On Temporally-Extended Reinforcement Learning in Dynamic Algorithm Configuration

Tilman Räuker

15.11.2021

A thesis presented for the degree of
Master of Science

Supervised by
Theresa Eimer, M. Sc.

Examination committee:
Prof. Dr. rer. nat. Marius Lindauer
Prof. Dr.-Ing. Bodo Rosenhahn

Gottfried Wilhelm Leibniz Universität Hannover

[Redacted: *Declaration of Authorship* and *Assignment of the Master Thesis*, because of personal information]

Contents

1 Why Temporally-Extended Reinforcement Learning in Dynamic Algorithm Configuration?

Finding well-performing hyperparameters for machine learning algorithms is a complex process. If we try to adjust the hyperparameter, either excellent knowledge about the specific domain or great computational effort is necessary to find a configuration for the problem at hand. Suppose we find a suitable configuration for our problem and the settings of the problem changes only slightly. In that case, we might be confronted with the fact that our previously tuned hyperparameters might not work at all. One solution to this problem is to update these hyperparameters iteratively. Dynamically adjusting hyperparameters of an algorithm has previously been established in the area of adaptive and reactive heuristics [1, 2]. Dynamic Algorithm Configuration laid the foundation for adjusting the hyperparameters of different problem instances online and formalized the process as a contextual Markov Decision Process [3]. Formulated as a Markov Decision Process, it becomes feasible to use Reinforcement Learning to solve the problem: an agent configures the hyperparameter of a different target algorithm at each time step of the decision process. The different hyperparameters build the action-space, various statistics about the target algorithm its states, and its successes as a reward.

But changing the hyperparameters *at each step* seems unlikely to be optimal, especially if we think about hyperparameters like the learning rate. Imagine a scenario where the task is to update an algorithm's learning rate iteratively. The algorithm takes a step, and the reward indicates some success of the executed learning rate. The optimal learning rate in the next step is probably close to the learning rate of the previous one. Since we are more interested in finding the minimum of a function rather than the performance of the learning rate at a particular time step, the performance of the adjusted value will become more evident when observed over an extended period of time. Therefore, probing random values now could lead to suboptimal feedback for the controller of the learning rate.

This thesis's central question is: does it improve performance to observe actions for multiple steps before readjusting for Dynamic Algorithm Configuration? What are reliable ways to include action repetition in Reinforcement Learning? Are those methods any faster in learning favourable policies than standard one-step methods? One way to answer these questions is through comparing standard methods to temporally extended methods on benchmarks. The overall performance of these different methods shows how helpful repetition of actions is for Dynamic Algorithm Configuration. Also, studies of the resulting policies show if action repetition were crucial to success.

First, the Reinforcements Learning basics are covered to understand Dynamic Algorithm Configuration (DAC) and its agents. Then, we will follow with literature on the related work of DAC and exploration in Reinforcement

Learning. Afterwards, we will compare the most promising methods in temporal extended Reinforcement Learning for DAC on suitable benchmarks and discuss the results. Finally, we provide an overview of promising research directions for the future.

2 Background of Dynamic Algorithm Configuration

The performance of an algorithm in the field of machine learning depends heavily on its hyperparameters (HP) [4, 5, 6, 7, 8]. When we adjust the hyperparameters of these algorithms for a problem, we get into a predicament: we either need superior domain expertise or computational resources to find hyperparameter configurations that work well [9, 10]. Automated approaches to hyperparameter optimization often neglect the iterative nature of most algorithms and assume that the optimal hyperparameter configuration won't change over time [11].

One way to find a policy to configure hyperparameters online is through Reinforcement Learning [12, 13, 14, 15, 16]. An agent sets a HP configuration and gets feedback on how the HP setting works for the problem, e.g., the successes measured by the problem environment. However, finding HP configurations within only one task results in a policy that most likely overfits to that specific or very similar instances of that problem. If we are interested in finding policies for adjusting HP for various problems online, we need another framework. One solution to this predicament is Dynamic Algorithm Configuration.

2.1 Reinforcement Learning

To build a mathematical basis for reinforcement learning, we define problems as Markov Decision Processes (MDP).

An MDP is a 4-tuple $\mathcal{M} := \{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}\}$, consisting of a state-space $\mathcal{S}$, an action-space $\mathcal{A}$, the transition probabilities between states $\mathcal{T}$ and the reward function $\mathcal{R}$. The premise of MDPs is that the probability of reaching a state $s_{t+1} \in \mathcal{S}$ from state $s_t \in \mathcal{S}$ depends merely on s_t and not on predecessors of s_t. The action a chosen at any time step influences these transitions, thus the transition probability can be formulated as $\mathcal{T}_a(s, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$. The reward $r \in \mathcal{R}$ is obtained after the transition from state s to state s'. Usually, it is interesting to find the mapping between state-space and action-space, also known as the policy π. In that case, the goal of the MDP could be a policy that maximizes cumulative rewards. How various algorithms achieve this is discussed in the chapter 6.

2.2 Reinforcement Learning with Context

Training an agent only on one instance of a problem will most likely lead to good performance on that specific instance. However, it does not necessarily generalize to variations of the problem, even if those have significant similarities and differ only in certain aspects. To address this, Hallak et al. introduced contextual MDPs [17]. The contextual MDP $\mathcal{M}_\mathcal{I}$ is modelled as a 4-tuple

$$\mathcal{M}_\mathcal{I} \coloneqq \{\mathcal{M}_i\}_{i \sim \mathcal{I}} \quad \text{with} \quad \mathcal{M}_i \coloneqq (\mathcal{S}, \mathcal{A}, \mathcal{T}_i, \mathcal{R}_i). \tag{1}$$

Here the mathematical notion of an instance is indicated by i in the index. The state- and action-space are shared across instances. The probability distribution $\mathcal{T}$ of algorithm state transitions and the reward function $\mathcal{R}$ are dependent on a particular instance. This means that an algorithm can be confronted with states in which applying an action in different instances can lead to different state transitions. It also means that transitions that are considered beneficial in one instance may be detrimental in another. There might be instance-specific information in some applications that could enhance the state-space; Hallak et al. refer to them as *context*. One example Hallak et al. present for contextual MDP is modelling a customer's behaviour when interacting with a website. The website's objective is to optimize the user experience by predicting the behaviour. While the state-space is some stage of the website, the contextual information in this problem could be information about the user, like age or location.

With the discussion on the different elements of contextual MDPs, it is worth emphasizing that they formulate a set of different MDPs with a shared action- and state-space. The goal in contextual MDPs is to learn a policy that generalizes across instances $i \in \mathcal{I}$.

2.3 Dynamic Algorithm Configuration

With DAC, we want to optimize hyperparameters across various versions of a target algorithm and as the algorithm itself unfolds. To achieve this, we formulate the problem as a contextual MDP. In the framework of DAC, the state-space $\mathcal{S}$ is a set of available information about the algorithm we want to configure - the target algorithm. This state information is e.g. the latest reward received or the amount of played time steps so far. The instance-specific information is e.g. the problem size at hand or the complexity of the problem, which could be the order of degree of a function we want to approximate.

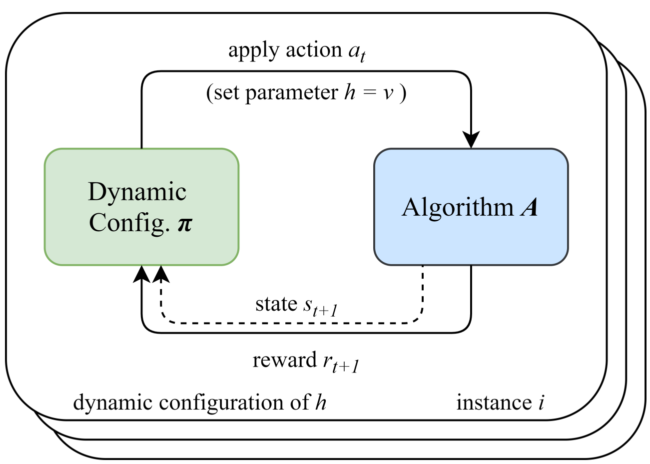

Figure 1: Interaction between DAC and the target algorithm

The action-space includes all HPs that DAC can change. These are hyperparameters like learning rate, batch size, or target update frequency, although they always depend on the target algorithm at hand. An overview of the general process can be found in the figure 1.

2.4 Examples of DAC

In the foundational paper of DAC by Biedenkapp et al. two white-box benchmarks, Luby and Sigmoid, were introduced [3]. The benchmarks are easily computable number sequences, and the agents' task is to approximate the underlying equation. The equations are the respective ground truth of the data, which makes the task transparently solvable. The instances consist of different lengths and different parameterizations of the functions. These benchmarks run quickly without an underlying target algorithm and thus are easy to implement and ideal for testing DAC agents. In the paper, the authors demonstrate an ϵ-greedy Q-learning algorithm once tabular and once with a function approximation inspired by DQN to show solvability.

In a subsequent paper, Shala and Biedenkapp et al. introduce the CMA-ES environment [18]. Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is an optimization algorithm of a black-box objective function. The goal is to learn a policy that controls the mutation step-size parameter of CMA-ES. The authors make use of guided policy search (GPS) to learn a DAC policy. They were able to show that DAC based methods generalize over the instances and beat hand-crafted heuristics.

Another example for the application of DAC is in AI planning problems, e.g. by Speck and Biedenkapp [19]. The Benchmark FastDownward includes six different planning problems that come in variations of difficulty and size. The task for the DAC agent is to choose the best, out of four different, planning heuristics at each time step. Using a $\epsilon-greedy$ deep Q-learning algorithm implemented as double DQN, they beat hand-crafted policies in 5/6 problem domains.

DACBench by Eimer et al. unites the previously named benchmarks into one library [20]. These benchmarks add up to Sigmoid and Luby [3], CMA-ES [18], FastDownward [19] and additionally adds ModEA, a sophisticated variation to the CMA-ES on BBOB functions [21], and SGD-DL, where the task is to control a neural networks learning-rate in a simple image classification task [13]. The paper also presented some baseline values for these environments, showing dynamic policies' superiority compared to static ones.

2.5 Implicit examples of DAC

Most methods that configure the hyperparameters of the algorithms with RL can be interpreted as DAC. In these settings, variations to the problem are usually not considered. There are a lot of examples of this; the following is a non-exhaustive list:

Jomaa et al. propose to adjust the hyperparameter of an algorithm merely through the feedback of the validation loss [16]. The target algorithms are classification algorithms for 50 randomly selected UCI data sets. Here LSTM-cells build the basis of the models' architecture.

The paper that inspired the SGD-DL Benchmark in DACBench also presents a DAC application[20]: Daniel et al. investigate algorithms that control neural networks' learning rate [13]. For this, the authors identified crucial features of the environment to learn the hyperparameter, e.g. the predictive change in function value. Note that the DAC agent is a controller that samples each episode from a distribution of 20 parameter vectors θ to learn the policy $\pi(\theta)$ The authors showed that the controller generalizes across different network architectures and tasks. Similar to this is the work of Xu et al. [22]. Here the authors use two main features unlike Daniel et al.: a learning rate scaling function as the action and the validation loss as reward of the controller.

Sakurai et al. suggest a method, "adaptive parameter control", which adjusts parameters for evolutionary algorithms [15]. Here the reward from the target algorithm incorporates the genetic algorithms search characteristic as additional information to the algorithm's success.

To address generalization in HP optimization Almeida et al. propose a system for learned optimizers [23]. They achieve this through updates on hyperparameters that merely happen based on the algorithm's performance. These updates are decoupled from the task at hand.

The paper by Sharma et al. also studies adjusting parameters of evolutionary algorithms [14]. The authors use a DDQN network to control the mutation strategies of Differential Evolution (DE) [24]. This multistage process is initialized by accumulating multiple DE features and analyzing different reward functions. Then the DDQN predicts the most rewarding mutation strategy for the parent according to the DE state.

3 Environments

To test possible temporal extended methods for DAC, we deploy them in two different environments from DACBench [20]: FastDownward, ToySGD and cSigmoid.

3.1 FastDownward

FastDownward is an AI planning system consisting of different tasks which need to be solved [25]. The agent in this environment chooses between different heuristics to solve each task. In the basic setting of this environment, those are two complementary heuristics. The task is to choose heuristics to solve the tasks as quickly as possible because each time step the agent doesn't solve the task, it receives a reward of -1. In total there are 30 instances consisting of different problem settings, resulting in various difficulties. The state-space consists of each heuristic's minimum, maximum, and average values and its variances and open list entries.

There are many advantages to using FastDownward to test temporally extended DAC. The optimal policy of the instances are known, and some of the policies rely on simple action repetition, while others switch the action often. These different action repetition lengths could ideally be learned faster and more robustly through temporally extended RL.

3.2 ToySGD

In ToySGD, abbreviated for toy Stochastic Gradient Descent, we have a set of different second-order polynomial functions. At every step, we try to determine the ideal step size to take steps according to the gradient to find the minimum of the function, see figure 2. We also control the momentum parameter of SGD, which guides the moving average of the gradient. Both actions are internally converted to 10^{action}. It is called ToySGD because it is a simplified version of the standard SGD benchmark, where the task is to control the learning rate of an optimizer in a neural network.

