Exploring Intrinsic Motivation for Quanser Reinforcement Learning Benchmark Systems

Erforschen von Intrinsischer Motivation für die Quanser Reinforcement Learning Benchmark Systems

Master-Thesis von Jakob Weimar aus Bad Soden

Februar 2020

TECHNISCHE UNIVERSITÄT DARMSTADT

Exploring Intrinsic Motivation for Quanser Reinforcement Learning Benchmark Systems
Erforschen von Intrinsischer Motivation für die Quanser Reinforcement Learning Benchmark Systems

Vorgelegte Master-Thesis von Jakob Weimar aus Bad Soden

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: M.Sc. Svenja Stark

Tag der Einreichung:

Erklärung zur Master-Thesis

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Jakob Weimar, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

In the submitted thesis the written copies and the electronic version are identical in content.

Datum / Date: Unterschrift / Signature:

_____ _____

Abstract

Deep Reinforcement Learning is a primary research topic for achieving superhuman level of control in robotics. However, efficient state space exploration still remains a difficult research topic. In this thesis we focus on improving the efficiency of state-of-the-art reinforcement learning algorithms such as DQN, DDPG, TRPO, and PPO by formalizing intrinsic motivation factors in order to mimic human learning behaviour. We evaluate our approach by training control policies for classical control tasks on the CartPole environment such as balancing, swinging up and swinging up in a sparse setting in simulations. We implement a novel way of parallel sampling for OpenAI Gym environments in order to improve the amount of samples our algorithms can use. At first, we tune the algorithms on the mentioned tasks without intrinsic motivation. After that, we introduce novel variations of *Surprisal* and *Prediction Error* intrinsic motivation for reward enhancement. The results prove that intrinsic motivation can help reduce the amount of samples needed for convergence when properly applied, at a cost of computational complexity.

Zusammenfassung

Deep Reinforcement Learning ist ein führendes Forschungsthema wenn es um das Erreichen von übermenschlichen Leistungen im Bereich der Robotik geht. Trotzdem bleibt eine effiziente Erforschung des Zustandsraums weiterhin ein schwieriges Forschungsthema. In dieser Arbeit konzentrieren wir uns auf die Verbesserung der Effizienz von state-of-the-art Reinforcement Learning Algorithmen wie DQN, DDPG, TRPO und PPO durch die Integration von intrinsischer Motivation, um menschliches Lernverhalten nachzuahmen. Wir bewerten unseren Ansatz anhand von Benchmarks für klassische Steuerungsaufgaben in der CartPole-Umgebung wie Balancieren, Aufschwingen und Aufschwingen in einer Umgebung mit kargen Belohnungsfaktoren in Simulationsumgebungen. Wir implementieren eine neuartige Methode zur parallelen Probenahme für OpenAI Gym Umgebungen um die Anzahl an Samples für unsere Algorithmen zu erhöhen. Zunächst stimmen wir die Algorithmen auf die genannten Aufgaben ohne intrinsische Motivation ab. Danach führen wir neuartige Varianten von intrinsische Motivation, namentlich *Surprisal* und *Prediction Error*, ein. Unsere Ergebnisse zeigen, dass intrinsische Motivation dazu beitragen kann, die Menge an Samples, die für die Konvergenz der Algorithmen benötigt werden, auf Kosten der Rechenkomplexität reduzieren kann.

Acknowledgments

Extensive calculations on the Lichtenberg high-performance computer of the Technische Universität Darmstadt were conducted for this research. The authors would like to thank the Hessian Competence Center for High Performance Computing – funded by the Hessen State Ministry of Higher Education, Research and the Arts – for helpful advice.
Also special thanks to Jonas Eschmann, Robin Menzenbach, and Christian Eilers giving helpful advice on the general hyperparameter tuning of DQN.

Contents

Figures

List of Figures

1 Motivation

As Reinforcement Learning (RL) agents become more adept at solving tasks with well-defined progression markers, also called rewards, research shifts towards exploring approaches that promise better performance in more difficult optimization landscapes such as ones featuring deceptive local optima or sparse rewards. These environments more closely resemble problems that, generally speaking, RL agents need to overcome in order to successfully solve real-world tasks where reward functions often do not exist or are expensive to produce. The most promising approaches to solve this class of problem include so-called hierarchical algorithms. Sometimes they are using manually designed proxy-rewards. Another option is intrinsic motivation, which is divided into subcategories such as curiosity, novelty, surprisal and learning progress motivation [Oudeyer and Kaplan, 2007]. We further introduce intrinsic motivation in Section 3.3.3.

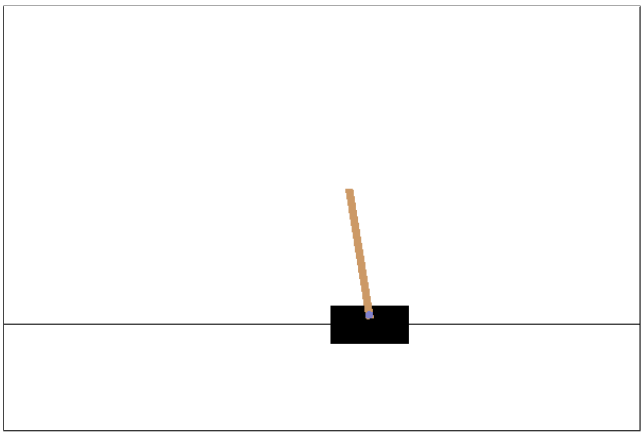


Figure 1.1.: A visual representation of the Quanser CartPole simulation.

Even environments such as the CartPole environment shown in Figure 1.1 can be difficult to effectively navigate for machine learning algorithms when trying to solve tasks where rewards are sparse. The reason for this problem is the fact that a sequence of correct inputs needs to be made in order to receive any non-zero reward signal. As the odds of this case to occur when only random actions are applied are very low, many RL algorithms never find any meaningful reward signals at all.

Another difficulty is the so-called "credit-assignment problem" [McDuff and Kapoor, 2018]: classical RL algorithms follow promising increases of the reward in order to maximise the expected reward over time. However, if rewards are sparse, it is not always possible to discern which action ultimately lead to the earned reward, if any reward-increasing state is found at all. One possible solution is classical supervised learning, where an algorithm is presented with labelled data that it simply regresses on. This approach generally works well when such labelled data is available, but it also implies that the algorithm is severely limited by the quantity and quality of available data. A good example is learning a pong algorithm from human demonstrations. If the human is a skilled player, the algorithm will usually also perform well. In this way, the algorithm does not learn any notion of good or bad moves, instead, it just blindly copies what the provided samples show. Furthermore, it will not generate any emerging strategies on its own.

When looking back at how humans learn, imitation of a teacher (supervised learning) is a central aspect but by far not the only process at work. When humans encounter new objects, they try a range of actions available to them and gather sensory feedback. Reinforcement learning tries to imitate this human behaviour and transfer it to machine learning agents. In this thesis, we focus on improving the way Reinforcement Learning agents explore their surroundings by taking inspiration from our understanding of learning in humans.

Classical approaches to solve this exploration problem for agents often feature difficult human design processes or rely on random chance for exploration. Intensive human intervention is exactly what we want to steer away from when using Reinforcement Learning since its goal is to have an agent learning the task for us, so humans do not need to provide a manually designed curriculum or produce lots of well-labelled samples.

At the same time, approaches that explore using random chance quickly break down when applied to tasks which require multiple steps or have deceptive local optima. As such, it would be much more natural to teach algorithms to explore the environment autonomously, taking previously gathered knowledge into account. One way to implement this idea is by using a heuristic to compute how information can be gained by acting off-policy at each point in time. This approach would make exploration much less random and much more focused on states which are unexplored and by the algorithms standards less predictable.

This approach not only serves an exploration purpose, it also encourages the algorithm to seek out states that are not yet fully understood, while still not being too advanced to make any learning progress in. Achieving such a behaviour would put the algorithm into a continuous, "flow"-like state. This idea can also be used to learn basic surrogate skills which are then used for more complicated tasks, effectively building a bridge towards hierarchical approaches. In human

psychology, such a behaviour is called intrinsically motivated. This term is often used as an umbrella term for machine learning approaches that try to imitate human learning behaviour.
Even though intrinsic motivation has garnered interested in the field of RL over the past years, there is still a distinct lack in comparison between different RL algorithms enhanced by intrinsic motivation. In this thesis, we provide an overview of the performance of different RL algorithms on the Quanser Reinforcement Learning Benchmark Systems using intrinsic motivation. We begin by introducing our research questions.
The first question we want to answer is how well the different algorithms perform on the Quanser Reinforcement Learning Benchmark Systems. In order to do this, we introduce baseline performances of the different algorithms and the different variants of intrinsic motivation are then compared against these baselines. These baselines also allow us to determine which tasks are sufficiently covered by classic exploration methods and which tasks might benefit from intrinsic motivation. We can also compare whether intrinsic motivation improves or worsens results of tasks that can already be solved.
The second question we want to answer is whether intrinsic motivation can help guide agents out of local optima. Wherever classic exploration methods cannot fully solve an environment, we investigate if intrinsic motivation can help the algorithm converge to the global optimum by effectively guiding it out of a local optimum. The answer to this question might include looking at tasks where the agent is not able to produce any results, but also ones where the agents get stuck in local optima.
The third question we want to answer is if intrinsic motivation can effectively guide an agent to explore systems where rewards are sparse. We also investigate tasks where finding any rewards is difficult for the existing algorithms, which helps us determine how effective intrinsic motivation is for exploration problems. Further, we explore methods by which we can determine whether the agents produce any useful behaviour at all, even if no reward signal was received.
The fourth question we want to answer is if intrinsic motivation affects the convergence properties of the agents. One major concern for using intrinsic motivation is whether an agent will be slowed down significantly during its optimization process if we introduce intrinsic motivation. We also investigate methods in order to reduce the impact of these problems.
The fifth question we want to answer is if intrinsic motivation affects the computational complexity of the algorithms. Another major concern when using intrinsic motivation is whether the used methods cause significant impacts on computational complexity and as such on the cost of learning. We aim to introduce as little computational overhead as possible.
The final question we want to answer is how the different RL algorithms compare to each other in general. We conclude with general assessments of the different RL algorithms. We focus on the characteristics of the different algorithms, their general exploration properties, their hyperparameter sensitivity and how the impact of our intrinsic motivation algorithms differs between RL algorithms.

1.1 Structure

Now that we have introduced our research questions, we present the structure we follow when answering them. In the next chapter, we discuss related work. Afterwards, we present the methods which we base our research on, beginning with RL, continuing with Deep RL and finishing with exploration for Deep RL. In Chapter 4, we introduce the intrinsic motivation variants we introduce. In Chapter 5, we present the tasks and environments we use, followed by the implementation chapter, in which we discuss implementation details and hyperparameter tuning. Following this, we present the results of our experiments and discuss them in detail. Based on this, we then answer our research questions in Chapter 8, which we finish with a general conclusion and an outlook on future research.

2 Related Work

In this chapter, related research is presented in order to inform about the general landscape of current research as well as for later reference. We begin by introducing some differing approaches to using intrinsic motivations before moving to approaches more similar to our own research.

2.1 Bootstrapping Intrinsically Motivated Learning with Human Demonstrations

[Nguyen et al., 2011] combine social learning and intrinsic motivation in an approach called "Socially Guided Intrinsic Motivationary Demonstration", or SGIMD. SGIMD is a two-level approach: The high-level algorithm sets goals, while the low-level algorithm tries to reach these goals. The intrinsic motivation is part of this algorithm in an approach called "Self-Adaptive Goal Generation-Robust Intelligent Adaptive Curiosity", also called SAGG-RIAC. This is a concrete application of the competence-based intrinsic motivation. The competence is defined as

$$\gamma_{y_g} = \begin{cases} \text{Sim}(y_g, y_f, \rho), & \text{if Sim}(y_g, y_f, \rho) \leq \epsilon_{\text{sim}} < 0 \\ 0, & \text{otherwise,} \end{cases}$$

where y_g is a set goal state, y_f is the goal the algorithm actually reached and ρ are other constraints. Then, $\text{Sim}(y_g, y_f, \rho)$ is a rating of how close y_g and y_f are, which directly relates to how close the algorithm is to actually reach the set goal. The high-level algorithm uses this to compute a measure of interest for each goal and chooses the next task for the low-level algorithm accordingly.
This paper does not address the difficulties that come with real world situations including correspondence and a biased teacher. Instead, it is evaluated on continuous, unbounded and non-preset environments, such as a "continuous 24-dimension action space" in simulations, e.g. throwing a fishing rod.

2.2 Accuracy-based Curriculum Learning in Deep Reinforcement Learning

[Fournier et al., 2018] use the DDPG-algorithm to perform automated curriculum learning. The algorithm is evaluated on the Reacher Environment. The way the learning progress is measured is taken directly from the SAGG-RIAC algorithm [Baranes and Oudeyer, 2013] (also explained in 2.1). The reward is defined as follows:

$$r = \begin{cases} 0, & \text{if } \left|p_{\text{finger}} - p_{\text{target}}\right| \leq \epsilon \\ \text{-1}, & \text{otherwise} \end{cases},$$

where ϵ is defining how close the agent needs to get the finger (end effector) to the target before it receives a reward. There are two modes discussed for choosing ϵ: RANDOM-ϵ, where ϵ is sampled uniform from $E = \{0.02, 0.03, 0.04, 0.05\}$ for each episode, and ACTIVE-ϵ, where ϵ is $P(\epsilon_i) = cp_i^\beta / \sum_k cp_k^\beta$, where cp_i is the competence progress for ϵ_i.

2.3 Unifying Count-Based Exploration and Intrinsic Motivation

[Bellemare et al., 2016] use information gain, which is commonly used to quantify novelty or curiosity to enhance exploration in non-tabular RL in order to perform well on Atari games such as Montezuma's Revenge. The information gain is used to improve the performance of Double DQN (DDQN). Density models are used to measure uncertainty. The information gain is defined as

$$\mathbf{IG}_n(x) := \mathbf{IG}_{x;x_{1:n}} := KL(w_n(\cdot, x)||w_n) \tag{2.1}$$

where $x \in X$ is a state and $w_n(x)$ is the posterior weight. This can be be approximated using PG:

$$\mathbf{PG}_n(x) := \log \rho_n'(x) - \log \rho_n(x) \tag{2.2}$$

2.4 Surprise-Based Intrinsic Motivation for Deep Reinforcement Learning

[Achiam and Sastry, 2017] learn a model of the MDP transition probabilities concurrently with the policy for continuous environments such as Atari games. "Surprisal" is used as intrinsic motivation flavour. The transformed reward used in Trust Region Policy Optimization (TRPO) is defined as

$$r'(s,a,s') = r(s,a,s') + \eta(\log P(s'|s,a) - \log P_\phi(s'|s,a)), \tag{2.3}$$

where $r(s,a,s')$ is the original reward, P_ϕ is the learned model, P is the transition probability function and $\eta > 0$ is an exploitation-exploration trade-off factor. Alternatively, the reward can be defined as

$$r'(s,a,s') = r(s,a,s') + \eta(\log P_{\phi_t}(s'|s,a) - \log P_{\phi_{t-k}}(s'|s,a)), \tag{2.4}$$

where the true transition probability function is not required for learning. Evaluation happens on OpenAI Gym environments that are modified to feature sparse rewards, such as sparse MountainCar, sparse CartPole Swing-Up and sparse HalfCheetah, as well as the Atari games such as Pong, BankHeist, Freeway and Venture.

3 Methods

In this chapter, we introduce all the general methodology about Reinforcement Learning (RL) and intrinsic motivation that is needed in order to guide our own approach in Chapter 4. We start by explaining RL in general. Afterwards, we explain Deep RL and the algorithms we use for our research, namely DQN, DDPG, TRPO, and PPO. Finally, we introduce the concept of the exploration-exploitation trade-off for RL and different classical exploration methods as well as intrinsic motivation.

3.1 Reinforcement Learning

In RL, an agent is tasked with acting in an environment in such a way that it maximises some form of reward [Sutton and Barto, 1998]. Usually, these environments are interpreted as some form of Markov decision process (MDP) defined as a tuple (S, A, R_a, P_a), where S is a set of states, A is the set of actions available to the agent, $R_a(s, s')$ is the reward the agent receives when taking action a in state s and ending up in state s'. $P_a(s)$ describes the state transition probabilities for all combinations of state and action. A visual representation of the RL loop can be found in Figure 3.1.

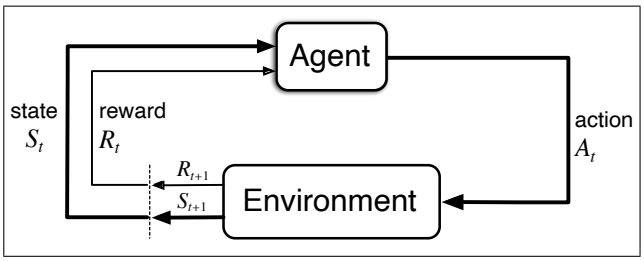


Figure 3.1.: A visual representation of the RL Loop. The agent receives state S_t and reward R_t and outputs an action A_t based on the observations. The environment then produces new S_t and R_t signals.
Source: [Sutton and Barto, 1998]

With this definition, we can compute G_t, the discounted return of an episode starting from time t up to the final step T as

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{3.1}$$

where R_k is the reward experienced by the agent at time k, plus all future rewards, exponentially decaying by some factor $0 < \gamma < 1$. $\gamma = 0$ means no future rewards are taken into account while $\gamma = 1$ implies that the total future reward is taken into account.

In order to formally define how agents take actions, we use a policy $\pi(s, a)$ such that $A_t = a$ given $S_t = s$, or in other words, how likely it is that we choose action a from all possible actions at time t given that we are in state s out of all possible states S_t at that point in time. The policy is sometimes simply noted as $\pi(s)$, in which case it returns some probability distribution that describes the likelihood of all actions.

With this definition, we can define functions that help us evaluate possible states and future actions. The first metric we introduce is the notion of "value", usually defined as $V_\pi(s)$. It indicates how valuable - or in other words desirable - it is to be in a certain state. The computation of this value function depends on the specific algorithm, the current policy and their parameters. The general definition according to [Sutton and Barto, 1998] is

$$V_\pi(s) = \mathop{\mathbb{E}}_{\pi}\left[G_t|_{S_t=s}\right] = \mathop{\mathbb{E}}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|_{S_t=s}\right], \quad \text{for all s} \in S, \tag{3.2}$$

all variable definitions being equal to their introduction at the beginning of this section. The value function corresponding to following an optimal policy is usually defined as $V^*(s)$.

In some scenarios it is helpful to differentiate between the different actions we can take and what their expected value is. In theory, having a way to compute the expected returns of each possible action would allow the agent to compare different actions in the current state under the assumption that we act according to some policy π in all future states. We call this function the q-function, and it is defined as

$$Q_\pi(s, a) = \mathop{\mathbb{E}}_{\pi}\left[G_t|_{S_t=s, A_t=a}\right] = \mathop{\mathbb{E}}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|_{S_t=s, A_t=a}\right], \quad \text{for all s} \in S, \tag{3.3}$$

where again, the discount γ, the reward R and the states S are defined analogous to the value function. In theory, being able to compute the optimal q-function $Q^*(s,a)$ would allow for optimal decision making. Both states and actions can be either continuous or discrete, depending on the algorithm and environment.
Additionally, in RL we are often faced with a difficult question: Is the expected return of a certain action a better than acting according to our policy or not? Which leads to the question whether we should change our policy to make that action more likely if the answer is yes or less likely if the answer is no. Simply looking at $Q_\pi(s,a)$ does not suffice to determine whether this is the case. For different states and different policies, vastly different ranges for Q can be considered high or low. Instead, we need to use a measurement that instead of encoding "good" or "bad" in terms that are not straightforward to interpret, tells us if a certain action is "better" or "worse".
Mathematically speaking, we now have the necessary groundwork to put this notion in a single value, which is generally called *advantage*. The advantage is defined as

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s). \tag{3.4}$$

As a quick reminder, $V_\pi(s)$ encodes the value of starting in state s and acting according to the policy π. $Q_\pi(s,a)$ encodes the value of starting in state s, taking action a and from then on acting according to policy π. For example, if we choose a $= \pi(s)$, this means that $Q_\pi(s,a) = V_\pi(s)$ and thus our advantage is 0. A positive advantage when choosing an action a differing from the policy would mean that $Q_\pi(s,a) > V_\pi(s)$ and thus we can improve our expected return by updating our policy to make that action more likely. Similarly, a negative advantage suggests that action a is worse in state s and the agent should adapt accordingly. Now that we have introduced different measures for deciding in which direction we want to update our agents, we need a way to actually apply updates to our agents.

