Sample effficient deep reinforcement learning for control

de Bruin, Tim

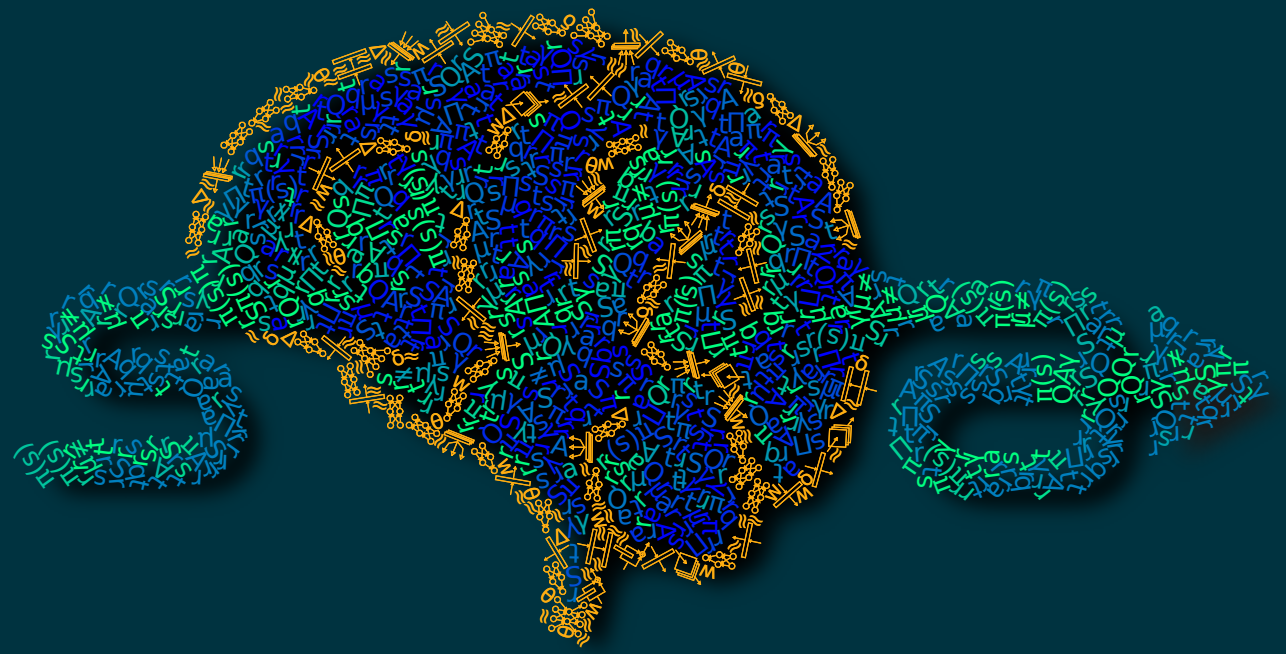
DOI
10.4233/uuid:f8faacb0-9a55-453d-97fd-0388a3c848ee

Publication date
2020

Document Version
Final published version

Citation (APA)
de Bruin, T. (2020). *Sample effficient deep reinforcement learning for control*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:f8faacb0-9a55-453d-97fd-0388a3c848ee

Sample Efficient Deep Reinforcement Learning for Control



Tim de Bruin

Sample Efficient Deep Reinforcement Learning for Control

T.D. de Bruin

Sample Efficient Deep Reinforcement Learning for Control

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus Prof. dr. ir. T. H. J. J. van der Hagen;
Chair of the Board for Doctorates
to be defended publicly on
Friday the 17th of January 2020 at 10:00 o'clock

by

Timon David DE BRUIN
Master of Science in Systems and Control, Delft University of Technology, the Netherlands
born in Amsterdam, the Nederlands

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	Chairperson
Prof. dr. R. Babuška	Delft University of Technology, promotor
Prof. dr. K. Tuyls	University of Liverpool, promotor
Dr.-Ing. J. Kober	Delft University of Technology, copromotor

Independent members:

Prof. dr. S. M. Bohte	Centrum voor Wiskunde en Informatica
Prof. Dr. O. Brock	Technical University of Berlin
Prof. dr. A. Nowé	Vrije Universiteit Brussel
Dr.-Ing. H. Vallery	Delft University of Technology
Prof. dr. M. Wisse	Delft University of Technology, reserve member

This work is part of the research programme Deep Learning for Robust Robot Control (DL-Force) with project number 656.000.003, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

Netherlands Organisation for Scientific Research
Physical Sciences

Printed by: Gildeprint

Front & Back: Tim de Bruin

Email: timdebruin89@gmail.com

ISBN 978-94-6384-096-5

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

Contents

Summary

The arrival of intelligent, general-purpose robots that can learn to perform new tasks autonomously has been promised for a long time now. Deep reinforcement learning, which combines reinforcement learning with deep neural network function approximation, has the potential to enable robots to learn to perform a wide range of new tasks while requiring very little prior knowledge or human help. This framework might therefore help to finally make general purpose robots a reality. However, the biggest successes of deep reinforcement learning have so far been in simulated game settings. To translate these successes to the real world, significant improvements are needed in the ability of these methods to learn quickly and safely. This thesis investigates what is needed to make this possible and makes contributions towards this goal.

Before deep reinforcement learning methods can be successfully applied in the robotics domain, an understanding is needed of how, when, and why deep learning and reinforcement learning work well together. This thesis therefore starts with a literature review, which is presented in Chapter 2. While the field is still in some regards in its infancy, it can already be noted that there are important components that are shared by successful algorithms. These components help to reconcile the differences between classical reinforcement learning methods and the training procedures used to successfully train deep neural networks. The main challenges in combining deep learning with reinforcement learning center around the interdependencies of the policy, the training data, and the training targets. Commonly used tools for managing the detrimental effects caused by these interdependencies include target networks, trust region updates, and experience replay buffers. Besides reviewing these components, a number of the more popular and historically relevant deep reinforcement learning methods are discussed.

Reinforcement learning involves learning through trial and error. However, robots (and their surroundings) are fragile, which makes these trials—and especially errors—very costly. Therefore, the amount of exploration that is performed will often need to be drastically reduced over time, especially once a reasonable behavior has already been found. We demonstrate how, using common experience replay techniques, this can quickly lead to forgetting previously learned successful behaviors. This problem is investigated in Chapter 3. Experiments are conducted to investigate what distribution of the experiences over the state-action space leads to desirable learning behavior and what distributions can cause problems. It is shown how actor-critic algorithms are especially sensitive to the lack of diversity in the action space that can result form reducing the amount of exploration over time. Further relations between the properties of the control problem at hand

and the required data distributions are also shown. These include a larger need for diversity in the action space when control frequencies are high and a reduced importance of data diversity for problems where generalizing the control strategy across the state-space is more difficult.

While Chapter 3 investigates what data distributions are most beneficial, Chapter 4 instead proposes practical algorithms to select useful experiences from a stream of experiences. We do not assume to have any control over the stream of experiences, which makes it possible to learn from additional sources of experience like other robots, experiences obtained while learning different tasks, and experiences obtained using predefined controllers. We make two separate judgments on the utility of individual experiences. The first judgment is on the long term utility of experiences, which is used to determine which experiences to keep in memory once the experience buffer is full. The second judgment is on the instantaneous utility of the experience to the learning agent. This judgment is used to determine which experiences should be sampled from the buffer to be learned from. To estimate the short and long term utility of the experiences we propose proxies based on the age, surprise, and the exploration intensity associated with the experiences. It is shown how prior knowledge of the control problem at hand can be used to decide which proxies to use. We additionally show how the knowledge of the control problem can be used to estimate the optimal size of the experience buffer and whether or not to use importance sampling to compensate for the bias introduced by the selection procedure. Together, these choices can lead to a more stable learning procedure and better performing controllers.

