Faculty of Aerospace Engineering

Flexible Curriculum Learning for Deep Reinforcement Learning Agents

J.L. Dorscheidt

November 23, 2018

Delft University of Technology

Challenge the future

Flexible Curriculum Learning for Deep Reinforcement Learning Agents

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering
at Delft University of Technology

J.L. Dorscheidt

November 23, 2018

Faculty of Aerospace Engineering · Delft University of Technology

Delft University Of Technology
Department Of
Control and Simulation

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"Flexible Curriculum Learning for Deep Reinforcement Learning Agents"** by **J.L. Dorscheidt** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: November 23, 2018

Readers:

dr.ir. E. Van Kampen

Preface

Originally my thesis would have been called 'Hear And Avoid for Micro Aerial Vehicles', the topic I chose to graduate on about one and a half year ago. Choosing a topic for my thesis was an easy process, at least I thought so. I wanted to include deep learning in my thesis, and so the first time I was handed a topic that included deep learning, I immediately accepted it. About 2 weeks later something felt wrong with my topic. Maybe it was the fact that deep learning was already familiar to me , already foreseeing the pitfalls of that topic from the beginning onwards. Maybe it was the fact that supervision was marginal and I was heavily looking to feedback on my approach. But after a month of struggling with these feelings, I decided to quit my original research topic. I had to start all over again, but I had understood what was wrong with my original topic, it could not win my enthusiasm because I would be working with techniques that were already familiar to me.

So the search for a research topic had to start over again. I send a mail to Erik-Jan because reinforcement learning had caught my attention. We scheduled a meeting and within a week I had a new research topic, for which the contents can be found in this report. Reinforcement learning was entirely new to me. With fresh enthusiasm I delved into reinforcement learning, and supervision from Erik-Jan was excellent. Weekly meetings with Erik-Jan where we would review my progress have been of incredible value to me. I am very grateful for the amount of time he has put into helping me with my research.

I would like to thank my family, who have always supported me in times I was struggling with my thesis. Finally I would like to thank Tom, Joost and Luc of geronimo.ai for the many times they offered me the flexibility to work on my thesis. I am looking forward to the first years of our start-up, working with some of my best friends on some of the most interesting machine learning problems.

Contents

List of Symbols

α	Learning rate
δ	Temporal difference error
ϵ	Exploration probability
γ	Reward discount factor
λ	Bootstrap factor
$\mathcal{N}$	Normal distribution
μ	Policy network
ω	Angular rate
π	Policy
σ^2	Variance
τ	Critic and policy network update rate
θ	Angle
$\mathbf{a}$	Action
$\mathbf{C}$	Critic network
$\mathbf{c}$	Friction constant
$\mathbf{G}$	Expected sum of rewards
$\mathbf{g}$	Gravitational constant
$\mathbf{l}$	Length
$\mathbf{m}$	Mass
$\mathbf{P}$	State transition probability
$\mathbf{Q}$	Action-Value function
$\mathbf{R}$	Reward probability
$\mathbf{S}$	State
$\mathbf{V}$	Value function

Acronyms

DDPG	Deep Deterministic Policy Gradients
DQN	Deep Q Network
GLIE	Greedy In The Limit With Infinite Exploration
IID	Independent and Identically Distributed
LIDAR	Light Detection And Ranging of Laser Imaging Detection And Ranging
LLR	Local Linear Regression
LSTM	Long Short Term Memory
MCPL	Monte Carlo Policy Learning
MDP	Markov Decision Process
MLAC	Model Learning Actor Critic
RL	Reinforcement Learning
SARSA	State Action Reward State Action
TD	Temporal Difference
TRPO	Trust Region Policy Optimization
UAV	Unmanned Aerial Vehicle

Chapter 1

Introduction

1-1 Background

in Reinforcement Learning (RL), an agent attempts to learn optimal policies by interacting with the environment. Every time the agent performs an action on the environment, a reward is received, as shown in figure 1-1.

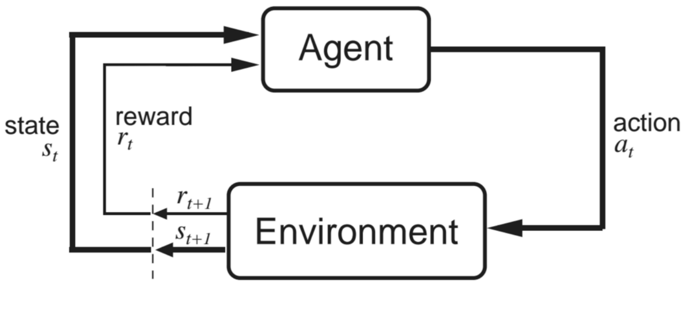


Figure 1-1: Schematic representation of an agent that interacts with it's environment. The agent receives a reward r_t coupled to the new state s_t that it transitions to because it performed action a_t. Taken from Sutton and Barto [2012].

One example of reinforcement learning is to search for a policy that finds the quickest route through a maze, the reward can be structured to be -1 for each step taken, and 100 for reaching the target position. This is where the elegance of reinforcement learning lies; it can develop a complex policy based on a simple reward function. In contrast with dynamic programming, reinforcement learning is a model free method that only has access to the states in an environment [Sutton and Barto, 2012]. Model-free algorithms offer ways of solving policies where the model might be too complex to derive analytically, or where the model might change over time. These properties of reinforcement learning offer very little restrictions on the problems they can be applied to. In comparison to more classical control

Figure 1-2: An example of a quadrotor. Autonomous flight of quadcopters is a domain that could benefit from reinforcement learning. Taken from DJI [2018]

.

algorithms, reinforcement learning offers great flexibility on what states can be used as input, and no model is required. Many applications in the aerospace sector could benefit from these properties. An example of this could be to use reinforcement learning for using camera data as state input for control tasks for Unmanned Aerial Vehicle (UAV)'s. Autonomous flight of quadrotors is an area that could benefit from such an implementation.

However sampling the environment implies that convergence to the optimal policy takes a long time, and many, often costly interactions with the environment have to be taken. Consequently reinforcement learning is seldom implemented in real-life applications [Silver, 2015]. Research in the reinforcement learning community focuses largely on reducing this sample complexity. Promising research is that of flexible heuristic dynamic programming [Helmer et al., 2018]. In [Helmer et al., 2018] the authors propose that learning might be faster when the agent is first trained using a subset of the states and actions. After this initial training stage, the agent is trained on all the states and actions. Analogous to learning to play the piano, a student first learns to play with one hand before it starts learning to play with both hands. The algorithm developed in [Helmer et al., 2018] modified the Model Learning Actor Critic (MLAC) algorithm from [Grondman, 2015] in a way that it is able to use flexible function approximators. Local Linear Regression (LLR) was used as a flexible function approximator. The final experiment consisted of training a quadcopter for stable hover mode in a flexible way, first the quadcopter was trained using a limited set of states and actions, after which the full set of states and actions was made available to the agent. Implementing the newly devised flexible heuristic dynamic programming algorithm showed that the flexible training case converged faster then the normal training case. However, it was noted by the author that the comparison was relying heavily on hyperparameter selection, therefore no solid conclusion could be made on whether the flexible approach is a better approach. The novelty of this new approach opens a door for different research possibilities.

1-2 Research objective

One of the limitations of the research in [Helmer et al., 2018] is the use of local linear regression as a function approximator. Local linear regression offers a limited way of generalization. A function approximator that is well known for it's generalizing capabilities are neural networks. Furthermore, in previous years, a lot of research has focused on using neural networks as deep function approximators in reinforcement learning [Mnih et al., 2013], [Lillicrap, 2016]. This master thesis is centered on combining the benefits of flexible function approximators with deep reinforcement learning. Therefore the research objective will be the following:

- *Reduce the sample complexity of reinforcement learning by learning in a flexible way with deep neural networks as global function approximators.*

The research questions that have to be answered in order to come to reach the research objective are the following:

1. What is the state-of-the-art in curriculum or transfer learning and deep reinforcement learning.
    - What is the state-of-the-art in deep reinforcement learning.
    - What is the state-of-the-art in curriculum or transfer learning in neural networks.
2. What is flexible reinforcement learning.
    - How can flexible reinforcement learning be formalized.
    - In which ways can problem be made flexible, e.g. how can they be categorized.
    - What are possible problems and benefits of flexible reinforcement learning in those cases.
3. What reinforcement learning architecture would be best suited for implementing neural networks as a flexible function approximator.
    - What are the criteria for a flexible function approximator.
    - What are methods for making deep neural networks flexible function approximators.
    - Of the existing deep reinforcement learning architectures, what limitations do they impose on the deep neural networks.
    - What limitations do neural networks impose on the reinforcement learning architectures.
4. What is the performance of deep flexible reinforcement learning on different environments.

1-3 Research approach

The hypothesis in this research is that using a flexible way of curriculum learning could lead to a smaller sample complexity in deep reinforcement learning agents. Recent work on flexible reinforcement learning [Helmer et al., 2018] shows that learning by gradually increasing the state and action space of the agent can lead to a smaller sample complexity in certain environments. In addition, recent work on preventing catastrophic forgetting, and curriculum learning in neural networks shows that knowledge can be retained and reused on new tasks [Bengio, 2009], [Rusu et al., 2016], [Krueger and Dayan, 2009]. The first research question is focused on exploring recent work in deep reinforcement learning, curriculum and transfer learning in neural networks. It should work as a guide through the many algorithms used in reinforcement learning and should result an overview of architectures used in deep reinforcement learning. In addition, research on the state-of-the-art in curriculum learning and transfer learning is helpful in multiple ways. It gives a theoretical background on curriculum learning, why curriculum learning might be a better way of learning, and how curriculum learning can be implemented in neural networks.

The second research question should give more insight on why a flexible way of learning would result in a lower sample complexity. In addition flexible reinforcement learning needs to be formalized, and a categorization of different forms of flexible reinforcement learning should be made in combination with advantages and disadvantages of those variants. When advantages and disadvantages are identified, different environments can be constructed that are able to test these advantages and disadvantages. A library will be used for construction of environments.

The third research question is the core of the research, the answer should consist of a reinforcement learning algorithm that can use a flexible neural network as a function approximator. It is essential to know what the criteria are for flexible function approximators. Once these criteria are known, research can focus on methods that transform neural networks into flexible function approximators. In addition it is important to find out which limitation the existing deep reinforcement learning architectures have on the neural networks. Vice versa it is important to note what limitations the neural networks impose on the reinforcement learning architectures. The answers to these questions should result in an architecture that implements flexible deep reinforcement learning.

The fourth and final research question should give a final answer on the performance of the flexible deep reinforcement learning architecture. It will be tested on the environments identified by the third research question to validate the identified advantages and disadvantages of a flexible approach of learning.

1-4 Report structure

Part I of this research consists of an article that contains the most important findings of the research. It will give an answer on the second,third and fourth research question. Flexible reinforcement learning will be formalized. In addition advantages and disadvantages of flexible reinforcement learning will be hypothesised. Two different environments will be constructed, on which the sample complexity of the flexible deep reinforcement learning agent will be compared to a non-flexible approach and a decentralized approach.

Part II contains the preliminary research. This is an experimental literature study that starts with simple reinforcement learning algorithms, and ends up with a deep reinforcement learning algorithm that is able to capture continuous states and actions. This final algorithm is called DDPG, and forms the basis of the flexible deep reinforcement learning algorithm.

Part III will contain additional results of the research. A more in-depth analysis of the policies that are found in part I will be given. Also the flexible deep reinforcement learning algorithm is evaluated on more environments, namely a sparse environment, and a weakly observable environment. In addition, the parallel implementation of the flexible DDPG algorithm will be elaborated. Also the methods used for verification and validation in this research will be given.

Finally, in part IV a conclusion on the work and recommendations for future research are given.

Part I

Article

Flexible Curriculum Learning for Deep Reinforcement Learning Agents

J.L. Dorscheidt
Delft University of Technology

Abstract

Reinforcement Learning (RL) is a learning paradigm that learns by interacting with the environment. In practice, a RL agent needs to perform many actions to sample rewards and state transitions from their environments. Recent advances in using deep neural networks as function approximators reduce the sample complexity in very high dimensional environments. Another way of reducing the sample complexity of reinforcement learning agents is by shaping the learning path into a curriculum. Flexible curriculum learning is a way of shaping where the action and/or state dimensionality of the agent is gradually increased, in an effort to slowly increase complexity of the problem. In this research, a deep reinforcement learning architecture, Deep Deterministic Policy Gradients (DDPG), is modified to learn in a flexible way. The flexible DDPG is evaluated on a problem where no coordination is required to test basic functionality of the flexible DDPG algorithm. In addition, the flexible DDPG algorithm is evaluated on a problem where coordination is required. Results show that the flexible DDPG algorithm has a comparable sample complexity to a non-flexible approach.

1 Introduction

RL is a learning paradigm that can solve complex tasks by interacting with the environment, receiving rewards, and learning from that information. For example finding a strategy for playing blackjack [Sutton and Barto, 2012], or playing Atari games [Mnih et al., 2013] are tasks where RL can be used to find a solution. In aerospace domains, RL can be used for control tasks. For example, in [Ng et al., 2006] RL is used for autonomous flight in a helicopter UAV. One of the benefits of the RL paradigm is that it is a data-driven and model-free method. However because RL is data-driven, it needs many interactions from the environment before it reaches the optimal policy. Therefore, much of the focus in RL research lies on reducing this sampling complexity. One of the ways of reducing the sample complexity is by using function approximators, this enables generalization across the state-space domain. Recent advancements in deep RL resulted in DDPG [Lillicrap, 2016] . DDPG is an algorithm that is able to use deep neural networks as a function approximator on continuous action spaces. Another method to reduce the sample complexity is to shape the environment into a curriculum [Bengio et al., 2009],[Heess et al., 2017]. This way the initial problem is learned faster, before moving to the harder problem. Flexible reinforcement learning was explored in [Helmer et al., 2018], it is a special form of curriculum learning, where the learning path is shaped by gradually adding states and actions to the agent. Inspired by learning processes in humans, a flexible way of learning would make sense. For example, when learning to play the piano, one first learns a good policy for one hand, before moving on to learning the policy for the other hand. In [Helmer et al., 2018], a RL algorithm was modified to use local linear regression as flexible function approximators that can gain dependency on new input states and output actions.

In this research, flexible reinforcement learning is applied on an RL algorithm that uses global function approximators, namely DDPG. In addition, flexible reinforcement learning is

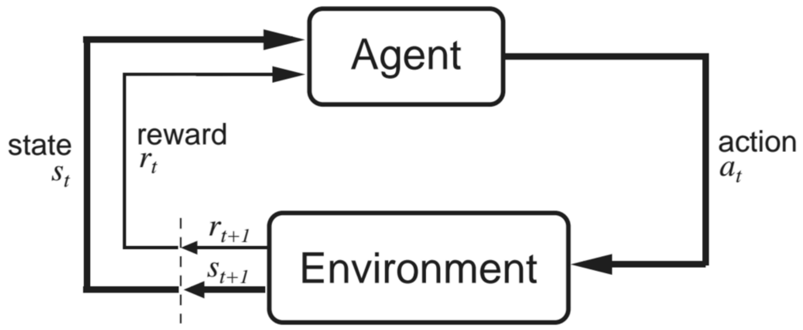


Figure 1: Interaction with the environment in RL. At each state S, the agent performs an action A. Consequently it receives a reward r and lands in a new state S'.

formalized and categorized. Advantages and disadvantages of flexible reinforcement learning are identified, and a comparison is made with decentralized reinforcement learning. The resulting flexible DDPG algorithm is evaluated on an environment that can be solved with decentralized reinforcement learning. In addition, the flexible DDPG algorithm is evaluated on an environment that can't be solved with decentralized reinforcement learning. The structure of this article is the following. First a background on reinforcement learning is given in section 2. Next, flexible reinforcement learning is formalized and categorized in section 3. In section 4 the DDPG algorithm is modified to learn in a flexible way. In section 5, and 6 the experiment design and the results of the experiments are given. In section 7 the results of the experiments are discussed. Finally section 8 gives a conclusion on the work and recommendations for future research.

2 Background

RL problems can be described by a Markov Decision Process (MDP). Each time step, the agent takes action A, receives a reward r and transitions from it's current state S to a new state S'. This process is depicted in figure 1. An MDP consists out of a state space S, and an action space A. Characteristic for a MDP is that state transition probabilities are determined by the current state and action, that is $P(s'|s,a) = P(s'|s^1, a^1, s^2, a^2...)$. In RL the behaviour that is learned is called the policy $\pi(S)$, and is a function of the current state. A policy can be discrete when it outputs a discrete set of actions, or continuous when it can model continuous action spaces. The return is the discounted sum of future rewards [Sutton and Barto, 2012].

$$R(t) = \sum_{i=t}^{T}(\gamma^{i-t} r_{(s_i,a_i)}) \tag{1}$$

RL attempts to find a policy that maximizes the expected returns over the initial state distribution.

$$J(\pi) = \mathbb{E}(R|\pi) \tag{2}$$

The Action-Value Q is the expected return at a state-action pair. Where the expectation is taken over the different trajectories and actions determined by the policy.

$$Q(s,a) = \mathbb{E}(R(t)|s_t, a_t) \tag{3}$$

2-1 RL vs dynamic programming

In dynamic programming, the state transition probabilities and reward functions are known and can be used to find an optimal policy. In RL, state-transitions and reward functions are unknown and are sampled from the environment. RL uses these samples to estimate the action values of the current policy. The policy can now be improved based on the estimated Action-Values of the current policy. A well known approach in a tabular setting is Q-learning [Watkins, 1989]. Q-learning is shown in algorithm 1.

Algorithm 1: Q-learning

```
Initialize Q(s,a) arbitrarily
for $i$ $in$ $episodes$ do
    Initialize start state S
    while $S \neq terminated$ do
        Choose $A$ from $S$ using policy derived from Q
        Take action $A$, observe $R$, $S'$
        Update Q-table, $Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma max_{A'} Q(S',A') - Q(S,A))$
        $S \leftarrow S'$
```

2-2 Continuous domains and Actor-Critic methods

Many real-world problems have state and action representations that are continuous. Tabular approaches are inefficient here because the state and action space need to be discretized. A better approach is to use function approximators for estimating action-values. This allows action-value estimation in continuous state domains, in addition, function approximators can aid in generalizing across states. A policy can also be explicitly represented by a function approximator, such that it can model continuous actions. The function approximator underlying the policy can be optimized using a policy gradient. For any differentiable policy π_θ, the policy gradient is [Sutton and Barto, 2012]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{4}$$

REINFORCE [Williams, 1992] is an example of an algorithm that uses a monte carlo estimate of the policy gradient by episodically sampling returns. However, using monte-carlo estimates of the policy gradient results in a high variance of the updates, and learning is slow. Actor-Critic methods estimate the Action-Value Q using a separate function approximator, this Critic is then used for calculating the policy gradient in 4.

2-3 Deep Q Network (DQN) and DDPG

A candidate for function approximators in reinforcement learning are neural networks. Neural networks are networks that consist out of layers of neurons. Each neuron receives an affinely transformed output of the previous layer as an input. The output of the neuron is the result of applying an activation function to the input.

$$y_i = act((\sum_{j=0}^{N}(w_{ij}y_j) + b_i)) \tag{5}$$

where j is the index of the neuron of the previous layer and i the index of the neuron of the current layer. The weights w_{ij} and biases b_i are the parameters of the network. Recently, neural networks have succesfully been implemented as a Q-value function approximators in reinforcement learning [Mnih et al., 2013]. In [Mnih et al., 2013] an algorithm is developed called DQN. DQN uses Q-learning to optimize an implicit policy that uses a neural network as a function approximator. One of the limitations of DQN is that is employs an implicit policy by taking the action corresponding to the maximum Action-Value Q in the Critic network. This means that the policy can only represent discrete actions. DDPG solves this problem by using an Actor-Critic approach, where the policy is explicitly modeled by another neural network. Using deep neural networks as function approximators in reinforcement learning comes with several challenges. First of all, in order to optimize the log-likelihood of the parameters of a neural network, the input data needs to be Independent and Identically Distributed (IID). In reinforcement learning, this condition is often violated, since samples from the environment are highly correlated with one another in a chronological fashion. In addition, the target values of the network have to be generated by the network itself. This implies that unwanted feedback loops are created and training might diverge. DQN and DDPG overcome these problems by using Q-learning with experience replay. Two networks for the actor and Critic are used. One network is used to generate the target values that the other network is trained on. This breaks unwanted feedback loops, and stabilizes training. Experience replay ensures that the data is more IID.

Because of the the ability of DDPG to model continuous actions, it is suited for applications in aerospace domains. For example, DDPG would be suited for finding a control policy for a quadcopter similar as in [Helmer et al., 2018]. DDPG is shown in algorithm 2.

Algorithm 2: Deep Deterministic Policy Gradient

```
Initialize replay memory D
Initialize Critic networks C and C', and Actor networks $\mu$ and $\mu'$ with random weights
for $i$ in episodes do
    Initialize start state S
    Initialize $\mathcal{N}$ while $s \neq$ terminated do
        select action A = $\mu + \mathcal{N}$ according to the current policy and exploration noise
        Take action A, observe R, S'
        append (S,A,R,S') to the replay memory
        sample random minibatch of N transitions from D
        set $y_j = R_j + \gamma * C'(s_{j+1}, \mu'(s_{j+1}))$
        update Critic by minimizing the loss $L = \frac{1}{N}\sum_j y_j - C(s_{j+1})$
        update Actor policy using the sampled policy gradient:
        $\nabla_{\theta\mu} J = \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q|_{s=s_i, a=\mu(s_i)} \nabla_{\theta\mu}\mu(s|\theta^\mu)|_{s_i}$
        update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
```

3 Flexible Reinforcement Learning

Reducing the sample complexity in reinforcement learning agents is an active area of research. One intuitive method to increase learning rates and reduce the amount of samples needed for learning is by shaping the environment of the agent into a curriculum. In machine learning, curriculum learning acts as a continuation method that aids the optimization process [Bengio et al., 2009]. Curriculum learning has also been evaluated in deep reinforcement learning. In [Heess et al., 2017] an environment is shaped where an agent has to clear obstacles that become increasingly difficult. Here the progression in the environment acts as a natural shaping of the learning process, easy tasks are learned before more difficult tasks are encountered.

In [Helmer et al., 2018] the concept of flexible reinforcement learning is introduced. [Helmer et al., 2018] designed an algorithm, called Flexible Heuristic Dynamic Programming, that is able to shape the learning process using a variable number of input states and output actions. [Helmer et al., 2018] modified the Model Learning Actor Critic (MLAC) algorithm of [Grondman, 2015], in a way that it is able to use Local Linear Regression (LLR) as a flexible function approximator. The LLR function approximator has the ability to learn by increasing the amount of states and actions. An experiment on a 2d quadcopter environment was done. The task in this environment was to bring the quadcopter to a desired state from a start state. The reward was the negative of the quadratic state-error of the quadcopter. In the initial phase, the quadcopter could only observe the altitude, and altitude derivate, and control the altitude action. In the second and final stage, the pitch angle and pitch angular velocity states are added in combination with the pitch action. It was observed that for a certain set of hyperparameters the flexible reinforcement learning setup required less iterations to converge to a solution compared to the normal reinforcement learning setup.

Here flexible reinforcement learning is formalized as follows:

Definition 1. *A policy π is to be found under an MDP which is comprised by a State (S) and an Action space (A) having dimensionality n and m respectively. The return is defined as the discounted sum of rewards from timestep t, $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$ This policy is optimized to maximize the expected return from the start state distribution $\mathbb{E}[R_1, \pi]$. Before the policy of the final state space is optimized, the agent only has access to a subset of the state and action space which are defined by the curriculum CU. The curriculum CU will return state and action masks as a function of the episode.*

$$(M_s, M_a) = CU(e) \tag{6}$$

Here M_s is the state mask vector, it has ones for the states that are observable and zeros for the states that are unobservable. In like manner, M_a is the action mask vector, it has ones for the actions that are controllable and zeros for actions that are uncontrollable.

3-1 Categorization of flexible reinforcement learning

Flexibility in the agent can be categorized in three categories, namely:

1. **Flexible action reinforcement learning.** In this category, the agent can only exhibit a subset of the action space in the initial training phase. After the initial policy has converged, the additional actions are added, and the policy is optimized on the new action space.

2. **Flexible state reinforcement learning.** In this category, the agent can only observe a subset of the state space in the initial training phase. After the initial policy has converged additional states are added, and the policy is optimized on the new state space.

3. **Combined flexible action and state reinforcement learning.** This is a combination of the previous two categories. In this category, input states and actions can be paired and masked together in the curriculum.

In addition, flexible reinforcement learning can be applied on an MDP that stays constant, or an MDP that is flexible. A flexible MDP can have flexible state transitions, a flexible reward, or a combination of both. In section 5 an environment with a flexible MDP will be given.

3-2 Disadvantages of flexible reinforcement learning

Before conducting experiments, possible advantages and disadvantages of flexible reinforcement learning can already be identified. Possible disadvantages of flexible reinforcement learning are the following:

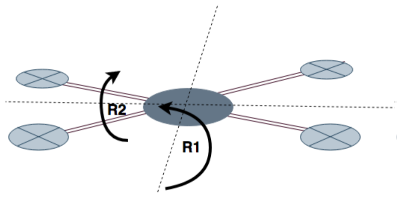
1. **Problems with Markov observability**: Adding flexibility in the input states of an agent could result in a weakly observable MDP. The state is not fully recoverable anymore from a single observation, and learning might be hindered.

2. **Problems in exploration and reward generation**: In flexible action reinforcement learning it can be difficult or impossible to explore the environment. One example where exploration might be totally impossible is the situation of learning to drive a car. In this environment two actions are present, steering and applying gas. However both actions are needed to explore the environment. Not applying gas, but only applying a steering action would result in no exploration whatsoever. Especially in sparse reward environments, reduced exploration capabilities can lead to suboptimal performance.

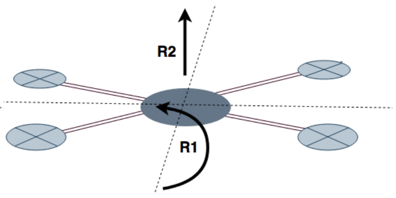
3. **Problems with coordination**: Environments with aggregated reward signals, could require coordination between different actions and states. This implies that the policies for the individual actions in the optimal policy are dependant on eachother. This could imply that a policy that is optimized while only controlling one action can be very far from the optimal policy when controlling both actions.

3-3 Advantages of flexible reinforcement learning

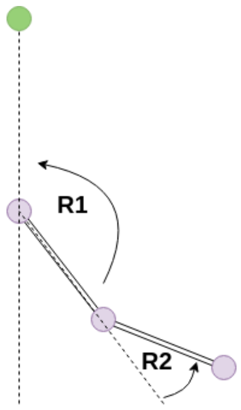
Flexible reinforcement learning can be considered a special case of curriculum learning. By shaping the environment into different learning stages that reduce the difficulty of the problem, the learning rate could possibly increase. Traditional curriculum learning in reinforcement learning is already proven to speed up learning in certain scenario's [Krueger and Dayan, 2009], [Heess et al., 2017]. Flexible reinforcement learning would shape the curriculum by increasing the number of states or actions an agent can control. When a task requires little coordination (that is, the observable states and controllable actions of a stage in the curriculum are not weakly observable), flexible learning could possibly result in increased learning rates, as the action space to be searched grows exponentially with the number of actions. This is analogous to the advantages identified in decentralized reinforcement learning [Busoniu et al., 2006]. However in some scenarios decentralized learning [Busoniu et al., 2006] might be a better approach to solve the MDP then flexible learning. In decentralized reinforcement learning, the MDP is split up into several MDP's, where each MDP is solved by a unique agent. This learning strategy can only be applied in environments where the reward signal is an aggregate of multiple rewards. In addition, the resulting Markov decision processes must be fully Markov observable and no coordination, or only indirect coordination should be required. Figure 2 shows an overview of environments that have aggregated reward signals. Flexible reinforcement learning could be beneficial in environments where there is a small amount of coordination required. When a small amount of coordination is required, a decentralized approach would be inefficient because it cannot optimize on the interaction between the decentralized MDP's.



(a) A control task with **no coordination** that can be solved with decentralized reinforcement learning. The goal is to bring the pitch angle and roll angle of the quadcopter to zero. The total reward is an aggregate of R1 and R2, representing the (negated) pitch and roll error respectively. In this example no coordination is required, since pitch dynamics have no effect on the roll dynamics and vice versa.



(b) A control task with **indirect coordination** that can be solved with decentralized reinforcement learning. The goal is to bring the quadcopter to a target altitude, and the roll angle to zero. The total reward is an aggregate of R1 and R2, representing the (negated) roll and altitude error respectively. In this example indirect coordination is required, since it is assumed that pitch dynamics have an effect on the altitude dynamics, however altitude dynamics have no effect on pitch dynamics.



(c) A control task with **direct coordination** that cannot be solved with decentralized reinforcement learning, because dynamics of both links have an influence on eachother. In this task the goal is to bring the end of the outer link to the most upward green position. The reward is an aggregate of R1 and R2, where R1 is proportional to the (negated) first link angle error, and R2 is proportional to the (negated) second link angle error. The links are connected with actuators. The environment is in the horizontal plane, so there is no effect of gravity.

Figure 2

4 Flexible DDPG

In this chapter, modifications to the existing DDPG algorithm are proposed that result in a modified algorithm that is able to learn in a flexible way.

4-1 Flexible neural networks

In DDPG both the Actor and the Critic are represented by deep neural networks. Gaining dependency on new inputs can be achieved in a straightforward way. The initial network has the amount of input states and output actions as in the final stage of a curriculum. However during the first stages the values of inputs that are unobservable or uncontrollable are put to zero by using masks generated by the curriculum. In addition, weights coming from the unobservable states and actions are put to zero. These modifications have the following implications:

1. **Weights coming from unobservable states or actions are not affected by gradient updates**. This is necessary since it is not required to learn behaviour based on those states. Because gradient descent uses updates of the following form:

$$\boldsymbol{\theta_{n+1}} = \boldsymbol{\theta_n} - \lambda \frac{\delta e}{\delta \boldsymbol{\theta_n}} \tag{7}$$

   where θ is the network parameter, e is the loss, and λ is the learning rate.

   and

$$\frac{\delta e}{\delta \boldsymbol{\theta_n}} = \frac{\delta e}{\delta \boldsymbol{x_{n+1}}} \frac{\delta \boldsymbol{x_{n+1}}}{\delta \boldsymbol{\theta_n}} \tag{8}$$

   where $\boldsymbol{x_{n+1}}$ is the network input to layer $n+1$. Since :

$$\boldsymbol{x_{n+1}} = \sum_{i=0}^{j} y_n^i w_i \tag{9}$$

   where y_n^i is the output value of the i'th neuron of layer n, (or when n equals zero, it is the i'th element of the input vector to the network). When y_n^i is equal to zero, it is straightforward to see that the weights of that input are not altered.