Gradients

The gradient of a function f is the vector field containing its partial derivatives [Ruder, 2016]. If we now see our agent as a function $f_\theta(s)$ that takes in a state and outputs an action and depends on the parameters θ, we can use the gradient in order to update our agent. Since our agent is defined by some parameters θ, we want to know how we need to shift these individual parameters in order to change the outputs our agent produces. The basic approach is that we define a loss function, such as the Mean Squared Error (MSE), which is defined as

$$\text{MSE}_\theta = V_{\text{desired}}(s) - V_\theta(s). \tag{3.5}$$

We want this error term to converge to 0, which means that our agent produces the desired outputs. The error can never get below zero since we square the difference. As a result, attempting to reduce the error to zero is equivalent to finding a global minimum. The gradient already tells us the direction of steepest ascent, so at each step we can go into the opposite direction in order to minimise this error.
Since the optimization landscape might not be smooth, hitting local minima instead of a desired, global minimum is highly likely. Being stuck in such a minimum is one of the key problems we attempt to solve in this thesis.

3.2 Deep Reinforcement Learning

In recent years, Deep RL has rapidly grown in popularity due to advances in computing resources. With Deep RL, neural networks of varying sizes are used as function approximators in varying forms, for example in order to estimate the Q- or Value-functions. We present four well known Deep RL algorithms: DQN, DDPG, TRPO and PPO. Afterwards, we present how classic exploration works for these algorithms before introducing intrinsic motivation.

3.2.1 Deep Q-networks (DQN)

Deep Q-networks (DQN) is a RL algorithm that optimizes in parameter space [Mnih et al., 2015]. It is a variant of Q-learning that approximates the q-function using deep neural networks. This approach offers the advantages of a non-linear function approximation, but also destabilizes the algorithm since small changes in parameter values can have big impacts on the resulting policy. A couple of measures to improve the stability are discussed after the general algorithm introduction. DQN learns the q-function and operates on a set of discrete, predefined actions. States can be either continuous or discrete. The agent keeps a replay buffer of the last N encountered state transitions and at each update step, a mini batch of transitions is sampled that will then be used to update the q-function approximator. The algorithm

uses a copy of this approximator that is only updated rarely in order to approximate the future rewards. The general update equation is

$$y_j = \begin{cases} R_j, & \text{if episode terminates at step j+1} \\ R_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-), & \text{otherwise.} \end{cases} \tag{3.6}$$

In this case, R_j is the reward experienced by the agent during state transition j, plus all future rewards, exponentially decaying by some factor $0 < \gamma < 1$. The delayed copy of the function approximator, $\hat{Q}(s_{j+1}, a'; \theta^-)$, is used to estimate the future rewards, based on the assumption that at each point in time the agent acts optimally according to the delayed q-function approximation.
A couple of tricks can be used to increase the stability of the algorithm. **Experience Replay** reduces the correlation between learning data and current policy. With this technique, the learning data is randomly sampled from an experience buffer that always stores a set number of experiences tuples. **Clamping error values** reduces the impact of outliers. The difference between y_j and the current network output is clamped between -1 and 1 before squaring. This clamping can also be detrimental to algorithm performance when outliers provide important learning examples. **Delayed Q-network updates** only update the network used for future predictions every C steps. This freezing of the function delays the impact that updates have on the current policy network. In other terms, it decouples the predicted values of states that are close to each other, preventing a catastrophic inflation in value function values. Additionally, we only update the policy network every ten steps during our research, which reduces computation time. This compromise is especially relevant when using intrinsic motivation variants that are computationally expensive.

3.2.2 Deep Deterministic Policy Gradients (DDPG)

While the ramifications of Deep Deterministic Policy Gradients (DDPG) [Lillicrap et al., 2015] are similar to DQN, there are two major differences that distinguish the algorithms. Firstly, DDPG is based on continuous action spaces, opposed to DQN's discrete actions. Secondly, DDPG uses two neural networks instead of one, namely an actor and a critic. The critic, in similar style to the DQN network, approximates the q-function $Q(s, a|\theta^Q)$, and is thus often referred to as Q. This q-function approximator however is not used to directly form the policy. That is the task of the actor network, which is the reason that it is often denoted as $\mu(s|\theta^\mu)$. Q is only used to build the gradient that updates μ. Again, the algorithm keeps a replay buffer and randomly samples from it in order to update the networks and delayed copies of the networks for approximating future rewards denoted as $Q(s, a|\theta^{Q'})$ and $\mu(s, a|\theta^{\mu'})$, though the update steps of these networks differ from DQN. The loss function and update rule for the critic network is

$$L = \frac{1}{N} \sum_j (y_j - Q(s_j, a_j|\theta^Q))^2, \tag{3.7}$$

which is the mean squared error of the current q-function approximator compared to its supposed value y_j. This value y_j is calculated as

$$y_j = \begin{cases} R_j, & \text{if episode terminates at step j+1} \\ R_j + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}), & \text{otherwise,} \end{cases} \tag{3.8}$$

where again, R_j is the reward experienced by the agent during state transition j, plus all expected future rewards when acting according to the delayed policy $\mu'(s)$, exponentially decaying by some factor $0 < \gamma < 1$.
The case for absorbing states is not denoted in the original DDPG paper, however it is integral for any experiments that are not infinite horizon by default and likely just an accidental omission of the original authors.
The gradient that is used to update the actor is defined as

$$\nabla_{\theta^\mu} \approx \frac{1}{N} \sum_j \nabla_a Q(s, a|\theta^Q)|_{s=s_j, a=\mu(s_j)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_j}, \tag{3.9}$$

where N is the number of samples used for the update step, and $\nabla_a Q(s, a|\theta^Q)|_{s=s_j, a=\mu(s_j)}$ is the gradient of the current q-function based on the current state transition samples. The policy gradient $\nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_j}$ is computed in an analogous manner.
Additionally, the update steps for the target networks is designed to be smoother than DQN updates, i.e. they are defined as

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'} \tag{3.10}$$

and

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}, \tag{3.11}$$

where $0 < \tau < 1$ is an update factor, usually close to 0 that slowly "drags" the delayed copies of the networks behind the current ones in a smoother manner than DQN which does full updates every n steps. The major disadvantages of DDPG are the increased computational complexity and the introduction of additional hyperparameters since there are two networks that need to be defined and updated. In our experience, using DDPG is still worthwhile since it is more stable in terms of learning progress and less sensitive to parameter misconfiguration. One major consideration when using DDPG is that it is very sensitive towards the output values of the actor network - thus any scalar outputs should be normalized to be between zero and one.

3.2.3 Trust Region Policy Optimization (TRPO)

Trust Region Policy Optimization (TRPO) [Schulman et al., 2015a] is a Monte Carlo method and limits updates of a policy to a so-called "Trust Region". This trust region is defined as the KL-Divergence between the old and the new policy. Update steps are also computed in such a way that each parameter is shifted proportionally to how much impact it has on the resulting policy, i.e. by taking its second derivative into account. The general expectation TRPO maximises is

$$\max_{\theta} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] - \beta KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]. \tag{3.12}$$

In each training iteration, the parameters are updated such that

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k \tag{3.13}$$

$$s.t. \quad \bar{D}_{KL}(\theta_{old}||\theta_{new}) \leq \delta, \tag{3.14}$$

where θ_k are the policy parameters at time-step k, $\hat{x}_k$ are the rates of changes of each parameter, $\hat{H}_k$ is the Hessian and δ is a hyperparameter limiting the difference between old and new policy. In order to improve efficiency, several approximations need to be made, such as the backtracking line search using the factor α^j where $0 < \alpha < 1$ is predefined and j is the smallest positive integer so that the constraint is not violated. Also, since computing the Hessian and its inverse for each iteration is very expensive, especially if the policy features a lot of parameters, the conjugate gradient algorithm is used together with the fact that we can compute Hx using

$$Hx = \nabla_\theta((\nabla_\theta \bar{D}_{KL}(\theta||\theta_k))^T x), \tag{3.15}$$

where $\nabla_\theta \bar{D}_{KL}(\theta||\theta_k)$ is the gradient of the policy describing the velocity with which each parameter changes the KL-divergence when updated [spi, 2019].
In general, TRPO is a very stable and well-performing algorithm due to the cautious update policy. Its downsides lie in the fact that policy samples cannot be reused often (if at all), resulting in a low sample efficiency. The algorithm can also get stuck in local optima due to its limited ability to perform large jumps in the policy landscape in order to escape out of said bad optima.

Generalized Advantage Estimation

In order to guide TRPO towards increasing rewards, we need some way to weigh our policy gradient. Actions that are considered positive for the task should increase in likelihood (positive weight) and actions that lead to negative consequences decrease in likelihood (negative weight). One possible option when using a roll-out based method is the total future reward. For this implementation, we choose the so-called Generalized Advantage Estimation (GAE). First introduced in [Schulman et al., 2015b], GAE not only discounts future rewards but also evaluates the actions and rewards compared to the currently expected rewards, forming a notion of "better or worse compared to what we know" rather than "good or bad". In order to do this, first we compute the residuals δ as

$$\delta_t^V = R_t + \gamma V(s_{t+1}) - V(s_t), \tag{3.16}$$

with R_t being the reward at time t and $V(s)$ being the estimated value of state s. Intuitively, these residuals quantify the difference between estimated value of state s_t compared to the actually received reward plus the estimated, discounted future rewards.
This estimation is then used in a similar fashion to how one would compute regular, discounted future rewards to form the Generalized Advantage Estimation $\hat{A}_t^{GAE(\gamma,\lambda)}$ as

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{l} (\gamma\lambda)\delta_{t+l}^{V}. \tag{3.17}$$

Note that the factor $0 < \gamma \leq 1$ is usually the discount factor and $0 < \lambda \leq 1$ controls the bias-variance trade-off, with high values denoting a low bias and high variance and low values denoting the opposite. In this case however, the math works out so that γ and λ form a single factor, which effectively means that for GAE looking further into the future means higher variance. We primarily use GAE for our experiments when using TRPO or PPO.

3.2.4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [Schulman et al., 2017] tries to solve the same general problem as TRPO: How can we ensure that an update step actually improves the overall policy? In a similar vein to TRPO, PPO achieves this guarantee by limiting the size of the possible update step, albeit in a different manner. For the purposes of this research, we use the clipped variant of the PPO loss, defined as

$$L^{CLIP}(\theta) = \mathbb{E}_t\left[\min(R_t(\theta)\hat{A}_t, \text{clip}(R_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right], \tag{3.18}$$

where $\hat{A}_t$ is the advantage and $0 < \epsilon < 1$ is some clipping value. A larger clipping value allows for larger policy steps, thus speeding up policy evolution, at the cost of less policy stability. The choice of ϵ is an important trade-off for PPO. On the one hand, lower ϵ values make the algorithm more stable, however it also follows that the algorithm is more likely to get stuck in local optima and takes longer to converge. On the other hand, larger ϵ values may lead to quicker convergence but also increase the likelihood of fatal changes in the policy.
The advantage is again computed using Generalized Advantage Estimation as described above. The upside of PPO when compared to TRPO is that the loss can be easily optimized using standard Gradient Descent algorithms such as ADAM or RMSprop, however one loses TRPO's parameter sensitivity.

3.3 Exploration-Exploitation Trade-Off

Since the agent does not know anything about how the environment works and thus how e.g. the value function might be computed. Therefore it needs to start by blindly exploring according to some initial policy $\pi(s)$ - this initial exploration is always necessary before the agent can make any informed decisions about which action to take. As time goes on and the agent learns about the environment, it is then able to take more informed actions in order to achieve its goal, which usually correlates to maximizing some reward function defined in the environment black box. So at each point in time, the agent has to choose between either maximizing the reward by *exploiting* its current knowledge or further *exploring* the environment. This trade-off has no immediately visible solution: If the agent is in a local optimum, it does not know whether it is also in a global optimum. Multiple steps might need to be completed before a better optimum can be found. We call this concept exploration-exploitation dilemma, and finding a good balance is essential for achieving good efficiency in any non-trivial RL problem. This problem is even further exacerbated in environments that only feature sparse rewards. In these cases, finding an action sequence that leads to any rewards at all can be very difficult, and even when such a sequence is found, it is not always clear which specific actions lead to the gain in reward. Overall, we want to maximise the amount of useful behaviour an agent exhibits. Usually, the only measure available to distinguish useful from non-useful behaviour is thus the reward function specific to the task and environment. We will now discuss different ways to solve the exploration-exploitation dilemma.

3.3.1 Classical Exploration Methods

Heuristics that aim to solve the exploration-exploitation dilemma exist in various forms. First, we want to introduce the well-known, classical examples of ϵ-greedy exploration, Ornstein-Uhlenbeck process, gaussian policies and reward shaping. Afterwards, we discuss intrinsic motivation variants.

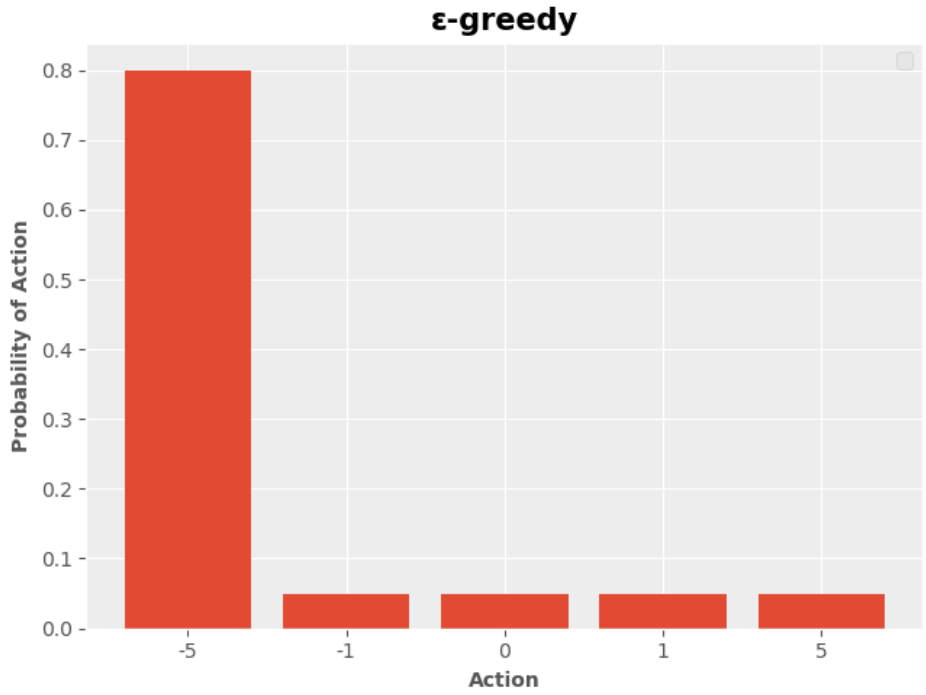


Figure 3.2.: ϵ-greedy exploration visualized for the discrete actions [-5.0, -1.0, 0.0, 1.0, 5.0]. In this case $\epsilon = 0.2$. The agent chose action "-5", which means that there is a 80% of taking that action and all other actions have a probability of being taken of 5%.

ϵ-greedy exploration

With the ϵ-greedy exploration approach, an agent has a chance of ϵ during each time-step to take a random action, i.e.

$$\pi_\epsilon(s) = \begin{cases} \text{random } a \in A_t, & x_{rnd} < \epsilon \text{ (exploration)} \\ \pi(s), & \text{otherwise (exploitation)}, \end{cases} \tag{3.19}$$

where $0 < x_{rnd} < 1$ is a random number chosen each time the $\pi_\epsilon(s)$ is evaluated. A visualization of the ϵ-greedy exploration can be found in Figure 3.2.
The probability ϵ may decay as training epochs go on, slowly making the agent more deterministic as exploitation is favoured over exploration. This approach functions without any definition of what useful behaviour is, it is thus completely unbiased. Nevertheless, this approach has proven effective, especially if the reward function of the environment is shaped well. However, it is suboptimal in different ways. Since it explores in an unbiased way, it might repeatedly explore parts of the state space that are already sufficiently explored, wasting computational time. At the same time, exploration mostly takes place close to the current policy, larger deviations exponentially decrease in likelihood with the amount of off-policy steps that have to be taken. As a result, ϵ-greedy exploration is not well-suited for sparse reward settings, where exactly such exploration is necessary.

Ornstein–Uhlenbeck Process

Typically, Ornstein-Uhlenbeck (OU) processes [Uhlenbeck and Ornstein, 1930] are used for exploration with DDPG [Lillicrap et al., 2015]. A short explanation of the OU process is that it is a random walk with a bias of moving back towards the mean of the process. A visualization can be found in Figure 3.3. The process is controlled by the four parameters a, μ, σ and θ. It is defined by the equations

$$dX(t) = \sigma \cdot (\mu - X(t))dt + \sigma dB(t), \text{ with } \quad X(0) = a. \tag{3.20}$$

$\mu \in R$ is the mean where the process usually starts and to which it moves back over time. $\sigma > 0$ is the standard deviation which controls the normal distribution from which the next state is sampled. $\theta > 0$ controls the bias of how strong the bias towards the mean is. $a \in R$ is the beginning state of the process and $B(t)$ is a Brownian motion. Correct choice of

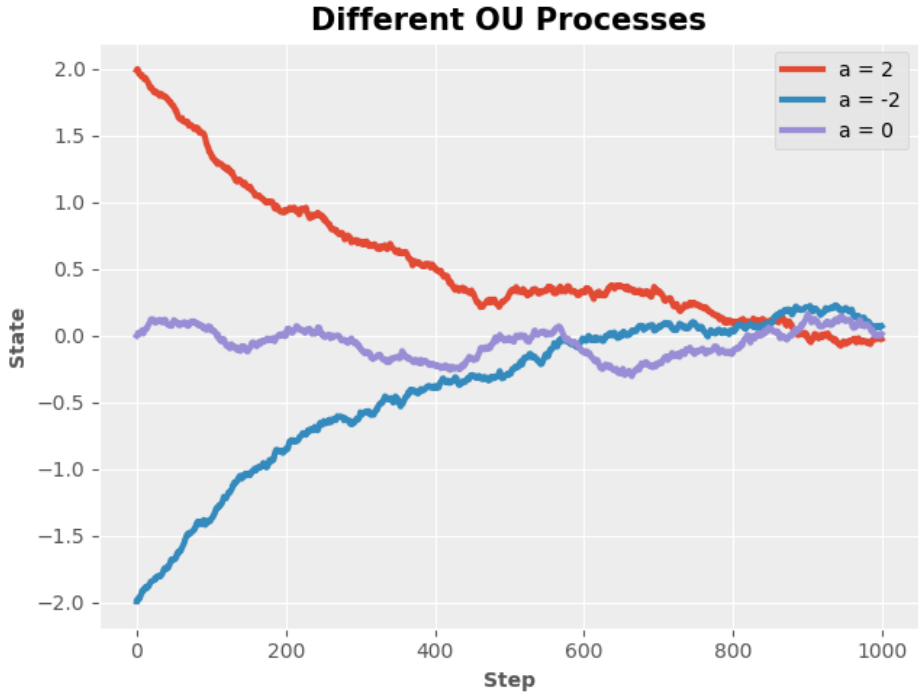


Figure 3.3.: Ornstein–Uhlenbeck Processes with $\theta = 0.4$, $\sigma = 0.1$, $\mu = 0$ and different starting positions a. One can observe the processes tendency to return to the mean over time, even when starting from positions further away.

these parameters is paramount for a successful run of DDPG. Even slightly wrong values or a bias in the random number generator of a computer can completely decide whether a run is successful or not.

Non-Deterministic Policies

Many algorithms that can work with continuous action spaces have non-deterministic policies as exploration method built-in. Algorithms such as TRPO and PPO usually output normal distributions, from which actions are then sampled. This exploration approach is similar to ϵ-greedy exploration, however it features some key differences. One is that the standard deviation of the distribution impacts the algorithm's consistency - higher standard deviation means more exploration, but a less stable policy. Thus, tuning of the standard deviation is important. One can configure the deviation to be an output of the neural network, a hyperparameter or a training variable. Another important difference is that the algorithm is highly unlikely to explore far away from the distribution mean, also visualized in Figure 3.4. This can lead to problems when using trust region algorithms such as TRPO or PPO. Since the distribution is not allowed to change too much for each iteration, and actions far away from the current action mean are unlikely to be sampled, it is possible for such an algorithm to never explore certain parts of the action space that are separated from the agents starting position by an optimization valley.