In Chapter 5 we look at what to learn form the collected data. The high price of data in the robotics domain makes it crucial to extract as much knowledge as possible from each and every datum. Reinforcement learning, by default, does not do so. We therefore supplement reinforcement learning with explicit state representation learning objectives. These objectives are based on the assumption that the neural network controller that is to be learned can be seen as consisting of two consecutive parts. The first part (referred to as the state encoder) maps the observed sensor data to a compact and concise representation of the state of the robot and its environment. The second part determines which actions to take based on this state representation. As the representation of the state of the world is useful for more than just completing the task at hand, it can also be trained with more general (state representation learning) objectives than just the reinforcement learning objective associated with the current task. We show how including these additional training objectives allows for learning a much more general state representation, which in turn makes it possible to learn broadly applicable control strategies more quickly. We also introduce a training method

that ensures that the added learning objectives further the goal of reinforcement learning, without destabilizing the learning process through their changes to the state encoder.

The final contribution of this thesis, presented in Chapter 6, focuses on the optimization procedure used to train the second part of the policy; the mapping from the state representation to the actions. While we show that the state encoder can be efficiently trained with standard gradient-based optimization techniques, perfecting this second mapping is more difficult. Obtaining high quality estimates of the gradients of the policy performance with respect to the parameters of this part of the neural network is usually not feasible. This means that while a reasonable policy can be obtained relatively quickly using gradient-based optimization approaches, this speed comes at the cost of the stability of the learning process as well as the final performance of the controller. Additionally, the unstable nature of this learning process brings with it an extreme sensitivity to the values of the hyper-parameters of the training method. This places an unfortunate emphasis on hyper-parameter tuning for getting deep reinforcement learning algorithms to work well. Gradient-free optimization algorithms can be more simple and stable, but tend to be much less sample efficient. We show how the desirable aspects of both methods can be combined by first training the entire network through gradient-based optimization and subsequently fine-tuning the final part of the network in a gradient-free manner. We demonstrate how this enables the policy to improve in a stable manner to a performance level not obtained by gradient-based optimization alone, using many fewer trials than methods using only gradient-free optimization.

Samenvatting

Al geruime tijd wordt de komst van intelligente, algemeen toepasbare robots—robots die zelfstandig nieuwe taken kunnen leren—aangekondigd. *Deep reinforcement learning*, een vorm van *reinforcement learning* waarbij functie benadering verricht wordt met behulp van diepe neurale netwerken, heeft de potentie om robots in staat te stellen om een breed scala aan nieuwe taken te leren met minimale menselijke hulp en voorkennis. Dit raamwerk zou daarom eindelijk de belofte van algemeen toepasbare robots in kunnen lossen. Tot nu toe liggen de voornaamste successen van *deep reinforcement learning* echter in het spelen van computerspelletjes. Om deze successen ook te behalen in de echte wereld zijn er significante verbeteringen nodig in het vermogen van deze methodes om snel en veilig te leren. Deze thesis onderzoekt wat er nodig is om dit mogelijk te maken en draagt bij aan dit doel.

Voordat *deep reinforcement learning* methodes succesvol toegepast kunnen worden in de robotica is er een begrip nodig van hoe, wanneer en waarom *deep learning* en *reinforcement learning* goed samenwerken. Deze thesis begint daarom in Hoofdstuk 2 met een literatuur onderzoek. Hoewel het *deep reinforcement learning* veld in veel opzichten nog in de kinderschoenen staat, kan al wel opgemerkt worden dat succesvolle methodes een aantal belangrijke componenten delen. Deze componenten helpen de verschillen te overbruggen tussen de klassieke *reinforcement learning* methodes en de trainings procedures van diepe neurale netwerken. De voornaamste uitdagingen bij het combineren van *deep learning* en *reinforcement learning* komen voort uit de onderlinge afhankelijkheden van de geleerde regelaar, de trainingsdata en de trainingsdoelen. Onder de vaak gebruikte onderdelen voor het inperken van de negatieve gevolgen van deze afhankelijkheden vallen doel-netwerken, vertrouwens regio updates en buffers voor het terugspelen van ervaringen. Naast deze componenten worden ook een aantal van de populairdere en historisch relevante *deep reinforcement learning* methodes besproken.

Bij *reinforcement learning* wordt er geleerd door middel van trial and error. Maar omdat robots (en hun omgeving) kwetsbaar zijn, is deze vorm van leren—met name het maken van fouten—erg kostbaar. Daarom moet de intensiteit waarmee nieuwe dingen worden uitgeprobeerd drastisch afnemen naarmate de tijd verstrijkt, vooral als een goede regelstrategie al gevonden is. Wij laten zien hoe, bij het gebruik van de standaard technieken voor het terugspelen van ervaringen, dit snel kan leiden tot het vergeten van eerder geleerde succesvolle regelstrategieën. Dit probleem wordt onderzocht in Hoofdstuk 3. In dit hoofdstuk worden experimenten uitgevoerd om te onderzoeken welke verdeling van de ervaringen over de toestand/observatie-actie ruimte leiden tot gewenst leergedrag en welke verdelin-

gen leiden tot problemen. Er wordt aangetoond dat *actor-critic* algoritmen extra gevoelig zijn voor het gebrek aan diversiteit in de actie ruimte dat kan voortkomen uit het verminderen van de exploratie. Tot slot worden verdere verbanden aangetoond tussen de benodigde distributie van de ervaringen en de eigenschappen van het op te lossen regelprobleem. Hieronder vallen een grotere behoefte aan diversiteit in de acties wanneer er op met grote frequentie geregeld wordt en een afname van het belang van diversiteit voor problemen waarbij het generaliseren van de regelstrategie over de toestand ruimte moeilijker is.

Waar Hoofdstuk 3 onderzocht welke ervarings verdelingen bevorderlijk zijn voor het leren, stelt Hoofdstuk 4 in plaats daarvan praktische algoritmes voor om te selecteren uit een stroom van ervaringen. Hierbij wordt aangenomen dat er geen controle is over de stroom van ervaringen, zodat er geleerd kan worden van extra bronnen van ervaringen zoals andere robots, ervaringen opgedaan tijdens het leren van andere taken en ervaringen opgedaan met vooraf geprogrammeerde regelaars. We maken twee afzonderlijke beoordelingen van het nut van individuele ervaringen. De eerste beoordeling heeft betrekking op de lange termijn waarde van de ervaring. Deze beoordeling wordt gebruikt om te bepalen welke ervaringen bewaard worden als de buffer eenmaal vol is. De tweede beoordeling heeft betrekking op de korte termijn en wordt gebruikt om te bepalen van welke ervaringen op het moment het beste geleerd kan worden. Om de waarde van de ervaringen op de korte en lange termijn in te schatten stellen we benaderings functies voor op basis van de leeftijd, verrassing, en de hoeveelheid exploratie die verbonden zijn aan de ervaring. Ook demonstreren we hoe kennis van het voorhanden zijnde regelprobleem gebruikt kan worden om een weloverwogen keuze te maken tussen deze functies. We laten ook zien hoe aan de hand van deze kennis de optimale grootte van de ervaringsbuffer geschat kan worden en hoe bepaald kan worden of *importance sampling* gebruikt moet worden om te compenseren voor de systematische fouten die geïntroduceerd worden door de selectie procedure. Samen kunnen deze keuzes leiden tot een stabieler leerproces dat resulteert in beter presterende regelaars.

In Hoofdstuk 5 kijken we naar wat er geleerd kan worden van de verzamelde ervaringen. De hoge kosten van ervaringen in de robotica maken het belangrijk om zo veel mogelijk kennis te destilleren uit iedere opgedane ervaring. *Reinforcement learning* algoritmes doen dit normaal gesproken niet. Daarom voegen we aan deze algoritmes explicite toestand-representatie leerdoelen toe. Deze leerdoelen zijn gebaseerd op de aanname dat de te leren neurale netwerk regelaar beschouwd wordt als twee opeenvolgende delen. Het eerste deel (dat we de toestand-codeermachine zullen noemen) beeldt de geobserveerde sensorsignalen af op een beknopte en bondige representatie van de toestand van de robot en diens omgeving. Het tweede deel bepaalt welke acties genomen worden op basis van

deze toestandsbeschrijving. Omdat de toestandsbeschrijving gebruikt kan worden voor meer dan alleen de huidige regeltaak, kan deze beschrijving ook geleerd worden van algemenere (toestand representatie leer) trainingsdoelen dan alleen het *reinforcement learning* leerdoel dat hoort bij de huidige regeltaak. We laten zien hoe het toevoegen van deze extra leerdoelen leidt tot het leren van een veel algemenere toestands omschrijving. Deze algemenere toestands omschrijving maakt het mogelijk om sneller algemeen toepasbare regelstrategieën te leren. Ook introduceren we een trainingsmethode die er voor zorgt dat de extra leerdoelen helpen bij het *reinforcement learning* doel, zonder het leerproces onstabiel te maken door de veranderingen aan de toestand-codeermachine.

