environment on a noise-free setting for the different algorithms. Please note the different colour bar scales.



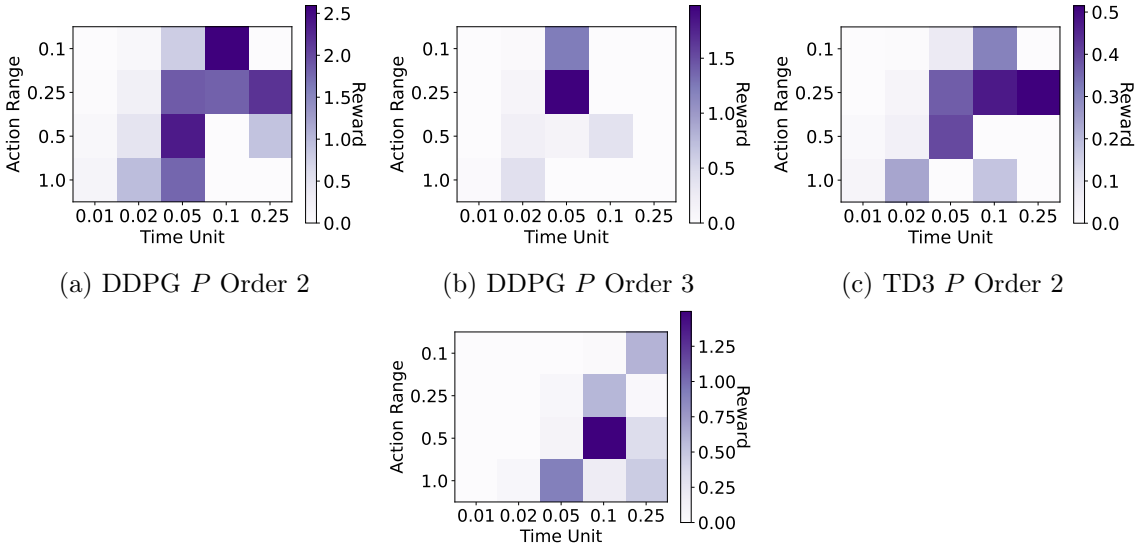
(a) DQN (b) Rainbow (c) A3C

Figure 45: Standard deviation of AUC of mean episodic reward at the end of training for the different algorithms when varying **transition noise and reward noise**. Please note the different colour bar scales.



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

Figure 46: Standard deviation of AUC of mean episodic reward when varying **transition and reward noise** when rolling out the policy that was learned on the noisy environment on a noise-free setting for the different algorithms. Please note the different colour bar scales.