3.3.2 Reward Shaping

Another common approach is reward shaping. With reward shaping, the reward function is enhanced such that the agent is guided towards a good policy, e.g. by adding action penalties, or metrics that improve in a steady manner as the agent approaches an optimal solution. In theory, such a well-shaped reward function together with an exploration method such as ϵ-greedy should allow an agent to efficiently reach a goal.
In practice, however, several problems arise. The first being that by providing such an explicit guideline, the probability of an agent developing a creative solution is severely diminished. Much like in supervised learning, where an algorithm is limited by the samples it is given, an algorithm that solely relies on a "friendly" reward function will always be limited to following a curriculum developed by humans. This problem directly ties into the next point: For each new task and/or environment, a new reward function needs to be developed, greatly reducing the general applicability of this approach. At the same time, generating such a reward function is not straightforward. Frequently, agents find unusual and unintended exploits in the optimization landscapes of such reward functions.

Figure 3.4.: Example of a non-deterministic policy. In this case a gaussian distribution that is the output of a TRPO network.

As a result, finding an exploration strategy that allows an agent to explore its environments independently would be far more applicable to real-world problems.

3.3.3 Intrinsic Motivation

According to [Ryan and Deci, 2000], "Intrinsic motivation [in humans] is defined as the doing of an activity for its inherent satisfaction rather than for some separable consequence. When intrinsically motivated, a person is moved to act for the fun or challenge entailed rather than because of external products, pressures, or rewards". One real-life example for this phenomenon can be seen in the way babies explore their environments, e.g. by putting new toys in their mouths. Another example would be adults learning new skills like juggling because it is fun, in contrast to learning a skill in order to impress other people or generate income. So in short, the distinction between intrinsic and extrinsic motivation is the reason a person (or in our case, an algorithm) does something [Legault, 2016]. The difference between internal and external rewards is similar. They are used to describe where the task reward stems from. As is not unusual in the field of psychology, these different rewards and motivations do not necessarily need to be exclusive to each other and drawing clear lines is not always possible, which is also reflected in their algorithmic counterparts, which we discuss in later parts of this work.

General Notations

The most general element of intrinsic motivation is usually some event k, also noted as

$$e^k \in \mathbb{E}. \tag{3.21}$$

The specific set of defined events $\mathbb{E}$ depends on both the RL problem and the form of intrinsic motivation used. This set of all events is not to be confused with E, the notation we use for prediction errors of all kind. Possible event definitions can be specific states, state-action pairs or being in a range of states.

Applying Intrinsic Motivation Rewards

One way of applying intrinsic motivation with any RL algorithm is to combine the intrinsic and extrinsic reward signals, possibly scaled by some factor. This approach features maximum compatibility and ease of application. However, it

puts a larger burden on the RL algorithm, which now has a noisy reward signal featuring two different sources of information. On top of that, the intrinsic reward signal is a constantly shifting goalpost, further increasing the difficulty of convergence. When using this approach, we evaluate closely whether the algorithms are conversely effected in their convergence property.
As combining both extrinsic and intrinsic rewards into a single denominator is straightforward but not necessarily ideal when it comes to the optimization landscape, we propose a second solution: Instead of combining the two rewards, we leave them separate until the last part of the optimization step. In a concrete example like TRPO, choosing this approach would mean computing two value functions that could then be used to compute two sets of advantage estimates. Optimization could then be done using either one of the two, corresponding to exploration or exploitation respectively. The decision when to use which of the two value functions would be of major importance for the general performance of this approach. Since this trade-off is difficult to fine tune and needs to be explicitly repeated for each algorithm, we do not use this approach during our research. Nevertheless, it is important to mention that intrinsic motivation can also be used to enhance algorithms in a more specific, targeted manner.
With these equations in place, we can define the following types of intrinsic motivation.

Mathematical Interpretations of Intrinsic Motivation

In this section, we give a general overview of how RL and intrinsic motivation can be modelled mathematically. Discussion and in-depth explanation is done in the corresponding sections in Chapter 4. [Oudeyer and Kaplan, 2007] is used to provide the general categorization as well as mathematical definitions. The introduced terms are used as a baseline to compare against for the different implementations in other papers.

Predictive Novelty Motivation

Predictive Novelty Motivation (NM) is a straightforward way to use the prediction error for rating samples as interesting, simply scaling the error by some constant. The formulation is

$$r(SM(\rightarrow t)) = C \cdot E_r(t), \tag{3.22}$$

where C is a scaling factor and $E_r(t)$ is some error that the agent can compute. In theory, this reward makes states where the error is large more interesting, resulting in them being sought out more often, which should provide relevant samples to reduce the error. If the policy is able to learn a correct model, this intrinsic motivation term should naturally taper out over time as the model gets perfected.

Learning Progress Motivation

Learning Progress Motivation (LPM) tries to maximise the prediction/learning progress. It compares the learning progress for regions that are already similar (denoted by $\mathscr{R}$). Finding these regions can be difficult, one general approach is clustering. More information can be found in [Oudeyer et al., 2007]. The general notation is defined as

$$r(SM(\rightarrow t)) = \left\langle E_r^{\mathscr{R}n}(t-\Sigma)\right\rangle - \left\langle E_r^{\mathscr{R}n}(t)\right\rangle, \tag{3.23}$$

where $r(SM(\rightarrow t))$ is a predictor for the reward based on the current state and action. $E_r^{\mathscr{R}n}(t)$ is the prediction error at time t and Σ is a predefined amount of time-steps.

4 Approach

In this chapter, we discuss ways in which intrinsic motivation rewards can be applied to algorithms. Traditionally, RL algorithms try to measure some aspects of the environment, such as an extrinsic reward that encodes good and bad states. We call this extrinsic reward $R_{\text{ex}}(e^k)$, where e^k is some event on the basis of which this reward is computed. This event can for example encode states and actions. The algorithm then tries to relate actions it has taken with this measure in order to update itself. If we have just one measure, this behaviour can be computed in a straightforward manner. Good rewards increase the likelihood of taking an action, bad rewards reduce it and we add some noise, e.g. for exploration. However, this approach is not always sufficient to allow RL agents to solve tasks. We are especially interested in algorithm performance in tasks with at least one local optima or non-informative (sparse or flat) rewards. In order to evaluate the algorithms in these regards, we use stabilization of a pendulum with both regular and non-informative reward functions, swing-up of a pendulum with a potential local optima and swing-up of a pendulum with a sparse reward function.
As a means of improving algorithm performance on these tasks, we introduce intrinsic motivation to the agents. We do this by developing new measures $R_{\text{im}}(e^k)$, which encode intrinsic motivation factors, where e^k is some event similar to the definition in $R_{\text{ex}}(e^k)$. By doing this, we aim to guide agents using $R_{\text{im}}(e^k)$ even when $R_{\text{ex}}(e^k)$ is non-informative or deceiving. As a result we arrive at two major questions:

1. How can we apply both $R_{\text{im}}(e^k)$ and $R_{\text{ex}}(e^k)$ to our algorithms?
2. How do we choose to compute $R_{\text{im}}(e^k)$ in the first place?

We will first discuss the former and then the latter.

4.1 Applying Intrinsic Motivation Rewards

In this section we briefly discuss the different ways to apply intrinsic motivation. The general trade-off in this regard is whether the approach is early in the RL loop, which might make it more generally applicable, or later in the loop, which make the agent to some extent aware of the difference between intrinsic and extrinsic rewards. However, our research is specifically targeted at producing a comparison between different algorithms. As a result, we stick with the approach of combining the rewards outside of the RL agents boundaries. Thus we enhance the regular, extrinsic reward as:

$$R_{\text{combined}}(s) = R_{\text{ex}}(s) + C \cdot R_{\text{im}}(s), \tag{4.1}$$

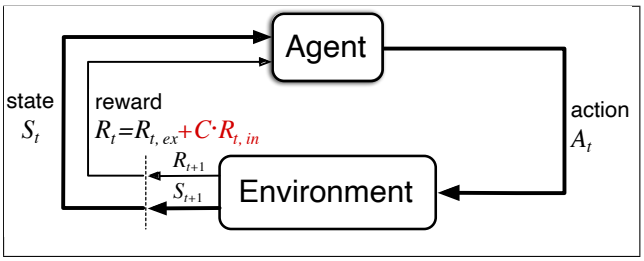


Figure 4.1.: Visualization of the RL loop with intrinsic motivation added according to the weighted addition approach.
Modified from: [Sutton and Barto, 1998]

with C being a manually chosen scaling constant. The new combined reward $R_{\text{combined}}(s)$ is then used in place of the regular reward $R_{\text{ex}}(s)$. A visualization of where in the RL loop this is applied can be found in Figure 4.1
This approach is a very plug-and-play solution. Basically any existing RL algorithm can be enhanced without changing fundamentals of the algorithm. However, one of the major problems is that it breaks the implicit promise of a single optimization target the agent moves towards. Instead of having a single goal that is to be optimized (exploring or exploiting), it gets a mash-up of the two, where the intrinsic reward is constantly moving. On top of that, the proper scaling between the rewards needs to be optimized. One approach is similar to ϵ-greedy decay: reducing the factor C over time to aid exploration in the beginning of the learning session while helping convergence once learning has progressed further.

4.2 Predictive Novelty Motivation using *Prediction Error*

Since concrete estimations of our probability distribution prove difficult and do not scale well with complex state spaces, we propose other methods of quantifying how familiar a state is. One approach to solve this problem is learning the state

transition function $P(s'|s,a)$. For this approach we assume a deterministic environment, so that $P(s'|s,a) = 1$ for exactly one $s' \in S$ and $P(s'|s,a) = 0$ for all other $s \in S$. If we can correctly predict the short term effects of an action a in state s, executing said action is unlikely to result in an improvement of the agents understanding of the state space. As such, we learn a neural network $\Phi(s,a) \rightarrow s'_\Phi$ with the mean squared error clamped to the interval $[0,1]$. We can then use the average error between the predicted next state, s'_Φ, and the actual observed next state, s', as a base for an intrinsic reward by computing

$$R_{\text{im}} = \frac{1}{N_s} \cdot \sum (\text{clip}_{[-1,1]}(s'_\Phi - s'))^2, \tag{4.2}$$

with $s \in R^{N_s}$ and N_s being the dimensionality of the environment specific states. If we assume that our network is capable of learning a full state transition model, this error will slowly converge towards zero over the course of the experiment. This behaviour means that the intrinsic reward also slowly tapers out. As a result, the agent learns to maximise the extrinsic reward R_{ex} without any noise from intrinsic rewards. Additionally, we subtract a baseline from R_{im}, i.e. the minimum R_{im} of the current update iteration. This measure ensures that the least familiar state-action pair always receives zero additional reward. A different option for a baseline would have been the mean, but that would cause the intrinsic rewards to reach into the negative numbers, something we want to specifically avoid since the agent could possibly benefit from ending an episode early in that case.

4.3 Learning Progress Motivation using *Surprisal*

The last approach we want to introduce is *Surprisal*, which is also sometimes called "Learning Progress Motivation". This approach is strongly influenced by [Achiam and Sastry, 2017], introduced in Section 2.4. *Surprisal* is also based on learning a state transition model $P(s'|s,a)$. The model we learn is of the form

$$P_\Phi(s'|s,a) = \mathcal{N}(\Phi(s,a), \sigma_\Phi), \tag{4.3}$$

where Φ is the trained neural network and σ is a separate, tunable parameter of the network. This differs from [Achiam and Sastry, 2017], who propose learning a bayesian neural network.
We then use this model to compute the intrinsic reward as

$$R_{im} = C \cdot (\log P_{\Phi'}(s'|s,a) - \log P_\Phi(s'|s,a)), \tag{4.4}$$

where $0 < C < 1$.
This particular approach has the disadvantage that in order to estimate the intrinsic rewards, an update to the network has to be made. This solution is well-suited for algorithms such as TRPO and PPO, which only update once for every N sampled episodes. For algorithms such as DQN and DDPG that feature a replay buffer and update multiple times per episode, this solution can be a problem. One has essentially two options in this case:
1) Predicting the intrinsic rewards for each update of the algorithm. So for each update, we also have to update the transition model of our *Surprisal* module. The problem with this approach is that it is computationally very expensive, making it impractical. Reducing the amount of algorithm updates per episode (and thus the amount of total updates needed for a training run) helps to reduce the impact of this problem, however this approach remains unfeasible.
2) Predicting the intrinsic rewards for each state-action pair in the replay buffer once every training episode. The first problem with this approach is that newly sampled transitions cannot be added to the replay buffer until the current episode is finished, since there is no intrinsic reward computed for them yet. The second problem is that for large replay buffers, one is likely to predict the intrinsic reward for state-action pairs that are never used in an algorithm update.
As such, we use the first option for the remainder of our research.

5 Experiments

The experiments feature multiple algorithms both in their base form as well as with added intrinsic motivation on multiple gym environments, with added evaluation of learned policies on real robots. In the next sections, we introduce the tasks and environments used for the evaluation of the different algorithms. Each environment can have different tasks, while we mostly focus on classic mechanical tasks.

5.1 OpenAI Pendulum

An example of the first environments used for each algorithm is the *Pendulum-v0*. Previous experience allowed us to make good guesses about certain parameters such as $\gamma = 0.999$. At the same time, the simplicity of the environment without many deceptive local optima and low stochasticity allowed for a wider range of parameters to produce good policies. On top of that, episodes of *Pendulum-v0* are fixed length and as such make advantage estimation more straightforward.
The *Pendulum-v0* is a standard OpenAI Gym environment [Brockman et al., 2016]. It consists of a single pendulum fixated on an axle. A visual representation is depicted in Figure 5.1. The state space of the Pendulum consists of the pendulum angle and velocity. The observation space is similar, however instead of the pendulum angle, its sine and cosine are returned. This transformation serves to remove jumps from the pendulum angle where it would jump from $-\pi$ to π or vice versa. The alternative solution of letting the angle continue outside of a range of 2π would result in state aliasing, which is also not desired in this case. Actions are clipped between -2.0 and 2.0.

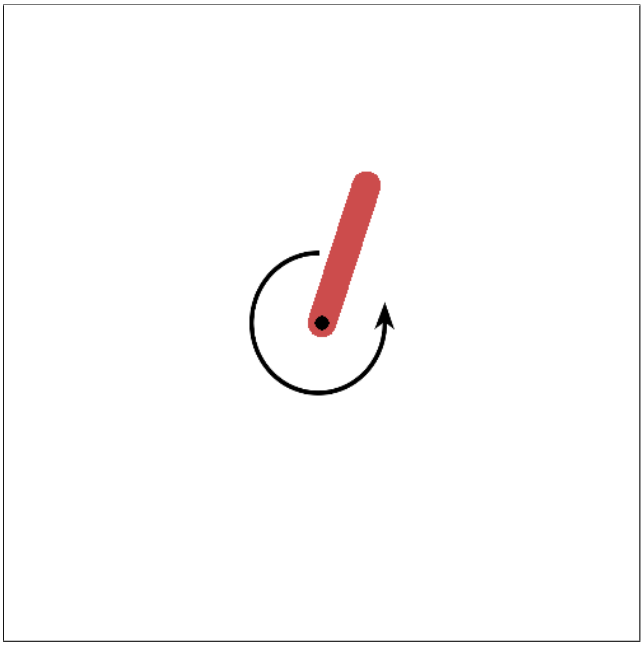


Figure 5.1.: A visual representation of the *Pendulum-v0* environment. The pendulum is mounted at a fixed axis around which it can turn. The current velocity is displayed as the arrow.

5.1.1 Pendulum-v0

The default task on the *Pendulum-v0* is the classic swing-up. The pendulum starts at a uniformly random angle with a random velocity. As such, it is possible that the pendulum is already in an upright, almost stable position at the beginning of the episode, significantly decreasing the difficulty of the task for that specific episode.
The reward function is defined as

$$R_t = -(\theta_t^2 + 0.1 \cdot \theta_{\mathrm{dt}_t}^2 + 0.001 \cdot a_t^2), \tag{5.1}$$

where θ_t is the pendulum angle, θ_{dt_t} is the pendulum angle velocity and a_t is the action taken at the current time-step, after being clipped. This cost naturally guides the agent to conserve energy. It also ensures that the agent does not take large actions that are clipped anyway so it does not leave the desired action space. This problem is mostly an issue for algorithms that do not use discrete action spaces such as TRPO and PPO. We see the effects of omitting such a penalty term when looking at environments that clip actions but do not penalize them in the reward.
It is important to note that just because an algorithm is able to solve this pendulum, does not necessarily mean that it is implemented without errors. In our case, a bug with the Generalized Advantage Estimation did not prevent TRPO from solving the pendulum since it only became apparent when bad actions could lead to a premature end of the current episode.

5.2 CartPole Environments

The main research platform we use are the Quanser Reinforcement Learning Benchmark Systems [Qua, 2019]. The following CartPole environments are implemented in the quanser robots repository [Belousov et al., 2020]. They all feature a free swinging pendulum mounted on a cart that can move along a line. The cart is controlled by applying voltage to a motor, in this case between -24.0 and 24.0 Volts, which directly correlates to the actions an agent can take. A visual representation is depicted in Figure 5.2. The state space of the CartPole consists of the cart position, cart velocity, pendulum angle and pendulum angle velocity. The observation space is similar, however instead of the pendulum angle, its sine and cosine are returned. Each episode either ends after 10000 time-steps regularly, or prematurely when the cart reaches its position limits to the left or the right edge of the visualization. For the purposes of our research, we have noticed an improvement in performance if we repeat actions and thus reduce the control frequencies. In all our experiments, each action is applied five times transparently for the algorithms and the rewards of these steps are added.

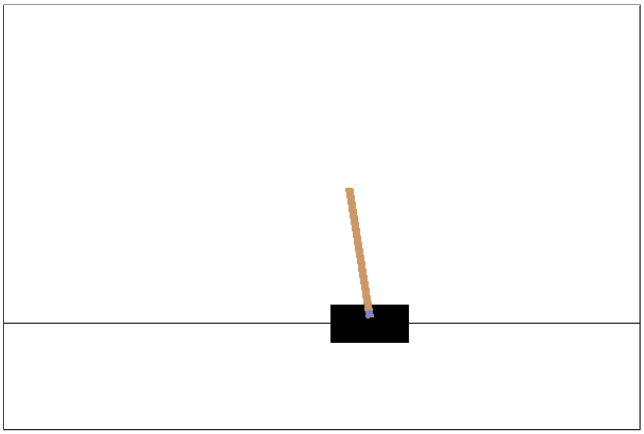


Figure 5.2.: A visual representation of the CartPole environments. The original visualization is part of the OpenAI Gym environment, though usage for the Quanser CartPole is analogous.

In our experiments we use an additional cut-off condition. Some agents can learn that spinning the pendulum at ever increasing speeds is a local optimum. After certain speeds, this high velocity means that the pendulum angle can reach arbitrary positions in each time-step, since the time deltas are large compared to the pendulum velocity. On top of that, the agent might perform a successful swing-up, but "skips" over the topmost position, which also falsifies the overall episode return.

Since policies that exhibit this behaviour exploit the simulation, and would also be potential damaging for the real robot and outright dangerous for any experimenters, we introduce a safety shut-off at any pendulum velocities greater than a certain threshold. Additionally, the CartPole environment technically allows for voltages in a range of $[-24.0, 24.0]$. However, applying these kinds of voltages to the real robot will result in increased wear, which is why we clip any actions to the range of $[-5.0, 5.0]$. This constraint still allows fulfilment of all tasks on the pendulum, however, they get harder.

5.2.1 CartpoleStabShort-v0

In this simulation of the CartPole, the pendulum starts in an upright position with low velocities for both the cart and the pendulum. In addition to the cut-off conditions mentioned in the previous section, the episodes of *CartpoleStabShort-v0* also end when the pendulum angle deviates too far from the upright position, meaning swinging around fully is impossible. The reward is defined as

$$R_t = -\cos(\theta_t) + 1, \tag{5.2}$$

where θ_t is the pendulum angle at time t. The highest reward is achieved when the pendulum is in an upright position. There are no penalties for taking actions and achieving large velocities. An episode is counted as terminated when either the time horizon is reached or when

$$\theta_t \mod 2\pi \in [\pi - 0.25, \pi + 0.25] \tag{5.3}$$

is no longer true, i.e. the pendulum deviates to far from an upright position.

Since the reward is the cosine of the pendulum angle, and any deviations from the desired position end the episode anyway, good total rewards are mostly dependent on keeping the episode going for as long as possible. A perfect policy is mostly expected to reach a reward nearing the full 10000, from gaining close to 2.0 reward at each of the 5000 possible time-steps. In our preliminary experiments, algorithms such as DDPG, TRPO and PPO should be able to learn perfect policies without extensive hyperparameter tuning. As such, this environment is well-suited for testing purposes, especially of the computations for rewards-to-go and Generalized Advantage Estimates.