De laatste bijdrage van deze thesis, die we uit de doeken doen in Hoofdstuk 6, richt zich op de optimalisatie procedure die gebruikt wordt voor het trainen van het tweede deel van de regelaar; het afbeelden van de toestandsrepresentatie op de stuur acties. We laten zien dat, hoewel de toestand-codeermachine efficiënt getraind kan worden met de standaard—op gradiënten gebaseerde—optimalisatie procedures, dit een stuk moeilijker is voor dit tweede deel van de regelaar. Het is meestal niet mogelijk om schattingen van goede kwaliteit te verkrijgen voor de afgeleiden van de prestaties van de regelaar ten opzichte van de parameters van het neurale netwerk. Dit betekent dat, hoewel op gradiënten gebaseerde optimalisatie procedures het mogelijk maken om snel een redelijke regelaar te trainen, deze snelheid ten koste gaat van de stabiliteit van het leerproces en daardoor de uiteindelijke prestaties van de regelaar. De instabiliteit van het leerproces zorgt verder voor een extreme gevoeligheid voor de waarden van de hyper-parameters van de leermethode. Dit zorgt voor een ongelukkige nadruk op het afstellen van deze waarden om *deep reinforcement learning* methodes goed te laten werken. Optimalisatie procedures die geen gebruik maken van gradiënten kunnen stabieler en simpeler zijn, maar zijn vaak ook veel minder efficient in het aantal benodigde ervaringen om een taak te leren. We laten zien hoe de aantrekkelijke eigenschappen van deze twee methodes gecombineerd kunnen worden. We doen dit door eerst het hele netwerk te trainen met een optimalisatie procedure die wel gradiënten gebruikt en daarna het laatste deel van de regelaar verder te optimaliseren met een optimalisatie procedure die dit niet doet. Dit resulteert in stabiele verbeteringen die resulteren in een beter presterende regelaar dan verkregen wordt met alleen het gebruik van op gradienten gebaseerde optimalisatie, maar met veel minder ervaringen dan bij het gebruik van alleen gradient vrije optimalisatie.

1

Introduction

For decades, robots have been useful as tools inside of factories. In these highly structured environments, they successfully perform simple repetitive tasks. For even longer, there has been the promise that they will someday soon become something more. That robots will help us in our everyday lives. That robots will become useful in the unstructured, changing, stochastic and ambiguous world that we live in. For this to happen, a paradigm shift is needed in their programming. It will no longer be possible to define all the behaviors they will require a priori. Instead, their programming will need to be adapted in the field. For robots to finally leave the factories, they need the ability to *learn*.

1.1 | Robots that learn

Robots that operate in changing environments will need to adapt existing behaviors. For robots to become more general purpose, like a household robot rather than a vacuum cleaning robot, they will additionally need the ability to learn completely new behaviors. They might even have to learn behaviors that have not been foreseen by their programmers. To make this learning process possible, their programming will need to include two key components.

The first component is a way to encode behaviors. These behaviors are defined by *mappings* from observations to actions. As illustrated in Figure 1.1, the robot's behavior results from first observing the state of the world through its sensors. The mapping is then used to determine which action to take, given the sensory observations. The robot performs the action, which changes the state of the world, and observes the new state of the world through its sensors. This process is repeated until the task is accomplished. To make sure the robot can exhibit even behaviors not thought of by its programmers, we need a way to encode a very diverse *set* of observation-to-action mappings. If we want a robot to exhibit a specific behavior, and no mapping exists in our set that represents this behavior, the robot has no way of learning it.

A robot that is theoretically able to exhibit very many behaviors is not yet useful. We want the robot to exhibit a behavior that actually solves a problem we are faced with. Therefore, we also require a second component: a way to search through this set of mappings for one that represents such a behavior. This *search process* is the mechanism by which the robot learns. An efficient search process will result in quick learning, while an inefficient search process results in a useless robot.

These two components are closely linked. The size and the structure of the set of mappings we encode will dictate which search strategies will be successful.

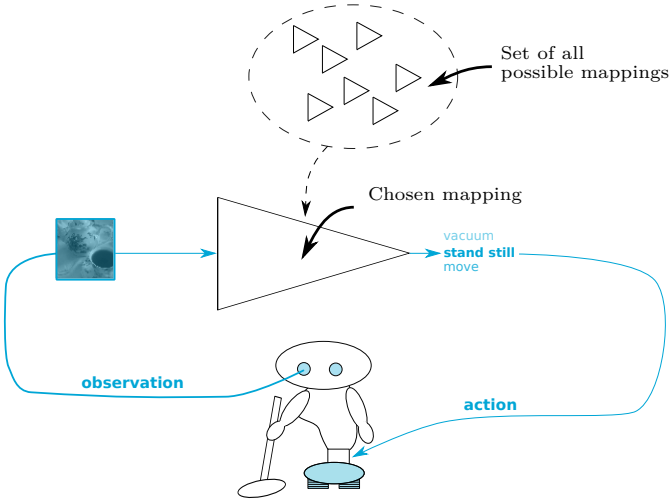


Figure 1.1: The robot's behavior results from repeatedly mapping its sensory observations to actions and performing those actions, which causes new observations. To make sure the robot can exhibit many different behaviors, we need a way to encode a large set of mappings from observation to action. To enable the robot to learn the right behavior, we need a way to search through this set of mappings to find one that induces a suitable behavior in the robot.

Encoding a varied set of behaviors and picking the right one

Lets examine the first component: a way to encode a broad set of mappings from observations to actions. We can divide these mappings into two sub-mappings. The first sub-mapping processes the observed sensor data by mapping it to a compact and concise representation of the state of the world. In the second sub-mapping, this *state representation* is mapped to an appropriate action.

In many robotic scenarios, a significant part of the complexity of representing the mapping from observation to action is in the sub-mapping from observation to state (Giusti et al., 2016). This is especially true for robots that need to perform tasks that we might consider trivial in environments not specifically designed for robots. Consider for example asking a general purpose robot to "clean under the couch". The robot might be outfitted with a microphone, which registers the request as a sequence of variations in air pressure. It can have a camera, which measures the intensity of the light bouncing off the objects in a room and hitting a grid of receptors. The first sub-mapping is from these signals to a relevant description of the state of the world. One such description could be the angle and distance of the robot relative to the couch and the presence of obstacles. Given this representation, the mapping to an appropriate action could be relatively simple. Extracting this representation is not. It requires understanding speech. It requires

understanding the equivalence of the word "couch" to the pattern of light that such a thing causes on a camera. It requires estimating depth from an image. It requires understanding what light intensity patterns represent drivable floor and what patterns represent obstacles.

The encoding scheme of the mappings from observations to actions should therefore be chosen such that we can efficiently learn to extract a representation of the state of the world from the robot's sensory signals as a part of this mapping. This requires an encoding scheme that is very general, as we might not know a priori what we will be looking for or listening to. At the same time, the scheme should include as much prior knowledge as is reasonable about the kind of sensory signals we need to process. This prior knowledge will help limit the set of mappings, and the amount of information needed to define one such mapping. This in turn will make it easier to search for a good mapping within the set, which allows for robots that learn more quickly.

