Deep Reinforcement Learning

Pretraining actor-critic networks using state representation learning

J. Munk

Master Thesis
Delft Center for System and Control
April 7, 2016

TU Delft

Deep Reinforcement Learning

Pretraining actor-critic networks using state representation learning

by

J. Munk

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday April 21, 2016 at 10:00 AM.

Student number:	1318578	
Project duration:	Februari 1, 2015 – April 7, 2016	
Thesis committee:	Prof. Dr. R. Babuška,	3mE-DCSC-IC&R, supervisor
	Dr. Ing. J. Kober,	3mE-DCSC-IC&R, supervisor
	Dr. Ing. S. Wahls,	3mE-DCSC-IC&R
	Dr. Ir. J. van Gemert,	EWI-Pattern Recognition & Bioinformatics Group

This thesis is confidential and cannot be made public until December 31, 2016.

TU Delft

Summary

In control, the objective is to find a mapping from states to actions that steer a system to a desired reference. A controller can be designed by an engineer, typically using some model of the system or it can be learned by an algorithm. Reinforcement Learning (RL) is one such algorithm. In RL, the controller is an agent that interacts with the system, with the aim of maximizing the rewards received over time.

In recent years, Deep Neural Networks (DNNs) have been successfully used as function approximators in RL algorithms. One particular algorithm, that is used to learn various continuous control tasks, is the Deep Deterministic Policy Gradient (DDPG). The DDPG learns two DNNs, an actor network that maps states to actions and a critic network that is used to find the policy gradient. The policy gradient is subsequently used to update the actor, in the direction that maximizes the rewards over time.

A disadvantage of using a DNN as function approximator is the amount of data that is necessary to train such a network. Data, which is not always available or can be expensive to obtain. An advantage of DNNs is that they can cope with high-dimensional state and actions spaces, something other (local) function approximators are less suitable for.

State Representation Learning (SRL) is a technique that is typically used to lower the dimensionality of the state space. Instead of learning from the raw observations of the system, SRL is used to map a high-dimensional observation to a low-dimensional state, before learning the RL task. The main idea is that the learning algorithm first learns how to extract the relevant information from the system, before it learns to control it.

In this thesis two algorithms are designed, the Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG) and the Model Learning Deep Deterministic Policy Gradient (ML-DDPG) that both combine SRL with the DDPG algorithm, with the aim of improving the data-efficiency and/or performance of the original algorithm. The two algorithms differ in the type of SRL method that is used. The RP-DDPG uses a concept known as the Robotic Priors, which describes a desirable structure of the state such that it is consistent with physics. The ML-DDPG learns a model of the system.

The algorithms are compared on three different benchmark problems. For each benchmark, various observation vectors are "designed", to simulate different ways of how the information about the state of the system is communicated to the agent. In our experiments, the RP-DDPG is unable to learn two out of three benchmarks problems and requires 4 times more data on the third problem. The ML-DDPG is more successful, it outperforms the original DDPG on one benchmark and performs similarly on the other two.

In general, the DDPG and the ML-DDPG do not learn *state-of-the-art* policies. When the task is to track a certain reference signal, the controlled system has a steady-state error and/or significant overshoot for some of the reference positions. It does, however, learn reasonable policies under very difficult circumstances. It can learn (to some degree) to ignore irrelevant inputs, deal with a reference position that is given in a different coordinate system than the configuration of its body and learn from high-dimensional observations. Most importantly, it does so, without the need to specifically design or alter the algorithm to deal with these challenges.

In a final experiment, the ability of a DNN to generalize what it has learned to examples that differ substantially from examples that it has seen during training is investigated. This ability is what sets a DNN apart from other function approximators and is believed to be the reason why DNN can cope with high-dimensional observations. In the experiment, it is shown that the actor can generalize its policy, i.e., it can produce a control action for observations that differ substantially from observations it has seen during training. Furthermore, the experiment support the claim that a DNN is able to generalize, by learning individual factors that each contribute to the control action independently.

In Darwin's time all of biology was a black box:
not only the cell, or the eye, or digestion, or immunity,
but every biological structure and function because, ultimately,
no one could explain how biological processes occurred.

MICHAEL BEHE

Contents

Acknowledgements vii

1 Introduction 1

1.1 Research Goals and Objectives . . . 2
1.2 Outline . . . 2

2 Background 3

2.1 Deep Learning (DL) . . . 3
2.1.1 Feed-Forward ReLU networks . . . 3
2.1.2 Optimization algorithms . . . 5
2.1.3 Visualization using t-SNE . . . 5
2.2 Reinforcement Learning (RL) . . . 6
2.2.1 Actor-Critic . . . 6
2.2.2 Deep Deterministic Policy Gradient (DDPG) . . . 7
2.3 State Representation Learning (SRL) . . . 8
2.3.1 Unsupervised methods . . . 8
2.3.2 (Semi)-supervised methods . . . 9
2.4 Conclusion . . . 9

3 Combining State Representation Learning with the DDPG 11

3.1 State Representation Learning (SRL) using DNN . . . 11
3.1.1 Robotic Prior method . . . 11
3.1.2 Model Learning method . . . 13
3.2 Saturation penalty . . . 14
3.3 Integration with the DDPG algorithm . . . 14
3.3.1 RP-DDPG . . . 15
3.3.2 ML-DDPG . . . 15
3.4 Conclusion . . . 15

4 Experiment design 17

4.1 General setup . . . 17
4.1.1 Data collection . . . 18
4.1.2 Data preprocessing . . . 18
4.1.3 Evaluation . . . 18
4.2 Benchmark 1: Inverted pendulum . . . 19
4.2.1 Setup . . . 19
4.2.2 Reward function . . . 19
4.2.3 Input design . . . 20
4.3 Benchmark 2: 2-link Arm . . . 20
4.3.1 Setup . . . 20
4.3.2 Reward function . . . 21
4.3.3 Input design . . . 21
4.4 Benchmark 3: Octopus . . . 22
4.4.1 Setup . . . 22
4.4.2 Reward function . . . 22
4.4.3 Input design . . . 22
4.5 Conclusion . . . 23

5 State Representation Learning - Experiments 25

5.1 Learning the state representation . . . 25
5.1.1 Robotic Prior method . . . 25
5.1.2 Model Learning method . . . 26

5.2 Visualising the learned representations 27
5.3 Conclusion 28

6 Policy learning - Experiments **29**
6.1 The Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG) 29
6.2 The Model Learning Deep Deterministic Policy Gradient (ML-DDPG) 29
6.2.1 Benchmark 1: Inverted Pendulum 30
6.2.2 Benchmark 2: 2-link arm. 31
6.2.3 Benchmark 3: Octopus. 33
6.3 Conclusion 33

7 Generalization **37**
7.1 The curse of dimensionality. 37
7.2 Experiment: Symmetry versus Independent factors. 38
7.3 Result 38
7.4 Conclusion 41

8 Conclusion and Recommendations **43**
8.1 Summary and Conclusions 43
8.2 Recommendations for future research 44
8.3 Final words 45

A Paper **47**

Bibliography **55**

Acronyms **59**

Acknowledgements

I would like to thank my supervisors Robert and Jens for their assistance during my research and writing of this thesis. I have really enjoyed the many discussion we had, the enthusiasm they showed and the guidance they have provided along the way.
I would also like to thank Tim, for the many conversations we had about my thesis, Torch and using the Amazon cloud. And for the free coffee during those conversations. It was a tremendous help to have a partner in crime that had to deal with the same practical problems as I did.
Of course, I do not want to forget my friends, who had to listen to all my frustrations and reassure me, without them having the slightest idea what I was talking about. And my parents for believing in me and supporting me financially. Finally, I would also like to thank my girlfriend.
Thank you all!

J. Munk
Delft, April 2016

Introduction

Even before the invention of the computer, self-learning algorithms have fascinated the great minds of the world. Although progress in this field have been slow and general Artificial Intelligence (AI) is still more science fiction than reality, learning algorithms have become more popular. In the field of control, where the objective is to find a mapping from states to actions that steer a system to a desired reference, AI algorithms are also becoming more popular.

The focus of this thesis is on AI algorithms that combine Reinforcement Learning (RL) with Deep Neural Networks (DNNs). In RL, an agent interacts with an environment, with the aim of maximizing the rewards received by the environment over time. A DNN is a machine learning technique, where a network is trained to approximate a certain function. DNNs can be used to solve a variety of problems like object and speech recognition and have been applied with great success [31]. DNN have learned to describe images [9], created new works of art in the style of famous painters [25] and won many Machine Learning (ML) competitions like *Imagenet* and *TIMIT* [7].

Many researchers still consider a DNN to be a black box, for which it is impossible to reason about how or what a network is actually learning. Although it is still hard to interpret how a DNN approximates a certain function, there certainly are more and more theories [2, 15] and theoretical results [28] that help explain why DNNs are so effective. One important difference between DNNs and local function approximators is how they generalize their knowledge to examples not seen during training. A DNN is claimed to generalize globally, where the example can differ substantially from the examples seen during training whereas local approximators can only generalize locally [2].

Recently DNNs have been combined with RL to solve various control problems [22], including the cart-pole benchmark [1] and the *cheetah* locomotion task introduced in [39]. For example, the Deep Deterministic Policy Gradient (DDPG) algorithm presented in [22] learns to map a high-dimensional visual input vector, raw pixel values from a digital camera, to a continuous control action. On many benchmarks, it reached a performance that was similar and sometimes even better than a state of the art planning controller, introduced in [36], with full access to the dynamics of the system. The DDPG trains two DNNs, an actor and a critic network, where the actor learns the control action and the critic is used to obtain the policy gradient from which the actor learns.

A very important component of learning to control is to learn how to efficiently gather the task-relevant sensory information necessary to make informed decisions [41]. Traditionally, one has to tell the learning controller what the relevant information is, which is not always trivial to do. Specifically in the case of a robot equipped to do a wide variety of tasks, the amount of sensory information necessary to do one such particular task is often far less than the total amount of information that is received by the robot [17]. Furthermore, this input may require preprocessing before it can be used effectively. Some sensory inputs may need to be combined (e.g. multiple joint angles that determine the position of an end effector), other inputs may need to be transformed into a common coordinate system and others may be ignored, because they are irrelevant to the task at hand. Selecting the right information and preprocessing it, to a suitable input for the learning controller, therefore, requires a fair amount of prior knowledge about the system. It would be much more

appealing if the learning controller could learn this by itself.

In the RL community, this is generally referred to as the State Representation Learning (SRL) problem [17]. In SRL, one tries to learn a mapping, between raw sensory inputs or observations to a state, before learning the control action [17] [33]. The challenge of SRL is that the optimal state, from which to solve the problem, is generally unknown. Learning the observation-to-state mapping, therefore, involves either making assumptions about the structure of the state representation (unsupervised learning) or learning the mapping as part of learning some other function (semi-supervised learning).

In the DDPG, a DNN learns to control end-to-end, i.e., the network learns to map the raw observations directly to a control signal, without the need for feature engineering. Although this is believed to reach a performance superior to a more indirect approach, it does require a lot of data [5]. Data which is not always available or can be very expensive to obtain. In order to learn more efficiently, some have tried to combine RL with SRL [5, 16, 17, 29], such that a mapping from observations-to-states is learned explicitly, prior to solving the RL task. In some cases, this has significantly improved both the speed of convergence and the final performance. These examples have, however, all focused on learning from visual observations and none of them integrates these methods with algorithms that combine RL with DNNs.

1.1. Research Goals and Objectives

The general goal of this thesis is to investigate how SRL can complement the DDPG algorithm and for which type of observations, i.e, an observation that contains irrelevant input or one that requires extensive preprocessing, this results in a better performance.

The following hypotheses are formulated about combining SRL with the DDPG algorithm:

- Learning a state representation prior to training an actor and critic network using the DDPG algorithm leads to faster convergence and/or improved performance of the final policy.
- Learning a state representation prior to training an actor and critic network makes the DDPG algorithm more data efficient.
- The benefit of using SRL increases when the observation vector requires more preprocessing or contains more irrelevant input.

In order to verify these hypotheses three benchmark problems are studied, the single pendulum, a 2 link robotic arm and an octopus arm. For each problem a set of observations are made available to simulate various situations in which the sensory information that is available is imperfect, i.e., more sensory information is available than necessary to solve the problem, the reference is given in a different coordinate system than the other inputs or velocity measurements are omitted and replaced by a series of subsequent positional measurements.

A second goal is to experimentally show how an actor network generalizes, the policy it has learned, to areas it has not seen during training. The goal of the experiment is to show to what extent generalization works and to provide some insight into why it works.

1.2. Outline

This thesis is organized as follows. Chapter 2 gives an overview of important prior work in the field of RL, Deep Learning (DL) and SRL. Chapter 3 details the implementation of two SRL methods using a DNN, which are subsequently combined with the DDPG to form two new algorithms the Model Learning Deep Deterministic Policy Gradient (ML-DDPG) and the Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG). In order to test the performance of these algorithms, three benchmark problems are introduced in Chapter 4.

The comparison of the algorithms is discussed in two separate chapters. In Chapter 5, the high-dimensional states, learned by the network, are visualized in order to compare the two SRL methods. In Chapter 6 an overview is given of the performance of each algorithm on all the benchmarks, to validate the hypotheses formulated in the previous section.

In Chapter 7 a practical experiment is performed to show how and to what extend a DNN can generalize the policy it has learned, to situations it has never seen before. The ability to generalize is important when the state and/or action dimensions are very high-dimensional. Finally, Chapter 8 concludes this thesis.

Appendix A contains a paper that is submitted to the Conference on Decision and Control and focuses specifically on the ML-DDPG.

2
Background

This chapter introduces three important concepts Deep Learning (DL), Reinforcement Learning (RL) and State Representation Learning (SRL) on which the work in the rest of this thesis is based. Behind each concept lies an entire field of research, the individual sections are therefore by no means meant to give a complete overview of the respective fields. The aim is to introduce the parts that are relevant for understanding the rest of this thesis. Furthermore, it should provide the reader with some perspective of how this thesis relates to previous work done within the respective fields of research.

2.1. Deep Learning (DL)

In the last decade, much progress has been claimed in the performance of Neural Networks (NNs). In both image recognition and voice recognition, they now outperform any other algorithm [31]. In many cases Deep Neural Networks (DNNs), where the network has multiple non-linear layers, are favored over shallower networks. These deeper architectures, and the ability to train such networks on large amounts of data, are believed to be the main reason that has led to the increased performance of NNs.
For an extensive introduction into how NNs work the reader is referred to [30]. In the following sections, some aspects of DL are discussed, that are considered important for understanding the research presented in this thesis. Section 2.1.1 introduces the equations that describe a feed-forward NN with Rectified Linear Unit (ReLU) activation functions and discusses recent theoretical work of these types of networks. Section 2.1.2 discusses the optimization algorithms that are used to train NNs. And finally, Section 2.1.3 explains t-Distributed Stochastic Neighbor Embedding (t-SNE), a method that can visualize the high-dimensional internal representations, learned by a network.

2.1.1. Feed-Forward ReLU networks

A NN consists of a collection of *neurons*, connected with each other according to some *architecture*. In this thesis the NNs that has been studied are feed-forward networks with fully connected layers, where each neuron belongs to a certain layer and is connected to all neurons in the next layer. Figure 2.1 shows a schematic diagram of this type of architecture with 1 hidden layer.
The mapping between some input vector $x \in \mathbb{R}^n$ and the output $y \in \mathbb{R}^m$ can be described as a composition of L layers that defines a function $P : \mathbb{R}^n \to \mathbb{R}^m$ as

$$P(x; w) = p_L \circ q_{L-1} \circ p_{L-1} .. \circ q_1 \circ p_1(x) \tag{2.1}$$

where p_l is an affine linear function and q_l a non-linear activation function and w the aggregated parameters of these functions. Calculating the output y from x, given some parameter w, can be seen as propagating the input signal through the network layer by layer. This is also referred to as performing a forward pass through the network. The output of the l-th layer is a vector $x^l \in \mathbb{R}^{n_l}$ for $l < L$ and is given by

$$x^l = q_l(p_l(x^{l-1})) \tag{2.2}$$

where $x^0 = x$. The function $p_l : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$ is an affine linear function

$$p_l(x^{l-1}) = W_l x^{l-1} + b_l \tag{2.3}$$

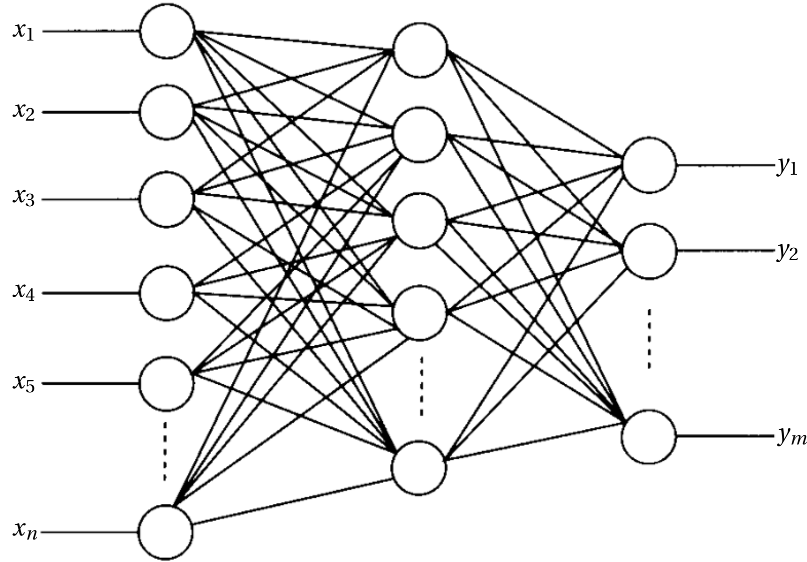


Figure 2.1: Schematic diagram of a feed-forward NN with fully connected layers.

where $W \in \mathbb{R}^{n_{l+1} \times n_l}$ is a matrix of weights and $b \in \mathbb{R}^{n_{l+1}}$ a vector of biases. The function q_l is the activation function, which for a ReLU neuron is defined as

$$q_l(z) = \max(0, z). \tag{2.4}$$

The advantage of the ReLU activation function is that gradients flow well on the active parts of neurons since the activation function does not alter the gradients for neurons that are activated. DNNs with ReLU neurons, therefore, do not suffer from the vanishing gradient problem [11]. The use of ReLU activation functions has also simplified the theoretical analysis, with respect to other activation functions that were used in the past, like the sigmoid or tanh functions [11]. This is because the non-linear activation function q does not alter the value of the output, other than determining whether it should be activated or not.
Denote by $\beta \in \{0,1\}^k$ a binary activation vector where k are the number of ReLU neurons in the network and $\beta_i = 1$ if the i-th neuron is activated (non-zero) and $\beta_i = 0$ if the i-th neuron is not activated. Then for each value of β there exists a matrix $W^- \in \mathbb{R}^{m \times n}$ and a vector of biases $b^- \in \mathbb{R}^m$ for which

$$y = W^- x + b^-. \tag{2.5}$$

The matrix W^- and bias vector b^- can be obtained by removing all neurons and connections to the neurons which are not activated and then traveling along each path from every input to every output and multiplying the individual weights between every individual input-output pair.
The NN essentially divides the input x into regions, where each region is associated with a unique value of β. Within each of these regions, the output y is an affine linear function of the input x. The NN learns to approximate a function by learning a collection of linear models and when to activate which model, given the input x. In [28] it is shown that, in the asymptotic limit of many hidden layers, a DNNs can separate the input space in exponentially more regions than a shallow NN for the same number of neurons. Deeper NNs, therefore, need less neurons and thus less weights to approximate a particular function than shallower ones.

A second important concept, that gives some insight in why DNN are such effective function approximators, is the concept of distributed feature representations [2]. In a distributed representation, given by a vector $upsilon \in \mathbb{R}^m$, each value in the vector $upsilon_i$ represents an independent factor that is activated to some degree or is zero. If we were to classify animals, these factors could represent things like hairy, 4-legged, striped, long neck etc. Different animals will then be represented by different activation patterns of these factors, where individual factors will be reused across different animals. Furthermore, each of these factors are assumed to contribute to the target function, independently of the activation of the other factors. At least for a DNN with a linear output layer. A DNN is a global approximator, it learns to identify these factors across the entire input space, whereas local approximation methods learn separate factors for each region of the

input space [2]. Local approximators are therefore known to suffer from the curse of dimensionality, which states that when the dimension of the input space grows, the number of parameters (and data to learn these parameters) grows exponentially.

2.1.2. Optimization algorithms

Optimization algorithms are used to find the parameters w of a DNN P, such that it minimizes some objective function E. In supervised learning, the objective function E is typically a minimization of the Mean Squared Error (MSE) of a set of N training examples $D = \{x, y\}$ given by

$$E(D, w) = \frac{1}{N} \sum_{x,y \in D} \frac{1}{2} (P(x; w) - y)^2 \tag{2.6}$$

where w are the parameters of the network defined by P. Since the objective function is based on a set of samples, which are drawn from some unknown distribution, the problem is considered a stochastic optimization function.

One of the advantages of NNs is that obtaining the partial derivative of the gradient with respect to each parameter is computationally just as expensive as evaluating the function itself. Because of this, most optimization methods used to train NN are first-order gradient methods.

In the most basic form, this algorithm is known as gradient descent in which the parameters are updated at every iteration, in the direction that minimizes the error. The update rule is simple

$$w_{k+1} = w_k - \alpha \frac{\partial E}{\partial w} \tag{2.7}$$

with $\alpha > 0$ the learning rate and k the iteration index.

Although obtaining the gradient for a single training example may be computationally inexpensive, training a DNN with gradient descent on a big dataset, can still be very slow. This is because the objective function is a summation of subfunctions, where each subfunction is evaluated by a different sample. Thus, in order to obtain the gradient $\frac{\partial E}{\partial w}$, one needs to make a forward and backward pass through the NN for each training example in the dataset.

Stochastic Gradient Descend (SGD) circumvents this problem by looping through each of the training examples, calculating the gradient for a single example and updating the weights accordingly. An implementation of SGD that is computationally more efficient randomly divides the dataset into a set of so-called mini-batches. During the optimization, the algorithm loops through the mini-batches and updates the weights w for each mini-batch. This makes training a DNN on a large dataset much more efficient [3].

An important parameter of any gradient descent algorithm is the learning rate α. A learning rate that is too low results in slow learning while a learning rate that is too large leads to oscillations during learning [30]. Finding the right learning rate, for a given problem, is usually done by manually tuning this parameter.

Adam, introduced in [18], is a stochastic optimization method which adjusts the learning rate for each of the individual parameters automatically, using statistics of the gradient collected during the optimization. It determines the learning rate based on the first order moment, the mean m, and the second order moment, the uncentered variance v. It uses these parameters to calculate a signal-to-noise ratio $m/\sqrt{v}$, where a high signal-to-noise ratio represents a low uncertainty about the gradient and a low signal-to-noise ratio a high uncertainty. The individual learning rate for each parameter is calculated by multiplying the signal-to-noise ratio with a base learning rate, resulting in the following update rule

$$w_{k+1} = w_k - \alpha \frac{m}{\sqrt{v}} \cdot \frac{\partial g}{\partial w} \tag{2.8}$$

where $\cdot$ represents the inner product.