5.2.2 CartpoleStabShortNoInfo-v0

CartpoleStabShortNoInfo-v0 is a slight variation to the regular *CartpoleStabShort-v0* environment we introduce additionally. The only difference lies in the way the reward is shaped. Instead of being dependent on the pendulum angle, it is always 2.0 until the episode ends. This task allows us to answer the question of whether the information about the angle that is usually encoded in the reward signal is important for the algorithm to learn stabilization or not.

5.2.3 CartpoleSwingShortNiceReward-v0

Reward shaping is a classical alternative to intrinsic motivation and is used by us as a baseline comparison. Using reward shaping can help guide an agent out of a local optimum, however, a reward function consisting of more variables is also more difficult to learn for an algorithm.

$$R_{\text{sh}} = R_{\text{ex}} - (\lambda_{\text{dt}}\theta_{\text{dt}}^2 + \lambda_{\text{a}}a^2), \tag{5.4}$$

with $\lambda_{\text{dt}} = 0.1$ and $\lambda_{\text{a}} = 0.001$
The difficulty when designing this task is that episodes can have variable lengths. As a result, the reward must not be negative; otherwise the algorithm can be incentivised to end episodes such that it does not accrue a bunch of negative reward early during the initial learning phases. This reward is also rescaled for each task so that the maximum reward that can be gained is comparable to the maximum reward achievable when using the regular reward function.

5.2.4 CartpoleSwingShort-v0

The *CartpoleSwingShort-v0* is an advanced task on the CartPole environment presented in Section 5.2. In this form of the CartPole environment, the pole starts in a downward hanging position. To achieve the maximum possible reward, an agent needs to perform a successful swing-up and then balance the pole for the remaining time of the episode. This task requires at least a basic ability to perform two-step tasks and basic state-space exploration. The reward is defined as

$$R_t = -\cos(\theta_t) + 1, \tag{5.5}$$

where θ_t is the pendulum angle at time t.
A close-to-perfect policy achieves around ∼9800 reward, only losing about ∼200 reward from the theoretical maximum during the swing-up period. A policy that accrues around ∼5000 reward usually performs a rotating motion, continuously swinging the pendulum in one direction but not learning to stabilize it at the top.
The difficulty for transition-based methods such as DQN lies in the sparsity of absorbing states. Any agent needs to learn the bounds in which the cart is allowed to move, since ending episodes prematurely by violating these bounds greatly reduces the gained reward. One episode usually features several thousand state transitions. Since DQN uses a replay buffer and only uses a small sample (usually in the double digit range) of these transitions each time-step, only every dozen updates may even include such an absorbing state, leading to the algorithm constantly "forgetting" and relearning the bounds. This behaviour leads to an erratic learning curve, where the agent jumps between policies that are perfect, policies that just rotate the pendulum and policies that prematurely end the episode.
The difficulty for rollout-based methods such as TRPO and PPO lies in the exploration of the state space. Where an algorithm like DQN randomly takes actions of which it has no idea how they will perform and can piece together transitions from different runs, the actions TRPO chooses are always highly correlated to the current policy.

5.2.5 CartpoleSwingShortSparse-v0

It is important to state that similar tasks have been mentioned before, e.g. in [Achiam and Sastry, 2017] (further explained in Section 2.4). The *CartpoleSwingShortSparse-v0* task is the last one we introduce for our research. This task is analogous to the regular *CartpoleSwingShort-v0*, however the reward function is significantly less informative. Specifically, the reward is defined as

$$R_t = \begin{cases} -\cos(\theta_t) + 1, & \theta_t \mod 2\pi \in [\pi - 0.25, \pi + 0.25] \\ 0, & \text{otherwise,} \end{cases} \tag{5.6}$$

where θ_t is the pendulum angle at time t. The cut-off criteria for the reward is analogous to the point at which the stabilization for all CartPole balancing tasks is considered to be failed.

6 Implementation

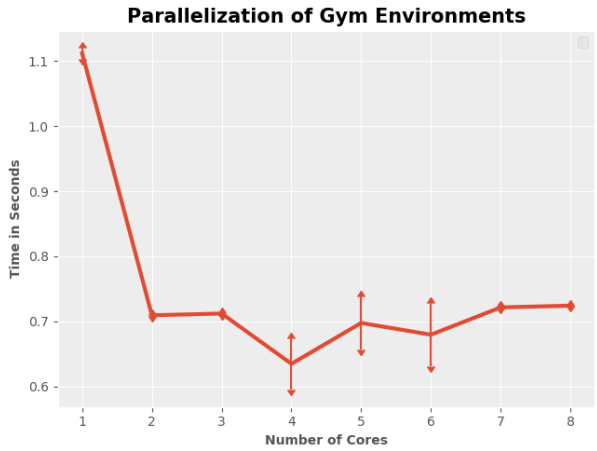
In this chapter, we talk about different implementation considerations for our environments, tasks and actions in order to aid reproducibility of our results. Another integral part of RL research is the tuning of hyperparameters. Efficient exploration of possible hyperparameters is an exploration problem on its own and we discuss our approach in this chapter as well.
The different environments introduced in the last chapter are used during the implementation and testing of the algorithm. This approach makes it easier to discern between bad hyperparameters and genuine bugs in the algorithms. Another important technique are comparisons to other implementations of the same algorithms. Different, albeit correct implementations of the same algorithm can still produce significantly deviating results, but establishing a general understanding for performance of different algorithm, parameter and environment combinations as well as comparing different intermediate results for sanity checking greatly increases the efficiency of our implementation efforts.

6.1 Performance Considerations of CartPole Environments

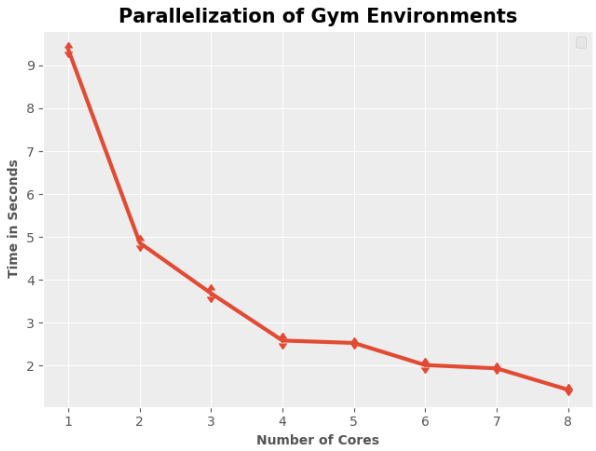
Computations for this research are conducted on the Lichtenberg high performance computer of the TU Darmstadt. As a result, we are able to repeatedly run experiments in order to increase the statistical validity of our research. One big consideration when running on the cluster computer is that the main performance gain comes from the availability of massive parallelization of execution. Profiling the execution of our algorithms, most of the computation time is spent on gathering simulation data and backpropagation, i.e. updating the neural networks involved. Individual intrinsic motivation implementations might also impact computation time significantly. Looking at modern machine learning frameworks such as PyTorch [Paszke et al., 2017], parallel execution comes as a standard feature and computing resources are adequately used in most cases. When looking at the environment simulations however, for example the concrete CartPole implementation used, not much gain can be made from having multiple computing cores available. This restriction proves to be a significant bottleneck for our learning progress. In order to improve our efficiency, we resort to running multiple simulations in parallel, effectively multiplying the amount of data we can gather by the amount of logical processors available to us apart from a small overhead. The two major problems with this approach are the following:
a) It is mostly suited for working with algorithms that sample complete episodes between updates, in our case this applies to TRPO and PPO.
b) Since episodes may vary in length, the total number of samples our algorithms use is not fully constant, though it should not vary by a large amount.
The worst case for this approach is when all episodes are short except for one. In that case, one would have minimal benefit from multiprocessing since a single episode cannot be split onto multiple cores. The benchmarks in Figure 6.1 show that in the best case, sampling time roughly halves when doubling the number of processes, which is an almost linear scaling in the amount of samples generated. In the worst case, there is still a benefit of having at least two processes. At that point, one process can sample the long episode, while the other process samples all 7 short episodes in the same time. While this result means that increasing the number of processes beyond two in the worst case has only marginal effects on sampling efficiency, it also means that we still benefit from applying this approach even in the worst case.

6.2 DQN-Specific Hyperparameter Tuning

This section is about DQN-specific hyperparameter tuning. One of the most important decisions when tuning DQN is whether to clip the Mean Squared Error to [-1.0, 1.0] or not. The advantage of this clipping is making the algorithm more stable in general by filtering outliers. The big disadvantage is that the algorithm learns slower and if a type of important state transition is rare, the algorithm might not fit properly in regards to those transitions.
The results for using DQN with and without MSE clipping are in Figure 6.2. As one can see, the 95% confidence interval is smaller for the clipped MSE. The clipped MSE variant also peaks around ∼1500 reward on average and then slowly declines towards ∼1000 reward. The variant without MSE clipping is more volatile. The confidence range is larger and the average in general is less stable. However, the overall reward gained is steadily increasing until it peaks around ∼7000 reward on average.
As it turns out, clipping the MSE hampers the ability of DQN to converge significantly for this task. While clipping stabilizes the algorithm performance even in this task, a closer look at the policies and learning progress of DQN reveals



(a) Worst case for the multiprocess sampling. One episode featuring the maximum length and seven episodes of minimum length. *CartpoleStabShort-v0*



(b) Best case for the multiprocess sampling. All eight episodes featuring the maximum length. *CartpoleStabShort-v0*

Figure 6.1.: Sampling benchmark results. Eight episodes are sampled on a multicore processor. Total number of samples is constant within each benchmark. We can see that even in the worst case, using at least two processes reduces computation time. In the best case, doubling the amount of processes almost halves the computation time needed to produce a set number of samples.

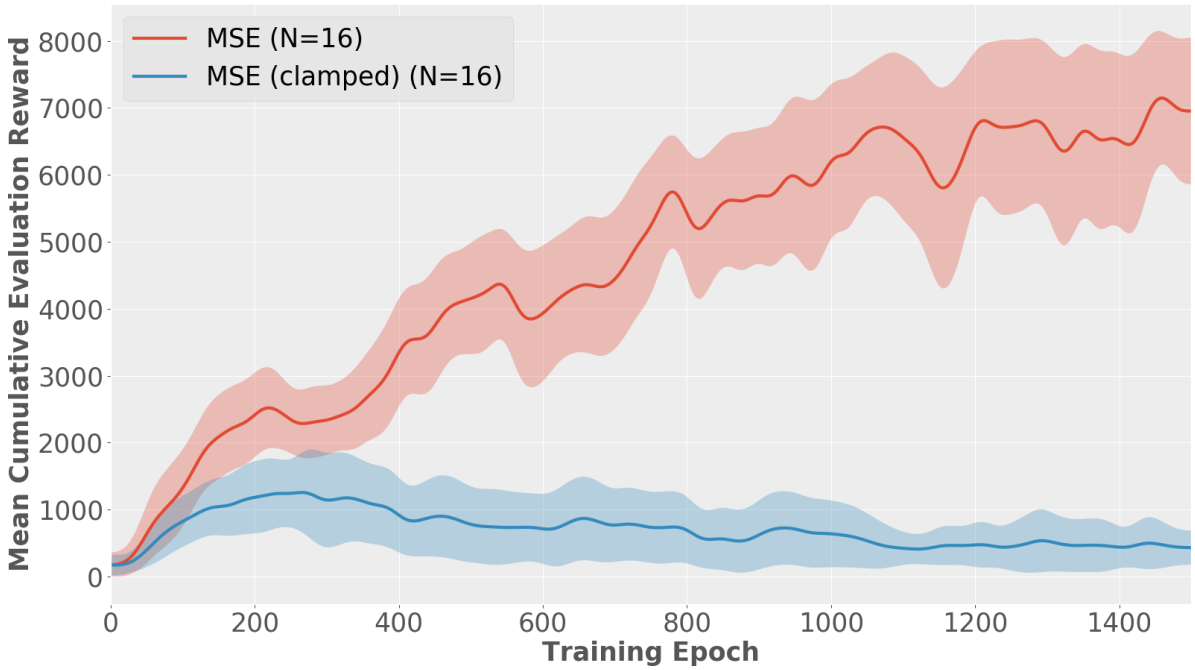


Figure 6.2.: Results for DQN on the Quanser *CartpoleSwingShort-v0* environment both with and without clipping the Mean Squared Error. One can see that clipping the error reduces variance in the runs but severely hampers the algorithm's ability to learn on the environment.

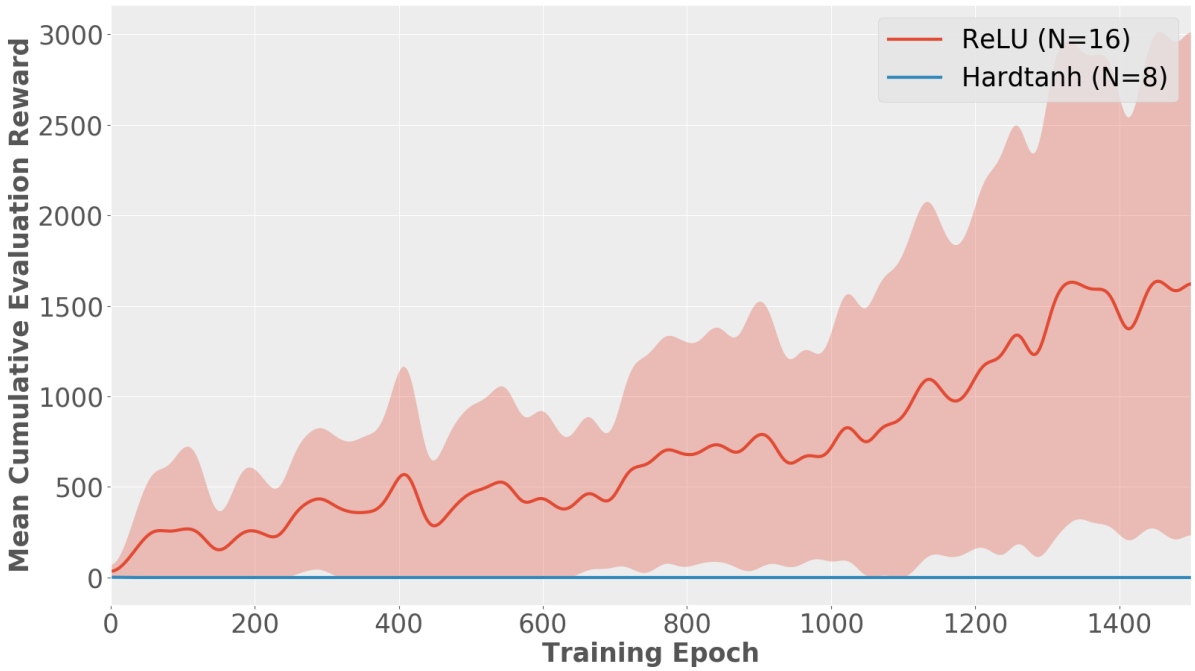


Figure 6.3.: Results for DDPG on the Quanser *CartpoleSwingShort-v0* environment with different activation functions. One can see that learning progress with a hardtanh activation function, this DDPG configuration is completely unable to learn.

the reason for the bad performance. Terminating states, such as the ones where the algorithm moves out of the allowed state space happen relatively rarely (about 1/1000). As a result, only every few updates one of these states is seen by the algorithm. In the case where we use a clipped error, since these state transitions are so rare, the Q-function approximator is never successfully fitted to reflect the lower values of these states. The result of this happening is that the MSE clipping prevents the algorithm from learning a policy that does not end episodes prematurely. The downsides of not clipping the error in comparison only make the algorithm slightly less stable and more sensitive to parameter changes in our experience. As such, we continue to use DQN without MSE clipping. For future work, investigating ways of sampling these states more often in conjunction with MSE clipping might increase algorithm performance.

6.3 DDPG-Specific Hyperparameter Tuning

Since the other algorithms generally benefited from having a clipped activation function, we also try using the hardtanh activation function for our DDPG configuration. The results can be found in Figure 6.3. In the case of DDPG, using the hardtanh function completely prevented the algorithm from learning, which is the reason we stick to the ReLU activation function for DDPG.

6.4 TRPO-Specific Hyperparameter Tuning

In this section, we present the hyperparameters that are most important in achieving stable algorithm performance for TRPO. We begin by exploring action clipping, followed by Different values for γ. Finally, we investigate the resulting policy.

6.4.1 Action Clipping

Action clipping means clipping the actions the agent applies after a certain threshold. The Quanser Benchmark Platform CartPole has a default action range of [-24.0, 24.0]. The problem with these large actions is that the actual robot might be damaged by the cart speeds one can achieve with such large actions. We clip all actions to the range of [-5.0, 5.0]. On the *CartpoleSwingShort-v0* environment, we can see that clipping the actions actually has a positive impact on our learning performance. The plot can be found in Figure 6.4.

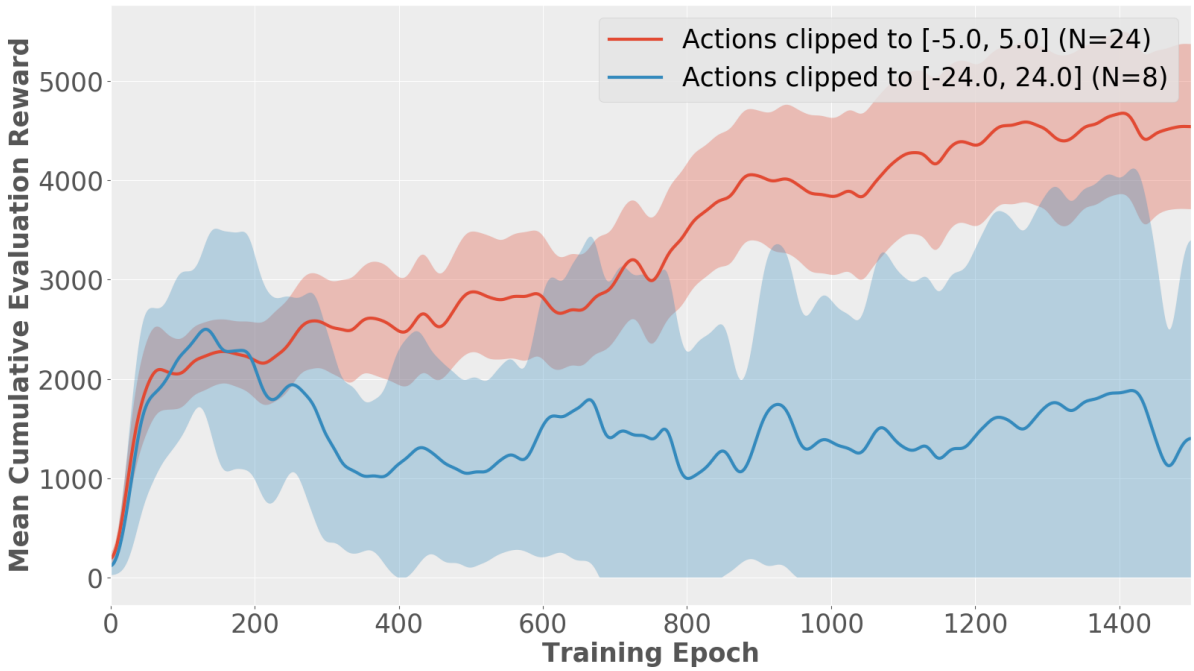


Figure 6.4.: Results for TRPO on the Quanser *CartpoleSwingShort-v0* environment with both action range being normally between [-24, 24] and clipped between [-5, 5]. One can see that larger actions do not improve the algorithm performance and in fact serve to reduce the algorithm's ability to properly learn on the environment.

6.4.2 γ and Action Repeating

γ and action repeating effectively control how far-sighted the agent acts. Higher γ means looking further into the future by reducing the decay of rewards further into the future. Higher action repeat values are technically transparent to the agent. However, if the agent looks five "logical" states into the future, and if the same agent has an action repeat of four applied, it actually looks up to twenty states into the future. In general, action repeat can be used in order to reduce the noise experienced by the agent.
These are the results for different values of γ and action repeat. Using just $\gamma = 0.99$ seems to be to short-sighted to achieve proper swing-ups at all. The same is true for just $\gamma = 0.999$ and no action repeats. The best performance is achieved with $\gamma = 0.99$ and an action repeat of 5. While not achieving stabilization, the best results are achieved using $\gamma = 0.99$ and an action repeat of 5. While not achieving swing-up and stabilization, this is the best result so far and we will stick with these hyperparameter settings. The confidence intervals are larger for the other configurations as well, however this is likely due to the fact that the number of experiments run was lower.