Artificial Neural Networks (ANNs) provide one such way to encode mappings. ANNs can approximate a very large class of functions arbitrarily well, given enough parameters (Hornik, 1991). More importantly, Deep Neural Networks (DNNs) can approximate certain functions very efficiently, and the functions that are found in nature happen to be of this type (Lin et al., 2017). An important part of the reason for this efficiency comes from the fact that DNNs encode an hierarchical structure, with mappings often built up out of shared, simpler sub-mappings. This same property is found all throughout nature with small numbers of simple building blocks repeatedly being combined into increasing numbers of more complex structures. Specialized neural network variants can additionally include prior knowledge about the structure of specific sensory signals, such as the spatial relations in images or the temporal structure of audio signals. This knowledge enables further sharing of sub-mappings, reducing the amount of information needed to encode the final mapping without significantly reducing the number of useful mappings that can be encoded.

This makes it possible to encode a general enough set of functions, while keeping the function space small and structured enough to make searching it tractable. The suitability of deep neural networks for processing natural data has been demonstrated many times over the last couple of years. Examples include state of the art results for learning functions of images (e.g. Karras et al., 2018; Krizhevsky et al., 2012), sounds (e.g. Hinton et al., 2012; Zeghidour et al., 2018), language (e.g. Collobert et al., 2011b; Vaswani et al., 2017), and combinations of these modalities (e.g. Ngiam et al., 2011; Vinyals et al., 2015).

Given the large number of mappings (and by extension behaviors) that can be encoded by neural networks, we need an efficient and effective search strategy to

find an appropriate one. In all the examples of successful methods cited above, an a priori fixed dataset is used to learn from. In most, this dataset contains samples of both the inputs and the outputs of the mapping that needs to be found. From the set of mappings, a mapping should be chosen that not only maps those inputs to (approximately) the corresponding outputs, but also gives appropriate outputs for inputs that were not included in the training set, but that could reasonably be expected.

With the structure of the neural network (most commonly) chosen manually, only the network parameters have to be determined to define the mapping. Simple stochastic gradient-based optimization techniques are most commonly used to find the right values for these parameters. These search techniques start with a random parameter vector—and therefore mapping—and repeatedly make small adjustments to the parameters to map given inputs closer to the corresponding outputs. For large enough networks a local minimum of the loss function will almost always provide adequate performance (Choromanska et al., 2015). This simple local search for better parameters should therefore result in learning a good mapping, provided the example data were sufficiently descriptive of the desired mapping.

Unfortunately, these techniques that rely on a provided set of input output examples of the desired mapping do not readily apply to our robotics setting. To adapt to new tasks and changing environments, the robots need to update their programming away from their programmers. If we buy a new couch with a flower pattern, and our robot mistakes the couch for a garden, we can not rely on a team of programmers to rush in and fix the problem. The robot will need to quickly learn, possibly with some help of a user, to vacuum under—rather than pour water over—the couch. For the same reason that we cannot rely on predefined behaviors, we can also not rely (at least not completely) on predefined datasets, simulators, or examples of correct behaviors.

Reinforcement Learning (RL) is a framework based on trial and error learning that can help collect both the input samples and estimates of the desired outputs of the mappings (behaviors) that are to be learned. In RL, the task description is given in the form of a reward function. This function maps task relevant aspects of the state of the world and the action performed in that state to a scalar measure of instantaneous desirability. It might for example give a high reward for clean floors, while penalizing actions that deplete the battery. The objective of reinforcement learning is to learn to maximize the (possibly discounted) sum of rewards over time. To do this, we search through the set of mappings that are encoded by the neural network parameters. Each mapping represents a behavior; for any sensory observation that goes in, an action comes out. The challenge is finding the best

behavior in the set. The one that, when applied in the states that the robot might encounter, is most likely to pick the actions that will lead to the highest sum of rewards over time.

Optimizing over long time horizons is hard. For a vacuuming robot, every speck of dust that is sucked up might lead to instant gratification. It is difficult to learn to drive away from the dust and towards a charger when the battery runs low. In fact, it is hard for two key reasons. The first is that the further we look into the future, the more uncertain our predictions become. Driving towards a pile of dust right in front of the robot will almost certainly be rewarding. Driving away and coming back for it after charging could lead to the same rewards. But it would require finding the way back and hoping that an impatient human has not cleaned up the pile in the meantime. To pick the best behavior from the set, we need to compare how good they are, which becomes much harder under this uncertainty. A second difficulty is that finding better behaviors requires trying different things. A small variation on a behavior that only ever collects dust will not suddenly lead to driving to chargers. While a significantly different behavior might lead to charging, it might also lead to falling off stairs. A very delicate balance needs to be struck between doing what is known to work and trying new things. In spite of these difficulties, researchers have used reinforcement learning with impressive results. By trying to maximize the score in Atari games, behaviors were found that taught human gamers new strategies (Mnih et al., 2015). By trying to maximize the probability of winning the game of Go, behaviors were found that were previously thought to require human intuition (Silver et al., 2016). While the largest successes have so far been limited to games (where trial and error learning is relatively unproblematic) these successes do make an intriguing case for RL as a framework for finding successful behaviors.

Is the whole less than the sum of the parts?

It is easy then, to motivate each of the individual components. For robots to become useful in unpredictable environments, they need the ability to learn. Deep neural networks give them the ability to represent the behaviors they need to learn in an efficient manner. Reinforcement learning allows for actually learning these behaviors, using very minimal (human) feedback. The combination could lead to truly useful robots. Ones that do not need to be reprogrammed every time their surroundings change and could learn new skills, not thought of by their programmers. However, while motivating a desire to combine these components is easy, getting the combination to work is not.

Part of the reason that ANNs can efficiently encode functions of natural data is that they are global function approximators; there is one function (mapping)

that applies to all inputs. DNNs can be even more efficient by composing this mapping out of reusable sub-mappings. For recognizing a couch by its outline, a useful sub-function would be a line detector. This line detector could then be reused for detecting tables. This efficiency through parameter reuse makes learning these functions tractable. At the same time, it requires diversity in the training data. For only recognizing couches it might be beneficial to have the line detectors specialize in detecting couch-shaped lines. Yet this would hurt the table-detecting performance. To prevent over-specializing the sub functions in a way that hurts their general usefulness, deep learning methods present a wide variety of a priori collected examples in a randomized order. In fact, collecting the right dataset is often one of the most influential factors in the performance of deep learning methods. The right dataset enables learning the truly general shared sub-mappings. This in turn makes it possible to learn complex mappings of natural data in an efficient way.

Reinforcement learning methods have been developed with a different mindset. An a priori collected dataset is in general not used. Instead, an agent moves through the world, learning from observations as they are experienced. This means that when a robot is exploring under a couch, all of the subsequent samples will be related to couches. This was not an issue for the classical reinforcement learning methods, as they did not use global function approximators to represent the obtained knowledge. Learning about a couch would not change anything about the knowledge of tables, as the two were completely separate. As a result, the combination of reinforcement learning with neural networks did not succeed until this discrepancy was addressed (Lin, 1992; Mnih et al., 2015; Riedmiller, 2005). The solution was found in delaying the learning. An agent gathers a collection of experiences based on variations of its current behavior. When enough experiences are collected, they are learned from in a randomized order, restoring some of the sample diversity that is crucial in training neural networks.

With a buffer in place, the combination of RL with DNNs—known as Deep Reinforcement Learning (DRL)—becomes somewhat more similar to the previously mentioned successful applications of DNNs. Yet problematic differences remain. One of these differences is that while the buffer gives us some diversity and stability in the input distribution, the corresponding outputs are not given. In reinforcement learning these outputs usually describe the long term effects of a behavior. Estimating these long term effects involves a combination of trying out the behaviors to get true samples of their effects and estimating the effects from samples of (somewhat) different behaviors. Trying out behaviors to get true samples of their effects is problematic when using robots, as it requires many expensive robot interactions (Kober et al., 2013). The other strategy involves training our mappings

to output predictions of long term effects, where these predictions are determined mostly by the mappings that we are trying to train. This feedback loop can quickly destabilize the optimization of DNNs, as the network parameters are optimized to reinforce the networks poor predictions (Mnih et al., 2015).