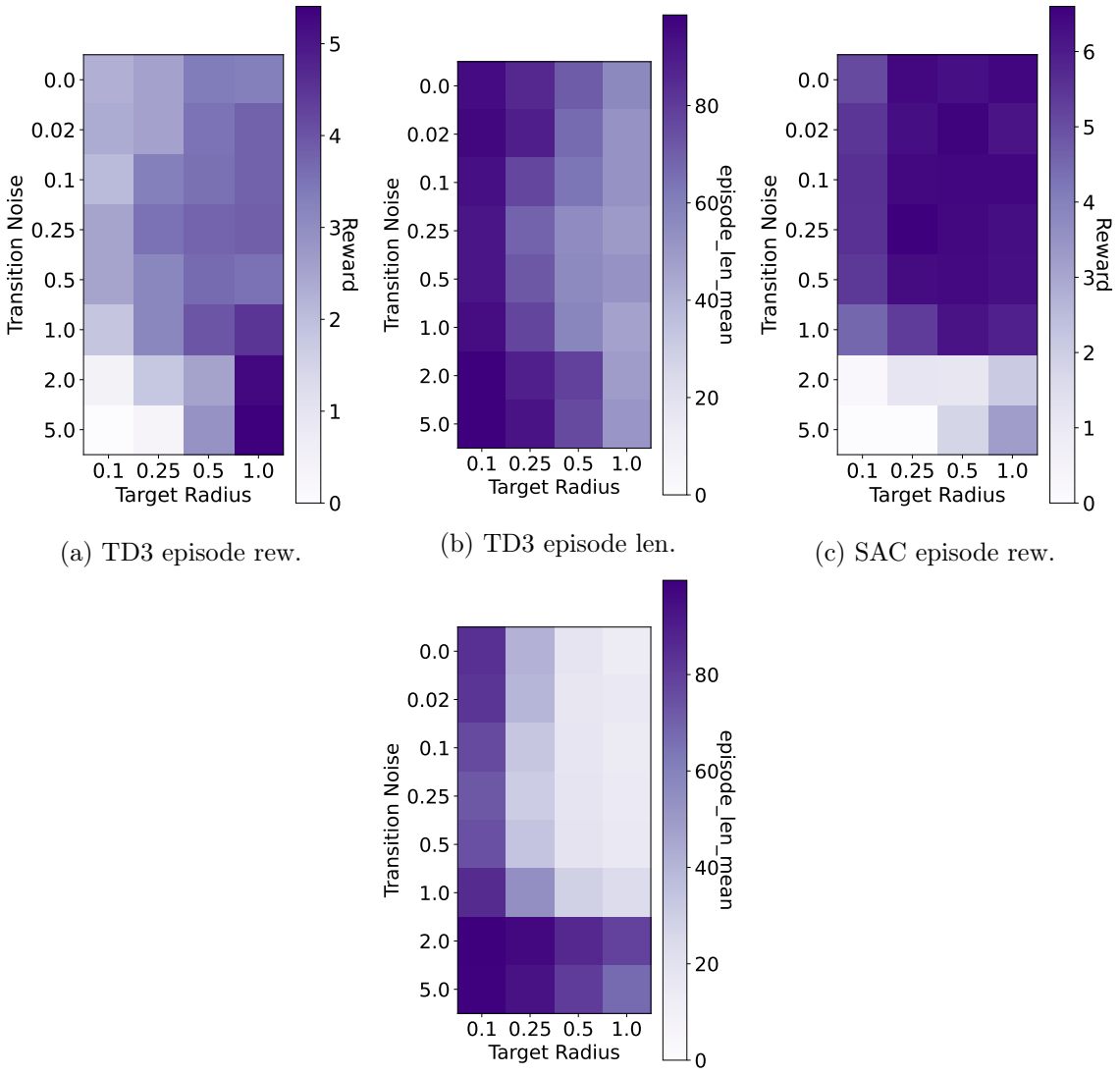
(a) DDPG P Order 2

(b) DDPG P Order 3

(c) TD3 P Order 2

(d) SAC P Order 2

Figure 47: AUC of episodic reward at the end of training for the different algorithms when varying **action space max and time unit for a given P order**. Please note the different colour bar scales.



(a) TD3 episode rew.

(b) TD3 episode len.

(c) SAC episode rew.

(d) SAC episode len.

Figure 48: AUC of episodic reward and lengths at the end of training for the different algorithms when varying P **noise and *target radius***. Please note the different colour bar scales.

Appendix F. 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.

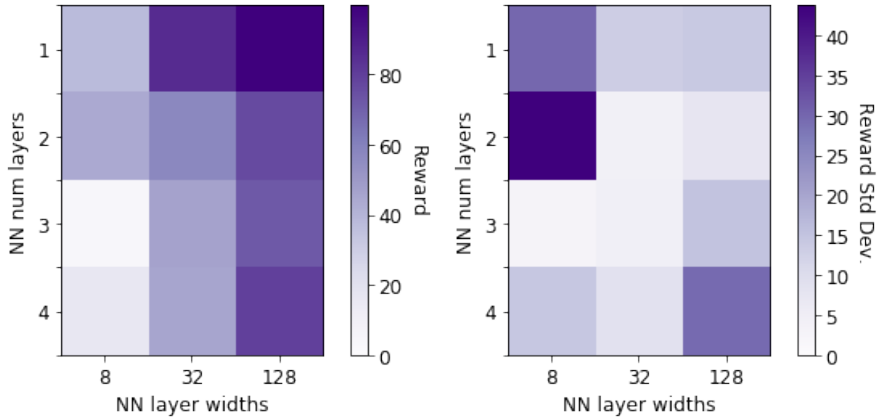
Hyperparameters were tuned for the vanilla environment; we did so manually in order to obtain good intuition about them before applying automated tools. We tuned the hyperparameters in sets, loosely in order of their significance and did 3 runs over each setting to get a more robust performance estimate. We describe an example of the tuning for DQN next. All hyperparameter settings for tuned agents can be found in Appendix G.

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 49a). However, the variance in performance was quite high (Figure 49b). 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 49c and 49d) 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 49e and 49f).