6.4.3 Investigating the Resulting Policy

In this section, we investigate the resulting policy. In Figure 6.6 we can see the cosine of the angle of the CartPole. On this task, the cosine of the pendulum angle is the only thing that the reward relies on. In this plot, a value of -1.0 relates to the pendulum being at the bottom (starting position) and a value of 1.0 corresponds to the pendulum being upright at the top. We can see that the agent does not manage to stabilize the pendulum, instead it swings up successfully, but then starts propelling the pendulum at ever increasing speeds, until the terminal velocity we introduced is reached shortly before the end of the episode. This result indicates that the agent has learned that spinning the pendulum fast is a good local optimum, up to the terminal velocity. It has also learned that adding too much velocity to the pendulum to quickly will end episodes prematurely and result in a reduced reward.

6.5 PPO-Specific Hyperparameter Tuning

For PPO, we start with hyperparameters strongly influenced by the information gathered during the TRPO optimization due to the general similarity of how the algorithms operate. One key hyperparameter we still need to manually tune is the error clipping value ϵ.

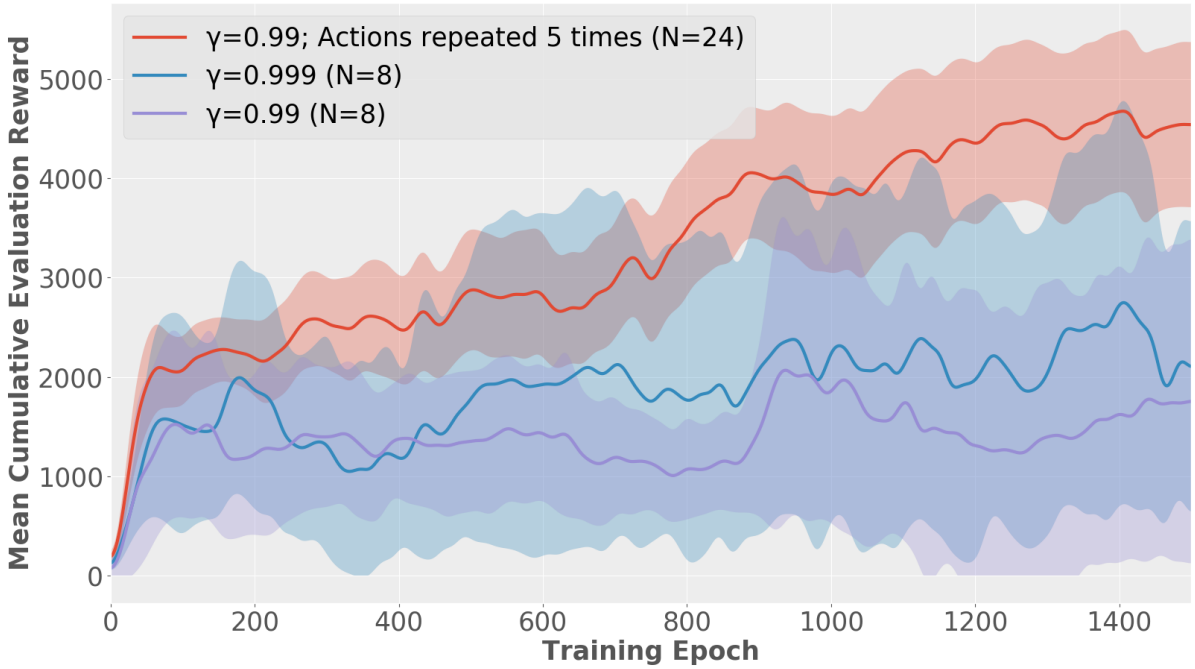


Figure 6.5.: Results for TRPO on the Quanser *CartpoleSwingShort-v0* environment with different γ values and action repeats. One can see the best performance is achieved with $\gamma = 0.99$ and repeating actions 5 times.

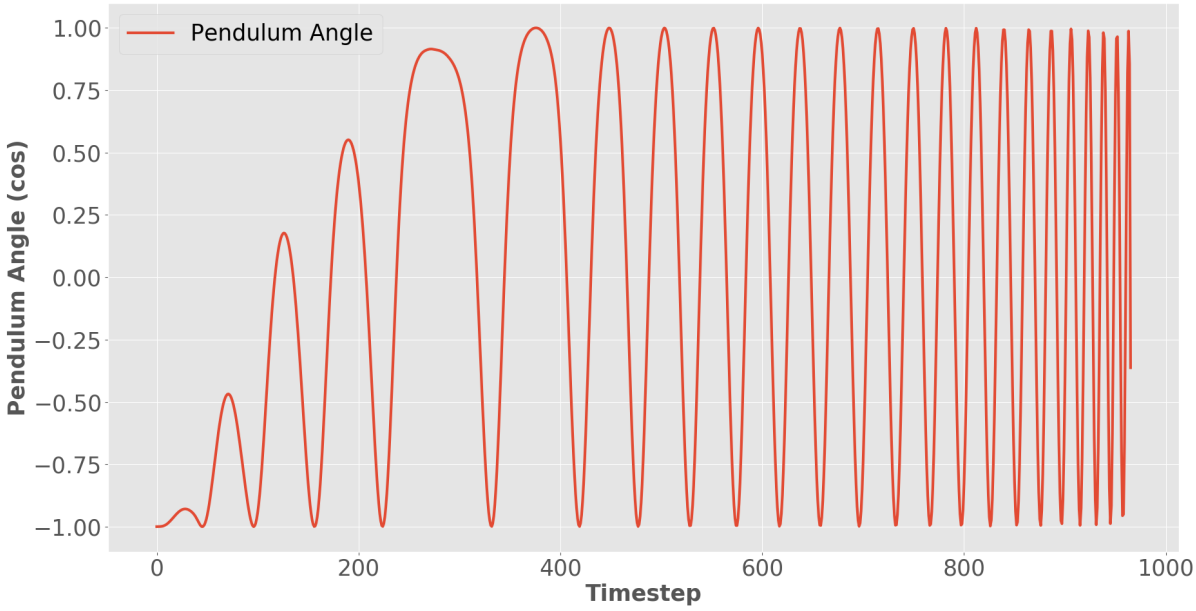
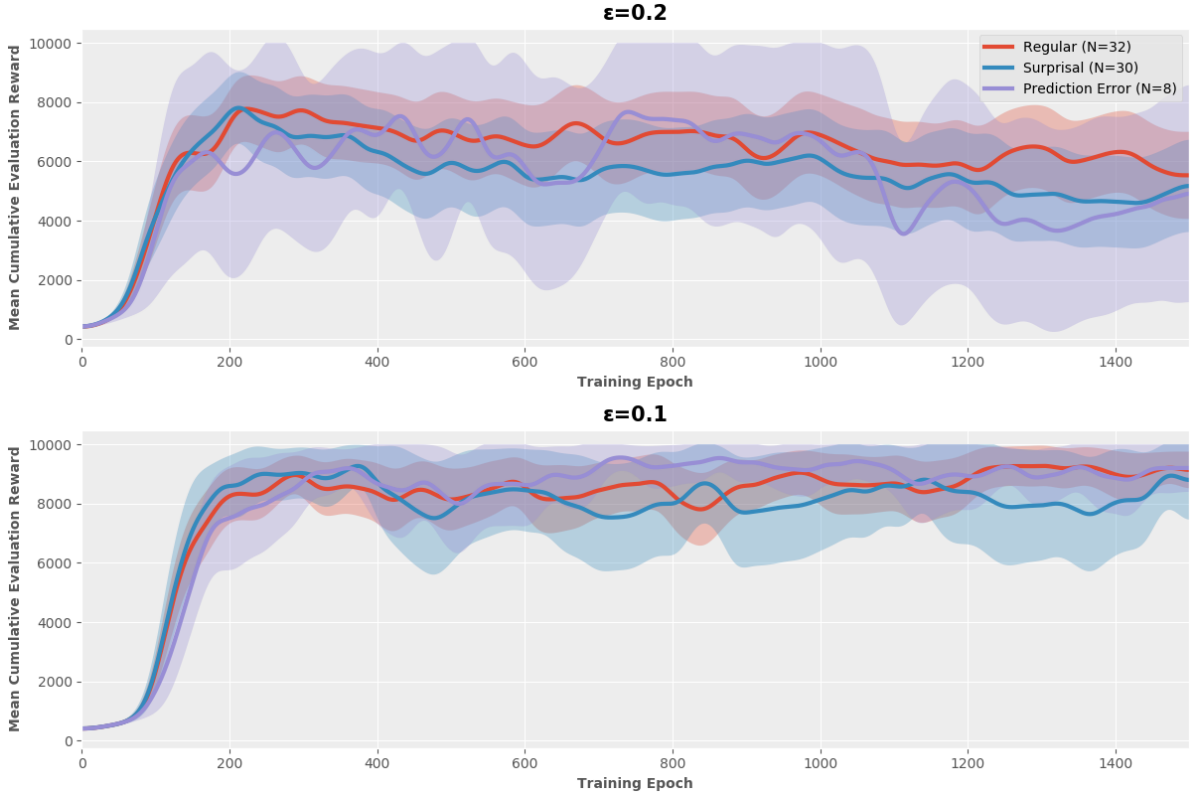


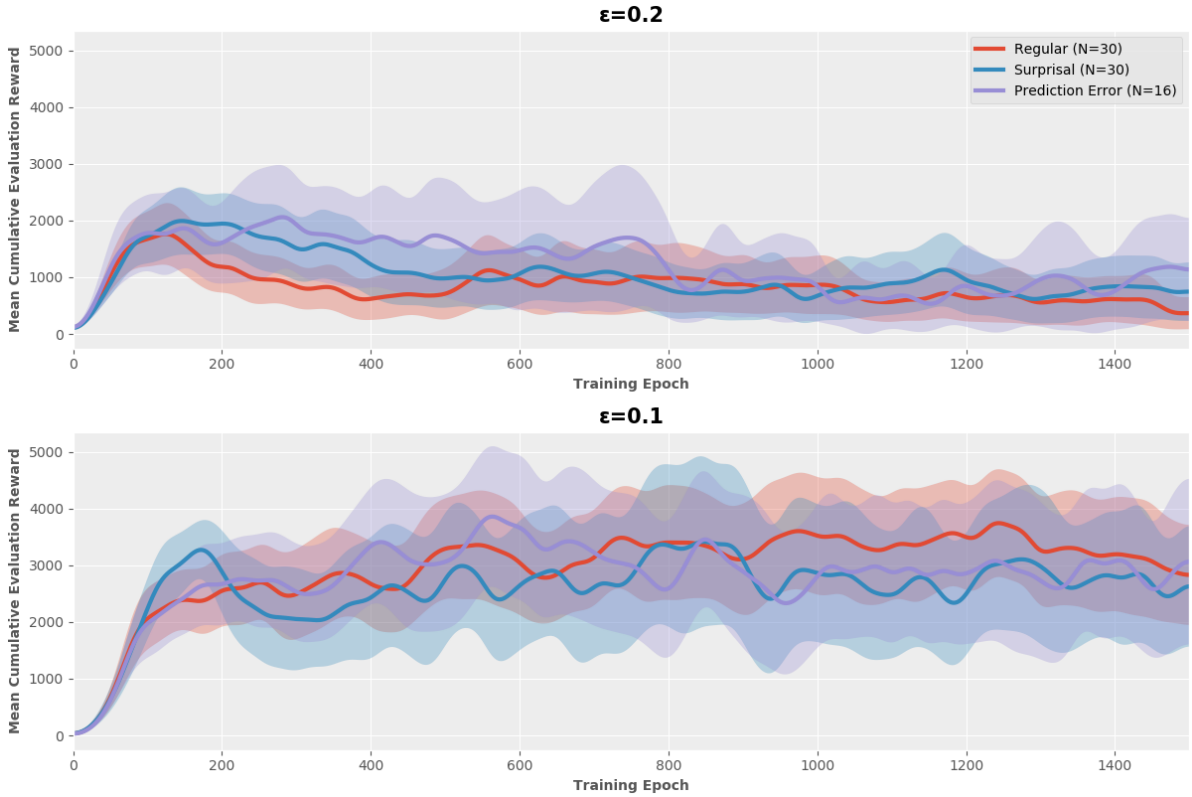
Figure 6.6.: Detailed look at the cosine of the pendulum angle of the policy TRPO learns on the Quanser *CartpoleSwingShort-v0*. One can see that the agent successfully swings up the pendulum, but does not manage to stabilize it properly. Instead, the pendulum is swung at ever increasing speeds until the episode ends. Effectively, this means that the TRPO agent is in a local optimum that it cannot escape.

6.5.1 Trust Region Parameter ϵ

The results for different values of ϵ can be found in Figure 6.7. One can see that in both the stabilization and swing-up tasks, a higher ϵ leads to marginally quicker initial learning progress. However, peak performance is lower for higher ϵ in the stabilization task and in the swing-up task a higher ϵ actually leads to performance degradation over time. While higher ϵ improve initial learning progress, they actually prevent the algorithm from reaching peak performance. This result is to be expected; a higher ϵ allows for quicker changes to the policy, explaining the fast initial learning progress. However, in the case of the stabilization task, larger policy changes actually make fine-tuning the policy harder. In the case of the swing-up task we can even see performance degradation, which indicates that the trust region is too large, which allows for policy changes that actually worsen overall performance. While in theory this configuration would help the agent escaping local optima, in this case it prevents the algorithm from reaching any meaningful performance peak at all.



(a) Different ϵ values compared on the stabilization task *CartpoleStabShort-v0*.



(b) Different ϵ values compared on the swing-up task *CartpoleSwingShort-v0*.

Figure 6.7.: Results for different ϵ values when evaluating PPO on the stabilization and swing-up tasks. Lower ϵ values decrease exploration but increase algorithm stability. In this case we can see that a higher ϵ is too unstable for the tasks we want the algorithm to perform.

7 Results

In this chapter, we present all results gathered during our research. All evaluations are run on the Lichtenberg high-performance computer of the Technical University of Darmstadt. A Conda environment definition can be found in the appendix in Chapter B. We begin by discussing the non-sparse stabilization and swing-up CartPole tasks. Afterwards, we present the results of the sparse swing-up. Lastly, we present general metrics such as the samples used and the wall-time of the different runs at the example of the non-sparse swing-up. We briefly discuss the results for each experiment before moving to the next chapter for a general conclusion.

7.1 NoInfo Stabilisation

We begin by comparing the *CartpoleStabShort-v0* (presented in Section 5.2.1) and *CartpoleStabShortNoInfo-v0* (introduced in Section 5.2.2) tasks, which only differ in their reward functions. The results can be found in Figure 7.1. DQN takes about ∼800 epochs before starting the learning progress, in the *CartpoleStabShortNoInfo-v0* while taking about ∼1000 epochs in the regular stabilization task. Overall, it manages to achieve higher rewards in the regular task. DDPG is able to achieve the perfect reward in the *CartpoleStabShortNoInfo-v0* task after about ∼500 episodes. It also manages to achieve this in the regular task towards the end of the learning process. TRPO also manages to learning quicker in the *CartpoleStabShortNoInfo-v0* task but also manages almost perfect rewards in the regular task, with a more stable learning curve in general. PPO has a larger variance in the *CartpoleStabShortNoInfo-v0* task and generally performs slightly worse until about epoch ∼1000.

Discussion

In general getting a close to perfect reward of ∼10000 is easier in the *CartpoleStabShortNoInfo-v0* task, since the only prerequisite is not ending the episode prematurely. Compared to the regular *CartpoleStabShort-v0* where one not only needs to reach the maximum episode length but also has to balance the pendulum upright instead of constantly shifting slightly left and right. Since this is the case, we do not expect our algorithms to learn to keep the pendulum in a stable position necessarily.
DQN seems to benefit from the additional information contained in the regular tasks reward. While it manages to learn some better than random policy significantly quicker with the flat reward, it has a more consistent learning progress with the regular reward.
DDPG manages to learn a nice stabilization in the sense that the pendulum is not being shifted all the time. Instead, it is very stable, even though technically there was no reward guiding it to this policy a human would consider optimal. This means that possibly the generated policy may not be stable in terms of larger deviations from this middle point.
TRPO manages to learn a consistent balancing of the pole. Looking at the policies in detail, the one learned with the more informative reward is less jittery overall, while still catching the pendulum every time it deviates from the perfect upright position.
PPO shows larger deviations between the two tasks. In contrast to TRPO, it does not manage to consistently balance the pendulum after each training epoch when using the non-informative reward.
Overall it is surprising how well the algorithms DDPG, TRPO and PPO manage to stabilize the pendulum instead of just swinging the pendulum from one side to the other.

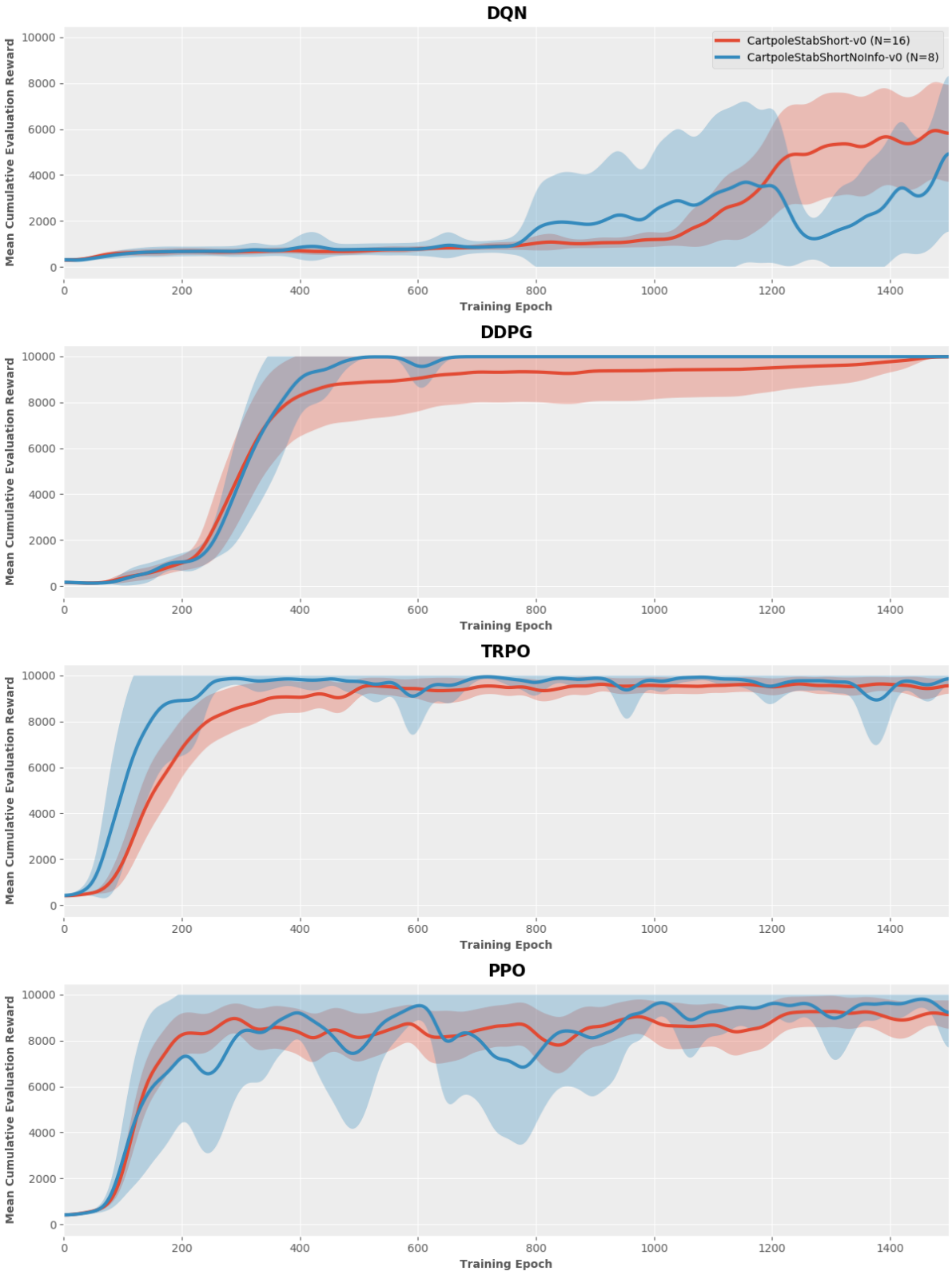


Figure 7.1.: Different algorithms on the *CartpoleStabShortNoInfo-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. This graph is based on the reward signals actually perceived by the agents. One can see that DDPG and TRPO manage to produce consistent results in both tasks, while PPO is somewhat unstable in the *CartpoleStabShortNoInfo-v0* task and DQN does not manage to learn a proper stabilization policy in the 1500 training epochs.