The successes of DRL have generated a lot of enthusiasm for the potential of these methods. Yet the challenges—of which we only provided a sample here—mean that success is far from guaranteed even when applying these methods to simple problems. The complexity that is born out of the combination of these relatively simple parts means that DRL methods are often poorly understood even by their creators (Tucker et al., 2018) and published results are often not statistically significant (Henderson et al., 2017). Even seemingly impressive results are sometimes no better than those obtained by much more simple methods (Mania et al., 2018).

To evaluate the use of these methods for enabling robots to learn autonomously, we should therefore first better understand when, how and why DRL works.

1.2 | This thesis

This thesis looks at combining deep learning and reinforcement learning, while keeping in mind the constraints that are imposed by the robotics domain. The aim is to understand why and when these individual parts work, and how the combination can exploit the strengths of both. A visual summary of the topics discussed in this thesis is given in Figure 1.2.

1.2.1 Deep Reinforcement learning

In Chapter 2 we begin by reviewing some preliminaries on both reinforcement learning and deep learning. Here the notation used in this thesis is also introduced. After discussing the relevant aspects of these two individual fields, we turn to the combination: deep reinforcement learning. A survey is presented of the problems that arise when combining these two fields. We discuss DRL methods from the literature, with a specific focus on how they address these common challenges. Attention is also given to the opportunities that the combination of DL and RL provides. Methods that attempt to exploit those opportunities are reviewed.

1.2.2 Experience Selection

After this review of existing methods, we will focus on one of the most crucial parts of any machine learning algorithm: data. As discussed, the view of data from the deep learning and (traditional) reinforcement learning communities is quite different. The success of typical deep learning methods often hinges on the

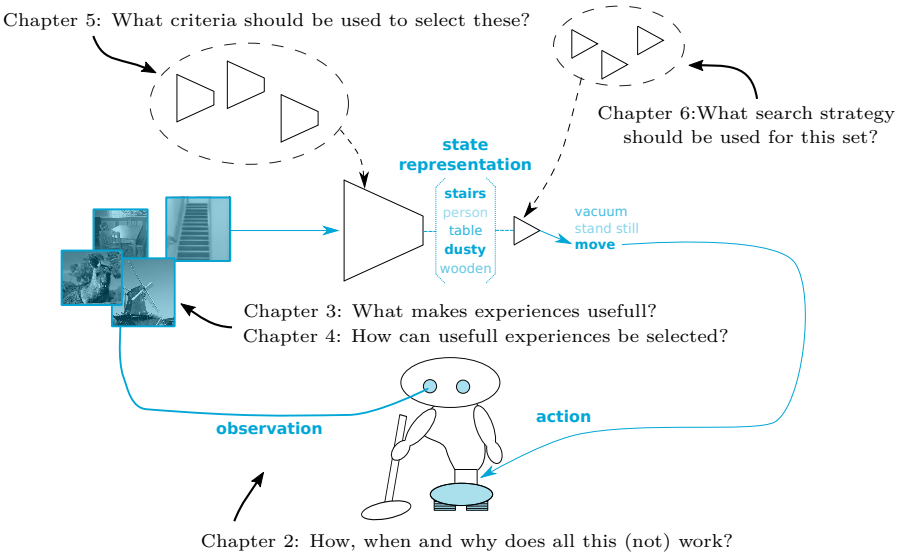


Figure 1.2: This thesis looks at some of the fundamental questions that arise when trying to combine deep learning with reinforcement under the constraints imposed by the robotics domain.

availability of a carefully curated data set. In reinforcement learning, the focus is very much on new data. Exploration strategies are developed to ensure that what is observed next is interesting enough to learn from. Once obtained, data quickly fall out of fancy. In almost all deep reinforcement learning methods, each data point only stays in a memory for a fixed amount of time before it is forgotten. But in robotics, where trials and especially errors come at a high price, can we really afford to forget about past mistakes? Is uninhibited exploration really feasible or should we try to learn optimally from any experience we can get? And how do we combine the data needs of neural networks and reinforcement learning? Should we get diverse data to allow the neural networks to learn general functions? Or should the data be closely related to the policy we want to evaluate for reinforcement learning to work?

In Chapter 3 we investigate how the performance of deep reinforcement learning depends on the contents of the experience buffer. The focus here is on learning to control physical systems. We investigate how the properties of these tasks influence the need for certain experience distributions. These properties include such factors as the sampling frequency, the need for generalization, the presence of noise and the RL algorithm used.

Where Chapter 3 investigates what data we would like to have for the combination of deep learning and reinforcement learning to work, the focus in Chapter 4 is on obtaining these data. We consider the scenario in which we cannot choose the data gathering behavior policy freely. This could happen when exploration is too damaging, when learning from the experiences obtained by other robots or when learning a new task from previous experiences. We therefore do not investigate exploration strategies. Instead, we consider the stream of experiences observed by the agent as a given. In this scenario we need to estimate the value of observed experiences and select the right ones for training. We do this by making two judgments. The first is, given a buffer of experiences, which ones to learn from. This requires a judgment on the immediate value to the learner of the experience. The second judgment is made when determining which experiences to keep once the buffer is full. This requires an estimate of the long term value of experiences.

1.2.3 State Representation Learning

The mappings encoded by deep neural networks consist of sub-mappings that range from very general to very task specific. The more general sub-mappings can be shared by several (sub-)tasks, which enables efficient learning. This also means that we can learn some of the building blocks of behavior functions by training on different tasks that also need these components. These tasks can often be much easier than the reinforcement learning objective of estimating long term effects. We investigate using a number of these additional learning objectives. By using a range of general objectives, very general sub-mappings are learned. These general sub-mappings enable learning behaviors that are themselves more general. Behaviors that solve tasks not just in the environment that they were trained in, but also in new environments. Unfortunately, all these different objectives perform a tug-of-war with our sub-functions. When a behavior is made up of building blocks that are suddenly changed to perform better on a different task, the behavior can change in an unpredictable way. In Chapter 5 we look into ways of preventing these unfortunate side effects.

1.2.4 Optimization strategies

With the data selected and the objectives chosen, all that is left to do is to use an optimization algorithm to search for parameter values that lead to good performance according to the chosen objectives on the collected data points. For the parameter values of deep neural networks, first-order gradient techniques are most commonly used. These techniques use estimates of the direction in which the parameters should be changed in order to improve the task performance. One challenge of this approach is that while we know the direction, we do not know the ideal size of the step that should be taken in this direction. Taking steps that

are too small slows down learning and can prevent escaping non-optimal critical points, while the nonlinear nature of neural networks means that taking a step that is too large can quickly lead to severely deteriorated performance. Potentially even more problematic is the fact that the estimates of the direction in which the parameters should be updated, as obtained through reinforcement learning, are not always accurate. For the more general sub-functions—encoded by the layers early in the network—the direction can be determined fairly well. For the more task specific sub functions–encoded by the later layers—this is much harder.

To see why, let us return to the vacuuming robot. Imagine it is standing in front of the stairs, deciding whether to drive forwards or backwards. Using a fairly standard reinforcement-learning technique, the neural network needs to be trained to map the sensor data (such as an image showing the stairs, and the drop beyond) to the expected long term sum of rewards for the two options. Early sub-mappings of the network might need to detect the lines that make up the stairs. These same functions can also be learned while learning drive around tables or learning not to crash into walls. Later sub-mappings might specifically detect stairs, for which we still get information whether we drive forwards or backwards. The final sub-mapping gives the long term return estimates. Not only is this difficult to determine, due to the fact that it involves predicting part of the future, but we also only get a single measurement of one of the two options. This makes the direction in which we update the parameters corresponding to this final sub-mapping of the neural network more uncertain than those of the sub-mappings before.

As the estimates of the direction in which we should update the parameters of the final sub-mapping can be poor, updates in wrong directions are common. This means that suddenly behaviors are tried that are worse than those that were found previously. When calculating the gradients this way, both the estimation of the long term effects as well as the data gathered are dependent on the current behavior. This means that the update direction estimates might become even worse and the optimizer can quickly loose its way in the parameter space. In practice this tends to mean that while these gradient-based optimization strategies can relatively quickly find a decent policy, they tend to be unstable and struggle to go from a decent policy to a great policy. In Chapter 6 we will therefore investigate a different way of fine-tuning these final parameters. Instead of repeated small steps in an uncertain direction, we will repeatedly sample from a distribution over the parameter space, test the sampled parameters for a while, and update the distribution to make the more successful parameters more likely to be sampled again. We show in Chapter 6 how this gradient-free fine-tuning leads to finding better behaviors in a more stable way.