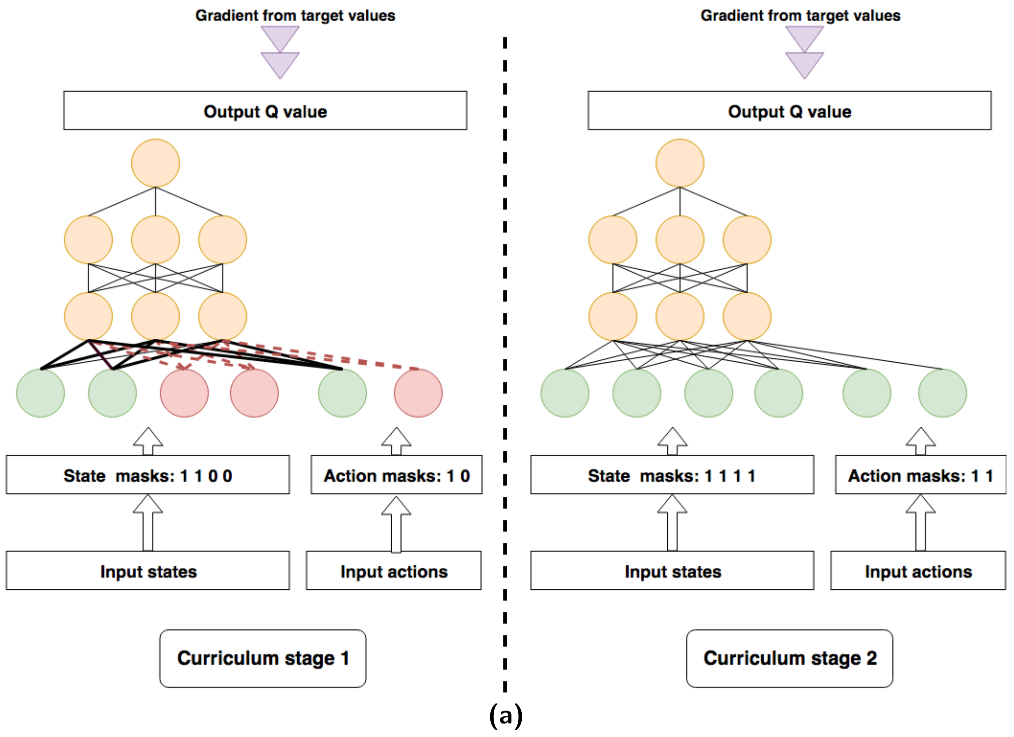
2. **The network has no dependency on unobservable states and actions.** This implies that both the Actor and Critic remain constant under variation of the unobservable states and actions. This is required since sudden inference from new states and actions in the curriculum will change the previously learned policy and value function. During the new stage, the new actor and Critic will slowly be altered by the new states and actions, since weights are now affected by the optimization process because input values are not zero anymore.

4-2 modifications to Critic, Actor and the replay memory

The Actor network can be split up into different columns that correspond to certain parts of the curriculum. This will put less constraints on optimizing each action, because no weights and neurons are shared between actions. When weights and neurons would be shared, it is likely difficult to optimize towards a newly added action in the curriculum, because the parameters of the network have fully moved towards values that are good for the actions in the earlier stage of the curriculum. This strategy can however not be applied on the Critic, since the Critic only outputs one Q-value, as opposed to multiple actions in the Actor. The modifications to the Actor and Critic networks are summarized in figure 3. The masking of data takes place just before it is put in the replay memory. This implies that in new stages of the curriculum, the replay memory still contains masked values from the old curriculum. Therefore it is chosen to empty the replay memory once a new stage of a curriculum is started. In addition, a new warmup period must start to fill the replay memory during the start of a new stage. In figure 4 an overview of the modified ddpg algorithm is shown. The modified DDPG algorithm is shown in algorithm 3.

Algorithm 3: Flexible Deep Deterministic Policy Gradient

```
Initialize curriculum CU
Initialize Critic networks C and C', and Actor networks $\mu$ and $\mu'$ with random weights
Set input weights from unobservable input states and uncontrollable actions to zero
for i in CU do
    Initialize replay memory D
    for j in warmup episodes do
        initialize start state S
        Generate state and action masks $M_s$ and $M_a$ from CU
        while $s \neq terminated$ do
            select action A = $\mu(S) + \mathcal{N}$
            mask action A with zeros from $M_a$, take action $A$, observe R, S'
            mask S' with zeros based on $M_s$
            append masked state transitions in replay memory, D $\leftarrow$ $(S,A,R,S')$
    for k in episodes do
        Initialize start state S
        Generate state and action masks $M_s$ and $M_a$ from CU
        Initialize noise process $\mathcal{N}$
        while $s \neq terminated$ do
            mask state S with zeros from $M_s$
            select action A = $\mu(S) + \mathcal{N}$
            mask action A with zeros from $M_a$
            take masked action A, observe R, S', mask S' with zeros based on $M_s$
            append masked state transitions in replay memory, D $\leftarrow$ $(S,A,R,S')$
            sample random minibatch of N transitions, $(S_j,A_j,R_j,S'_j) \leftarrow$ D
            set $y_j = R_j + \gamma * C'(S'_j, \mu(S'_j))$
            update Critic by minimizing the loss $L = \frac{1}{N}\sum_j y_j - C(S_j, A_j)$
            update Actor policy using the sampled policy gradient:
            $\nabla_{\theta^\mu} J = \frac{1}{N}\sum_j \nabla_a Q(S, A|\theta^Q|_{S=S_j,A=\mu(S_j)})\nabla_{\theta^\mu}\mu(S|\theta^\mu)|_{S_j}$
            update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
```



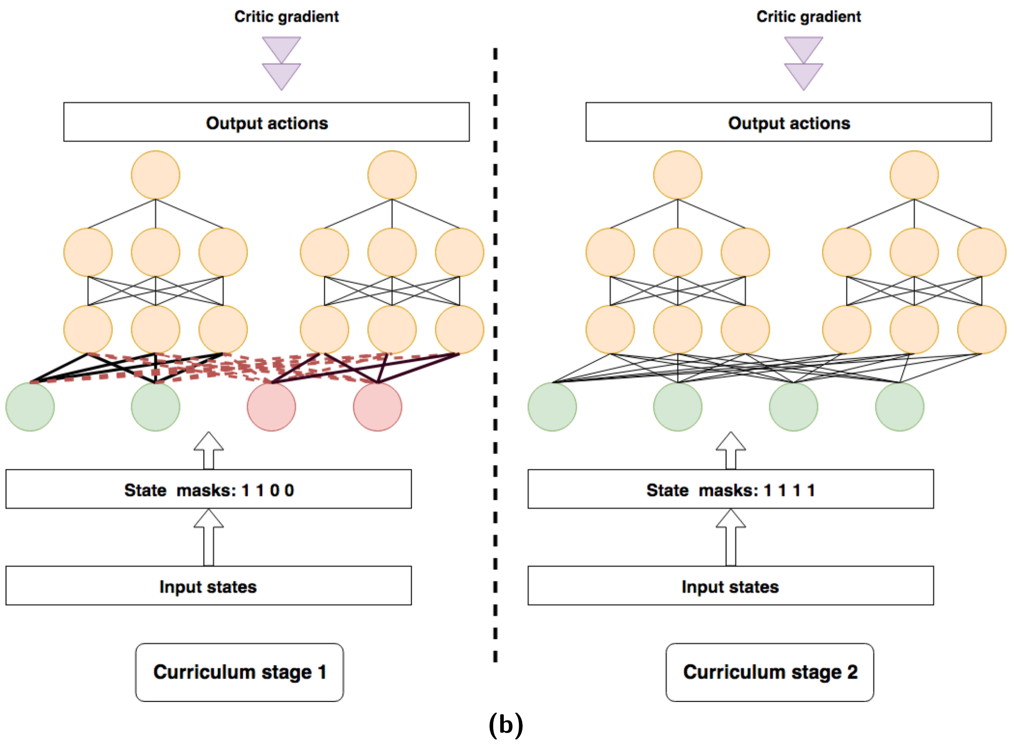


Figure 3: Modifications to the Actor (bottom) and Critic (top) networks. For the Actor network, multiple columns are constructed to prevent sharing of neurons. Input states are multiplied with the state masks. Red input neurons are unobservable states, green input neurons are observable states. Red connections are weights that are zero. Initially no coordination is assumed, hence weights interconnecting states and actions from different parts of the curriculum are initialized at zero. During stage 2 of this curriculum, all the states are observable and inference from the new states will move the red input weights away from zero.

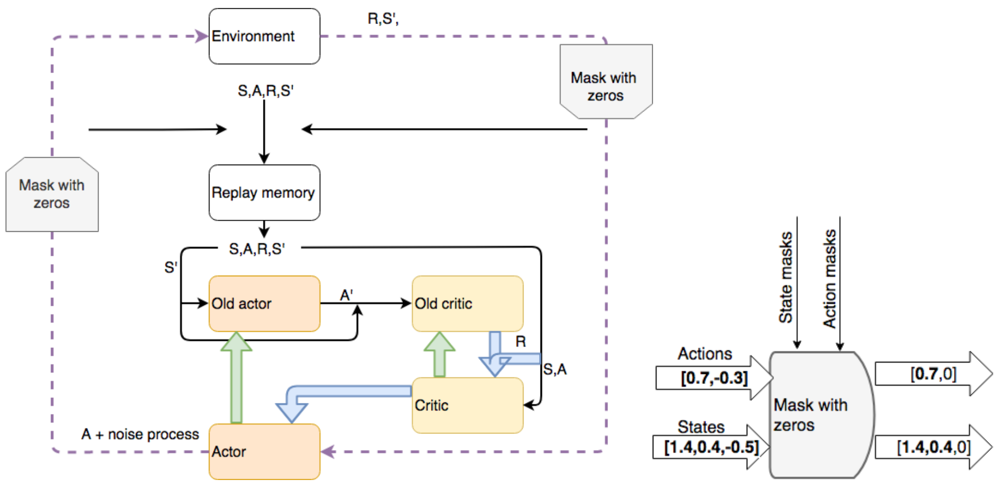


Figure 4: Overview of the flexible DDPG algorithm (left) and masking process (right).The purple dotted arrow represents actions and states involved in interaction with the environment. The black arrows represent action and states involved in training of the Critic and Actor. Bold blue arrows represent gradient updates, and bold green arrows represent updates of parameters of the target networks. Minor changes to the algorithm are required to train with a flexible curriculum, namely a modification of the structure of the Actor network, and the masking of data.

5 Experiment design

To test the performance of the flexible DDPG algorithm, environments are constructed from the Deepmind Control Suite library [Tassa et al., 2018]. The Deepmind Control Suite library uses standardized reward signals, such that each time step a maximum reward of one can be received. It uses the MuJoCo [Todorov et al., 2012] engine for calculating transitions of the environment. Flexible DDPG should be able to 'port' a policy from an early stage in the curriculum to a new stage in the curriculum. In addition, the Actor and Critic networks should be adaptable and learn behaviour based on new states and actions in the environment. To eliminate the possibility that an environment might suffer from some of the disadvantages of flexible reinforcement learning described previously, an environment is created that requires no coordination between different stages of the curriculum, this environment is called the TwoCartpoles environment and is shown in figure 5. The TwoCartpoles environment is a superposition of the Cartpole environment. In the Cartpole environment, the goal of the agent is to swing a lever to the upward position by controlling the forward or backward action of a cart. In the TwoCartpoles environment, the reward is a superposition of the rewards of the first cart and the second cart, both multiplied by 0.5 to get a maximum reward of one. This environment can be used to test if the flexible DDPG algorithm is able to retain old behaviour, and learn new behaviour throughout the curriculum. In addition, an environment is created that is difficult to solve with decentralized reinforcement learning. In the Reacher environment, the agent controls two links. It is the goal of the agent to move the outer link to a randomized target position. The reward signal in the reacher environment is constructed based on the distance between the outer link and the target. The curriculum has two stages. The first stage consists of an MDP that has only one link, the second stage consists of an MDP that contains two links. The reacher environment is shown in figure 6. The state vector description and curriculum can be found in Appendix A. The Reacher environment is difficult

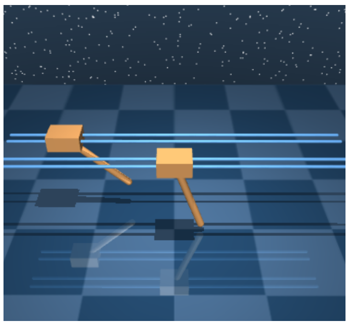


Figure 5: In the TwoCartpoles environment, the task is to move both both pendulums to the upwards position, this can be done by moving the carts back and forwards. For both pendulums, a reward is received when they are within a range of the upward position, these reward signals can be aggregated into the final reward that is received by a centralized or flexible agent. A decentralized agent has access to the individual reward signals of the pendulums.

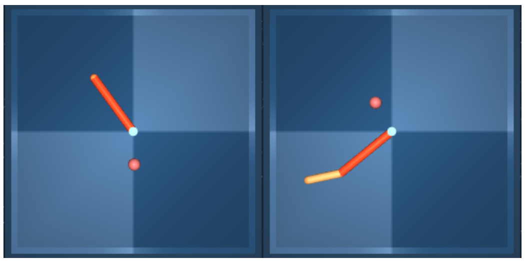
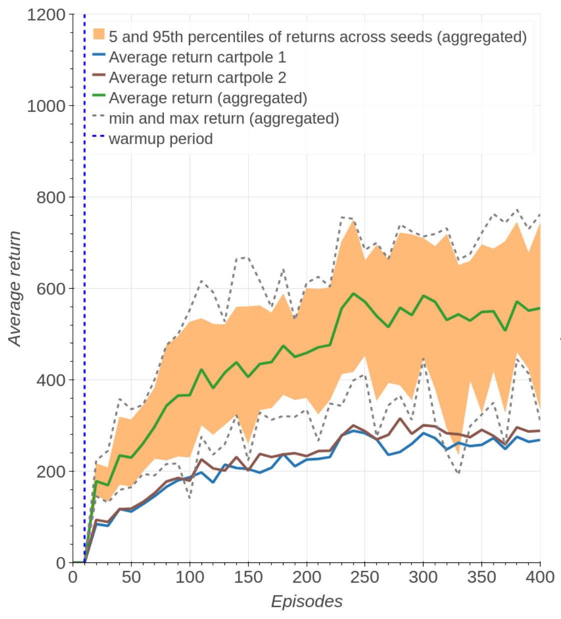


Figure 6: First (left) and second (right) stage if the flexible Reacher task. The goal of the agent is to move the outer link to the red target zone. This can be done by rotating the two joints of the reacher. The links are in the horizontal plane, so there is no effect of gravity. A curriculum can be constructed where both the agent and the environment is flexible. During every stage of the curriculum the reward is inversely proportional to the distance between the outermost link of that stage and the target. Every episode, the angular and radial position of the target is randomized. Exact parameters for this environment and the curriculum can be found in table 1 in appendix A

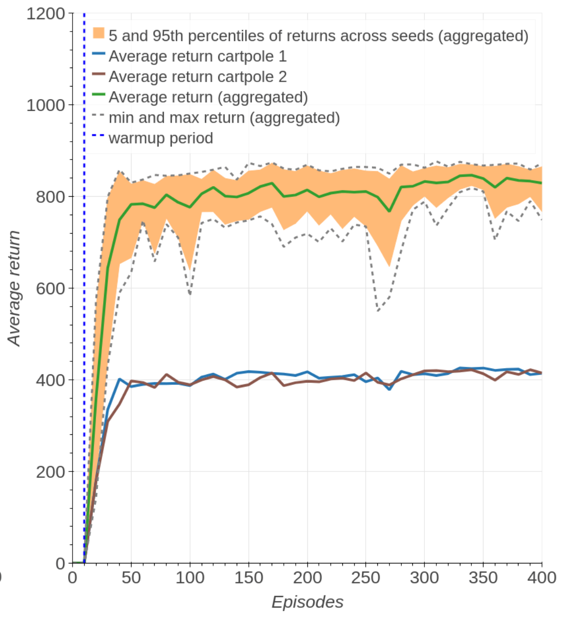
to solve with decentralized reinforcement learning, because it has the following properties:

- **Direct coordination is required between the subprocesses that are represented by the curriculum.** The dynamics of the first link have a direct effect on the dynamics of the second link and vice versa. Therefore treating the two processes as seperate MDP processes results in weakly observable MDP's.
- **Defining a separate reward signal for one of the subprocesses in the curriculum is difficult.** In [Busoniu et al., 2006] the environment allows for construction of two seperate reward functions, and a decentralized approach is possible. However, in the Reacher environment, it is difficult to structure a separate reward signal for the second link, since the target position is randomized.

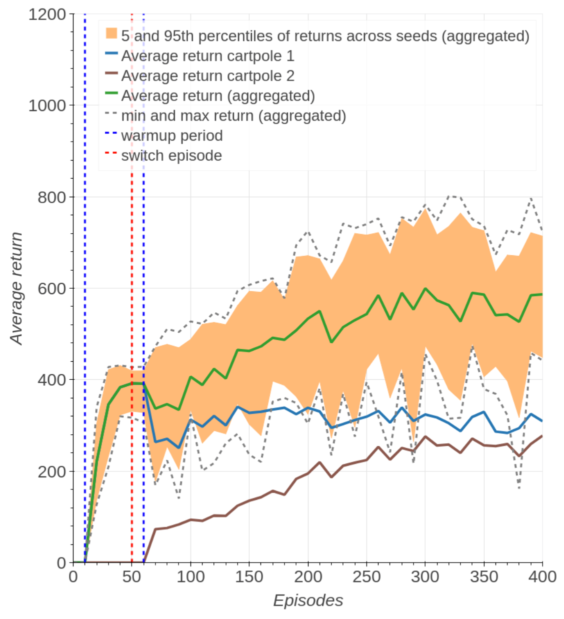
Hyperparameters used in both experiments can be found in Appendix B.



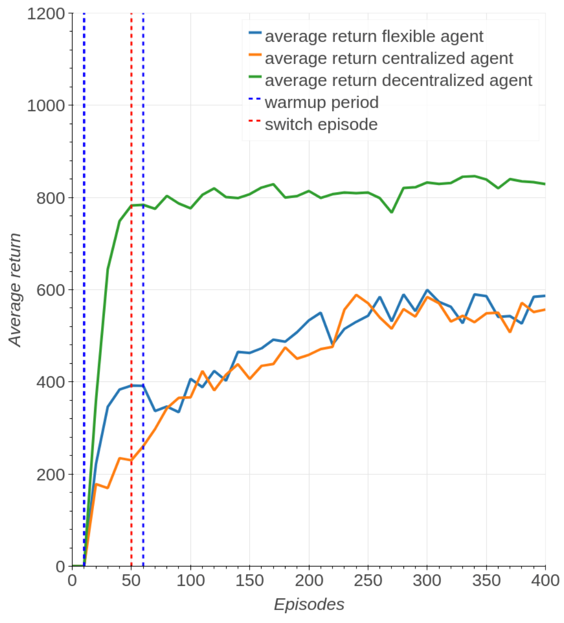
(a) Learning curve of the central DDPG agent



(b) Learning curve of the decentralized DDPG agent

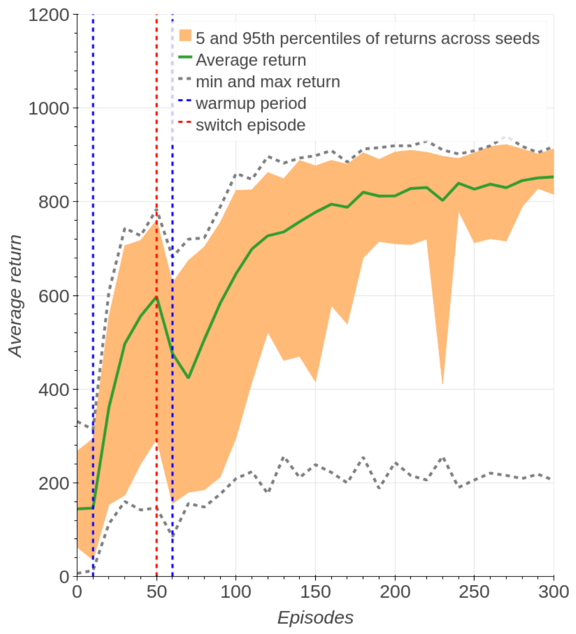


(c) Learning curve of the flexible DDPG agent

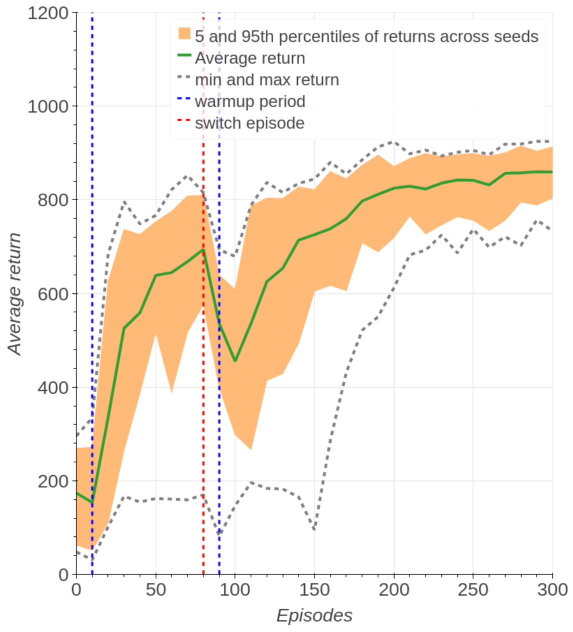


(d) Comparison of the learning curves of the central DDPG, decentralized DDPG and flexile DDPG agents

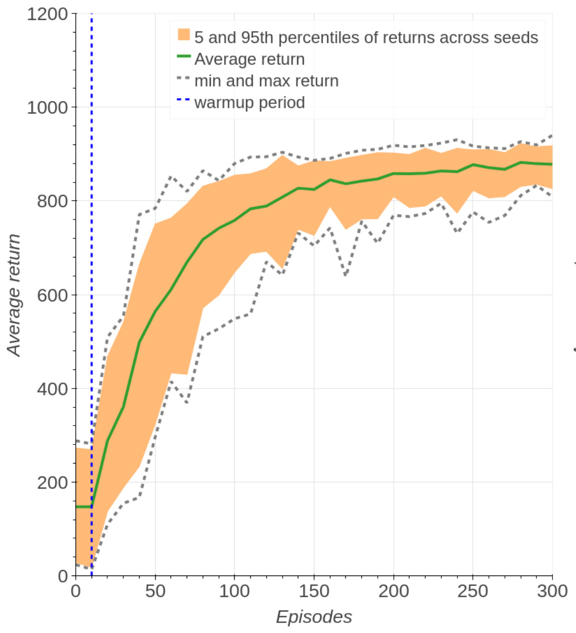
Figure 7: Learning curves for the normal DDPG, decentralized DDPG and flexible DDPG algorithms on the TwoCartpoles environment. The flexible DDPG algorithm uses the curriculum described in appendix A. 15 experiments are run to account for stochasticity in the algorithm. Every 10 episodes, the policy of each experiment is evaluated with 10 different start state initializations to test generalization capabilities. In these start states, the cart is at the center position with zero velocities and the lever pointing downwards. Start states are randomized by adding a small amount of noise to this start state.



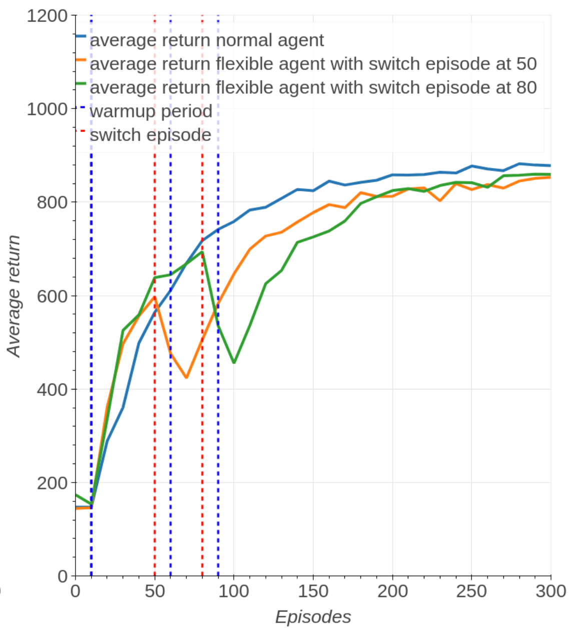
(a) Learning curve for the flexible DDPG algorithm on the reacher environment, switch episode is at 50 episodes.



(b) Learning curve for the flexible DDPG algorithm on the reacher environment, switch episode is at 80 episodes.



(c) Learning curve for the normal DDPG algorithm on the reacher environment



(d) Comparison of the learning curves for the flexible DDPG agents with switch episode at 50 and 80 episodes, and the normal DDPG algorithm

Figure 8: Learning curves for the normal DDPG and the flexible DDPG algorithms on the Reacher environment. The flexible DDPG algorithm uses the curriculum described in appendix A. 30 experiments are run to account for stochasticity in the algorithm. Every 10 episodes, the policy of each experiment is evaluated with 10 different start state initializations to test generalization capabilities. Every start state initialization, both the angles of the link, as well as the target positions are randomized.

6 Results

In figure 7 and 8, a comparison of the learning curve for the normal DDPG and the flexible DDPG algorithm on the TwoCartpoles environment, and the Reacher environment is shown.

7 Discussion

It can be seen that the convergence of the flexible DDPG algorithm on the TwoCartpoles environment is comparable to the normal DDPG algorithm. Two inherent disadvantages of the flexible DDPG algorithm are that the replay memory is emptied, and that the critic needs to refit it's parameters to the new additional part of the reward function, while not disturbing the previously learned values too much. It can be seen in figure 7 that the policy for the first cartpole is disturbed once the new stage initiates. However, the old policy is quickly recovered again after a few episodes, although never fully regaining the old performance. This sub-optimal performance of the flexible and non-flexible agents is likely the result of using a centralized critic for two separate MDP processes. Because the centralized critic needs to find a mapping from the input states of both cartpole processes to one output value, it is difficult for the critic to find a mapping that has action gradients that are invariant to the input states of the opposing cartpole. More mathematically speaking, the mapping from input states to output value of the total MDP is a superposition of both processes because no coordination is required between the processes. that is:

$$Q(S_{c1}, A_{c1}, S_{c2}, A_{c2}) = f(S_{c1}, A_{c1}) + f(S_{c2}, A_{c2}) \tag{10}$$

Where subscripts $_{c1}$ correspond to the states and actions belonging to the MDP for the first cartpole, and subscripts $_{c2}$ belong to the states and actions for the MDP of the second cartpole. Because of this, the action gradient for the first cartpole is only a function of the states of the first cartpole, and the action gradients for the second cartpole is only a function of the second cartpole:

$$\frac{\delta Q(S_{c1}, A_{c1})}{\delta A_{c1})} = f(S_{c1}, A_{c1}) \tag{11}$$

and

$$\frac{\delta Q(S_{c2}, A_{c2})}{\delta A_{c2})} = f(S_{c2}, A_{c2}) \tag{12}$$

However, because a structure for the critic is required that uses sharing of neurons (to capture potential coordination), it is difficult for the critic to make a mapping where the action gradients for the actions of both cartpoles are independent of the states of the opposing cartpole. This results in gradient updates to the actor that have a dependency on input states for the opposing cartpoles. So in short, it is difficult for the critic to 'realize' that both processes are independent processes, and thus an unnecessary link is made between input states that have no influence on one of the two processes. So using a central critic for two independent MDP has a detrimental effect on performance in this environment. However, using a central critic is essential in flexible reinforcement learning, since capturing potential coordination is required.

The TwoCartpoles experiment shows that the flexible DDPG algorithm is able to 'port', to some extend, the previously learned policy over to the new stage of the curriculum. In addition, the flexible DDPG algorithm is able to learn a new policy (the second cartpole) while not disturbing the old policy too much.

Figure 8 shows that convergence of the flexible DDPG on a flexible environment is comparable or slightly worse compared to the convergence of a non-flexible approach. It can be seen that in the first stage of the curriculum, the flexible agent quickly converges to a good policy for controlling the first link. Once the second stage is initiated, the average return drops a bit. This is because the replay memory is emptied, and because the reward signal is changed. However it can also be seen that the policy for the second link is quickly implemented in the actor and critic, as the learning rate is rather steep after the switch epoch. The benefits of using a flexible approach in this environment are marginal compared to a normal approach. It could be that the environment is not complex enough, and thus the benefits of a flexible way of learning are not seen in this environment.

8 Conclusion

In this research flexible deep reinforcement learning is introduced, in an effort to reduce the sample complexity of a deep reinforcement learning agent. The existing DDPG algorithm is modified to learn in a flexible way, that is, a policy is formed by gradually increasing the state and action space. Flexible curriculum learning is defined by a curriculum that specifies which states and actions are observable and controllable in a part of the learning process. Flexibility of the deep reinforcement learning agent is achieved in the following way. The DDPG algorithm has the same amount of input states and output actions as the final stage of a curriculum. However, during earlier stages, the unobservable states and actions are put to zero by state and action masks generated by the curriculum. In addition weights coming from unobservable states and actions are initialized at zero, this ensures that unobservable states have no inference on the actor and critic. In this manner, a flexible neural network that can expand it's input neurons is mimicked. Flexible reinforcement learning is hypothesised to have the same benefits as decentralized reinforcement learning, namely, it speeds up learning because the action space to be searched is gradually increased. However flexible reinforcement learning can be applied in certain cases where a decentralized approach is not possible. These are scenarios in which either the reward signal is not splittable, or when coordination is required between different subprocesses. The flexible DDPG algorithm is evaluated on a task where no coordination is required, to eliminate possible Markov observability problems. It can be seen that the flexible DDPG agent is able to retain behavior learned from earlier stages of the curriculum, and learn new behavior from new states and actions added by the curriculum, however a decentralized approach has a much lower sample complexity on a task that requires no coordination. In addition the flexible DDPG algorithm is tested on an environment that cannot be solved with decentralized reinforcement learning. Results show that learning rates for the flexible DDPG and the normal DDPG agents on this environment are comparable. It is likely that this last environment is relatively easy to solve with normal reinforcement learning, such that the benefits of flexible DDPG are marginal.