7.2 Stabilization with Intrinsic Motivation

In this section, the results for the CartPole stabilization task *CartpoleStabShort-v0* are presented and discussed. This variant features no sparse rewards. The results can be found in Figure 7.2. The shadows behind the lines are the 95% confidence intervals. DQN performs worse than DDPG, TRPO and PPO. DQN has a relatively flat learning curve for the first ∼600 epochs. DDPG starts the learning progress later than TRPO and PPO, though it also manages to learn proper stabilization. The *Surprisal* variant takes the longest to do so, while the *Prediction Error* manages to converge to an almost perfect stabilization after about ∼300 epochs. In general, PPO seems to start the learning progress earlier than TRPO, with and without intrinsic motivation. TRPO without intrinsic motivation performs the best and is the most stable. It manages to reach close to perfect performance after around ∼400 training epochs. TRPO with intrinsic motivation also performs well, but is on average about ∼500 reward worse and has larger error bars. PPO, despite learning quicker in the beginning, does not manage to converge to a stable policy and falls off as the training epochs progress. This effect is worsened when used in conjunction with *Surprisal*.

Discussion

DQN manages to produce some results towards the end of the training period. However, it is still the worst performing algorithm on the stabilization task. The use of intrinsic motivation based on *Prediction Error* and *Surprisal* allowed the algorithm to learn quicker in the beginning. However, *Surprisal* learns the slowest in the long-term and as such falls behind regular ϵ-greedy exploration and *Prediction Error* variants, which are both close to equal at the end of training. This result indicates that intrinsic motivation helps early exploration in this case, but does not help the algorithm converge to a better optimum. As the possible exploration in this task is rather limited, this is in line with our expectations.
DDPG performs better than expected. We can clearly see in the graph that *Surprisal* slows down learning overall while still allowing the learning of an almost perfect policy towards the end of the training session. Different hyperparameters help a quicker convergence here. We can also see that the *Prediction Error* variant performs especially well, converging after only ∼500 epochs in total.
TRPO is producing far better results than DQN and DDPG. Early learning progress is quick and once the algorithm converges towards the optimum policy, the error bars are consistent. Comparing the baseline of no intrinsic motivation, which solely relies on policy randomness for exploration, one can see that both *Surprisal* and *Prediction Error* learn quicker in the beginning. *Surprisal* then averages around ∼500 reward less than regular TRPO with larger error bars. This result indicates that the *Surprisal* overall forces the agent off of the global optimum but never manages to completely move it out of its general vicinity. The *Prediction Error* variant does not suffer from this problem. Its error is only marginally larger, and on average it seems to outperform regular TRPO, but not to a significant degree.
PPO is also performing far better than DQN and DDPG, but overall slightly worse than TRPO in this task. For PPO, the *Prediction Error* outperforms regular exploration. However, it learns slower in the beginning. The *Surprisal* variant learns quicker at first, but is all together less stable and as such performs worse than the regular variant long-term.
Overall, DDPG, TRPO and PPO vastly outperform DQN in this task. This result is probably caused by the fact that this task requires relatively little exploration overall since there are no local optima and the reward function is well-defined. The trust region methods as such can play to their strengths of guaranteeing policy improvements within certain assumptions, without the risk of getting stuck in a local optimum. It is surprising that DDPG converges very early on, especially when using the *Prediction Error* variant. Nevertheless, it is interesting to see that intrinsic motivation can still help with early exploration in a significant manner for all algorithms.

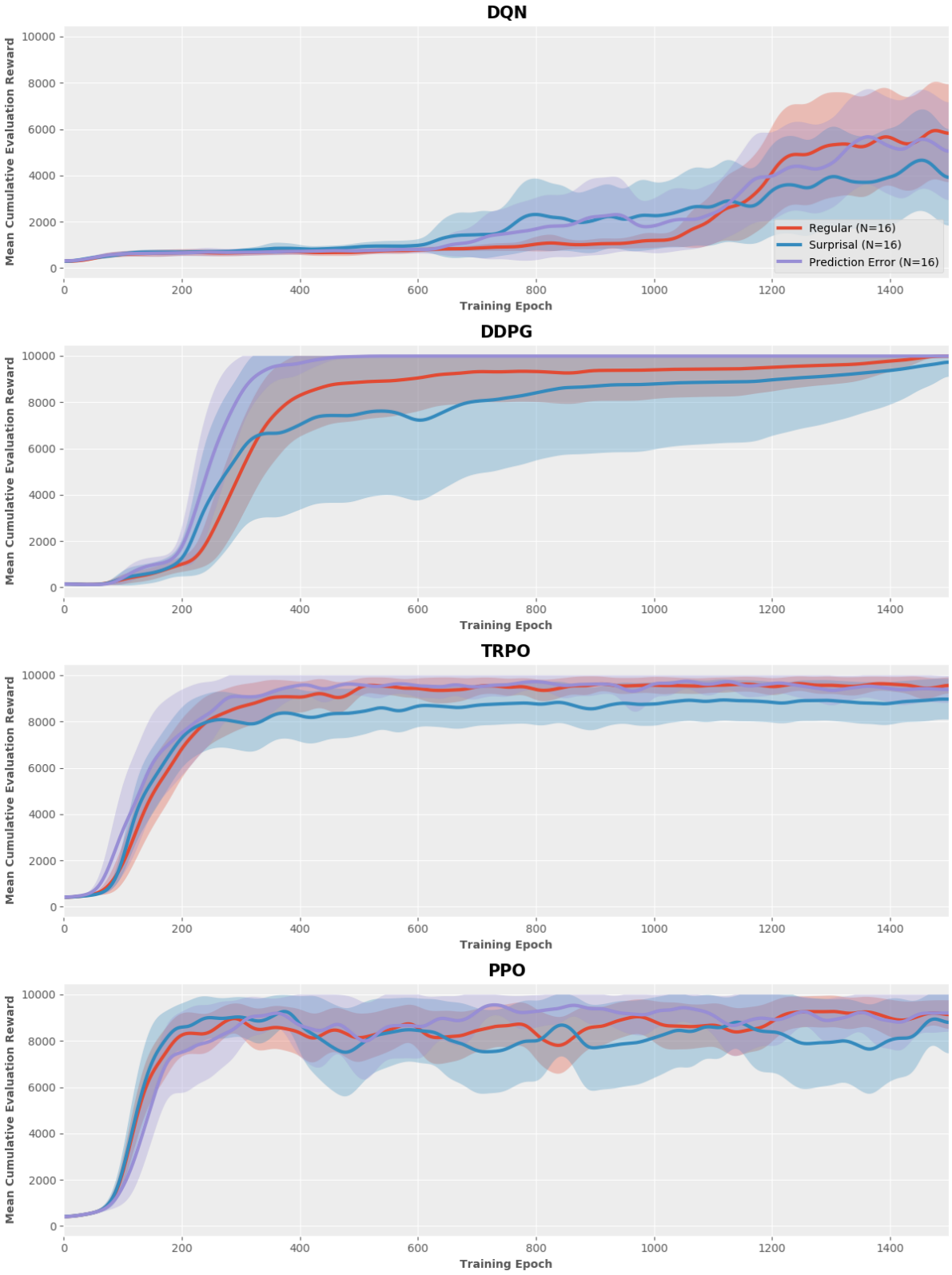


Figure 7.2.: Different algorithms on the *CartpoleStabShort-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. TRPO performs the best, followed by PPO. Intrinsic motivation has minor effects on these two algorithms and larger effects on DDPG and DQN.

7.3 Manually Shaped Reward Swing-Up

We begin the swing-up evaluations with our manually shaped reward function introduced in Section 5.2.3. The results can be found in Figure 7.3. DQN starts around ∼800 reward and then quickly climbs towards around ∼4000 reward, where it stays for the rest of the training period. DDPG does not start the learning progress until about ∼600 training epochs have passed, after which it slowly climbs up to about ∼2000 reward. TRPO manages to achieve around ∼3000 reward from the very beginning, but stays at that point for the remainder of the learning process. PPO is very similar in performance, although it starts with less variance and actually climbs to around ∼4000 reward at maximum.

Discussion

Since the theoretically highest achievable reward is ∼10000, we can conclude that none of our algorithms come close to achieving a perfect policy. DDPG might improve learning after the 1500 epochs given, but it is highly unlikely that it will manage to outperform the other algorithms. Since all algorithms converge to similar maximum rewards, we can conclude that our reward function contains a local optima around the reward values achieved by the algorithms.
A closer look at the policies reveals that they barely move the cart and as such are far from performing a swing-up. This leaves us with the conclusion that our penalty for large actions and velocities is still too punishing and as such the algorithms learn that not moving the pendulum angle by much is desirable. This leaves us with a dilemma where we would need to fine-tune these penalties exactly right. Having too little penalty would mean the shaped reward function essentially regresses towards the original reward, defeating the purpose of an enhancement. Having too much penalty resorts in the behaviour we see here. It is also possible that there is no sweet spot between the two extremes, i.e. the behaviour would shift from not doing anything to being analogous with the non-shaped reward function.

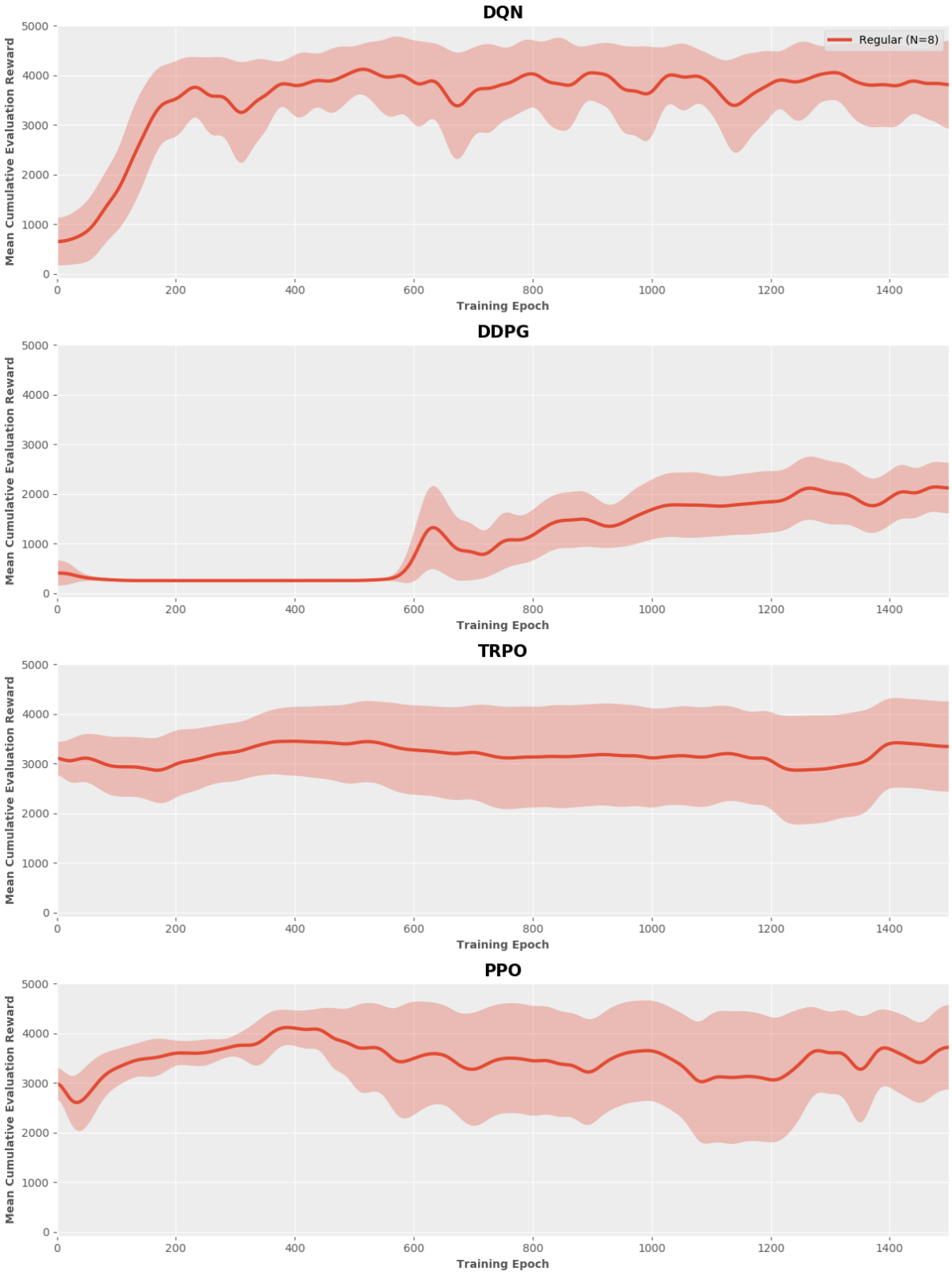


Figure 7.3.: Different algorithms on the *CartpoleSwingShortNiceReward-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. This graph is based on the reward signals actually perceived by the agents. One can see that all algorithms except for DDPG get stuck in local optima around ∼3000 to ∼4000 reward.

7.4 Swing-Up Non-Sparse

These are the results for the *CartpoleSwingShort-v0* task. This variant features no sparse rewards. The results can be found in Figure 7.2. DQN manages to convert to a result of around ∼7000 reward overall. The different intrinsic motivation variants seem to make little difference. However, *Surprisal* seems to incur more reward in a shorter amount of training epochs than the other variants.
DDPG performs worse, only reaching about ∼3500 reward at maximum. The *Surprisal* variant performs worse than regular DDPG and the *Prediction Error*. The *Prediction Error* is similar in performance to regular exploration. However, it seems to decrease in performance towards the end of the training period. TRPO with and without *Surprisal* converges around ∼4000 total reward. PPO with and without *Surprisal* is worse, starting to rise to about ∼2000 total reward and then slowly converging around ∼1000 total reward, with the *Surprisal* variant taking longer to decline. Taking a closer look at the TRPO policies, one can see that the algorithm is generally converging towards a policy where it spins the pendulum at high speeds instead of balancing, resulting in only capturing around 50% at maximum of the total possible reward. This behaviour is discussed in Section 6.4.3 in greater detail.

Discussion

In this task, **DQN** is the best-performing algorithm by a significant margin. It manages to learn swing-up and stabilization most of the times, even though it can be unstable in terms of learning progress. DQN does not manage to learn a stable balancing policy - instead the pendulum is constantly being tipped slightly left and right. This behaviour might be caused by the discrete actions DQN has available in conjunction with its generally lower ability to fine-tune policies when compared to TRPO or PPO. Nevertheless, the resulting policy manages to keep the pendulum upright most of the time. This can be seen as a weakness in the reward function: Since the cosine is used as a reward scaling based on the angle, slight shifts in angle have barely any effect on the total reward gained, as long as the pendulum stays upright.
DDPG is the worst performing algorithm. In general, it seems like the exploration of DDPG is very fragile and strongly dependent on the specific parameters of the Ornstein–Uhlenbeck process that is used for exploration.
TRPO seems to be stuck in a local optimum of spinning the pole instead of properly balancing it. This can be seen as a deficiency in exploration, which is in line with our expectation of TRPO being prone to this kind of issue. Interestingly, adding *Surprisal* or *Prediction Error* as intrinsic motivation does not yield the expected result of helping the exploration in this case. In fact, it has barely any impact on learning progress.
PPO, performs significantly worse than TRPO. Intrinsic motivation seems to slow the rate of skill decay after the initial peak around training epoch ∼100, but ultimately the performance is still within the realms of statistical insignificance as time progresses.
Overall DDPG is performing badly in this task. PPO and TRPO are suffering from getting stuck in local optima, which greatly hampers their ability to find the global optimum, a stark contrast to the stabilization task. In this task, DQN performs significantly better, since it is less prone to getting stuck in local optima and its ϵ-greedy exploration is not biased by the current policy state.

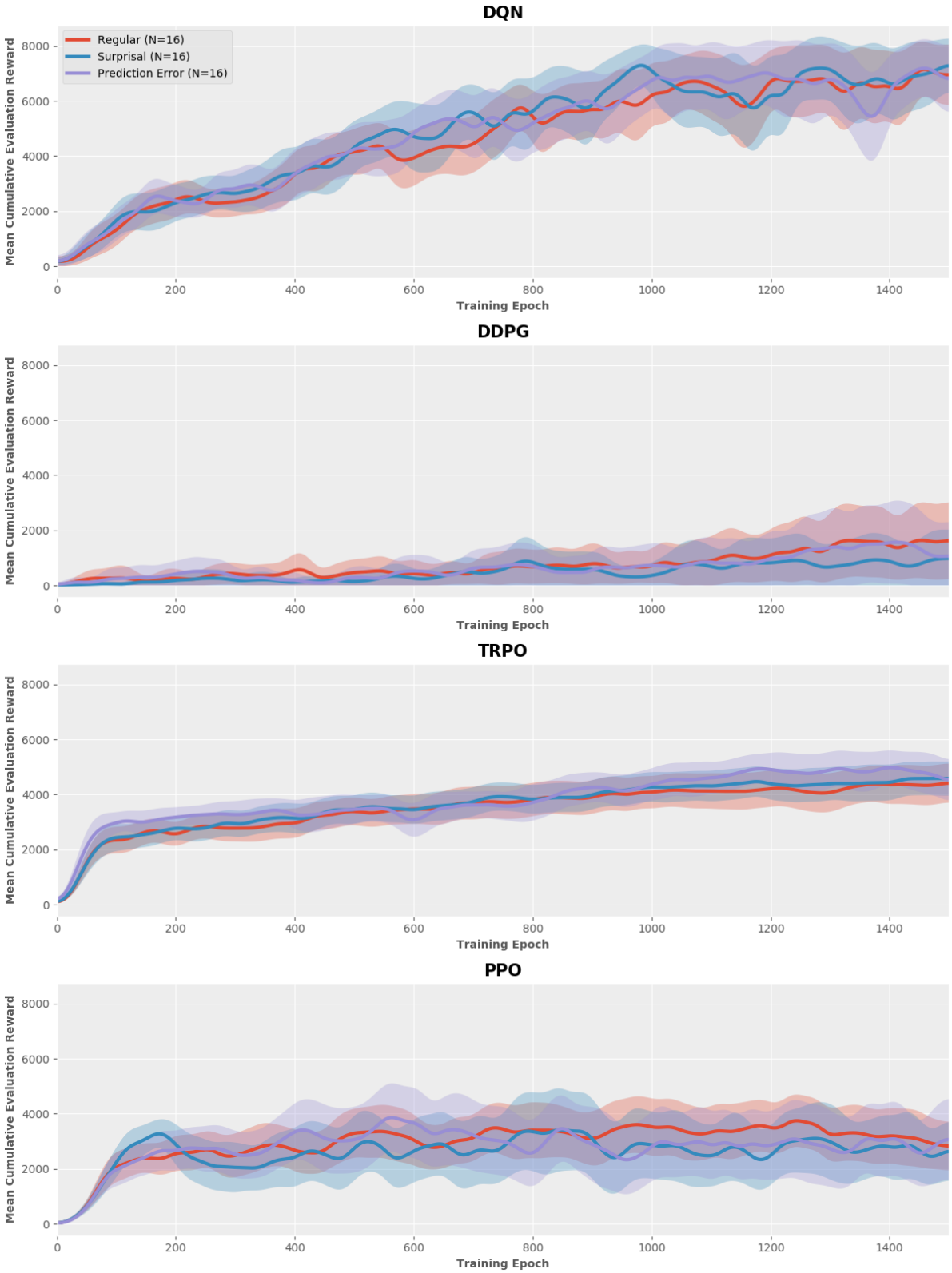


Figure 7.4.: Different algorithms on the *CartpoleSwingShort-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. In this task, TRPO and PPO get stuck in local optima. DDPG needs a lot of samples before learning progress is significant. DQN performs the best.