2

Deep Reinforcement Learning

Parts of this chapter have previously been published in:

Buşoniu, L., de Bruin, T., Tolić, D., Kober, J., Palunko, I. (2018). *"Reinforcement learning for control: Performance, stability, and deep approximators"*. Annual Reviews in Control (ARC).

This chapter will discus the basic components that are combined in this thesis: reinforcement learning and deep learning (for more detailed reviews, see Goodfellow et al., 2016; Sutton and Barto, 2018). Besides covering the required preliminary knowledge and the notation used in this thesis, attention will be given to the aspects of these methods that will provide the opportunities and the pitfalls resulting from their combination. We will also discuss existing methods in the deep reinforcement learning subfield, with a focus on the common strategies for coping with the problems resulting from the combination, as well as exploiting the opportunities.

2.1 | Reinforcement Learning

Reinforcement learning is a framework that enables solving sequential decision making problems. These problems can be framed as Markov Decision Processes (MDPs). An MDP is defined by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a dynamics function $\mathcal{P}(s, s'|a)$ that describes the probability of transitioning from state s to state s' when taking action a, and a reward function $\mathcal{R}(s, a, s')$ that describes the instantaneous desirability of the transition from s to s' using action a as a scalar.

Instead of only considering the reward for a single transition, decisions should be made with the aim of transitioning towards states that are more rewarding in the long term. The optimality of a sequence of control decisions will be measured by the return R, which is the long term sum of rewards. In this thesis the return is defined as:

$$R = \sum_{k=0}^{K} \gamma^k r_k, \tag{2.1}$$

where $\gamma \in [0, 1)$ is a discount factor that keeps the sum finite and enables emphasizing shorter term rewards. We use k to indicate the discrete time steps (of an episode) at which control decisions are made. K is the time-step at which the environment terminates, which can be ∞. The return gives a measure of the quality of the sequence of decisions made from a single initial state s_0. The aim of reinforcement learning is to come up with a policy that optimizes the return for the initial state distribution $\mathcal{S}_0$:

$$J = \mathbb{E}_{s_{k=0} \sim \mathcal{S}_0} R(s_{k=0}). \tag{2.2}$$

For episodic tasks where the environment is guaranteed to terminate in a finite number of steps ($K < \infty$), γ *can* be 1. However, even the when we are interested in optimizing for this undiscounted return, the discounted return is often used as a proxy that is easier to optimize for (Marbach and Tsitsiklis, 2003; Schulman et al.,

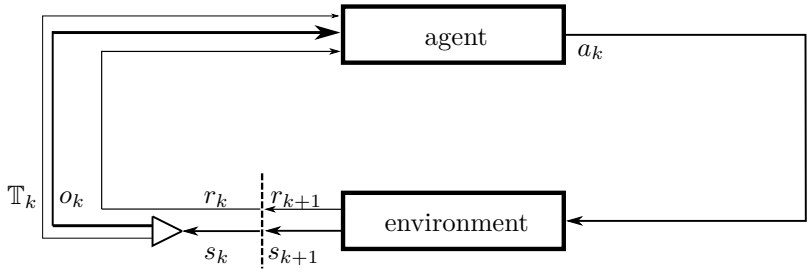


Figure 2.1: The agent-environment interface. The agent sends actions a to the environment, which change the environments internal state s. After a delay one one time step k, the agent receives a reward r, an observation o that describes the new environment state, and a signal $\mathbb{T}$ which indicates whether the new environment state is terminal.

2015b). In all but the last chapter of this thesis we consider the discounted return as the optimization objective.

The MDP framework is important for developing the theory of RL. In practice, we consider only the set of actions $\mathcal{A}$ to be directly available to the algorithms we use. In this thesis, we consider the actions that can be taken to be independent of the state. Besides the actions, we can only sample the rest of the MDP through the agent-environment interface shown in Figure 2.1.

At every time-step k, an observation o_k is obtained from the environment. In this thesis, we consider this observation to contain all of the information contained in the state s of the environment, although potentially in an ambiguous and highly redundant manner. The high dimensional, highly redundant encoding is a property of physics and tends to be true for sensor data (Lin et al., 2017). The (Markov) assumption that previous observations will not contain any information about the current state that is not also included in the current observation will not often hold on actual robotic tasks. However, it is (approximately) true for the tasks considered in this work and allows a focus on other challenges facing deep reinforcement learning.

Based on the observation o an action a is selected. The environment then transitions from state s to s', which is observed as o' and the reward r for the underlying transition is received. In addition, we will record whether the environment terminates after the transition with the terminal indicator $\mathbb{T}$, which is 1 for terminal states and 0 otherwise. The interactions with the environment result in experience tuples $\{o, a, o', r, \mathbb{T}\}$.

Our aim is to come up with a policy that maps observations to actions in a way that maximizes (2.2). In this thesis we consider deterministic policies: $a = \pi(o)$. Stochastic policies can have advantages in exploration, robustness and convergence

stability and can be optimal in certain settings (Haarnoja et al., 2017). However, in this thesis we are looking at the effects of data distributions, representation learning and optimization procedures on deep reinforcement learning. To isolate these effects, we prefer the simpler deterministic algorithms.

To learn policies we will make use of value functions (Sutton and Barto, 2018). These functions give the expectation of (2.1) under a specific policy. We start here by defining value functions (and policies) based on the true states $s \in \mathcal{S}$ of the system. In the next section we will address the fact that we do not presume access to these states, but only to their corresponding observations o. Two value functions are used:

- the state value function $V^\pi(s)$ is defined as the expected value of the return when starting from state s and following policy π:

$$V^\pi(s) \doteq \mathbb{E}_\pi\Big[\sum_{k=0}^{K} \gamma^k r_k | s_{k=0} = s\Big], \tag{2.3}$$

- the state-action value function $Q^\pi(s, a)$ is defined as the expected value of the return from state s when taking action a for the first time step and following the policy π afterwards:

$$Q^\pi(s, a) \doteq \mathbb{E}_\pi\Big[\sum_{k=0}^{K} \gamma^k r_k | s_{k=0} = s, a_{k=0} = a\Big] \tag{2.4}$$

To learn value functions, estimates of the return are needed. Different estimators for the return from a state exist. On opposite ends of the bias-variance spectrum we have the Monte Carlo estimator: by simply following the policy from a given state and calculating the return with the collected reward samples, an unbiased sample of the return distribution is obtained. However, as every transition might be stochastic, this estimator can have very high variance. If, on the other hand, we already have an estimator for the value function, we can use the recursive property of the definition of value functions. For instance:

$$q^\pi(s, a) = r + \gamma \hat{Q}^\pi(s', \pi(s')). \tag{2.5}$$

Note here that we use capitals for functions and lower case notation for (point) estimates. This estimator does not suffer from the high variance of the Monte Carlo estimator, since evaluating $Q^\pi(s', \pi(s'))$ provides an estimate of the expected value of the return distribution from s' rather than a sample from the distribution. However, since the estimate stems from the function that is being learned, it is

almost certainly biased. This idea of using the function that is to be learned for generating part of its own learning targets is known as bootstrapping.

With samples from an MDP, it is possible to form estimators for the expected value of the return of the policy that generated the data (an on-policy estimator), or for another policy. In this thesis the focus is on off-policy methods: methods that use samples from an arbitrary policy to estimate the value function of the optimal policy. Our interest in these methods stems from the fact that they (theoretically) allow for better sample efficiency. When the policy changes, older samples can still be used. Additionally, samples from other agents or controllers that are known to work could theoretically be used. In practice, learning becomes difficult when the sample distribution that would be induced by the policy that is to be learned differs too much from the sample distribution that it should be learned from. We investigate these considerations in detail in Chapters 3 and 4.