The environment provides information about the gradient, the limit of steps, and the last performed action. We take a total of 11 steps until the environment terminates. The reward is the negative log regret of the action.

Using an extended temporal algorithm in this environment could be beneficial for training. If the target algorithm changes the skip length too often, the DAC agent gets less valuable feedback about the specific quality of the current learning rate. However, if the target algorithm sticks to a learning rate for multiple steps, it could be possible that the DAC agents find reasonable working learning rates more robustly. The flip side of that is, of course, that if the agent changes the rate too seldom, it might not find a valuable learning rate as quickly.

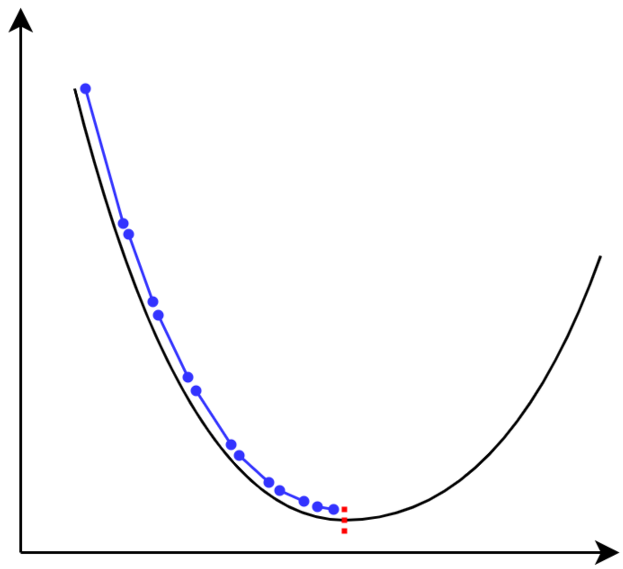

Figure 2: The figure shows the skip length approaching the minimum, usually shortening on the way to the minimum

.

3.3 Continuous Sigmoid

A variation of the Sigmoid Benchmark proposed by Biedenkapp et al. [3] is Continuous Sigmoid: the task is to approximate Sigmoid functions at each time step t of the function with continuous values

$$sig(t; s_{i,h}, p_{i,h}) = (1 + e^{-s_{i,h}(t-p_{i,h})})^{-1}. \quad (2)$$

The instances set consists of different parameterizations of the Sigmoid function, the inflection point p and the scaling factor s, which have the dimension h. From this follows that the agent not only has to learn to approximate the values for some set of the sigmoid function but to learn the shape of sigmoid functions within each specific instance. Here, the state-space includes these values as well as a time feature of the function. The reward ultimately is the absolute distance between the chosen values and the actual function. Using action repetition, in this case, could be interesting because different sigmoid functions have different curvatures. Especially if some function are not that steep, the agent could benefit from sticking to actions. On the other hand, if the slopes are steep, action repetition would fail to achieve more rewards.

4 Exploration in Reinforcement Learning

There are two main problems within exploration in Reinforcement Learning [26]. The first is the "hard problem" of exploration: handling sparse rewards in a large environment. Random exploration will rarely yield helpful rewards if an environment rarely gives feedback on how successful the agent behaves. The second challenge is the noisy-TV problem, which addresses the issue of using novelty as motivation in exploration. Because a noisy display continuously produces random new states, an agent would observe the TV until the game terminates. Naively implemented novelty bonuses could lead to an agent making no progress in a game at all.

Exploration strategies usually aim to address at least one of these problems. We will call strategies that primarily try to address the hard problem as targeted exploration strategies. These strategies are not wholly based on taking random steps in the environment but try to aim for novelty in some regard. On the other hand, we will call strategies that mainly undertake the noisy TV problem as non-targeted exploration strategies.

4.1 Non-targeted Exploration

Non-targeted exploration can be technically described as an exploration where the agent's previous state, reward, or action does not influence the exploration process. This is usually achieved by introducing noise to various points during the learning process. However, one could imagine non-targeted exploration through heuristically modifying values. First, we focus on noise in the action-space.

In ϵ-greedy, the agent follows one central premise: take the optimal action and with a probability of ϵ sample action from a uniform distribution [27]. Usually, after some steps, the agent learns something in the environment and does not need to heavily explore as it does at the beginning. Because of this, often a ϵ-decay is implemented, starting at some initial ϵ (e.g. 1) and decreasing for some amount of time steps at a final ϵ (e.g. 0.01).

But noise offers more: it is also the primary tool for exploration in environments with continuous action-spaces. If the scope of the action-space is too large, simply sampling from the space may lead to impractical exploration. Two consecutive actions could be quite far apart and result in unusable agent behaviour, of course not necessarily. One way to incorporate exploration in a controlled manner is to add noise to the predicted actions during training, e.g. Gaussian noise [28].

Besides noise in the action-space, noise could be added at various other points to ensure the stochastic behaviour of the agent. In one paper, noise is added to the agents' parameters, e.g. observations, to improve their exploratory behaviour [28]. In another paper, the authors add noise to the weights of a neural network to aid efficient exploration [29].

4.2 Targeted Exploration

In targeted exploration, previous states, rewards or actions of the agent influences its exploration.

4.2.1 Upper Confidence Bounds

In Upper Confidence Bounds (UCB), the agent measures the potential of high valued action. The UCB is given as $\hat{U}_t(a)$ in time step t and maps an inverse dependence for action and the amount of execution of that action so far. If an agent executes an action often, the agent does not put extra weight in exploring that specific action, but the weight increases if the agent rarely chooses that action [27]. The action could be chosen by

$$a_t = \arg\max_{a \in \mathcal{A}} \hat{Q}_t(a) + \hat{U}_t(a). \tag{3}$$

One prominent example of UCB is by Auer et al. with 'UCB1':

$$\hat{U}_t(a) = c\sqrt{\frac{\ln t}{N_t(a)}}. \tag{4}$$

Here $c > 0$ controls the degree of exploration [30]. Another examples of UCB in RL can be found by Azar with 'UCBVI' [31]. Here bonuses for the Q-function are computed and inserted in the Q function update. The authors introduce two possible approaches to calculate the bonus b, elaborate equations proportional to the horizon-length, the length of the episode and the size of action- and state-space and inverse proportional to $N(s, a)$ the number of visits of state-action pairs.

Another example in RL is by Chen et al., which builds on top of multiple Q-functions [32]. In its essence, the agent has access to an assemble of k Q-functions and creates an empirical mean $\mu(s_t, a)$ and standard deviation $\sigma(s_t, a)$ over all $\{Q_k(s_t, a)\}_{k=1}^{K}$. The agent then chooses its action that maximizes this UCB

$$\arg\max_{a}\{\mu(s_t, a) + c\sigma(s_t, a)\} \tag{5}$$

c, similar as in equation 4, controlling the degree of exploration. The diversity of Q values results in a more significant standard deviation and thus in a greater bonus for choosing an action during exploration.

4.2.2 Entropy

In information theory, the entropy of a random variable is the amount of information the variable bears. Entropy can be used in various places to guide exploration, e.g. on stochastic policy parameters [33].

One prominent example is given with the Soft-Actor-Critic method [34, 35, 36]. Here the objective function, maximizing the cumulative reward, is extended with the entropy of the policy. In exploration, this means we favour the action that is the most unpredictable. The entropy of the policy is multiplied with a coefficient α called the temperature parameter. It weighs the relevance of the entropy versus the reward in the objective function.

4.2.3 Intrinsic Rewards

If the environment has too sparse rewards, it might help to augment rewards for the agent. This could be compared with intrinsic motivation in psychology, where curiosity offers an internal drive for action [37].

One strategy might be to track all states, like in Bellemare's et al. paper on count-based exploration [38]. The idea is to use density models to estimate pseudo-counts of states and use these counts to design the intrinsic reward. Another method is to hash the states into lower-dimensional cells and keep track of a visit count that way [39]. A method in the realm of intrinsic rewards is presented in 'Exploration by Random Network Distillation' by Burda & Edwards et al.: Here, the prediction error between a static and a trained neural network is used to reward the agent during exploration [40]. Both networks are provided with the state and give some feature representation as output. The training network tries to learn from the difference of the outputs and approximates the static network. The difference between the outputs is also the intrinsic reward the agent gets. Known states have a slight deviation as the network has already seen them; new states cause a more significant deviation. Thus, the static neural net produces a novel feature representation the training network probably fails to estimate.

Another similar approach to indirectly counting states is presented by Machado et al. with the Count-Based Exploration with Successor Representation. In this method, features or state visitations are counted to guide the exploration process [41].

5 Exploration with Temporal Extensions

There have been several successful developments of exploration through action repetition in Reinforcement Learning.

5.1 Options

The basis for the temporal abstraction was laid out in 1998 by Sutton et al. with the notion of *options* [27] . The main idea behind options is that a system might need different timescales to be controlled, e.g. when flying a helicopter. On the one hand, there are defined manoeuvres in brief periods, e.g. adjusting the vehicle's direction. Those are considered on very short timescales, with attention on every muscle twitch. On the other hand, there are tasks like *flying across the continent*, which would be unwise to plan at the level of muscle twitches.

Options combine these levels of comprehensiveness in one MDP. In a general abstraction, options can depend on the trajectory between time step t and the initialized time step $t+k$. This trajectory is called a *history* with $h_{t:t+k} \equiv s_t a_t s_{t+1} \dots s_{t+k-1} a_{t+k}$, and $\mathcal{H}$ as space of all possible histories. An option is a tuple of $\omega \equiv \{\mathcal{O}_\omega, \pi_\omega, \beta_\omega\}$. Here $\mathcal{O} \subset \mathcal{S}$ is the space of all possible states from which an agent can initialize an option. An option depends on its history since initiation. The policy π_ω assigns some probability to each action conditioned on a given history. β_ω is the probability that the option terminates after an observed history h. Thus, when an option is selected, the agent takes actions after observing h, $a \sim \pi_\omega(\,\cdot \mid h)$ terminating after each step with probability $\beta_\omega(h)$. This leads us to a point where the initial definition of Markov doesn't hold: the action is chosen by the policy π_ω in state s_τ that is not merely dependent on s_τ but also on the option being followed. The option subsequently depends on the entire history h_t since initiation. The extension of an MDP with a semi-Markov option thus forms a semi-MDP. There is a lot of literature on discovering options [42, 43, 44, 45] and exploration through options [46, 47, 48, 49], though always in the context of abstracting an MDP into a semi-MDP.

5.2 ϵz-greedy

In the ϵ-greedy algorithm, the agent chooses the optimal action with a probability ϵ of taking an exploratory action. Dabney et al. modified ϵ-greedy by using the probability of ϵ to repeat the exploratory action for a random amount of n time steps [50]. This extension of ϵ-greedy is referred to as ϵz-greedy, which get its theoretical backbone from options:

With an ϵ probability, an option ω_n is taken. Within this option, an action is chosen randomly and repeated for n-times. n is obtained by sampling from some distribution. The authors empirically found the best distribution to sample from is the zeta distribution with the distributional parameter $\mu = 2$.

To test ϵz-greedy, the authors first examined small-scale environments like Grid World, DeepSea, and Mountain Car. As a baseline within these environ-

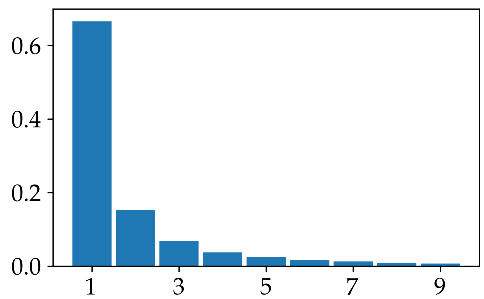

Figure 3: Zipf Distribution cut off at 10

ments, primarily tabular function approximators with Q-learning were considered.

The results show that through ϵz-greedy exploration, the agent gets more state visits than regular exploration. As a result of this improved exploration, the agent beats the regular agent in all test environments: the ϵz-greedy agent is not faster but achieves better performance.

With successful results in the tabular setting, the authors also extended the method to the deep RL setting. They implemented ϵz-greedy for two agents from literature, Rainbow [51] and R2D2 [52], to solve games from the arcade learning environment. Within these tests, ϵz-greedy is compared to: exploration by Random Network Distillation (RND) by Burda et al. [40], exploration by bootstrapping, inspired by Osband et al.[53], and the intrinsic motivation-based exploration algorithm with CTS-based pseudo counts from Bellemare et al. [38]. The results show an increase in the mean performance of ϵz-greedy compared to the other exploration strategies.

To recap: ϵz-greedy relies on the random repetition of actions to support the exploration process of the agent. One way to expand on this idea is to introduce learning of the repetition of action, as TempoRL does.

5.3 TempoRL

In TempoRL, skip-connections for MDP's was introduced [54]. In this framework, an action is executed over multiple consecutive states, which leads to a faster propagation of information about expected future rewards.