7.5 Swing-Up Sparse

These are the results for the *CartpoleSwingShortSparse-v0* task. This variant features no sparse rewards. The sparse rewards actually perceived by the agents are plotted in Figure 7.5. The theoretical non-sparse rewards analogous to the *CartpoleSwingShort-v0* task are plotted in Figure 7.6. Plotting these rewards as well helps us differentiate between policies that do nothing and policies that do show useful behaviour but do not manage a full swing-up. Some additional plots of median values can be found in appendix Chapter C.
DQN barely managed to find any rewards before the ∼200 epoch mark, but does not start to incur any meaningful rewards until about training epoch ∼800. Towards the end of the training period, even regular DQN manages to find some actual rewards, but overall less than ∼200 reward. The *Surprisal* variant actually manages around ∼400 reward and the *Prediction Error* peaks at around ∼700 reward at the end of the training period. Looking at the theoretical non-sparse reward, one can see that DQN is actually producing some useful behaviour from the very beginning of the training, probably due to its ϵ-greedy exploration.
DDPG does not manage to produce any meaningful behaviour at all, even when looking at the non-sparse reward. DDPG without intrinsic motivation just drives the cart outside of the allowed state range using maximum actions. However, while the same is true for *Surprisal*, some policies apply constant actions of around 2.5 instead of the maximum of 5 Volt. The policies learned using the *Prediction Error* apply mostly 0 Volt as actions and thus do not produce any rewards at all either.
TRPO both without intrinsic motivation and using *Surprisal* does not manage to find any actual rewards during training. TRPO with *Prediction Error* actually shows a promising learning curve. However, looking at the individual runs shows that only a single run out of 16 managed to show learning progress at all, which increases the average score significantly. Looking at the theoretical non-sparse reward, TRPO without intrinsic component still does not manage to produce rewards of more than ∼15. Using *Surprisal*, one can see that TRPO manages to score a very consistent amount of reward around ∼75 during almost all runs. The variant using the *Prediction Error* shows learning progress early and then stays around the ∼500 reward mark.
It is important to note that PPO without intrinsic motivation component manages to find a little reward only to decline again immediately towards the final training episodes. Looking just at the actually perceived rewards, *Surprisal* is performing only marginally better. The first perceived reward using the *Prediction Error* occurred after about ∼400 training epochs, peaking around ∼800 and then declining to close to 0 around the ∼1000 training epochs mark.
Looking at the theoretical rewards that would have been received by the agents in a non-sparse setting, one can see that PPO without intrinsic motivation is producing around ∼20 reward throughout training runs, even though it never manages to receive any actual rewards. *Surprisal* and *Prediction Error* start with similar learning curves. However, *Surprisal* converges to a lower reward of around ∼250 towards the end of training. The *Prediction Error* variant continues to hover around ∼500 reward with a larger confidence interval until the end of the training period.

Discussion

Producing any actual received reward in the sparse setting directly correlates to learning a successfully swing-up of the pendulum at least once.
DQN is the best-performing algorithm on the sparse CartPole by a significant margin. This statement is also true for the variant without intrinsic motivation, which even outperforms other algorithms that use intrinsic motivation. The variant using *Surprisal* is slightly better than regular DQN. The best results are achieved by DQN together with the *Prediction Error* Method. However, since it looks like DQN just starts learning progress when the limit of 1500 epochs is reached, we start another evaluation run for DQN with a limit of 3000 epochs, the results of which we discuss in the next section.
DDPG is not producing any useful behaviour at all. For the variant that does not use any intrinsic motivation, which is in line with our expectations. When using intrinsic motivation such as *Surprisal* or the *Prediction Error*, the algorithm does not produce better results, either. This fact leads to our conclusion that the Ornstein–Uhlenbeck process that is used for exploration with DDPG is apparently not suited to being extended using intrinsic motivation in this task.
TRPO, both with and without intrinsic motivation does not manage to perform a successful swing-up at all during our evaluation except for one run with the *Prediction Error* intrinsic motivation. We expected this result for the setting without intrinsic motivation, as exploration is not guided. However, it is less expected for the variant using *Surprisal*. Inspecting the theoretical non-sparse reward reveals a very consistent reward of ∼75, which is surprising since the intrinsic motivation should serve to decrease consistency. Looking at the produced policies, this confusion is cleared up. The agent consistently learns policies that simply accelerate the cart into a single direction until it crashes into the simulation bounds. In this case, this means the *Surprisal* has failed to help exploration and instead keeps the agent in a local optimum. This phenomenon can be explained by the way *Surprisal* works. As a form of Learning Progress Motivation, rewards are gained whenever the agent manages to find states where learning progress can be made. Once the agent is trapped in a policy space that consistently produces similar states, over time the network will fit a very

accurate model, resulting in little to no intrinsic rewards that could guide the agent out of said local optimum. In this sense, *Surprisal* fails to help agents move out of local optima.

PPO performs less consistent but better overall. PPO without any external guidance even manages to find a single policy that manages a single swing-up. Looking at the non-sparse reward, PPO without *Surprisal* component is still worse than the crashing TRPO policy. When using PPO with *Surprisal*, one can see that it manages to swing-up the pendulum after about ∼200 episodes. The peak average is around ∼300 reward at episode ∼800. After that, the agents decline in performance again to around ∼75 reward. This result suggests that after finding any rewards at all, the agent is consistently able to find rewards, while not getting stuck in a local optimum. Unfortunately, this result means that it leaves the optimum it found and then never manages to find a better policy again. Looking at the theoretical non-sparse rewards, one can see an average of about ∼500 reward starting from episode ∼200 all the way up to epoch ∼1200, at which point it starts to decline. When comparing this development to the purely sparse rewards, which started to decline around ∼200 epochs earlier, this result suggests that the agent started exploring again. During this ∼200 epoch long process, while the sparse rewards declined, the overall useful behaviour of the agent did not decline. After that point, the theoretical reward stayed consistent with the actually perceived reward in the sparse setting but overall useful behaviour declined.

Overall, DDPG does not produce any meaningful results. TRPO and PPO suffer from sticking to their trusted regions, which makes quick exploration of the state space difficult. DQN is able to explore the state space the best by a significant margin.

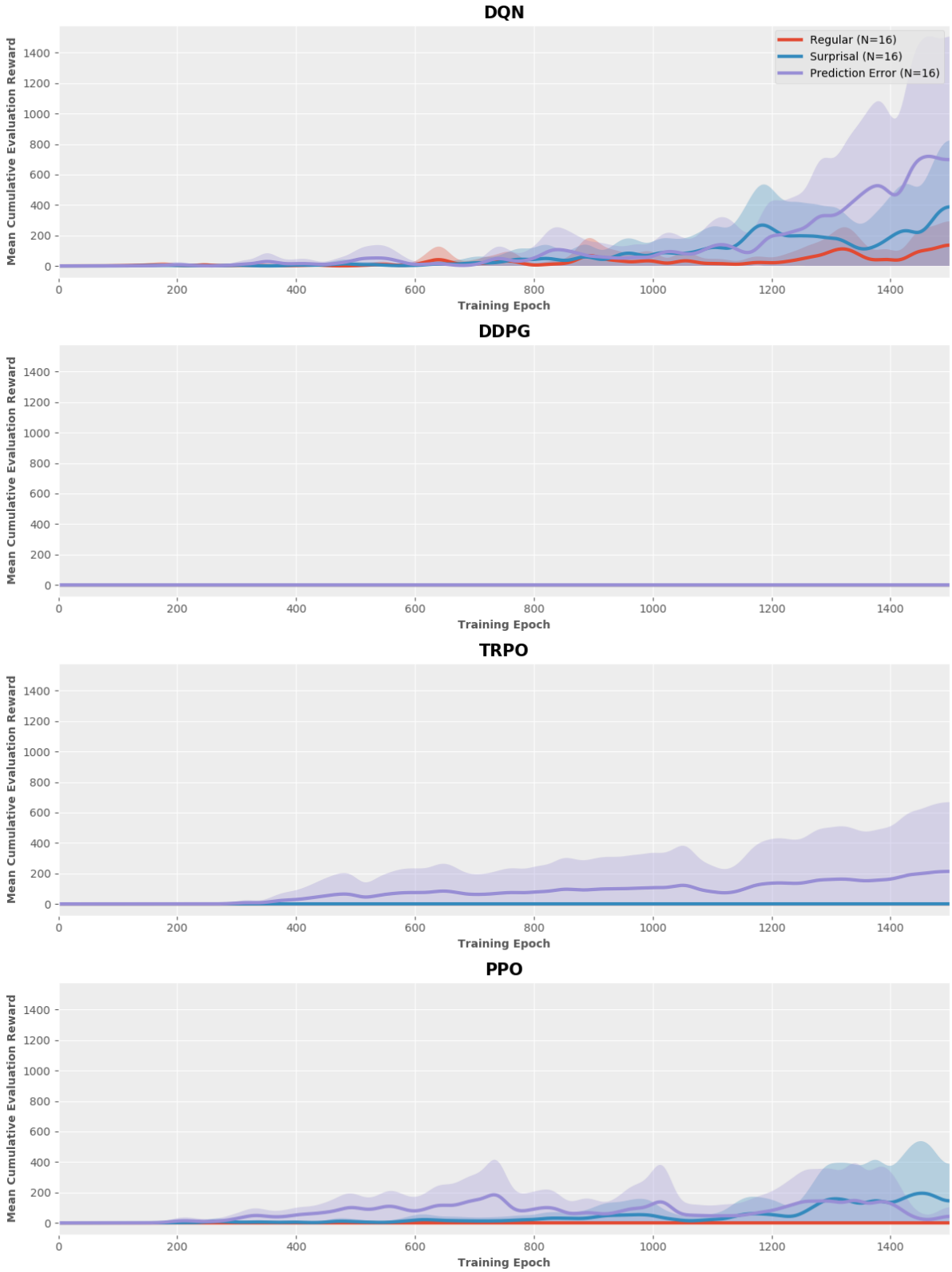


Figure 7.5.: Different algorithms on the *CartpoleSwingShortSparse-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. This graph is based on the reward signals actually perceived by the agents. One can see that DDPG, TRPO and PPO show almost no complete swing-ups. DQN performs the best.

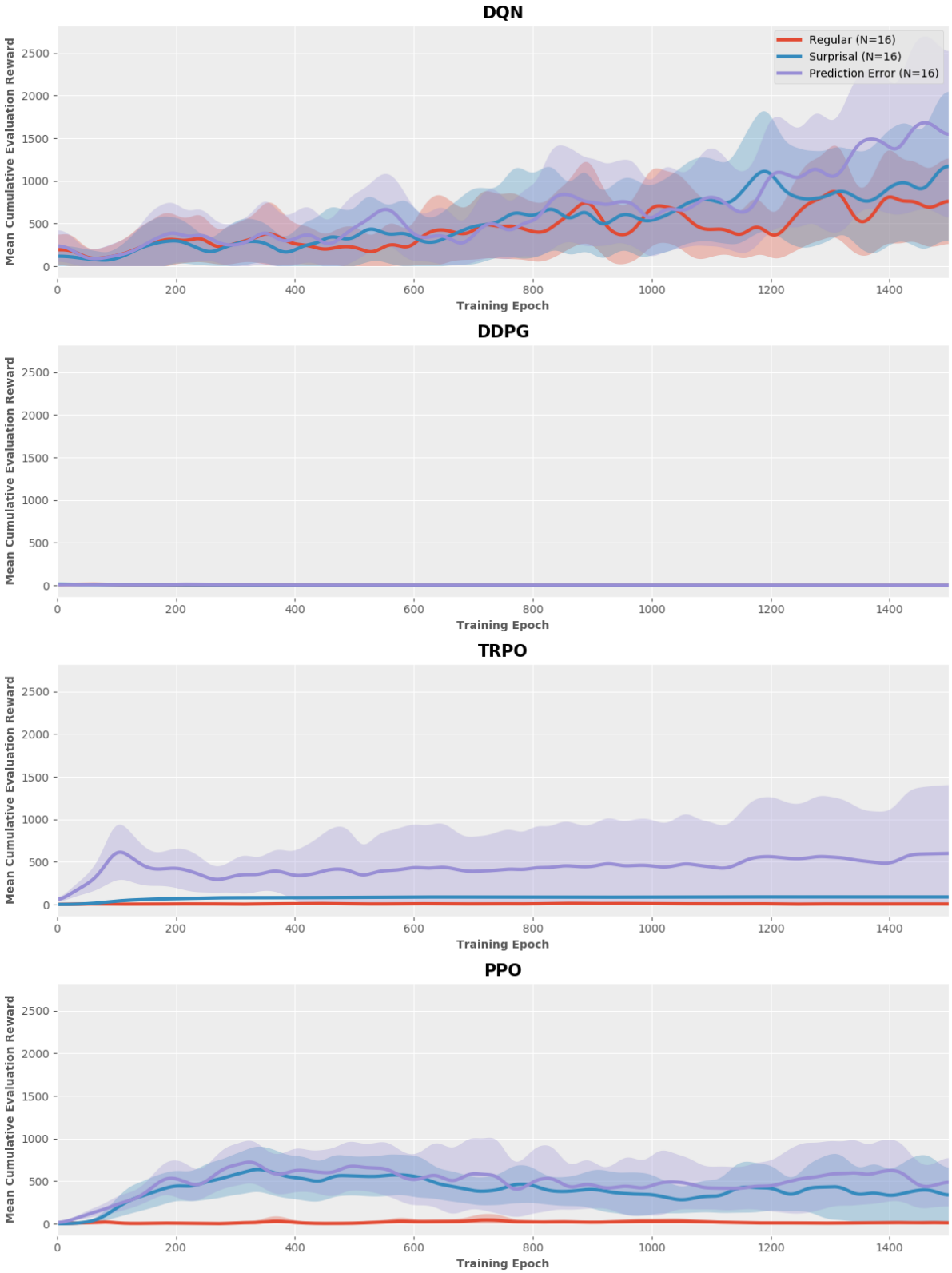


Figure 7.6.: Different algorithms on the *CartpoleSwingShortSparse-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. This graph is based on the reward signals that would have been perceived in the non-sparse setting. One can see that DDPG still shows no useful behaviour at all. PPO using *Surprisal* shows some useful behaviour in contrast to the previous Figure.

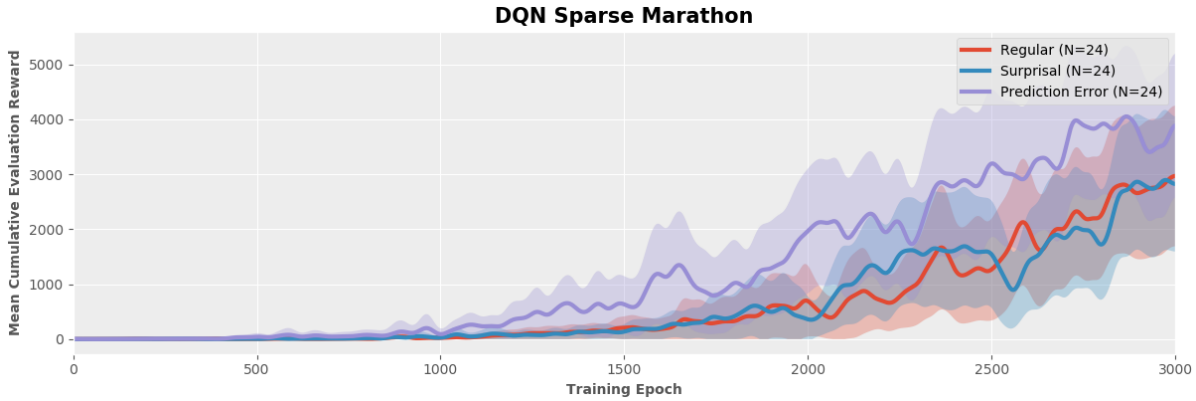


Figure 7.7.: DQN on the *CartpoleSwingShortSparse-v0-marathon* task. This graph is based on the reward signals actually perceived by the agent. One can see that the *Prediction Error* manages to outperform both regular TRPO and the variant using *Surprisal*.

7.6 Swing-Up Sparse Marathon DQN

In this section we discuss our findings when giving DQN more training epochs in the sparse settings. The epoch limit is now 3000. Everything else is configured analogous to the previous section. The results can be found in Figure 7.7. We can see that the *Prediction Error* starts actual learning progress for the first time at around epoch ∼500. Nevertheless, it takes until about episode ∼1000 before any meaningful shift in the mean cumulative reward occur. Both *Surprisal* and regular DQN take until about ∼1000 episodes before before any shifts in reward can be seen and until about epoch ∼1400 before learning progress starts in a meaningful way. Afterwards, the curves for *Surprisal* and regular DQN are similar in variance and average return. All algorithm variants continue on an upwards slope until epoch ∼3000. The *Surprisal* and regular TRPO variants peak at around ∼3000 reward at the end of the training session, while the *Prediction Error* variant manages to collect around ∼4000 reward.

Discussion

These test results show *Surprisal* working similarly well to regular ϵ-greedy exploration, which is invalidating the results of the previous sections which suggested DQN might benefit from *Surprisal* in this task. In contrast to this, the *Prediction Error* variant is showing better performance than the other two approaches. All three approaches have not converged to a fixed reward level at the end of 3000 epochs. It is possible that they would eventually converge to the same results, nevertheless we would expect the *Prediction Error* variant to arrive at such a value first. It is important to note that while *Prediction Error* learns a better policy on fewer samples, it does incur large costs in the wall-time duration of learning and is as such most useful when gathering samples for learning is time-consuming in itself.

7.7 Computation Time

The results for the computation time of the different algorithms can be found in Figure 7.9. **DQN and DDPG** without any intrinsic motivation are the quickest algorithms by a significant margin. However, using intrinsic motivation significantly slows them down, since both algorithms perform a lot of updates and for each algorithm update, we also need to predict and update the intrinsic motivation variants. The *Prediction Error* has a far bigger impact on the computation times of these algorithms than *Surprisal*. **TRPO and PPO** are very similar in computation time. They both take about twice as long as DQN and DDPG. However, adding intrinsic motivation has a lesser impact than for the other algorithms. Since both TRPO and PPO only perform 1500 updates each, significantly less time is needed to predict and update the intrinsic motivation modules. For these two algorithms, *Surprisal* has less impact on computation than the *Prediction Error*.

7.8 Amount of Samples

The amount of samples used by each algorithm can be found in Figure 7.8. One can see that both **DQN and DDPG** use around ∼150,000 samples each over the 1500 training epochs. **TRPO** uses the most samples, about ∼1,000,000 in total. It is important to note that even though it uses around six times as many samples as DQN and DDPG, it only takes about twice the computation time as is established in the previous section. This effect is most likely caused by the parallel sampling introduced in Section 6.1. **PPO** uses about ∼600,000 to ∼800,000 samples overall. It is interesting to note that more samples are used during the early training epochs. PPO is also the only algorithm where we can see that an intrinsic motivation variant, in this case *Surprisal*, uses less samples overall. Seeing as the *Surprisal* variant offers similar performance in terms of rewards gained, this is a positive result.

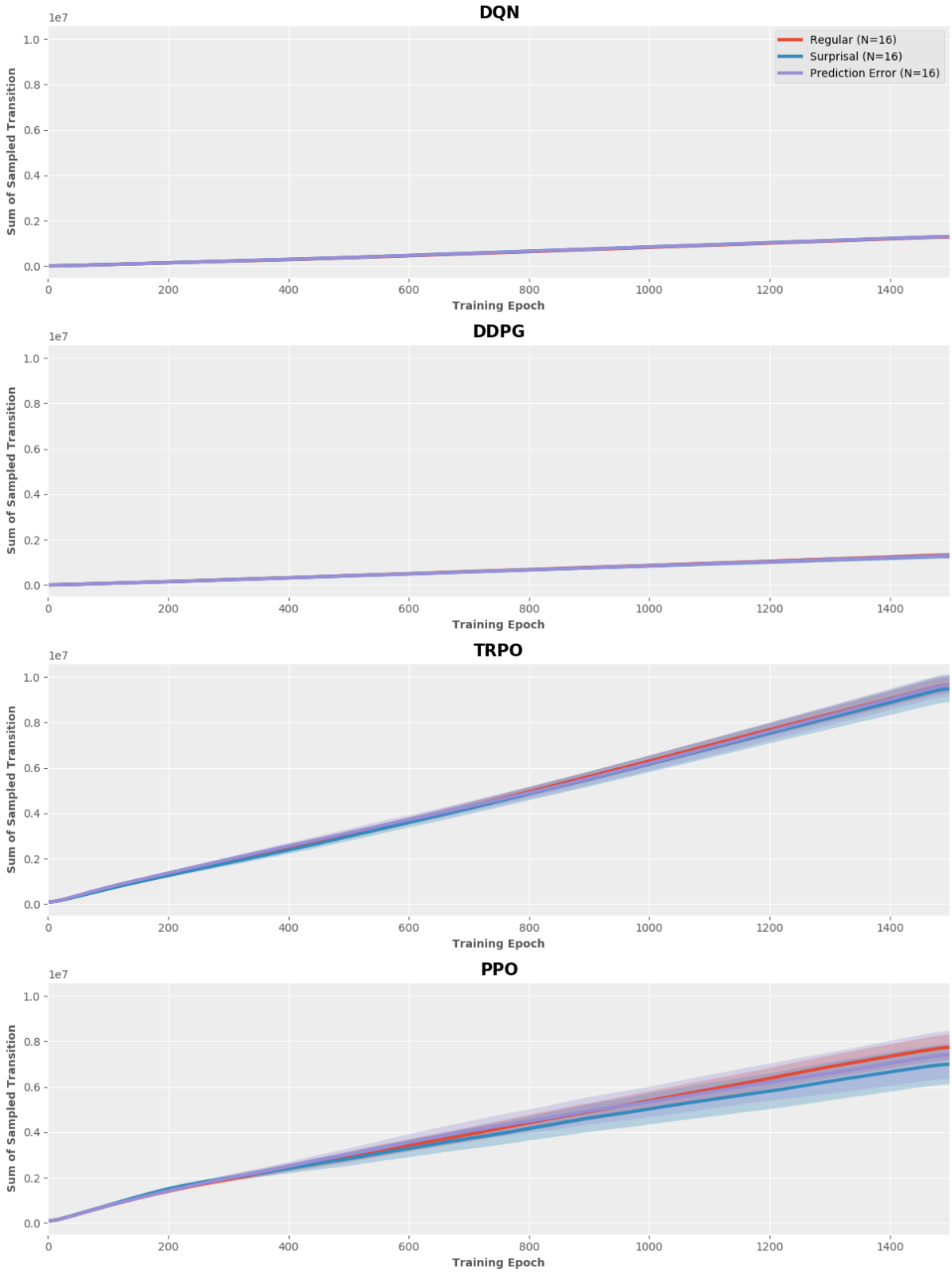


Figure 7.8.: Different algorithms on the *CartpoleSwingShortSparse-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. This graph shows the total number of transitions sampled during learning. One can see that DQN and DDPG use vastly less samples than TRPO and PPO.