Most commonly, the samples of the MDP that are used to learn the value functions from are obtained by exploration policies. These policies $\tilde{\pi}$ tend to be stochastic variants of the policy π that is being learned. The aim of these policies is to obtain more diverse samples of the MDP than the policy π would. The reason for this need for diversity is twofold. First, to learn the optimal policy, the value functions need to be accurate for those states that the optimal policy visits. The current policy π might not be optimal yet, so it might not sample those states. By adding stochasticity to the policy, the probability of visiting states that correspond to a better policy than π is increased. Second, as will be investigated in Chapter 3, sufficient sample diversity is crucial when combining RL with deep neural networks, even for (near) optimal π.

Three types of exploration policies are used in this thesis:

- epsilon-greedy: at every time step, $a = \pi(s)$ with probability $(1 - \epsilon)$. With probability ϵ, $a \sim U(\mathcal{A})$,
- additive noise: $a_k = \pi(s_k) + \mathcal{O}_k$, where $\mathcal{O}_k$ is a noise process that might be temporally correlated,
- parameter noise: In Chapter 6 we will use a form of exploration that adds noise to the policy parameters rather than the actions. This causes the exploration to be state dependent.

To learn a policy, two basic strategies are used in this thesis. The first is to use a state-action value function, and take the action associated with the highest expected return value:

$$\pi(s) = \arg\max_a Q(s, a) \tag{2.6}$$

For discrete actions, (2.6) can be easily evaluated. When using continuous actions, finding the maximum becomes more involved. The second strategy is therefore to use an explicitly parameterized policy. Estimates of the return (such as value functions or Monte Carlo estimates) can then be used to optimize the parameters of the policy using the policy gradient theorem (Schulman et al., 2015b; Silver et al., 2014; Sutton et al., 2000). While this section has discussed some of the basic ideas behind reinforcement learning, Section 2.3.2 will discuss concrete algorithms.

2.2 | Deep Learning

Much of the original theory of RL was developed in a tabular setting, where the values and actions for every state can be stored explicitly. When the number of states becomes too large, for instance when the state-space is continuous, it becomes infeasible to sample, store, and compute the values of all states. Therefore, the value functions and optimal policies will need to be approximated by functions that are able to generalize to some extent across the state-space. Worse still, in our robotic setting, we do not assume to have access to the true state of the environment at all. Instead, we have access to sensory observations of a subset of relevant states of the environment, which tend to be high dimensional and include a lot of redundancy. The high dimensional nature of the observations means that local function approximation will not work without a proper distance measure, which we are unlikely to have access to (Friedman et al., 2001).[1] Although feature engineering could be used to extract the relevant aspects of the state from the sensor data, this defeats the purpose of using RL; we would like to have a general algorithm that requires minimal prior knowledge about the task that needs to be solved.

Therefore, we require some function that can learn to approximate the value functions (2.3) and (2.4) from the observations corresponding to the states. This will require the function approximator to not just learn a mapping from (representations of) states to return estimates, but also a mapping from observations to a representation of these states. Although we will investigate explicitly learning this mapping of observations to state representations in Chapter 5, this mapping is generally learned implicitly as part of the mapping from observations to return estimates or actions.

The mapping from observations to return estimates or actions requires a global, expressive function approximator that can be trained in a statistically efficient

[1] According to the manifold hypothesis, the observations will be clustered around low(er) dimensional nonlinear manifolds within the high-dimensional observation space (Goodfellow et al., 2016). While this could enable local function approximation on these manifolds, calculating distances between points on the manifold requires knowing the shape of the manifold.

manner. There are many different function approximators to choose from, and all make some assumptions about the functions that need to be approximated. Neural Networks (NNs) make only smoothness assumptions and, as a consequence, are able to represent any smooth function arbitrarily well given enough parameters (Hornik, 1991), making them a very general approximator option. However, without assumptions in addition to smoothness, it is impossible to learn to approximate certain complex functions in a statistically efficient manner (Bengio et al., 2006). The most important additional assumption made in Deep Neural Networks (DNNs) is that the function that needs to be approximated can be composed of a hierarchy of simpler functions (Goodfellow et al., 2016). This assumption is expressed through the architecture of DNNs, which have multiple hidden layers that compute nonlinear transformations of the outputs of previous layers. This decomposability assumption has proven very useful, especially when learning functions of natural data—which we assume our sensory observations to be—such as images, sounds and languages. This is due to the fact that this compositional structure is found all throughout the physical processes that generate these natural data (Lin et al., 2017).

Since we consider the challenge of learning to control from sensory observations, we will use DNNs as functions approximators in this work. The notation for the DNN approximations of value functions and policies we use in this thesis will be as follows. For a state-action value function approximation we write $\hat{Q}^{\pi}(o, a; \theta)$, where θ is the parameter vector of the network. When the corresponding policy and parameter vector are clear from the context, we will abbreviate this to: $\hat{Q}(o, a)$. For the policy we will write: $\pi(o; \theta)$ which will similarly be abbreviated to $\pi(o)$ when this does not lead to ambiguity.

The combination of RL with DNN function approximation is known as Deep Reinforcement Learning (DRL). This sub-field has already shown impressive results, such as achieving super-human performance on the game of Go, which until recently was believed to require human intuition (Silver et al., 2016). It is however important to realize that the assumptions behind DNNs do not always hold and that they do come at a price. We outline the assumptions, the opportunities they offer and the potential pitfalls of combining RL with DNNs in Section 2.2.1. In Section 2.3.1, we describe common strategies to deal with the challenges of DRL, while Section 2.3.2 gives an overview of popular DRL algorithms and how they implement these solutions. Section 2.3.3 describes ways in which the opportunities provided by the DNN assumptions can be exploited further.

2.2.1 Opportunities and pitfalls

In order to decide whether using a DNN as a function approximator is a good idea, and to realize the potential when one is used, it is important to be aware of the consequences stemming from the assumptions underlying deep learning.

Universal function approximation

The use of a universal function approximator, which can theoretically approximate any smooth function arbitrarily well, makes it possible to learn complex nonlinear policies and value functions. The combination of RL with DNNs gives a very general algorithm. However, this does mean that the space of possible functions is very large, making the optimization problem of finding a good set of parameters difficult. When more is known about the properties of the function that needs to be approximated, including this knowledge and thereby reducing the search space can be very beneficial. Although additional assumptions might introduce a bias in the learned function, they might also make the problem of learning the function tractable. Additionally, the use of a universal function approximator makes it more likely to over-fit to the training data. Rajeswaran et al. (2017) showed how, on a set of benchmarks often used to test DRL algorithms, RL with simpler function approximators learned faster and resulted in more robust policies, as the neural network policies over-fitted on the initial state distribution and did not work well when initialized from different states.

Stochastic gradient descent

A number of optimization techniques could be used to fit the parameters of a neural network (e.g. evolutionary strategies, Koutník et al., 2013; Salimans et al., 2017). However, the large number of parameters in most neural networks combined with the reliance on functional composition mean that first-order gradient methods are by far the most popular choice in practice. These techniques calculate an estimate of the first-order gradient of the cost function with respect to all of the network parameters. In the simplest case, the parameters are simply adjusted slightly in the (opposite) direction of the gradient, although often techniques are used that incorporate momentum and adaptive learning rates per parameter such as *RMSprop* (Tieleman and Hinton, 2012) and *Adam* (Kingma and Ba, 2015).

Neural networks can learn in a statistically efficient way because their parameters can apply globally and the decomposition into functions of functions allows the efficient reuse of parameters (Carter et al., 2019). While this allows for the generalization of a policy to unexplored parts of the state-space, it also means that the gradient estimates should be representative of the entire state-action space and not biased towards any particular part of it. Therefore, gradient estimates

are usually averaged over individual gradients computed for a *batch* of experiences spread out over the state-space. Subsequent gradient estimates should similarly be unbiased; they should be independent and identically distributed (i.i.d.) over the relevant state-action space distribution. When the gradient estimates suffer from high variance (as is the case for Monte-Carlo estimates of the policy gradient), they should be averaged over a larger batch to get a more reliable estimate.