Skip-connections are feasible through the use of contextual information in the MDP. Similar to options, skip-connections c are a triple $\langle s, a, j \rangle$: s as the starting state for a skip transition, a as the action that is repeated during the skips, and j as the total length of the skip transition with $s \in \mathcal{S}$, $a \in \mathcal{A}$ and $j \in \{1, ..., J\}$. With the action repeated j-times and the state after the action-repetition as s' the skip-transition function follows:

$$\mathcal{T}_j(s,a,s') = \begin{cases} \prod_{k=0}^{j} \mathcal{T}^a_{s_k s_{k+1}} & \text{if reachable} \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Similar to the transition function, the reward function is changed to:

$$\mathcal{R}_j(s,a,s') = \begin{cases} \sum_{k=0}^{j} \gamma^k \mathcal{R}^a_{s_k s_{k+1}} & \text{if reachable} \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

It is worth noticing that those definitions of rewards and transitions are - with a skip length of 1 - no different from the definitions of regular MDPs.

The authors propose to learn the skip length followed by the action in a learning hierarchy. First, a behaviour policy determines an action a for the current state. Then, in a second step, a skip policy prescribes how long the agent should act upon an action, which is done through the lens of classical Q-learning. The Q-function,

$$Q^\pi(s_t,a) \coloneqq \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s = s_t, a\right], \tag{8}$$

can be extended to n-step Q-learning, without changing the action at each step, to:

$$Q^{\pi J}(s,j|a) \coloneqq \mathbb{E}\left[\sum_{k=0}^{j-1} \gamma^k r_{t+k} + \gamma^j Q^\pi(s_{t+j}, a_{t+1}) | s = s_t, a, j\right]. \tag{9}$$

When the action a and skip-length j are determined, the standard temporal difference update for the action and the overarching skip-observation can be made.

The authors tested the method TempoRL in tabular and deep settings. For tabular RL, the authors examined the Gridworld problem *cliff* in different settings: the agent has to walk beside a cliff to some goal state.

Here the agent receives a positive reward for reaching the goal and a negative reward for going over the cliff. The TempoRL agent was compared to standard DQN, with different ϵ-schedules for ϵ-greedy under consideration. TempoRL invariably outperforms the agent in this environment by learning faster and needing fewer decisions than the classic DQN agent.

In the deep RL setting, the authors compared TempoRL against other skip Q-function architectures, *dynamic actions repetition* [55] and *Fine-grained action repetition* [56]. The agents were compared in classic gym environments with different maximal skip lengths and different architectures. In environments with continuous action-spaces (Pendulum), the temporally extended agents were worse than the standard implementation. In the environments with discrete action-space (MountainCar, LunarLander), TempoRL beat the other temporally extended agents and the agent's standard implementation. When tested on Atari games, the TempoRL agent also performs better than the other agents, although it usually took more time to learn than the normal DQN.

When analyzing the learned policies, the focus is on the skip-decision the agent takes in certain states. Analyzing the skips in the tabular setting of Cliff, the agent chooses only to make decisions at crucial stages of the environment: when starting, when facing a cliff and readjusting the direction, and when heading towards a goal. In MountainCar, we can observe the TempoRL agent chooses large skips in states where the car needs to accelerate in one specific direction.

5.4 Not-normal Noise

As previously mentioned, continuous action-spaces may require exploration strategies, unlike their discrete counterparts. This also included particular temporally extended exploration strategies. Imagine a robot where the action-space relates to the movement of some body part; the action could be some exerting force a opposed on the object. The position of the object being x the formula for motion is written as $m\ddot{x} = a$. If we change the formula for x, we integrate over our action, thus smoothing the operation. With Gaussian noise, we would get the centre of the bell curve (usually 0) as the approximate result, thus moving the object around 0, not making any progress [57]. Wawrzyński made the first case regarding this issue in 2015 by considering autocorrelated noise for consecutive actions. Here the author showed that correlated noise reduced the shaking of robots and more guided exploration [58]. The author continued the ideas from this paper in a subsequent paper with Szulc and Łyskawa in 2020 [59]. They test the idea with a new agent called ACERAC (**A**ctor-**C**ritic with **E**xperience **R**eplay and **A**utocorrelated a**C**tions) and get better results than state of the art agents in some gym environments.

Another strategy that presents quasi temporal-extension to the action-space is presented by Lillicrap et al. [60]. Here, an Ornstein-Uhlenbeck process generates temporally correlated noise that persists longer in a particular direction and ultimately does not cancel itself out. It is worth mentioning that follow-ups to the DDPG do not find significant improvements of Ornstein-Uhlenbeck noise compared to simple Gaussian Noise, e.g. in TD3 [61].

5.5 Other Forms of Temporal Extended RL

The previous examples focus on the temporally extended aspect of action-repetition, which can be added to existing architectures. Now we focus on methods that introduce their algorithmic focus on temporal extension and not necessarily on exploration.

5.5.1 Go-Explore

A new approach to exploration was presented in the paper "First return, then explore" [62]. The central premise is that exploration strategies currently suffer from detachment and derailing: (1) algorithms forget how to return to previously reached states and (2) fail to first return to some previously visited states and continue to explore from it. Under these considerations, the authors introduce Go-Explore. The Go-Explore agent keeps an archive of all visited states in the environment with additional information of the action trajectory, the reward and the number of visits. This archive starts with the initial state of the environment and is built by the agent iteratively. Once terminated, the agent uses some heuristic to decide which state to visit next, e.g. by revisiting rarely visited states. During training, the archive is persistently updated with new states and information if already visited states can be reached, e.g. with fewer steps or higher scores. To reduce the computational effort, the states in the archive are stored in a simplified form. In the second part of the method, the best trajectory is robustified through imitation learning. The authors used Salimans and Chen's method of imitation learning as the basis for the algorithm [63]. However, they modified the approach by supporting multiple demonstrations to learn. The premise of the algorithm is to start the imitation learning near the end of the trajectory and increasingly extend the length the agent has to play. To add some stochasticity, the authors add "sticky actions" in the robustification phase. This is very similar to ϵz-greedy: After the initial time step of the environment, the agent repeats with 25 % the latest executed action, again.

5.5.2 Expected Eligibility Traces

Eligibility Traces are used to make online updates based on past experiences:

$$\Delta w_t \equiv \alpha \delta_t e_t, \qquad \text{with} \qquad e_t = \gamma_t \lambda e_{t-1} + \nabla_w u_w(S_t), \tag{10}$$

where α is the learning rate, δ_t the temporal-difference (see chapter 2.1) and e_t, the accumulated eligibility trace [27]. The eligibility trace references itself and represents a trajectory of previous transitions. $\lambda \in [0, 1]$ controls the depth of the trace. Note that $\lambda = 0$ corresponds to the regular temporal difference update, see chapter 2.1.

With regular eligibility traces the updates happen with an existing trace $e(\tau_{0:t})$ in some trajectory τ, as in equation 10. With expected eligibility traces, the updates happen through $z_\theta(S_t)$, the expected traces, which is a learned approximation with parameters $\theta \in \mathcal{R}^d$ [64]. Then the weight update follows as $\Delta w_t \equiv \alpha \delta_t z_\theta(S_t)$. z_θ is updated, e.g. by minimizing the loss between z and e for every component.

There are some possible interpretations of expected traces. For one, the updates on traces can be read as counterfactual updating on possible trajectories preceding the current state. Expected traces can also be interpreted as predecessors of the current state, thus enabling an extended temporal view on action trajectories.

5.5.3 Sequence Modeling

Transformers are used to translate a sequence of characters into another sequence of characters. This has been especially useful in neutral language processing (NLP), as many of the most advanced engines are based on transformers, e.g. the GPT architectures by OpenAI [65, 66]. Recently there have been some approaches to using transformers in the scope of RL [67]. Here the problem of RL is represented as a conditional sequence modelling problem, called Decision Transformer: actions, states and rewards are subsequently fed into the transformer, which tries to predict actions autoregressively.

6 Implementation

Now we shall test the temporally extended methods. We consider a double DQN (DDQN) agent for discrete environments, and for the continuous environments, we choose a Soft-Actor-Critic. All of these are based on Q-learning, which will be covered first. The code is available at:
`https://github.com/automl-private/TempoDAC/`

6.1 Q-learning

Q-learning is the process of finding the Q-function, the expected future reward of state-action pairs:

$$Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}. \tag{11}$$

With a perfect function of Q, the best action to take is the action that maximizes Q given the state s. Note that actions are discrete values.

The basis for updating the Q-function is the Bellman equation, which is constructed as an iterative value update to Q:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \tag{12}$$

The elements of the function can be considered individually. $Q^{new}(s_t, a_t)$ expresses the updated Q-function. $\max_a Q(s_{t+1}, a)$ is the estimate of the optimal future value in the next step: the subsequent state is set, and we choose the action a that maximizes Q from this point forward. γ discounts this possible future reward with values in the far future becoming increasingly smaller. Therefore the value of γ is usually slightly below 1. Together with the reward received in the next state we get $R_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ as the temporal difference target. The difference between this value and the estimate of the future reward in the state s is the temporal difference, sometimes referenced as the temporal difference error (TD error). α carries how greatly the Q-function should be updated on the temporal difference error. For this reason, it is called the learning rate [27].

Standard Deep Q-networks (DQN) was introduced by Mnih et al. and established using a non-linear function approximator for the Q-function in the form of a neural network [68]. The Q-function is parameterized by the neural network's weights θ, which is noted as $Q(s, a; \theta)$.

The authors further proposed using two separate networks when calculating the loss during the update of the Q-function:

$$R_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t^-) - Q(s_t, a_t; \theta_t). \tag{13}$$

The values of the θ-network are adjusted each updated; the values of the θ^--network are copied from the θ-network every C step. The delayed update on the second neural network makes the algorithm more stable because it hinders oscillation or divergence due to the similarity of consecutive states. They also introduce the replay buffer: states, actions, following states, rewards and termination flags are stored and sampled during learning. The algorithm learning becomes more stable with a replay buffer because the TD-update can execute in batches across a wide range of different experiences.

DDQN considers another aspect of value assigning and action selection[69]. In the calculation for the target in regular DQN, the Q-function does both: selecting the most promising action and assigning its value. This can lead to an overestimation of actions and thus reduce the quality of the agent. DDQN tries to prevent this by relocating the value assignment to yet another Q-function:

$$R_{t+1} + \gamma Q(s_{t+1}, \text{argmax}_a Q(s_{t+1}, a; \theta_t); \theta'_t) - Q(s_t, a_t; \theta_t). \tag{14}$$

Here the action selection weights are θ_t, and the weights for the policy Q-function is parameterized by θ'_t. The weights of the Q-functions are updated by evenly switching the roles of θ and θ'. It should be noted that only using 2 Q-functions suffices for implementing DDQN: One static for calculating the TD target and the other for getting the action within the TD target and getting the current estimate of the Q-value.

We choose DDQN as a baseline for FastDownward because it strikes as both strong enough to achieve excellent performance on its own but still sufficiently malleable to include temporally extended methods.

6.2 Policy Gradient Methods

Another mechanism in RL that builds on top of MDPs are Policy Gradient Methods, which present a natural extension to TD learning [27]. The central premise of policy gradient methods is to model the policy under some parameterization θ directly. While some methods exclusively learn the policy, like REINFORCE [70], a significant improvement is to use a separate function that learns to evaluate state action pairs prompted by the policy. These algorithms are called actor-critic methods: a policy structure that selects the actions, the actor, and an additional value function that evaluates the actor, the critic. The vanilla policy gradient algorithm represents a probability distribution from which an action a in state s is selected. Note that actions are obtained from a stochastic distribution and thus can be continuous in contrast to Q-learning.

One example of policy gradient is the Soft-Actor-Critic (SAC) algorithm [34, 35, 36]. As discussed in chapter 4, the SAC uses entropy regularization: the objective of getting the most reward is weighed with the entropy of the current policy. Also, SAC learns two Q-functions for the critic. During updates, SAC uses the Q-function with the smaller estimates, as overestimation poses an issue in Q-functions.

The policy in SAC is stochastic, which means we have a probability distribution over all actions in each state and not a clearly defined action. So when optimizing the policy in SAC, we make use of the "reparameterization trick". While learning the policy, we want to maximize $\mathbb{E}_{a\sim\pi}[Q^\pi(s,a)] + \alpha H(\pi(\cdot|s))$, with the assurance that sample from the policy are differentiable. In order to assure this, when sampling for the action, we apply a tanh over a squashed Gaussian policy $\mu_\theta(s) + \sigma_\theta(s) \cdot \xi$, with ξ as independent noise. The tanh binds the sampled action in a finite range. The σ and μ are both depended on the weights θ and not the actions and thus we can write the expected value for the policy loss as $\mathbb{E}_{\xi \sim \mathbb{N}}[Q^{\pi_\theta}(s, \tilde{a}_\theta(s, \xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s, \xi)|s)]$. When evaluating the agent, we only chose the mean action.

We choose SAC as a baseline for environments with continuous action-spaces. Another possibility is to use TD3, which shares some similarities with SAC. The choice falls on SAC because we slightly favour the entropy-based exploration in DAC. During an initial test with a deep deterministic policy gradient agent, we could see that the agent favoured using either the highest or lowest action - a failure mode that could be remedied through the entropy of policy.

6.3 Design Choices

6.3.1 Baseline

First, some considerations for the neural network architecture of the Q-function in the DDQN: we choose a fully connected network with three hidden layers á 64 units. Since FastDownward includes a wide variety of state-spaces, the main idea is to use a more extensive network, even if it means more extended training. For the replay buffer, we choose a simple replay buffer, which provides random samples during updates of the Q-function. The exploration strategy in the experiments was the ϵ-greedy linear decay method, which we discussed in chapter 4. After some initial steps, the agent starts updating Q according to the DDQN update described in the equation 14. Every n step, the agent is evaluated in the environment without exploration to measure the progress.