9 Recommendations

In an aggregated reward environment such as the TwoCartpoles environment, it is observed that flexible DDPG has some difficulty in mapping two independent MDP processes to a single Q-value, because a centralized critic is used where neurons are shared between the mappings of both independent MDP's. A different neural structure as in progressive neural networks [Rusu et al., 2016], might be able to circumvent sharing of neurons, by assigning a new column of neurons to each new stage in the curriculum. The output Q-value of all the columns would then need to be aggregated after all the final layers. This structure could capture coordination by implementing cross connections between the hidden layers of the individual columns. This structure could make it easier for the agent to maintain previously learned policies, while learning additional policies.

Because flexible reinforcement learning is applicable in environments that require a small amount of coordination between subprocesses, the current exploration process could be naive . For example, in an environment with a little bit of coordination, if a policy is formed in the first stage, it is known that this policy needs to be altered by a small amount due to the changed MDP in the new stage of the curriculum. It could be chosen to lower the exploration noise on the previously learned action. This different exploration strategy could have a positive effect on the learning rate of flexible DDPG.

Furthermore in this research flexible DDPG is only tested on environments where exploration is possible during every stage of the environment. However, a flexible approach is not possible in some environments because excluding actions could greatly hinder exploration. One way to solve this problem is to use a 'teacher'algorithm for the uncontrollable actions. For example in learning to drive a car, this teacher policy could control the policy for the gas pedal in the initial phase such that exploration of the environment is possible. In the next stage, the teacher would be removed and the agent has to learn the additional action.

Finally, in this research only combined flexible state and action reinforcement learning is tested. It is assumed that flexible state reinforcement learning mostly results in Markov observability problems. However there might be a niche of problems in which flexible state reinforcement learning might be beneficial. In these environments, the initial phase of the curriculum has a certain amount of observability, and each phase of the curriculum adds a bit more observability to the agent. The actor and critic networks are then optimized on gradually increasing input dimensions, which could aid the optimization process.

References

Yoshua Bengio, Umontrealca Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum Learning. In *Proceedings of the 26th Int. Conference on Machine Learning*, pages 41–48. Arxiv, 2009. doi: 10.1145/1553374.1553380.

Lucian Busoniu, Bart De Schutter, and Robert Babuska. Decentralized Reinforcement Learning Control of a Robotic Manipulator. In *2006 9th International Conference on Control, Automation, Robotics and Vision*, pages 1–6. IEEE, 2006. ISBN 1-4244-0341-3. doi: 10.1109/ICARCV.2006.345351.

Ivo Grondman. *Online Model Learning Algorithms for Actor-Critic Control.* PhD thesis, Delft University of Technology, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, pages 1026–1034, 2015. doi: 10.1109/ICCV.2015.123.

Nicolas Heess, Dhruva Tb, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S M Ali Eslami, Martin Riedmiller, and David Silver Deepmind. Emergence of Locomotion Behaviours in Rich Environments. *arxiv*, 2017.

Alexander Helmer, Coen C. de Visser, and Erik-Jan Van Kampen. Flexible Heuristic Dynamic Programming for Reinforcement Learning in Quad-Rotors. In *2018 AIAA Information Systems-AIAA Infotech @ Aerospace*, Reston, Virginia, jan 2018. American Institute of Aeronautics and Astronautics. ISBN 978-1-62410-527-2. doi: 10.2514/6.2018-2134.

Diederik P Kingma and Jimmy Lei Ba. Adam: A method for stochastic gradient descent. *ICLR: International Conference on Learning Representations*, pages 1–15, 2015.

Kai A Krueger and Peter Dayan. Flexible shaping: How learning in small steps helps. *Cognition*, 110:380–394, 2009. doi: 10.1016/j.cognition.2008.11.014.

Timothy Lillicrap. Continuous control with deep reinforcement learning. *arxiv*, 2016. ISSN 1935-8237. doi: 10.1561/2200000006.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arxiv*, 2013.

Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement earning. *Springer Tracts in Advanced Robotics*, 21:363–372, 2006. ISSN 16107438. doi: 10.1007/11552246_35.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive Neural Networks. *arxiv*, 2016.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction.* MIT Press, 2012. ISBN 0262193981. doi: 10.1109/TNN.1998.712192.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego De, Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind Control Suite. Technical report, 2018.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IEEE International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. ISBN 9781467317375. doi: 10.1109/IROS.2012.6386109.

C. J. C. H. Watkins. *Learning from Delay Rewards.* PhD thesis, University of Cambridge, 1989.

Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3):229–256, 1992. ISSN 15730565. doi: 10.1023/A:1022672621406.

A Curricula and environment specifications

The reward function in the first and second stage of the continuous Reacher environment is the following:

$$R(dist) = \begin{cases} 1 - \frac{dist}{0.16}, & \text{if } dist \leq 0.16 \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

The reward function in the first and second stage of the TwoCartpoles environment is simply a superposition of the continuous reward function for the Cartpole environment in the DeepMind Control Suite. Where each reward is multiplied with 0.5 in order to receive a maximum reward of one.

Table 1: Environment parameters for the Reacher environment

Environment parameter	**Value**
Inner link length (m)	0.16
Outer link length (m)	0.08
Inner link mass (kg)	0.1
Outer link mass (kg)	0.1
Target radius (m)	Random between 0.08 and 0.24
Target angle (rad)	Random between 0 and 2π

Table 2: Curriculum used in the Twocartpoles experiment.

	cart1 x position (m)	*cart1 link x position (m)*	*cart1 link y position (m)*	*cart2 x position (m)*	*cart2 link x position (m)*	*cart2 link y position (m)*	*cart1 x velocity (m/s)*	*cart1 link angular velocity (rad/s)*	*cart2 x velocity (m/s)*	*cart2 link angular velocity (rad/s)*	*cart1 action*	*cart2 action*
stage 0	1	1	1	0	0	0	1	1	0	0	1	0
stage 1	1	1	1	1	1	1	1	1	1	1	1	1

Table 3: Curriculum used in the flexible reacher experiment.

	inner link angle (rad)	*inner link x distance to target (m)*	*inner link y distance to target (m)*	*inner link angular velocity (rad/s)*	*outer link angle (rad)*	*outer link x distance to target (m)*	*outer link y distance to target (m)*	*outer link angular velocity (rad/s)*	*inner link torque*	*outer link torque*
stage 0	1	1	1	1	0	0	0	0	1	0
stage 1	1	1	1	1	1	1	1	1	1	1

B Hyperparameters used for experiments

Table 4: hyperparameters used in the Flexible Reacher and the Twocartpoles experiments

parameter	**value**
batch size	64
learning rate Critic	0.0001
learning rate Actor	0.0001
number of hidden layers	3
neurons per layer	200 for critic (100 in decentralized DDPG), 100 per action column in the actor
activation function	Leaky Rectified Linear (He et al. [2015])
optimization method	Adam (Kingma and Ba [2015])
exploration noise process	Ornstein Uhlenbeck process
Ornstein Uhlenbeck process mean (μ)	0.0
Ornstein Uhlenbeck process variance (σ)	0.3
Ornstein Uhlenbeck process theta (θ)	0.15
target network tracking rate	0.01
buffer size	100000

C Policy visualization

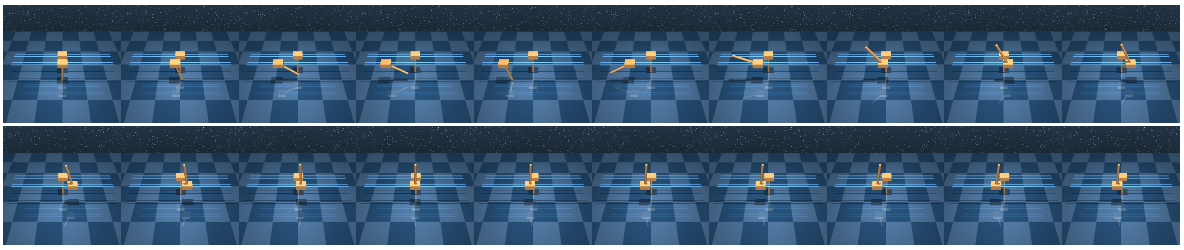


Figure 9: Policy visualization for flexible DDPG algorithm on the TwoCartpoles task after final episode of the first stage of the curriculum (snapshots taken every 10 timesteps, starting from the first timestep)

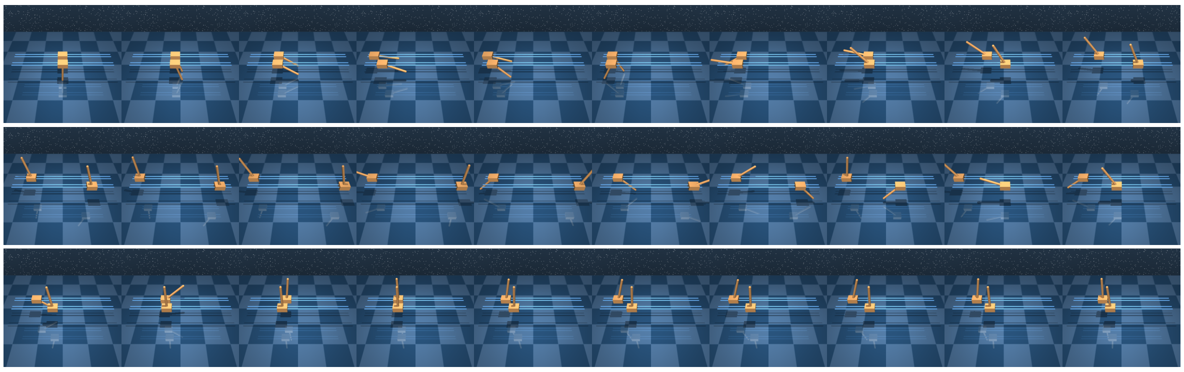


Figure 10: Policy visualization for flexible DDPG algorithm on the TwoCartpoles task after final episode of the second stage of the curriculum (snapshots taken every 10 timesteps, starting from the first timestep)

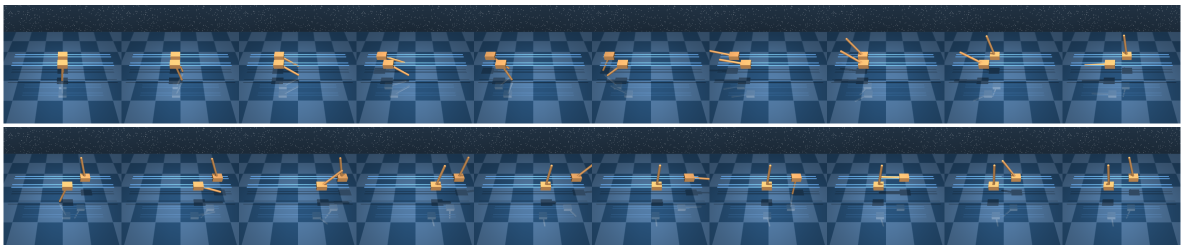


Figure 11: Policy visualization for normal DDPG algorithm on the TwoCartpoles task after final episode (snapshots taken every 10 timesteps, starting from the first timestep)

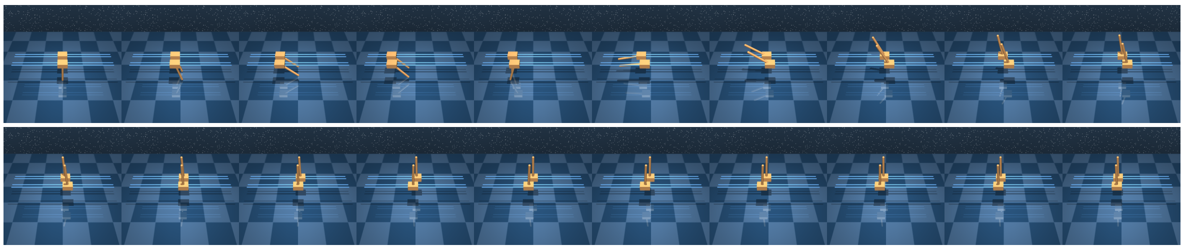


Figure 12: Policy visualization for the decentralized DDPG algorithm on the TwoCartpoles task after final episode (snapshots taken every 10 timesteps, starting from the first timestep)

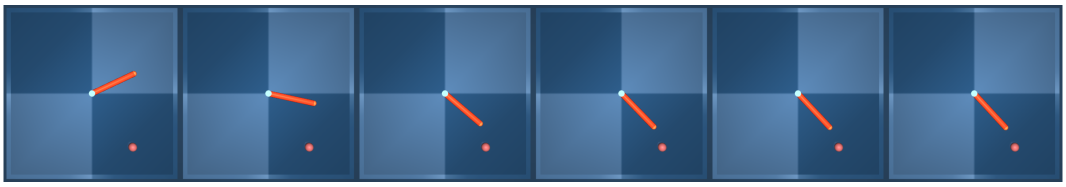


Figure 13: Policy visualization for flexible DDPG algorithm on the Reacher task after final episode of the first stage (50 episodes) of the curriculum with switch episode at 50 episodes. (snapshots taken every 10 timesteps, starting from the first timestep)

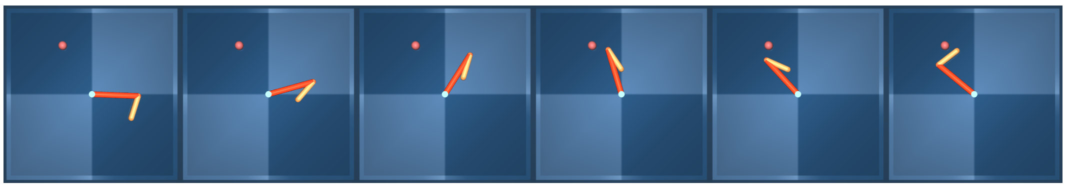


Figure 14: Policy visualization for flexible DDPG algorithm on the Reacher task after final episode of the second stage (300 episodes) of the curriculum with switch episode at 50 episodes. (snapshots taken every 10 timesteps, starting from the first timestep)

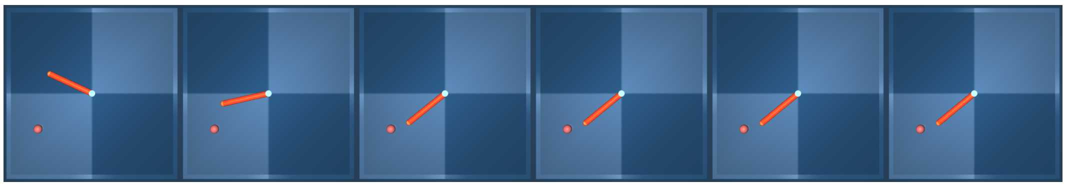


Figure 15: Policy visualization for flexible DDPG algorithm on the Reacher task after final episode of the first stage (80 episodes) of the curriculum with switch episode at 80 episodes. (snapshots taken every 10 timesteps, starting from the first timestep)

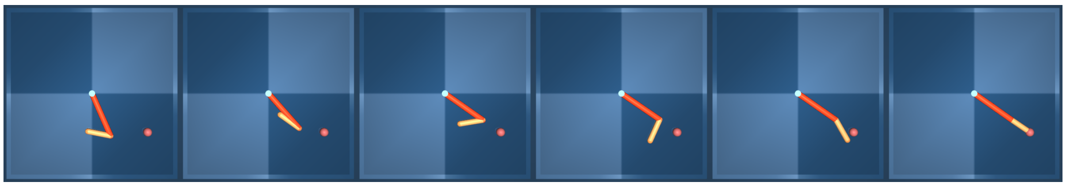


Figure 16: Policy visualization for flexible DDPG algorithm on the Reacher task after final episode of the second stage (300 episodes) of the curriculum with switch episode at 80 episodes. (snapshots taken every 10 timesteps, starting from the first timestep)

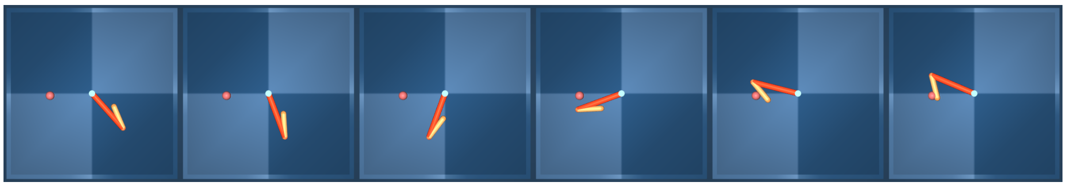


Figure 17: Policy visualization for the normal DDPG algorithm on the Reacher task after 300 episodes (snapshots taken every 10 timesteps, starting from the first timestep)

Part II

Preliminary thesis

Chapter 2

Literature Survey

2-1 Reinforcement Learning Fundementals

In order to get a better understanding of the current state of research in reinforcement learning, it is wise to first give a brief overview of the underlying concepts of reinforcement learning. Most of the derivations and algorithms in this chapter come from Sutton and Barto [2012] and the lecture videos of Silver [2015]. Reinforcement learning is in fact a very broad topic, and many algorithms can be classified as reinforcement learning. At the end of this chapter, a taxonomy is created that gives an overview of the different reinforcement learning methods.

2-1-1 Markov Decision Processes

In order to start with the fundementals of reinforcement learning, it is essential to understand the principles of a Markov Process. A process has the Markov properety if the future state is exclusively a function of the current state and action, that is

$$P(S^{t+1}|S^t, S^{t-1}, S^{t-2}, ..., S^0) = P(S^{t+1}|S^t) \tag{2-1}$$

An example of a Markov Process is for example a clock, where the next time is only dependant on the current time, and not on the history of all times.

Markov Processes can be extended to Markov Decision Process (MDP). A Markov Decision Process also involves an action (A) and reward (R). Markov Decision Processes are described as follows:

$$P(S^{t+1}, R|A^t, S^t, S^{t-1}, S^{t-2}, ..., S^0) = P(S^{t+1}, R)|A^t, S^t) \tag{2-2}$$

2-1-2 Value functions and Q-values

Reinforcement learning focuses on maximum the expected return by selecting the most optimal action for a given state. If we let

$$P^a_{s',s} = P(S^{t+1} = s'|S^t = s, A^t = a) \tag{2-3}$$

and

$$R^a_s = \mathbb{E}(R^{t+1}|S^t = s, A^t = a) \tag{2-4}$$

The return is defined as the sequence of rewards following from a certain state.

$$G_t = \sum(\gamma R^{t+1}, \gamma^{t+2}R^{t+2}, \gamma^{t+3}R^{t+3}, ...) \tag{2-5}$$

A discount factor γ is introduced for the following two reasons.

1. To take into account infinite time horizons. If no discount factor would be introduced, the sum of rewards in non-episodic environments would result in infinity. Hence the discount factor makes this property tractable.
2. To give a temporal importance measure for rewards. For most problems, rewards that are sooner on the time-horizon are more important then rewards that are further away.

The probability distribution of actions as a function of the state is also known as the policy $\pi(s,a)$. Since both the state transition probabilities and the policy are functions of the current state and action, the state transition probabilities can be expressed as a function of the policy.

$$P^\pi_{s,s'} = \sum_a (P^a_{s',s}\pi(s,a)) \tag{2-6}$$

In the same way the reward function can be rewritten.

$$R^\pi_s = \sum_a (R^a_s \pi(s,a)) \tag{2-7}$$

The expected value of returns is known as the value function and is dependent on the current state and policy:

$$V(s)^\pi = \mathbb{E}_\pi(G^t|S^t = s) \tag{2-8}$$

This can also be expressed in a recursive way:

$$V(s)^\pi = \mathbb{E}_\pi(R^{t+1} + \gamma G^{t+1}|S^t = s) \tag{2-9}$$

Including the state transition probabilities and reward function, this results in the famous bellman expectation equation.

$$V(s)^\pi = \sum_a (\pi(s,a))(R^a_s + \gamma \sum_{s'} (P^a_{s,s'}\gamma V(s'))) \tag{2-10}$$

The optimal policy is the policy that maximizes the value function for all states.

$$\pi^*(s) = argmax_\pi V(s)^\pi \tag{2-11}$$

The value function is a powerful measure of the quality of a certain state. However, if values for all states are given, it is still difficult to determine a policy that is an improvement on the current policy. That is because no information about state transition probabilities is included in the value function. A solution to this would be to make the value function also a function of the action. This is called the Q-value.

$$Q(s,a)^{\pi} = \mathbb{E}_{\pi}(G^t|S^t = s, A^t = a) \tag{2-12}$$

The bellman equation for the Q-values is the following:

$$Q(s,a)^{\pi} = R_s^a + \gamma \sum_{s'} P_{s',s}^a \sum_{a'} \pi(a',s')Q(s',a') \tag{2-13}$$

Dynamic programming deals with finding the optimal value function or policy. This can be expressed in the Bellman Optimality equations for V and Q:

$$V^*(s) = max_a(R_s^a + \gamma \sum_{s'} P_{s',s}^a V^*(s')) \tag{2-14}$$

$$Q^*(s,a) = R_s^a + \gamma \sum_{s'} P_{s',s}^a max_{a'} Q^*(s',a') \tag{2-15}$$

Two well known algorithms exist to find the optimal policy in an MDP, value iteration and policy iteration. In value iteration, the values of the MDP are synchronously updated towards the maximum Q-values of that state. The value iteration algorithm is shown in algorithm 1

Algorithm 1: Value Iteration

```
Initialize V(s)
while not converged do
    for s in S do
        for a in A do
            $Q(s,a) = R_s^a + \gamma \sum_{s'} P_{s',s}^a V(s')$
    for s in S do
        $V(s) = max_a Q(s,a)$
```

One disadvantage of value iteration is that the optimal policy can be extracted once the value function is converged to the optimal value function. In Policy Iteration, the policy is iteratively improved by acting greedy on the estimated value function of the previous policy. Policy iteration consists out of two steps: policy evaluation and policy improvement. Policy evaluation estimates the value function under the policy, and policy improvement adjusts the policy to act in the direction of higher values. Policy Iteration is shown in algorithm 2

Algorithm 2: Policy Iteration

```
Initialize V(s)
while $\pi_(s)$ not converged do
    while $V(s)$ not converged do
        for $s$ in $S$ do
            for $a$ in $A$ do
                $V(s) = \sum_a R_s^a \pi(a, s) + \gamma \sum_{s'} P_{s',s}^{\pi} V(s')$
    $\pi_(s) = argmax_a(R_s^a + \gamma \sum_{s'} P_{s',s}^a V(s'))$
```

2-1-3 Sampling the environment

If the state transition probabilities and the reward function is fully known, the optimal policy can be found by planning methods such as value iteration and policy iteration. However often the state transition probabilities and reward function are not known. In this case the optimal policy and value function can still be estimated by sampling the rewards from the environment. One of the most used methods to estimate the value function is Temporal Difference (TD) learning. In TD learning, the value function is updated as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1} - V(S_t))) \tag{2-16}$$

where

$$R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2-17}$$

is called the TD error and

$$R_{t+1} + \gamma V(S_{t+1}) \tag{2-18}$$

is called the TD target.

Temporal difference learning has some bias but a low variance because it bootstraps values from the next state. Another way of estimating the value function is by Monte Carlo learning. In Monte Carlo learning the value of a state is estimated by averaging over all the returns acquired from that state. Monte Carlo learning has a high variance but zero bias since it does not bootstrap (uses real sampled returns compared to estimates in TD learning). Monte Carlo learning converges very slowly to the true value function. A more flexible way of selecting the bias and variance of the method would be to use $TD(\lambda)$. In $TD(\lambda)$ the TD target is a weighted sum of the future estimates of returns G_t. Where:

$$G_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} (\lambda^{n-1} G_t^n) \tag{2-19}$$

and

$$V(S_t) = V(S_t) + \alpha(G_t^{\lambda} - V(S_t)) \tag{2-20}$$

This is also known as forward view TD since it updates the value function with respect to future returns. It is important to note that if λ is equal to zero it corresponds to vanilla temporal difference learning. Equivalently, TD(1) corresponds with Monte Carlo learning. An online variant of $TD(\lambda)$ learning is backward view TD. In backward view TD eligibility traces are kept for each state that is visited. Everytime a new state and value is reached each value is updated according to it's eligibility trace. The backward TD update rule is:

$$E_t(s) = \lambda\gamma E_{t-1}(s) + \mathbb{1}(S_t = s), \tag{2-21a}$$

$$V(s) = V(s) + \alpha\delta_t E_t(s) \tag{2-21b}$$

where δ_t is the TD error. An overview of the different methods is given in figure 2-1

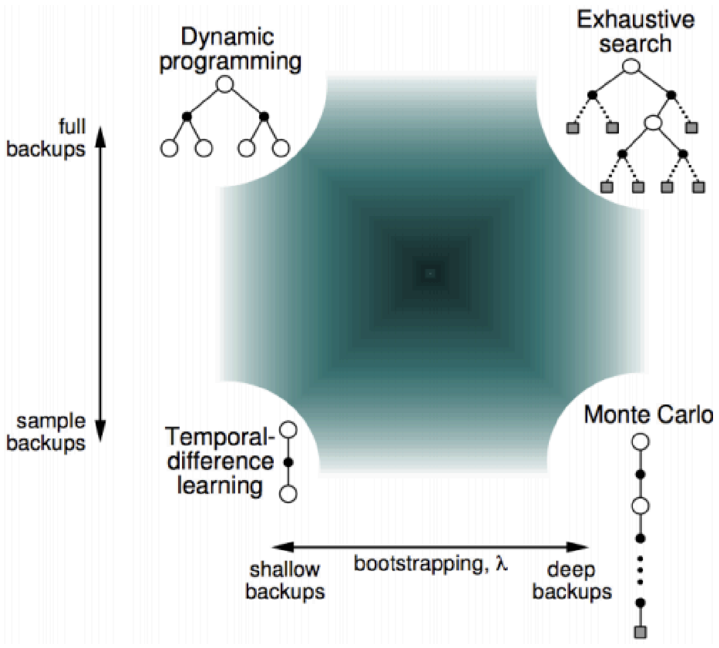


Figure 2-1: Overview of methods for solving MDP's. Taken from Sutton and Barto [2012]

2-1-4 Finding the optimal policy

So far different methods have been evaluated to estimate the underlying value function for a given policy. However a much more valuable property of the MDP is the optimal policy. Different methods exist for finding the optimal policy in discrete problems. A much used method is policy iteration. In policy iteration, the value function is estimate using one of the previous mentioned methods (TD(λ), or Monte Carlo learning for example). After the value funcion is estimated using a few iterations of sampling the environment, the policy is changed by acting ϵ greedy on the value function. Acting ϵ greedy on the value function implies that with a certain randomness factor ϵ, a random action is chosen, otherwise the action which results in the most optimal value function is chosen. However in model free control, the action resulting in the optimal next state value function is not known. In this case Q-value approximation methods must be used. One of these methods is State Action Reward State Action (SARSA). In SARSA, samples of the environment are recorded in state, action, reward, landing state, and subsequent action tuples. Now updates on the Q-values can be made analogous to TD learning.

$$Q(s,a) = Q(s,a) - \alpha(Q(s',a') + R - Q(s,a)) \tag{2-22}$$

Reducing the variance of this method can be achieved in the same manner as in Temporal Difference learning. This results in the SARSA(λ) method. Backwards view SARSA(λ) can be used online and is shown in equation 2-23 and 2-24.

$$E_t(s,a) = \lambda\gamma E_t(s,a) - \mathbb{1}(S_t = s, A_t = a) \tag{2-23}$$

$$Q(s,a) = Q(s,a) + \alpha\delta_t E_t(s) \tag{2-24}$$

where δ_t is the TD error.

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \tag{2-25}$$

The psuedo code for backward view SARSA(λ) is given in algorithm 3

Algorithm 3: backward SARSA(λ)

```
Initialize Q(s,a)
for i in episodes do
    E(s,a) = 0 for all s,a
    Initialize s,a
    while $s \neq terminated$ do
        Take action A, observe R, S'
        Choose A' from S' using $\epsilon$ -greedy policy
        $\delta_t$ = R + $\gamma$Q(S',A') - Q(S,A)
        E(S,A) = E(S,A) + 1
        for all S,A do
            Q(S,A) = Q(S,A) + $\alpha$ $\delta_t$ E(S,A)
            E(S,A) = $\lambda$ $\gamma$ E(S,A)
```

2-1-5 Off-policy learning

The previous methods all considered on-policy learning. That is, the value function or Q-values that are estimated are all corresponding to the policy that is used to sample the environment. Off-policy learning focuses on learning the value function or Q-values of a different policy then the policy that is used to sample. An interesting challenge is to estimate the optimum policy by sampling from a different behavioural policy. An algorithm that converges quickly to the optimum policy by sampling from a different behavioural policy is Q-learning. In Q-learning the Q-values are not updates with respect to the returns corresponding to the future action. Instead the Q-values are updates to the maximum future Q-value ($max(Q(s', a'))$). Q-learning is described in algorithm 4.

Algorithm 4: The Q-learning algorithm

```
Initialize Q(s,a)
for i in episodes do
    Initialize s,a
    while $s \neq terminated$ do
        Take action A, observe R, S'
        Choose A' from S' using $\epsilon$ -greedy policy
        Q(S,A) = Q(S,A) + $\alpha$ (R + $\gamma(max_{A'}Q(S', A')$ - $Q(S,A)$))
```

2-1-6 Function approximation

In the previous methods, the Q-values or value functions were all represented as some sort of look-up table. Each state has a corresponding value or Q-value. This representation can become very cumbersome if the number of states is very large, which is often the case. Consider for example a rubics cube, this has already 43,252,003,274,489,856,000 unique permutations. A way of representing the Q-values and or value functions is to use function approximators. A function approximator for the Q-values would take as input a state and action pair, and outputs the corresponding approximated Q-value. Another benefit of using function approximators is that they can be applied in continuous environments, such as the control of a quadcopter. Linear value function approximation estimate the values by calculating the linear sum of the features multiplied by the function approximator weights.

$$V(S, w) = \mathbf{x}(S, a)^T \mathbf{w} \tag{2-26}$$

To estimate the weights one can apply gradient descent on the mean squared difference of the estimated value function and the sampled value function.

$$\Delta \mathbf{w} = \alpha(V_\pi(S) - V(S, \mathbf{w}))\mathbf{x}(S) \tag{2-27}$$

2-1-7 Policy gradient methods

Thus far, policies have been derived by taking actions based on the value function or Q-value function. One can also explicitly encode a policy. That is, the policy is parametrized by θ

$$\pi = \pi(s, a; \theta) \tag{2-28}$$

Encoding a policy explicitly has the following advantages:

1. Explicit policies can be stochastic, and stochastic policies can behave better in certain environments. Consider the case with ambiguous states, that have a different optimal policy. In this case it is better to have a stochastic policy since the Q-values can only lead to one action. An example of such an environment is given in figure 2-2

2. Encoding a policy explicitly can be applied to continuous environments. Using Q-values or Q-value approximation involves taking the maximum of the actions. If the action space is continuous or very large, calculating this maximum might be very expensive. In these environments it is much more efficient to explicitly parametrize the policy.