The main advantage of using Adam is that it automatically finds the right learning rate for many different problems, without the need to change any of the hyper-parameters [18]. Another advantage of Adam is that it scales down the learning rate during learning since the signal-to-noise ratio tends to become closer to zero near the optimum, which improves the overall convergence. In all algorithms, described in Chapter 3, Adam was used as the optimization method to update both the actor and critic networks.

2.1.3. Visualization using t-SNE

One of the drawbacks of using DNNs is that it is difficult to interpret the output of the hidden layers within the network, due to the dimensionality of these outputs. Even for a simple benchmark problem, like the pendulum introduced in Chapter 4, the outputs of the hidden layers are 100-dimensional. For people, that live in

a 3-dimensional world, points in a 100-dimensional space are hard to visualize let alone interpreted.
One approach, that is commonly used to circumvent this problem, is dimensionality reduction, in which a set of high dimensional datapoints is converted to a set of 2-dimensional datapoints that can subsequently be shown in a scatterplot. The challenge, when performing dimensionality reduction, is to preserve as much of the relevant structure as possible, i.e., datapoints that were similar should appear close together and datapoints that were dissimilar should appear far from each other, on the 2-dimensional map.
One method that is often used within the DL community is t-SNE, which claims to be capable of capturing much of the local structure of the original dataset while also revealing the presence of clusters at several scales [23]. The method converts the Euclidian distance between two high-dimensional vectors x_i and x_j, to probabilities that x_i would choose x_j as its neighbor, given that neighbors are picked in proportion to their probability density under a Gaussian centered at x_i [23]. It calculates the same probabilities for the points on the 2-dimensional map and then optimizes the transformation such that the difference between those two probabilities is minimized.
t-SNE can be used to visualize the output of the hidden layers of a NN, to see if the representation that the network has learned, reveals certain clusters. By labeling each datapoint, one can investigate if these clusters represent something meaningful. For instance, observations of a pendulum (presented as a benchmark in Section 4.2) could be labeled according to the current angle of the pendulum and its angular velocity, to visualize the relation between these characteristics and the high-dimensional data vector the network has learned. One would expect to see observations that were taken at similar angles and angular velocities, to appear close together on the 2-dimensional map. This experiment is carried out and included in Chapter 5.

2.2. Reinforcement Learning (RL)

In RL, a learning agent interacts with an environment with the aim of maximizing the rewards received from the environment over time [35]. RL is applicable in a wide variety of learning problems and is therefore considered by many as the Artificial Intelligence (AI) algorithm. It is an algorithm that allows agents to learn autonomously by means of trial and error.

More formally, an RL problem is modeled as an Markov Decision Process (MDP) described by the tuple $M = (S, A, f, r)$, where the state space S is a set of states $s \in \mathbb{R}^m$, the action space A is a set of actions $a \in \mathbb{R}^p$, $f : S \times A \to S$ is the state transition function, and $r : S \times A \to \mathbb{R}$ is the reward function. At each timestep t, the agent receives an observation $o_t \in \mathbb{R}^n$ that determines its current state s_t, it chooses an action a_t, receives a scalar reward $r_{t+1} \in \mathbb{R}$ according to the reward function r and transits to state s_{t+1} according to the transition function f.

The goal in RL is to learn a control policy $\pi : S \to A$ that maximizes the discounted sum of future rewards $R_t = \sum_{t=i}^{T} \gamma^{i-t} r(s_t, a_t)$ where $\gamma \in [0,1]$ is the discount factor and T the number of time steps per learning episode.

The action-value function Q is often used in RL algorithms to denote the expected future reward given an action a_t taken in state s_t and thereafter following the policy π by taking the action $a^\pi = \pi(s)$. The Q-function, in the form of a difference equation, is given by

$$Q^\pi(s_t, a_t) = r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})). \tag{2.9}$$

The optimal action-value function Q^*, that maximizes the discounted sum of rewards, is defined as

$$Q^*(s_t, a_t) = \max_\pi \Big(r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})) \Big). \tag{2.10}$$

Given that an agent has learned the optimal action-value function Q^* it can optimize its behavior by choosing the optimal action a_t at every time instance. The optimal policy π^* is therefore defined as

$$\pi^*(s_t) = \arg\max_a \Big(Q^*(s_t, a_t) \Big).$$

This method is known as a critic-only method. The downside of this method is that for continuous action spaces, determining the policy requires a complicated optimization at every time step. To circumvent this optimization actor-critic methods can be used which are explained in the next section.

2.2.1. Actor-Critic

In applications like robotics, where the state and action spaces are continuous, function approximators have to be used to approximate both the action-value function Q and the policy π [4]. Actor-critic algorithms are suitable in these situations since they allow both of these functions to be learned separately.

In actor-critic methods, the critic learns the action-value function Q while the actor learns the policy π. In order to ensure that updates of the actor improve the expected discounted return, the update should follow the policy gradient [34]. The main idea behind actor-critic algorithms is that the critic provides the actor with the policy gradient. In theory, the critic should have converged before it can provide the actor with an unbiased estimate of the policy gradient, in practice, however, this requirement can be relaxed as long as the actor learns slower than the critic [34].

2.2.2. Deep Deterministic Policy Gradient (DDPG)

The Deep Deterministic Policy Gradient (DDPG) algorithm is an off-policy actor-critic algorithm, first introduced in [22]. In this algorithm, both the actor and the critic are approximated by a DNN with parameter vectors ζ and ξ, respectively. The critic is trained by minimizing the squared Temporal Difference (TD) error given by

$$\mathscr{L}(\xi) = \Big(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\zeta)|\xi) - Q(s_t, a_t|\xi)\Big)^2. \tag{2.11}$$

The actor is updated in the direction of the policy gradient $\nabla_\zeta Q$ using the current approximation of the critic. The update of ζ with $\triangle\zeta$ is given by

$$\triangle \zeta = \nabla_a Q(s_t, \pi(s_t|\zeta)|\xi) \nabla_\zeta \pi(s_t|\zeta).$$

According to [32] the Q-function should be in the *compatible* form in order for the policy gradient to be unbiased. Although this is violated in the DDPG algorithm, with the addition of a few extra stability measures, the algorithm has been shown to work well in practice. See Algorithm 1 for an overview of DDPG.

Algorithm 1 Deep Deterministic Policy Gradient (DDPG)

{Actor-Critic Learning}
Randomly initialize network weights ζ and ξ
$\zeta^- \leftarrow \zeta$ and $\xi^- \leftarrow \xi$ {set weights of target network}
for learning step = 1 **to** N **do**
  Sample random mini-batch from Experience Replay Database
  Calculate $\triangle\zeta$ and $\mathscr{L}_c$ over mini-batch
  $\zeta \leftarrow \zeta - \alpha \triangle \zeta$ and $\xi \leftarrow \xi - \alpha \frac{\partial \mathscr{L}_c}{\partial \xi}$
  $\zeta^- \leftarrow \tau\zeta + (1-\tau)\zeta^-$ and $\xi^- \leftarrow \tau\xi + (1-\tau)\xi^-$ {update target network}
end for

Earlier work, that had tried to use DNNs in RL often suffered from the fact that the Q-function is not guaranteed to converge when using a global approximator like a NN. In the past years, many papers [22, 24, 29] have focused on solving this problem, motivated by the performance of DNNs in other fields. This has led to a set of "tricks" that are considered necessary to achieve practical results, although they have not yet led to any theoretical proof of convergence.

Experience Replay Database

One problem of using a global approximator is that changes to the parameters of the function approximator have global consequences. If one updates these parameters based on a local error, i.e., an error that occurs in a certain state, the approximator can suffer from what is known as catastrophic forgetting where the network forgets what it has learned in some part when updating some other part.
One solution to prevent catastrophic forgetting was proposed in [29] and involves the so-called experience replay database. In an experience replay database samples from interacting with the system are stored, such that they can be reused at a later stage. Instead of learning from a single experience, the experiences stored in the database are used to create mini-batches, the errors (e.g., TD-error for the critic and policy gradient for the actor) are then calculated for the entire mini-batch. Since the mini-batches are assembled randomly, the combined error gives a good approximation of the global error, instead of a local error.
Using an experience replay database solves the issue of catastrophic forgetting although care should be taken to keep the data within the database varied enough to prevent the network from over-fitting to certain areas of the state space [6].

Using separate target networks
Another significant problem occurs when minimizing (2.11) [24]. The updates of the parameter ξ not only change the output of the critic network $Q(s_t, a_t|\xi)$, but they also change the target function $r_{t+1}+\gamma Q(s_{t+1}, \pi(s_{t+1}|\zeta)|\xi)$ that the network is learning. This is due to the recursive nature of the action-value function. Similarly, updates to the actor parameter ζ also change the target function. This coupling can lead to unstable behavior and can cause the learning of the action-value function to diverge.

A solution, proposed in [22], that reduces the coupling between the target function and the actor and critic networks, is to update the parameters of the target function using "soft" updates. Instead of using ζ and ξ directly, a separate set of weights ζ^- and ξ^- are used, which slowly track the parameters ζ and ξ of the actor and critic networks.

The "soft" updates are performed after each learning step, using the following update rule

$$\zeta^- \leftarrow \tau\zeta + (1-\tau)\zeta^-, \qquad \xi^- \leftarrow \tau\xi + (1-\tau)\xi^-$$

where $\tau \in (0, 1]$ represents the trade-off between the learning speed and stability. Using these new parameters, the squared TD error becomes

$$\mathscr{L}(\xi) = \Big(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\xi^-)|\zeta^-) - Q(s_t, a_t|\xi)\Big)^2.$$

L2-penalty on critic weights
A final problem that can hamper the convergence of the action-value function is the fact that the Q-values, the critic is learning, are not within a predefined range. This is caused by the recurrent nature of the action-value function. In supervised learning, the target values are usually normalized to be within a certain range, before the network is trained to learn these values. The Q-values that the critic is learning are, however, unknown before the learning starts. As the learning progresses these Q-values tend to increase, which is undesirable. In order to keep the Q-values within a certain range, an L2-penalty on the weights can be added to the cost function. An L2-penalty adds the minimization of the 2-norm of all weights as an extra optimization objective. The reasoning behind this is that in order for the network to output large values, the weights need to be large since the input will always stay within a certain range. By preventing the weights from growing unrestrictively, the Q-values are ensured to be bounded. Including the L2 penalty, the loss function of the critic becomes

$$\mathscr{L}_c(\xi) = \Big(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\xi^-)|\zeta^-) - Q(s_t, a_t|\xi)\Big)^2 + \|\zeta\|.$$

2.3. State Representation Learning (SRL)

In RL, the state s is not always directly accessible but needs to be constructed from a set of observations o. Such an observation-to-state map $\Sigma : O \rightarrow S$ can be the result of feature engineering, in which an engineer selects the observations and design the mapping, but this can also be learned from data. The process of learning the observation-to-state mapping is called State Representation Learning (SRL) [17].

State representation learning is a form of unsupervised learning, i.e., there are no training examples available since it is not known a priori what the most suitable state representation is to solve the problem. Learning an observation-to-state mapping, therefore, involves either making assumptions about the structure of the state representation (unsupervised method) or learning the mapping as part of learning some other function (semi-supervised method). For both approaches, a few examples are given in the following sections.

2.3.1. Unsupervised methods

In [10, 19, 37], an auto-encoder is used to find an observation-to-state mapping in which the observations are compressed into a low-dimensional state vector. The objective, during training, is to find states from which the original observations can be reconstructed. It subsequently learns a state representation that captures only the unique features of the observation, i.e., how they differ from other observations.

Another unsupervised method is Slow Feature Analysis (SFA) [40], which is based on the idea that most phenomena in the world change slowly over time. In [10, 20] this assumption is used to learn a mapping between visual observations and a state representation that gradually changes over time.

In [17], these and several other assumptions about the structure of a good state representation are combined into the so-called Robotic Priors. They are divided into

- **The simplicity prior** states that only a small amount of world properties are relevant to solve a particular task and is usually implemented by forcing the state to be low-dimensional.

- **The temporal coherence prior** is also known as Slow Feature Analysis (SFA) [40] and is based on the idea that most properties in the world change slowly over time.
- **The proportionality prior** states that the amount of change in the state representation, as a result of the action taken in that state, should be proportional to the action that was taken in that state. This is based on Newton's second law of motion $F = m \times a$ which governs how forces transform into motion in physical systems.
- **The causality prior** states that the reward is a direct result of taking an action in a particular state. Similar state action tuples should, therefore, lead to similar rewards.
- **The repeatability prior** states that the whenever an action is taken twice, given the same or similar states, the resulting change in the state should be similar. This prior is also based on Newton's second law of physics, but also follows from the Markov property which assumes that future states only depend on the current state and the action taken in that state.

For each of these priors, a loss function is defined. An observation-to-state mapping is subsequently trained to minimize the combined loss functions of the individual priors. The paper then shows a performance increase when using the learned state representation instead of the raw observations as input to the Neural Fitted Q-iteration algorithm [29].

2.3.2. (Semi)-supervised methods

In [16, 37, 38] an observation-to-state mapping is learned using a semi-supervised approach. Instead of relying on priors that describe the structure of a state, the mapping is learned as part of learning a function for which there are training samples available. There are two candidate functions, for which these training samples are generally available in an RL setting, that is, the reward function r and the state transition function f.

In [16] Radial Basis Functions (RBFs) are used to learn an observation-to-state mapping Σ that maps a vector of continuous observations $o \in \mathbb{R}^n$ to a binary state vector $s \in \{0,1\}^m$. The parameters of the RBF functions are updated in a way that minimizes a loss function defined as

$$\mathscr{L}(\Sigma) = \mathscr{L}_{\text{transition}}(\Sigma) + \mathscr{L}_{\text{reward}}(\Sigma) + \|\Sigma\|$$

where $\|\Sigma\|$ is included for regularization, e.g., to keep the number of activations in s small. The $\mathscr{L}_{\text{transition}}(\Sigma)$ and $\mathscr{L}_{\text{reward}}(\Sigma)$ are determined at every iteration by using a Nearest Neighbour predictor to learn the transition and reward function, given the current observation-to-state mapping. One motivation for using both functions is an analysis done in [27] that shows that a linear value function approximation of the Bellman equation (2.9) can also be decomposed in an approximation of the reward and transition function.

A problem for the method described in [16] is the lack of computational efficiency. Since the errors resulting from the loss function do not provide a gradient, a stochastic search algorithm is used which requires 10^5 evaluations of the loss function to converge. For each of those evaluations, it needs to retrain the Nearest Neighbour predictor to predict both the reward and transition functions given the current observation-to-state mapping, which makes this approach very computationally expensive.

2.4. Conclusion

In this chapter, the DDPG algorithm is introduced, which is an off-policy actor-critic algorithm that uses DNNs as function approximator. Various important elements of the algorithm are explained, the ReLU neurons, the Adam optimization algorithm and the loss functions of the actor and critic networks. Furthermore, multiple SRL methods are explained, which are categorized as either an unsupervised method or a semi-supervised method. In the following chapters, new algorithms are created that combine the DDPG with SRL.

3

Combining State Representation Learning with the DDPG

In this chapter, two new algorithms are introduced the Model Learning Deep Deterministic Policy Gradient (ML-DDPG) and the Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG) both of which combine a different State Representation Learning (SRL) method with the original Deep Deterministic Policy Gradient (DDPG). Both algorithms aim to improve the DDPG, specifically in settings where the amount of data is limited and the observations the agent receives from the system are not very suitable to learn from. Instead of learning the actor and critic networks end-to-end, using the raw observations as input to the network, both use SRL to first learn an observation-to-state map such that both the actor and critic network can learn from a shared state representation.
In Section 3.1, the two SRL methods described in Chapter 2.3 are formulated as a Deep Learning (DL) problem. By creating a Deep Neural Network (DNN) that learns to map observation to states, SRL can be easily combined with the DNNs from the DDPG algorithm. In Section 3.2, a specific problem is addressed that concerns the way all three algorithms deal with input saturation. The fact that this problem occurred in our setup and was not mentioned in [22], is probably caused by the fact that in our experiments the algorithms learned off-policy using a dataset that was created before the start of the experiment. This setup was chosen to be able to compare the different algorithms on exactly the same dataset. Finally, in Section 3.3 the full algorithms are both introduced.

3.1. State Representation Learning (SRL) using DNN

This section describes the implementation of two different approaches to SRL. First, the implementation of an unsupervised approach is described, in which a Neural Network (NN) tries to learn a state representation that is consistent with physics. This method is referred to as the Robotic Prior method. Second a semi-supervised SRL method is explained. This method is also referred to as a Model Learning method since it essentially learns a prediction model of the system while learning the observation-to-state mapping. For each of these methods the architecture of the NN is discussed and the loss function on which the network is trained is specified.

3.1.1. Robotic Prior method

The robotic prior method, introduced in [17], is an unsupervised learning technique. In the original paper, it is used to map a high-dimensional visual input (e.g. all pixels of a screen) to a state representation, using a linear mapping function. In this thesis, the implementation of the method is slightly altered, because our agent does not learn from visual inputs and our objective is to integrate the method with the DNNs from the DDPG. The following sections will specify our implementation of the Robotic Priors.

Network architecture

The network architecture, shown in Figure 3.1, consist of a single layer that maps an observation o_t to a state s_t. In contrast to the network presented in the next section, the learned representation is judged on its consistency with physics [17], therefore, no other outputs then the state s_t are needed.

In the loss function, presented in the next section, up to four different observations need to be mapped to a state. The network is therefore cloned four times and assembled in parallel, where each network shares the weights with the other networks. The advantage of this implementation is that the error and partial derivatives with respect to the weights can be done in a single forward and backward pass, instead of four individual passes.

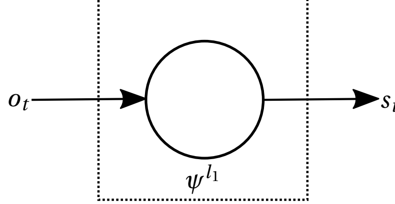


Figure 3.1: Network architecture of a DNN used to map observations to states. The network takes an observation o_t as input and produces a state s_t.

Loss function

The Robotic Priors, as defined in [17], are the *simplicity prior*, the *temporal coherence prior*, the *proportionality prior*, the *causality prior* and the *repeatability prior*. For each of the different priors, an individual loss function is defined. The final loss function $\mathscr{L}_{\text{rp}}$ is a summation of the individual loss functions given by

$$\mathscr{L}_{\text{RP}}(\Sigma) = \mathscr{L}_{\text{simplicity}}(\Sigma) + \mathscr{L}_{\text{temporal coherence}}(\Sigma) + \mathscr{L}_{\text{proportionelity}}(\Sigma) + \mathscr{L}_{\text{causality}}(\Sigma) + \mathscr{L}_{\text{repeatability}}(\Sigma)$$

where Σ is the current observation-to-state mapping. There are three important differences between the implementation described in [17] and this implementation. The first is the *simplicity loss*, which is implemented by a sparsity penalty on the state representation (3.1). In [17] it is enforced by a state representation of a fixed, low dimensionality. For a network with a single layer, the number of weights depends only on the number of inputs and outputs. Since the dimension of the input, for the benchmark problems defined in Chapter 4, is already relatively low, enforcing a low dimensional output would severely limit the amount of parameters of the network.
The second difference is that the Robotic Priors defined in [17] assumes a discreet action and reward space, since this assumption does not hold in our setup, the loss function of some of the priors needs to be altered. The problem arises from the fact that some of the priors (i.e. the *proportionality prior*, the *causality prior* and the *repeatability prior*) are conditioned on the fact that the same action is taken at two different time instances $a_{t_1} = a_{t_2}$. Since, in our setup, an action is assumed to be a continuous value (see Section 2.2), the chances of applying exactly the same action twice are rather slim. Therefore, the requirement of the exact same action is replaced by a similarity term $e^{-\|a_{t_2}-a_{t_1}\|_2}$. This similarity term is 1 if the actions taken at time instance t_1 and t_2 are equal and approaches 0 with increasing distance between the actions. A similar term is already used in the loss function of the *causality prior*, to measure the similarity between two continuous states [17]. Similarly, the *causality prior* is also conditioned on a different reward $r_{t1} \neq r_{t2}$ received at two different time instances. This condition is replaced by adding a dissimilarity term $1 - e^{-\|r_{t_2}-r_{t_1}\|_2}$ to the loss function.

Many of the Robotic Priors depend on a comparison between two samples. For instance, the *proportionality prior* states that, if the robot has performed the same action at times t_1 and t_2, the change of the state $\triangle s_t = s_{t+1} - s_t$ must change by the same magnitude. A benefit of replacing the conditions with the similarity (or dissimilarity) terms is that now these two samples can be *any* two samples. The third difference is therefore that instead of finding a second sample using some search method, a second sample is just chosen randomly.
Given the current observation-state-mapping $s_t = \Sigma(o_t)$ learned by the NN and two randomly selected samples $\{o_{t_1}, a_{t_1}, r_{t_1}, o_{t_1+1}\}$ and $\{o_{t_2}, a_{t_2}, r_{t_2}, o_{t_2+1}\}$ the individual loss functions of the different Robotic Priors are

given by

$$\begin{aligned} \mathscr{L}_{\text{simplicity}}(w_{f_1}) &= \|s_{t_1}\|_1 \\ \mathscr{L}_{\text{temporal coherence}}(w_{f_1}) &= \|\triangle s_{t_1}\|_2^2 \\ \mathscr{L}_{\text{proportionelity}}(w_{f_1}) &= (\|\triangle s_{t_2}\|_2 - \|\triangle s_{t_1}\|_2)^2 e^{-\|a_{t_2}-a_{t_1}\|_2} \\ \mathscr{L}_{\text{causality}}(w_{f_1}) &= e^{-\|s_{t_2}-s_{t_1}\|_2} e^{-\|a_{t_2}-a_{t_1}\|_2}(1-e^{-\|r_{t_2}-r_{t_1}\|_2}) \\ \mathscr{L}_{\text{repeatability}}(w_{f_1}) &= e^{-\|s_{t_2}-s_{t_1}\|_2} \|\triangle s_{t_2} - \triangle s_{t_1}\|_2^2 e^{-\|a_{t_2}-a_{t_1}\|_2} \end{aligned} \tag{3.1}$$

where s_t is used instead of $\Sigma(o_t)$ and $\triangle s_t$ instead of $s_{t+1} - s_t$ for better readability.

3.1.2. Model Learning method

The Model Learning method learns a prediction model of the system by training a model network. The model network learns a mapping from the observation-action tuple $\{o_t, a_t\}$ to the next state and reward $\{s_{t+1}, r_{t+1}\}$. In contrast to other Model Learning algorithms [12, 16, 27], the model is designed to predict the next state s_{t+1} instead of the next observation o_{t+1}. This becomes important if the observation o_t contains task-irrelevant information. A state that needs to be able to predict the next observation still has to contain this task-irrelevant information to make the prediction, whereas in the proposed case this information can be ignored altogether.

Network architecture

The network architecture, shown in Figure 3.2, consists of two Rectified Linear Unit (ReLU) layers followed by two linear output layers in parallel. The circles in the image represent a single layer containing multiple neurons, the lines are n-dimensional signals. The observations o are the input to the first layer, which outputs the state s. This state s, together with the action a form the input to the second layer. The output from the second layer is then used in both linear output layers to produce a prediction of the next state $\hat{s}_{t+1}$ and the reward $\hat{r}_{t+1}$ respectively.