For the SAC, we choose a two-layer neural network with 256 hidden units, similar to the proposed network in the original paper [34]. The exploration mechanism during training in the standard case is based on noise. After some initial time steps of a purely random sampling of the action-space, the action provided by the actor is multiplied with Gaussian noise. In addition to the noise, the actor's exploration is also guided by the entropy of the policy, as described in chapter 4. During the evaluation, the mean of the policy distribution in the state is chosen. Therefore, no noise is added to the action.

6.3.2 Design Choices: ϵz-greedy

ϵz-greedy, as discussed in chapter 5, is incorporated into the DDQN agent as well as in SAC. In DDQN, the ϵ-greedy linear decay is extended to include sampling from the zeta-distribution. Because the zeta-distribution is heavy-tailed, but the environment is not very long, we cut off any value sampled over 10 for FastDownward and cutoff any value higher than 3 in ToySGD and cSigmoid. Again being without exploration, the evaluation does not differ from the evaluation for the basic DDQN.

In SAC, the primary exploration mechanism is not based on ϵ-greedy, so we introduce ϵ as a variable to guide the action repetition process during training.

6.3.3 Design Choices: TempoRL

In DDQN, we implement a second Q network for learning the skip length. For this, we consider a fully connected two hidden layer neural network á 64 units. As discussed in chapter 5, the update on the Q-function are similar to the updates in regular DDQN. The update for the skip Q-function is on the Smooth L1 loss between the target Q-values and Q-values of the skip function.

For SAC, the ϵ variable guides the skip exploration like in ϵz-greedy. The difference to the DDQN implementation is that we can also use the twin Q network and choose the network that does not overestimate Q values.

6.4 Hyperparameter Configuration

With the neural network architecture set, we can focus on the settings of the hyperparameters of each agent depending on the environment. For tuning the HP, we considered an automatic method for finding the best HP: SMAC, which stands for *sequential model-based algorithm configuration* [9]. With SMAC, we configure the HP: batch size, the number of time steps in the ϵ-decay, amount of gradient-clipping, the size of the replay-buffer, the learning rate, and ϵ of the ADAM optimizer. We choose these HP because they are crucial to successful learning and also greatly influence exploration. In FastDownward, the agents are tested on 1.2 million time steps. A previous test showed that any duration shorter than that included the possibility of momentary overfitting at the beginning of the test. In cSigmoid, as in ToySGD, the agents run for 100k time steps. In SMAC, we differentiate between the learning rate and ϵ_{Adam} for the actor and critic. More details of the SMAC test setup and results can be found in the appendix A.

7 Experiments

With the implementation from the previous chapter, we conduct experiments to answer the question "Is it worth including temporal extensions when running agents in DAC?". To answer this question, we will discuss performance differences of the agents and examine the resulting policies. For the experiments we run the agents over 5 different seeds on FastDownward for 1.2×10^6 training time steps and ToySGD and continuous Sigmoid for 5×10^5 training time steps. More training details can be found in appendix A.

7.1 Which exploration strategy works best for DAC?

First, we are going to discuss the results of the individual agents in FastDownward.

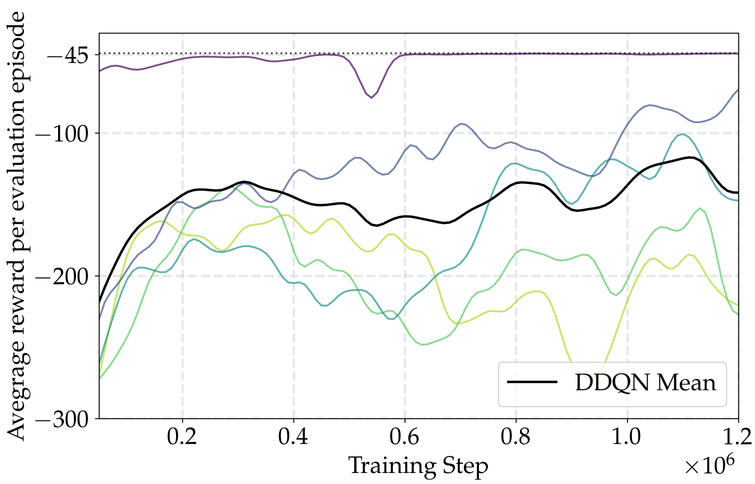

Figure 4: Performance of DDQN in FastDownward, smoothed with a Gaussian Filter with $\sigma = 2$

In figure 4 we see that the performance of the regular DDQN. The x-axis shows the training steps of the agents within the environment. The corresponding values on the y-axis are the average performance evaluated over all instances. Here, the traces show that the performance of the run of one seed significantly outperforms the runs of the other seeds. The best run starts with a policy close to the optimal policy at around -46 and constantly holds its performance. While two other runs show signs of performance gain, two others fail to get any better over their run. This difference indicates great instability of the performance of the algorithms on FastDownward. On average, the DDQN starts at -200 points and only very slowly and unstably improves to around -120 points, which is still 75 points away from the optimal policy.

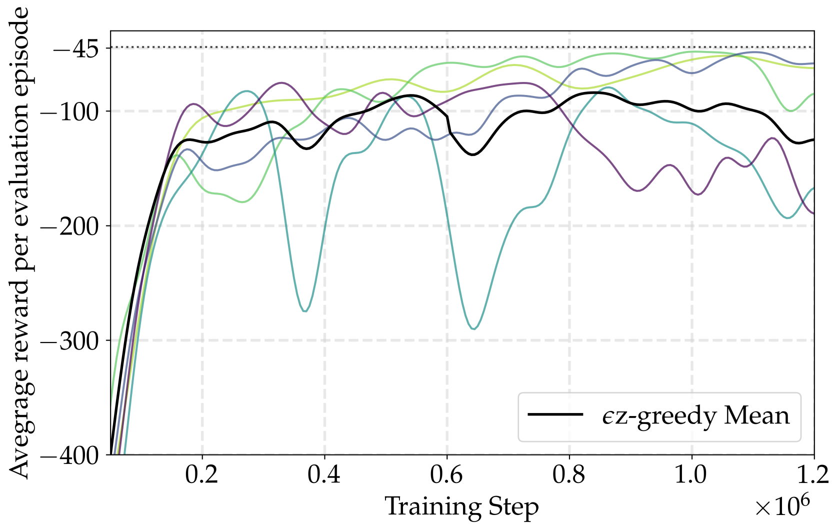

Figure 5: Performance of the ϵz-greedy DDQN in FastDownward, smoothed with a Gaussian Filter with $\sigma = 4$

The performance of the DDQN with ϵz-greedy in figure 5 looks more promising. Although the runs start much worse than regular DDQN, three runs approach the optimal policy closely, on average only 20 points missing. However, the other two struggle to perform, again showing signs of inability to learn anything valuable. On average, the performance differs from the optimum around 50 point.

Lastly, the TempoRL version of the DDQN in figure 6 shows much greater success. The performance starts at 30 points worse than regular DDQN, but ultimately all runs improve generously and approach the optimal policy with only 5 points missing on average.

In figure 7, all the agents are compared according to their mean and standard deviation. The agents with temporal extension improved their performance over their vanilla predecessors by an average of at least 30 points.

This ranks TempoRL as the best performing, ϵz-greedy as a strong but unstable successor and regular DDQN as the worst method for FastDownward. It is worth noting that ϵz-greedy has a very steep improvement curve, requiring only half as many time steps as the TempoRL agent to reach an average of -120 reward.

While this shows how good action repetition for DAC is, we should gain more insight into the success of temporally extended agents.

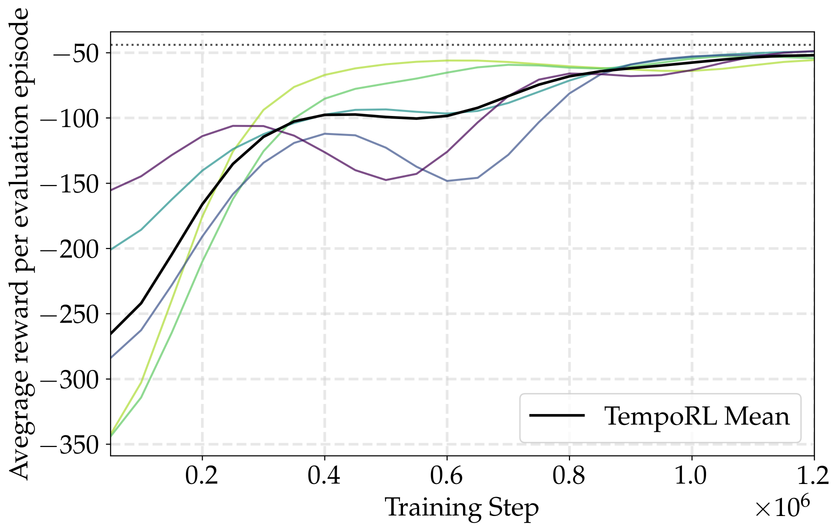

Figure 6: Performance of the TempoRL DDQN agent, smoothed with a Gaussian Filter with $\sigma = 2$

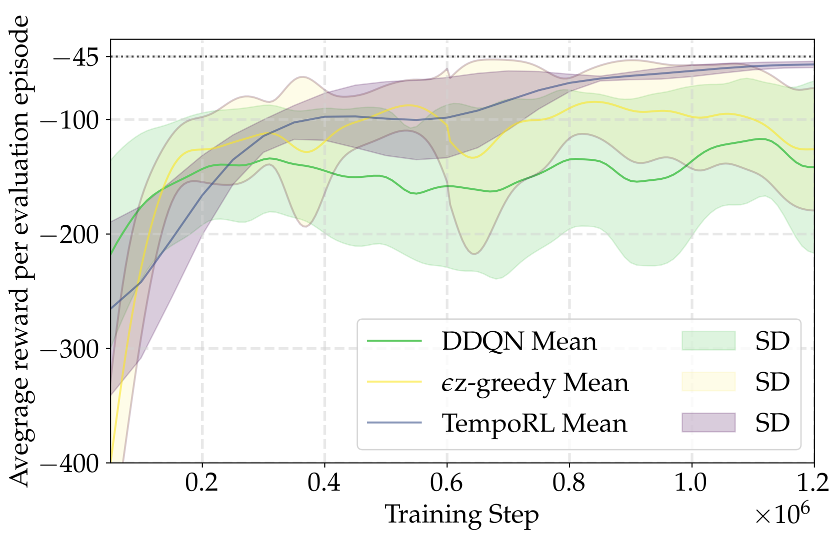

Figure 7: Mean performance between agents in FastDownward with standard deviation

7.2 Does the TempoRL agent need fewer decisions?

The TempoRL agent is capable of making decisions to perform actions for multiple time steps, which significantly affects its performance. We now examine its use during evaluation.

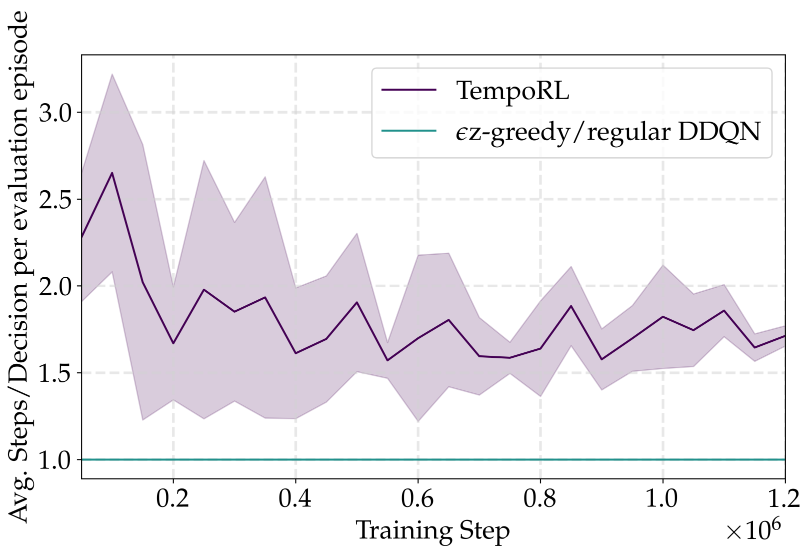

Figure 8: Mean steps per decision with standard deviation of the agents

In figure 8 we can see the average step in the evaluation episode normalized on the decision. To no surprise is this a constant 1 for ϵz-greedy and regular DDQN. More interestingly is the curve from the TempoRL agent. In the beginning, the agent makes around 2.5 steps per decision and, after about 0.4×10^6 steps, resolves to 1.6 steps per decision on average. This indicates that learned action repetition is helpful for DAC, as the agent robustly uses it and the performance increases.

7.3 Do temporally extended agents focus on action repetition?

Another question is: does ϵz-greedy or TempoRL favour action repetition compared to standard DDQN. One simple metric we observe to weigh this is the number of action switches the agent performs during evaluation.