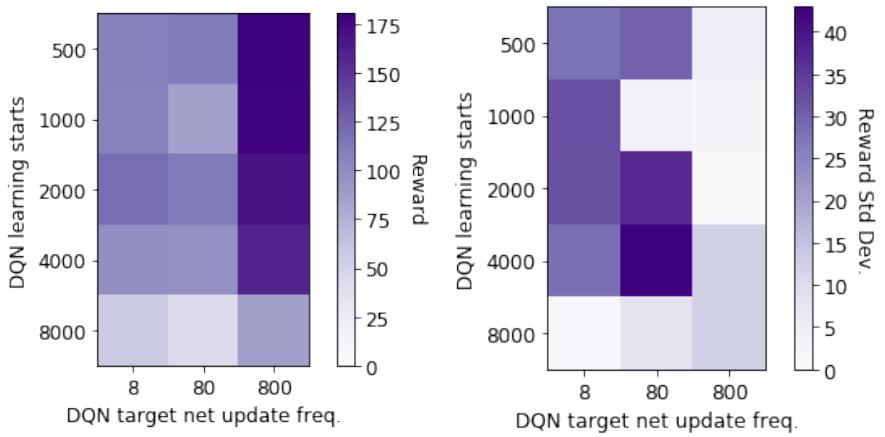
We then changed the network number of neurons grid to [128, 256, 512] and changed target network update frequency grid to [80, 800, 8000] and continued with further tuning using the grid values specified in Appendix G.

Appendix G. Tuned Hyperparameters

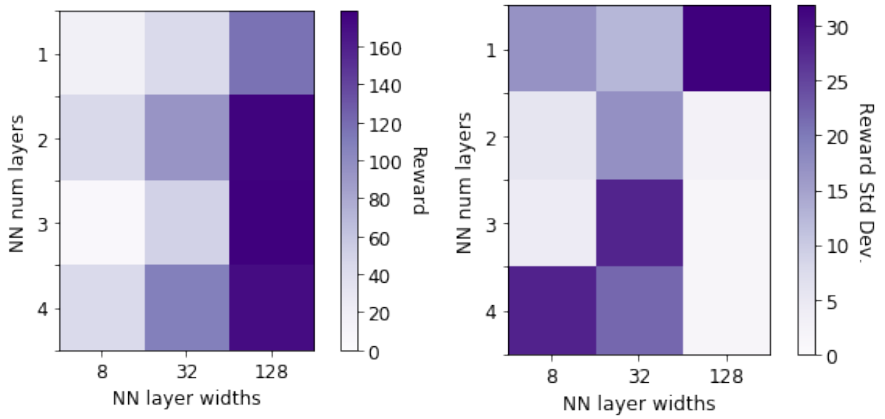
The code for corresponding experiments for both discrete and continuous environments can be found in the accompanying code for the paper. The experiments with *_tune_hps* in the names contain the grid of HPs that were tuned over. In some instances (where *_tune_hps* experiments do not exist), in order to save costs, we used the default HPs in Ray. The README in the GitHub repo 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 first few 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 agents again anyway, we decided to use the newer Ray version. The names of the hyperparameters for the algorithms in the *config* files will match those used in the respective Ray versions (i.e., 0.7.3 and 0.9.0).



(a) Reward (b) Std dev.



(c) Reward (d) Std. Dev.



(e) Reward (f) Std dev.

Figure 49: AUC of episodic reward at the end of training for different hyperparameter sets for DQN. Please note the different colour bar scales.

Appendix H. CPU Specifications

Cluster experiments were run on *Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz* cores for approximately 300000 CPU hours.

H.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 300000 hours of computation was performed on hardware of type Intel Xeon E5-2630v4 (TDP of 85W).

Total emissions are estimated to be 5140.8 kgCO_2eq of which 0 percents were directly offset.

Estimations were conducted using the MachineLearning Impact calculator presented in Lacoste et al. (2019).

The cluster CPU core specifications were:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
stepping        : 1
microcode       : 0xb000020
cpu MHz         : 1200.170
cache size      : 25600 KB
physical id     : 0
siblings        : 20
core id         : 0
cpu cores       : 10
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
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 arch_perfmon pebs bts rep_good
nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor
    ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1
    sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c r
drand lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3 invpcid_single intel_ppin
    intel_pt tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1
    hle avx2 smep bmi2 erms invpcid rtm cqm rdt_a rdseed adx smap x
saveopt cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat
    pln pts
bogomips        : 4389.48
```