The network architecture mimics the architecture of the critic network from the DDPG where the action also enters the network in the second layer. This allows for an easier integration of the two networks later on but there is also a more fundamental reason behind it. That is, in order to make a prediction of the next state s_{t+1} irrespective of the action taken in that state, the mapping from observations to states cannot contain an action as input. Furthermore, since there is no apriori reason to assume that the underlying system is input affine, a nonlinear layer is necessary after the action enters the network and before the linear output layers.

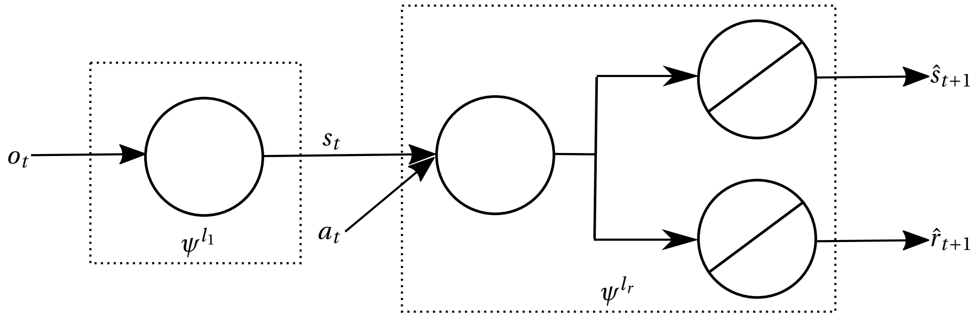


Figure 3.2: Network architecture of the DNN during training. The first layer takes an observation o_t as input and produces a state s_t which together with the actions a_t form the inputs to the second layer. The output of the network is a prediction of the next state $\hat{s}_{t+1}$ and the reward $\hat{r}_{t+1}$. The parameters of the network are ψ^{l_1} and ψ^{l^r} which represent the parameters in the first layer and the rest of the layers respectively.

Loss function

The DNN is trained to minimize the loss of the objective function given by

$$\mathscr{L}_{ML} = \|\hat{s}_{t+1} - s_{t+1}\|_2^2 + \lambda_{\text{ml}} \|\hat{r}_{t+1} - r_{t+1}\|_2^2$$

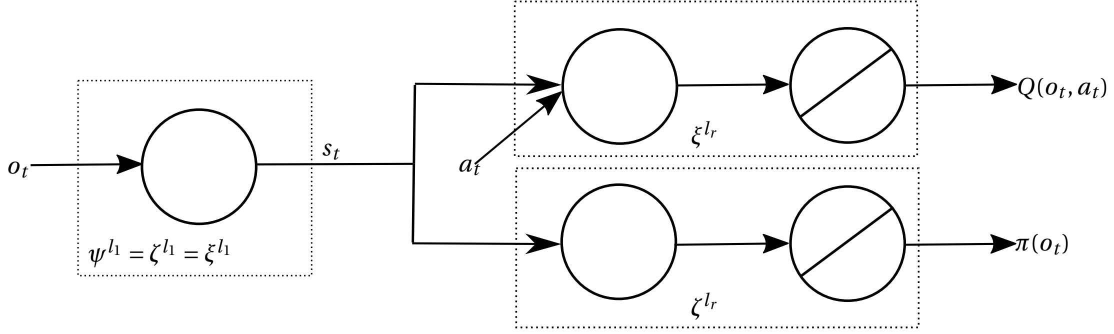


Figure 3.3: Integration of the model, actor and critic network.

where λ_{ml} is a scaling parameter which controls the relative weighting of the two objectives. Note that, to obtain s_{t+1}, the current approximation of the observation-to-state mapping is used, to map the next observation o_{t+1} to the next state s_{t+1}. This could potentially lead to convergence problems since the target depends on the current approximation. In practice, however, these problems did not occur. In all experiments, which all started from different random initial conditions, the learning converged to similar local optima.

3.2. Saturation penalty

One specific problem we encountered, with both the DDPG, the ML-DDPG and the RP-DDPG was the fact that the actor sometimes learned actions that lay outside the saturation limits of the actuator. This is caused in part because all the samples from which the agent learns are collected prior to the experiment. If an agent learns actions outside the range, in which data was originally collected. The policy gradient, evaluated at these actions, is based on extrapolating the critic network, which for large deviations is very unreliable. This creates instability issues in both networks which hamper the convergence of the algorithm.

In order to restrict the action space, a *saturation penalty* is added to the loss function of the actor. The loss function becomes

$$\mathcal{L}_a(\zeta) = -Q(s, \pi(s|\zeta)) + \lambda_{\text{sat}}\Big(\max(\pi(s|\zeta) - 5, 0) + \max(-\pi(s|\zeta) - 5, 0)\Big)^2$$

where λ_{sat} represents the trade-off between maximizing the reward and minimizing the saturation penalty. The actions are scaled such that they have zero mean and a standard deviation of 1, which puts the saturation limit at 5 times the standard deviation of the original exploration policy.

3.3. Integration with the DDPG algorithm

In the DDPG algorithm [22], the actor and critic network each consists of three layers. The main idea behind the two new algorithms presented in this section is that that up to a certain point, an actor and critic network, can benefit from sharing their intermediate representation. Furthermore, we argue that this intermediate representation can be learned more effectively, either by using the Robotic Prior method or by using Model Learning. We will refer to this intermediate representation as the state and the transformation of the input to this state, as an observation-to-state mapping.
The first layer of both networks is considered to represent the observation-to-state mapping. The parameter vectors ζ and ξ of the actor and critic respectively are therefore split into two parts where, ζ^{l_1} and ξ^{l_1} represent the weights of the first layer and ζ^{l_r} and ξ^{l_r} the weights of the remaining layers.
The following sections specify the two algorithms RP-DDPG and ML-DDPG that respectively use Robotic Priors and Model Learning to learn the observation-to-state mapping to determine ζ^{l_1} and ξ^{l_1}. Figure 3.3 shows how the observation-to-state mapping is integrated with both the actor and critic network in both algorithms. Table 3.1 shows the values of all the different parameters that were used, the parameters were kept the same across the different benchmarks.

3.3.1. RP-DDPG

As in the previous algorithm, the RP-DDPG learns the observation-to-state mapping before training the actor and critic network. The parameters of the observation-to-state mapping can be combined in the parameter vector ψ^{l_1}. In contrast to the ML-DDPG, there is no need to split the parameter vector since the network in the robotic prior method only consists of a single layer. After learning the observation-to-state mapping Σ the weights are integrated into the actor and critic network by setting

$$\zeta^{l_1} = \psi^{l_1}, \xi^{l_1} = \psi^{l_1} \tag{3.2}$$

and freezing these weights afterward. The full algorithm is given in Algorithm 2.

Algorithm 2 Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG)

{Robotic Prior Learning learning}
Randomly initialize network weights ψ^{l_1}, ζ and ξ
for pre-training step = 1 **to** M **do**
  Sample random mini-batch from DB
  Calculate $\mathscr{L}_{\text{RP}}$ over mini-batch
  $\psi^{l_1} \leftarrow \psi^{l_1} - \alpha \frac{\partial \mathscr{L}_{\text{RP}}}{\partial \psi^{l_1}}$
end for
{Actor-Critic Learning}
$\zeta^{l_1} \leftarrow \psi^{l_1}$ and $\xi^{l_1} \leftarrow \psi^{l_1}$ {copy weights to actor and critic}
$\zeta^- \leftarrow \zeta$ and $\xi^- \leftarrow \xi$ {set weights of target network}
for learning step = 1 **to** N **do**
  Sample random mini-batch from DB
  Calculate $\mathscr{L}_a$ and $\mathscr{L}_c$ over mini-batch
  $\zeta^{l_r} \leftarrow \zeta^{l_r} - \alpha \frac{\partial \mathscr{L}_a}{\partial \zeta^{l_r}}$ and $\xi^{l_r} \leftarrow \xi^{l_r} - \alpha \frac{\partial \mathscr{L}_c}{\partial \xi^{l_r}}$
  $\zeta^- \leftarrow \tau\zeta + (1-\tau)\zeta^-$ and $\xi^- \leftarrow \tau\xi + (1-\tau)\xi^-$ {update target network}
end for

3.3.2. ML-DDPG

In the ML-DDPG the model is trained before the actor and critic networks, using data collected from the system. The parameters of the model network can be combined in a parameter vector ψ. The vector ψ can subsequently be split into two parts where ψ^{l_1} represent the weights of the first layer and ψ^{l_r} the weights of the remaining layers. The observation-to-state mapping learned by the model network can then be integrated into the actor and critic network by setting

$$\zeta^{l_1} = \psi^{l_1}, \xi^{l_1} = \psi^{l_1} \tag{3.3}$$

and freezing these weights afterward. The full algorithm is given in Algorithm 3.

3.4. Conclusion

In this chapter two new algorithms are presented, the RP-DDPG and the ML-DDPG. In both algorithms, SRL is used to pretrain the first layer of the actor and critic networks of the DDPG. The algorithms differ in the specific SRL method that is used during this pretraining. The algorithms allow us to test the hypotheses that the DDPG can benefit from learning a shared observation-to-state mapping explicetly as opposed to learning this implicetly in both the actor and critic networks. Since normally it would learn such a mapping in both networks separatly, while training the rest of the network. It also allows us to see which of the two SRL methods is a more suitable method in this situation.

In addition, a saturation penalty is introduced that is added to the objective function of the agent in all algorithms. The saturation penalty prevents the agent from learning actions that are outside the range for which data was originally collected. In many cases, such large actions, lay outside the saturation limits of the actuator and would therefore be undesirable anyway.

Algorithm 3 Model Learning Deep Deterministic Policy Gradient (ML-DDPG)

{Model learning}
Randomly initialize network weights ψ, ζ and ξ
for pre-training step = 1 **to** M **do**
  Sample random mini-batch from DB
  Calculate $\mathscr{L}_{\text{ML}}$ over mini-batch
  $\psi \leftarrow \psi - \alpha \frac{\partial \mathscr{L}_{\text{ML}}}{\partial \psi}$
end for
{Actor-Critic Learning}
$\zeta^{l_1} \leftarrow \psi^{l_1}$ and $\xi^{l_1} \leftarrow \psi^{l_1}$ {copy weights to actor and critic}
$\zeta^- \leftarrow \zeta$ and $\xi^- \leftarrow \xi$ {set weights of target network}
for learning step = 1 **to** N **do**
  Sample random mini-batch from DB
  Calculate $\mathscr{L}_a$ and $\mathscr{L}_c$ over mini-batch
  $\zeta^{l_r} \leftarrow \zeta^{l_r} - \alpha \frac{\partial \mathscr{L}_a}{\partial \zeta^{l_r}}$ and $\xi^{l_r} \leftarrow \xi^{l_r} - \alpha \frac{\partial \mathscr{L}_c}{\partial \xi^{l_r}}$
  $\zeta^- \leftarrow \tau\zeta + (1-\tau)\zeta^-$ and $\xi^- \leftarrow \tau\xi + (1-\tau)\xi^-$ {update target network}
end for

Table 3.1: List of parameters for each algorithm

	DDPG	ML-DDPG	RP-DDPG
Actor Network			
Neurons in hidden layer 1	100	100	100
Neurons in hidden layer 2	100	100	100
Base learning rate	10^{-4}	10^{-4}	10^{-4}
λ_{sat}	50	50	50
Critic Network			
Neurons in hidden layer 1	100	100	100
Neurons in hidden layer 2	100	100	100
Base learning rate	10^{-3}	10^{-3}	10^{-3}
λ_{L2}	0.002	0.002	0.002
τ	10^{-3}	10^{-3}	10^{-3}
Other parameters			
γ	0.99	0.99	0.99
Batch size	200	200	200
λ_{ML}	-	10	-

4

Experiment design

In order to compare the ML-DDPG and the RP-DDPG with the original DDPG, each algorithm is applied to various benchmark problems. In each case, the task of the algorithm is to find a continuous control policy that maximizes the reward over time. The reward function is specified uniquely for each benchmark, but is always based on some distance measure between the current state of the system and some desired state.
Part of this thesis is to investigate how well DL methods deal with inputs that contain either task-irrelevant information or inputs that are constructed by extending a partially observable state with measurements taken at previous time instances. Both scenarios are often encountered in practical applications. A robot designed to do a wide variety of tasks, often receives far more information than necessary to do one particular task, i.e., in order to grab an object with one of its arms it can generally ignore the sensor measurements of its other arm. On the other hand, it is hard to directly measure the velocity of an object, therefore, either an observer needs to be used or the agent needs to learn based on a series of positional measurements. In these situations feature engineering is normally used, to extract from the inputs only those features pertinent to solving the task at hand [33], before using the inputs in the Machine Learning (ML) algorithm. In DL, however, this is claimed to be unneccessary [15]. In order to investigate this claim, for each benchmark, the input data is transformed into three different observation vectors known as:

- **Classical observation** $o^{\text{classical}}$: The classical observation $o^{\text{classical}}$ is equal to the internal state of the system, i.e., the state that is also used to simulate the dynamics.
- **Unrelated observation** $o^{\text{unrelated}}$: The unrelated observation $o^{\text{unrelated}}$ contains the internal state, extended with a vector of white noise inputs.
- **Redundant observation** $o^{\text{redundant}}$: The redundant observation $o^{\text{redundant}}$ contains a measurement of part of the internal state (without velocity information), extended with measurements at n previous timesteps and the actions taken at these timesteps. The observation is considered *redundant* since n is taken larger then necessary to give the observation vector the Markov property.

Each benchmark that is included, is chosen because of a particular reason. The pendulum is included because of it's simplicity and was used primarly to tune the algorithm. The 2-DOF arm is a more difficult extension of the pendulum. The octopus problem is chosen because of the high dimensionality of the state and action spaces.
Section 4.1 explains the general setup of an experiment. The other sections explain each of the benchmarks in more detail.

4.1. General setup

The experiments on the various benchmarks are all done in a similar fashion. The first step is to collect data (Section 4.1.1), by following some random exploration policy. The data is then preprocessed (Section 4.1.2) before it is made available to the different algorithms. Each algorithm then learns off-policy, from exactly the same data points as the other algorithms. Each algorithm runs for the same amount of iterations, where each iteration represents a single update of the actor and critic network (one learning step). During learning, the policy is evaluated every n learning steps (Section 4.1.3). In the following sections, each of these steps is explained in more detail.

Table 4.1: List of parameters that determine how data is collected, for each benchmark.

	Inverted Pendulum	2-link arm	Octopus
Sample time	2	2	1
β	0.4	0.4	0.01
σ	2	0.5	0.1

4.1.1. Data collection

Each experiment starts with the creation of an experience replay database. An experience replay database is a collection of n-samples, where each sample is created from a single interaction with the system. A sample consist of an observation o_t, an action a_t that is applied to the system and the next observation o_{t+1} and reward r_{t+1} that are received by the agent after the interaction. Together these form a tuple $\{o_t, a_t, r_{t+1}, o_{t+1}\}$ which we refer to as a sample. The action a_t is the result of a random Ohrnstein-Uhlenbeck process, which is given by

$$a_t = \beta a_{t-1} + \mathcal{N}(0, \sigma)$$

where β and σ are the parameters that determine the temporal correlation and magnitude of the actuation signal. The reason for using the Ohrnstein-Uhlenbeck process instead of random exploration is twofold. First, pure random exploration can cause problems in real actuators due to the high frequency content in the actuation signal. Second, an exploration signal that is temporally correlated explores a bigger part of the state space, since some parts of the state space can only be reached after a sustained temporal bias in the actuation signal, which would be highly unlikely in pure random exploration.

The experienced replay database is created by exploroning for several episodes of 1500 time steps each, starting from some initial state with $a_0 = 0$. Each benchmark requires tuning of the parameters β and σ to ensure that the system is actuated in a way that is within the saturation limits of the actuator and results in the exploration of the entire state space. Table 4.1 shows the values of β, σ and the sampling time that was used in each benchmark.

4.1.2. Data preprocessing

The samples collected from the system are preprocessed to create an experience replay database. The observation and action vectors are normalized to have a mean of zero and a standard deviation of 1. The mean and standard deviation are saved together with the data, such that actions from the policy network can be denormalized before applying them to the real system and observations from the system can be normalized before feeding them to the DNNs. The reward is normalised such that the average reward is -1 and the rewards are always negative. This is done to ensure that the Q-value, that is learned by the critic network, has an output in the same range across the different benchmarks.

After normalization, the samples are randomly assigned to a train, validation or test set with a probability of 0.8, 0.1 and 0.1 respectively. Within these sets the samples are randomly assigned to form mini-batches of 200 samples each.

4.1.3. Evaluation

In order to evaluate the learning during an experiment, the intermediate policy π is evaluated every 100 learning steps, using a pre-defined reference signal and initial state. The performance of the policy is defined as the undiscounted cumulative reward collected during the evaluation given by

$$R_j = \sum_{t=0}^{T} r(s_t, a_t) \tag{4.1}$$

where j is the number of learning steps and T the duration of the evaluation. Note that the state s_t, is the internal state of the system, which is not always available to the agent directly.

For every experiment the learning trials are repeated l times, to ensure that the results are statistically significant. In order to make a quantitative comparison between the learning curves, the settling time τ_s, rise time τ_{rise} and the average performance $\bar{R}$ is also calculated. To define the settling time and the rise time of the learning curve, first introduce the undiscounted return after j learning steps averaged over the number N_e of

learning experiments:

$$\bar{R}_j = \frac{1}{N_e} \sum_{l=1}^{N_e} \sum_{t=0}^{T} r(s_t, a_t)$$

where T is the duration of the evaluation, referring to the jth learning step within the lth learning experiment. For each reward, the sequence $\bar{R}_1, \bar{R}_2, \ldots, \bar{R}_{N_t}$ is normalized so that the minimum value of this sequence is -1.

The performance $\bar{R}_f$ at the end of learning is defined as the average normalized undiscounted return in the last c learning steps:

$$\bar{R}_f = \frac{1}{c} \sum_{j=N_t-c+1}^{N_t} \bar{R}_j$$

The settling time τ_s of the learning curve is then defined as the number of learning steps after which the learning curve enters and remains within a band ϵ of the final value $\bar{R}_f$:

$$\tau_s = T_t \cdot \underset{j}{\text{argmax}}(|\bar{R}_f - \bar{R}_j| \geq \epsilon \bar{R}_f)$$

In this thesis c and ϵ are set to 1000 and 0.05 respectively. The rise time is defined as the number of learning steps required to climb from the 10% performance level to the 90% performance level:

$$\tau_{\text{rise}} = \tau_{90} - \tau_{10}$$

with τ_p defined as:

$$\tau_p = T_t \cdot \underset{j}{\text{argmax}} \left(\frac{\bar{R}_j - \bar{R}_1}{\bar{R}_f - \bar{R}_1} \geq \frac{p}{100} \right)$$

for $p = 10\%$ and 90%.

4.2. Benchmark 1: Inverted pendulum

The inverted pendulum is often used as a benchmark problem to test and demonstrate new control algorithms on [12]. The task, in this case, is to track a certain reference angle θ^r. The main reason for including a relatively simple control problem is because of its simplicity. Its state and action spaces are low-dimensional, which makes it easy to visualise what is learned. Its primary use is, therefore, to get valuable insights while developing the various algorithms and observation-to-state mappings. Only once the algorithms are able to successfully solve this problem are they tested on more difficult benchmarks.
The following sections will give a detailed description of the setup, the reward function and the different observations that are made available to the learning agent.

4.2.1. Setup

A schematic diagram of the inverted pendulum is shown in Figure 4.1. The pendulum has a certain angle θ measured from the downward position and exists within the domain $[-\pi, \pi]$. The reference angle θ^r can vary within this same domain and is changed every n time steps. The pendulum is actuated by an electro-motor which can receive an input signal $u \in [-10, 10]V$. The equation of motion is given by

$$J\ddot{\theta} = Mgl\sin(\theta - \pi) - (b + \frac{K^2}{R})\dot{\theta} + \frac{K}{R}u \tag{4.2}$$

where the model parameters can be found in Table 4.2. Figure 4.1 also shows the x, y coordinates of the tip of the pendulum, measured relative to the suspension point. These coordinates are included because some of the learning agents will only have access to position and velocity measurements in the Cartesian coordinate system.

4.2.2. Reward function

The reward function that is used consists of three elements, the distance between the current angle θ and the reference angle θ^r, the angular velocity $\dot{\theta}$ and a third term that is -1 far from the reference and goes exponentially to zero closer to it. The latter was used to create an extra steep increase in the reward close to the reference. The function is given by

$$R(\theta, \dot{\theta}, \theta^r) = -(\theta - \theta^r)^2 - w_1(1 - e^{b(\theta - \theta^r)^2}) - w_2\dot{\theta}^2$$

where $w_1 = 8$, $b = 10$ and $w_2 = 0.02$.

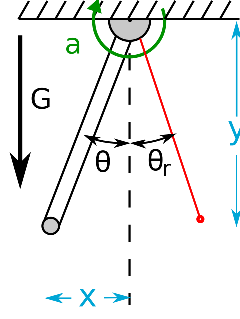


Figure 4.1: A schematic diagram of the inverted pendulum

Table 4.2: Model parameters of the inverted pendulum.

Name	Value
J	$1.91e-4$
M	$5.5e-2$
g	9.81
l	$4.2e-2$
b	$3e-6$
K	$5.36e-2$
R	9.5

4.2.3. Input design

The classical observation $o^{\text{classical}}$ consist of the internal state of the system extended with the reference position and is defined as

$$o^{\text{classical}}(t) = [\theta_t \quad \dot{\theta}_t \quad \theta^r_t]$$

where t is the timestep.
The second observation $o^{\text{unrelated}}$ uses the Cartesian coordinate system to denote the the position and velocity of the pendulum in, where as the reference position is given in angular coordinates. The DNN therefore needs to learn which coordinates correspond to which angles, while learning the task. Furthermore a vector of random white noise signals e with $E[e] = 0$ is included in the observation, to mimic inputs that are uncorrelated with the task at hand. The size of the vector is equal to the size of the other inputs combined, such that half of the inputs are unrelated to the task. The full observation vector $o^{\text{unrelated}}$ is defined as

$$o^{\text{unrelated}}(t) = [x_t \quad y_t \quad \dot{x}_t \quad \dot{y}_t \quad e_t \quad \theta^r_t]$$

where x and y are the coordinates of the tip of the pendulum as shown in Figure 4.1.
The third observation $o^{\text{redundant}}$ contains the reference angle θ^r, the current angle θ and the angle and action from the previous 5 timesteps and is given by

$$o^{\text{redundant}}(t) = [\theta_t \quad .. \quad \theta_{t-5} \quad a_{t-1} \quad .. \quad a_{t-5} \quad \theta^r_t].$$

4.3. Benchmark 2: 2-link Arm

The 2-link arm benchmark [6] is a simple robotic arm with 2 Degrees of Freedom (DOF). The task is to reach, with the tip of the second link to a certain reference position. Since there are multiple configurations that position the tip of the second link at the reference position, part of the task is to find the fastest path towards the reference position.
The following sections will give a detailed description of the setup, the reward function and the different observations that are made available to the learning agent.