In figure 9 we evaluate this with representative runs of all three agents. In the beginning, we can see that the TempoRL agent makes slightly fewer and the ϵz-greedy agent slightly more action switches than standard DDQN. Interestingly, the temporal extended agents steadily lower their need for action repetition and slowly decrease the switching over the run to the optimum of around 5 switches during evaluation. The regular DDQN agent also decreases

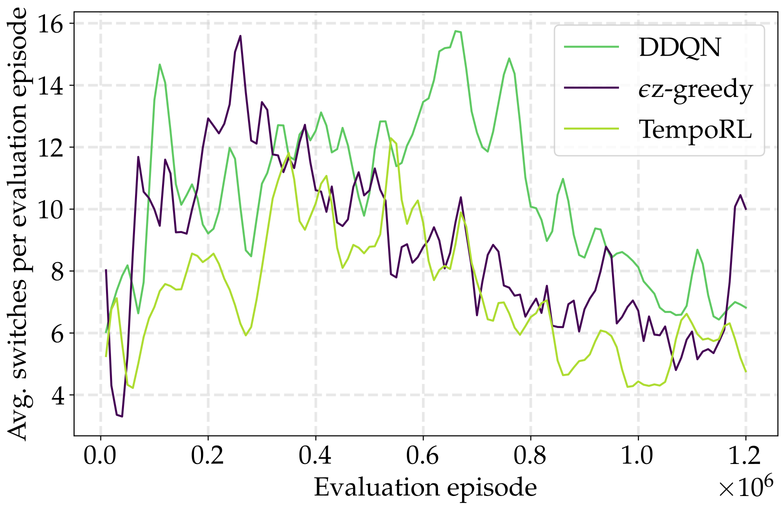

Figure 9: Comparison of the number of switches the agent makes in the evaluation episodes.

its action switches, though compared to the other agents later. The increased action switching of ϵz-greedy, in the beginning, is due to its poor performance at the start of learning: if the evaluation episodes take long to finish, the agent has more time to switch actions. In conclusion, the temporally extended agents are better able to control action changes than their predecessors.

7.4 Which exploration strategy works best in continuous environments?

In this section, we first evaluate the temporally extended agents in ToySGD. To begin, we are going to see the results from the regular Soft Actor-Critic (SAC).

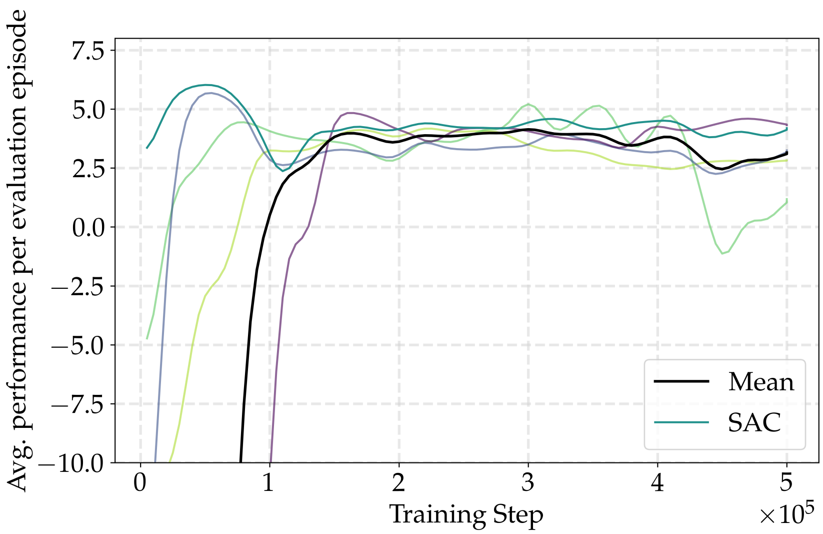

Figure 10: Performance of the regular Soft Actor-Critic, smoothed with a Gaussian Filter $\sigma = 2$

The figure 10 shows the runs over 5 different seeds. Two performance traces start with high rewards in the early stages of the run but lose the ability to hold this performance and remaining on a lower level. Other traces show typical behaviour by starting at lower performance and steadily improve, averaging at a reward of 4.

In figure 11 are the performance plots of the ϵz-greedy agent and we can see similar behaviour as with our regular SAC: three traces initially get a high reward of almost 5 but ultimately converge to lower overall performance. Some traces seem to get the same final performance as the regular SAC of 4, while others perform on average 2 points worse than that.

In figure 12 we see the performance of the TempoRL agent over 5 seeds. These runs show a more concerning performance. The agents start with a performance much worse than regular or ϵz-greedy agent, starting on average at -80. Ultimately only one of the runs holds their performance for a stable amount of time. In the other runs the performance of the agent differs around 10 points, usually oscillating around 0 reward. For a larger display of the plot, see appendix B.

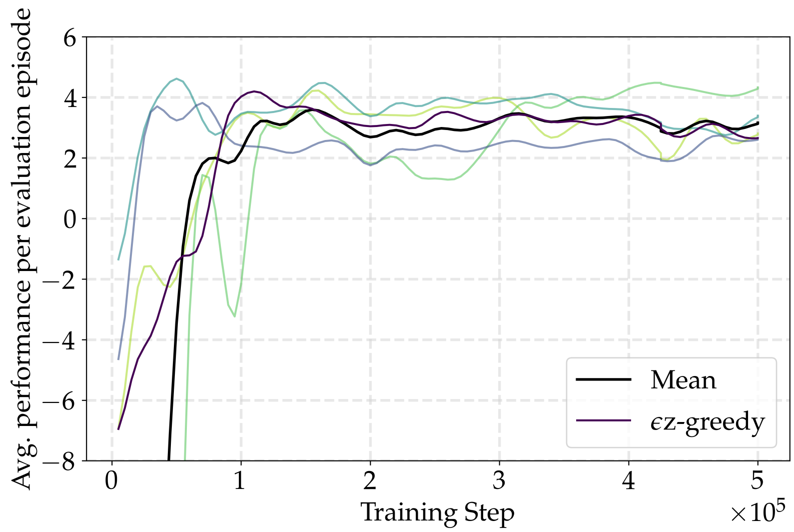

Figure 11: Performance of the ϵz-greedy SAC smoothed with a Gaussian Filter $\sigma = 2$

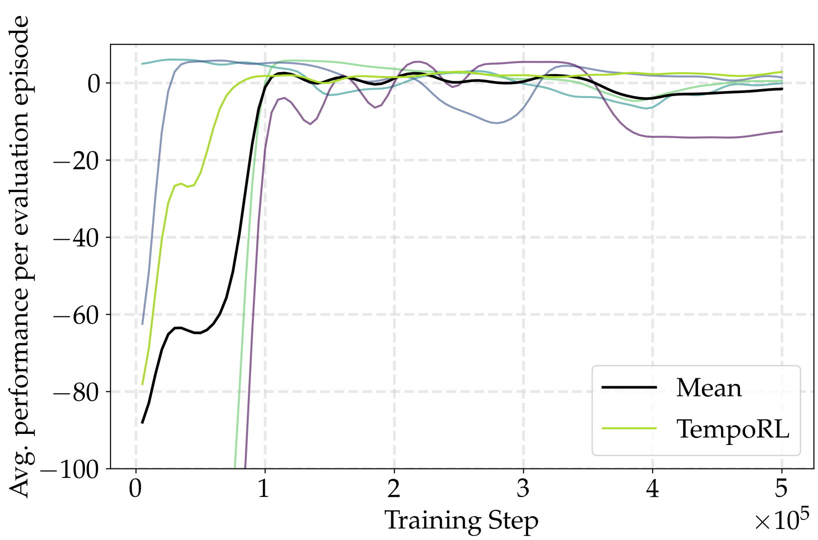

Figure 12: Performance of the TempoRL SAC, smoothed with a Gaussian Filter $\sigma = 2$

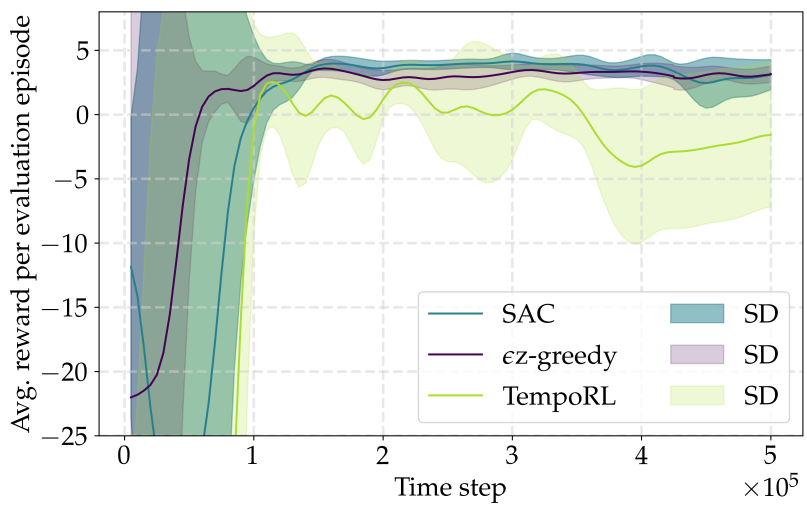

Figure 13: Mean performance in ToySGD over 5 seeds with standard deviation

In figure 13 we see the mean performance of the different agents. The regular SAC needs on average 1×10^5 steps to find a policy that yields positive reward and hold its performance for the remainder of the run at around 4. The ϵz-greedy agent trumps the speed of the standard SAC and achieves positive reward after only 0.6×10^5 training steps, but performs slightly worse with an average reward of around 3 for the remainder of the run. The TempoRL agent has difficulties. The performance starts off worse than regular and ϵz-greedy SAC and fails to converge to one value.

The results of the different agents show that while there are successes in some runs, we need to explore why sometimes the agent fails to regain their initial performance. Then we will focus on the runs where the agent performs well and if this good performance is due to action repetition.

7.4.1 Instance specific Analysis of ToySGD

Let us analyze why performance briefly increases after a few training steps but ultimately drops off again. For this, we will analyze the labelled trace from the figure 10. We are interested in how the agent improves over the instances, to gain some insight of the overall performance. For this we plot the reward of the single instances at an early step. In figure 14 on the left side we observe the reward of all instances at once and sort them by the amount of reward received.

If we take the difference between the reward per instance in a time step at the beginning and one where the performance converges, we can measure the differences and observe the respective learning process. Figure 14 on the right displays this change.

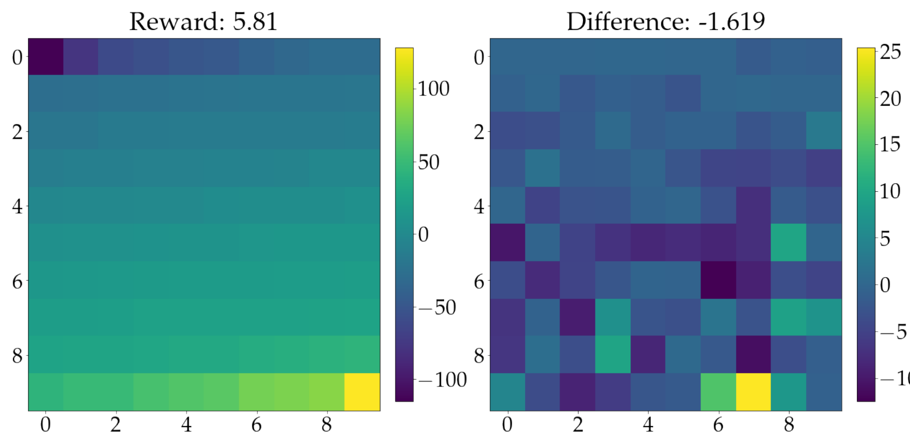

Figure 14: On the left side we see the individual reward of all instances at time step 0.3×10^5 sorted by least to most reward obtained by the SAC. On the right side is the difference in reward in each instance between time step 0.3×10^5 and 2×10^5.

We can observe that the agent is able to improve its performance on some of the initially more difficult instances (top rows) and even excelled on one that was already working well. Unfortunately, we see that the agent loses a lot of performance on some of the already well working instances, basically trading performance. This shows that the agent learns but fails to generalize over all instances.

The same experiment is done for ϵz-greedy in the figure 15 with the labeled trace from figure 11. Here the agent can outperform previously hard instances (top row) compared to regular SAC. Moreover, it holds the performance on its already top performing instances. However, it should be noted that the ϵz-greedy agent starts worse than the standard SAC. The scale might falsely indicate more significant gain and thus better overall performance. In the appendix B there are multiple snapshots of the performances during the total run for both the regular and the ϵz-greedy agent.