```
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

The laptop CPU core specifications 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:
```

Appendix I. Additional Learning Curves for Toy Environments

Please find the learning curves for this section in the arxiv version of our paper: `https://arxiv.org/abs/1909.07750v3`. We present there plots of the learning curves of the experiments which provide more details into how learning progressed during training. The plots show how each seed performed. As a result, we plot them only for 10 seeds as opposed to 100 seeds for the main paper to maintain visual clarity.

Appendix J. Additional Learning Curves for Complex environments

Please find the learning curves for this section in the arxiv version of our paper: `https://arxiv.org/abs/1909.07750v3`. The curves contain a lot of data points and hence have large file sizes which leads to exceeding the file size limit for this submission.

References

Andersson, O., & Doherty, P. (2018). Toward robust deep rl via better benchmarks: Identifying neglected problem dimensions. In *2nd Reproducibility in Machine Learning Workshop at ICML, Stockholm, Sweden.*

Andrychowicz, M., Raichuk, A., Stanczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., & Bachem, O. (2021). What matters for on-policy deep actor-critic methods? A large-scale study. In *9th International Conference on Learning Representations.*

Arjona-Medina, J. A., Gillhofer, M., Widrich, M., Unterthiner, T., Brandstetter, J., & Hochreiter, S. (2019). RUDDER: return decomposition for delayed rewards. In *Proceedings of the 32nd International Conference on Advances in Neural Information Processing Systems*, pp. 13544–13555.

Atrey, A., Clary, K., & Jensen, D. D. (2020). Exploratory not explanatory: Counterfactual analysis of saliency maps for deep reinforcement learning. In *8th International Conference on Learning Representations.*

Bacchus, F., Boutilier, C., & Grove, A. (1996). Rewarding behaviors. In *Proceedings of the National Conference on Artificial Intelligence.*

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

Biedenkapp, A., Rajan, R., Hutter, F., & Lindauer, M. (2021). TempoRL: Learning when to act. In *Proceedings of the 38th International Conference on Machine Learning (ICML).*

Cassandra, A. R., Kaelbling, L. P., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In Hayes-Roth, B., & Korf, R. E. (Eds.), *Proceedings of the 12th National Conference on Artificial Intelligence*, pp. 1023–1028. AAAI Press / The MIT Press.

Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2018). Back to basics: Benchmarking canonical evolution strategies for playing atari. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pp. 1419–1426.

Cobbe, K., Hesse, C., Hilton, J., & Schulman, J. (2020). Leveraging procedural generation to benchmark reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, Vol. 119 of *Proceedings of Machine Learning Research*, pp. 2048–2056. PMLR.

Colas, C., Sigaud, O., & Oudeyer, P.-Y. (2018). How many random seeds? statistical power analysis in deep reinforcement learning experiments..

Dehghani, M., Tay, Y., Gritsenko, A. A., Zhao, Z., Houlsby, N., Diaz, F., Metzler, D., & Vinyals, O. (2021). The benchmark lottery. *arXiv*, *2107.07002*.

Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., & Hester, T. (2020). An empirical investigation of the challenges of real-world reinforcement learning. *arXiv*, *2003.11881*.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., & Madry, A. (2020). Implementation matters in deep RL: A case study on PPO and TRPO. In *8th International Conference on Learning Representations*.

Finn, C., Abbeel, P., & Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70, pp. 1126–1135.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Hessel, M., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., & Legg, S. (2018). Noisy networks for exploration. In *Proceedings of the International Conference on Learning Representations*.

François-Lavet, V., Fonteneau, R., & Ernst, D. (2015). How to discount deep reinforcement learning: Towards new dynamic strategies..

Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 1582–1591.

Gaina, R. D., Lucas, S. M., & Pérez-Liébana, D. (2019). Tackling sparse rewards in real-time games with statistical forward planning methods. In *Proceedings of the 33rd Conference on Artificial Intelligence*, pp. 1691–1698.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft Actor-Critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 1856–1865.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters. In *Proceedings of the Conference on Artificial Intelligence*, pp. 3207–3214.

Hendrycks, D., & Dietterich, T. G. (2019). Benchmarking neural network robustness to common corruptions and perturbations. In *Proceedings of the International Conference on Learning Representations*.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining improvements in deep

reinforcement learning. In *Proceedings of the Conference on Artificial Intelligence*, pp. 3215–3222.

Hu, Y., Wang, W., Jia, H., Wang, Y., Chen, Y., Hao, J., Wu, F., & Fan, C. (2020). Learning to utilize shaping rewards: A new approach of reward shaping. *Advances in Neural Information Processing Systems*, *33*, 15931–15941.

Irpan, A. (2018). Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`.