4.3.1. Setup

A schematic diagram of the 2-link arm is shown in Figure 4.2. The 2-link arm consists of two links, both of which can be controlled by a motorized joint. The angle of the first link $\theta_1 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ is measured with respect

to the downward position and the angle of the second link $\theta_2 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ with respect to the first link. The two motorized joints can be controlled by setting a_1 and a_2, which are (a scaled version of) the motor voltages. Finally the Cartesian reference position $p^{\text{ref}} = \begin{bmatrix} x^{\text{ref}} & y^{\text{ref}} \end{bmatrix}$ determines the desired position of the tip of the second link.
Instead of using the equations of motion, a 3D-model of the system is created and simulated using the Gazebo simulator.

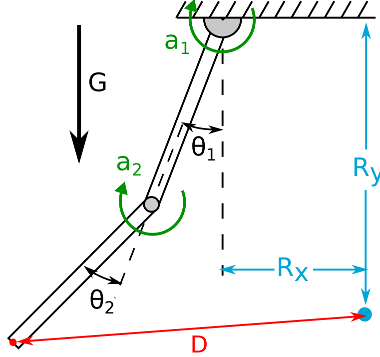


Figure 4.2: A schematic diagram of the 2-link arm

4.3.2. Reward function

The reward function penalizes the distance D, between the current position of the tip of the second link and the reference position, and the angular velocity $\dot{\theta}$ of both links. The reward function is given by

$$r(D, \dot{\theta}) = -\left(D + w|\dot{\theta}|_2\right)$$

where $w = 0.1$ is the parameter that weight the different terms.

4.3.3. Input design

The classical observation $o^{\text{classical}}$ consist of the angles $\theta = [\theta_1, \theta_2]$, the angular velocity $\dot{\theta} = [\dot{\theta}_1, \dot{\theta}_2]$ and the reference position p^{ref} of the tip of the second link as shown in Figure 4.2 and is defined as

$$o^{classical}(t) = \begin{bmatrix} \theta_t & \dot{\theta}_t & p_t^{\text{ref}} \end{bmatrix}$$

where t is the time step.
The second observation $o^{\text{unrelated}}$ uses the Cartesian coordinate system to denote the position and the velocity of the 2 links. These variables are given by

$$\begin{aligned} x &= [\sin(\theta_1), \sin(\theta_1) + \sin(\theta_1 + \theta_2)] \\ y &= [\cos(\theta_1), \cos(\theta_1) + \cos(\theta_1 + \theta_2)] \\ \dot{x} &= [\sin(\theta_1)\dot{\theta}_1, \sin(\theta_1 + \theta_2)\dot{\theta}_2] \\ \dot{y} &= [\cos(\theta_1)\dot{\theta}_1, \cos(\theta_1 + \theta_2)\dot{\theta}_2] \end{aligned} \tag{4.3}$$

As in the previous benchmark a vector of random white noise signals e with $E[e] = 0$ is also included in the observation. The full observation vector $o^{\text{unrelated}}$ is defined as

$$o^{\text{unrelated}}(t) = \begin{bmatrix} x_t & y_t & \dot{x}_t & \dot{y}_t & e_t & p_t^{\text{ref}} \end{bmatrix}.$$

The third observation $o^{\text{redundant}}$ contains the reference position r, the current angles θ and the angles and actions from the previous 5 time steps and is given by

$$o^{\text{redundant}}(t) = \begin{bmatrix} \theta(t) & .. & \theta(t-5) & a_{t-1} & .. & a_{t-5} & p^{\text{ref}}(t) \end{bmatrix}$$

where $a = [a_1, a_2]$.

4.4. Benchmark 3: Octopus

The octopus benchmark is based on a simulation of an octopus arm, where the aim is to hit a food target with any part of the body. A screen shot of the simulator is shown in Figure 4.3 where the red dot represents the food target. The octopus arm is considered to be a very challenging task for a Reinforcement Learning (RL) algorithm, because of the high dimensionality of the state and action spaces.
In order to reduce the complexity of the task, some have used a few simplifications. In [8] the high-dimensional action space was simplified by defining 6 macro-actions that correspond to particular patterns of muscle activation. The task of the agent was then reduced to choosing when to use which macro-action. A less drastic simplification was used in [13] where the muscle activations were restricted to be of a boolean value, which is a way of discretizing the action space. Others [13, 32] have reduced the dimensionality of both the state and action space by limiting the number of segments to 6.
In this thesis none of these simplification are used, the only thing that was added to the original environment was to add a distance measure to the reward function, which was also done in [13, 32]. The following sections will give a detailed description of the setup, the reward function and the different observations that are made available to the learning agent.

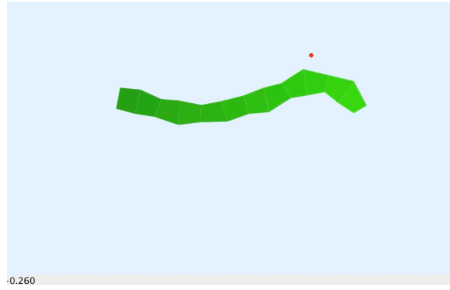


Figure 4.3: A screenshot of the octopus simulator

4.4.1. Setup

The octopus arm consists of 13 segments, attached to a base at one side and moves in a 2-dimensional plane. The state of each segment is defined by the position and velocity of the top and bottom in the x, y coordinate system and the angle and angular velocity with respect to the base. Since the segments are attached to each other the total state vector consists of $8(C-1)+2$ values, where C are the number of segments.
Each segment has 3 muscles (dorsal, transversal, central). The muscles are controlled by specifying its stiffness $a \in [0, 1]$. The food target is placed somewhere in the environment, each episode the arm starts with the same initial conditions.

4.4.2. Reward function

The reward from the environment is based on the distance D between the food and the segment that is closest to the food. Whenever the goal is reached an extra bonus B is given. The reward function is given by

$$r(D, B) = (B-2) - D \tag{4.4}$$

where $B = 2$ whenever the goal is reached and $B = 0$ otherwise.

4.4.3. Input design

The classical observation $o^{\text{classical}}$ consist of the full state s^{full} as specified by the simulator and explained in Section 4.4.1 and is defined as

$$o^{\text{classical}}(t) = s_t^{\text{full}}$$

The observation vector $o^{\text{unrelated}}$ contains the full state s^{full} as specified by the simulator and a vector of unrelated white noise inputs e_t with the same size as the state vector

$$o^{\text{unrelated}}(t) = \begin{bmatrix} s_t^{\text{full}} & e_t \end{bmatrix}.$$

Table 4.3: Dimension table for the three benchmarks

	Action	$o^{\text{classical}}$	$o^{\text{unrelated}}$	$o^{\text{redundant}}$
Inverted Pendulum 2	3	10	12	
2-link arm	2	6	18	24
Octopus	36	96	192	308

The observation vector $o^{\text{redundant}}$ contains the positional state information s^{position} from the state as specified by the simulator and this state s^{position} and the action a at the three previous time instances and is given by

$$o^{\text{redundant}}(t) = \begin{bmatrix} s_t^{\text{position}} & .. & s_{t-3}^{\text{position}} & a_{t-1} & .. & a_{t-3} \end{bmatrix}.$$

4.5. Conclusion

In this chapter, the experimental setup is explained and three different benchmark problems are defined. For each of the benchmarks three observation vectors are defined, which simulate different conditions under which the learning algorithm should still function. This allows us to compare the sensitiveness of the algorithms to the way the agent can observe the system. Table 4.3 shows the dimensions of the state and action spaces for each of the benchmarks. It ranges from low, a 2 dimensional action space and 3 dimensional state space for the Inverted pendulum, to very high, a 36 dimensional action space and a 308 dimensional state space for the Octopus.

5

State Representation Learning - Experiments

In this chapter the two different State Representation Learning (SRL) methods that were introduced in Chapter 3 are compared on the pendulum benchmark problem from Chapter 4.2. First in Section 5.1 the training curves of both the Robotic Prior method and the Model Learning method are shown. Subsequently in Section 5.2 the state representations that are learned are visualised using t-Distributed Stochastic Neighbor Embedding (t-SNE) and compared to an untrained observation-to-state mapping and the state representations that is learned in the actor network that does not explicitly use SRL.

5.1. Learning the state representation

In Chapter 3 two methods were introduced that learn an observation-to-state mapping by training a Deep Neural Network (DNN), the the Robotic Prior method and the Model Learning method. Both learn from samples collected from the system, where a sample consists of $\{o_t, a_t, r_{t+1}, o_{t+1}\}$ where $o \in \mathbb{R}^n$ is a vector of observations, $a \in \mathbb{R}^m$ a vector of control actions and $r \in \mathbb{R}$ is a scalar reward. A collection of samples is used for training the DNN, a different set of samples is used for validating the results as explained in Section 4.1. The following sections present the results for each of the two methods.

5.1.1. Robotic Prior method

In Chapter 4 three different types of observation vectors are defined, that differ in the way information about the state of the system is provided to the DNN. Figure 5.1 shows the mean squared error on the validation set during training for each of these observation vectors. The experiment is repeated 5 times where each time, the weights of the DNN are randomly initialized. The plot shows the mean (thick line) and the 95% confidence region (shaded area) for each type of observation. The confidence region is quite small, indicating the network converges to similar local minima, each time the algorithm is run.
In general, the Robotic Prior method converges much faster (within 10 iterations), than the Model Learning method (Figure 5.2). One reason that could explain the faster convergence is the fact that the Robotic Prior network contains far less parameters, since it does not contain a second and third layer, like the Model Learning network. Another reason could be that the objectives formulated by the Robotic Prior method are just easier to satisfy.

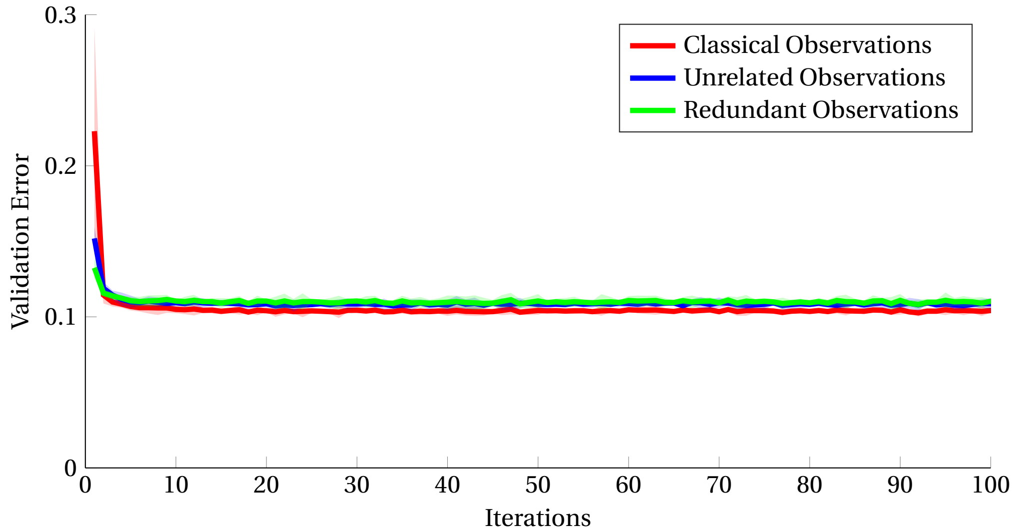


Figure 5.1: Plot of the mean squared error on the validation set during training when learning the mapping from observations to states for the inverted pendulum problem using the Robotic Prior method from Section 3.1.2. The thick line is the mean over 5 experiments, the shaded area shows the 95% confidence region. Each observation set originates from the same dataset.

5.1.2. Model Learning method

The experiment is repeated for the Model Learning method. Figure 5.2 shows the mean squared error on the validation set during training for each of the observation vectors. Again the algorithm has the worst performance on the $o^{\text{redundant}}$ observation type. The algorithm reaches the lowest error for the $o^{\text{classical}}$ and $o^{\text{unrelated}}$ observation types. It is interesting to see that the algorithm converges faster in case of the $o^{\text{unrelated}}$ observation type since in that case it has to learn to ignore the unrelated white noise signals. The algorithm has the largest error on the $o^{\text{redundant}}$ observation type. This is also arguably the most difficult observation type since the network has to learn how to extract velocity information from a sequence of positions. A direct comparison, between the performance reached by the Robotic Prior method and the Model Learning method, is impossible. Both mappings are optimized using very different objective functions and, as stated before, the “optimal” state representation to solve the reinforcement task is unknown.

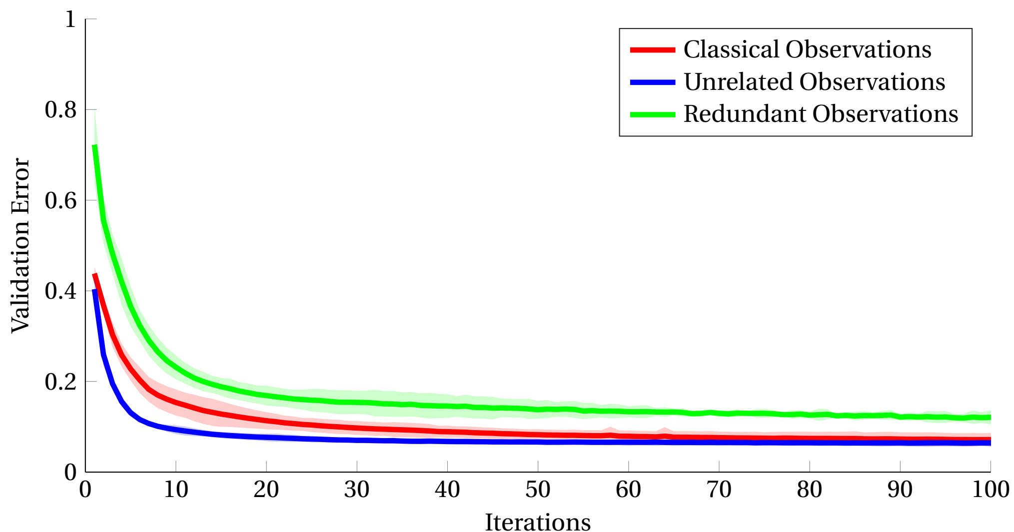
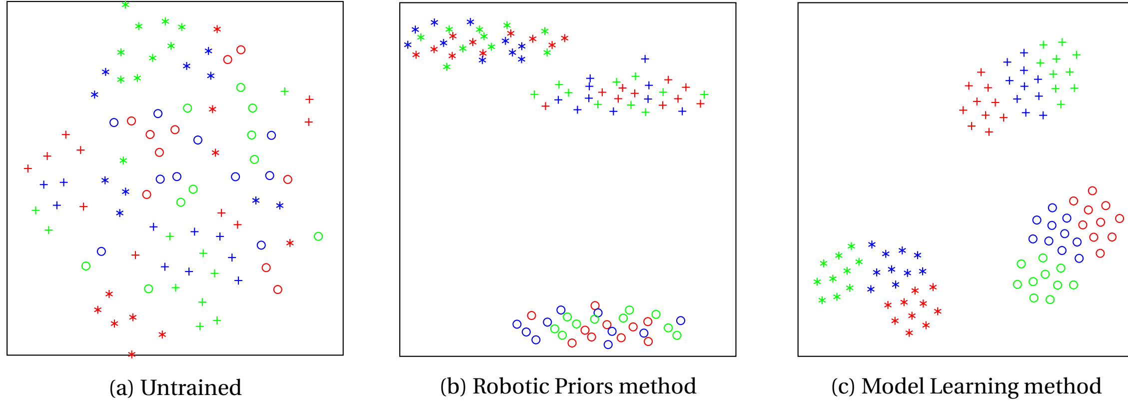


Figure 5.2: Plot of the mean squared error on the validation set during training when learning the mapping from observations to states for the inverted pendulum problem using the model learning method from Section 3.1.2. The thick line is the mean over 5 experiments, the shaded area shows the 95% confidence region. Each observation set originates from the same dataset.



(a) Untrained (b) Robotic Priors method (c) Model Learning method

Figure 5.3: Each figure is a t-SNE plot that displays a 100-dimensional state representations as points on a 2D-map. The distance between two points represents a difference in representation. Each observation ($o^{\text{unrelated}}$) is labelled such that similar symbols correspond to similar angles θ and similar colours correspond to a similar angular velocity $\dot{\theta}$, the reference angle θ_r is kept constant for all observations.

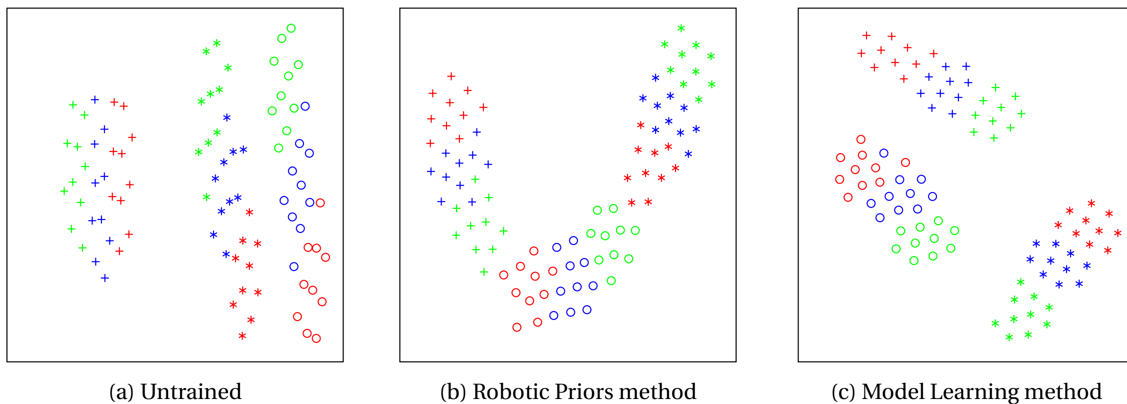
5.2. Visualising the learned representations

It is in principal hard to measure the usefulness of the observation-to-state mapping Σ that is learned by either two methods, let alone compare the two. In this section t-SNE, as explained in Chapter 2.1.3, is used as a visualisation method, that allows us to plot the 100-dimensional state representation that is learned, on a 2-dimensional map. The assumption behind the approach is that a good state representation (from the perspective of a controller), is a representation in which it is easy to discriminate between states, that differ from a physical perspective. For an inverted pendulum, for which the equations of motion are known, the physical state can be described by the angle and the angular velocity. A set of 90 observations is therefore labelled based on there physical state, where each observation has two labels, one for the angle and one for the angular velocity. The observations are subsequently mapped to states, using the observation-to-state map learned by the Model Learning method and the Robotic Prior method. The idea is that in a good state representation clusters should be visible which would allow easy discrimination between different angles and/or angular velocities.

Figure 5.3 shows the visualisation of 90 state representations for three different observation-to-state mappings Σ, each maps $o^{\text{unrelated}}$ observations to their respective state representation. In the figure, similar symbols correspond to similar angles θ and similar colours correspond to a similar angular velocity $\dot{\theta}$. In Figure 5.3a, an untrained network is used in which the weights were randomly initialized. This is included as a reference, to see how the state representation would look on an untrained network. It is clear that no clusters are visible. In Figure 5.3b, the Robotic Prior method was used to learn Σ. The data points are segregated in three distinct clusters that share a similar symbol. The Robotic Prior method allows us to easily discriminate between state representations that belong to different angles. Within each cluster, however, there is no structure. It is, therefore, impossible to discriminate, for each angle, between state representations that belong to different angular velocities. In Figure 5.3c, the Model Learning method was used to learn Σ. As in the second image, three clusters appear that discriminate between different angles. In addition within each cluster, the representations are grouped together according to the angular velocity. Based on this visualisation it looks like the Model Learning method learns a state representation where a different state representation corresponds to a different physical state and a similar state representation corresponds to a similar physical state.

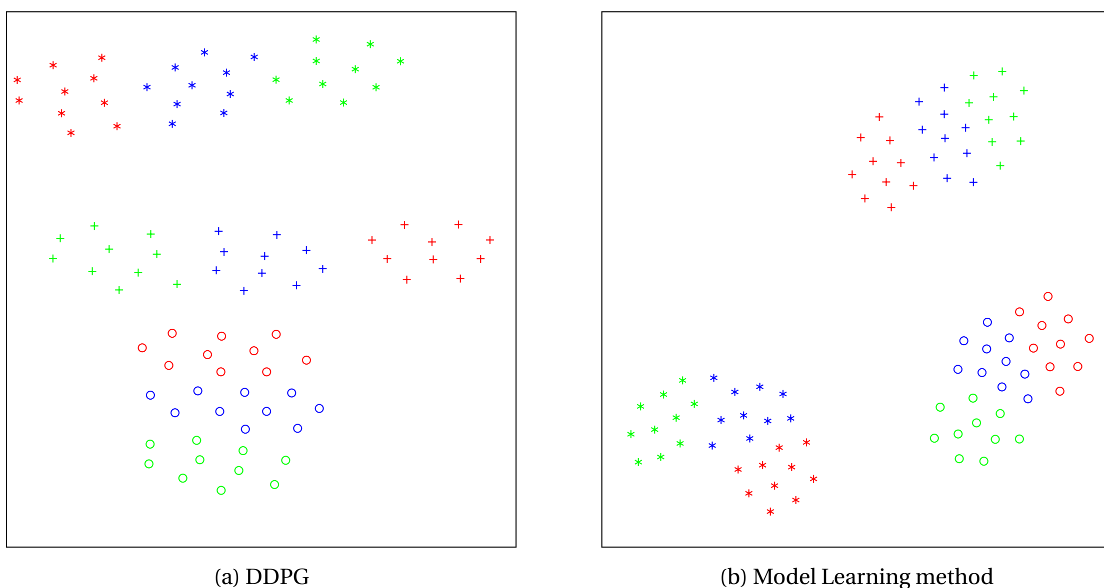
In Figure 5.4 the same approach is taken for the $o^{\text{redundant}}$ observation vector. In this case, the state representations from the untrained network (5.4a) already show three clusters (one for each angle). This is because the observation vector does not contain white noise inputs, like in the $o^{\text{unrelated}}$ observation vector, which causes the observation vector to have a much stronger correlation with the label. It is now easier to discriminate between different angles and angular velocities for the state representations of the Robotic Prior method. The distance between representations of different angles is, however, smaller then in the case of the Model Learning method.

In order to see how the state representation learned by the Model Learning method compares to the observation-to-state map that is implicitly learned in the actor network of the Deep Deterministic Policy Gradient (DDPG), another t-SNE plot is created. For the DDPG the first layer of the actor is taken as the



(a) Untrained (b) Robotic Priors method (c) Model Learning method

Figure 5.4: Each figure is a t-SNE plot that displays a 100-dimensional state representations as points on a 2D-map. The distance between two points represents a difference in representation. Each observation ($o^{\text{redundant}}$) is labeled such that similar symbols correspond to similar angles θ and similar colours correspond to a similar angular velocity $\dot{\theta}$, the reference angle θ_r is kept constant for all observations.



(a) DDPG (b) Model Learning method

Figure 5.5: Comparison of a t-SNE plot of the state representation learned implicitly by the the DDPG and explicetly using the Model Learning method. In each case the $o^{\text{unrelated}}$ observation vector is used.

observation-to-state mapping Σ. The results for the $o^{\text{unrelated}}$ observation vector are shown in Figure 5.5. The two representations are quite similar, suggesting the DDPG implicitly learns a state representation that allows it to discriminate between different angles and angular velocities, in the first layer of the network.