7.4.2 Influence of the agents on action repetition

We want to determine if action repetition influences the behaviour of the agent. Ultimately, we would not expect the regular or the ϵz-greedy agent to repeat a specific actions due to the stochastic nature of the action-space. However, it is not surprising for the agent to learn to connect actions close to each other as a result of observed action repetition. As a metric to assess the similarity of actions a within a policy we calculate the mean squared error: $MSE = 1/n \sum_{i=1}^{n}(\hat{a} - a_i)$, with $\hat{a}$ begin the mean of all action within the policy. If the actor chooses actions approximate to each other, the MSE becomes smaller. A more considerable error could occur when the actor chooses to repeat actions, but as a result continues

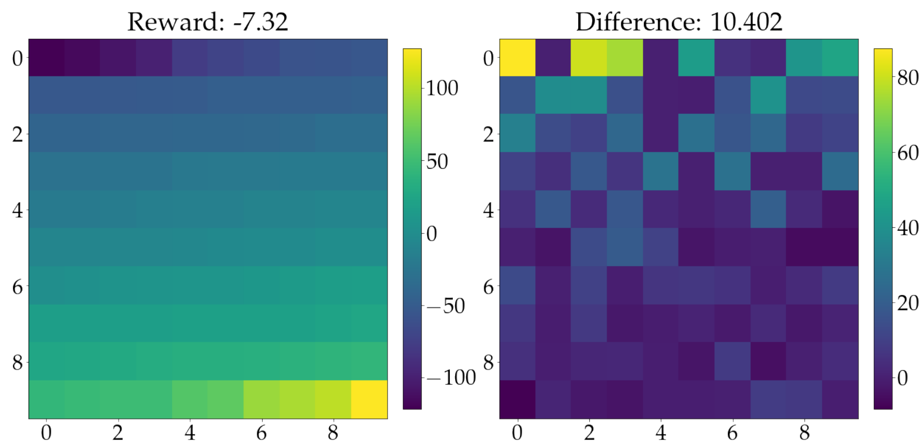

Figure 15: On the left side we see the individual reward of all instances at time step 0.3×10^5 sorted by least to most reward obtained by the ϵz-greedy agent. On the right side is the difference in reward in each instance between time step 0.3×10^5 and 2×10^5.

thereafter with an action that has a greater distance from the previous ones.

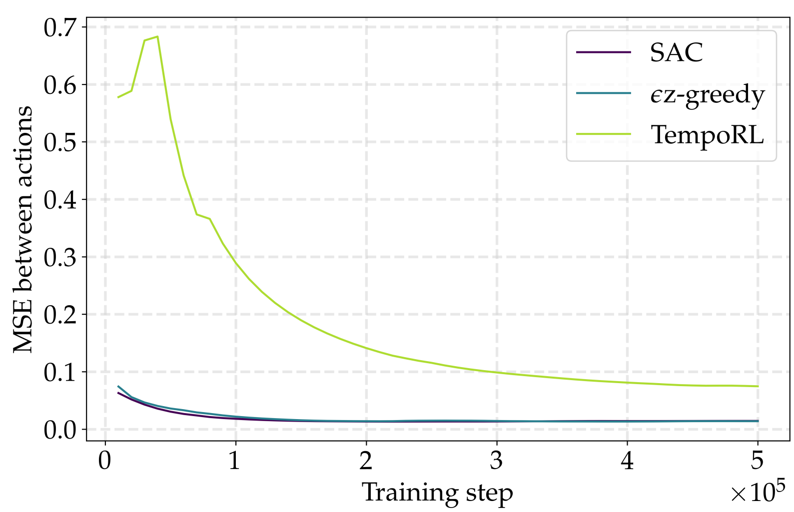

Figure 16: MSE between the actions sequence, learning rate only

In figure 16 we see the action differences for the best runs. We can see that the action-repetition process in TempoRL does not result in smaller distances: over the duration of an episode, it is more difficult for the agent to find actions that are close to each other. If we review a random policy, we observe the larger error is in fact due to larger differences between actions if the agent decides to repeats an action, see appendix 6. The trace of the regular and the ϵz-greedy agent show high similarity and consequently more promising results.

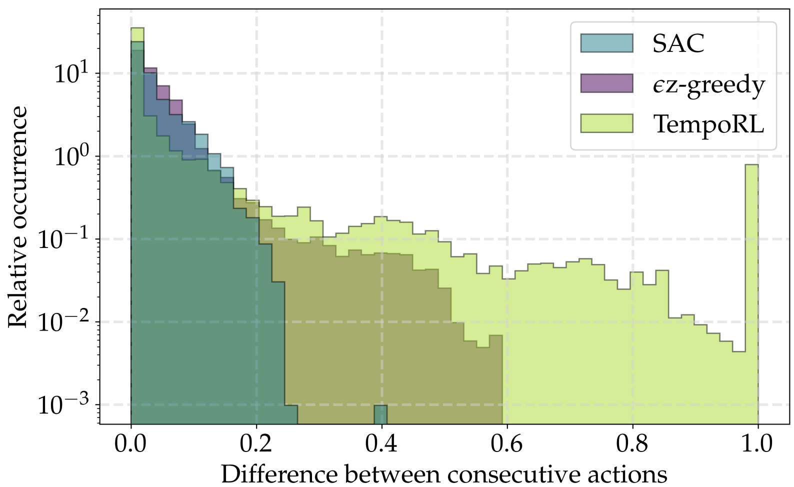

Figure 17: Histogram of the action switches of the learning rate

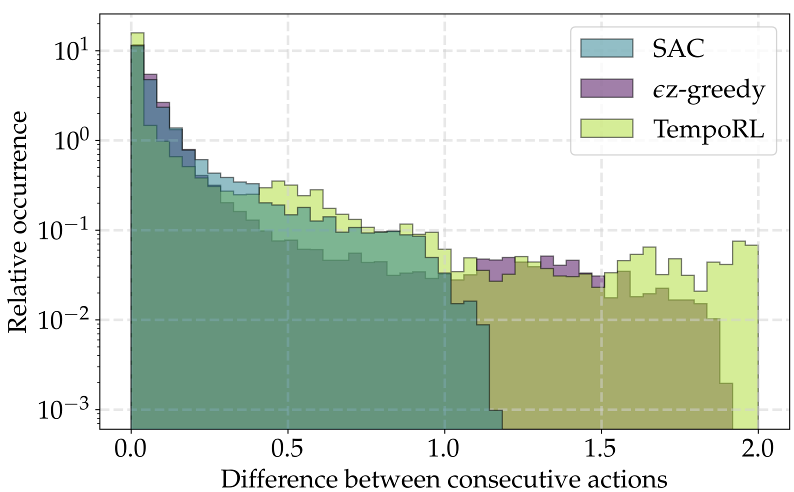

Figure 18: Histogram of the action switches of the momentum

Another metric to consider is how far subsequent actions differ from each other. To visualize this information we calculate the relative difference of successive actions and cluster them in bins.

In figure 17 and 18, we see the difference of consecutive actions on the x-axis and the relative amount of their appearance on the logarithmic y-axis. We see in both graphs that the regular SAC uses actions with closer proximity to each other: for the learning rate at most 0.4 and the momentum at most 1.2 difference between actions. The ϵz-greedy agent uses in comparison to the regular agent a broader amount of action switches. The agent uses differences between actions up to 0.6 for the learning rate and almost 2 for the momentum. Interestingly, the ϵz-greedy agent uses fewer learning rates very close to each other (the bin at 0) but slightly more in the immediately following bins. The TempoRL agent has a larger proportion of their action differences at 0, as consecutive actions are identical. Much greater is also the extend of action differences of TempoRL. For the learning rate, it goes beyond 1. A few single actions go past that but are binned in 1 for clarity of the figure. For the momentum, the action difference also reach 2 as largest difference between actions.

The main action differences TempoRL loses are those close to 0: for the learning rate differences at around 0.1 are almost half compared to the other agents. This shows that the TempoRL agent does not produce smooth action curves, which is important for success in ToySGD.

One failure mode of the temporally extended methods in ToySGD could be because the action-space is continuous. Already in the original paper for TempoRL, the authors observed the slow learning behaviour in environments with continuous action-spaces. Another reason the new methods fail in the environment could be due its short nature. 11 steps to test action repetition might offer too little opportunity.

To test this thesis, we will apply our three agents to another benchmark with continuous action-space that is approximately twice as long: continuous Sigmoid (cSigmoid).

7.4.3 Insights through cSigmoid

With a similar set-up as in ToySGD, we apply all three agents to cSigmoid and compare progress.

In figure 19 we see similar behaviour as in ToySGD: the regular SAC performances best of the other agents, the ϵz-greedy agent learning slightly faster at the start but ultimately performing worse than regular SAC. The TempoRL agent starts worse than the other two agents but slowly increases its performance throughout the run. That verified that these temporally extended methods fail to produce the same performance as their regular predecessor in environments with continuous action-space.

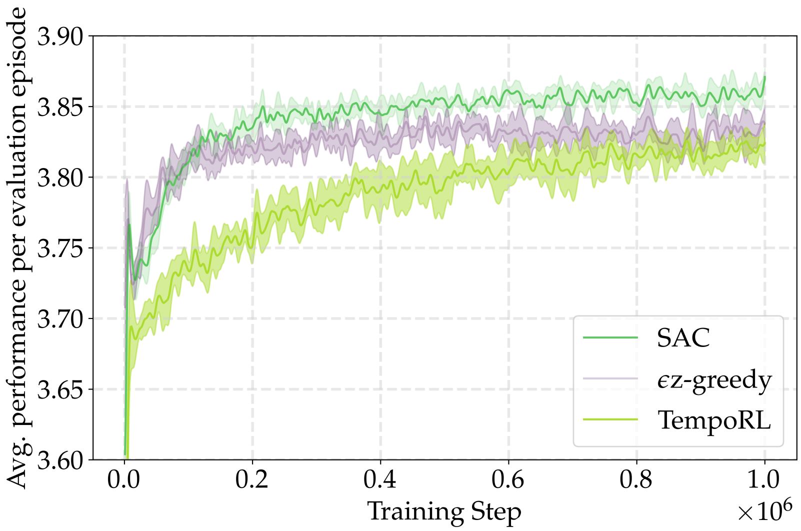

Figure 19: Mean performance in cSigmoid, and the results are smoothed with a Gaussian Filter $\sigma = 3$

8 Prospects of Temporal Extension in DAC

The methods from previous experiments of temporally extended RL show encouraging results. Now it is time to take a look into possible continuations of this work, which consists of two things: Consolidating the methods on more benchmarks and extending methods.

8.1 More Benchmarks

While the chosen benchmarks and results indicated some improvement over standard RL methods, these benchmarks are only a part of DACBench. Once more promising methods for environments with continuous action-space are available, they could be thoroughly tested on benchmarks like SGD or CMA-ES. One significant difference of these benchmarks is that the individual episodes are much longer than those in our tested benchmarks. This could significantly impact the agent, as there are more opportunities to effectively use repeated actions in an episode.

8.2 Guided Noise

The continuous benchmarks didn't thrive through temporally extended actions as their discrete pendants did. One possibility is that strict action repetition is not needed, but action repetition that allows minor variances of action. Similarly to the not-normal noise in chapter 5, we think there are some prospecting opportunities to include guided noise or structured noise as exploration in continuous action. While not the pure action repetition initially intended, it could solve the issue of insufficient information by changing the HP too often. The ideas of Wawrzyńki of using correlated noise for exploration seem beneficial. Aiding exploration through random walk type noise could lead to successful structured exploration [28]. One step in the direction made Lillicrap et al. in the DDPG paper, which discussed the Ornstein-Uhlbeck processes during exploration. One continuance could be to use random functions, but still smooth enough to hold values in some interval for some time range. Smooth random functions are discussed by, e.g. Filip et al. [71] and could be implemented in the exploration process. However, it should be noted that this approach may be too close to ϵz-greedy and could lead to the same problems of unsatisfactory performance.

8.3 Go-Explore

Go-Explore, as described in the chapter 5, could be very interesting for temporally extended RL. One facet in the imitation phase is returning to promising states and continuing the exploration process to find the best policy from that state onward. This will ultimately lead to a very fine-tuned policy and will bear great information about the most favourable sequence of actions, either including repetition of actions or not. Ultimately this approach could be considered a counterpart to the other methods, as we strategically probe all possible actions from these promising states. Of course, there are many essential aspects to consider, e.g. if we have a shared state-space across instances, do we have to create an archive for every instance?

8.4 Performance Measure

The ability to evaluate temporally extended methods depends on the overall quality of performance measurement in DAC. Suppose the difficulty of single instances differs significantly. In this case, the question might be whether the achievements in one instance outweigh the accomplishments of others, especially if they are given equal weight in the evaluation as in FastDownward. Here the optimal policy of instance 9 is 10 times larger than the optimal policy of the shortest instance. As a result, a poorly performing instance 9 means a low-performing agent overall, ignoring success in other shorter instances.

9 Conclusion

We wanted to answer the question, whether observing hyperparameters for multiple time steps improve DAC's performance. Based on this question, we studied the literature on temporally extended RL and chose two promising methods, ϵz-greedy and TempoRL. We tested the agents on three benchmarks: FastDownward with discrete action-space and ToySGD and cSigmoid with a continuous action-space. In FastDownward, the temporally extended methods beat the regular agent. ϵz-greedy earned performance gains through action repetition during exploration. The TempoRL agent not only benefited from this exploration but learned action repetition and ultimately performed best of all other agents. We showed that the methods under tests effectively learned to control actions switches more robustly than their predecessor.

The experiments with ToySGD have turned out differently: here, the temporally extended methods failed to perform as well as their antecedent. While ϵz-greedy could find a decent performing policy faster, the method was not robust enough to be the overall best contender. TempoRL was unfortunately hardly able to find a good performing policy in this benchmark. The agents' failure is likely due to its continuous action-space and the diminishing returns of repeating unique actions multiple times. Ultimately, we recommend using regular exploration for dynamic algorithm configuration with continuous action-space until new methods provide new insights. However, for environments with discrete action-space, the use of TempoRL should be considered as it showed the greatest performance compared to the other agents.