3. An explicit policy is able to efficiently represent the action in environments with a large amount of states. In addition it is able to generalize between certain states.

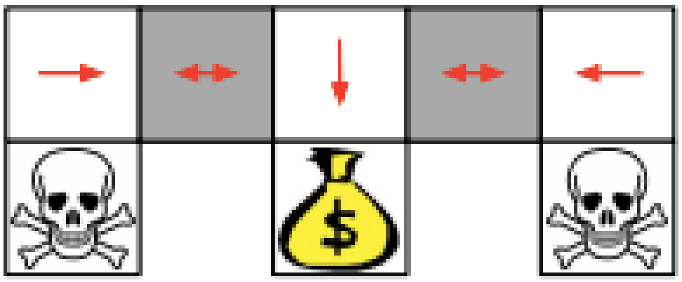


Figure 2-2: The grey areas are indistinguishable states to the agent. A deterministic policy based on the Q-values of the states will therefore always send the agent to the right or the left. This means that they are either stuck in the right or left part of the environment. A stochastic explicit policy can always terminate the episode at the prize. Taken from Silver [2015]

The optimal policy can be found by maximizing some objective function that measures the quality of the policy. The policy gradient is defined as the gradient of the policy with respect to this objective function.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log(\pi_\theta(s,a))Q_\pi(s,a)] \tag{2-29}$$

A very simple implementation of training a policy using policy gradients is Monte Carlo Policy Learning (MCPL) or the REINFORCE algorithm. In MCPL (REINFORCE) the sample returns of the states are used as an unbiased estimate of the Q-values. The algorithm is shown in algorithm 5

Algorithm 5: Monte-Carlo Policy Learning (REINFORCE)

```
Initialize $\pi_\theta(s,a)$
for $i$ in episodes do
    Sample $(s_1, a_1, r_1, s_2, a_2, r_2...., s_T, a_T, r_T)$
    for s,a,r in samples do
        $\theta \leftarrow \alpha \nabla_\theta log(\pi_\theta(s_t, a_t))v_t$
```

Because MCPL involves a Monte-Carlo sampling estimate of the Q-values, it has a high variance. The variance of policy gradient methods can be reduced by using so-called actor critic methods. In actor-critic methods the actor is defined by a parametrized policy, and the critic is defined by a parametrized value function or Q-value function. The critic is used as a biased estimate of the policy gradient as shown in equation 2-30

$$\Delta\theta = \alpha \nabla_\theta log(\pi_\theta(s,a))Q_\pi(s,a) \tag{2-30}$$

A schematic representation of actor-critic methods is shown in figure 2-3

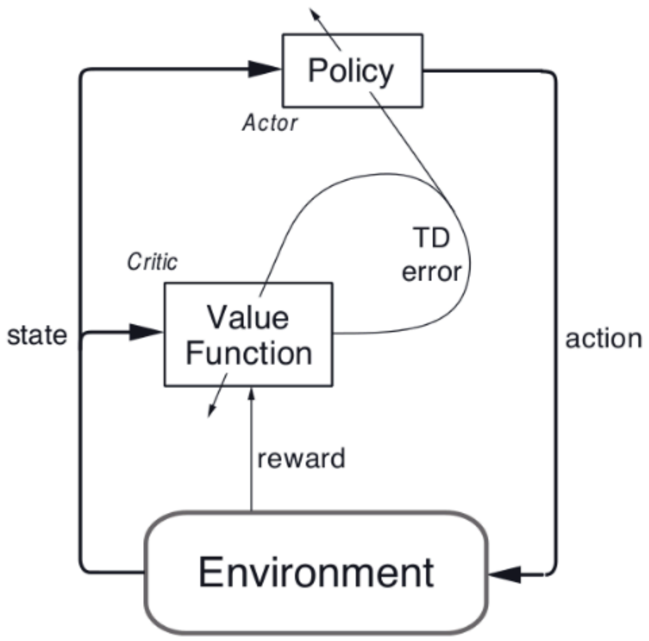


Figure 2-3: A schematic representation of an actor-critic method where the temporal difference error from the critic is directly used to calculate the policy gradient for the actor. Taken from Sutton and Barto [2012]

2-1-8 Model Based Reinforcement Learning

Intuitively one of the advantages of reinforcement learning is that it does not need a model in order to determine the correct actions. It simply learns from experience which actions lead to the highest average reward. This model-free property of reinforcement learning is a very appealing property. However it is also inherently connected to the problem of dimensionality in reinforcement learning. Using the experience to construct a value function of an environment might be a very time intensive process, whereas using a simple model to simulate the rewards and subsequent states might be a much better way of capturing the information about the value of a state.

2-2 Challenges in reinforcement learning

Reinforcement learning is still a very active field of research. This chapter will focus on key challenges in reinforcement learning that the research community is focusing on.

2-2-1 Curse of dimensionality

Many real world problems can be represented by an incredibly large, if not infinite set of states. Function approximation offer a solution in order to generalize between states and condense the large dimensionality to a single parametrized function. However one of the biggest challenges in reinforcement learning is the sampling problem. In order to estimate the optimal policy for a given problem, the value function for each state has to be estimated in some way, in reinforcement learning this is done by sampling the environment. Sampling the environment can be a very time-costly process. In addition, sampling the environment might be very dangerous in real time applications. For example, using model-free reinforcement learning on a quadcopter can, and most likely, will result in a lot of crashes. Using different forms of learning can reduce the curse of dimensionality. For example imitation learning can use the knowledge of expert behaviour in order to efficiently or safely sample the environment. Another promising solution is model-based reinforcement learning. By interacting with the environment, model-based reinforcement learning builds up a model of the transition function and reward function. This model can now be used to simulate trajectories, and apply reinforcement learning in an off-line way, as is the case in DYNA-Q Sutton [1990]. In addition the model can be used to get a better estimate of the policy gradient, as is done in MLAC Grondman [2015].

2-2-2 To explore or exploit?

This issue is strongly related to the previous issue. In order to improve a policy, the policy has to pick actions that lead to higher rewards. However this process is based on the value function of the environment, and this value function in turn is a sampled estimate of the true value function. So in order to correctly pick the right action, one first has to explore in order to estimate the correct value function. This is called the exploitation/exploration dilemma. A lot of research has been done on this field and a possible solution is to use upper confidence bounds Auer [2003].

2-2-3 Credit assignment problem

One of the main difficulties in reinforcement learning is to estimate the values of the states. One can sample returns from a policy and use those samples as unbiased estimates for the value of the states. However this is a very noisy estimate of the value function. Updating values of states by bootstrapping onto sampled values of upcoming states reduces the variance of this process, as is done in TD learning. Also combination of Temporal Difference and Monte Carlo methods exist, as in TD(λ) learning. All these methods try to update the values of states by some estimate of the return. Some use bootstrapping as an estimate of the return, and some use the return of a sampled trajectory as an estimate of the return. The problem here is that sampling trajectories is a stochastic process, and it is difficult to determine which state was responsible for the current return. This is called the credit assignment problem.

2-3 Overview of reinforcement learning architectures

Many reinforcement learning algorithms exist to date, and they can be structured under different dimensions. This chapter will explain these dimensions of reinforcement learning, and give a final overview of existing reinforcement learning architectures, classified under these dimensions.

Model based reinforcement learning

Intuitively one of the advantages of reinforcement learning is that it does not need a model in order to determine the correct actions. It simply learns from experience which actions lead to the highest average reward. This model-free property of reinforcement learning is a very appealing property. However it is also inherently connected to the problem of dimensionality in reinforcement learning. Using the experience to construct a value function of an environment might be a very time intensive process, whereas using a simple model to simulate the rewards and subsequent states might be a much better way of capturing the information about the value of a state. This is exactly done in DYNA architectures Sutton [1990]. In addition, a model can be used to estimate the policy gradient, as is done in MLAC.

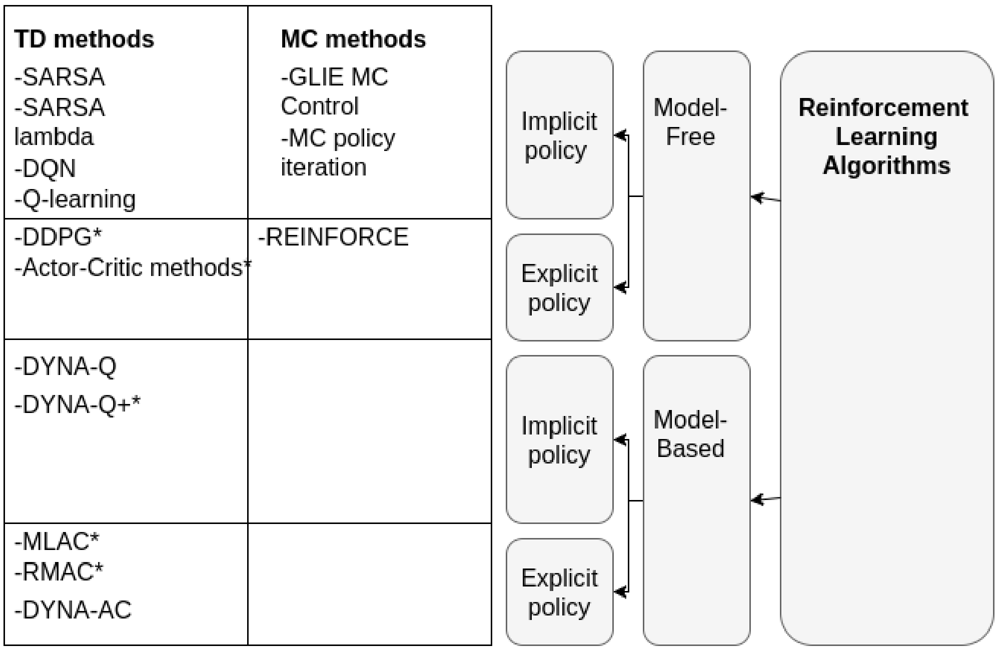
Explicit or Implicit policy

As explained in section 2-1, a policy can be explicitly or implicitly captured. Capturing a policy explicitly has the advantage of being able to model stochastic policies, or modelling continuous action spaces. In explicit methods, a policy gradient has to be estimated to adjust the policy in the direction of higher returns. In actor-critic methods a critic is used to estimate the values of the MDP, this critic is then used to estimate the policy gradient for adjusting the policy. Model Learning Actor Critic is an algorithm that does exactly this.

Temporal difference or Monte Carlo learning

Estimating the values or Q-values under a MDP can be a very noisy process. Monte Carlo methods estimate these values by sampling the exact episodic returns, and then averaging these samples. This process has a high variance but is unbiased. TD methods 'bootstrap' the sampled reward and the value of the upcoming state to update the current value, as is shown in equation 2-16. TD methods are much less noisy then Monte Carlo methods, however convergence to the optimal value function is not always guaranteed.

In figure 2-4 a structural overview based on these dimensions is shown.



TD methods	MC methods
-SARSA -SARSA lambda -DQN -Q-learning	-GLIE MC Control -MC policy iteration
-DDPG* -Actor-Critic methods*	-REINFORCE
-DYNA-Q -DYNA-Q+*	
-MLAC* -RMAC* -DYNA-AC	

Figure 2-4: Structural identification of several reinforcement learning architectures (SARSA = State Action Reward State Action, DQN = Deep Q Network, DDPG = Deep Deterministic Policy Gradients, MLAC = Model Learning Actor Critic, RMAC = Reference Model Actor Critic, DYNA-AC = DYNA Actor-Critic, GLIE = Greedy in the Limit with Infinite Exploration).

2-4 From discrete to deep continuous reinforcement learning

This chapter is dedicated to showing some of the evolutionary steps that reinforcement learning algorithms have taken in order to come to the current state of research. It serves as a way to understand why there is a need for deep continuous reinforcement learning, what it's challenges are, and which fields are currently being researched. First a very simple discrete problem will be shown. This problem will be used to show the effects of bootstrapping (in monte-carlo vs temporal-difference methods), and credit assignment. Three different algorithms will be shown, Monte-Carlo learning, SARSA, and SARSA(λ) learning. Subsequently a more complex continuous problem will be shown, that of an inverted pendulum. This problem is first solved with a deep neural network with discrete action spaces (Deep Q Network (DQN)). Next the problem is solved using the MLAC algorithm, that uses linear function approximators but is able to model continuous actions. Finally an algorithm called DDPG is used to solve the problem with continuous actions and deep function approximators.

2-4-1 Monte-Carlo, Temporal-difference learning and TD-lambda algorithms for control

A very simple algorithm in reinforcement learning is the Monte-Carlo control algorithm. Monte-Carlo control is a value iteration method that uses unbiased samples from the environment to update the Q-values of each state-action pair encountered. The algorithm for Monte-Carlo control is shown in algorithm 6.

Algorithm 6: Monte Carlo control

```
Initialize Q(s,a)
Initialize N(S,A) == 0 for all s $\in$ S and a *in* A
**for** *i in episodes* **do**
    Randomly Initialize start state and action (S,A)
    Initialize empty list [(S,A,R)] recording the State Action Reward history
    **while** $s \neq terminated$ **do**
        Take action A, observe R, S'
        append (S,A,R) to State Action Reward history Choose A' from S' using $\epsilon$ -greedy
          policy
        N(S,A) = N(S,A) + 1
    **for** *(S,A,R) in State Action Reward history* **do**
        Q(S,A) =Q(S,A) + $\frac{G-Q(S,A)}{N(S,A)}$
    $\epsilon$ = 1 / i
```

In Silver [2015], it is claimed that Monte-Carlo control converges to the optimum policy if it is Greedy In The Limit With Infinite Exploration (GLIE). GLIE implies that the ϵ factor approaches zero in the limit.

Example 2.1: Monte-Carlo Control applied to a Maze

A commonly used example for testing reinforcement learning algorithms, is a maze environment. The maze environment used in this example is shown in figure 2-5. It is the goal of the agent to find the shortest path from the blue starting position to the green goal position. The states of the maze environment are simply the positions of the agent. At each position, the agent can move to the left, up, right or down depending if there is an obstacle or not. If the agent hits an obstacle by performing that action, the new state of the agent is simply the old state of the agent. Hyperparameters used for this example can be found in table A-1.

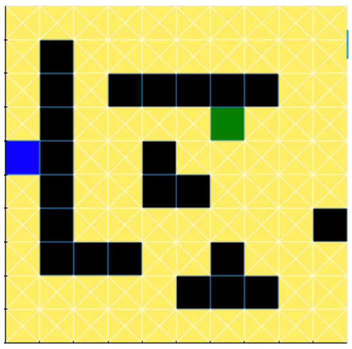


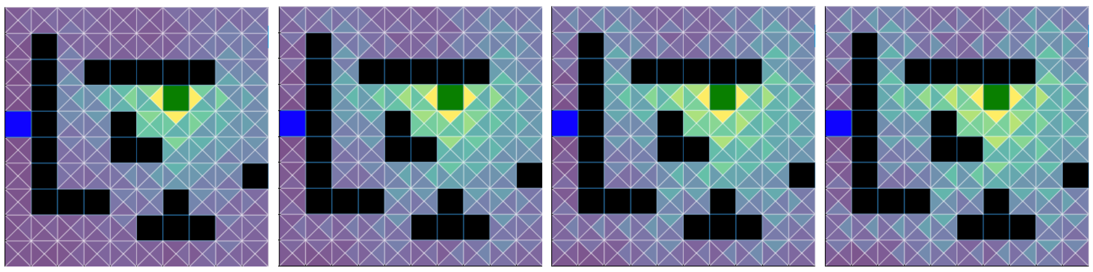
Figure 2-5: Initial state of the maze, the blue square represents the start position, and the green square represents the goal position

The rewards of the maze are the following:

Table 2-1: Rewards for the maze environment

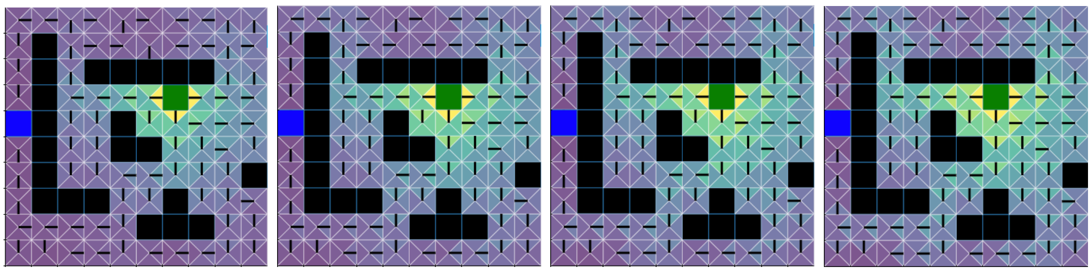
state	**reward**
not goal state	-1
goal state	100

Figure 2-6 shows the evolution of the state-action values as the algorithm progresses. Additionally, in figure 2-7 the maximum Q-value actions are shown with black arrows.



(a) Q-values after 1000 episodes **(b)** Q-values after 2000 episodes **(c)** Q-values after 3000 episodes **(d)** Q-values after 4000 episodes

Figure 2-6: Evolution of Q-values for the Monte Carlo control algorithm. Lighter colors mean higher Q-values.



(a) max Q-values after 1000 episodes

(b) max Q-values after 2000 episodes

(c) max Q-values after 3000 episodes

(d) max Q-values after 4000 episodes

Figure 2-7: Evolution of max Q-values for the Monte Carlo control algorithm. Hyperparameters are the same as in figure 2-6.

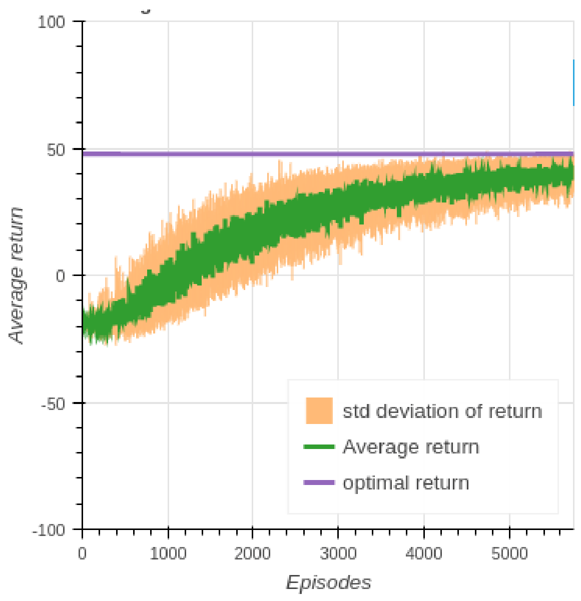
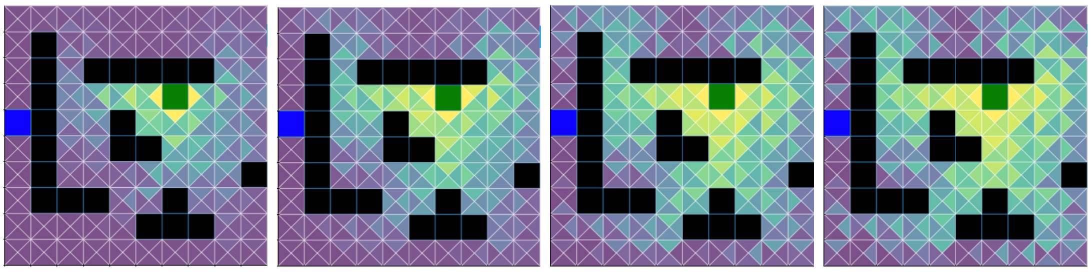


Figure 2-8: Evolution of the average return and the standard deviation of the average return from the blue start square to the goal. This average return is calculated by simulating 10 episodes from the starting position. The return never reaches the optimal return since the stochastic policy is evaluated until ϵ is equal to 0.1.

A big disadvantage of the Monte-Carlo control algorithm is that it has a high variance in estimating the Q-values. It therefore takes a long time for the algorithm to converge to the optimal policy. TD methods offer a solution to this problem. As explained in chapter 2-1, TD learning updates the Q-values by 'bootstrapping' onto the Q-values experienced in the next state and action. This lowers the variance. However the Q-values are now a biased estimate of the true Q-values. In addition, because TD methods use bootstrapping for estimating the Q-values, it can be used in on-line methods. A well known example of an on-line temporal difference learning algorithm is the SARSA algorithm. The SARSA algorithm is shown in example 2.2

Example 2.2: SARSA applied to a Maze

One of the most simple, yet powerful TD algorithms is SARSA. In SARSA, every time a state and action is executed, the current Q-value get's updated by as a function of the received reward and the Q-value of the upcoming state-action value, as shown in equation 2-22. The SARSA algorithm is applied to the maze problem from example 2-4-1. The evolution of the Q-values and maximum Q-values can be seen in figure 2-9 and 2-10 respectively.



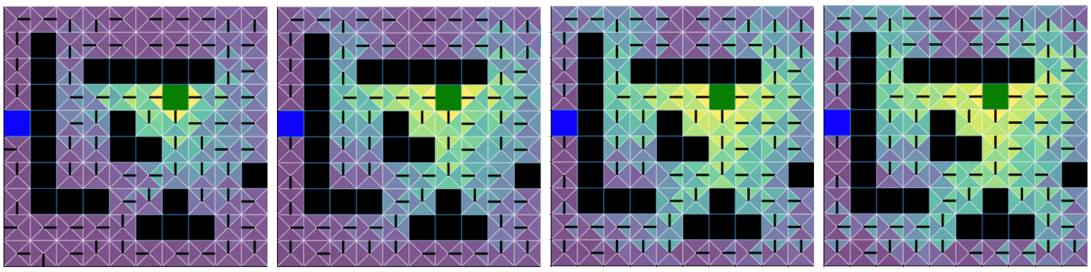
(a) Q-values after 500 episodes

(b) Q-values after 1000 episodes

(c) Q-values after 1500 episodes

(d) Q-values after 2000 episodes

Figure 2-9: Evolution of Q-values for the SARSA algorithm. Lighter colors mean higher Q-values.



(a) max Q-values after 500 episodes

(b) max Q-values after 1000 episodes

(c) max Q-values after 1500 episodes

(d) max Q-values after 2000 episodes

Figure 2-10: Evolution of max Q-values for the SARSA algorithm. Problem specific hyperparameters are the same as in figure 2-6.

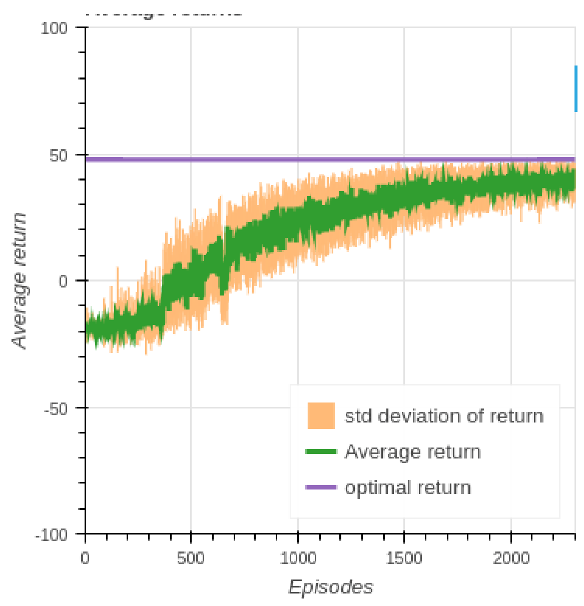
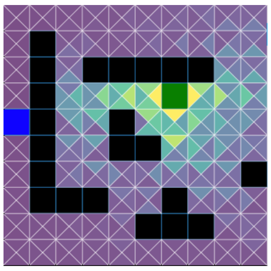


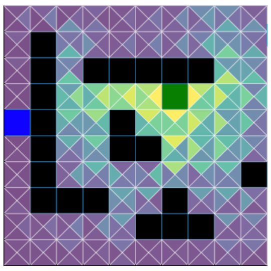
Figure 2-11: Evolution of the average return and the standard deviation of the average return from the blue start square to the goal.This average return is calculated by simulating 10 episodes from the starting position. The return never reaches the optimal return since the stochastic policy is evaluated up until ϵ is equal to 0.1.

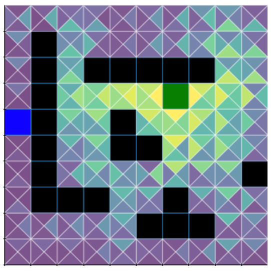
One interesting property to notice about the SARSA algorithm, is that it shows the power of value iteration methods over policy iteration methods. Although the Q-values take a long time to converge to the optimal Q-values, the policy converges rather fast to the optimal policy. Another interesting fact is that SARSA slowly propagates the correct qvalues towards the begin state of the agent. This is because at the beginning, the temporal difference error is highest near the goal position since the reward is high there. Only at the next episode, the newly learned high Q-values surrounding the goal position can be propagated further backward. The algorithm cannot determine that earlier states and actions also contributed towards the final return. This is also called the credit assignment problem. A way to improve the propagation of high temporal difference errors is to use eligibility traces, as mentioned earlier before in chapter 2-1. SARSA(λ) backwards is an extension on SARSA that implements eligibility traces to reduce the credit assignment problem. The SARSA(λ) backward algorithm is shown in algorithm 3. In example 2.3 the SARSA(λ) backward algorithm is evaluated on the maze environment.

Example 2.3: SARSA(λ) backwards applied to a Maze

A more efficient way to assign credit to different states is the SARSA(λ) algorithm. In figure 2-12 the evolution of Q-values can be seen. Comparing figure 2-12 to 2-9, it can be seen that high Q-values propagate quicker from the high reward at the goal state to the begin state of the maze.


(a) Q-values after 150 episodes


(b) Q-values after 300 episodes


(c) Q-values after 450 episodes

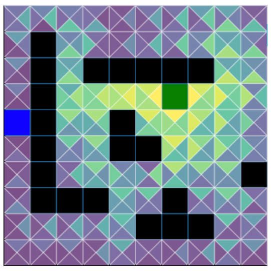
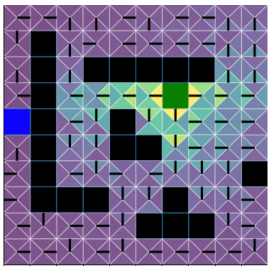
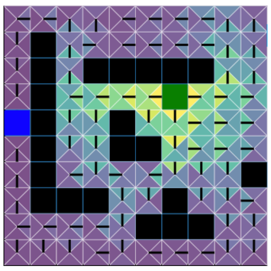
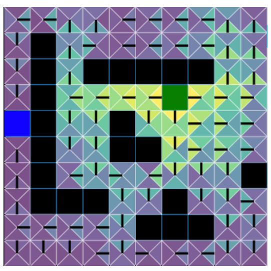

(d) Q-values after 550 episodes

Figure 2-12: Evolution of Q-values for the SARSA(λ). Lighter colors mean higher Q-values.


(a) max Q-values after 150 episodes


(b) max Q-values after 300 episodes


(c) max Q-values after 450 episodes

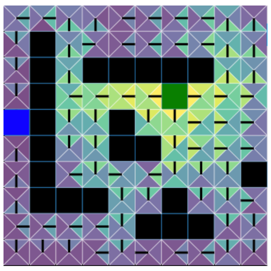

(d) max Q-values after 550 episodes

Figure 2-13: Evolution of max Q-values. Problem specific hyperparameters are the same as in figure 2-6.

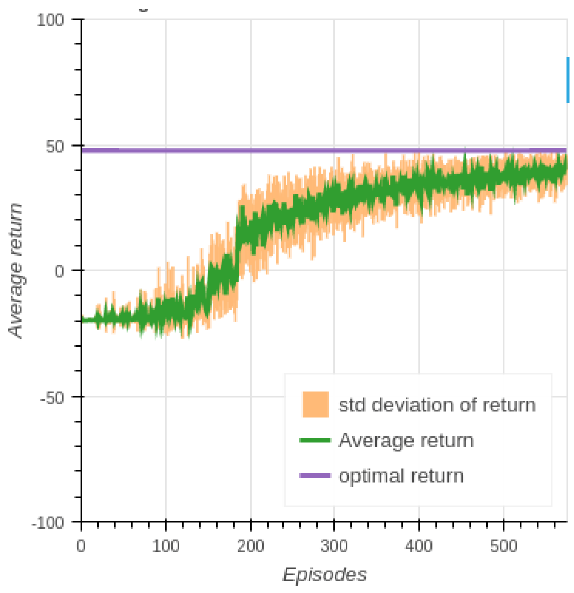


Figure 2-14: Evolution of the average return and the standard deviation of the average return from the blue start square to the goal.This average return is calculated by simulating 10 episodes from the starting position. The return never reaches the optimal return since the stochastic policy is evaluated up until ϵ is equal to 0.1.

2-4-2 Continuous control with reinforcement learning

The maze example from the previous chapter shows a domain where reinforcement learning offers an efficient solution towards an optimal policy. In the maze environment the state-space representation is discrete, however in many real world environments, the state-space representation is continuous. For example, driving a car or flying a quadrotor are both domains where the input states are continouus. In these continuous domains, the need for function approximators for estimating the Q-values becomes evident. As explained in chapter 2-1, many real-world problems have a very large number of states, function approximators are an efficient solution to map these states to Q-values. In addition, function approximators make it possible to generalize between states.

2-4-3 Neural Networks

A candidate for function approximators in reinforcement learning are neural networks. Neural networks are networks that consist out of layers of neurons. each individual neuron receives an affinely transformed output of the previous layer as an input. The output of the neuron is the result of applying an activation function to the input.

$$y_i = act((\sum_{j=0}^{N}(w_{ij}y_j) + b_i)) \tag{2-31}$$

where j is the index of the neuron of the previous layer and i the index of the neuron of the current layer. The weights w_{ij} and biases b_i are the parameters of the network.

Since the invention of the backpropagation of errors Rumelhart et al. [1986], neural networks have gained a lot of interest in the research community. Especially stochastic gradient descent has proven to be an efficient way of optimizing the parameters of a neural network. Stochastic gradient descent minimizes the loss of a neural network by applying gradient descent with respect to the loss over the parameters of the network:

$$\theta = argmin_\theta \frac{1}{N} \sum_{i=0}^{N} l(x_i, \theta) \tag{2-32}$$

where $l(x_i, \theta)$ is a loss metric, such as cross-entropy loss or squared error. This results in the following parameter updates:

$$\theta = \theta - \alpha * \frac{\delta l(x_n))}{\delta \theta} \tag{2-33}$$

where n are the examples in a minibatch.