Jaksch, T., Ortner, R., & Auer, P. (2010). Near-optimal regret bounds for reinforcement learning. *JMLR*, *11*, 1563–1600.

Juliani, A., Khalifa, A., Berges, V., Harper, J., Teng, E., Henry, H., Crespi, A., Togelius, J., & Lange, D. (2019). Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 2684–2691.

Kanervisto, A., Scheller, C., & Hautamäki, V. (2020). Action space shaping in deep reinforcement learning. In *IEEE Conference on Games, CoG*, pp. 479–486.

Kim, H. J., Jordan, M. I., Sastry, S., & Ng, A. Y. (2004). Autonomous helicopter flight via reinforcement learning. In *Advances in neural information processing systems*, pp. 799–806.

Kirkebøen, G., & Nordbye, G. H. (2017). Intuitive choices lead to intensified positive emotions: An overlooked reason for "intuition bias"?. *Frontiers in Psychology*, *8*, 1942.

Klink, P., Abdulsamad, H., Belousov, B., & Peters, J. (2019). Self-paced contextual reinforcement learning. In *3rd Annual Conference on Robot Learning*, pp. 513–529.

Lacoste, A., Luccioni, A., Schmidt, V., & Dandres, T. (2019). Quantifying the carbon emissions of machine learning. *arXiv*, *1910.09700*.

LeCun, Y. (2012). Learning invariant feature hierarchies. In *European Conference on Computer Vision*, pp. 496–505.

LeCun, Y., & Cortes, C. (2010). MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., & Stoica, I. (2018). RLlib: Abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80, pp. 3059–3068.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In *Proceedings of the International Conference on Learning Representations*.

Littman, M. L., Topcu, U., Fu, J., Jr., C. L. I., Wen, M., & MacGlashan, J. (2017). Environment-independent task specifications via GLTL. *arXiv*, *1704.04341*.

Maillard, O., Mann, T. A., & Mannor, S. (2014). How hard is my mdp?" the distribution-norm to the rescue". In *Advances in Neural Information Processing Systems 27*, pp. 1835–1843.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, Vol. 48, pp. 1928–1937.

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

Mu, N., & Gilmer, J. (2019). MNIST-C: A robustness benchmark for computer vision. *arXiv*, *1906.02337*.

Nau, D. S. (1983). Pathology on game trees revisited, and an alternative to minimaxing. *Artificial Intelligence*, *21*(1-2), 221–244.

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepezvari, C., Singh, S., Roy, B. V., Sutton, R., Silver, D., & Hasselt, H. V. (2019). Behaviour suite for reinforcement learning. In *Proceedings of the International Conference on Learning Representations*.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley.

Rajendran, J., Ganhotra, J., Singh, S., & Polymenakos, L. (2018). Learning end-to-end goal-oriented dialog with multiple answers. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium*, pp. 3834–3843.

Ramanujan, R., Sabharwal, A., & Selman, B. (2010). On adversarial search spaces and sampling-based planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pp. 242–245.

Runge, F., Stoll, D., Falkner, S., & Hutter, F. (2019). Learning to design rna. In *International Conference on Learning Representations*.

Subramanian, J., Sinha, A., Seraj, R., & Mahajan, A. (2020). Approximate information state for approximate planning and reinforcement learning in partially observed systems. *arXiv*, *2010.08843*.

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

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press.

Tessler, C., & Mannor, S. (2020). Reward tweaking: Maximizing the total reward while planning for short horizons..

Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic robotics*. Intelligent robotics and autonomous agents. MIT Press.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 23–30.

Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*, pp. 5026–5033.

Tosch, E., Clary, K., Foley, J., & Jensen, D. D. (2019). Toybox: A suite of environments for experimental evaluation of deep reinforcement learning. *arXiv*, *1905.02825*.

van Hasselt, H. (2010). Double q-learning. In *Proceedings of the 24th International Conference on Advances in Neural Information Processing Systems*, pp. 2613–2621.

Wang, T., Bao, X., Clavera, I., Hoang, J., Wen, Y., Langlois, E., Zhang, S., Zhang, G., Abbeel, P., & Ba, J. (2019). Benchmarking model-based reinforcement learning. *arXiv*, *1907.02057*.

Xu, Z., van Hasselt, H. P., & Silver, D. (2018). Meta-gradient reinforcement learning. *Advances in neural information processing systems*, *31*.

Young, K., & Tian, T. (2019). Minatar: An atari-inspired testbed for more efficient reinforcement learning experiments. *arXiv*, *1903.03176*.

Zhang, B., Rajan, R., Pineda, L., Lambert, N., Biedenkapp, A., Chua, K., Hutter, F., & Calandra, R. (2021). On the Importance of Hyperparameter Optimization for Model-based Reinforcement Learning. In *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics*.

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2017). Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations*.

Zheng, Z., Oh, J., & Singh, S. (2018). On learning intrinsic rewards for policy gradient methods. *Advances in Neural Information Processing Systems*, *31*.

Zou, H., Ren, T., Yan, D., Su, H., & Zhu, J. (2021). Learning task-distribution reward shaping with meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35, pp. 11210–11218.