5.3. Conclusion

In this chapter t-SNE is used to visualise the high dimensional state representations learned by different networks. t-SNE allows us to visualize and compare different SRL methods. If the results are compared to an untrained method, that was initialized randomly, it becomes clear that each method is able to improve, i.e., it learns a representation for which it is easier to discriminate between states that differ from a physical perspective. For both the $o^{\text{unrelated}}$ and $o^{\text{redundant}}$ observation vectors, the Model Learning method does a better job then the Robotic Prior method. If the Model Learning method is compared to the observation-to-state mapping Σ, learned implicitly in the first layer of the DDPG, the results are quite similar. The different visualisations of the state representation that are learned do, however, not directly prove the usefulness of the observation-to-state mapping. In the following chapters the different algorithms, that differ in the way the observation-to-state mapping is learned, will be compared. Only then can we see of these results correlate with the visualisation made here.

6

Policy learning - Experiments

In this chapter, the two algorithms introduced in Chapter 3 are compared to each other and the original Deep Deterministic Policy Gradient (DDPG). Section 6.1 presents the results of the Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG), this algorithm is discussed separately because initial results showed a very poor performance. The results of the Model Learning Deep Deterministic Policy Gradient (ML-DDPG) and the DDPG are compared in Section 6.2. Each of the two algorithms is run on each of the three benchmarks presented in Chapter 4 for different sizes of the experience replay database. Section 6.3 concludes this chapter by evaluating the hypothesis stated in the introduction chapter.

6.1. The Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG)

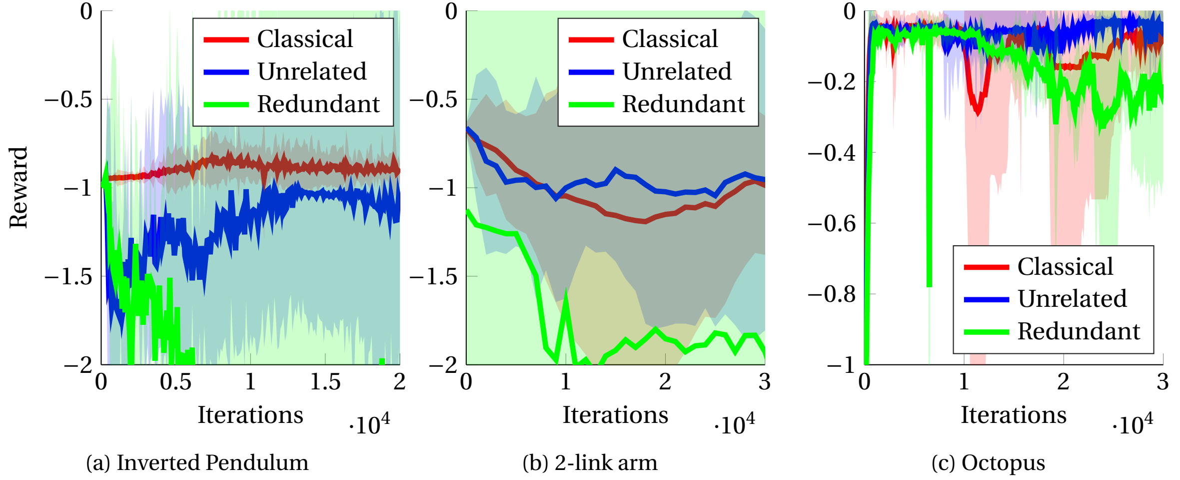
In Chapter 5.2 visualizations of the state representation, learned using the Robotic Priors method, were worse than the Model Learning method. The real test, however, is to see if the actor and critic networks are able to learn, using the observation-to-state mapping Σ learned using the Robotic Priors.

Figure 6.1 shows the result on each of the benchmarks, for the different observation vectors defined in Chapter 4. For the Inverted Pendulum benchmark and the 2-link arm, the results are very poor. In both cases, the algorithm never reached a performance that is significantly better than the performance at the start of the learning trial, when both the actor and critic networks are initialized randomly. The results on the octopus, however, are quite good. It learns to hit the target in less than 1000 learning steps. This performance, however, required a fairly large experiment replay database that contained $60K$ samples. When the number of samples was decreased, the performance also degraded significantly. In order to see why the Robotic Prior method fails on the Inverted Pendulum and the 2-link arm benchmark, we compare our setup to the setup used in [17]. First of all, in [17] the observations consisted of a vector of pixel values instead of positional and velocity measurements and the action space is discretized as opposed to our setup in which the action space is continuous. The most important difference, however, are the benchmark problems on which the algorithm is tested. In [17] the algorithm is tested on a navigation task and a slot-car racetrack, in both cases, the physical state of the system that needs to be extracted to control the system consists only of positional information, e.g., the position in the room and the position of the car on the track. This is confirmed in [17], where it is shown that in both cases the dimension of the state is 2, representing the x and y coordinates.

The benchmark problems used in this thesis, particularly the Inverted Pendulum and the 2-link arm, also requires velocity information to be extracted from the observation. One specific Robotic Priors does not seem particularly suitable to extract both positional and velocity information. For instance, the *proportionality prior* states that the amount of change in the state representation, as a result of the action taken in that state, should be proportional to the action that was taken in that state. In a control setup, that is sampled with a high frequency, the change in position is more related to the current velocity then the control action. This prior would therefore not hold. It seems probable that this explains the poor performance on the Inverted Pendulum and the 2-link arm for the RP-DDPG.

6.2. The Model Learning Deep Deterministic Policy Gradient (ML-DDPG)

The performance of the ML-DDPG is compared to the DDPG on three benchmark problems and for each of the three observations vectors from Chapter 4. In each of these settings, the experiments are repeated



(a) Inverted Pendulum (b) 2-link arm (c) Octopus

Figure 6.1: Learning curve of the RP-DDPG on the three different benchmarks. Each plot shows the average cumulative reward over 5 learning trials (thick line) and the 95% uncertainty region (shaded area) for each of the different observation vectors.

Table 6.1: Learning curve characteristics on the Inverted Pendulum for different sizes of the experience replay database. The rise time τ_{rise} and settling time τ_s are denoted in ×1000 learning steps.

Input type	DB size	DDPG			ML-DDPG		
		τ_{rise}	τ_s	$\bar{R}$	τ_{rise}	τ_s	$\bar{R}$
$o^{\text{classical}}$	7.5K	3.9	11.7	−0.371	**3.3**	**5.5**	**−0.253**
	30K	11.2	13.7	**−0.233**	**5.2**	**7.8**	−0.262
	120K	**9.2**	**11.1**	**−0.231**	10.5	12.1	−0.293
$o^{\text{unrelated}}$	7.5K	**6.7**	**7**	**−0.429**	6.8	16.5	−0.445
	30K	10.7	13.5	**−0.246**	**8.5**	**9.4**	−0.297
	120K	12.3	16.2	−0.285	**9.7**	**11.4**	**−0.235**
$o^{\text{redundant}}$	7.5K	**0**	16.8	−0.985	17.9	**15.5**	**−0.928**
	30K	14.8	**15.4**	−0.500	**13.2**	16.8	**−0.412**
	120K	**14.5**	16.2	**−0.344**	15.3	**16.1**	−0.394

for different sizes of the experience replay database, to investigate how the algorithm performs when data is either scarce or abundant. In order to make a quantitative comparison between the learning curves, the settling time τ_s, rise time τ_{rise} and the average performance $\bar{R}$ are calculated, see Section 4.1.3.

6.2.1. Benchmark 1: Inverted Pendulum

Table 6.1 shows the results for the different observation vectors on the Inverted Pendulum problem. There is no clear overall winner. The ML-DDPG outperforms the DDPG on the classical and redundant observation set when the size of the experience is small while the DDPG performs better on these observation sets for larger sizes of the experience database. For the unrelated dataset, the result is the reverse. In general, the performance becomes better if the size of the database grows, although even here one can find exceptions.

In order to take a closer look at the final control policy that is learned by the actor, a time-domain plot of the angle and control action can be made. The reference signal is the same as the one that is used when testing the performance of the algorithm. From the 5 trials that are available, the actor with the median performance is selected.

Figure 6.2 shows the time-domain plot for the $o^{\text{unrelated}}$ observation vector. Two things stand out. First, for three out of five reference positions, the noise on the control action (probably caused by the white noise inputs) is significantly lower with the ML-DDPG then for the DDPG. It seems, at least in certain areas, that learning a model helped the network to ignore these unrelated inputs. It is not clear why it failed to learn this specifically for the fourth and fifth reference position. When looking at a time-domain plot of the control action for one of the other trials, this did not happen. The second thing that stands out is the relatively large

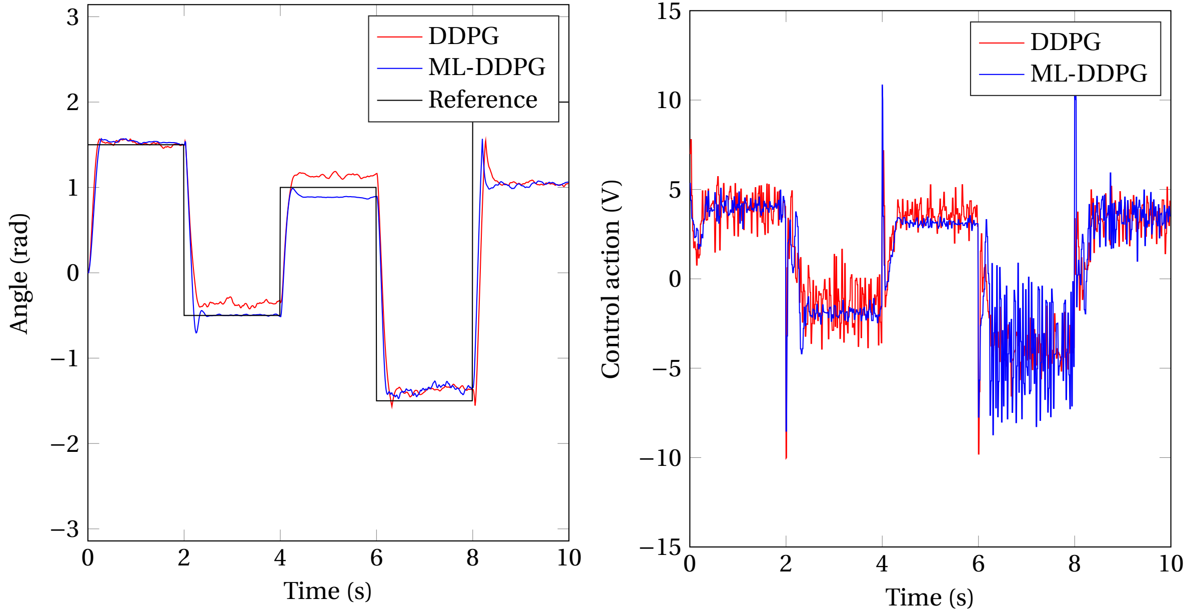


Figure 6.2: Time domain plot, showing the state and control action, of the final policy learned by the ML-DDPG and DDPG on the Inverted Pendulum problem using the $o^{\text{unrelated}}$ observation vector. Both algorithms are trained using $120K$ samples.

steady state error on the last reference position. This is caused by the fact that the position of the pendulum in the observation vector is given in the x and y coordinates, whereas the reference is given in angles. The pendulum goes to the right x-position but not the right y-position.

Figure 6.3 shows time-domain plot for the $o^{\text{redundant}}$ observation vector. The plot shows that the DDPG is able to learn a more stable response then the ML-DDPG. It seems the DDPG is better in extracting the velocity information from the observation vector, which is necessary to prevent the type of oscillations the ML-DDPG is suffering from.

6.2.2. Benchmark 2: 2-link arm

Figure 6.4 shows the mean (thick line) and standard deviation (shaded area) of the learning curve for the $o^{\text{unrelated}}$ and $o^{\text{redundant}}$ observation vectors using an experience replay database of 30K samples. The results from the other experiments are presented in Table 6.2. For the 2-link arm benchmark, the ML-DDPG outperforms the DDPG algorithm in final performance $\bar{R}$ (+38.0% on the $o^{\text{classical}}$ +37.8% on $o^{\text{unrelated}}$ and +8.1% on $o^{\text{redundant}}$) and in rise time τ_{rise} −29% and settling time τ_s −36.2% on the $o^{\text{unrelated}}$ observation type. It does have slower convergence on the $o^{\text{redundant}}$ type (rise time τ_{rise} +28.7% and settling time τ_s +39%). Both algorithms perform better, in terms of final performance, if more data is available. The advantage of the ML-DDPG over the DDPG seems relatively constant and not, as was expected, degrade when data becomes abundant.

Figure 6.5 shows a time-domain plot of the x-coordinate of the tip of the second link and (one of the) accompanying control actions when learning from the $o^{\text{redundant}}$ observation vector. It is clear that the performance is incomparable to other optimal control methods, the controlled system has a steady-state error and a significant overshoot. It is important to note, however, that the controller does not use a separate observer, although the system is partially observable and the reference is given in Cartesian coordinates whereas the position of the 2 links is given in joint angles. The controller, therefore, needs to learn the non-linear mapping between the two, while also learning the unobservable states from a sequence of measurements. Seen in this light, we think the performance is actually quite good. We also believe the performance can be further increased by tuning the reward function and/or architecture of the Deep Neural Networks (DNNs) which has not been done extensively to get these results.

Figure 6.6 shows a time-domain plot of the state and control action, of the final policy learned by the ML-DDPG and DDPG on the 2-link arm problem using the $o^{\text{classical}}$ observation vector. Figure 6.6 shows the x-coordinate of the tip of the second link. For the first two reference positions, the steady state error of the

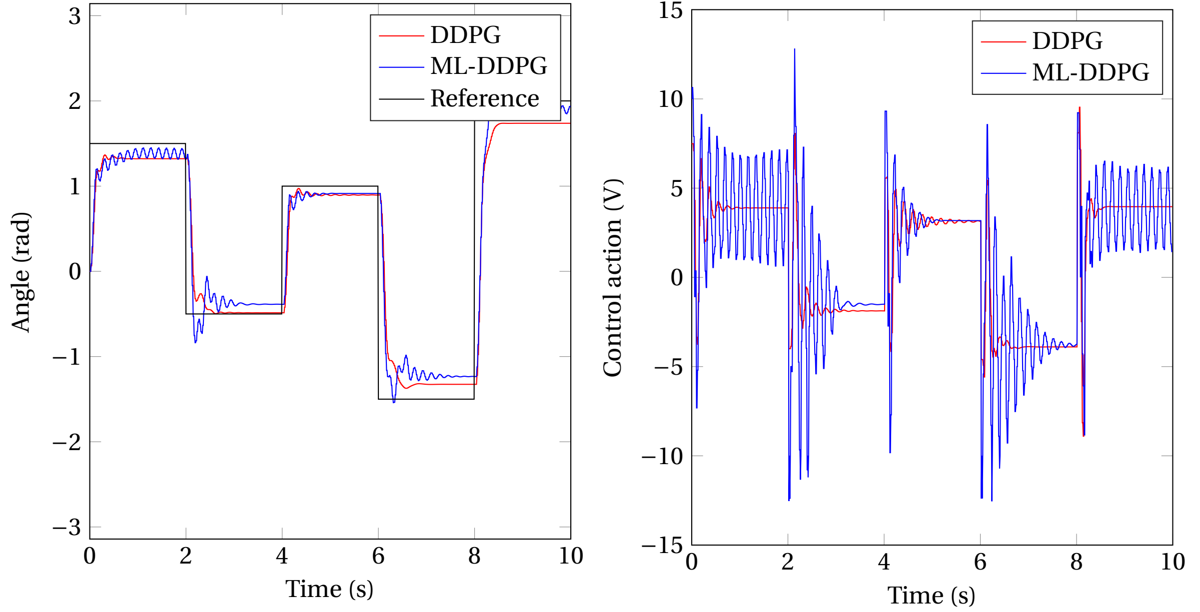
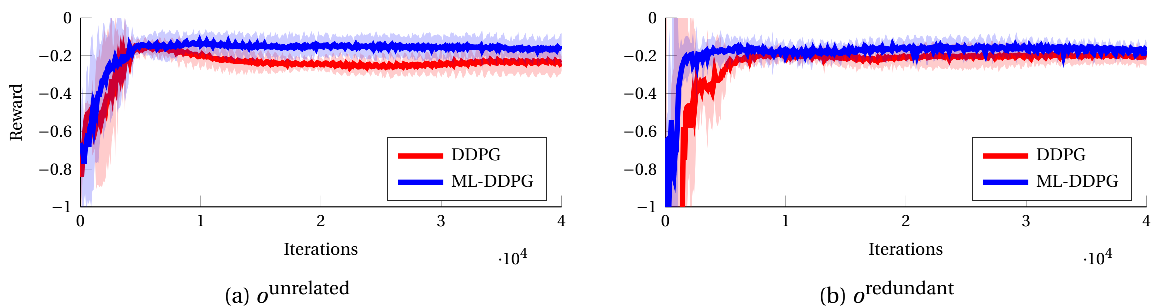


Figure 6.3: Time domain plot, showing the state and control action, of the final policy learned by the ML-DDPG and DDPG on the Inverted Pendulum problem using the $o^{\text{redundant}}$ observation vector. Both algorithms are trained using $120K$ samples.

Table 6.2: Learning curve characteristics on the 2-link arm for different sizes of the experience replay database. The rise time τ_{rise} and settling time τ_s are denoted in ×1000 learning steps.

Input type	DB size	DDPG			ML-DDPG		
		τ_{rise}	τ_s	$\bar{R}$	τ_{rise}	τ_s	$\bar{R}$
$o^{\text{classical}}$	15K	2.4	11.9	−0.251	**0.9**	**2.1**	**−0.167**
	30K	**2**	6	−0.190	2.1	**2.3**	**−0.116**
	90K	13.5	**2.9**	−0.139	**2.5**	3.4	**−0.127**
$o^{\text{unrelated}}$	15K	3	**3.2**	−0.222	**2**	3.5	**−0.180**
	30K	3	8.6	−0.235	**2.8**	**3.8**	**−0.164**
	90K	3.4	3.7	−0.191	**1.9**	**2.6**	**−0.126**
$o^{\text{redundant}}$	15K	**2.5**	**11.6**	**−0.256**	3.7	17.2	−0.276
	30K	4.8	5.1	−0.205	**1.9**	**2.5**	**−0.175**
	90K	**5.9**	**6.2**	−0.187	11.3	12.2	**−0.148**



(a) $o^{\text{unrelated}}$ (b) $o^{\text{redundant}}$

Figure 6.4: Learning curve 2-link arm using 60K samples

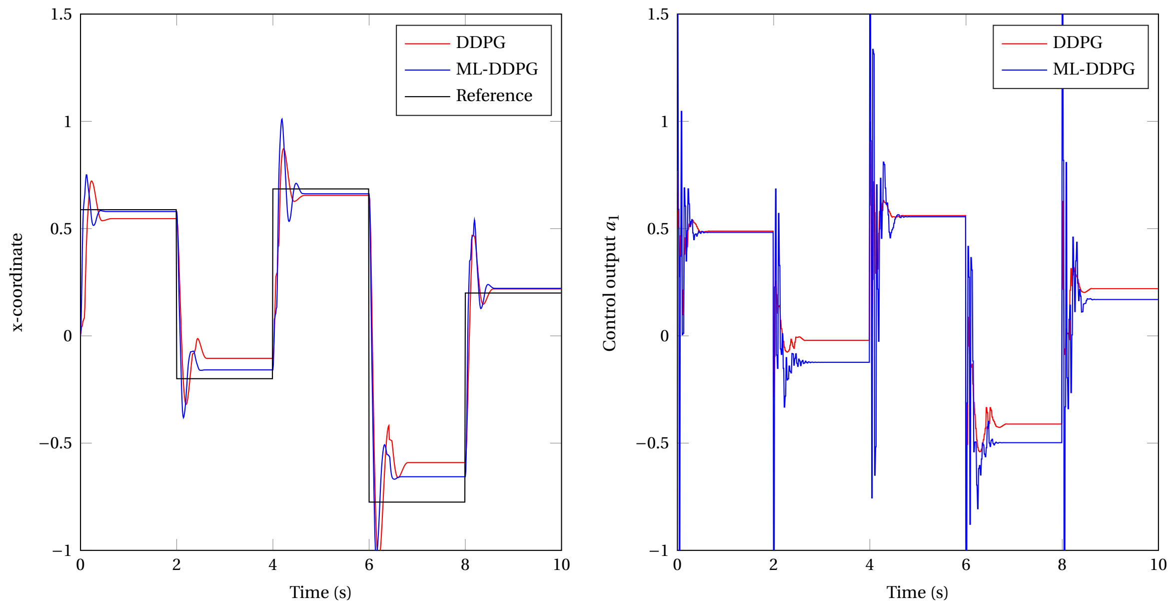


Figure 6.5: Time domain plot of the final policy on the 2-link arm benchmark, showing one of the states (left) and control actions (right). The $o^{\text{unrelated}}$ observation type is used and both algorithms are trained using $90K$ samples.

ML-DDPG is much smaller than for the DDPG.

6.2.3. Benchmark 3: Octopus

The results on the octopus benchmark are quite surprising. The benchmark is considered very difficult, because of the high dimensionality of the state and action spaces. In our experiments, however, each algorithm was able to learn the task relatively fast on a relatively small dataset (see Table 6.3). In the $7.5K$ dataset, the samples on which the algorithms are trained contains only one example in which the target was hit. Since it is hard to visualize the learned policy for a benchmark with these dimensions, videos of the octopus are made available that shows the final policy for the DDPG (https://youtu.be/-G9HV-baMC8), ML-DDPG (https://youtu.be/FctMsWc6RM0) and RP-DDPG (https://youtu.be/IYk-6ORiSkQ) algorithms.
The octopus has 36 individual muscles which can all be controlled individually by specifying its stiffness over time. When looking at the final control policy, 26 of these muscles have a stiffness that is either 1 or 0 during the entire episode and a further 7 are almost always 1 or 0. Only for 3 muscles, it learns a policy that really depends on the current state. This is also visible when looking at the video. From the start, the octopus quickly flexes its muscles to create a particular shape, from which it is able, using only a few of its muscles, to slowly move towards the target. Although this is certainly a novel way to complete the task, it is very different from the delicate control action that is necessary to bring the Inverted Pendulum or the 2-link arm to a certain reference position.

In order to see if the learned policy also generalized to other initial positions, the octopus arm was randomly excited for 2s before testing the learned policy again. Also, in these cases, the octopus was successful in reaching the food.

6.3. Conclusion

In this chapter, the RP-DDPG, ML-DDPG and the DDPG are compared on three different benchmark problems. From the initial results, it quickly became clear that the RP-DDPG was unable to learn anything on the Inverted Pendulum and 2-link arm benchmark. Further analysis led us to believe, that the reason behind the poor performance is the inability of the Robotic Prior method to extract both positional and velocity information from the observation vector.