The power of neural networks lies in the fact that it uses hierarchical distributed representations of concepts as opposed to localist representations Geoffrey E. Hinton [1986], Bengio [2009]. In addition, deep neural networks are able to learn their own features in comparison to shallow function approximators. These two attributes of neural networks make them very strong function approximators that have high generalization capabilities in very high dimensional data. The development of convolutional neural networks Lecun et al. [1998] have brought the method to the state-of-the art in image classification Krizhevsky et al. [2012]. Research in deep learning has gained large interest and over the years, many new improvements and features have been developed that take the performance to an even higher level. In particular, the most imporant advancements are that of parametric rectified linear units He et al. [2015], and batch normalization Ioffe and Szagedy [2015]. Using conventional activation functions such as tanh or sigmoid, gradient descent often results in exploding gradients. Rectified linear units, shown in equation 2-34, do not have the problem of exploding gradients since their derivative with respect to their input is either one or zero.

$$ReLu(x) = max(0, x) \tag{2-34}$$

In addition, Rectified Linear units encourage more sparse networks, offering a better disentanglement of factors of variation in the data Bengio [2009].

Both batch normalization and rectified linear units have greatly improved and encouraged optimization of deeper networks. Batch normalization is an efficient way of reducing the internal covariate shift in the layers of a deep neural network. Batch normalization 'whitens' the input data in each layer by adjusting the data to have zero mean and unit variance.

2-4-4 Using neural networks as function approximators in reinforcement learning

Recently, neural networks have succesfully been implemented as a Q-value function approximator in reinforcement learning Mnih et al. [2013]. Mnih et al. [2013] designed an algorithm called DQN that is able to find policies using images as input data. The performance of the DQN was tested on many atari games, often outperforming human performance. Using deep neural networks as function approximators in reinforcement learning comes with several challenges. First of all, in order to optimize the log-likelihood of the parameters of a neural network, the input data needs to be Independent and Identically Distributed (IID). In reinforcement learning, this condition is often violated, since samples from the environment are highly correlated with one another in a chronological fashion. In addition, the target values of the network have to be generated by the network itself. This implies that unwanted feedback loops are created and training might diverge. DQN solve these problems by using Q-learning with experience replay. In DQN two networks are used. One network is used to generate the target values that the other network is trained on. After an amount of training steps, the first network is updated with the parameters of the second network. Using a separate Q-network to estimate the target Q values also shows the need for Q-learning, since the values of a different older policy are used to optimize the current policy. The DQN algorithm is shown in 7.

Algorithm 7: Deep Q network

```
Initialize replay memory D
Initialize critic networks C and C' with random weights
for $i$ in episodes do
    Initialize start state and action S
    while $s \neq terminated$ do
        with probability $\epsilon$ select action A, otherwise select A = argmax(Q(s,a))
        Take action A, observe R, S'
        append (S,A,R,S') to the replay memory
        sample random minibatch of N transitions from D
        set $y_j = R_j + \gamma * max(C'(S_j, A_j))$
        perform gradient descent step on C with calculated target values
        $\epsilon$ = $\epsilon$-discount-factor * $\epsilon$
```

An example of the DQN algorithm applied on an inverted pendulum is shown in example 2.4.

Example 2.4: DQN applied to an inverted pendulum

One very popular problem in reinforcement learning is that of an underactuated inverted pendulum. In the underactuated inverted pendulum problem, the agent must search for a policy that swings the pendulum to the upward position and is able to keep it balanced from there. The pendulum needs to use momentum to swing the pendulum to the top position, since it does not have enough torque to simply rotate it to the top. The states in the inverted pendulum problem are the angle θ and the angular speed ω. The angle is defined to be $-\pi$ at the bottom, inducing clockwise rotation increases the angle to zero at the top and to π at the bottom again. Applying positive torque induces clockwise rotation on the pendulum. The equations of motion for the pendulum are as follows,

$$\dot{\theta} = \omega \tag{2-35}$$

$$\dot{\omega} = sin(\theta) * \frac{g}{l} + \frac{1}{ml^2}u - \frac{c\omega}{ml^2} \tag{2-36}$$

Here l is the length of the pendulum, m is the mass of the pendulum, g is the gravitational constant, and c is a friction constant. The parameters used in this example are the following:

Table 2-2: physical parameters used in the inverted pendulum example

m	g	l	c	u
1	9.8	1	0.1	[-5,5]

Runge Kutta integration with a time-step of 0.05 was used to simulate the pendulum.

The problem environment is shown in figure 2-15.

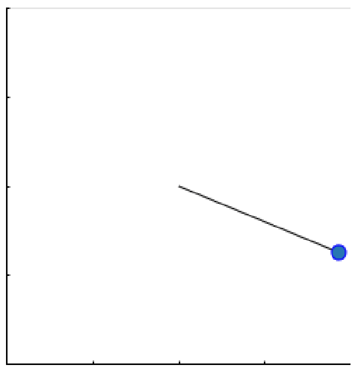
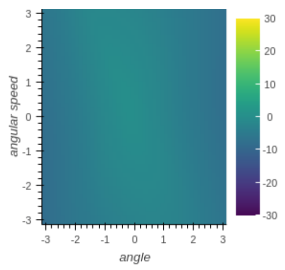


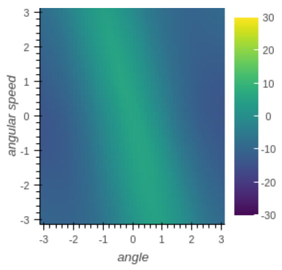
Figure 2-15: Example state of the inverted pendulum

Hyperparameters used in this example can be found in table A-4.

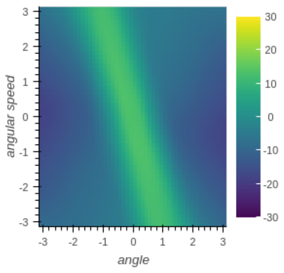
The evolution of the Q-values and the policy can be seen in figure 2-16 and 2-17 respectively. After about 100 episodes the agent has found a policy that swings the pendulum up and keeps it balanced at the top position.



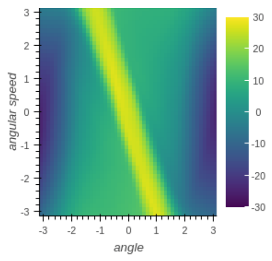
(a) cw Q-values after 20 episodes



(b) cw Q-values after 40 episodes

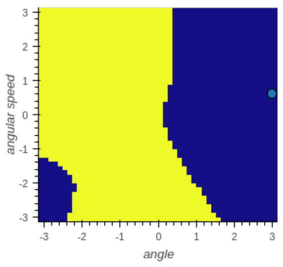


(c) cw Q-values after 60 episodes

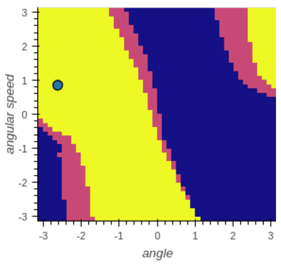


(d) cw Q-values after 100 episodes

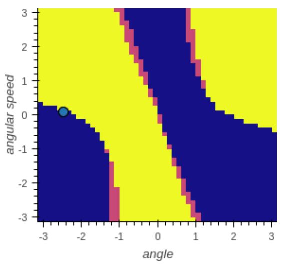
Figure 2-16: Evolution of clockwise Q-values for the DQN algorithm applied on the underactuated pendulum problem. The skewed light band that starts to appear after 100 episodes is logical; having a positive angular rate and being located just before the top position is a good state



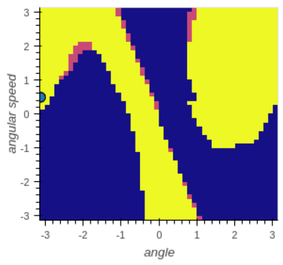
(a) Policy after 20 episodes



(b) Policy after 40 episodes



(c) Policy after 60 episodes



(d) Policy after 100 episodes

Figure 2-17: Evolution of the maximum Q-values over episodes. Yellow colors mean clockwise torque, red values mean no torque, and blue values mean counterclockwise torque.

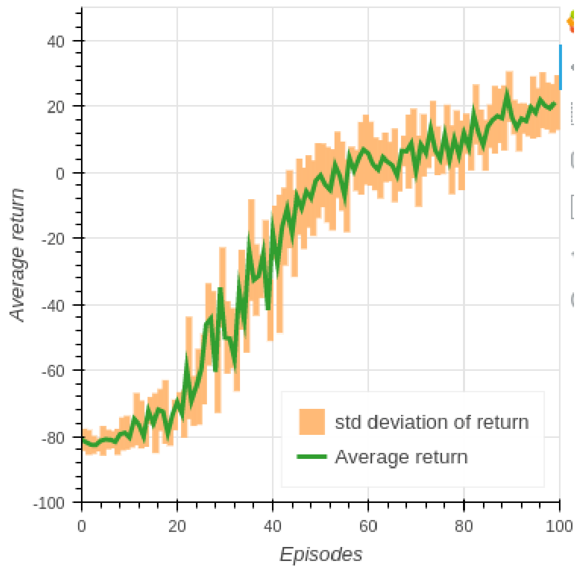


Figure 2-18: Evolution of the average return and the standard deviation of the average return. This value is calculating by simulating 10 episodes from the initial start position (pendulum in rest position). The return never reaches the optimal return since the stochastic policy is evaluated up until ϵ is equal to 0.1.

2-4-5 Model Learning Actor Critic

In the previous sections, the benefits of using function approximators for Q-value estimation is shown. However the action space is still discrete in these examples. Several methods have been developed to account for continuous action spaces in reinforcement learning problems. In Grondman [2015] a model-based approach called MLAC is used for achieving continuous control. In MLAC, the actor, critic and model are all modeled by function approximators. The gradient of the model is combined with the gradient of the critic and the gradient of the actor in a way that the actor moves its policy to a state that leads to a higher value estimation from the critic. The MLAC algorithm is shown in algorithm 8. The advantage of the MLAC algorithm with respect to standard Actor-Critic algorithms is that MLAC captures the process model explicitly. This means that a direct estimation can be made on how to adjust the parameters of the actor to achieve higher returns Grondman [2015]. Local function approximators can be used for MLAC, such as Local Linear Regression (as used in the thesis of Helmer et al. [2018], or radial basis functions.

In example 2.5 the MLAC algorithm applied on the underactuated pendulum is shown. Radial basis functions are used as a function approximator.

Algorithm 8: Model Learning Actor Critic

```
Initialize Critic $V_\theta$, Actor $\pi_\theta$ and Model $M_\theta$ function approximators
for $i$ in episodes do
    Initialize S and A
    while $s \neq$ terminated do
        Take action A, observe R, S'
        $\delta$ = R + $\gamma V_\theta$(S') - $V_\theta$(S)
        A' = $\pi_\theta(S')$
        //Update Actor//
        $\theta_\pi = \theta_\pi + \alpha(\nabla_u \rho(S,A)^T + \gamma \nabla_x V(S')^T \nabla_u M(S,A)) \nabla_\theta \pi_\theta(S)$
        //Update Model//
        $\theta_M = \theta_M + \alpha(S' - M_\theta(S,A)) \nabla_\theta M_\theta(S,A)$
        //Update Critic//
        $z = \lambda\gamma z + \nabla_\theta V_\theta(S)$
        $\theta_M = \theta_M + \alpha\delta z$
        choose exploration $\Delta A \sim \mathcal{N}(0, \sigma^2)$
        $A = A' + \Delta A$
        $\sigma = \sigma * \sigma$-discount-factor
```

Example 2.5: Model Learning Actor Critic applied to an inverted pendulum

The MLAC can also be applied on the inverted pendulum problem. The parameters and model of the pendulum in this example are the same as in example 2.4. Radial basis functions are used as a local function approximator for the critic, actor and model. The parameters to learn are the weights for each local basis function, for example the output of the critic is:

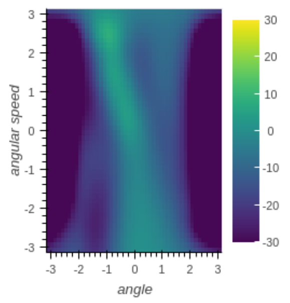
$$V_\theta(x) = \theta^T \phi(x) \tag{2-37}$$

Where ϕ is a normalized radial basis function:

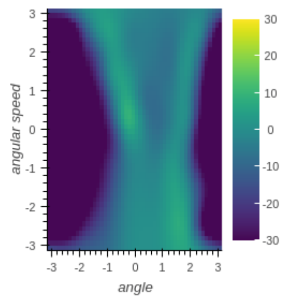
$$\phi_i = \frac{\phi_i'(x)}{\sum_j \phi_j'(x)} \tag{2-38}$$

The limit of the torque in the actor is implemented by having a tanh function applied to equation 2-37.

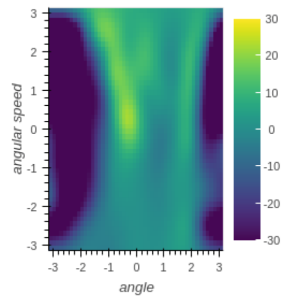
Hyperparameters used in this example can be found in table A-5. The evolution of the Q-values and the policy can be seen in figure 2-19 and 2-20 respectively. It must be noted that the algorithm with the current hyperparameters is able to learn the behaviour to swing the pendulum up, but not able to balance the pendulum at the top. It might be that the hyperparameters are not optimal yet, as the policy does converge somehow to the optimal policy in example 2.4. Another reason could be that the implementation of the final tanh function in the actor results in very small gradients.



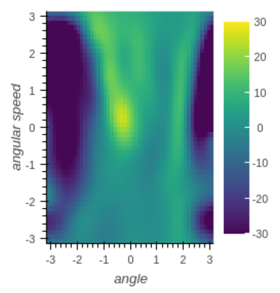
(a) Q-values for maximum positive torque after 100 episodes



(b) Q-values for maximum positive torque after 200 episodes

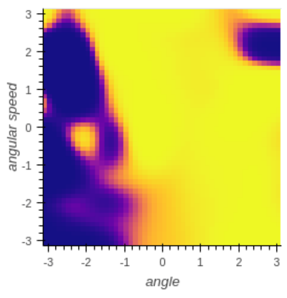


(c) Q-values for maximum positive torque after 300 episodes

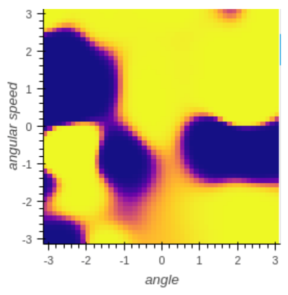


(d) Q-values for maximum positive torque after 400 episodes

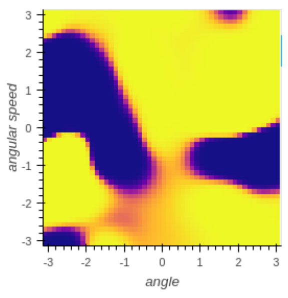
Figure 2-19: Evolution of Q-values for maximum positive torque of the MLAC algorithm applied on the underactuated pendulum problem.



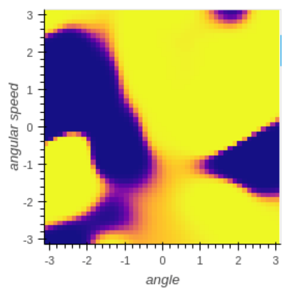
(a) Policy after 100 episodes



(b) Policy after 200 episodes



(c) Policy after 300 episodes



(d) Policy after 400 episodes

Figure 2-20: Evolution of the policy of the MLAC algorithm. Yellow colors mean clockwise torque, red values mean no torque, and blue values mean counterclockwise torque.

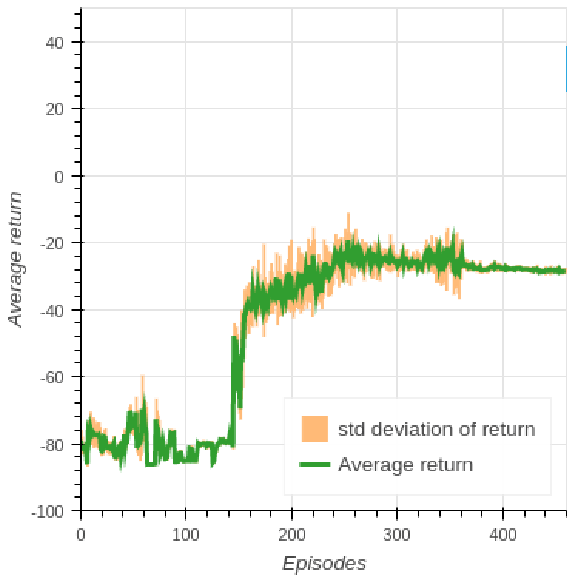


Figure 2-21: Evolution of the average return and the standard deviation of the average return. This value is calculating by simulating 10 episodes from the initial start position (pendulum in rest position). The return never reaches the optimal return since the stochastic policy is evaluated up until ϵ is equal to 0.1.

It can be seen from example 2.5 that it takes a lot of episodes for the algorithm to converge to a policy that is similar to the final policy in example 2.4. This is most likely due to the torque limit being implemented by a final activation function that is a tanh, and hyperparameters not being optimal.

2-4-6 Deep Deterministic Policy Gradient

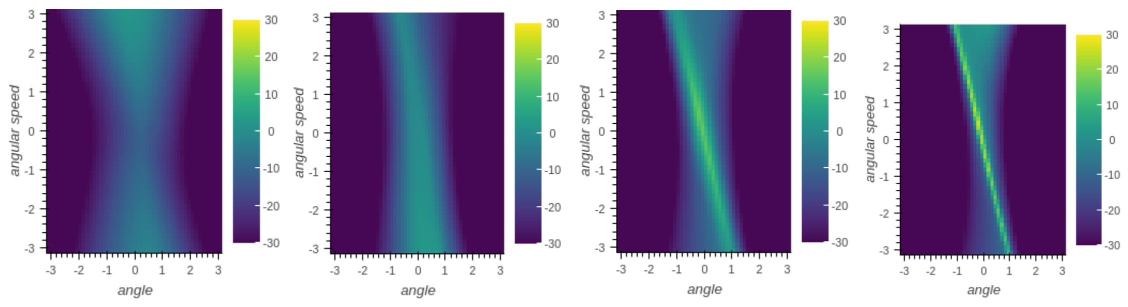
The big disadvantage of Deep Q Networks Mnih et al. [2013] is that it learns an implicit policy by taking the actions of the maximum Q-values. Actions can therefore only be modeled in a discrete way. The work of Lillicrap [2016] resulted in an algorithm that is able to model continuous control with deep reinforcment learning. DDPG uses an actor critic approach to estimate a deterministic policy gradient. DDPG combines DQN Mnih et al. [2013], with the work of Silver et al. [2014]. In Silver et al. [2014] a model free method of estimating a policy gradient is derived. Similar to DQN, DDPG uses two critics. In addition, DDPG implements two actors, where one actor is an old version of the new actor. The DDPG algorithm is shown in algorithm 9.

Algorithm 9: Deep Deterministic Policy Gradient

```
Initialize replay memory D
Initialize critic networks C and C', and actors $\mu$ and $\mu'$ with random weights
for $i$ in episodes do
    Initialize start state S
    Initialize $\mathcal{N}$ while $s \neq terminated$ do
        select action A $= \mu + \mathcal{N}$ according to the current policy and exploration noise
        Take action A, observe R, S'
        append (S,A,R,S') to the replay memory
        sample random minibatch of N transitions from D
        set $y_j = R_j + \gamma * C'(s_{j+1}, \mu'(s_{j+1}))$
        update critic by minimizing the loss $L = \frac{1}{N}\sum_j y_j - C(s_{j+1})$
        update actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J = \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
        update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
```

Example 2.6: Deep Deterministic Policy Gradients applied to an inverted pendulum

Hyperparameters used in this example can be found in table A-6. The evolution of the Q-values and the policy can be seen in figure 2-22 and 2-23 respectively. After about 60 episodes the agent has found a policy that swings the pendulum up and keeps it balanced at the top position.



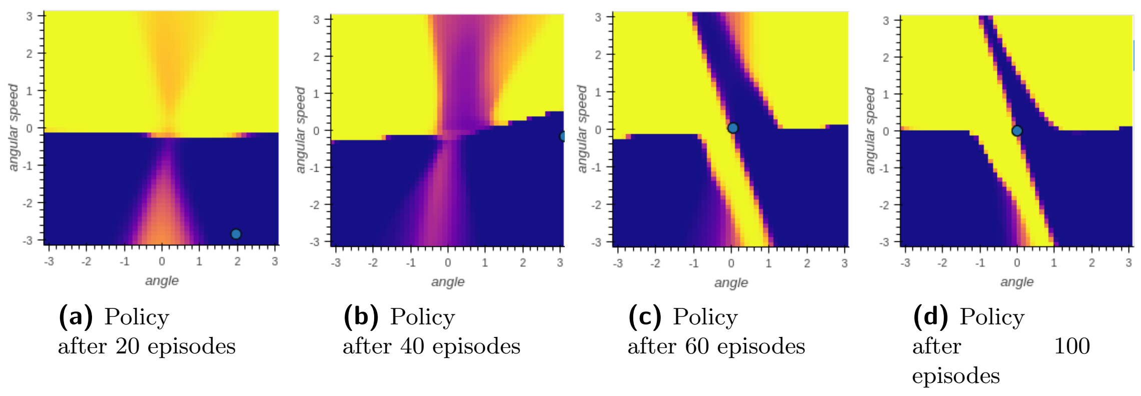
(a) Q-values for maximum positive torque after 20 episodes

(b) Q-values for maximum positive torque after 40 episodes

(c) Q-values for maximum positive torque after 60 episodes

(d) Q-values for maximum positive torque after 100 episodes

Figure 2-22: Evolution of Q-values for maximum negative torque of the DDPG algorithm applied on the underactuated pendulum problem. The skewed light band that starts to appear after 20 episodes is logical; having a positive angular rate and being located just before the top position is a good state



(a) Policy after 20 episodes

(b) Policy after 40 episodes

(c) Policy after 60 episodes

(d) Policy after 100 episodes

Figure 2-23: Evolution of the policy over episodes. Yellow colors mean clockwise torque, red values mean no torque, and blue values mean counterclockwise torque.

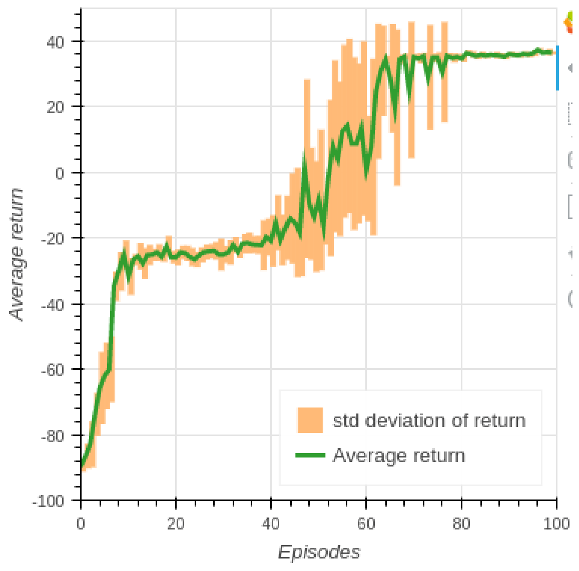


Figure 2-24: Evolution of the average return and the standard deviation of the average return from the blue start square to the goal. This value is calculating by simulating 10 episodes from the initial start position (pendulum in rest position). The return never reaches the optimal return since the stochastic policy is evaluated up until ϵ is equal to 0.1.

2-4-7 Conclusions on use of algorithms

The previous chapters showed how reinforcement learning can be used to find a policy that optimizes the return of an agent in an environment. In the maze problem three different algorithms are evaluated, MC control, SARSA and SARSA(λ). Although no accurate and fair hyperparameter optimization was used on the algorithms to compare performance, it was clear that the SARSA(λ) algorithm converged the fastest to the optimal policy. The effects of using 'bootstrapping' for updating the Q-values and using eligibilty traces to reduce the credit assignment problem is evident here. Although the maze problem is often used in evaluating reinforcement learning problems, it is one that has both discrete state and action spaces. The inverted pendulum problem therefore is a better problem to test the efficiency of algorithms that are able to model continuous input states (DQN), or continous output states, or both (MLAC,DDPG). It was shown that using deep neural networks as function approximators for modeling continuous input states converges quickly to an optimal policy, as shown in example 2.4. The MLAC algorithm is an algorithm that is able to model both continuous input states and continuous actions. It is therefore a very appealing algorithm to use for aerospace control problems, and therefore chosen as the base algorithm for flexible heuristic dynamic programming in Helmer et al. [2018]. However using shallow function approximators is a limitation on many high dimensional real world problems. The DDPG algorithm evaluated on example 2.6 shows that is an efficient way of dealing with continuous input states and action spaces. In combination with the fact that it uses deep neural networks as function approximators, makes this a good base candidate for a deep flexible heuristic dynamic programming algorithm. Further research will need to answer the questions posed in research question 2 . These answers will help in further determation whether the DDPG is a good candidate for deep flexible heuristic dynamic programming.

2-5 Curriculum Learning

Reinforcement learning can very bluntly be put as learning by trial and error. In a certain way, humans perform the same learning. But for almost all complex tasks, the task is learned by teaching. This thesis could not have been written by not having years of education in engineering topics. Learning a complex task by splitting it up in easier tasks is called "shaping" in psychology. The thesis of Helmer et al. [2018] showed that constraining a reinforcement learning problem to a simpler subset of states before training an agent on the full set of states showed promising results. This type of learning shows similarities with curriculum learning Bengio et al. [2009]. Bengio et al. [2009] states that shaping in machine learning can be achieved by forming a training sequence over examples that are being sampled with different weights from a training distribution. This importance sampling gradually increases the weights of more difficult examples towards the end of the curriculum. More formally, curriculum learning is stated as follows: A sequence of distributions Q_λ is called a curriculum if the entropy of these distributions increases, and the weight of each sample, W_λ, is monotonically increasing. This process can be seen as an optimizing process that first optimizes a smoothed loss function that becomes less and less smoothed. This is also known as a continuation method. Similar to unsupervised initialization strategies for deep neural networks, the authors suggest that curriculum learning acts as regularization. This claim is backed by showing improved generalization errors on simple toy examples as compared to non-curriculum learning tasks. One of the experiments was to train a 3-layer neural network on a shapes dataset. The shapes dataset consisted out of basic geometric shapes. These shapes can be distinguished into easy shapes and difficult shapes, and are shown in figure 2-25.

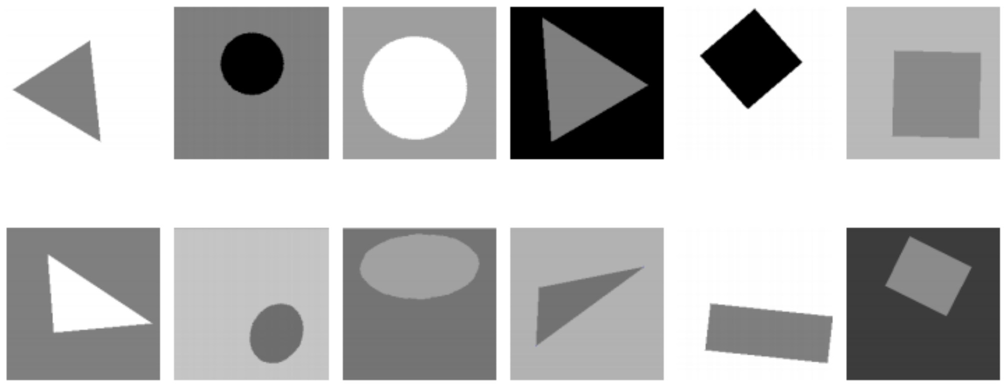


Figure 2-25: the geometry dataset that the neural network was trained on in the experiment of Bengio et al. [2009]. The neural network was trained on a curriculum of easy and difficult examples. The upper row shows the easy examples of the curriculum, and the bottom row depicts the difficult examples. Taken from Bengio et al. [2009]

.

The neural network was trained in a curricular fashion. First the easy shapes are trained and after a certain epoch, the switch epoch, the difficult examples are added. This switch epoch was varied in the experiment, and the network was tested on the test set. All other hyperparameters were determined by optimizing the network on the total network without curricular training. Figure 2-26 shows the distribution of the test error for 20 different initializations of network parameters per switch epoch.

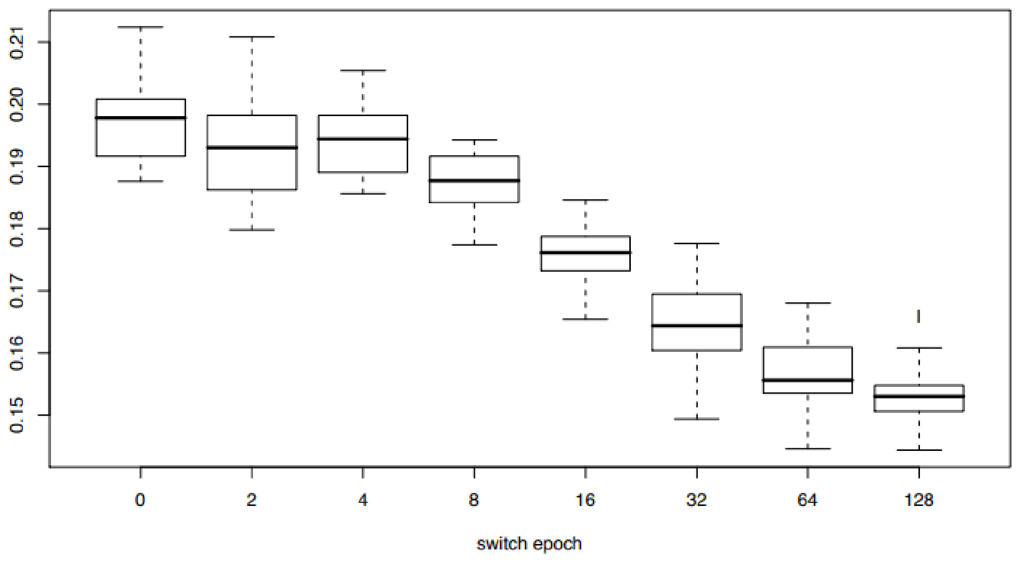


Figure 2-26: Box plot of the test classification error vs the switch epoch with a 3 hidden layer neural network trained on stochastic gradient descent. Each box corresponds to 20 different initializations of network parameters. Taken from Bengio et al. [2009]

The authors conclude with two hypotheses for the improved generalization capabilities, namely:

1. Faster training in an on-line setting. Because the learner wastes less time on noisy or harder to predict examples
2. Curriculum learning can be seen as a continuation method. The method offers as a guide to better regions in parameter space, into basins of attraction.