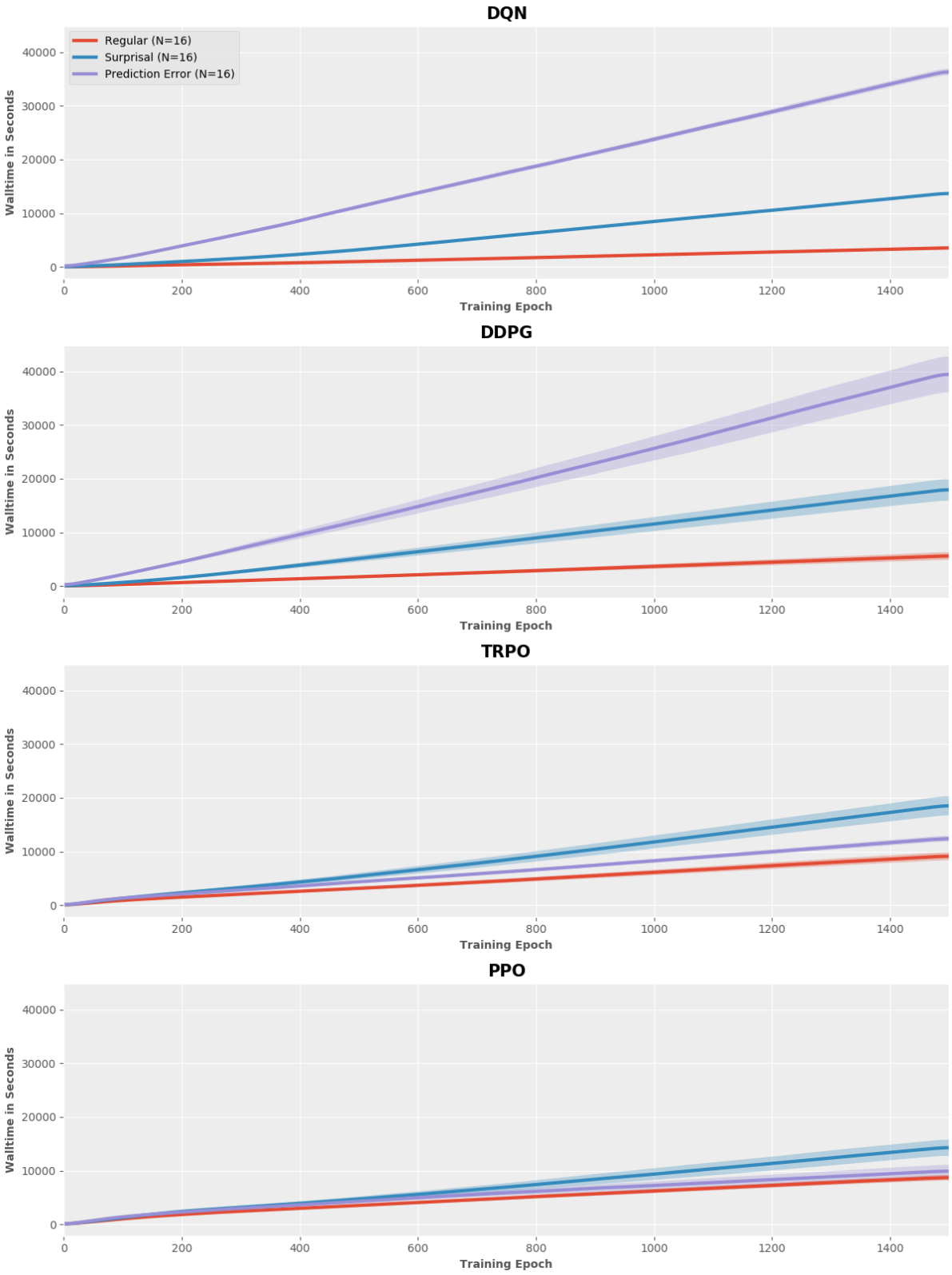


Figure 7.9.: Different algorithms on the *CartpoleSwingShortSparse-v0* task. Top to bottom: DQN, DDPG, TRPO and PPO. This graph shows the total wall-time the algorithm execution took. TRPO and PPO take about twice as long than DDPG and DQN without any intrinsic motivation. The *Prediction Error* impacts computation time of DQN and DDPG a lot, while *Surprisal* has less impact. The situation is reversed for TRPO and PPO.

8 Discussion

During the research for this thesis, we aimed to investigate how intrinsic motivation can be used to improve performance on the Quanser Reinforcement Learning Benchmark Systems in both sparse and non-sparse settings. We implemented several new tasks in the CartPole environment to aid our research. We also enabled parallel sampling of OpenAI Gym environments so that computation can take place on a multiprocessing scale. In order to properly discuss our results and put our work into context, we begin by answering the questions we established in our motivation.

The first question we wanted to answer is how well do the different algorithms perform on the Quanser Reinforcement Learning Benchmark Systems. Especially DQN has problems on a task like the CartPole stabilization, which requires fine-tuning of policies. This task is where DDPG, TRPO, and PPO excel, with TRPO being slightly better than PPO. All algorithms manage to learn a swing-up of some form in the non-sparse setting. However, DDPG, TRPO, and PPO do not learn to swing-up and stabilize the pendulum at the same time, which DQN manages. Nevertheless, the stabilization policy is suboptimal.

The second question we wanted to answer is if intrinsic motivation can help to guide agents out of local optima. In the non-sparse settings, we are not able to effectively guide agents out of local optima. However, for some algorithms the learning process is faster when using intrinsic motivation. A notable example of this is DDPG when using *Prediction Error* intrinsic motivation.

The third question we wanted to answer is if intrinsic motivation can effectively guide an agent to explore systems where rewards are sparse. In our research, intrinsic motivation did change the agent's behaviour significantly in a sparse environment. DQN seems to be the most benefiting from this exploration, while DDPG did not produce any useful behaviour even when paired with intrinsic motivation. DQN was able to perform some useful behaviour even in a sparse environment, which is improved by adding intrinsic motivation. TRPO and PPO went from barely any useful behaviour towards performing at least some exploration, which is overall positive but far from learning proper swing-up and stabilization, which is also difficult for them in non-sparse settings.

The fourth question we wanted to answer is how intrinsic motivation affects the convergence properties of these agents. With our chosen parameters, the agent's convergence properties were somewhat improved with *Prediction Error* in cases such as TRPO and DDPG on the stabilization. *Surprisal* seems to slow down convergence overall in terms of samples needed before a stable point is needed. This proves to be a constant trade-off: Having higher scaled intrinsic rewards negatively impacts convergence while lower scaled intrinsic rewards reduce the benefits of intrinsic motivation harshly.

The fifth question we wanted to answer is how intrinsic motivation affects the computational complexity of the algorithms. Overall, there is a significant impact on computational complexity for some intrinsic motivation variants. For DQN and DDPG, *Surprisal* roughly doubled overall computation time and the *Prediction Error* took roughly seven times longer than the regular algorithms. TRPO and PPO, on the other hand, are hardly impacted by the *Prediction Error* variant. *Surprisal* roughly doubles the runtime of TRPO and impacts the computation time of PPO by 50%.

The final question we wanted to answer is how the different RL algorithms compare to each other. DQN and DDPG are very sensitive to parameter tuning. Slightly wrong parameters can prevent the algorithms from converging at all. Since we add additional noise to the observations from the perspective of the algorithm, this tendency makes using them with intrinsic motivation difficult. DDPG is especially impacted negatively by this problem. TRPO and PPO suffer from the inverse problem. They are in general very stable algorithms that converge to local optima nicely, even if in our tests TRPO proved to be slightly more stable than PPO. However, this property negatively impacts their ability to explore state spaces, even when supporting them in this regard with intrinsic motivation.

Based on our results, we come to the conclusion that the right forms of intrinsic motivation can help the convergence of RL agents significantly by guiding them towards interesting parts of the state space. A specific example of this is DQN with the *Prediction Error* using significantly less samples before learning progress starts to show.

However, using intrinsic motivation comes at a cost of higher computational complexity, which invalidates the advantage of needing less samples depending on how expensive it is to gather them. On top of that, proper tuning of the parameters is necessary in order to ensure that convergence happens as quickly as possible. Looking at the exploration properties in sparse settings, we were able to show that the algorithms TRPO and PPO were able to produce useful behaviour only when paired with intrinsic motivation. Notably, DQN managed to explore the sparse setting even without intrinsic motivation, though slower.

8.1 Outlook

Further research should be conducted towards the most promising approach using DQN, for example by using different, possibly adversarial strategies for sampling from the replay buffer. Further research into the exploration properties may potentially explain some of the quirks found but is not advised as a result of the generally poor performance and high sensitivity to parameter and environment changes. TRPO and PPO prove to be highly stable and suited for optimizing existing policies. As such, approaches that use the exploration properties of an algorithm like DQN and then transfer the resulting policy to algorithms such as TRPO and PPO might be promising. Additional computational resources could also be used to benchmark the results on different tasks.
Reward shaping proved to be as difficult as expected, as such further research into intrinsic motivation is advised. During our research, we mostly focused on enhancing the existing rewards. Different approaches such as automated curriculum learning and adversarial learning show promising results for tasks such as Atari games, where learning progress is difficult to measure and models are harder to learn.

Bibliography

[Qua, 2019] (2019). Quanser innovate educate. https://www.quanser.com/. Accessed: 2019-09-30.

[spi, 2019] (2019). Trust region policy optimization. https://spinningup.openai.com/en/latest/algorithms/trpo.html. Accessed: 2019-09-30.

[Achiam and Sastry, 2017] Achiam, J. and Sastry, S. S. (2017). Surprise-based intrinsic motivation for deep reinforcement learning. *ArXiv*, abs/1703.01732.

[Baranes and Oudeyer, 2013] Baranes, A. and Oudeyer, P.-Y. (2013). Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robot. Auton. Syst.*

[Bellemare et al., 2016] Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*.

[Belousov et al., 2020] Belousov, B., Muratore, F., Abdulsamad, H., Eschmann, J., Menzenbach, R., Eilers, C., Tosatto, S., and Lutter, M. (2020). Quanser robots. https://git.ias.informatik.tu-darmstadt.de/quanser/clients.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *ArXiv*, abs/1606.01540.

[Fournier et al., 2018] Fournier, P., Sigaud, O., Chetouani, M., and Oudeyer, P.-Y. (2018). Accuracy-based curriculum learning in deep reinforcement learning. *ArXiv*, abs/1806.09614.

[Legault, 2016] Legault, L. (2016). Intrinsic and extrinsic motivation. *Encyclopedia of Personality and Individual Differences*.

[Lillicrap et al., 2015] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N. M. O., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *ArXiv*, abs/1509.02971.

[McDuff and Kapoor, 2018] McDuff, D. J. and Kapoor, A. (2018). Visceral machines: Risk-aversion in reinforcement learning with intrinsic physiological rewards. In *ICLR*.

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

[Nguyen et al., 2011] Nguyen, S. M., Baranes, A., and Oudeyer, P. (2011). Bootstrapping intrinsically motivated learning with human demonstrations. *2011 IEEE International Conference on Development and Learning, ICDL 2011*.

[Oudeyer et al., 2007] Oudeyer, P., Kaplan, F., and Hafner, V. V. (2007). Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*.

[Oudeyer and Kaplan, 2007] Oudeyer, P.-Y. and Kaplan, F. (2007). What is intrinsic motivation? a typology of computational approaches. *Frontiers in Neurorobotics*.

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747.

[Ryan and Deci, 2000] Ryan, R. M. and Deci, E. L. (2000). Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary Educational Psychology*.

[Schulman et al., 2015a] Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P. (2015a). Trust region policy optimization. In *ICML*.

[Schulman et al., 2015b] Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to reinforcement learning*. MIT press Cambridge.

[Uhlenbeck and Ornstein, 1930] Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Phys. Rev.*

A Hyperparameters and Algorithm Tweaks

This chapter contains all hyperparameters used as well as information about data normalization and algorithm adaptations.

```
  trpoSwingupNormal: &trpoSwingupNormal
    algorithm: "trpo"
    env_name: "CartpoleSwingShort-v0"
    step_number: 1
    max_steps_per_episode: 5000
    max_episodes: 1500
    action_space: [-24.0, 24.0]
    episodes_per_update: 10
    actor_shape: [64, 64]
    l_rate_critic: 0.001
    optimizer: "Adam"
    gamma: 0.99
    activation_actor: "ReLU"
    activation_critic: "ReLU"
    alpha: 0.8
    max_j: 10
    delta: 0.01
    train_v_iters: 80
    lambda_: 0.97
    damping: 0.01
    l2_reg: 0.001

  ddpgswingupRARE:
  algorithm: "ddpg"
  action_space: [-5.0, 5.0]
  batch_size: 200
  steps_between_frozen_network_updates: 5000
  replay_storage_size: 50000
  l_rate_actor: 0.0001
  l_rate_critic: 0.001
  optimizer: "Adam"
  gamma: 0.99
  activation: "Hardtanh"
  target_network_update_factor: 0.001
  steps_between_updates: 10

dqnswingupRARE:
  algorithm: "dqn"
  batch_size: 128
  steps_between_frozen_network_updates: 5000
  replay_storage_size: 50000
  epsilon: 0.1
  actions: [-5.0, -1.0, 0.0, 1.0, 5.0]
  l_rate: 0.001
  optimizer: "Adam"
  gamma: 0.99
  activation: "Hardtanh"
  steps_between_updates: 1
  criterion: "MSE"
  steps_between_updates: 10
```

```
ppoSwingupNormal3:
  algorithm: "ppo"
  action_space: [-5.0, 5.0]
  episodes_per_update: 8
  actor_shape: [64, 64]
  l_rate_critic: 0.001
  optimizer: "Adam"
  gamma: 0.99
  activation_actor: "Hardtanh"
  activation_critic: "ReLU"
  train_v_iters: 80
  lambda_: 0.95
  l2_reg: 0.001
  weight_decay_actor: 0.0001
  policy_epochs: 4
  eps_ppo: 0.1
  mini_batch_size: 64
```

B Conda Environment

This is the Conda environment file used for our research. Additionally, one needs to install the quanser clients from [Belousov et al., 2020]

```
name: jw-masterthesis
channels:
  - vpython
  - defaults
  - pytorch
dependencies:
  - matplotlib=3.0.1
  - numpy=1.16.2
  - numpy-base=1.16.2
  - pandas=0.24.1
  - pip=19.1.1
  - pyqtgraph=0.10.0
  - python=3.6.6
  - pytorch=1.3.1
  - scikit-learn=0.20.2
  - scipy=1.2.1
  - vpython=7.5.0
  - statsmodels=0.10.1
  - pyyaml=5.1.2
  - pip:
    - gym==0.12.0
    - pyinstrument==3.0.1
```

C Median Plots

This chapter contains some additional results where we plot the median of run evaluations instead of the combination of average and 95% confidence interval.

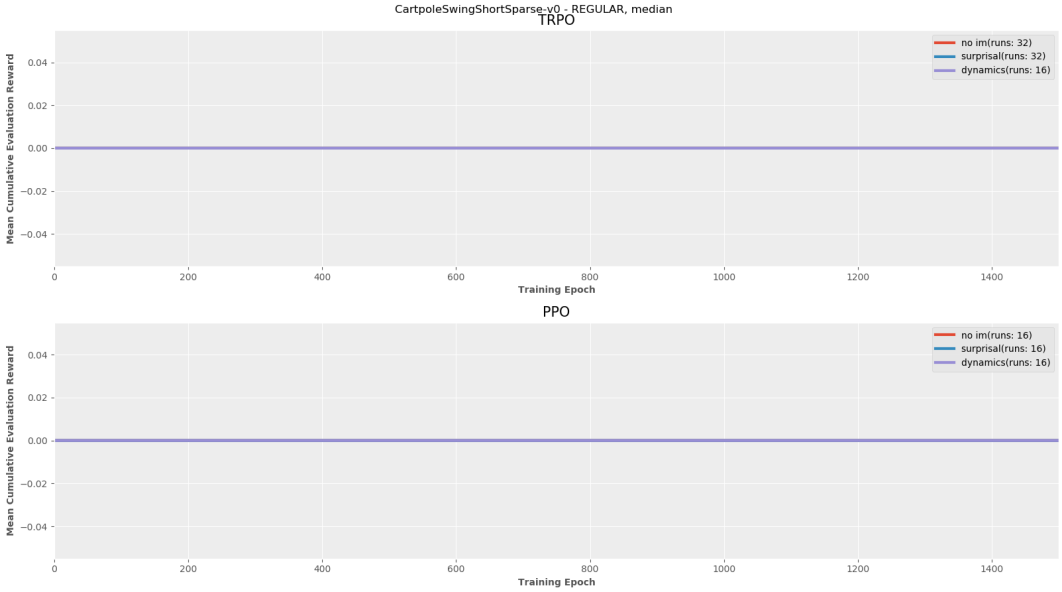


Figure C.1.: Different algorithms on the *CartpoleSwingShortSparse-v0* task, median comparison. One can see that when looking just at the median of the sparse reward function, TRPO and PPO did not manage to produce useful behaviour.

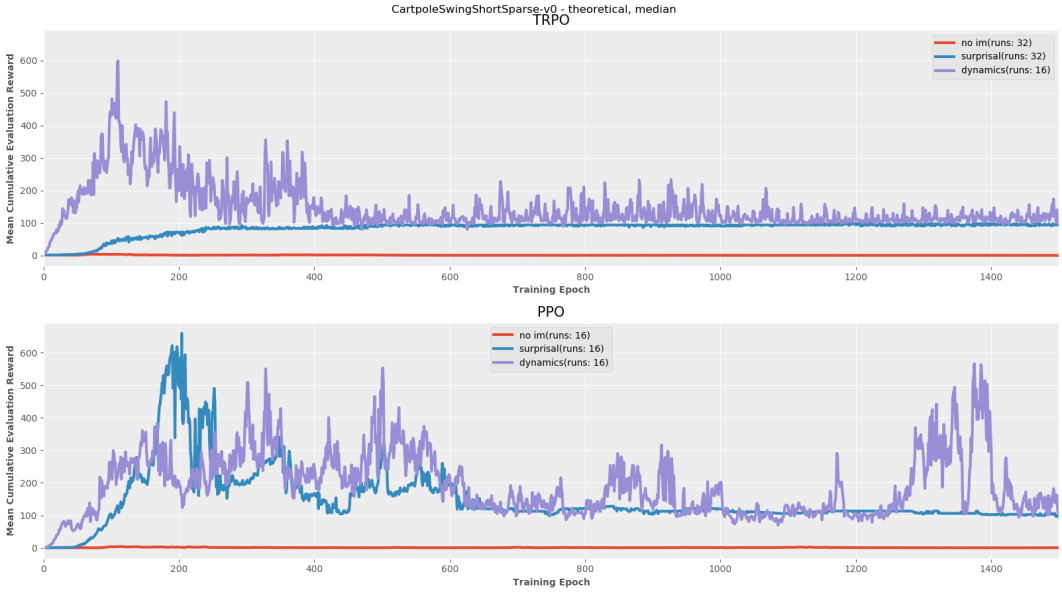


Figure C.2.: Different algorithms on the *CartpoleSwingShortSparse-v0* task, median comparison of the non-sparse reward function. One can see that TRPO and PPO did produce some useful behaviour.