The ML-DDPG outperforms the DDPG consistently on the 2-link arm benchmark. It is interesting to see that this is true even for large datasets. Since earlier research [21] suggested end-to-end training of a DNNs is always preferable to dividing up the problem and learning it in phases (like the Model Learning method

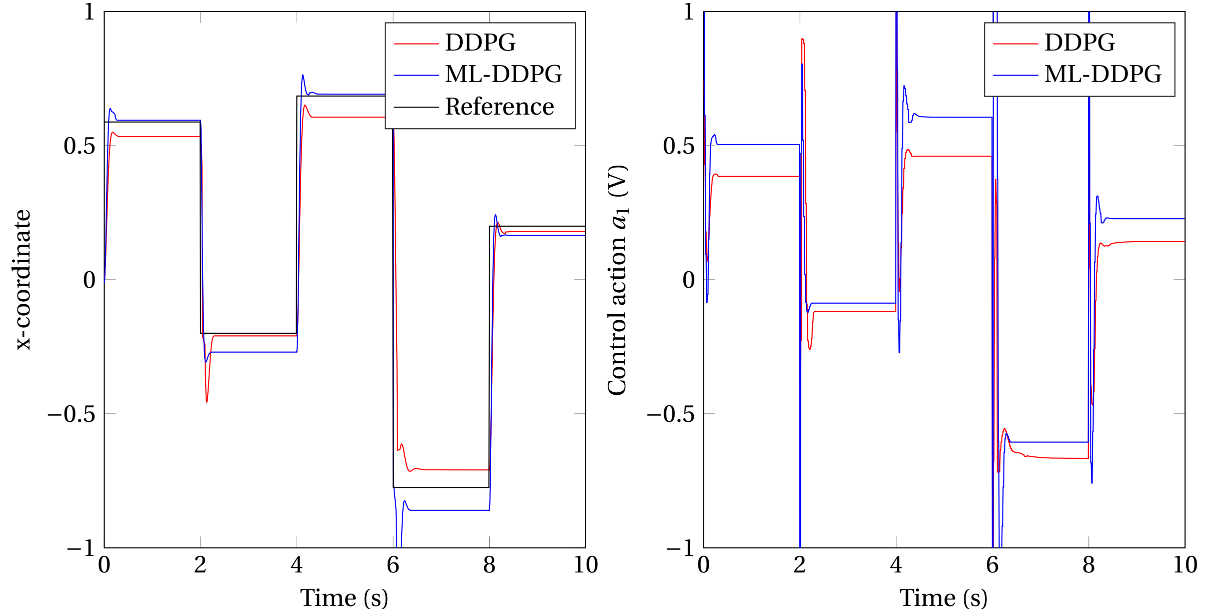


Figure 6.6: Time domain plot, showing the state and control action, of the final policy learned by the ML-DDPG and DDPG on the 2-link arm problem using the $o^{\text{classical}}$ observation vector. Both algorithms are trained using $90K$ samples.

Table 6.3: Learning curve characteristics on the octopus for different sizes of the experience replay database. The rise time τ_{rise} and settling time τ_s are denoted in ×1000 learning steps.

Input type	DB size	DDPG			ML-DDPG		
		τ_{rise}	τ_s	$\bar{R}$	τ_{rise}	τ_s	$\bar{R}$
$o^{\text{classical}}$	7.5K	26.1	28.8	−0.235	**2.3**	**3.5**	**−0.111**
	15K	**1**	**2**	**−0.058**	1.9	2.3	−0.069
	30K	**1.2**	**1.9**	**−0.036**	9.2	12.4	−0.040
$o^{\text{unrelated}}$	7.5K	0.5	23.5	−0.140	**0.4**	**1**	**−0.086**
	15K	0.8	22.1	**−0.061**	**0.7**	**1.3**	−0.068
	30K	**1.9**	**3.5**	**−0.024**	11.2	15	−0.057
$o^{\text{redundant}}$	7.5K	0.8	**2.2**	**−0.036**	**0.6**	6.9	−0.064
	15K	**1.1**	**6.7**	**−0.049**	2.8	18.1	−0.146
	30K	**0.6**	**13.5**	−0.057	20	21	**−0.053**

does), whenever the dataset is large enough. For the other two benchmarks, the results are not as conclusive.

In general, the overall performance of either the DDPG or the ML-DDPG on the Inverted Pendulum and the 2-link arm benchmark is not as good as you would normally expect from an optimal control method. As is visible in Figure 6.6, for some reference positions the response is quite optimal, but for other reference positions, the policy results in a large overshoot or steady-state error. Furthermore, if we were to create a time-domain response of the policy learned in one of the other trials, we find these problems happening for different reference positions. The main problem is therefore that the performance of the policies learned by the algorithms are not very reliable. This also makes it hard to tune these algorithms to lower, for instance, the steady-state error at certain reference positions.

From another perspective, the results are quite impressive. For instance, the $o^{\text{redundant}}$ observation vector of the 2-link arm benchmark has 24 dimensions, does not contain velocities and contains the reference angle in a different coordinate system than the configuration of the arm. Both the DDPG and the ML-DDPG can handle, the combination of these challenges, without making any specific change to the algorithm (to specifically deal with these type of challenges) and reach a performance, close to the performance it reaches using the $o^{\text{classical}}$ observation vector. Furthermore, the same algorithm can deal with a case in which half of the inputs are white noise or a benchmark problem like the Octopus in which the state space has up to 308 dimensions. It is therefore the ability to learn, in a wide variety of settings and in relatively difficult circumstances that make the performance of these algorithms very impressive.

7

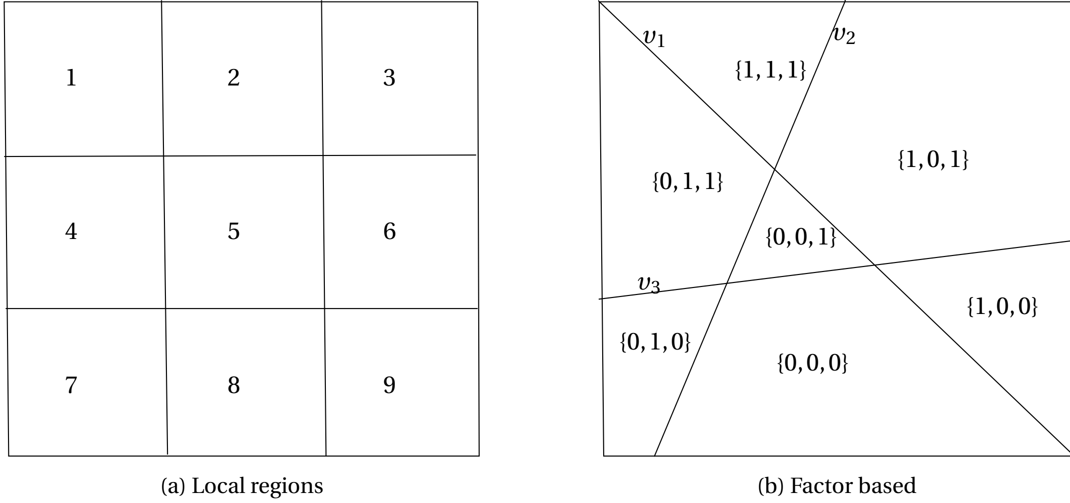
Generalization

In this chapter, the ability of a DNN to generalize, within the context of Reinforcement Learning (RL), is investigated. Generalization is taking one or a few facts and making a broader, more universal statement. In the context of function approximators, generalization is the ability to make good approximations of inputs that differ from the ones seen during training.
The claimed advantage of DNNs over local approximation methods is that they can generalize globally, i.e., they can generalize to examples that differ substantially from the examples seen during training. This ability of a DNN to generalize globally is said to help tackle the curse of dimensionality [2]. Section 7.1 will discuss the curse of dimensionality in more detail. In Section 7.2 an experiment is designed that tests to what extent an actor network can generalize the policy it has learned, by excluding certain parts of the observation space from the training set. The results of these experiments are presented in Section 7.3. Section 7.4 concludes this chapter.

7.1. The curse of dimensionality

In most local approximation methods the observation space $O \subset \mathbb{R}^n$ is divided into local regions. The simplest approach is to create an n-dimensional grid where each dimension is divided into p regions. This results in p^n regions, which grows exponentially with the number of dimensions. This exponential growth is generally referred to as the *curse of dimensionality* [4]. More complicated methods try to focus the resolution on certain parts of the observation-space, by making the regions smaller in places where the function that is approximated is more complex and making the regions bigger in places where the function is simpler or the approximation is less relevant [4]. This, however, does not solve the problem entirely, it just makes it more manageable for slightly larger values of n.
The core of the problem is that these local methods only exploit the principle of local generalization based on the *smoothness* assumption, which assumes that similar inputs have similar outputs. In local generalization, an approximation of an unseen example is done by interpolating between neighboring training examples [2]. This, however, always requires that there are data points in the training set for each region.
As was already explained in Section 2.1.1 a DNN takes a different approach. A DNN tries to find a set of independent factors $v \in \mathbb{R}^m$, that can be linearly combined into the target function. The number of factors, the network is learning, is independent of the dimensionality of the input n. Instead, it is related to the fundamental complexity of the function it is trying to approximate. These factors are independent in the sense that their contribution to (the value of) the target function, is independent of the activation of other factors. In order to learn how each factor contributes to the target function, the network does not need to see all possible combinations of factors (which would grow exponentially in m), it only needs to see an example in which the factor is activated and one for which the factor is deactivated (which grows linearly $2m$).
Figure 7.1 illustrates the two different approaches for a 2 dimensional observation space. In Figure 7.1a, the observation space is divided into local regions which requires data points in each individual region. In Figure 7.1b the observation space is partitioned based on factors that each divide the observation space in two, a region where the factor is present and one in which it is not. Multiple factors combined create an exponential number of regions. In the figure, each region contains a vector $w \in 0,1^m$ that denotes which of the factors is activated in the respective region. Since the factors are assumed to be independent, with respect to the

activation of other factors, the number of data points necessary to learn about these individual factors scales linearly with the number of factors.



(a) Local regions (b) Factor based

Figure 7.1: Two different approaches to partitioning a 2-dimensional observation space.

7.2. Experiment: Symmetry versus Independent factors

An experiment is designed to experimentally determine to what extent the actor can generalize its policy to parts of the observation space it has not seen during training. The basic idea is to train the DDPG on a dataset in which certain parts of the observation space are excluded. The learned policy of the actor is subsequently tested using a reference signal that forces the system to visit parts of the observation space that were not part of the training set. The performance of the algorithm is then compared to the performance of the algorithm when trained on a "normal" dataset, which was sampled uniformly from the observation-space.
The 2-link arm presented in Section 4.3 is used as a benchmark for this experiment. As input to the algorithm, the $o^{\text{classical}}$ observation vector is used. The observation space of the 2-link arm is divided into 2^4 regions, by discriminating between positive and negative values of the angle and angular velocity of each link. We denote each region as a string of four symbols $++--$, representation either positive or negative values for θ_1, θ_2, $\dot{\theta}_1$ and $\dot{\theta}_2$ respectively. Three datasets are then created

- **Uniform** - The uniform datasets contain samples belonging to each of the individual regions, the dataset is created by following a random exploration strategy based on the Ohrnstein-Uhlenbeck process.
- **Symmetry** - The symmetry dataset contain only data points in which both angles are positive. No criteria are used on the angular velocities.
- **Independent factors** The independent factor dataset contain all combinations in which 3 out of 4 symbols are positive.

The symmetry and independent factor datasets each cover 4 out of the 16 different regions. They are created by filtering a uniform dataset that is 4 times the size of the uniform dataset that is used in the experiment. This way each dataset contains roughly the same (30K) number of samples.
In the symmetry dataset, as the name implies, the actor network is assumed to generalize by exploiting the symmetry of the system. In the independent factors dataset, each symbol is assumed to represent an independent factor, the dataset contains both positive and negative values for each symbol. The negative values are, however, only present in the context of all other symbols being positive. Table 7.1 shows a list of all regions and lists, which regions are covered in each of the datasets.

7.3. Result

Figure 7.2 shows a plot of 5 training sequences of the DDPG for each of the datasets. Surprisingly the *independent* dataset and the *uniform* dataset reach the same performance. The *symmetry* dataset performs

Table 7.1: List of datasets and the regions covered by the dataset. A string of four symbols $++--$, representation either positive or negative values for $\theta_1, \theta_2, \dot{\theta}_1$ and $\dot{\theta}_2$ respectively.

	Uniform	Symmetry	Independent factors
$++++$	x	x	-
$+++-$	x	x	x
$++-+$	x	x	x
$+-++$	x	-	x
$-+++$	x	-	x
$++--$	x	x	-
$--++$	x	-	-
$+--+$	x	-	-
$-++-$	x	-	-
$+-+-$	x	-	-
$-+-+$	x	-	-
$---+$	x	-	-
$--+-$	x	-	-
$-+--$	x	-	-
$+---$	x	-	-
$----$	x	-	-

significantly worse. The latter is not very surprising, giving the way a DNN with Rectified Linear Unit (ReLU) activation neurons computes its output. Due to the $\max(0, z)$ operator, neurons that are activated given a certain input z will not be activated whenever the input changes sign. This essentially makes it impossible for a DNN ReLU activation neurons to generalize based on symmetry, which requires an input-output mapping where a change in sign on the inputs results in a change in sign on the outputs.

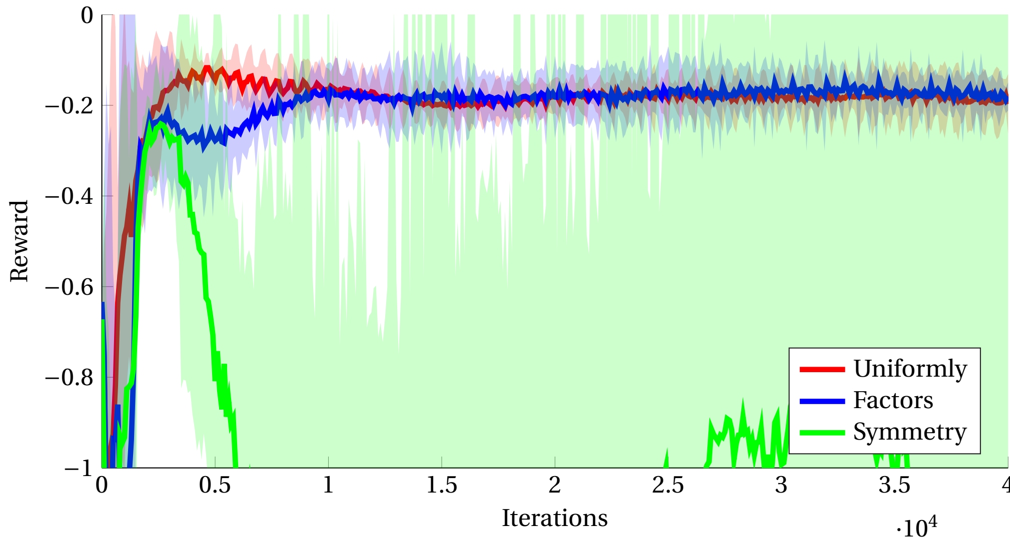


Figure 7.2: Plot of 5 training sequences of the DDPG for three different datasets on the 2-link arm benchmark problem. The $o^{\text{classical}}$ observation vector is used as input. The bold line is the mean score and the area shows the 95% uncertainty region. The reward is determined every 100 iterations by testing the intermediate policy, learned by the actor, on a predetermined reference signal for the duration of 1000 time steps.

Figure 7.3 shows the time-domain response of the final policy. Only the x-position of the tip of the second link is shown. It clearly shows a better performance (a lower steady state error) for the *independent* dataset whenever the angles are positive, which correspond to positive values of the x-coordinate. While the performance of the *uniform* dataset is better for negative angles. This can be attributed to the fact that the

independent dataset has significant more samples in regions of the observation space where the angles are positive, due to the way the dataset is constructed. This allows the algorithm to learn to track the reference more precisely in these regions. In regions where the angles are negative, it does learn a policy that stabilizes the system within 0.5s, although during training it has never been in a situation where both angles were negative.

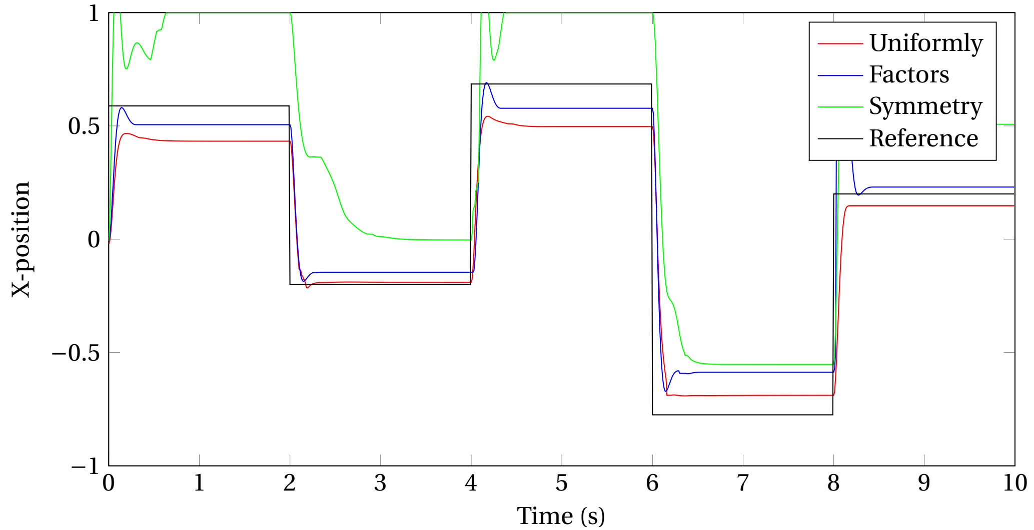


Figure 7.3: Time domain plot of the final policy learned by the DDPG for three different datasets on the 2-link arm benchmark problem. On the y-axis the x-coordinate of the tip of the second link is shown as well as the reference position.

In the experiment, the independent factors are assumed to be based on the angles and angular velocities of the 2-link arm, where each factor is based on one of these properties independently. The $o^{\text{classical}}$ observation vector also contains each of these properties independently, which could be argued, is the reason the DNN learns to identify these as the independent factors. The experiment is therefore repeated, but this time, the $o^{\text{redundant}}$ observation vector is used, which does not include the angular velocities directly. The results shown in Figure 7.4 shows that even in this scenario, the *independent* dataset and the *uniform* dataset reach the same performance.

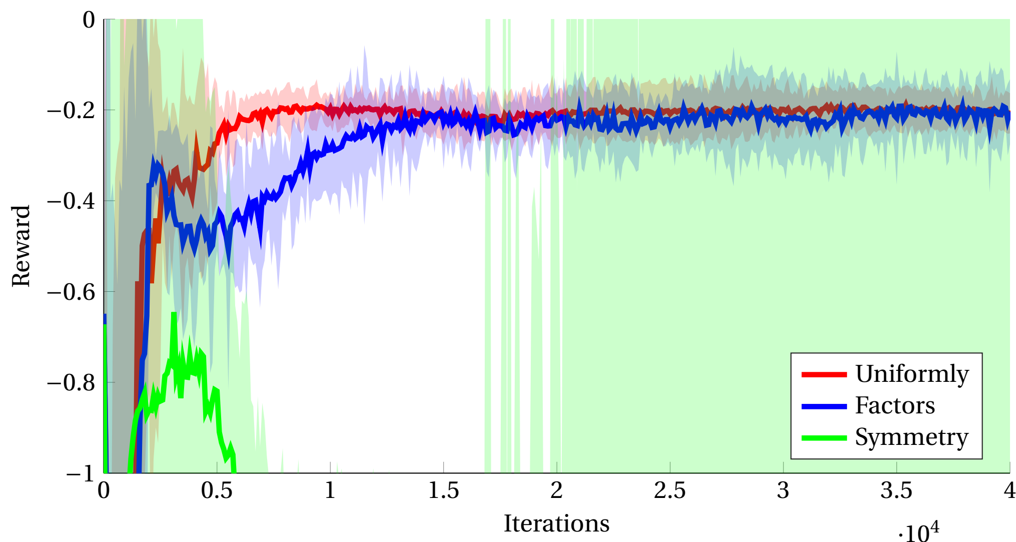


Figure 7.4: Plot of 5 training sequences of the DDPG for three different datasets on the 2-link arm benchmark problem. The $o^{\text{redundant}}$ observation vector is used as input.

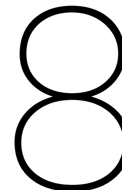
7.4. Conclusion

The experiments show, that for this benchmark, a DNN is able to generalize globally. An actor network, that is trained on a subset of the observation space, outputs a very reasonable control action in regions of the observation space it has never seen before. It can perform this generalization, however, only when it is able to learn all the different factors that explains the input-output relation of the underlying system. If the subset of the observation space it sees during training, does not include all the different factors, generalization will fail, as was the case with the symmetry dataset.

The experiments suggest a DNN generalizes very differently then local function approximators. Instead of interpolating between examples that are close, based on the similarity of the input, a DNN interpolates between examples that share similar factors. It learns to extract these factors, which share the fact that they contribute to the target value independent of each other. In case of the actor network, this target value is the control action. In contrast to local approximators the amount of data that is necessary to make a good approximation scales based on the intrinsic dimension of the system, given by the number of factors, instead of the input dimension. This makes sense since multiple input vectors of different dimension can be used to describe the same system, as shown in Chapter 4.

Besides the fact that the intrinsic dimension of the problem is sometimes smaller than the input dimension, the reason DNN will ultimately require much fewer data then local approximators, is the way the data requirements scale when either dimension increases. In case of a DNN, the assumption that the different factors are independent of each other means that the amount of data scales linearly with the number of factors. The different input values, in case of a local approximator, are obviously not independent of each other. This means local approximators need to see all combinations which scales exponentially in the number of dimensions.

Before the DNN has the advantage over a local approximator, however, it should have learned to extract these individual factors, which carries its own data requirements. This is why problems, for which a low dimensional observation vector can be found, are often solved more efficiently using a local approximator.



Conclusion and Recommendations

In this thesis, two new algorithms were designed, the Model Learning Deep Deterministic Policy Gradient (ML-DDPG) and the Robotic Prior Deep Deterministic Policy Gradient (RP-DDPG) that both combine State Representation Learning (SRL) with an actor-critic Reinforcement Learning (RL) algorithm. The performance of these algorithms is compared to the performance of Deep Deterministic Policy Gradient (DDPG), on three different benchmark problems. Furthermore, an experiment was carried out to investigate how a Deep Neural Network (DNN) can generalize the policy it has learned to parts of the state space it has not seen during training. This chapter summarizes the most important findings and proposes a few recommendations about interesting future research topics.

8.1. Summary and Conclusions

In this thesis, DNNs are used as function approximator of an actor-critic RL algorithms. The main advantage of using DNNs is that they can cope with high dimensional state and action spaces. Another approach, that is used to deal with a high dimensional state space, is to use SRL to learn a mapping from a high dimensional observation to a low dimensional state vector, prior to solving the RL task. This is based on the assumption that a RL algorithm can learn more efficiently from the learned state representation, then from the raw observation directly.

In the DDPG, both the actor and critic network learn *end-to-end*, i.e., they learn to map raw observations directly to actions and Q-values. In both DNNs, the first layer maps the observation to some internal state, which could be seen as an (implicitly learned) observation-to-state mapping. A disadvantage of learning *end-to-end* is that it requires lots of data, which is not always available. This thesis set out to investigate if the DDPG could benefit from learning a shared observation-to-state mapping explicitly, using a SRL method. In the two proposed algorithms, the actor and critic networks share an observation-to-state mapping by sharing the weights of their first layer. Furthermore, it uses SRL to learn these shared weights, before training the weights of the other layers.