2-5-1 Curriculum learning in deep reinforcement learning agents

Several implementations of curriculum learning or shaping in reinforcement learning agents exist. The most prominent research will be summarized in the upcoming sections.

2-5-2 Emergence of locomotion behaviours in rich environments

One particular implementation of curriculum reinforcement learning in deep continuous reinforcement learning is the work of Heess et al. [2017]. In Heess et al. [2017] a Trust Region Policy Optimization (TRPO) scheme is used to find rich behaviours from simple reward functions. Tasks are structured to be simple in the beginning and more complex towards the end of learning. Experiments showed that rich behaviour emerged due to the curriculum approach of learning.

Advantages and disadvantages:

- + Uses curriculum learning exactly as proposed in Bengio et al. [2009]. The learning can be viewed as a curriculum where more difficult examples are gradually added in the learning process.
- - A flexible approach is not used. The structure of the actions, and input states is kept the same over the training procedure.

2-5-3 Flexible shaping in neural networks

An experiment that uses a flexible capacity of a neural network in combination with curriculum learning is that in Krueger and Dayan [2009]. In Krueger and Dayan [2009] the AX-12 working-memory experiment is used to compare a curriculum strategy to a normal learning strategy. The AX-12 working-memory experiment is an experiment where a sequence of symbols has to be classified as either L or R. If a AX occurs in a sequence and the first number preceding it is a 1, it is classified as an L, if an BY occurs and the first number preceding it is a 2, it is classified as an R. The experiment shows the need for remembering a long-term memory, namely the 1 or 2, and a short-term memory namely the AX or BY. A Long Short Term Memory (LSTM), which is a specific type of a recurrent neural network, is used as a classifier for this experiment. The curriculum is structured to first train the long term memory on the numbers, whereafter the short-term memory is trained on the symbols. Each part of the curriculum is trained by a seperate LSTM module, making this a flexible approach of learning. In the end all the different 'specialized' modules were combined and trained further to asses performance. It was shown that the flexible curriculum learning approach required about half the amount of epochs to converge to a certain criteria, as opposed to the normal training case.

Advantages and Disadvantages:

- + Training is somehow flexible. The flexibility resides in being able to add and combine modules that are trained on subtasks. However these subtasks all have the same input and output structure, so this is different then the work from Helmer et al. [2018].
- - The different modules are not able to reuse knowledge from other modules, because they are trained independent from one another.

2-5-4 Progressive neural networks

One phenomena that neural networks suffer from is catastrophic forgetting. Catastrophic forgetting is the symptom that when training neural networks on a new task, previous experience is forgotten. If the shaping procedure consists of stages that do not have any overlap in "information" or "knowledge", catastrophic forgetting will delete all previously learned knowledge in the new shaping stage. For example, one could use shaping to teach a neural network how to detect pets. The first stage of the shaping procedure could be to learn all breeds of dogs, and the second stage of the shaping procedure could be to learn all breeds of cats. Catastrophic forgetting would result in the network to lose all the gained information

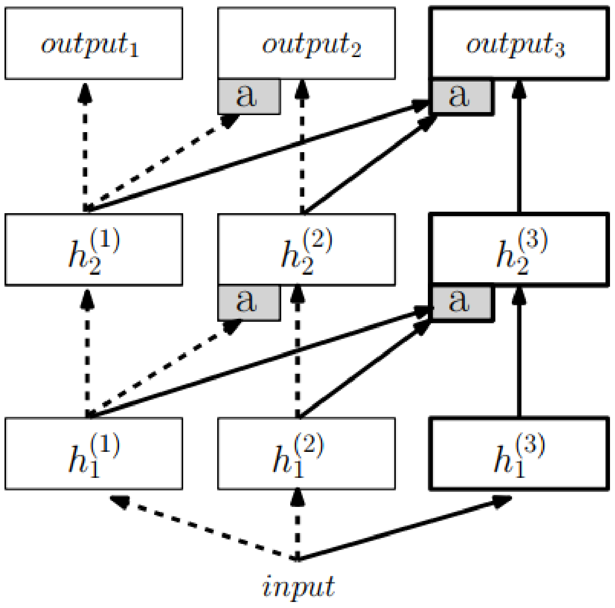


Figure 2-27: A progressive network architecture. Every block is a layer of neurons. Each column is trained on a different task. A layer in a column, depicted by a h, receives an input from the previous hidden layer of that column and that layer from all the preceding columns. This way knowledge from previous tasks can be reused, while not updating the weights of these tasks. Taken from Rusu et al. [2016]

on dogs while training on the breeds of cats. Some research has been done on preventing catastrophic forgetting in neural networks.

In Rusu et al. [2016], a method, called progressive neural networks, is devised to improve transfer learning and overcome catastrophic forgetting in deep reinforcement learning. An asynchronous version of the DQN network, called A3C is used in the experiments. The function approximator consists of a neural network that can be expanded by adding new columns. Every layer in this new column is connected to the previous layers of the old columns, this way previously gained knowledge can be reused in the added column of neurons. During training, the parameters of the previous network are frozen, this way catastrophic forgetting is circumvened. Experiments consisted of training a reinforcement learning agent on different tasks, namely the ATARI game pong, and a modified variant of pong, called pong-soup. It was shown that the progressive networks outperformed techniques that use fine-tuning for transferring knowledge.

Advantages and Disadvantages:

- + Efficiently reuses knowledge from multiple previously learned tasks.
- + Can still retain knowledge from previously learned tasks.
- - Training is not flexible, the input and output structure must be the same for the different tasks.
- - Reusing knowledge and preventing catastrophic forgetting is achieved by increasing

capacity in the network. This is not very scalable.

2-5-5 Curriculum learning in flexible heuristic dynamic programming

The work of Bengio et al. [2009] showed that curriculum learning can be a powerful way of improving generalization errors on machine learning tasks. Flexible heuristic dynamic programming can be viewed as a version of curriculum learning where the input domain is flexible. None of the research mentioned above implements a flexible way of curriculum learning, the mentioned research uses input domains that have a fixed number of dimensions over the curriculum. This is exactly the novelty of flexible heuristic dynamic programming, using a flexible input domain to structure the curriculum. Since reinforcement learning can be a costly process, curriculum learning can help reducing the amount of interaction with the environment required to obtain a decent policy.

2-6 Literature survey conclusion

flexible heuristic dynamic programming is a novice reinforcement learning technique devised by Helmer et al. [2018]. In flexible heuristic dynamic programming, the agent learns by gradually adding more states in the environment. This can be identified as a form of curriculum learning. Helmer et al. [2018] Used local linear regression as a function approximator. Consequently generalization and transfer of knowledge is very low due to the local nature of these function approximators. In recent years, reinforcement learning has seen a lot of progression in using deep neural networks as a function approximator, Mnih et al. [2013], Lillicrap [2016]. The research objective of this thesis is to *Reduce the sample complexity of reinforcement learning by learning in a flexible way with deep neural networks as global function approximators.*. The corresponding research questions for this research objective can be found in chapter 1 .

This preliminary thesis gives an answer to the first main research questions of the research objective, wich are:

1. What is the state of the art in deep reinforcement learning
2. What is the state of the art in curriculum learning

Subquestion number 1 is answered in chapter 2-4. Reinforcement learning algorithms are implemented on test examples to gain familiarity with the algorithms and to test the specific features of those algorithms. The algorithms are implemented from simple to more complex algorithms. They start at one of the simplest discrete reinforcement learning algorithms, and end up in a deep continuous state and action space algorithm. This final algorithm called DDPG, is an algorithm that uses deep neural networks as a function approximator. In addition it can be applied on a quadcopter control task, because of it's ability to deal with continuous input states and continuous action spaces. Depending on further research, it could be a good candidate to base the deep flexible heuristic dynamic programming algorithm on. In chapter 2-5 subquestion 2 is answered. It first gives a summary of the findings by Bengio [2009]. It formalizes curriculum learning, and gives hypotheses on why curriculum learning can result in better performance in machine learning algorithms. The chapter continuous by evaluating papers that have found novice architectures that implement curriculum or transfer learning. However no papers have been identified that implement curriculum learning in a flexible way. This chapter can serve as an overview of methods that implement transfer or curriculum learning in neural networks.

Chapter 3

Final research approach

Now that the state-of-the-art in deep reinforcement learning and curriculum learning has been evaluated, a plan for the final research can be made. In this chapter the final steps that have to be taken in order to reach the research objective are given.

The remaining research questions are the following:

1. What is flexible reinforcement learning.
2. How can DDPG be modified to suit flexible reinforcement learning.
3. What is the performance of flexible DDPG on different environments.

The first research question is a very important one, and one of the central questions of this research. Before coming up with an algorithm for deep flexible reinforcement learning, flexible reinforcement learning itself first needs to be formalized, and an analysis needs to be made on the potential benefits and caveats of flexible reinforcement learning. Because a lot of different MPD environments exist, flexible reinforcement learning will likely behave very differently per environment. Environments that are likely to show a hypothesised benefit or caveat of flexible reinforcement learning will be identified, and possibly used for the final evaluation. In addition, different environments will be shown to get a clear understanding of flexible reinforcement learning.

When flexible reinforcement learning has been formalized, an algorithm for deep flexible reinforcement learning can be constructed. This algorithm will be based on the DDPG algorithm, so that it uses deep function approximators for the critic and the actor. First a method for flexible learning in deep neural networks needs to be devised. When this method is developed, an analysis needs to be made on how the flexible neural network effects the different parts of the DDPG algorithm. Finally the final flexible DDPG algorithm can be constructed.

When the previous steps are completed, a study can be done on how the flexible DDPG algorithm performs on different environments. Experiments need to be done on different

MDP environments, ideally each experiment would be used to validate one or more of the earlier mentioned hypotheses on the performance of flexible reinforcement learning.

Part III

Additional results

Chapter 4

In-depth policy analysis of the TwoCartpoles environment

In this chapter, the policies found by the normal DDPG algorithm, flexible DDPG algorithm, and the decentralized DDPG algorithm on the TwoCartpoles environment are analyzed in-depth. State and action plots of an episode after both stages of the curriculum are given for all variants of the DDPG algorithm.

4-1 State plot of flexible DDPG, normal DDPG, and decentralized DDPG on the TwoCartpoles environment

In figure 4-1, 4-2, 4-3 and 4-4 the state-plot and action-plot for an evaluation of the policy after the first and final stage of the flexible DDPG agent on the TwoCartpoles environment are shown, respectively. It can be seen that although a good policy is learned for the second cartpole, the policy is not optimal, as the lever has lost a balance after 400 seconds. In addition, the policy for the first cartpole is altered and some performance is lost, as it can be seen that the balance is lost after 700 seconds. However, in both cases the balance is quickly regained after it is lost. A comparable state and action plot for the non-flexible DDPG algorithm can be found in 4-5 and 4-6. This sub-optimal performance of the flexible and non-flexible agents is likely the result of two causes:

1. **Neurons are shared in the centralized critic that is used for mapping the Q-values of two independent MDP processes.**

   Because the centralized critic needs to find a mapping from the input states of both cartpoles to one output value, it is difficult for the critic to find a mapping that has action gradients that are invariant to the input states of the opposing cartpole. More mathematically speaking, the mapping from input states to Q-value of the total MDP is a superposition of the Q-values of the individual processes because no coordination is required. that is:

$$Q(S_{c1}, A_{c1}, S_{c2}, A_{c2}) = f(S_{c1}, A_{c1}, S_{c2}, A_{c2}) = f(S_{c1}, A_{c1}) + f(S_{c2}, A_{c2}) \tag{4-1}$$

Where subscripts $_{c1}$ correspond to the states and actions belonging to the MDP for the first cartpole, and subscripts $_{c2}$ belong to the states and actions for the MDP of the second cartpole. Because of this, the action gradient for the first cartpole is only a function of the states of the first cartpole, and the action gradients for the second cartpole is only a function of the second cartpole.

$$\frac{\delta Q(S_{c1}, A_{c1})}{\delta A_{c1})} = f(S_{c1}, A_{c1}) \tag{4-2}$$

and

$$\frac{\delta Q(S_{c2}, A_{c2})}{\delta A_{c2})} = f(S_{c2}, A_{c2}) \tag{4-3}$$

However, because a structure for the critic is required that uses sharing of neurons (to capture potential coordination), it is difficult for the critic to make a mapping where the action gradients for the actions of both cartpoles are independent of the states of the opposing cartpole. This results in gradient updates to the actor that have a dependency on input states for the opposing cartpoles. So in short, it is difficult for the critic to 'realize' that both processes are independent processes, and thus an unnecessary link is made between input states that have no influence on one of the two processes. This might be an explanation why the final policy in the central DDPG agent, and the flexible DDPG agent results in rather stochastic actions, and a sub-optimal policy.

2. **The reward signal received by the centralized Critic is unidentifiable.**

   Because the aggregated reward signal is the only signal available to the agent, it is difficult to make a mapping for the agent that determines which actions lead to which portion of the reward. This means that in the particular case of the TwoCartpoles environment, when one cartpole receives a reward, the actions of the other cartpole are reinforced as a result of the actions of the first cartpole that was responsible for the reward.

In figure 4-7 and 4-8 the state and action plots for the decentralized agent are shown. It can clearly be seen that both cartpoles are able to balance, and that the balance is not lost for the remainder of the episode. In addition, the actions that are used to balance the carts are much more constant. Both of the decentralized agents only make a mapping from input states to values and actions for the states and actions that belong to their specific MDP processes. Hence, individual mappings from input states to actions for the individual cartpoles can be made, and the final policy is much smoother.

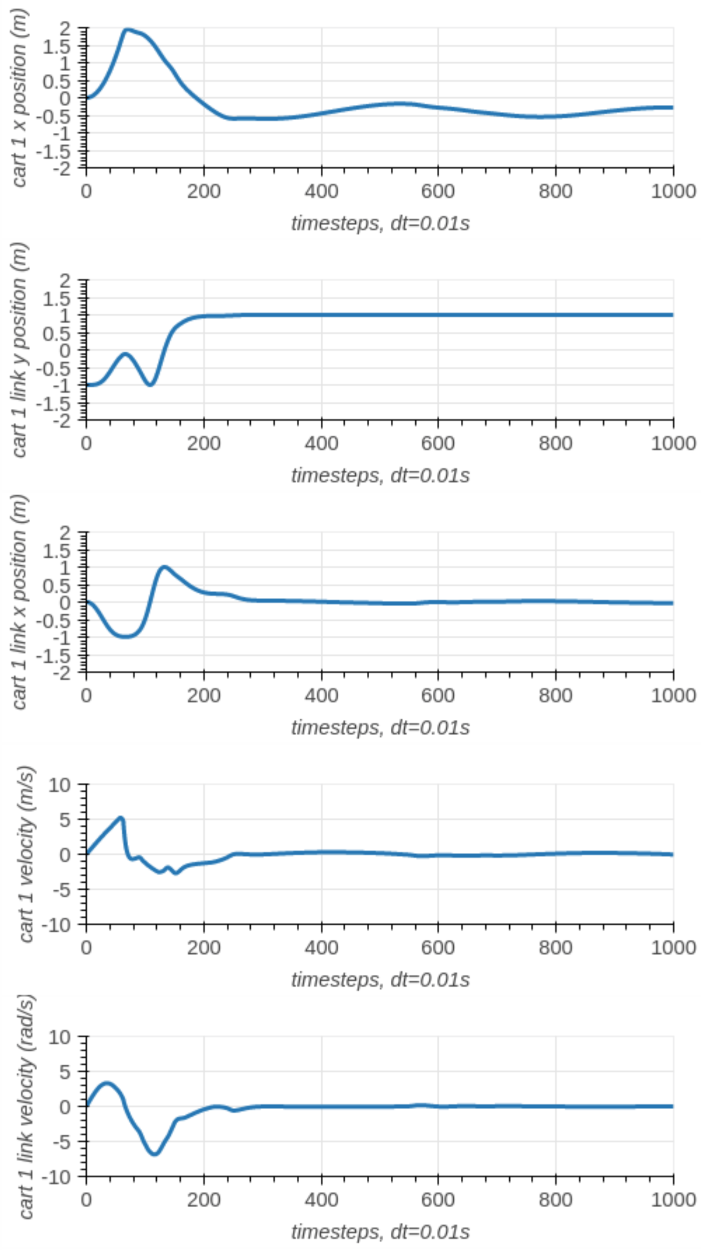


Figure 4-1: Analysis of the states of the policy of the flexible DDPG algorithm after the first stage of the curriculum on the TwoCartpoles environment. It can be seen that the agent is able to balance the lever at vertical position after about 200 timesteps.

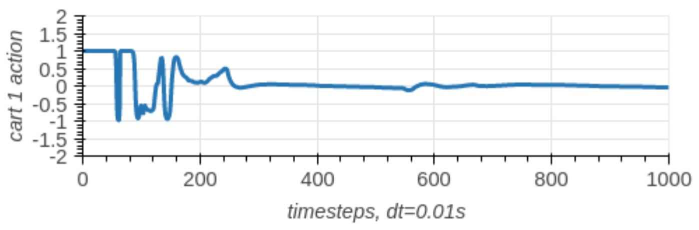


Figure 4-2: Analysis of the actions of the policy of the flexible DDPG algorithm after the first stage of the curriculum on the TwoCartpoles environment. It can be seen that once the agent is able to balance the cartpole, only small actions are required.

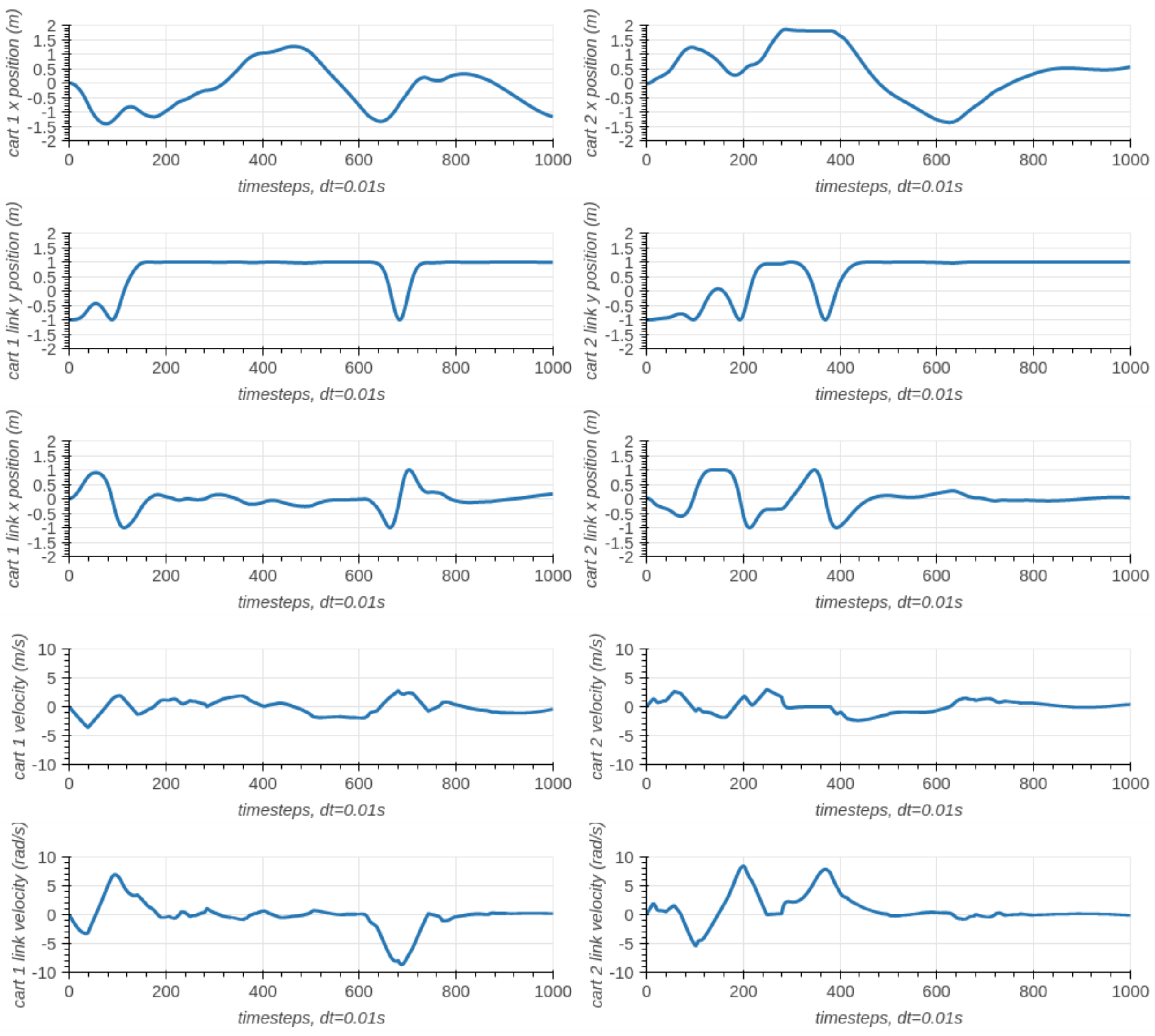


Figure 4-3: Analysis of the states of the policy of the flexible DDPG algorithm after the final stage of the curriculum on the TwoCartpoles environment. It is interesting to see that the agent has lost a bit of the original policy for the first cartpole, since it can be seen that it unable to hold the lever upwards after about 700 timesteps. This is likely due to inference from the states of the second cartpole. The policy is unnecessary trying to use a form of coordination. This is caused by sharing neurons in the critic. Because of neuron sharing in the critic, it is difficult for the critic to result in an action gradient that is invariant to changes in the input states for the second cartpole. The result is that connections from input states of the second cartpole are being reinforced towards actions for the first cartpole.

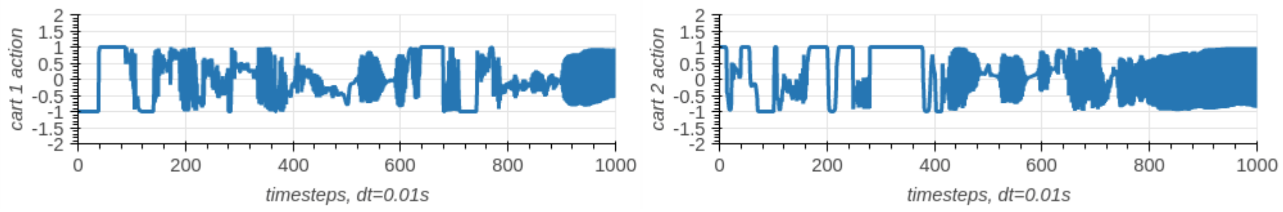


Figure 4-4: Analysis of the actions of the policy of the flexible DDPG algorithm after the final stage of the curriculum on the TwoCartpoles environment. It can be seen that the policy for the first cartpole is much less smooth then the policy after the initial stage. This can again be explained by sharing of neurons in the critic. The actions for cartpole 1 are influenced by states for cartpole 2 and vice versa.

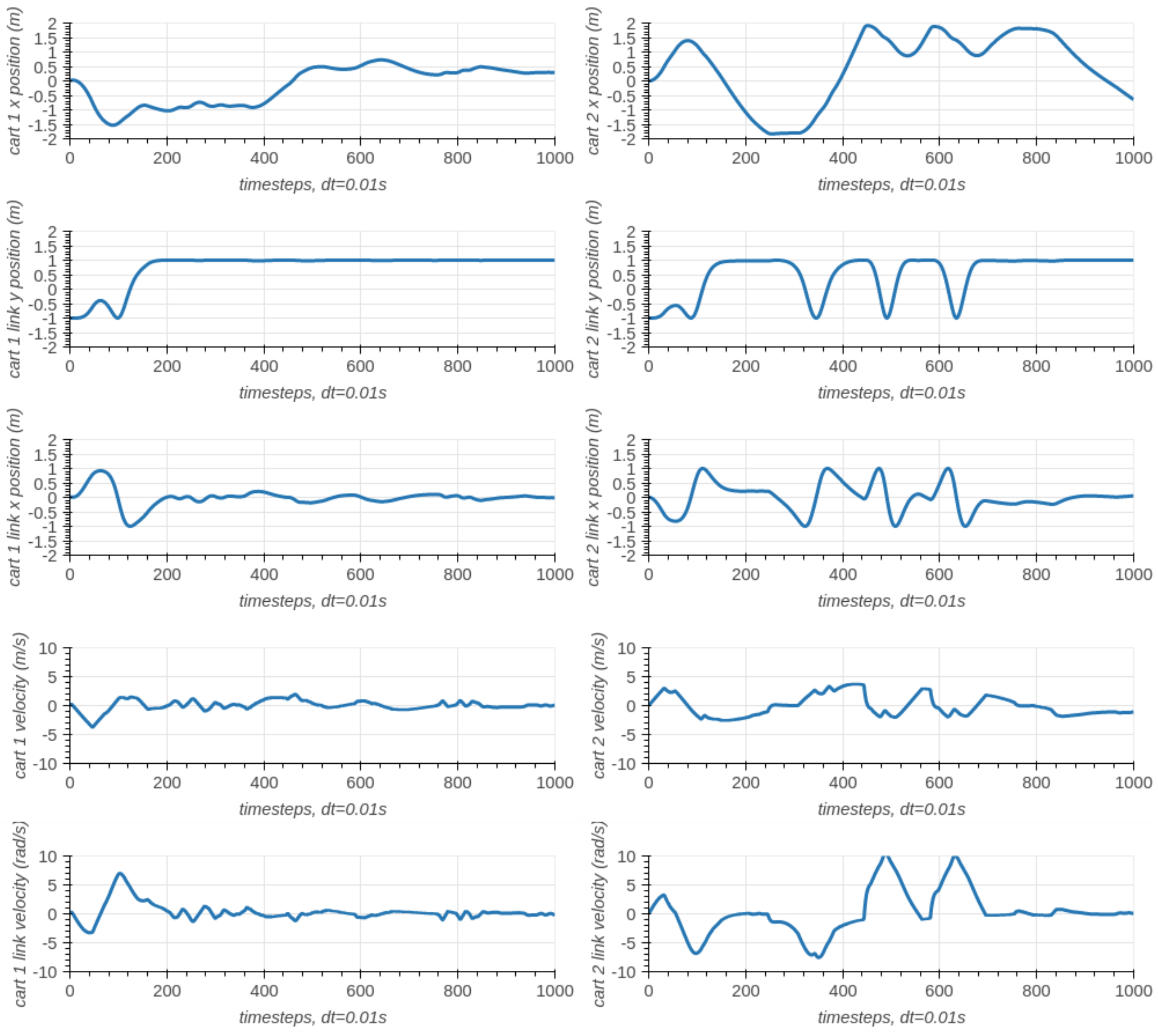


Figure 4-5: Analysis of the states of the final policy of the normal DDPG algorithm on the TwoCartpoles environment. Again it can be seen that sharing of neurons in the critic is having a negative effect on the policies of both cartpoles.

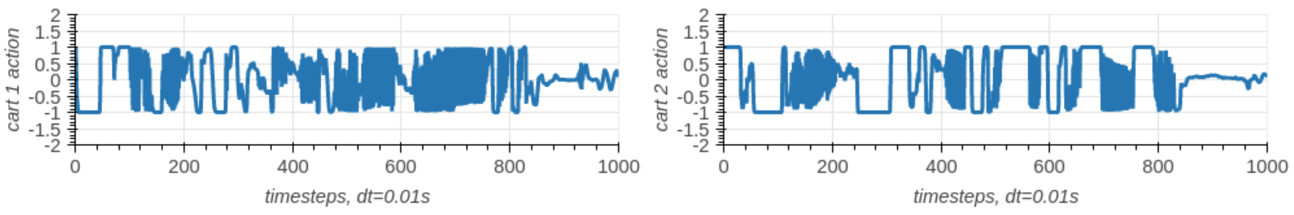


Figure 4-6: Analysis of the actions of the final policy of the normal DDPG algorithm on the TwoCartpoles environment. The policies for both actions are not smooth.

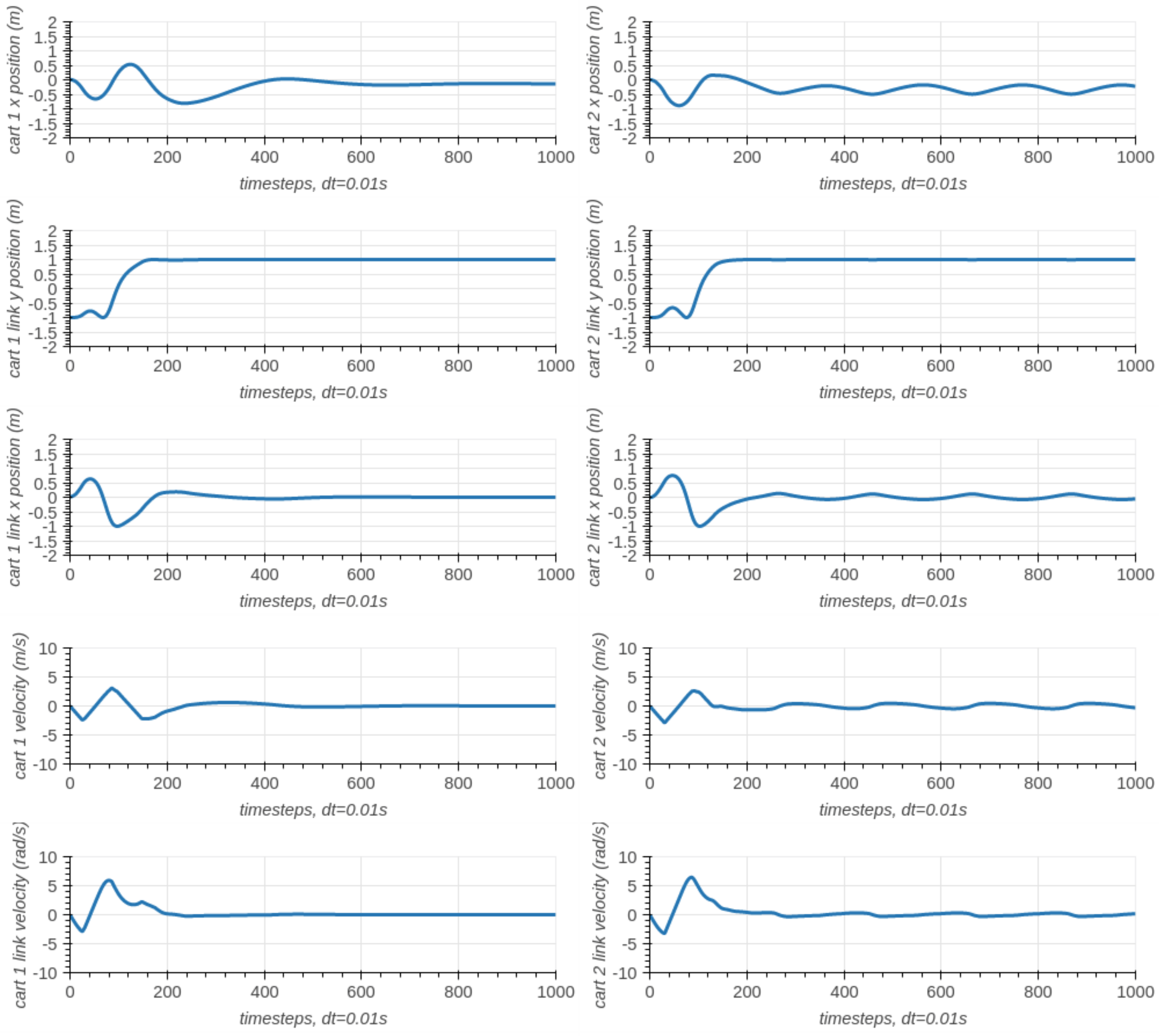


Figure 4-7: Analysis of the states of the final policy of the decentralized DDPG algorithm on the TwoCartpoles environment. It can be seen that balancing of both cartpoles is easily achieved by the decentralized agents. This is because each agent has it's own parameters, and rewards are fully identifyable.