We hope that this thesis opens the door for promising future research in exploration for DAC. Future work in the space could be on the shortcomings of the agents with continuous action-space or gaining further insights when and how action repetition could guide the exploration process. Other work could be on methods like Go-Explore, which could provide knowledge about the use of action repetition. Lastly, it could be enriching to test these or new methods on various other benchmarks.

References

[1] Giorgos Karafotias, Mark Hoogendoorn, and A. E. Eiben. "Parameter Control in Evolutionary Algorithms: Trends and Challenges". In: *IEEE Transactions on Evolutionary Computation* 19.2 (2015), pp. 167–187. DOI: 10.1109/TEVC.2014.2308294.

[2] Carlos Ansótegui et al. "Reactive Dialectic Search Portfolios for MaxSAT". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI'17. San Francisco, California, USA: AAAI Press, 2017, pp. 765–772.

[3] André Biedenkapp et al. "Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework". In: *ECAI*. 2020.

[4] Jan N. van Rijn and Frank Hutter. "Hyperparameter Importance Across Datasets". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (July 2018). DOI: 10.1145/3219819.3220058. URL: http://dx.doi.org/10.1145/3219819.3220058.

[5] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. "An Efficient Approach for Assessing Hyperparameter Importance". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, 2014, pp. 754–762. URL: https://proceedings.mlr.press/v32/hutter14.html.

[6] Jan N. van Rijn and Frank Hutter. "An Empirical Study of Hyperparameter Importance Across Datasets". In: *AutoML@PKDD/ECML*. 2017.

[7] Mohamadjavad Bahmani et al. *To tune or not to tune? An Approach for Recommending Important Hyperparameters*. 2021. arXiv: 2108.13066 [cs.LG].

[8] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *CoRR* abs/1206.5533 (2012). arXiv: 1206.5533. URL: http://arxiv.org/abs/1206.5533.

[9] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. "An evaluation of sequential model-based optimization for expensive blackbox functions". In: July 2013, pp. 1209–1216. DOI: 10.1145/2464576.2501592.

[10] André Biedenkapp. *Dynamic Algorithm Configuration*. Website. Online: https://www.automl.org/dynamic-algorithm-configuration/; retrieved 03.10.2021. 2020.

[11] Benjamin Doerr and Carola Doerr. "Theory of Parameter Control for Discrete Black-Box Optimization: Provable Performance Gains Through Dynamic Parameter Choices". In: *CoRR* abs/1804.05650 (2018). arXiv: 1804.05650. URL: http://arxiv.org/abs/1804.05650.

[12] Roberto Battiti and Paolo Campigotto. "An Investigation of Reinforcement Learning for Reactive Search Optimization". In: *Autonomous Search*. Ed. by Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. Springer, 2012, pp. 131–160. DOI: 10.1007/978-3-642-21434-9_6. URL: https://doi.org/10.1007/978-3-642-21434-9%5C_6.

[13] Christian Daniel, Jonathan Taylor, and Sebastian Nowozin. "Learning Step Size Controllers for Robust Neural Network Training". In: *National Conference of the American Association for Artificial Intelligence (AAAI)*. AAAI - Association for the Advancement of Artificial Intelligence, Feb. 2016. URL: https://www.microsoft.com/en-us/research/publication/learning-step-size-controllers-for-robust-neural-network-training/.

[14] Mudita Sharma et al. "Deep Reinforcement Learning Based Parameter Control in Differential Evolution". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. Prague, Czech Republic: Association for Computing Machinery, 2019, pp. 709–717. ISBN: 9781450361118. DOI: 10.1145/3321707.3321813. URL: https://doi.org/10.1145/3321707.3321813.

[15] Yoshitaka Sakurai et al. "A Method to Control Parameters of Evolutionary Algorithms by Using Reinforcement Learning". In: *2010 Sixth International Conference on Signal-Image Technology and Internet Based Systems*. 2010, pp. 74–79. DOI: 10.1109/SITIS.2010.22.

[16] Hadi S. Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. "Hyp-RL : Hyperparameter Optimization by Reinforcement Learning". In: *CoRR* abs/1906.11527 (2019). arXiv: 1906.11527. URL: http://arxiv.org/abs/1906.11527.

[17] Assaf Hallak, Dotan Di Castro, and Shie Mannor. *Contextual Markov Decision Processes*. 2015. arXiv: 1502.02259 [stat.ML].

[18] Gresa Shala et al. "Learning Step-Size Adaptation in CMA-ES". In: *Parallel Problem Solving from Nature – PPSN XVI*. Ed. by Thomas Bäck et al. Cham: Springer International Publishing, 2020, pp. 691–706. ISBN: 978-3-030-58112-1.

[19] David Speck et al. "Learning Heuristic Selection with Dynamic Algorithm Configuration". In: *Proceedings of the International Conference on Automated Planning and Scheduling* 31.1 (May 2021), pp. 597–605. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/16008.

[20] Theresa Eimer et al. *DACBench: A Benchmark Library for Dynamic Algorithm Configuration*. 2021. arXiv: 2105.08541 [cs.AI].

[21] Diederick Vermetten et al. "Online Selection of CMA-ES Variants". In: *CoRR* abs/1904.07801 (2019). arXiv: 1904.07801. URL: http://arxiv.org/abs/1904.07801.

[22] Zhen Xu et al. "Learning an Adaptive Learning Rate Schedule". In: *CoRR* abs/1909.09712 (2019). arXiv: 1909.09712. URL: http://arxiv.org/abs/1909.09712.

[23] Diogo Almeida et al. "A Generalizable Approach to Learning Optimizers". Jan. 1, 2021. URL: `https://arxiv.org/pdf/2106.00958.pdf` (visited on 01/01/2021). published.

[24] Rainer Storn and Kenneth Price. "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces". In: *J. of Global Optimization* 11.4 (Dec. 1997), pp. 341–359. ISSN: 0925-5001. DOI: `10.1023/A:1008202821328`. URL: `https://doi.org/10.1023/A:1008202821328`.

[25] Malte Helmert. "The Fast Downward Planning System". In: *CoRR* abs/1109.6051 (2011). arXiv: `1109.6051`. URL: `http://arxiv.org/abs/1109.6051`.

[26] Lilian Weng. "Exploration Strategies in Deep Reinforcement Learning". In: *lilianweng.github.io/lil-log* (2020). URL: `https://lilianweng.github.io/lil-log/2020/06/07/exploration-strategies-in-deep-reinforcement-learning.html`.

[27] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN: 9780262352703. URL: `https://books.google.de/books?id=uWV0DwAAQBAJ`.

[28] Matthias Plappert et al. "Parameter Space Noise for Exploration". In: *CoRR* abs/1706.01905 (2017). arXiv: `1706.01905`. URL: `http://arxiv.org/abs/1706.01905`.

[29] Meire Fortunato et al. "Noisy Networks for Exploration". In: *CoRR* abs/1706.10295 (2017). arXiv: `1706.10295`. URL: `http://arxiv.org/abs/1706.10295`.

[30] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47 (2004), pp. 235–256.

[31] Mohammad Gheshlaghi Azar, Ian Osband, and Rémi Munos. *Minimax Regret Bounds for Reinforcement Learning*. 2017. arXiv: `1703.05449 [stat.ML]`.

[32] Richard Y. Chen et al. "UCB and InfoGain Exploration via $\backslash boldsymbol\{Q\}$-Ensembles". In: *CoRR* abs/1706.01502 (2017). arXiv: `1706.01502`. URL: `http://arxiv.org/abs/1706.01502`.

[33] Yang Liu et al. "Stein Variational Policy Gradient". In: *CoRR* abs/1704.02399 (2017). arXiv: `1704.02399`. URL: `http://arxiv.org/abs/1704.02399`.

[34] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *CoRR* abs/1801.01290 (2018). arXiv: `1801.01290`. URL: `http://arxiv.org/abs/1801.01290`.

[35] Tuomas Haarnoja et al. "Soft Actor-Critic Algorithms and Applications". In: *CoRR* abs/1812.05905 (2018). arXiv: `1812.05905`. URL: `http://arxiv.org/abs/1812.05905`.

[36] Tuomas Haarnoja et al. "Learning to Walk via Deep Reinforcement Learning". In: *CoRR* abs/1812.11103 (2018). arXiv: 1812.11103. URL: http://arxiv.org/abs/1812.11103.

[37] Pierre-Yves Oudeyer and Frederic Kaplan. "How can we define intrinsic motivation?" In: (July 2013).

[38] Marc G. Bellemare et al. *Unifying Count-Based Exploration and Intrinsic Motivation*. 2016. arXiv: 1606.01868 [cs.AI].

[39] Haoran Tang et al. "#Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning". In: *CoRR* abs/1611.04717 (2016). arXiv: 1611.04717. URL: http://arxiv.org/abs/1611.04717.

[40] Yuri Burda et al. "Exploration by Random Network Distillation". In: *CoRR* abs/1810.12894 (2018). arXiv: 1810.12894. URL: http://arxiv.org/abs/1810.12894.

[41] Marlos C. Machado, Marc G. Bellemare, and Michael Bowling. "Count-Based Exploration with the Successor Representation". In: *CoRR* abs/1807.11622 (2018). arXiv: 1807.11622. URL: http://arxiv.org/abs/1807.11622.

[42] Özgür Şimşek, Alicia P. Wolfe, and Andrew G. Barto. "Identifying Useful Subgoals in Reinforcement Learning by Local Graph Partitioning". In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: Association for Computing Machinery, 2005, pp. 816–823. ISBN: 1595931805. DOI: 10.1145/1102351.1102454. URL: https://doi.org/10.1145/1102351.1102454.

[43] Amy Mcgovern and Andrew Barto. "Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density". In: *Proceedings of the 18th International Conference on Machine Learning (ICML 2001)* (July 2001).

[44] Alec Solway et al. "Optimal Behavioral Hierarchy". English (US). In: *PLoS Computational Biology* 10.8 (Aug. 2014). Funding Information: James S. McDonnell Foundation (http://www.jsmf.org/). National Science Foundation (#1207833) (nsf.gov). John Templeton Foundation (http://www.templeton.org/). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript. Publisher Copyright: 2014 Solway et al. ISSN: 1553-734X. DOI: 10.1371/journal.pcbi.1003779.

[45] Marlos C. Machado, Marc G. Bellemare, and Michael H. Bowling. "A Laplacian Framework for Option Discovery in Reinforcement Learning". In: *CoRR* abs/1703.00956 (2017). arXiv: 1703.00956. URL: http://arxiv.org/abs/1703.00956.

[46] Yuu Jinnai et al. "Exploration in Reinforcement Learning with Deep Covering Options". In: *ICLR*. 2020.

[47] Tejas D. Kulkarni et al. "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation". In: *CoRR* abs/1604.06057 (2016). arXiv: 1604.06057. URL: http://arxiv.org/abs/1604.06057.

[48] Karol Gregor, Danilo Jimenez Rezende, and Daan Wierstra. "Variational Intrinsic Control". In: *CoRR* abs/1611.07507 (2016). arXiv: `1611.07507`. URL: `http://arxiv.org/abs/1611.07507`.

[49] Yuu Jinnai et al. "Discovering Options for Exploration by Minimizing Cover Time". In: *CoRR* abs/1903.00606 (2019). arXiv: `1903.00606`. URL: `http://arxiv.org/abs/1903.00606`.

[50] Will Dabney, Georg Ostrovski, and André Barreto. *Temporally-Extended ϵ-Greedy Exploration*. 2020. arXiv: `2006.01782 [cs.LG]`.

[51] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *CoRR* abs/1710.02298 (2017). arXiv: `1710.02298`. URL: `http://arxiv.org/abs/1710.02298`.

[52] Steven Kapturowski et al. "Recurrent Experience Replay in Distributed Reinforcement Learning". In: *International Conference on Learning Representations*. 2019. URL: `https://openreview.net/forum?id=r1lyTjAqYX`.

[53] Ian Osband et al. "Deep Exploration via Bootstrapped DQN". In: *CoRR* abs/1602.04621 (2016). arXiv: `1602.04621`. URL: `http://arxiv.org/abs/1602.04621`.

[54] Andre Biedenkapp et al. "Towards TempoRL: Learning When to Act". In: *Workshop on Inductive Biases, Invariances and Generalization in Reinforcement Learning (BIG@ICML'20)*. 2020.

[55] Aravind S. Lakshminarayanan, Sahil Sharma, and Balaraman Ravindran. "Dynamic Action Repetition for Deep Reinforcement Learning". In: *AAAI*. 2017, pp. 2133–2139. URL: `http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14866`.

[56] Sahil Sharma, Aravind S. Lakshminarayanan, and Balaraman Ravindran. "Learning to Repeat: Fine Grained Action Repetition for Deep Reinforcement Learning". In: *CoRR* abs/1702.06054 (2017). arXiv: `1702.06054`. URL: `http://arxiv.org/abs/1702.06054`.