Two SRL methods were implemented in Chapter 3, a Model Learning method and the Robotic Prior method, which led to two new algorithms, the ML-DDPG and the RP-DDPG. Each of these algorithms was tested extensively on three different benchmarks, introduced in Chapter 4. For each benchmark, three different types of observation vectors were defined, to simulate imperfect learning conditions.

- The *classical* observation vector, containing the physical state of the system, in the most suitable form.
- The *unrelated* observation vector, containing several white noise signals, which are added to simulate situations in which some of the inputs are irrelevant for the task at hand.
- The *redundant* observation vector, containing a series of positional measurements and previous actions. Velocity information is omitted, since this is often not available (or requires the engineer to design a separate observer).

The most important results were as follows.

- The RP-DDPG method never reached a performance that was significantly better than the performance at the start of learning, on two of the three benchmarks (2-link arm and the Inverted Pendulum). An early sign of its weak performance was the fact that for the state representation it learned, it was hard to categorize representations based on the velocity component of the physical state of the system. In order to reach a reasonable performance on the third benchmark (the Octopus), the RP-DDPG required four times more data than the other two algorithms.
- The ML-DDPG outperformed the DDPG method on one of the benchmarks (2-link arm), for each of the observation vectors and for different sizes of the dataset. It converged faster and reached a significantly better final performance. On the other two benchmarks, the results were not as conclusive. For some observation vectors and sizes of the dataset the ML-DDPG reached a better performance, for other combinations the DDPG reached a better performance. There was no clear underlying link that explained these different outcomes.
- In general, the control policies learned by both the DDPG and the ML-DDPG are not very optimal, compared to other optimal control methods. The controlled system has a significant steady state error and overshoot for some of the reference positions. Furthermore in each learning trial, these errors appear at different reference positions, which makes it hard to optimize these algorithms, for specific reference positions.
- Both the DDPG and the ML-DDPG can learn reasonable policies, under very difficult circumstances, for a wide variety of problems. They can deal with unrelated inputs, measurement and reference signals that are given in different coordinate systems and the absence of velocity information, without the need to specifically tune the algorithm to deal with these challenges.
- The actor network is able to generalize the policy it has learned, to states that substantially differ from the states it has seen during training. In the experiment, the network could generalize, from a situation in which the angles are positive and the angular velocities were not (and vice versa), to situations in which both the angles and angular velocities are either positive or negative.

8.2. Recommendations for future research

The research in this thesis has shown some of the strengths and weaknesses of combining Deep Learning (DL) and RL. The following areas are interesting topics for future research.

- One topic for which the ML-DDPG is well suited is multi-modal learning. In multi-modal learning the agent needs to combine information from different sources (e.g. visual, position and depth sensors) to learn a particular task [26]. The ML-DDPG could be used to learn a shared state representation from all these different observations. One advantage of this approach is that it would be fairly straightforward to see if and how, adding an extra source, lowers the prediction error of the model. Furthermore, it is much easier to test the performance of different network architectures, that each combine these sources in a different way in the semi-supervised Model Learning method then directly in the DDPG.
- Many systems are only partially observable, which means a single observation contains insufficient information to determine a Markov state of a system. In this thesis, that situation was simulated in the *redundant* observation type and solved by taking a series of observations to guarantee that a Markov state of the system could be determined. Another approach is to use a Recurrent Neural Network (RNN) that has its own internal state. This approach is used in [14] which created a variant of the DDPG with Long Short-Term Memory (LSTM) networks. For this algorithm, analogous to the ML-DDPG, the question is if it would not be better to train the recurrent part of the network by means of SRL and reusing this in both the actor and critic, instead of training two individual LSTM networks.
- One aspect that was very critical, when tuning the algorithms, was the weighting of the different components in the reward function. The trade-off between penalizing the distance D and the velocity component was an important parameter that, if set incorrectly, resulted in a very poor performance. Inspired by the results of Chapter 7, it should be possible to split the reward into a vector of separate rewards and train an actor-critic algorithm on each of the rewards independently. This produces multiple control policies, each one optimized to maximize a different reward over time. Once these networks are trained a final control policy can then be created by mixing the individual policies. The advantage

of this approach would be that you can weight the different components of the reward function after training and directly observe how this affects the final policy.

8.3. Final words

DNNs have become more popular as function approximator in RL algorithms. This has allowed RL algorithms to be used even in situations where the state and action dimensions are very high. This means that, although special tricks are necessary to make a DNN converge in a RL setting, in many cases they are now the preferred option. Many researchers still consider DNN to be a black box, which is hard to interpret and reason about. I believe this view to be outdated and counterproductive. Yes, a DNN has many parameters and can learn a complicated input-output mapping, but in many cases, this is justified since many systems in the real world are complicated. Especially if the data is not preprocessed by an engineer to simplify the input-output data of the system. A feed-forward Neural Network (NN) with Rectified Linear Unit (ReLU), however, is just a collection of linear models, like many other function approximators. In that sense, the technique is by no means more of a black box than other methods.

The ML-DDPG has shown that it can be beneficial to learn a model of the system prior to training the DDPG algorithm. This is not always the case, but it shows that good alternatives to *end-to-end* learning exist. I believe we have only seen a glimpse of the potential of DNNs as function approximator in RL algorithms. And there is much potential for methods that train DNNs in more complicated ways. Whether that is by training and sharing individual layers, using alternative objective functions or use entirely new ideas remains to be seen. In any case, I strongly advise others to be creative in the way they construct and train these NNs if only to get a better insight of what these networks are capable of.

A

Paper

Learning State Representation for Deep Actor-Critic Control

Jelle Munk, Jens Kober and Robert Babuška

Abstract— **Deep Neural Networks (DNNs) can be used as function approximators in Reinforcement Learning (RL). One advantage of DNNs is that they can cope with large input dimensions. Instead of relying on feature engineering to lower the input dimension, DNNs can extract the features from raw observations. The drawback of this *end-to-end learning* is that it usually requires a large amount of data, which for real-world control applications is not always available. In this paper, a new algorithm, Model Learning Deep Deterministic Policy Gradient (ML-DDPG), is proposed that combines RL with state representation learning, i.e., learning a mapping from an input vector to a state *before* solving the RL task. The ML-DDPG algorithm uses a concept we call *predictive priors* to learn a model network which is subsequently used to pre-train the first layer of the actor and critic networks. Simulation results show that the ML-DDPG can learn reasonable continuous control policies from high-dimensional observations that contain task-irrelevant information. Furthermore, in some cases, this approach significantly improves the final performance in comparison to end-to-end learning.**

I. Introduction

During recent years, there has been a growing interest in the use of Deep Neural Network (DNN) as function approximators in Reinforcement Learning (RL). DNNs were used in [1] and [2] to approximate the Q-function when Q-learning was used to learn a task from raw visual observations. In [3] this technique was combined with the actor-critic approach to form the Deep Deterministic Policy Gradient (DDPG) algorithm, which was able to solve several continuous control tasks, including the cart-pole benchmark [4] and the *cheetah* locomotion task introduced in [5].

One important aspect of learning to control is to learn how to efficiently gather the task-relevant sensory information necessary to make informed decisions [6]. Specifically in the case of a robot designed for a wide variety of tasks, the amount of sensory information needed for one particular task, is often far less than the total amount of information that the robot gathers through its sensors [7]. Furthermore, the sensory information (observations) typically requires preprocessing before it can be used in control as state information.

Instead of relying on an engineer to design a state estimator to reconstruct the state vector from a set of observations, it is preferable if this too can be learned by the machine. Learning such an observation-to-state mapping, prior to solving the RL problem, is known in the literature as *state representation learning* [7]. Several examples exist in which this approach has been successfully applied, for instance: by using an auto-encoder network [8], Slow Feature Analysis (SFA) [9], robotic priors [7] or by simultaneously learning the reward and transition function [10]. These examples have, however, all focused on learning from visual observations and none of them integrates these methods with algorithms that combine RL with DNNs.

This paper introduces a new algorithm called Model Learning Deep Deterministic Policy Gradient (ML-DDPG), in which state representation learning is combined with a RL algorithm that uses DNNs as a function approximator. The algorithm is designed to learn a wide range of continuous control policies on a varied range of challenging sets of observations, that are typically unsuitable for other control algorithms. It learns from a high-dimensional stream of data that can include information that is both relevant and irrelevant to the task at hand, and can include a Markov state directly or indirectly, by including sequences of actions and observations.

The ML-DDPG algorithm learns a model network based on the *predictive priors*, which assumes that the next state representation and the reward should both be predictable, given the current state and the action taken in that state. The model network is constructed in such a way that it learns the observation-to-state mapping by back-propagating the prediction errors, which results in a state representation that is inherently predictable. Both the actor and the critic then learn from the state representation instead of the raw observations.

The motivation behind the algorithm is to compare an approach based on end-to-end learning [11] with an approach in which learning a state representation from a set of observations precedes the learning of a good control policy for a given task, based on the learned state representation. Given enough data and learning time, end-to-end learning is believed to reach a performance that is superior to other approaches, since there are no constraints imposed on the network that restrict it in any way [12]. However, end-to-end learning requires a large amount of data [13], which is not always available. The aim of the ML-DDPG is, therefore, to outperform the DDPG when data is scarce without putting a constraint on the performance when data is abundant.

The algorithm is tested on several continuous control tasks, where for each task two challenging observation sets are defined. In one observation set, half of the inputs are "noise-states", which represent sensor measurements that have no relation to the task for which the agent receives rewards. For the second observation set, the system is assumed to be partially observable; the data set is therefore made up of a sequence of actions and measurements, where the sequence

J. Munk, J. Kober and R. Babuška are with the Delft Center for Systems and Control of Delft University of Technology, The Netherlands j.munk@student.tudelft.nl, {j.kober, r.babuska}@tudelft.nl.

is intentionally chosen to be larger than necessary to guarantee the Markov property. We show that our algorithm is able to extract from the observations the information that is relevant for the task at hand, and so improve the final policy learned by the agent.

The paper is organized as follows. Section II introduces RL, the DDPG and state representation learning. Section III details the new algorithm, the ML-DDPG. Simulation results are presented in Section IV and Section V concludes the paper.

II. Preliminaries

This work builds on earlier work that has been done within the RL community, specifically on the DDPG actor-critic algorithm introduced in [3], and on the concept known as state representation learning. The rest of this section explains this prior work in more detail.

A. Reinforcement Learning (RL)

In RL a learning agent interacts with an environment with the aim of maximizing the rewards received from the environment over time. A RL problem is modelled as a Markov Decision Process (MDP) described by the tuple $M = (S, A, f, r)$, where the state space S is a set of states $s \in \mathbb{R}^m$, the action space A is a set of actions $a \in \mathbb{R}^p$, $f : S \times A \to S$ is the state transition function, and $r : S \times A \to \mathbb{R}$ is the reward function. At each timestep t, the agent receives an observation $o_t \in \mathbb{R}^n$ that determines its current state s_t, it chooses an action a_t, receives a scalar reward $r_{t+1} \in \mathbb{R}$ according to the reward function r and transits to state s_{t+1} according to the transition function f.

The goal in RL is to learn a control policy $\pi : S \to A$ that maximizes the discounted sum of future rewards $R_t = \sum_{t=i}^{T} \gamma^{i-t} r(s_t, a_t)$ where $\gamma \in [0, 1]$ is the discount factor and T the number of time steps per learning episode.

The action-value function Q is often used in RL algorithms to denote the expected future reward given an action a_t taken in state s_t and thereafter following the policy π by taking the action $a^\pi = \pi(s)$. The Q function, in the form of a difference equation is given by

$$Q^\pi(s_t, a_t) = r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})).$$

B. Actor-Critic

In applications like robotics, where the state and action spaces are continuous, function approximators have to be used to approximate both the action-value function Q and the policy π [14]. Actor-critic algorithms are suitable in these situations since they allow both of these functions to be learned separately. This is in contrast with critic-only methods, which require a complicated optimization at every time step to find the policy.

In actor-critic methods, the critic learns the action-value function Q while the actor learns the policy π. In order to ensure that updates of the actor improve the expected discounted return, the update should follow the policy gradient [15]. The main idea behind actor-critic algorithms is that the critic provides the actor with the policy gradient. In theory, the critic should have converged before it can provide the actor with an unbiased estimate of the policy gradient, in practice however this requirement can be relaxed as long as the actor learns slower than the critic [15].

C. Deep Deterministic Policy Gradient (DDPG)

The DDPG algorithm is an off-policy actor-critic algorithm, first introduced in [3]. In this algorithm, both the actor and the critic are approximated by a DNN with parameter vectors ζ and ξ, respectively. The critic is trained by minimizing the squared Temporal Difference (TD) error given by

$$\mathcal{L}(\xi) = \Big(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\zeta)|\xi) - Q(s_t, a_t|\xi)\Big)^2. \quad (1)$$

The actor is updated in the direction of the policy gradient $\bigtriangledown Q_\zeta$ using the current approximation of the critic. The update of ζ with $\triangle\zeta$ is given by

$$\triangle\zeta = \bigtriangledown_a Q(s_t, \pi(s_t|\zeta)|\xi) \bigtriangledown_\zeta \pi(s_t|\zeta).$$

According to [16] the Q-function should be in the *compatible* form in order for the policy gradient to be unbiased. Although this is generally violated in the DDPG algorithm, with the addition of a few extra stability measures the algorithm has been shown to work well in practice.

A significant problem occurs when minimizing (1) [2]. The updates of the parameter ξ not only change the output of the critic network $Q(s_t, a_t|\xi)$, but they also change the target function $r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\zeta)|\xi)$ that the network is learning. This is due to the recursive nature of the action-value function. Similarly, updates to theactor parameter ζ also change the target function. This coupling can lead to unstable behaviour and can cause the learning process of the action-value function approximation to diverge.

A solution, proposed in [3], that reduces the coupling between the target function and the actor and critic networks, is to update the parameters of the target function using "soft" updates. Instead of using ζ and ξ directly, a separate set of weights ζ^- and ξ^- are used, which slowly track the parameters ζ and ξ of the actor and critic networks.

The "soft" updates are performed after each learning step, using the following update rule

$$\zeta^- \leftarrow \tau\zeta + (1-\tau)\zeta^-, \qquad \xi^- \leftarrow \tau\xi + (1-\tau)\xi^-$$

where $\tau \in (0, 1]$ represents the trade-off between the learning speed and stability. Using these new parameters, the squared TD error becomes

$$\mathcal{L}_c(\xi) = \Big(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}|\xi^-)|\zeta^-) - Q(s_t, a_t|\xi)\Big)^2.$$

A second measure to ensure stable learning is to use an experience replay database [1]. Samples collected from the system are stored in this database such that they can be reused at a later stage. This is necessary because DNN are global approximators and are prone to catastrophic forgetting, i.e., the network forgets what it has learned in some part when updating some other part. In order to prevent

Algorithm 1 DDPG

{Actor-Critic Learning}
Randomly initialize network weights ζ and ξ
$\zeta^- \leftarrow \zeta$ and $\xi^- \leftarrow \xi$ {set weights of target network}
for learning step = 1 **to** N **do**
  Sample random mini-batch from DB
  Calculate $\triangle\zeta$ and $\mathcal{L}_c$ over mini-batch
  $\zeta \leftarrow \zeta - \alpha \triangle \zeta$ and $\xi \leftarrow \xi - \alpha \frac{\partial \mathcal{L}_c}{\partial \xi}$
  $\zeta^- \leftarrow \tau\zeta + (1-\tau)\zeta^-$ and $\xi^- \leftarrow \tau\xi + (1-\tau)\xi^-$ {update target network}
end for

catastrophic forgetting, a DNN should be trained with mini-batches, where the samples in a mini-batch are independently and identically distributed. The experience replay database is necessary to create such mini-batches, although care should be taken to keep the data within the database varied enough to prevent it from over-fitting [17]. See Algorithm 1 for an overview of DDPG.

D. State Representation Learning

In RL, the state s is not always directly accessible, but needs to be constructed from a set of observations o. Such an observation-to-state map $f : O \rightarrow S$ can be the result of feature engineering, in which an engineer selects the observations and design the mapping, but this can also be learned from data. The process of learning the observation-to-state mapping is called state representation learning [7].

State representation learning is a form of unsupervised learning, i.e., there are no training examples available since it is not known apriori what the most suitable state representation is to solve the problem. Learning an observation-to-state mapping therefore involves either making assumptions about the structure of the state representation or learning the mapping as part of learning some other function.

In [8], [18], [19], an auto-encoder is used to find an observation-to-state mapping in which the observations are compressed into a low-dimensional state vector. The objective, during training, is to find states from which the original observations can be reconstructed. It subsequently learns a state representation that captures only the unique features of the observation, i.e., how they differ from other observations.

Another unsupervised method is Slow Feature Analysis (SFA) [9], which is based on the idea that most phenomena in the world change slowly over time. In [20], [18] this assumption is used to learn a mapping between visual observations and a state representation that gradually changes over time.

In [7], these and several other assumptions about the structure of a good state representation are combined into the so-called Robotic Priors. They are divided into the simplicity prior, the temporal coherence prior, the proportionality prior, the causality prior and the repeatability prior. For each of these priors, a loss function is defined. An observation-to-state mapping is subsequently trained to minimize the combined loss functions of the individual priors. The paper then shows a performance increase when using the learned state representation instead of the raw observations as input to the Neural Fitted Q-iteration algorithm [1].

III. Model Learning Deep Deterministic Policy Gradient (ML-DDPG)

A DNN approximates a function by learning a transformation from an input vector to a feature representation, that can be linearly combined in the output layer, to a target function [21]. Viewed in this way, an internal signal, between two layers of a DNN represents some intermediate representation that is somewhere between the original input and the final feature representation. The main idea behind the ML-DDPG is that that up to a certain point, an actor and critic network, can benefit from sharing their intermediate representation. Furthermore, we argue that this intermediate representation can be learned more effectively by a third (model learning) network. In the rest of this paper, we refer to this intermediate representation as the state and the transformation of the input to this state, as an observation-to-state mapping.

The ML-DDPG consist of three DNNs, a model network, a critic network and an actor network. The model network is trained by using a concept we call *predictive priors* and is integrated with the actor and critic networks by copying some of its weights. In the experiments, the creation of the experience replay database, learning the model network and training the actor and critic is done as separate steps. This allowed us to train both the DDPG and the ML-DDPG on exactly the same dataset and it simplified the training of individual layers of a DNN. Ultimately, however, the goal is to train the model network simultaneously with the actor and critic networks and create the experience replay database while learning, as in the original DDPG.

A. Predictive priors

The *predictive priors* consist of two separate priors. The first prior is the *predictable transition prior* which states that, given a certain state s_t and an action a_t taken in that state, one can predict the next state s_{t+1}. An important difference with other methods like [22], [10], is that we do not predict the next observation o_{t+1} but the next state s_{t+1}. This becomes important if the observation o_t contains task-irrelevant information. A state that needs to be able to predict the next observation still has to contain this task-irrelevant information to make the prediction, whereas in the proposed case this information can be ignored altogether. The second prior is the *predictable reward prior* which states that, given a certain state s_t and an action a_t taken in that state, one can predict the next reward r_{t+1}. This prior enforces that all information relevant to the task is available in the state, which helps the *predictable transition prior* to converge to something meaningful.

The *predictive priors* essentially enforces the two elementary properties of a MDP. The relevant information is, however, already present in the original observation. The point of using the predictive priors is to find a state representation from which it is easier, i.e., fewer transformations

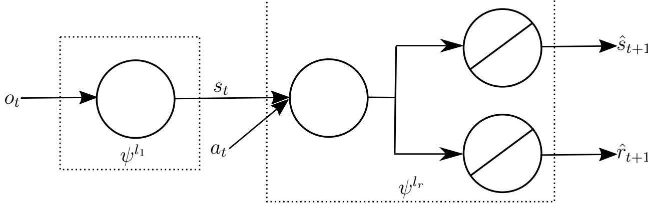


Fig. 1: Network architecture of the model network learning an observation-to-state mapping using predictive priors.

are necessary, to actually learn the transition and reward function.

The advantage of using the *predictive priors* is that state representation learning is transformed from an unsupervised learning problem to a supervised learning problem. A single interaction with the system produces a sample $\{o_t, a_t, r_{t+1}, o_{t+1}\}$, which can be mapped to $\{s_t, a_t, r_{t+1}, s_{t+1}\}$ using the current approximation of the observation-to-state mapping. This gives us both the inputs o_t and a_t as well as the target values r_{t+1} and s_{t+1}. Given a set of such samples and some function approximator, supervised learning can be used to find these mappings.

Another advantage of this approach is that the state representation that is learned is goal directed. Observations that do not correlate with the reward or are inherently unpredictable will not be encoded in the state representation. This is in contrast to methods like an auto-encoder or SFA since these methods do not differentiate between observations that are useful and observations that are not, to solve a particular task.

B. Model network

The *predictive priors* are implemented by a model network that learns a mapping from the observation-action tuple $\{o_t, a_t\}$ to the next state and reward $\{s_{t+1}, r_{t+1}\}$. The architecture is shown in Figure 1. The circles in the image represent a single layer containing multiple neurons, the lines are n-dimensional signals. The observations are the input to the first layer, which outputs the state. This state, together with the actions form the input to the second layer. Finally, two parallel linear output layers produce a prediction of the next state $\hat{s}_{t+1}$ and the reward $\hat{r}_{t+1}$ respectively. The observation-to-state mapping is encoded in the first layer of the network.

The network is trained by minimizing the following objective function

$$\mathcal{L}_m = \|\hat{s}_{t+1} - s_{t+1}\|_2^2 + \lambda \|\hat{r}_{t+1} - r_{t+1}\|_2^2$$

where λ represents the trade-off between predicting the reward and the next state. Note that, to obtain s_{t+1}, the current approximation of the observation-to-state mapping is used, to map the next observation o_{t+1} to the next state s_{t+1}. This could potentially lead to convergence problems, since the target depends on the current approximation. In practice however these problems did not occur. In all experiments, which all started from different random initial conditions, the learning converged to similar local optima.

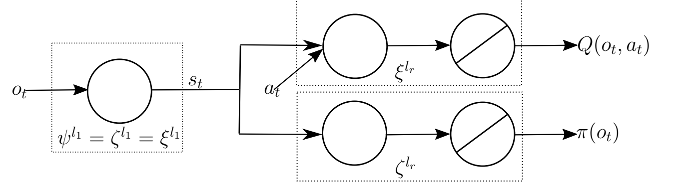


Fig. 2: Integration of the model, actor and critic network.