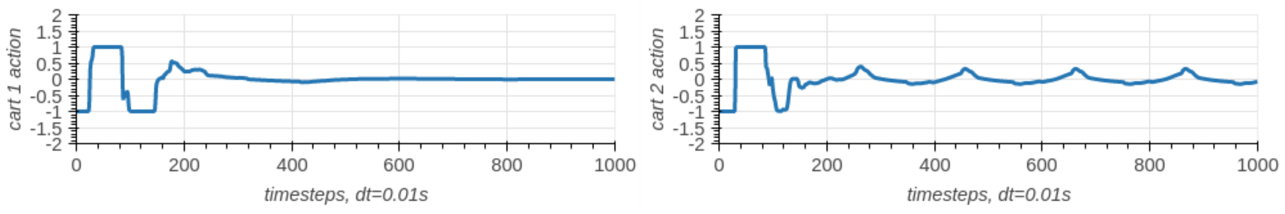


Figure 4-8: Analysis of the actions of the final policy of the decentralized DDPG algorithm on the TwoCartpoles environment. It can be seen that the actions are very smooth.

Chapter 5

Flexibility of the Critic

In a new stage of the flexible DDPG algorithm, new states and actions are added to the Critic and Actor networks. The Actor must be able to adapt it's policy based on these new states. Analogous to the Actor, the Critic must also be able to adapt it's action-values based on the new states and actions. It might be difficult for the Critic to adjust it's parameters to incorporate a dependence on the new states and actions, while not disturbing the previously learned policy and action-value function too much. Optimizing the Q-values or the policy based on additional states and actions might be difficult since all parameters have moved towards a parameter space that fit the Q-values or policy for the initial stage of the curriculum. If a single column design for the neural networks is chosen, hidden layers have to be shared between mappings of the different MDP's that constitute the curriculum. This results in mappings from both MDP's towards a single Q-value that has some inherent form of coordination, even when no coordination is required. In figure 5-1 the final policy for the flexible DDPG algorithm on the TwoCartpoles environment is shown. It can be seen that it has learned an unnecessary form of coordination due to sharing of neurons, because the policy for the second cartpole has found an unnecessary dependency on the input states for the first cartpole.

This chapter is devoted to give an insight on how big the effect of sharing of neurons in the Actor and Critic networks is, by making a comparison with a network architectures that has no sharing of neurons.

Two different architecture designs for both the Critic will be evaluated. One architecture has inherent sharing of neurons across inputs, and the other architecture is designed such that each new set of inputs has a new dedicated set of parameters. This last architecture is called a 'multi-column' design, because each set of inputs has a dedicated column of hidden layers.

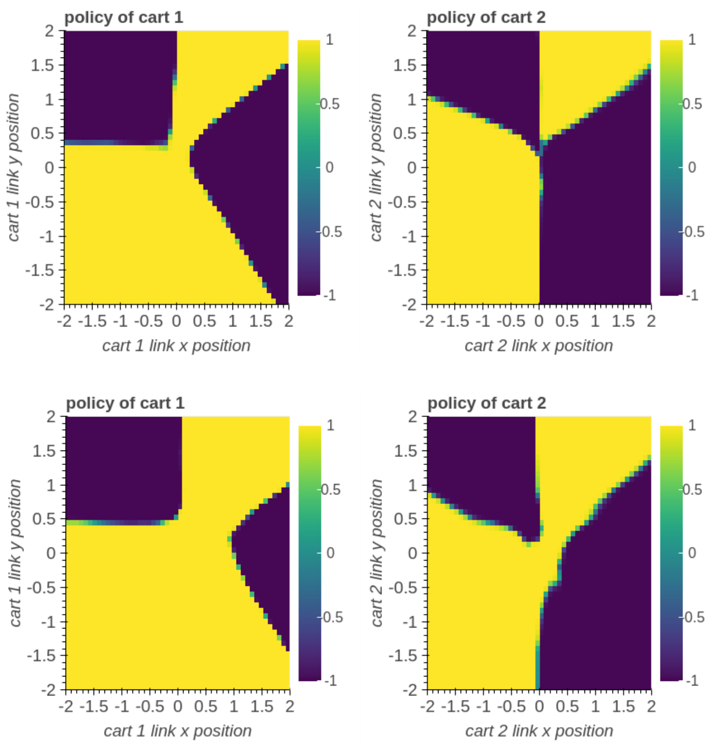


Figure 5-1: Actions of cartpole 1 and cartpole 2 versus the x position of the link and y position of the link for each cart, evaluated with the remaining input states at zero. The bottom row depict the policy when the x position of cart1 is changed to -1. It can be seen that the policy for cart2 also changes, this is undesired form of coordination is likely caused by using using a single-column critic to map the Q-values of an aggregated reward signal.

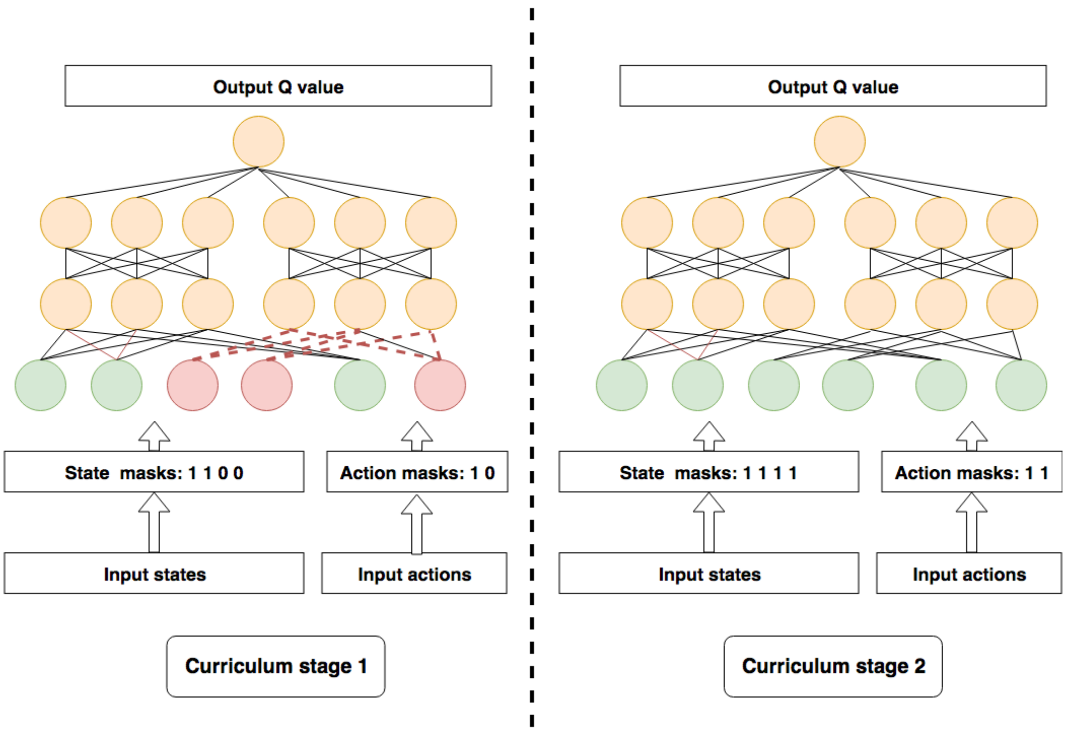


Figure 5-2: A schematic representation of the multi-column Critic network. In the multi-column Critic network, parameters are not shared between the different inputs defined by the stages in the curriculum. This can only be applied in environments where the different stages of the curriculum have no interaction with each other. Red neurons are neurons that are masked during that stage in the curriculum.

5-1 Comparison with a multi-column critic architecture

A multi-column architecture as is used in the actor in flexible DDPG is impossible to use in the Critic, because the Critic only outputs one action-value. In addition, the Critic needs to combine information from all the inputs into one column of hidden layers, otherwise capturing interaction between inputs would not be possible. However a multi-column approach can still be mimicked, by combining the outputs of all the columns in a final output value. It must then be ensured that inputs belonging to different stages of the curriculum are exclusively connected to one column of the network, otherwise all the parameters of the network are still influenced by a stage in a curriculum. In figure 5-2 the architecture for the multi-column Critic can be seen.

It must be noted that this type of architecture can only be used in environments where there is no coordination required between the inputs that are defined by the stages in the curriculum. This is because each final layer of all the columns are summed in the final output, hence the network would not be able to capture any interactions between inputs of different columns. Therefore using a Critic architecture like this is of not much use the flexible DDPG algorithm, because it would then have the same disadvantage as decentralized reinforcement learning, namely that it is not able to capture coordination. However it is used in this experiment as a test to see how much better the algorithm performs if each new set of inputs has access to it's

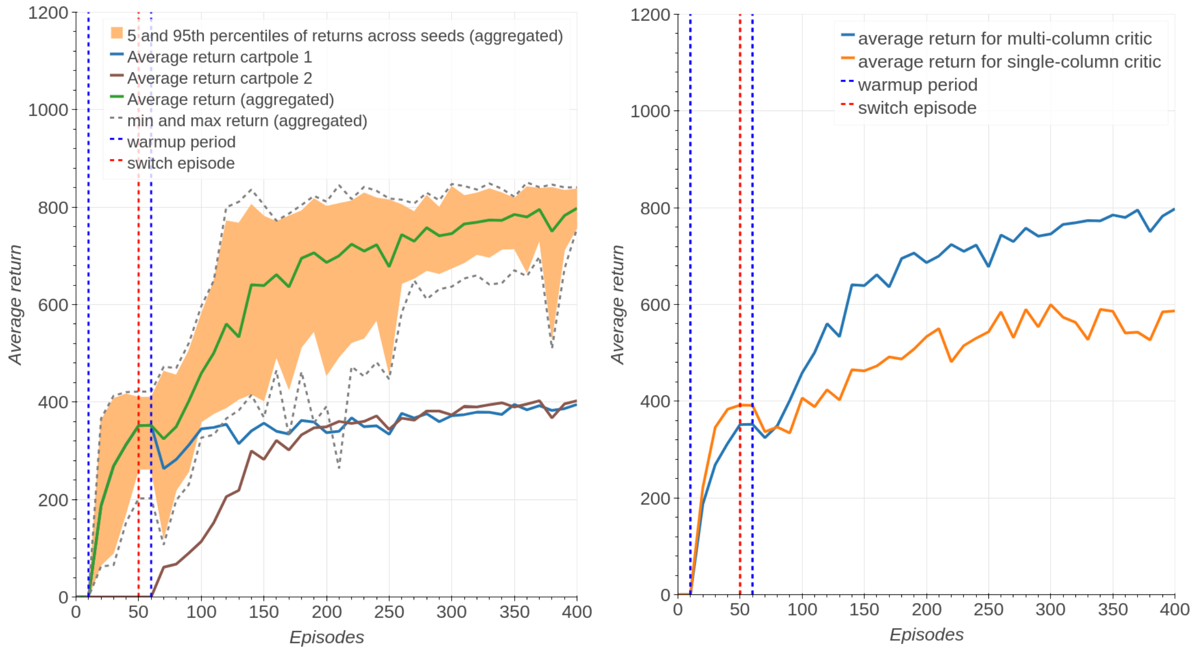


Figure 5-3: Learning curve for the flexible DDPG algorithm with a multi-column Critic architecture (left), and a comparison with a single-column architecture. In all experiments the actor architecture was also multi-column. It can be seen that the multi-column critic architecture converges quicker to an optimal policy, because each stage (and thus cartpole) in the curriculum has access to it's own parameters. The only difference with a decentralized approach is that the Critic still outputs one Q-value, because it only has access to the aggregated reward signal.

own parameters. It is important to note that this strategy is not the same as a decentralized approach, since the reward signal is still unidentifiable. The Critic only has access to the aggregated reward signal, and not the individual reward signals. This makes the reward signal unidentifiable, because it is unclear what portion of the reward can be assigned to which MDP.

The multi-column Critic will be compared with the single column Critic architecture that is used in the flexible DDPG architecture. The Actor network in the comparison is also multi-column. For a fair comparison, the amount of neurons used in the single-column architecture is put at 200 per hidden layer, while the amount of neurons in the multi-column architecture is put at 100 per hidden layer per column. The learning curve of the multi-column Critic and a comparison with the single-column Critic on the TwoCartpoles environment can be found in figure 5-3.

From 5-3 it can be seen that learning new behaviour while maintaining old behaviour is much easier if each stage of a curriculum is connected to a network that has it's own parameters. It is interesting to see that the cartpole in the second stage is not learning as fast as the cartpole in the first stage. This could be do to the unidentifiability of the reward signal. In figure 5-4 the policy after the final episode of the flexible DDPG algorithm with a multi-column Critic architecture is shown. It can be seen that the policy for the second cartpole remains much more constant under variation of input states that are relevant for the first cartpole.

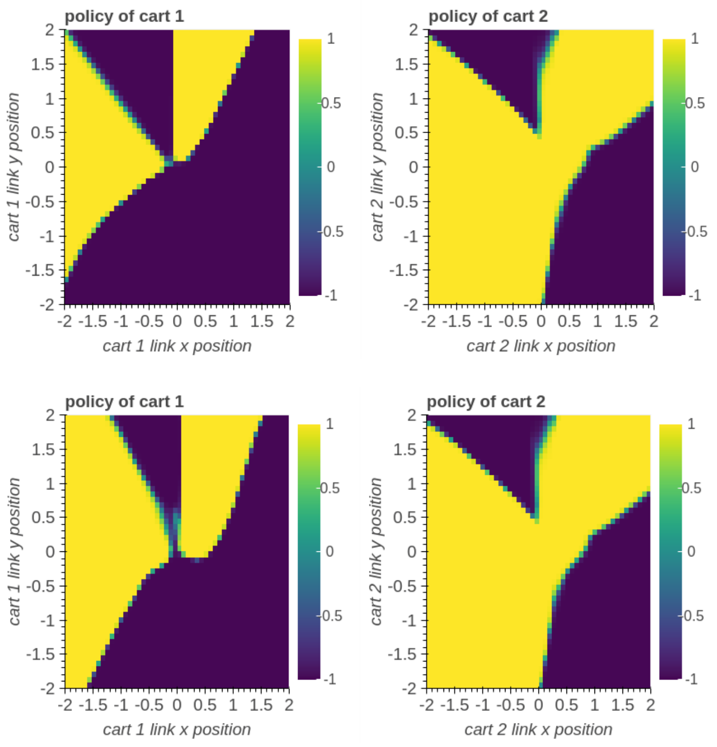


Figure 5-4: Final policy for the flexible DDPG algorithm with a multi-column architecture for the Critic network. Actions of cartpole 1 and cartpole 2 are plotted versus the x position of the link and y position of the link for each cart, evaluated with the remaining input states at zero. The bottom row depict the policy when the x position of cart1 is changed to -1. It can be seen that the policy for cart2 remains rather constant. This is desired since the policy for the second cartpole should have no dependency on input states for the first cartpole.

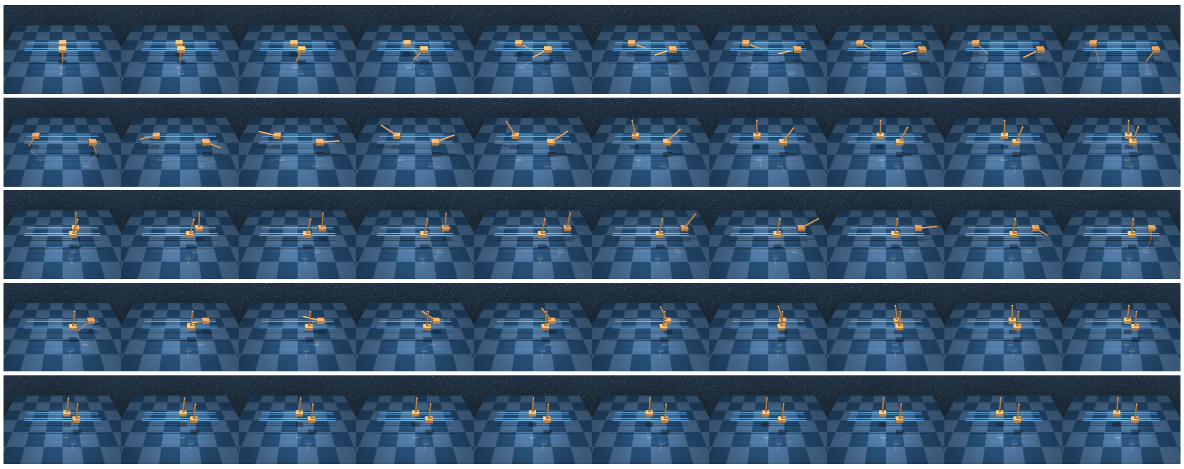


Figure 5-5: Policy visualization for the DDPG algorithm with a multi-column critic architecture on the TwoCartpoles task after the final stage in the curriculum (after 500 episodes) (snapshots taken every 10 timesteps, starting from the first timestep)

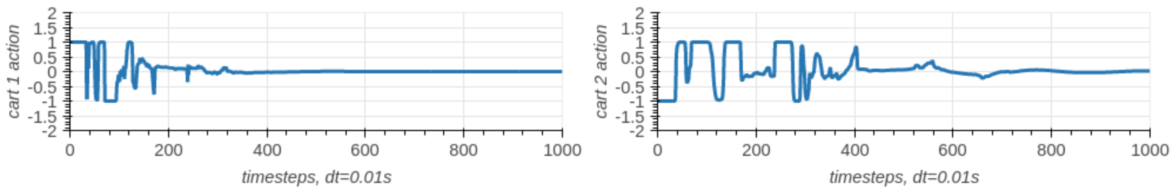


Figure 5-6: Action plot for the final policy of the flexible DDPG algorithm with the multi-column Critic architecture. It can be seen that the actions are much more smooth compared to a single-column critic architecture. This is because neurons do not need to be shared between mappings towards the Q-values of the individual cartpole MDP processes.

In figure 5-5 the final policy for the TwoCartpoles environment of the flexible DDPG agent with a splitted Critic architecture is shown.

A potential architecture where each stage could have a unique column of hidden layers, while still offering the ability to capture coordination, would be the architecture used in Progressive Neural Networks [Rusu et al., 2016]. In [Rusu et al., 2016] transfer learning and catastrophic forgetting in neural networks is circumvented by using an architecture that is able to reuse knowledge from previously learned tasks. The authors progressively train a network on different tasks, where knowledge learned in previous tasks can be reused in new tasks. This is done by adding a new column of hidden layers to each new task that is being trained, each new column has connections to the hidden layers from the previous tasks.

Each stage in a curriculum would be able to use a new column of neurons that are connected with columns that are trained on previous tasks. The progressive network is shown in 2-27. The progressive network architecture would be able to capture coordination by the connections between the different columns of the network. Ideally these connections would be initialized at zero such that by default there is no interaction between different stages of the curriculum. The amount of coordination will then be learned by the Critic.

Chapter 6

Performance on a weakly observable environment

In this chapter the effect of creating a curriculum that results in a weakly observable environment is shown. It is fairly trivial that a weakly observable environment is difficult to solve with reinforcement learning. However for completeness it is shown here.

6-1 Weakly observable Cartpole environment

In the Cartpole environment, the goal of the agent is to swing a lever to the upward position by moving a cart to the left or right along a shaft. The Cartpole environment is directly taken from the Deepmind Control Suite Library. The states and curriculum that is constructed for the Cartpole environment can be found in table 6-1

The learning curve for the flexible DDPG algorithm on the Cartpole environment with the curriculum of table 6-1 can be found in figure 6-1. Example policies found after the first stage and final stage of the curriculum can be found in figure 6-2 and 6-3 respectively.

It is fairly trivial that the agent is not able to find a good policy for the cartpole in the initial stage of the curriculum since it has no information about the velocity of the cart and the cartpole. This results in a big ambiguity in the states. No distinction can be made between being in a certain position with a velocity of zero and a velocity of one. However this information is crucial for reaching the optimal policy, and hence the agent is not able to learn a good policy in the initial stage.

In this environment, the problem of the Markov process being weakly observable is fairly trivial to see. However, there might be environments where a curriculum can be constructed by adding states where every stage is not fully weakly observable. The general idea then is that every additional state adds a bit of information of the environment, making the MDP of that stage more observable. An example of this could be an environment where a differential

Table 6-1: Curriculum used in the Pendulum experiment.

	cart x position (m)	*cart link x position (m)*	*cart link y position (m)*	*cart link angular velocity (rad/s)*	*cart link velocity (rad/s)*	*action*
stage 0	1	1	1	0	0	1
stage 1	1	1	1	1	1	1

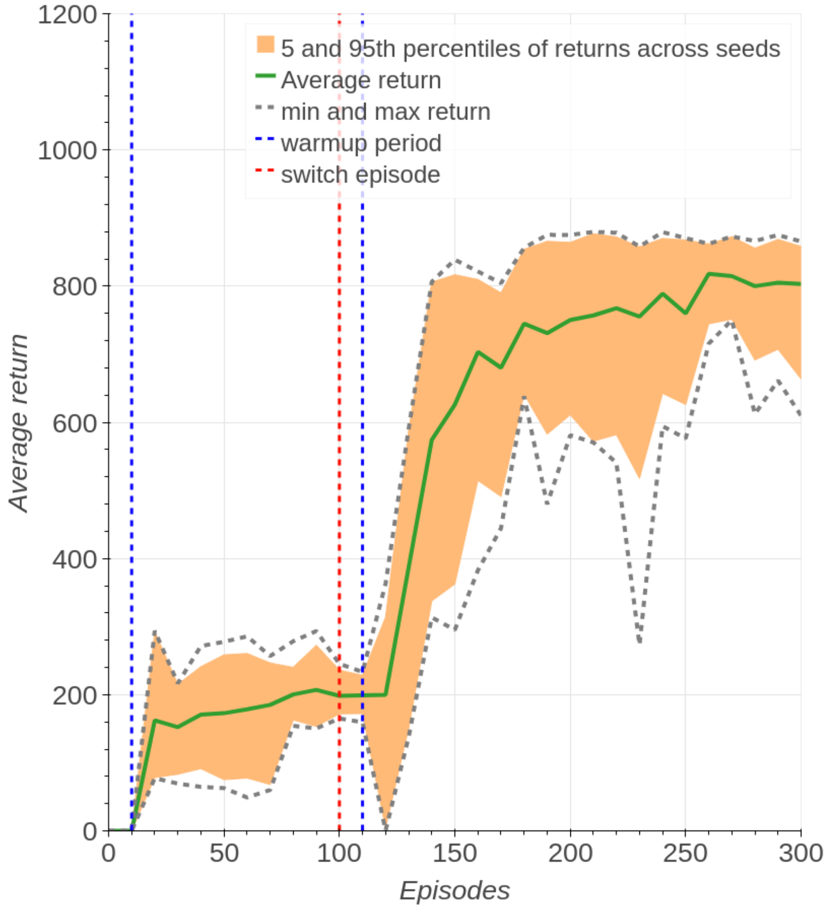


Figure 6-1: Learning curve of flexible DDPG on a curriculum that is weakly observable in the first stage of the curriculum. It can clearly be seen that learning only commences once the second stage of the curriculum is entered.

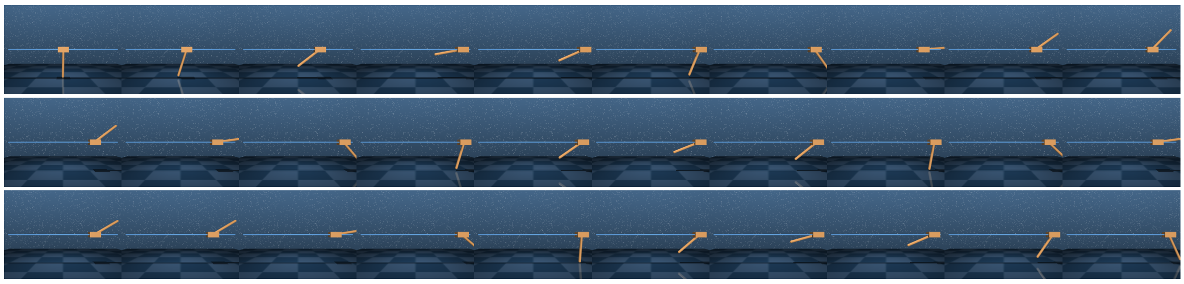


Figure 6-2: Policy visualization for the DDPG algorithm on the Cartpole task after the first stage in the curriculum (after 100 episodes) (snapshots taken every 10 timesteps, starting from the first timestep)

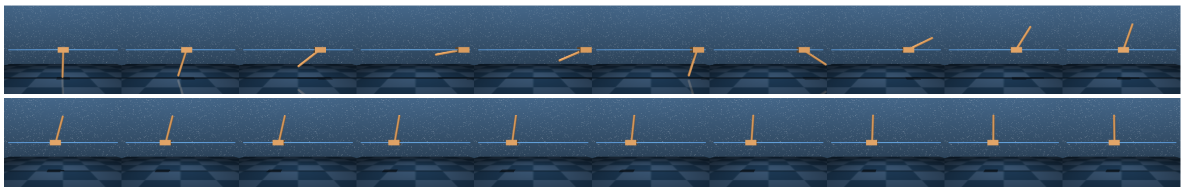


Figure 6-3: Policy visualization for the DDPG algorithm on the Cartpole task after the final stage in the curriculum (after 300 episodes) (snapshots taken every 10 timesteps, starting from the first timestep)

drive robot is equipped with a Light Detection And Ranging of Laser Imaging Detection And Ranging (LIDAR) that is facing to the front. The goal of the robot could be to navigate towards a goal in an environment with obstacles. In the initial stage of the curriculum, a policy can be found using only the front facing LIDAR. In a new stage of the curriculum, a LIDAR that is facing towards the back can be added. This could add more valuable information about the environment, and a flexible way of learning might aid the optimization process.

Chapter 7

Performance on the TwoCartpoles environment with sparse rewards

Previously, all experiments with flexible DDPG were done on environments with continuous rewards. In this chapter, the performance of the flexible DDPG algorithm on an environment with sparse rewards is evaluated.

7-1 Difficulties of sparse reward environments

Sparse reward environments do not have any form of reward shaping. In this sense sparse reward environments are closer to the semi-supervised learning paradigm that reinforcement learning claims to be, because the environment only holds information about what is the goal of the task. In environments that have continuous rewards, rewards are shaped, and additional information is inserted about how to reach the goal state. For example in the TwoCartpoles environment, a goal is to swing both levers to the upward position, and bring both cartpoles to the center position. A sparse reward can be constructed that is one when this state is reached. Now the reward function is only holding information on which state is the goal state, and no information about how to get there. This ensures freedom in finding a strategy to reach this goal state. However it might take a long time for the agent to explore the environment, and to discover the goal state, hence learning is slow. In a continuous reward TwoCartpoles environment however, the reward slowly grows from zero to one based on how far off both cartpoles are from the center position with the lever pointing upwards. In the continuous reward environment, a reward is always received, and the agent does not need a lot of exploration to reach the optimal policy. However because the agent is optimizing on maximization of the reward signal, it might not be gauranteed that the ideal behaviour (staying at the upwards position for as long as possible) is found.

7-2 Sparse reward structure in the TwoCartpoles environment

A variant of the TwoCartpoles environment is constructed that has a sparse reward. The sparse reward of the single cartpole environment is based on the position of the lever, and the position of the cart, and can be seen in equation 7-1.

$$R_{cartpole}(x_{cart}, y_{lever}) = \begin{cases} 1, & \text{if } -0.25 \leq x_{cart} \leq 0.25 \wedge 0.995 \leq y_{lever} \leq 1. \\ 0, & \text{otherwise} \end{cases} \tag{7-1}$$

For the TwoCartpoles environment, the reward is simply a superposition of the reward of the single cartpoles multiplied by 0.5.

$$R_{TwoCartpoles}(x_{cart1}, y_{lever1}, x_{cart2}, y_{lever2}) = 0.5{*}R_{cartpole1}(x_{cart1}, y_{lever1}) + 0.5{*}R_{cartpole2}(x_{cart2}, y_{lever2}) \tag{7-2}$$

7-3 Results on the sparse TwoCartpoles environment

In figure 7-1 the learning curve of the flexible DDPG algorithm on the sparse TwoCartpoles environment can be found. In addition, the multi-column critic architecture as described in chapter 5 is also evaluated on the sparse TwoCartpoles environment, to eliminate possible problems caused by sharing of neurons in the critic network.

In figure 7-2 and 7-3 the policy visualization after 200 episodes, and the final policy for the flexible DDPG algorithm on the sparse TwoCartpoles environment can be found.

It can be seen that during the second stage of the curriculum, both the single-column and multi-column architecture cannot find a good policy for the second cartpole. It is difficult to determine why this is happening.

A possible explanation could be that problems arise due to the unidentifiability of the reward signal. During the first stage of the curriculum, the reward signal can be fully assigned to actions and states of the first cartpole system. However during the second stage of the curriculum, the reward signal becomes unidentifiable. Now the first cartpole is receiving more rewards then the second cartpole, since the behavioral policy is correlated with the optimal policy learned for the first cartpole. In combination with the fact that the policy for the first cartpole is already optimized, and because exploration only rarely encounters a reward for the second cartpole, it is likely that actions for the second cartpole are often reinforced by rewards generated by the first cartpole. Hence the second cartpole is not optimizing towards a good policy.