[57] Edouard Leurent. *Answer to 'Why do we use the Ornstein Uhlenbeck Process in the exploration of DDPG??'* 2018. URL: `https://www.quora.com/Why-do-we-use-the-Ornstein-Uhlenbeck-Process-in-the-exploration-of-DDPG` (visited on 10/12/2021).

[58] Pawel Wawrzynski. "Control Policy with Autocorrelated Noise in Reinforcement Learning for Robotics". In: *International Journal of Machine Learning and Computing* 5 (Apr. 2015), pp. 91–95. DOI: `10.7763/IJMLC.2015.V5.489`.

[59] Marcin Szulc, Jakub Lyskawa, and Pawel Wawrzynski. "A framework for reinforcement learning with autocorrelated actions". In: *CoRR* abs/2009.04777 (2020). arXiv: `2009.04777`. URL: `https://arxiv.org/abs/2009.04777`.

[60] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[61] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: *CoRR* abs/1802.09477 (2018). arXiv: 1802.09477. URL: http://arxiv.org/abs/1802.09477.

[62] Adrien Ecoffet et al. *First return then explore*. 2020. arXiv: 2004.12919. URL: https://arxiv.org/abs/2004.12919.

[63] Tim Salimans and Richard Chen. "Learning Montezuma's Revenge from a Single Demonstration". In: *CoRR* abs/1812.03381 (2018). arXiv: 1812.03381. URL: http://arxiv.org/abs/1812.03381.

[64] Hado van Hasselt et al. "Expected Eligibility Traces". In: *CoRR* abs/2007.01839 (2020). arXiv: 2007.01839. URL: https://arxiv.org/abs/2007.01839.

[65] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2018). URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[66] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[67] Michael Janner, Qiyang Li, and Sergey Levine. "Reinforcement Learning as One Big Sequence Modeling Problem". In: *CoRR* abs/2106.02039 (2021). arXiv: 2106.02039. URL: https://arxiv.org/abs/2106.02039.

[68] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015), pp. 529–33. DOI: 10.1038/nature14236.

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

[70] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8 (2004), pp. 229–256.

[71] Silviu-Ioan Filip, Aurya Javeed, and Lloyd Nicholas Trefethen. "Smooth random functions, random ODEs, and Gaussian processes". In: *SIAM Review* 61.1 (Feb. 2019), pp. 185–205. DOI: 10.1137/17M1161853. URL: https://hal.inria.fr/hal-01944992.

A Experiment Details

	standard	ϵz-greedy	TempoRL
Target algorithm runs	60	60	60
n steps unit evaluation	10k	10k	50k

Table 1: SMAC: Setup details for the FD

	standard	ϵz-greedy	TempoRL
Target algorithm runs	100	100	100
n steps unit evaluation	5k	5k	5k

Table 2: SMAC: Setup details for the cSigmoid/ToySGD

	standard	ϵz-greedy	TempoRL
batch size	278	275	222
ϵ-time steps	11089	23946	92560
grad-clip	95	293	230
replay-buffer	239072	53262	765942
learning-rate	2.4611164354334e-4	3.8054207575027998e-4	2.7848167856432e-4
ϵ_{ADAM}	7.000259403149209e-5	2.3290087314832e-2	5.86186532660397e-5
skip learning-rate	-	-	6.9656041549348e-1

Table 3: Resulting HP from SMAC for FastDownward

B More Results

Note that the reward per instance trace in figure 21 and 22 are both sorted by the reward obtained the first step displayed.

	standard	ϵz-greedy	TempoRL
batch size	287	292	273
ϵ-time steps	1152	2760	3748
replay-buffer	31313	20815	37955
learning-rate$_{\text{Critic}}$	1.2040803388404977e-3	3.2262874314196413e-4	1.0940409181702218e-3
learning-rate$_{\text{Actor}}$	3.2452505512504413e-6	1.0920133927934668e-5	4.211326615852546e-5
$\epsilon_{\text{ADAM, Critic}}$	6.1016821107851345e-2	2.8078151250751957e-4	1.0369898859136326e-7
$\epsilon_{\text{ADAM, Actor}}$	1.799360761230243e-5	6.15016826596228e-2	4.876674927991768e-7
skip learning-rate	-	-	4.872790513473528e-3
$\epsilon_{\text{ADAM, skip}}$	-	-	1.804226347427204e-6

Table 4: Resulting HP from SMAC for ToySGD

	standard	ϵz-greedy	TempoRL
batch size	229	288	209
ϵ-time steps	2048	1000	2840
replay-buffer	7853	4089	5395
learning-rate$_{\text{Critic}}$	1.6652916897287964e-4	2.080608555940994e-3	3.450666434507508e-5
learning-rate$_{\text{Actor}}$	6.63593238127607e-4	1.3560609818309601e-3	1.532304435840087e-3
$\epsilon_{\text{ADAM, Critic}}$	3.602894629751527e-6	8.991285337344055e-7	5.559711949304255e-6
$\epsilon_{\text{ADAM, Actor}}$	1.8174453527401532e-5	2.4931664221053354e-7	1.6550739239352137e-2
skip learning-rate	-	-	1.0404612599553022e-6
$\epsilon_{\text{ADAM, skip}}$	-	-	4.096035811204022e-5

Table 5: Resulting HP from SMAC for cSigmoid

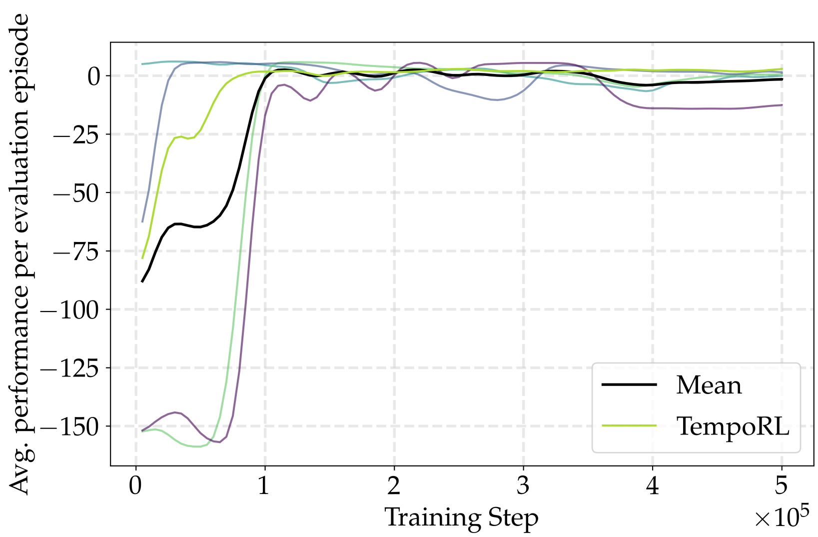

Figure 20: Mean performance of the TempoRL SAC

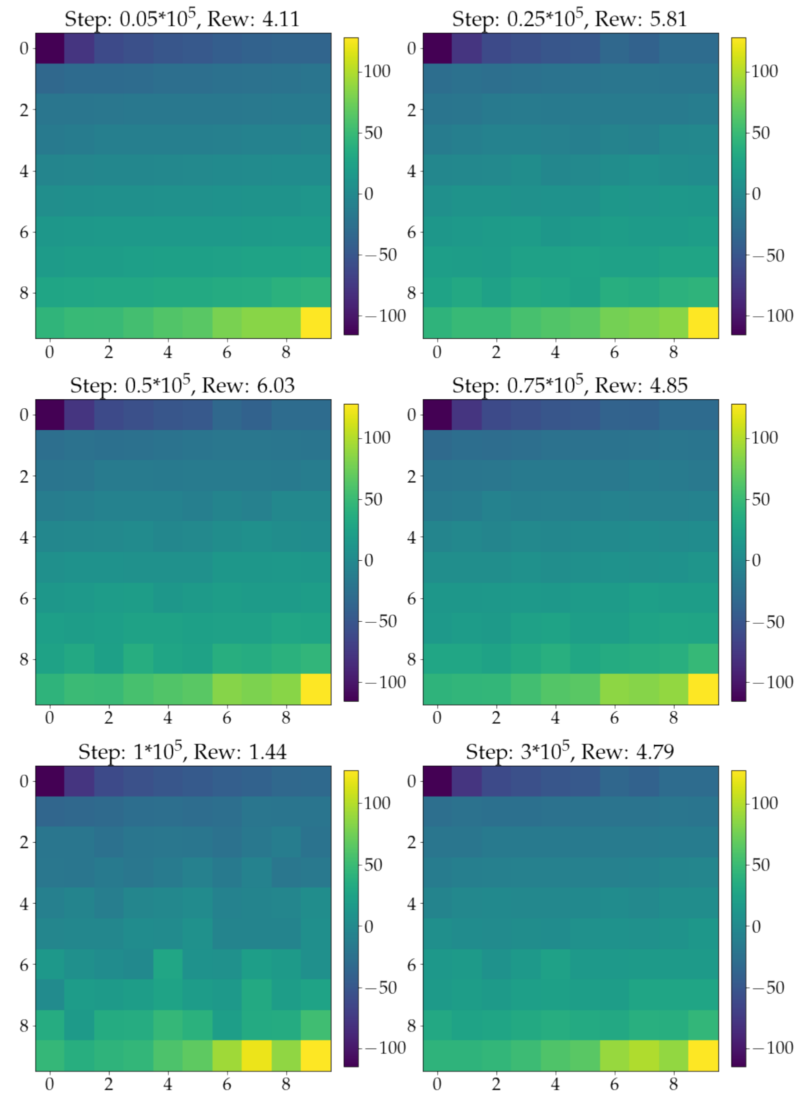

Figure 21: Progress of the per-instance reward in one run of regular Soft Actor-Critic.

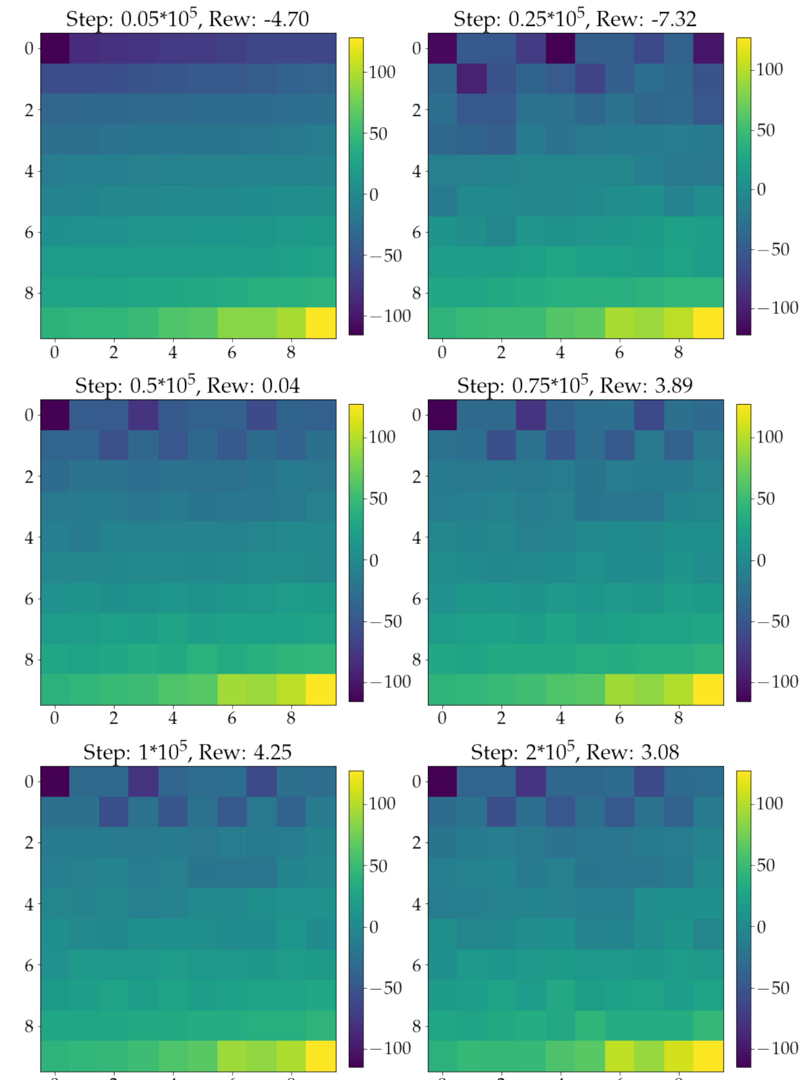

Figure 22: Progress of the per-instance reward in one run of ϵz-greedy Soft Actor-Critic.

	regular SAC	ϵz-greedy	TempoRL
	-0.83339	-0.96797	-0.85545
	-0.97097	-0.97154	-0.85545
	-0.9595	-0.93119	-0.8821
	-0.94788	-0.94679	-0.8821
	-0.92637	-0.90555	-0.8557
	-0.89099	-0.88726	-0.8557
	-0.84054	-0.83578	-0.25648
	-0.76335	-0.78044	-0.25648
	-0.66717	-0.71234	-0.24145
	-0.51871	-0.63748	-0.24145
	-0.32788	-0.55608	-0.07478
Total reward:	19.18	19.87	18.11

Table 6: Rounded learning rates of the agents at time step 499999 in instance 82. Note that this a part of the action executed and converted through 10^{action} internally.