Algorithm 2 ML-DDPG

{Model learning}
Randomly initialize network weights ψ, ζ and ξ
for pre-training step = 1 **to** M **do**
  Sample random mini-batch from DB
  Calculate $\mathcal{L}_m$ over mini-batch
  $\psi \leftarrow \psi - \alpha \frac{\partial \mathcal{L}_m}{\partial \psi}$
end for
{Actor-Critic Learning}
$\zeta^{l_1} \leftarrow \psi^{l_1}$ and $\xi^{l_1} \leftarrow \psi^{l_1}$ {copy weights to actor and critic}
$\zeta^- \leftarrow \zeta$ and $\xi^- \leftarrow \xi$ {set weights of target network}
for learning step = 1 **to** N **do**
  Sample random mini-batch from DB
  Calculate $\mathcal{L}_a$ and $\mathcal{L}_c$ over mini-batch
  $\zeta^{l_r} \leftarrow \zeta^{l_r} - \alpha \frac{\partial \mathcal{L}_a}{\partial \zeta^{l_r}}$ and $\xi^{l_r} \leftarrow \xi^{l_r} - \alpha \frac{\partial \mathcal{L}_c}{\partial \xi^{l_r}}$
  $\zeta^- \leftarrow \tau\zeta + (1-\tau)\zeta^-$ and $\xi^- \leftarrow \tau\xi + (1-\tau)\xi^-$ {update target network}
end for

C. Integrating the model network

The model network is trained first, before the other two networks. Afterwards the observation-to-state mapping, that is encoded in the first layer of the model network, is copied to the first layer of, both the actor and the critic network. The parameter vectors ψ, ζ and ξ of the model, actor and critic respectively are therefore split in two parts where, ψ^{l_1}, ζ^{l_1} and ξ^{l_1} represent the weights of the first layer and ψ^{l_r}, ζ^{l_r} and ξ^{l_r} the weights of the remaining layers. Figure **??** shows how the actor and critic networks use the same observation-to-state mapping that is learned by the model network.

After pre-training, i.e., the model learning, the weights of the first layers ζ^{l_1} and ξ^{l_1} are fixed. Subsequently, standard actor-critic is used to learn the weights in the remaining layers of the actor ζ^{l_r} and critic ξ^{l_r}, see Algorithm 2. As in the DDPG (Algorithm 1) an experience replay database, batch-normalization, an L2 penalty on the critic weights and "soft" updates of the target networks are used to stabilize the learning. Adam [23] is used for learning the weights of all three DNNs with a base learning rate of 10^{-3}, 10^{-4} and 10^{-3} for the model, actor and critic respectively. The hidden layers of all three networks contain 100 neurons each.

TABLE I: Dimension table for the two benchmarks

	Action	Internal state	$o^{\text{unrelated}}$	$o^{\text{redundant}}$
2-link arm	2	6	18	24
Octopus	36	96	192	308

D. Saturation penalty

One specific problem we encountered, with both the DDPG and the ML-DDPG, was the fact that the actor sometimes learned actions that lay outside the saturation limits of the actuator. This is caused in part because all the samples from which the agents learn are collected prior to the experiment. If an agent learns actions outside the range, in which data was originally collected. The policy gradient, evaluated at these actions, is based on extrapolating the critic network, which for large deviations is very unreliable. This creates instability issues in both networks which hamper the convergence of the algorithm.

In order to restrict the action space a *saturation penalty* is added to the loss function of the actor. The loss function becomes

$$\mathcal{L}_a(\zeta) = -\,Q(s,\pi(s|\zeta)) + \lambda\Big(\max(\pi(s|\zeta)-5,0) + \max(-\pi(s|\zeta)-5,0)\Big)^2$$

where λ represents the trade-off between maximizing the reward and minimizing the saturation penalty. The actions are scaled such that they have zero mean and a standard deviation of 1, which puts the saturation limit at 5 times the standard deviation of the original exploration policy.

IV. Simulation Results

In order to compare the performance of the ML-DDPG algorithm with the DDPG algorithm, they are both applied to the 2-link arm problem from [17] and the octopus problem from [24]. In both benchmarks the state is not directly available, instead there are two types of observations:

1) $o^{\text{redundant}}$ - Includes a sequence of actions and measurements of a *partially observable* system. It is called redundant since the length of the sequence is chosen larger than necessary to give the observation vector the Markov property.
2) $o^{\text{unrelated}}$ - Includes the full state extended by a vector of white noise inputs.

Table I shows the dimensions of the action space, the internal state and the two observation types for both benchmarks.

Before the algorithms are run, data is collected by following a random policy based on the Ohrnstein-Uhlenbeck process [25]. The algorithms are then trained off-policy, on the same database. A single learning step consists of a single update of the weights of the actor and the critic, each experiment consists of 40000 learning steps for the 2-link arm problem and 30000 learning steps for the octopus. Every 100 learning steps the policy π is evaluated using a pre-defined reference signal. The performance of each of the two algorithms is compared on both of the observation types. Furthermore, the experiments are repeated for different sizes of the experience replay database, to compare how the algorithms perform when data is either scarce or abundant.

In order to make a quantitative comparison between the learning curves, the settling time τ_s, rise time τ_{rise} and the average performance $\bar{R}$ are calculated, see Appendix. In all experiments, the saturation penalty described in Section III-D, was a necessary condition for the algorithms to converge.

A. 2-link arm

The 2-link arm consists of two links, both of which can be controlled by a motorized joint. The angle of the first link $\theta_1 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ is measured with respect to the downward position and the angle of the second link $\theta_2 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ with respect to the first link. The two motorized joints can be controlled by setting a_1 and a_2, which are (a scaled version of) the motor voltages. Finally the reference position $p^{\text{ref}} = \begin{bmatrix} x^{\text{ref}} & y^{\text{ref}} \end{bmatrix}$ determines the desired position of the tip of the second link.

The reward from the environment is based on the Euclidian distance D between the reference position and the current position of the tip of the second link and on the angular velocities $\dot{\theta}$ of the two links. The reward function is given by

$$r(D,\dot{\theta}) = -\Big(D + w|\dot{\theta}|_2\Big)$$

where w represent the trade-off between the two terms.

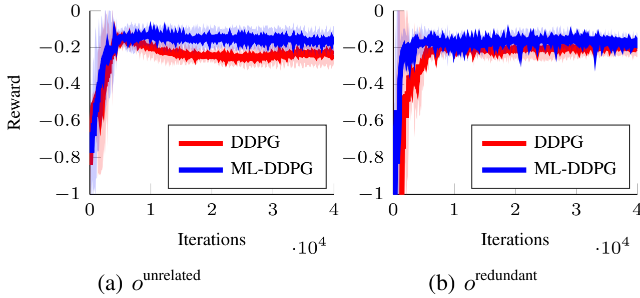
The observation vector $o^{\text{redundant}}$ contains the current angles θ and the angles and actions from the previous 5 timesteps and is given by

$$o_t^{\text{redundant}} = \begin{bmatrix} \theta_t & .. & \theta_{t-5} & a_{t-1} & .. & a_{t-5} & p_t^{\text{ref}} \end{bmatrix}$$

where θ_t is a vector of angles at time instance t, a_t a vector of the two control actions and p_t^{ref} the Cartesian reference position at timestep t. The observation vector $o^{\text{unrelated}}$ contains the current angles θ, the angular velocity $\dot{\theta}$, a vector of unrelated white noise inputs e_t and the reference position p_t^{ref} and is given by

$$o_t^{\text{unrelated}} = \begin{bmatrix} \theta_t & \dot{\theta}_t & e_t & p_t^{\text{ref}} \end{bmatrix}.$$

Figure 3 shows the mean (thick line) and standard deviation (shaded area) of the learning curve for both observation vectors using an experience replay database of 30K samples. The results from the other experiments are presented in Table II. For the 2-link arm benchmark, the ML-DDPG outperforms the DDPG algorithm in final performance $\bar{R}$ (+37.8% on $o^{\text{unrelated}}$ and $+8.1\%$ on $o^{\text{redundant}}$) and in rise time τ_{rise} -29% and settling time τ_s -36.2% on the $o^{\text{unrelated}}$ observation type. It does have slower convergence on the $o^{\text{redundant}}$ type (rise time τ_{rise} $+28.7\%$ and settling time τ_s $+39\%$). Both algorithms perform better, in terms of final performance, if more data is available. The advantage of the ML-DDPG over the DDPG seems relatively constant and not, as was expected, degrade when data becomes abundant.



(a) $o^{\text{unrelated}}$ (b) $o^{\text{redundant}}$

Fig. 3: Learning curve 2-link arm using 60K samples

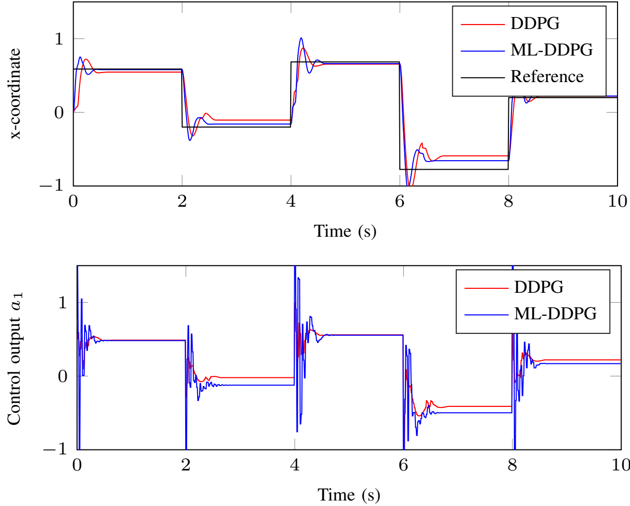


Fig. 4: Time domain plot of the final policy on the 2-link arm benchmark, showing one of the states (top) and control actions (bottom).

Figure 4 shows a time-domain plot of the x-coordinate of the tip of the second link and (one of the) accompanying control actions. It is clear that the performance is incomparable to other optimal control methods, the controlled system has a steady-state error and a significant overshoot. It is important to note, however, that the controller does not use a separate observer, although the system is partially observable and that the reference is given in Cartesian coordinates whereas the position of the 2 links are given in joint angles. The controller, therefore, needs to learn the non-linear mapping between the two, while also learning the unobservable states from a sequence of measurements. Seen in this light, we think the performance is actually quite good. We also believe the performance can be further increased by tuning the reward function and/or architecture of the DNNs which has not been done extensively to get these results.

B. Octopus

The octopus arm [24] consists of 13 segments, each of which contains multiple muscles. The muscles can be controlled by specifying its stiffness $a \in [0, 1]$. The arm is attached to a base at one side and moves in a 2-dimensional plane. The task is to reach to a randomly placed food target.

TABLE II: Learning curve characteristics on the 2-link arm for different sizes of the experience replay database. The rise time τ_{rise} and settling time τ_s are denoted in $\times 1000$ learning steps.

Input type	DB size	DDPG			ML-DDPG		
		τ_{rise}	τ_s	$\bar{R}$	τ_{rise}	τ_s	$\bar{R}$
$o^{\text{unrelated}}$	15K	3	**3.2**	−0.22	**2**	3.5	**−0.18**
	30K	3	8.6	−0.24	**2.8**	**3.8**	**−0.16**
	90K	3.4	3.7	−0.19	**1.9**	**2.6**	**−0.13**
$o^{\text{redundant}}$	15K	**2.5**	**11.6**	**−0.26**	3.7	17.2	−0.28
	30K	4.8	5.1	−0.21	**1.9**	**2.5**	**−0.18**
	90K	**5.9**	**6.2**	−0.19	11.3	12.2	**−0.15**

The task is completed if it touches the food target with any part of the arm.

The reward from the environment is based on the Euclidean distance D between the food and the segment that is closest to the food. Whenever the goal is reached an extra bonus B is given. The reward function is given by

$$r(D, B) = (B - 2) - D$$

where $B = 2$ whenever the goal is reached and $B = 0$ otherwise.

As in the 2-link arm benchmark two observation vectors are defined $o^{\text{redundant}}$, in which the velocity information is replaced by positions and actions at previous time instances, and $o^{\text{unrelated}}$, in which the state is extended with a vector of white noise inputs.

The results on the Octopus benchmark are not as conclusive as with the 2-link arm. Both algorithms have a similar rise and settling time and are able to learn the task in about 2000 learning steps. Both successfully learn to reach for the food, which takes them around 1.5s from their starting position. In order to see if the learned policy also generalized to other initial positions, the octopus arm was randomly excited for 2s before testing the learned policy again. Also, in these cases, the octopus was successful in reaching the food. Perhaps in spite of the high dimensionality of the problem, the Octopus problem is relatively easy since it does not require a very precise control action, like in the case of the 2-link arm.

V. Conclusion

This paper introduces a new algorithm, the ML-DDPG, that trains a model network using the *predictive priors* before learning the RL task. The main benefits of using the *predictive priors*, is that it can be formulated as a supervised learning problem using data from an experience replay database and can deal with task-irrelevant information in the observations. Learning an observation-to-state mapping, before training the actor and critic networks, has been shown to increase the final performance on a 2-link arm benchmark, even on a relatively large dataset.

Results in this paper confirm that using DNNs in actor-critic algorithms, is a very promising field of research, especially for cases in which the state and action dimensions

of the problem are very high. More work is necessary to visualise what kind of state representation the ML-DDPG is actually learning and how it performs on other benchmarks. Other future work will try to answer the more general question of how DNN seems to escape the curse of dimensionality.

APPENDIX

To define the settling time and the rise time of the learning curve, first introduce the undiscounted return after j learning steps averaged over the number N_e of learning experiments:

$$\bar{R}_j = \frac{1}{N_e} \sum_{l=1}^{N_e} \sum_{t=0}^{T} r(s_t, a_t)$$

where T is the duration of the evaluation, referring to the jth learning step within the lth learning experiment. For each reward, the sequence $\bar{R}_1, \bar{R}_2, \ldots, \bar{R}_{N_t}$ is normalized so that the minimum value of this sequence is -1.

The performance $\bar{R}_f$ at the end of learning is defined as the average normalized undiscounted return in the last c learning steps:

$$\bar{R}_f = \frac{1}{c} \sum_{j=N_t-c+1}^{N_t} \bar{R}_j$$

The settling time τ_s of the learning curve is then defined as the number of learning steps after which the learning curve enters and remains within a band ϵ of the final value $\bar{R}_f$:

$$\tau_s = T_t \cdot \arg\max_j (|\bar{R}_f - \bar{R}_j| \geq \epsilon \bar{R}_f)$$

In this paper c and ϵ are set to 1000 and 0.05 respectively. The rise time is defined as the number of learning steps required to climb from the 10% performance level to the 90% performance level:

$$\tau_{\text{rise}} = \tau_{90} - \tau_{10}$$

with τ_p defined as:

$$\tau_p = T_t \cdot \arg\max_j \left(\frac{\bar{R}_j - \bar{R}_1}{\bar{R}_f - \bar{R}_1} \geq \frac{p}{100} \right)$$

for $p = 10\%$ and 90%.

REFERENCES

[1] M. Riedmiller, "Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3720 LNAI, pp. 317–328, 2005.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[4] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *Systems, Man and Cybernetics, IEEE Transactions on*, no. 5, pp. 834–846, 1983.

[5] P. Wawrzyński, "Real-time reinforcement learning by sequential actor–critics and experience replay," *Neural Networks*, vol. 22, no. 10, pp. 1484–1497, 2009.

[6] D. M. Wolpert, J. Diedrichsen, and J. R. Flanagan, "Principles of sensorimotor learning.," *Nature reviews. Neuroscience*, vol. 12, pp. 739–51, Dec. 2011.

[7] R. Jonschkowski and O. Brock, "State Representation Learning in Robotics: Using Prior Knowledge about Physical Interaction," *Robotics: Science and Systems*, 2014.

[8] S. Lange, M. Riedmiller, and A. Voigtlander, "Autonomous reinforcement learning on raw visual input data in a real world application," in *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pp. 1–8, IEEE, 2012.

[9] L. Wiskott and T. J. Sejnowski, "Slow feature analysis: Unsupervised learning of invariances," *Neural computation*, vol. 14, no. 4, pp. 715–770, 2002.

[10] N. Jetchev, T. Lang, and M. Toussaint, "Learning grounded relational symbols from continuous data for abstract reasoning," 2013.

[11] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. L. Cun, "Off-road obstacle avoidance through end-to-end learning," in *Advances in neural information processing systems*, pp. 739–746, 2005.

[12] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *arXiv preprint arXiv:1504.00702*, 2015.

[13] W. Böhmer, J. T. Springenberg, J. Boedecker, M. Riedmiller, and K. Obermayer, "Autonomous learning of state representations for control: An emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations," *KI - Künstliche Intelligenz*, pp. 1–10, 2015.

[14] L. Bus, B. D. Schutter, and D. Ernst, *Reinforcement learning and dynamic programming using function approximators*.

[15] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems 12* (S. Solla, T. Leen, and K. Müller, eds.), pp. 1057–1063, MIT Press, 2000.

[16] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[17] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, "The importance of experience replay database composition in deep reinforcement learning,"

[18] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, "Learning visual feature spaces for robotic manipulation with deep spatial autoencoders," *arXiv preprint arXiv:1509.06113*, 2015.

[19] N. Wahlström, T. B. Schön, and M. P. Deisenroth, "Learning deep dynamical models from image pixels," *IFAC-PapersOnLine*, vol. 48, no. 28, pp. 1059–1064, 2015.

[20] R. Legenstein, N. Wilbert, and L. Wiskott, "Reinforcement learning on slow features of high-dimensional input streams," *PLoS Comput Biol*, vol. 6, no. 8, p. e1000894, 2010.

[21] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 1993, pp. 1–30, 2013.

[22] I. Grondman, M. Vaandrager, L. Buşoniu, R. Babuška, and E. Schuitema, "Efficient model learning methods for actor–critic control," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 42, no. 3, pp. 591–602, 2012.

[23] D. P. Kingma and J. L. Ba, "Adam: a Method for Stochastic Optimization," *International Conference on Learning Representations*, pp. 1–13, 2015.

[24] Y. Engel, P. Szabo, and D. Volkinshtein, "Learning to control an octopus arm with gaussian process temporal difference methods," in *Advances in neural information processing systems*, pp. 347–354, 2005.

[25] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Physical review*, vol. 36, no. 5, p. 823, 1930.

Bibliography

[1] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, (5):834–846, 1983.

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (1993):1–30, 2013. ISSN 01628828. doi: 10.1109/TPAMI.2013.50. URL `http://arxiv.org/abs/1206.5538$\delimiter"026E3B2$nhttp://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6472238`.

[3] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. URL `http://leon.bottou.org/papers/bottou-98x`. revised, oct 2012.

[4] Lucian Bus, Bart De Schutter, and Damien Ernst. *Reinforcement learning and dynamic programming using function approximators.*

[5] Wendelin Böhmer, Jost Tobias Springenberg, Joschka Boedecker, Martin Riedmiller, and Klaus Obermayer. Autonomous learning of state representations for control: An emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations. *KI - Künstliche Intelligenz*, pages 1–10, 2015. ISSN 0933-1875. doi: 10.1007/s13218-015-0356-1. URL `http://dx.doi.org/10.1007/s13218-015-0356-1`.

[6] Tim de Bruin, Jens Kober, Karl Tuyls, and Robert Babuška. The importance of experience replay database composition in deep reinforcement learning.

[7] Li Deng and Xiao Li. Machine learning paradigms for speech recognition: An overview. *Audio, Speech, and Language Processing, IEEE Transactions on*, 21(5):1060–1089, 2013.

[8] Yaakov Engel, Peter Szabo, and Dmitry Volkinshtein. Learning to control an octopus arm with gaussian process temporal difference methods. In *Advances in neural information processing systems*, pages 347–354, 2005.

[9] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable object detection using deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2154, 2014.

[10] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *arXiv preprint arXiv:1509.06113*, 2015.

[11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[12] I. Grondman, M. Vaandrager, L. Buşoniu, R. Babuška, and E. Schuitema. Efficient model learning methods for actor–critic control. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 42 (3):591–602, 2012.

[13] Nicolas Heess, David Silver, and Yee Whye Teh. Actor-critic reinforcement learning with energy-based policies. In *EWRL*, pages 43–58. Citeseer, 2012.

[14] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[15] G. E. Hinton and R.R Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science (New York, N.Y.)*, 313(July):504–507, 2006. ISSN 0036-8075.

[16] Nikolay Jetchev, Tobias Lang, and Marc Toussaint. Learning grounded relational symbols from continuous data for abstract reasoning. 2013.

[17] Rico Jonschkowski and Oliver Brock. State Representation Learning in Robotics: Using Prior Knowledge about Physical Interaction. *Robotics: Science and Systems*, 2014.

[18] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13, 2015.

[19] Stanislav Lange, Martin Riedmiller, and Arne Voigtlander. Autonomous reinforcement learning on raw visual input data in a real world application. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pages 1–8. IEEE, 2012.

[20] Robert Legenstein, Niko Wilbert, and Laurenz Wiskott. Reinforcement learning on slow features of high-dimensional input streams. *PLoS Comput Biol*, 6(8):e1000894, 2010.

[21] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702*, 2015.

[22] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[23] Laurens Van Der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. ISSN 02545330. doi: 10.1007/s10479-011-0841-3.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 0028-0836. doi: 10.1038/nature14236. URL `http://dx.doi.org/10.1038/nature14236`.

[25] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks. *Google Research Blog. Retrieved June*, 20, 2015.

[26] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696, 2011.

[27] Ronald Parr, Lihong Li, Gavin Taylor, Christopher Painter-Wakefield, and Michael L. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 752–759, 2008. doi: 10.1145/1390156.1390251. URL `http://portal.acm.org/citation.cfm?doid=1390156.1390251`.

[28] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098*, 2013.

[29] Martin Riedmiller. Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3720 LNAI:317–328, 2005. ISSN 03029743. doi: 10.1007/11564096_32.

[30] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach (International Edition)*. {Pearson US Imports & PHIPEs}, 2002.

[31] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[32] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[33] Tobias Springenberg, Joschka Boedecker, Martin Riedmiller, and Klaus Obermayer. Autonomous Learning of State Representations for Control. 29(4):1–10, 2015.

[34] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063. Citeseer, 1999.

[35] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

[36] Emanuel Todorov and Weiwei Li. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *American Control Conference, 2005. Proceedings of the 2005*, pages 300–306. IEEE, 2005.

[37] Niklas Wahlström, Thomas B Schön, and Marc Peter Deisenroth. Learning deep dynamical models from image pixels. *IFAC-PapersOnLine*, 48(28):1059–1064, 2015.

[38] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. In *Advances in Neural Information Processing Systems*, pages 2728–2736, 2015.

[39] Paweł Wawrzyński. Real-time reinforcement learning by sequential actor–critics and experience replay. *Neural Networks*, 22(10):1484–1497, 2009.

[40] Laurenz Wiskott and Terrence J Sejnowski. Slow feature analysis: Unsupervised learning of invariances. *Neural computation*, 14(4):715–770, 2002.

[41] Daniel M Wolpert, Jörn Diedrichsen, and J Randall Flanagan. Principles of sensorimotor learning. *Nature reviews. Neuroscience*, 12(12):739–51, December 2011. ISSN 1471-0048. doi: 10.1038/nrn3112. URL http://www.ncbi.nlm.nih.gov/pubmed/22033537.

Acronyms

AI	Artificial Intelligence.
DDPG	Deep Deterministic Policy Gradient.
DL	Deep Learning.
DNN	Deep Neural Network.
DOF	Degrees of Freedom.
LSTM	Long Short-Term Memory.
MDP	Markov Decision Process.
ML	Machine Learning.
ML-DDPG	Model Learning Deep Deterministic Policy Gradient.
MSE	Mean Squared Error.
NN	Neural Network.
RBF	Radial Basis Function.
ReLU	Rectified Linear Unit.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
RP-DDPG	Robotic Prior Deep Deterministic Policy Gradient.
SFA	Slow Feature Analysis.
SGD	Stochastic Gradient Descend.
SRL	State Representation Learning.
t-SNE	t-Distributed Stochastic Neighbor Embedding.
TD	Temporal Difference.