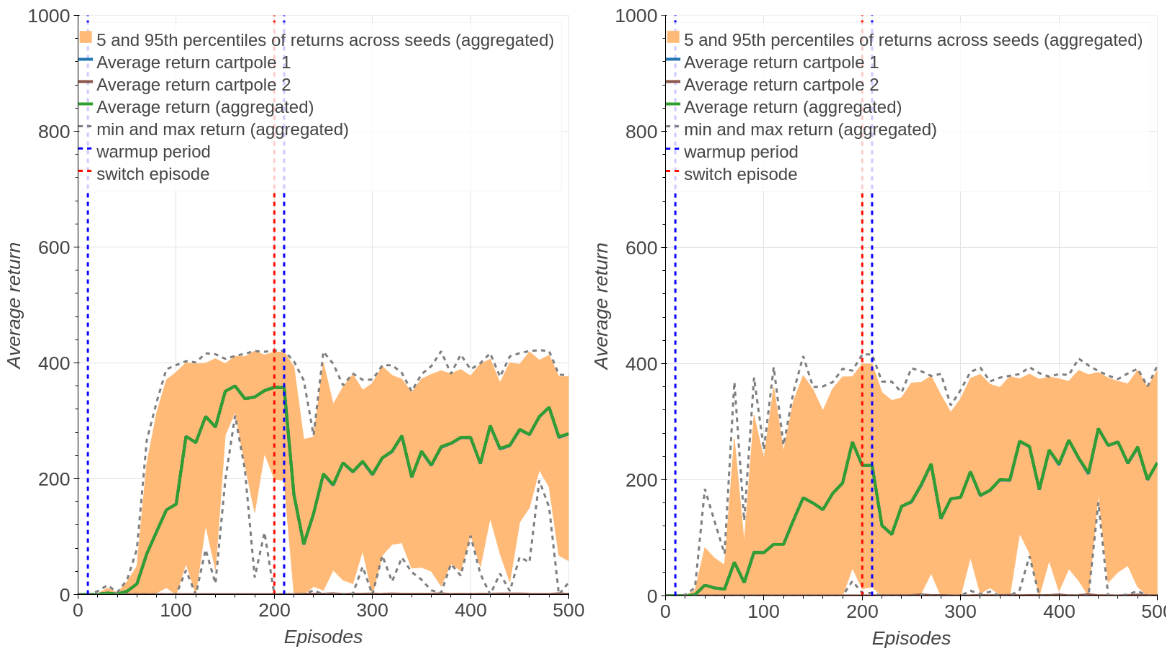


Figure 7-1: Learning curve for the flexible DDPG algorithm on the TwoCartpoles environment with the normal critic architecture (left) and a multi-column critic architecture (right) as described in chapter 5. The aggregated average return is almost entirely made up by cartpole 1, hence the aggregated return and return from cartpole 1 overlap. It can be seen that a good policy for the second cartpole is not learned

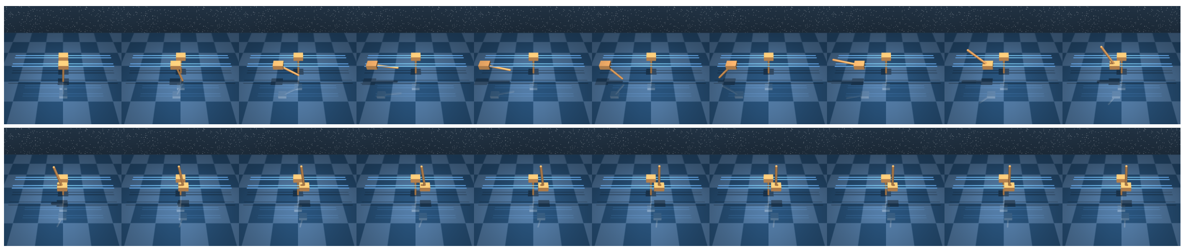


Figure 7-2: Policy visualization for the DDPG algorithm with a single-column critic architecture on the TwoCartpoles task after the first stage in the curriculum (after 200 episodes) (snapshots taken every 10 timesteps, starting from the first timestep)

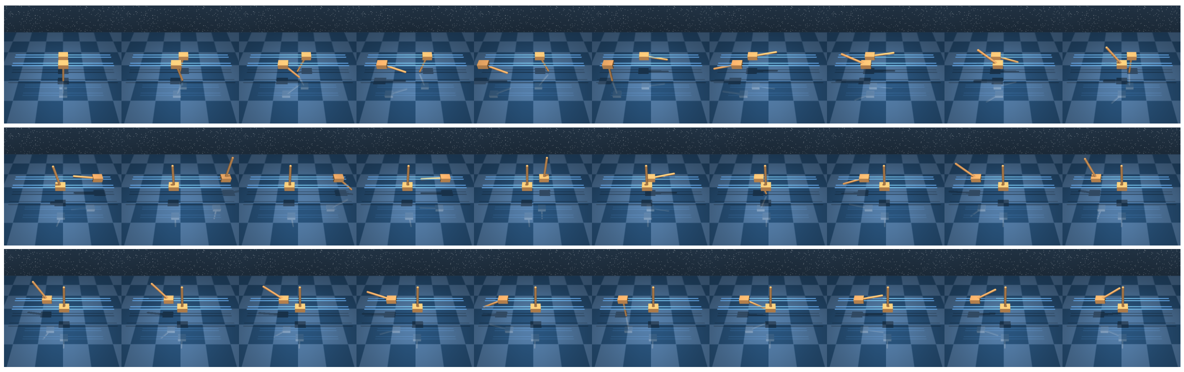


Figure 7-3: Policy visualization for the DDPG algorithm with a single-column critic architecture on the TwoCartpoles task after the first stage in the curriculum (after 500 episodes) (snapshots taken every 10 timesteps, starting from the first timestep)

Chapter 8

Parallel implementation of DDPG

Evaluation of the DDPG and flexible DDPG algorithm is fairly computational intensive. This is because multiple experiments are ran, and each experiment is trained for a number of episodes. Each episode contains a number of timesteps, where each timestep the state transition of the environment, an evaluation step of the Critic and Actor, and a train step of the Critic and Actor needs to be computed. Because of stochastic processes in the algorithms, each experiment has to be repeated multiple times, such that the performance can be averaged. To reduce computational time, both flexible DDPG and DDPG are implemented in a way that is able to run experiments in parallel.

8-1 Modifications to the Actor, Critic and the replay memory

Both the Actor and Critic are neural networks that are compiled to run on the GPU. Because of this, it is difficult to parallelize it using normal multiprocessing techniques in Python. Instead, it is chosen to add the number of experiments as an extra dimension in both the actor and Critic networks. This way one big neural network is compiled at the start of the program. The dimensionality of the input for the actor and critic is increased with one dimension. For the Actor the dimensionality of the input now looks as follows: $[N_{experiments}, N_{batchsize}, N_{states}]$. Equivalently, for the Critic it has become: $[N_{experiments}, N_{batchsize}, N_{states+actions}]$.

To calculate the input to a hidden layer, normally the dot product of the weight matrix and the output of the previous layer would be taken:

$$P = O \cdot W + \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} b \tag{8-1}$$

Or using tensor notation:

$$P_{n,k} = O_{n,m}W_{m,k} + b_k \tag{8-2}$$

Here O is an $N \times M$ matrix representing the output of the previous layer:

$$O = \begin{pmatrix} o_{1,1} & o_{1,2} & \cdots & o_{1,M} \\ o_{2,1} & o_{2,2} & \cdots & o_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ o_{N,1} & o_{N,2} & \cdots & o_{N,M} \end{pmatrix} \tag{8-3}$$

W is an $M \times K$ matrix representing the weight matrix for the connections between the two layers:

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,K} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ w_{M,1} & w_{M,2} & \cdots & w_{M,K} \end{pmatrix} \tag{8-4}$$

and b is a K dimensional bias vector

$$b = \begin{pmatrix} b_1 & b_2 & \cdots & b_K \end{pmatrix} \tag{8-5}$$

Such that dimension N corresponds to the batch size (using index n), dimension M corresponds to the number of incoming hidden neuron (using index m), and dimension K corresponds to the number of upcoming hidden neurons (using index k).

When running the experiments in parallel, these operations need to be generalized to an extra dimension (corresponding to the experiment number). The dot product is still taken over the same dimensions. Expression this new operation in matrix multiplication is difficult, however using tensor notation it is the following:

$$P_{e,i,k} = O_{e,i,j}W_{e,j,k} + b_{e,k} \tag{8-6}$$

Where index e correspond to the experiment number.

Because experiments are run in parallel, the replay memory also gains an extra dimension that corresponds to the experiment. Normally the replay memory contains tuples of states, actions, rewards, and new states, $\langle S, A, R, S' \rangle$. A batch is constructed by randomly drawing samples of tuples from the replay memory, resulting in an array where the first dimension corresponds to the batch size, and the second dimension corresponds to the length of the tuple.

However, in a parallel implementation, the arrays that are sampled during training each get an extra dimension. Now the resulting array has a first dimension that corresponds to the number of experiments, a second dimension that correponds to the batch size, and a third dimension that corresponds to the length of the tuple.

It must be noted that there is a small correlation between samples that are drawn from individual experiments, since at one step, all the state transitions from the experiments are

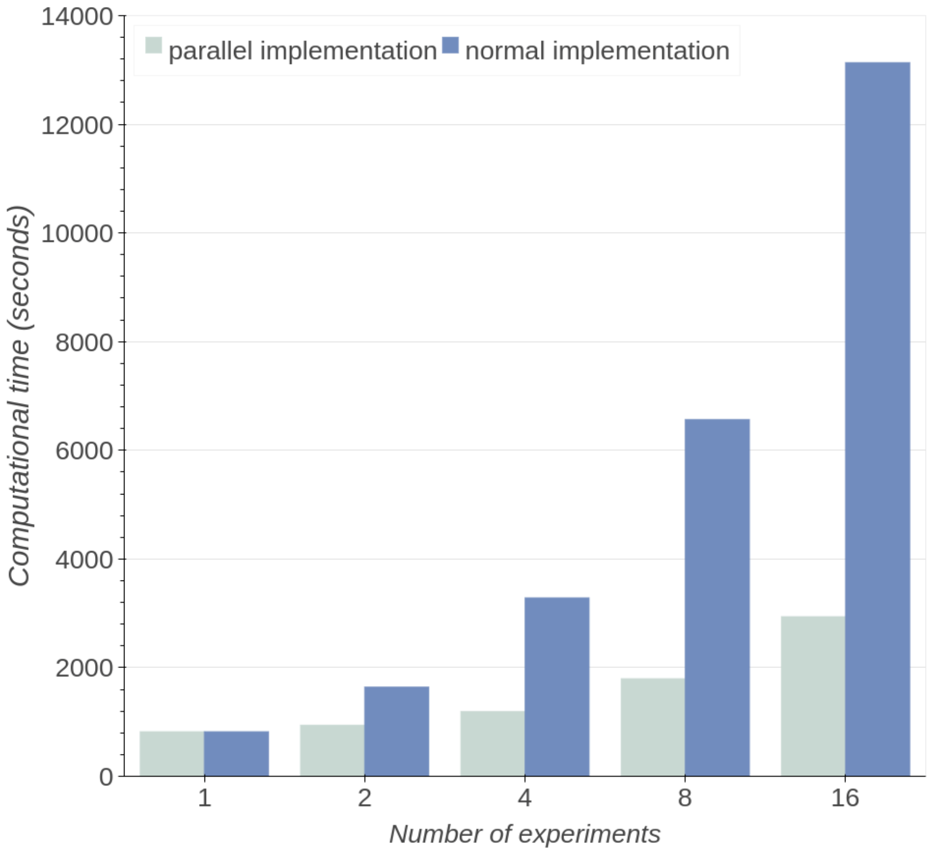


Figure 8-1: Comparison of computational time between the parallel implementation and the normal implementation of the DDPG algorithm. These values correspond to runs of the flexible DDPG algorithm on the cartpole environment with 200 episodes.

combined into one array. When sampling from the replay memory, these arrays are left intact and combined with other arrays to form a batch. So when sampling a batch from the replay memory, the number of unique timesteps corresponds to the batch size, whereas in the ideal case, there would be $N_{experiments} \times N_{batchsize}$ number of unique timesteps. However it is assumed that other stochastic processes (intialization of the neural networks, random initialization, random exploration) ensure that there is a large variation between samples taken at the same timesteps between different experiments.

8-2 Computational efficiency

Because of the parallel implementation of the neural networks, the computational time required for the training and evaluation steps of the networks is decreased. In addition, in the parallel implementation, the networks only need to be compiled at the start of the program, whereas in the normal implementation, a new network has to be compiled every experiment run. In figure 8-1 an overview of the amount of computational time required for both the parallel implementation and the normal implementation can be found.

Chapter 9

Verification and Validation

In this chapter the verification and validation process that was used in this research is summarized.

9-1 Verification

Ideally a reinforcement learning algorithm would be tested on real applications. However this would be a very time-consuming process in practice. Therefore in this research, simulations of environments are used to test the performance of reinforcement learning algorithms. In order to draw a conclusion on the performance of a RL algorithm, it is of uttermost importance that the models of the dynamics in these environments are accurate. If the fidelity of the model is low, the algorithm would likely exploit any simplifications in the model. In addition, dynamics that are unrealistic could result in an environment that is difficult or even impossible to solve. For the preliminary analysis, the model for the pendulum environment has been developed by hand. Verification of this model was done by applying manual input on the model, and checking if behaviour was realistic. For the remaining part of the research, environments of the Deepmind Control Suite library [Tassa et al., 2018] are used. Since this library is built for the purpose of benchmarking reinforcement learning agents, it is assumed that dynamics are realistic and the environment can be solved. This way, the performance of a reinforcement learning algorithm can readily be determined based on these environments.

Calculating the policy-gradients and the gradient of the critic by hand would involve many computations and leave a lot of room for errors. Therefore, Theano [Bergstra et al., 2010] was used as an auto-differentiation toolbox. In addition, Lasagne [Dieleman et al., 2015] was used as a library for building the layers of the neural networks and the optimizer of the neural networks. Theano raises errors if neurons are not properly connected in the network. Theano and Lasagne are libraries that are widely used in deep-learning research, and have been tested and verified thoroughly.

Verification of the implementation of the DDPG algorithm and flexible DDPG algorithm is a difficult process because a lot of modules and steps are involved in the algorithm. One method that was used to verify if the algorithm is working as desired is to use real-time plotting of the learning-curve of the algorithm, in combination with a visualization of the outputs of the neural networks. Real-time plotting of the learning curve of the algorithm could quickly identify if the algorithm is learning properly or not. Visualization of the policy consisted of a surface plot of the actions that are plotted against two input states, the other input states are held constant and can be controlled with sliders. The critic is visualized in a similar manner. By visualization of the policy and critic, it could quickly be determined if the algorithm is learning in a correct way or not. Some pattern should slowly be formed in the actor and critic, because the policy and values have a dependency on the states and actions. In addition it could be checked if the critic is stable, that is, if the critic is mapping towards values that are within the range of the theoretical maximum action-value that is specific for the environment.

9-2 Validation

The validation process of this thesis mainly contains the question if the right process has been used to answer the research objective. The research objective is to reduce the sample complexity of deep reinforcement learning agents by learning in a flexible way. However because environments can have very diverse characteristics, testing the performance of flexible DDPG on only one environment would not lead to a valuable conclusion on the general performance of flexible DDPG. Therefore, in this research diverse environments have been constructed in order to draw a broad conclusion on the performance of flexible DDPG.

Various reinforcement learning architectures have been compared in this research. Drawing a conclusion on the performance solely on the learning curve of the algorithms would be dangerous, since reward shaping is used in the continuous reward environments, and optimizing the actions based on the expected future sum of rewards could lead to undesired behaviour. For example, in the cartpole environment, the goal is to swing the lever up and keep it balanced around the center position. However reward shaping could lead to an optimized policy that keeps rotating the lever around the center position, generating behaviour that is not in line with the goal behaviour of the agent. Visualizations of the policies of the various algorithms have therefore been made in order to see if the learned behaviour of the agent is also in line with the goal behaviour of the agent. In addition, state and action plots (such as in figure 4-3) have been made in order to draw a better conclusion on the performance of a certain algorithm.

Finally, because the DDPG and flexible DDPG algorithms contain many sources of stochasticity, experiments are repeated and the performance of the agents is based on the mean of the returns. In addition, each time an agent is evaluated, the start state is based on a random initialization process. This is to prevent exploitation of 'rote' solutions, memorizations of sequences of actions that lead to high returns.

Part IV

Wrap-up

Chapter 10

Conclusion

In this research the effect of learning in a flexible way for a deep reinforcement learning algorithm is evaluated. The goal of this research is to reduce the sample complexity of a deep reinforcement learning agent by using flexible curriculum learning. Flexible curriculum learning, or flexible reinforcement learning, is a form of learning where the agent can gradually add states and actions to form a policy, in an effort to reduce the sample complexity. This idea is inspired by the way that humans learn, first forming behaviour based on a subset of states and actions, before moving on to the remaining states and actions. For example, when learning to play the piano, one first learns to play with one hand, before playing with both hands. Earlier work on flexible reinforcement learning [Helmer et al., 2018] showed a reduced sample complexity on a quadcopter environment. In [Helmer et al., 2018], a model based Actor-Critic approach was used in combination with Local Linear Regression as a function approximator. Furthermore, recent work showed that deep neural networks can be used as function approximators in reinforcement learning agents [Mnih et al., 2013], [Lillicrap, 2016]. This research modifies a deep reinforcement learning architecture to learn in a flexible way.

The core of this research is twofold, one aspect of it explores flexible reinforcement learning, why it would work, and in which scenario's it would work. The other aspect of this research focuses on the modification of a deep reinforcement learning algorithm such that it can learn in a flexible way.

Flexible reinforcement learning is categorized into three categories, namely flexible state reinforcement learning, flexible action reinforcement learning, and a combination of the two. Flexible state reinforcement learning is likely to result in a weakly observable MDP that is hard to solve. Flexible action reinforcement learning likely results in exploration problems. The last category, namely that of combined flexible state and action reinforcement learning is thought to have the most potential in environments with aggregated reward signals that require some amount of coordination. In those environments, flexible curriculum learning is hypothesised to have the same benefits as decentralized reinforcement learning, namely reduction of the action search space. Because flexible curriculum learning uses a central agent, it is able to capture the coordination between the different subprocesses that are defined by the stages of the curriculum. This is not possible in decentralized reinforcement learning. However it is noted that if there is a large amount of coordination required, optimizing an early stage might result in a policy that is far away from the optimal policy.

A deep reinforcement learning agent called DDPG [Lillicrap, 2016] is modified to learn in a flexible way. DDPG uses neural networks to model both the critic and the actor, and is able to model continuous actions. DDPG uses a replay memory with experience replay to train both the actor and the critic. The neural networks of the DDPG algorithm are modified to learn in a flexible way. The networks have the same amount of input states and actions as the final stage of the curriculum. However input values that are unobservable in the curriculum are masked with zeros, and output actions that are uncontrollable in the curriculum are masked with zeros. This way, gradient updates do not affect the input weights of the unobservable input states. In addition, weights coming from the unobservable states and actions in the curriculum are initialized at zero. This way, once a new stage of the curriculum is opened, the newly observable states have no effect on the original policy, and the original policy is maintained. In new stages of the curriculum, certain input states become observable, and are not zero anymore. This results in the gradient updates now having an affect on those input weights. So the effect of the new input states and actions is gradually learned while the policy of the older stages is maintained for a large part. In addition, because data is masked with zeros by the curriculum, the replay memory is cleared every time a new stage of the curriculum is entered.

To test the performance of flexible DDPG, various environments are created from the Deepmind Control Suite Library [Todorov et al., 2012]. An environment is created where each stage of the curriculum is dedicated to a strongly observable MDP. This environment is called the TwoCartpoles environment, and is solvable with a decentralized reinforcement learning agent. Flexible DDPG is evaluated on the TwoCartpoles environment using a curriculum where in the initial stage the first cartpole is learned and after 50 episodes the second cartpole is added to the agent. Results show that the flexible agent is able to roughly maintain it's old policy for the first cartpole while learning a new policy for the second cartpole. However, the sample complexity is comparable to that of a normal DDPG agent. This is likely due to the clearing of the replay memory having a negative effect on the performance of the flexible DDPG agent. It is also compared with a decentralized variant of DDPG. The decentralized DDPG algorithm has superior performance on the TwoCartpoles environment. Results show that the final policy and critic for the flexible DDPG and central DDPG agents have learned some (unnecessary) form of coordination, so inputs for the second cartpole have an influence on the action of the first cartpole. This is caused by sharing of neurons, and is an explanation why the decentralized variant outperforms the central and flexible DDPG algorithms.

In addition, the flexible DDPG algorithm is evaluated on an environment that is unable to solve with decentralized reinforcement learning. This environment is called the Reacher environment. Both the flexible DDPG and the normal DDPG algorithms are evaluated on this environment. Again sample complexity is comparable for both algorithms. It is difficult to make a conclusion on the performance of flexible DDPG based on these two environments. It could be that the Reacher environment is too easy to solve by a normal agent, hence learning in a flexible way has too little benefits. However some conclusions can be made on the functionality of the flexible DDPG algorithm. First of all, the agent is able to adjust it's policy based on the new stages in the curriculum. Second of all, the agent is able to maintain a policy learned in the previous stages of the curriculum to some extend. One disadvantages of the flexible DDPG algorithm is that the replay memory needs to be emptied, resulting in a loss of performance after every switch of stages in the curriculum. In addition, the network that is used for the critic is a single-column design. This is required to capture possible coordination in environments. However, due to this single-column design, inputs from new stages in the curriculum have to share neurons with the mapping that is learned for the Q-values in the previous stages. In situations where there is no coordination required, sharing neurons makes it difficult for the critic to make a mapping between input and output states of a new stage in the curriculum that is independent on inputs from the previous stage. This can be seen in the results for the TwoCartpoles environment, where the critic finds an unnecessary link between the inputs of one cartpole, and the actions of the other cartpole. However, sharing of neurons in the critic is required to be able to capture potential interactions between different inputs of different stages in the curriculum. This also explains why a decentralized approach is very efficient in the TwoCartpoles environment, because no coordination between cartpoles is required for the optimal policy.

Although recent research on RL algorithms is promising, it is a technique that is still rarely implemented as a control algorithm on real applications. Especially in the aerospace sector, using reinforcement learning as a control algorithm is unpopular due to it's high sample complexity and difficult nature to proof stability. Often RL agents are trained in a simulation, after which the agent is deployed in a real environment. However this requires a model of the environment to be known beforehand, and the fidelity of this model needs to be very high in order to prevent the agent to take advantage of any discrepancies between the simulation and the real world. To conclude, RL still has a long way to go before real-life training and implementation in the aerospace domain become a reality. However the potential of RL is big, since it can learn behaviour in environments where the state transition dynamics are unknown or difficult to model.

Chapter 11

Recommendations

It was shown that flexible DDPG has a comparable sample complexity to a non-flexible approach on certain environments. The flexible DDPG algorithm has a few disadvantages.

The flexible DDPG algorithm uses a single-column critic architecture because it needs to be able to capture coordination between inputs from different stages in the curriculum. This causes the critic to sometimes learn a form of coordination when it is not required. A multi-column architecture inspired by progressive neural networks [Rusu et al., 2016] might be more optimal, since then each stage in the curriculum is assigned a unique parameter space, while cross connections between the columns could capture interactions between inputs. In addition, this research notices some problems of variants of flexible reinforcement learning. Some of those problems might be exploration or coordination problems. Those problems could be relieved when a 'teacher-student' approach is implemented. In this approach, the teacher is an already optimized algorithm that controls the uncontrollable actions in the environment. This would ensure that exploration is possible, and that the policy that is formed in early stages of the environment is already close to the optimal policy.

Because flexible reinforcement learning is applicable in environments that require a small amount of coordination between subprocesses, the current exploration process could be naive . For example, in an environment with a little bit of coordination, if a policy is formed in the first stage, it is known that this policy needs to be altered by a small amount due to the changed MDP in the new stage of the curriculum. It could be chosen to lower the exploration noise on the previously learned action. This different exploration strategy could have a positive effect on the learning rate of flexible DDPG.

Different strategies to reduce the sample complexity of reinforcement learning agents might be more fruitful. One of these methods might be to leverage expert knowledge in a domain and learn from demonstrations. In addition, the current state of exploration strategies seem to be very naive. Smarter exploration strategies could result in faster learning. Also hierarchical reinforcement learning offers a more natural approach to learning, hierarchically mapping patterns of actions, such that complex policies can be constructed by combining different lower level policies.

Appendix A

Hyperparameters used in examples

Table A-1: Hyperparameters used in example 2.1

parameter	**description**	**value**
ϵ	factor by which the exploration probability get's multiplied after each episode.	0.9996
γ	discount factor	0.95
episode length	max amount of steps of an episode	100

Table A-2: Hyperparameters used in example 2.2

parameter	**description**	**value**
ϵ	factor by which the exploration probability get's multiplied after each episode.	0.999
γ	discount factor	0.95
α	learning rate	0.2
episode length	max amount of steps of an episode	100

Table A-3: Hyperparameters used in example 2.3

parameter	**description**	**value**
ϵ	factor by which the exploration probability get's multiplied after each episode.	0.996
γ	discount factor	0.95
λ	eligibility trace factor	0.7
α	learning rate	0.2
episode length	max amount of steps of an episode	100

Table A-4: Hyperparameters used in example 2.4

parameter	**description**	**value**
ϵ	factor by which the exploration probability get's multiplied after each episode.	0.98
γ	discount factor	0.99
α **actor**	learning rate of the actor	0.0005
buffer size	size of the replay memory	100000
episode time(s)	max time of an episode	10
update critic steps	number of steps after which the parameters of the old critic get updated	10
number of hidden layers	-	2
number of neurons per layer	-	400
activation function	-	$relu$

Table A-5: Hyperparameters used in example 2.5

parameter	description	value
ϵ	factor by which the exploration probability get's multiplied after each episode.	0.999
γ	discount factor	0.99
λ	eligibility trace factor	0.7
α **actor**	learning rate of the actor	0.1
α **critic**	learning rate of the critic	0.01
α **model**	learning rate of the model	0.1
episode time(s)	max time of an episode	10
number of radial basis functions	amount of radial basis functions per dimension	12
radial basis function width	W parameter of the radial basis functions	4/11

Table A-6: Hyperparameters used in example 2.6

parameter	description	value
ϵ	factor by which the exploration probability get's multiplied after each episode.	0.98
γ	discount factor	0.99
α **actor**	learning rate of the actor	0.001
α **critic**	learning rate of the critic	0.0001
buffer size	size of the replay memory	100000
episode time(s)	max time of an episode	15
update critic steps	number of steps after which the parameters of the old critic get updated	100
update critic steps	number of steps after which the parameters of the old actor get updated	100
number of hidden layers	-	2
number of neurons per layer	-	400
activation function	-	*relu*

Bibliography

Peter Auer. Using Confidence Bounds for Exploitation-Exploration Trade-offs. *Journal of Machine Learning Research*, 3:397–422, 2003. doi: 10.1109/SFCS.2000.892116.

Yoshua Bengio. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009. URL `http://www.iro.umontreal.ca/bengioy`.

Yoshua Bengio, Umontrealca Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum Learning. In *Proceedings of the 26th Int. Conference on Machine Learning*, pages 41–48. Arxiv, 2009. URL `https://ronan.collobert.com/pub/matos/2009_-curriculum_icml.pdf`.

James Bergstra, Olivier Breuleux, Frederic Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-farley, and Yoshua Bengio. Theano: a CPU and GPU math compiler in Python. *Proc. 9th Python in Science Conference (SCIPY 2010)*, (Scipy):1–7, 2010. ISSN 01436228 (ISSN). doi: 10.1016/j.apgeog.2014.11.014.

Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Eric Battenberg, and Aäron Van den Oord. Lasagne: First Release, 2015. ISSN 09213449. URL `http://dx.doi.org/10.5281/zenodo.27878`.

DJI. DJI Mavic Filters - Cinema Series 6-Pack — Dr. Drone. 2018. URL `https://www.drdrone.ca/products/dji-mavic-filters-cinema-series-6-pack`.

Geoffrey E. Hinton. Learning Distributed Representations of Concepts. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 1:1–12, 1986. URL `https://www.cs.toronto.edu/~hinton/absps/families.pdf`.

Ivo Grondman. *Online Model Learning Algorithms for Actor-Critic Control.* PhD thesis, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, pages 1026–1034, 2015. doi: 10.1109/ICCV.2015.123.

Nicolas Heess, Dhruva Tb, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S M Ali Eslami, Martin Riedmiller, and David Silver Deepmind. Emergence of Locomotion Behaviours in Rich Environments. *arxiv*, 2017. URL `https://arxiv.org/pdf/1707.02286.pdf`.

Alexander Helmer, Coen C. de Visser, and Erik-Jan Van Kampen. Flexible Heuristic Dynamic Programming for Reinforcement Learning in Quad-Rotors. In *2018 AIAA Information Systems-AIAA Infotech @ Aerospace*, Reston, Virginia, jan 2018. American Institute of Aeronautics and Astronautics. ISBN 978-1-62410-527-2. doi: 10.2514/6.2018-2134.

Sergey Ioffe and Christian Szagedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd Int. Conference on Machine Learning*, pages 448–456, 2015. URL `https://arxiv.org/pdf/1502.03167.pdf`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information*, pages 1097–1105, 2012. doi: 10.1145/3065386.

Kai A Krueger and Peter Dayan. Flexible shaping: How learning in small steps helps. *Cognition*, 110:380–394, 2009. doi: 10.1016/j.cognition.2008.11.014.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. ISSN 00189219. doi: 10.1109/5.726791.

Timothy Lillicrap. Continuous control with deep reinforcement learning. *arxiv*, 2016. ISSN 1935-8237. doi: 10.1561/2200000006.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arxiv*, 2013. URL `https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf`.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, oct 1986. ISSN 0028-0836. doi: 10.1038/323533a0. URL `http://www.nature.com/doifinder/10.1038/323533a0`.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive Neural Networks. *arxiv*, 2016. URL `http://arxiv.org/abs/1606.04671`.

David Silver. Lecture 5 : Model-Free Control. *UCL,Computer Science Department, Reinforcement Learning Lectures*, 2015. URL `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/control.pdf`.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014. ISSN 1938-7228.

Richard S Sutton. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Proceedings of the Seventh Int. Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction.* MIT Press, 2012. ISBN 0262193981. doi: 10.1109/TNN.1998.712192. URL `https://mitpress.mit.edu/books/reinforcement-learning`.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego De, Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind Control Suite. Technical report, 2018. URL `https://arxiv.org/pdf/1801.00690.pdf`.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IEEE International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. ISBN 9781467317375. doi: 10.1109/IROS.2012.6386109.