Even when using adaptive learning rates and momentum, popular DNN optimization techniques still base their parameter updates on stochastic first order gradients estimates. Determining the right step size is problematic and sometimes the direction of the gradient is simply wrong. Combined with the highly nonlinear nature of DNNs, this means that some updates will inevitably have detrimental effects.

Functions of functions

The assumption that the function that needs to be approximated is composed of a hierarchy of simpler functions is encoded in DNNs by having multiple layers, with each layer computing a function of the outputs of the previous layer. The number of unique functions that the entire network can represent scales exponentially with the number of layers (Raghu et al., 2016) and the optimization of deeper networks has theoretically been shown to be less likely to result in a poor local optimum (Choromanska et al., 2015).

When determining the gradient of the loss function with respect to the parameters, the repeated multiplications with the derivative of a layer with respect to its inputs, resulting from the chain rule, can cause the gradients to become too large or small to effectively learn from. This problem is especially pronounced in recurrent neural networks, which are effectively very deep in time and repeatedly apply the same function (Hochreiter et al., 2001).

Complexity

DNNs have shown remarkable results in practice. The theoretical foundations are however still somewhat incomplete (Zhang et al., 2016). DRL lacks the theoretical guarantees offered by RL with some other types of function approximators. At the same time, it has been shown to scale to problems where the alternatives are intractable.

The complexity resulting from the interplay between the different components of DRL algorithms makes the learning curve fairly steep for beginning practitioners. Implementation details not mentioned in papers can have a more significant influence on the performance of a method than the parameters that are the focus of the work (Henderson et al., 2017; Tucker et al., 2018). The complexity of the

domains DRL is often tested on also contributes to a relatively high computational complexity. This means that DRL papers often include fewer repetitions of the experiments than are needed to get statistically significant results (Henderson et al., 2017).

2.3 | Deep Reinforcement Learning

The particularities of training deep neural networks mentioned previously mean that using DNNs as plug and play function approximators in standard RL algorithms is unlikely to result in more than disappointment. Section 2.3.1 will discuss some of the ideas needed to make deep reinforcement learning work in practice. Some of the DRL algorithms implementing these ideas will be discussed in Section 2.3.2. Of course, the combination of RL and DNNs also leads to specific opportunities, some of which are briefly mention in Section 2.3.3.

2.3.1 Common solution components

A substantial number of DRL algorithms have been proposed recently. These algorithms all have to address the challenges that arise from combining RL with DNNs. More specifically, these algorithms need to provide sufficiently diverse and independent samples from the observation space and corresponding stable yet accurate training targets. Based on this, the network parameters need to be updated in an efficient manner, without changing the policy so drastically that the obtained samples and learned functions are no longer valid.

To do this, most DRL methods are based on a few shared ideas. While most of these ideas and the problems they address are not limited to RL with DNN function approximation, they have proven crucial for making DRL work.

Delayed targets

To learn to approximate a value function with a DNN, the difference between the network predictions and estimates of the returns is minimized. When the return estimates are obtained through bootstrapping, this means that the predictions and the training targets are highly correlated, especially for high values of γ. While this is true for other forms of function approximation as well, it is especially problematic for DNNs for at least two reasons. The first is that parameter updates can have a global effect; a poor update in one part of the state space can affect the predictions—and therefore the training targets—in other parts of the state space as well. Secondly, the large number of parameters of DNNs means that it is usually only feasible to obtain a stochastic estimate of the first order parameter gradient. Combined with the highly nonlinear nature of DNNs this can mean that a parameter update can sometimes have large unforeseen consequences. These

effects combined with the direct feedback of predictions to training targets can quickly cause the learning process to diverge (Mnih et al., 2015). To ameliorate this problem, the target (return estimate) values can be calculated using an older version of the (action) value function network, often called a target network. This reduces the correlation and allows the network to update the predictions towards a more stable target, providing a chance to recover from bad updates.

Trust region updates

To limit the detrimental effects of too large steps in the parameter space, small learning rates can help. However, the resulting increase in training time and the required amount of training samples mean that preventing problems in this manner is often infeasible in practice. The problems are especially pronounced for on-policy methods relying on roll-outs, where the gradients additionally exhibit high variance. Changes to the policy can quickly change the distribution of states visited by the updated policy away from the on-policy distribution for which the update was valid.

To increase the likelihood of the updates to the policy resulting in improved performance, the changes in the policy distribution should therefore be kept small, while still maximizing the improvement in the parameter space. Several schemes have been proposed to prevent the changes to the parameters of the policy from resulting in too large changes to the policy distribution. These include adding a constraint on the policy distribution change to the optimization (Schulman et al., 2015a), clipping the objective function such that only small changes to the policy distribution are considered beneficial (Schulman et al., 2017), and constraining the policy parameters to be close to the running average of previous policies (Wang et al., 2017).

n-step returns

A problem that is inherent to bootstrapping methods is that they result in biased updates since the targets are based largely on an approximation of a function that should still be learned and is therefore by definition incorrect. This bias can prevent these methods from converging. On the other hand, Monte-Carlo based methods, while unbiased, suffer from high variance. This is because the return calculated for each roll-out trajectory represents only a single sample from the return distribution, while value functions represent the expectation of the return distribution.

On the complex domains that DRL is often applied to, the high variance of Monte-Carlo based methods tends to result in learning that is infeasibly slow. At the same time, the bias of methods based exclusively on learning value functions through

bootstrapping results in learning that can sometimes be faster, while other times failing to learn anything useful at all. A common strategy therefore is to interpolate between these extremes, for instance by using n-step algorithms (Watkins, 1989). To estimate the return from a certain state, these algorithms use the true rewards observed during n time-steps and the learned value estimate for the state at time step $n+1$. For instance, an n-step on-policy action value return estimate becomes:

$$q(o_k, a_k) = r_k + \gamma r_{k+1} + \cdots + \gamma^{n-1} r_{k+n-1} + \gamma^n \hat{Q}^{\bar{\pi}}(o_{k+n}, a_{k+n}) \tag{2.7}$$

n-step returns are related to eligibility traces (Sutton and Barto, 2018). The n-step return is preferred in DRL because it tends to be easier to use with momentum based optimization and recurrent neural networks (Mnih et al., 2016).

Just like the use of target networks, the use of n-step return targets reduces the correlations between the value function that is being learned and the optimization targets. Whereas the use of targets networks slows down the learning process in order to obtain the convergence gains, the use of n-step returns can speed up learning, provided the policy that obtained the trajectories is close to the policy for which the return should be estimated.

Experience replay

One of the largest mismatches between the RL framework and the stochastic gradient descent optimization algorithms used to train DNNs is the requirement of the latter for i.i.d. estimates of the gradients. This requirement can be (approximately) satisfied by using an experience replay buffer. The consecutive, strongly correlated experiences obtained through interaction with the environment are saved to the buffer. When batches of experiences are needed to estimate the gradients, these batches are assembled by sampling from the buffer in a randomized order, breaking the temporal correlations. The fact that off-policy algorithms can learn about the optimal policy from data obtained by another policy means that a fairly large amount of previous experiences can be retained. This in turn means that even if the policy changes suddenly, the data distribution used to calculate the gradients changes only slowly, which aids the stability (and therefore the convergence properties) of the optimization process. Finally, the fact that old experiences can be reused aids the sample efficiency of algorithms using an experience replay buffer $\mathcal{B}$. Extensions to this basic idea have been proposed, with the most popular being to replace uniform sampling from the buffer with sampling based on a distribution determined by the temporal difference error associated with the experiences (Schaul et al., 2016). By sampling surprising experiences more often, the learning process can be sped up significantly. This is similar to the classical idea of prioritized sweeping (Moore and Atkeson, 1993).

When using n-step returns with $n > 1$, it is necessary to compensate for the fact that the samples are not from the policy for which we want to estimate the return. Importance sampling is a popular choice that prevents bias (Precup et al., 2000). The downside of importance sampling is that when the difference between the policies is large, the importance weights quickly become either very small—effectively rendering the sampled experiences useless—or very large—resulting in updates with very high variance. Other compensation strategies that ameliorate these issues have been proposed, see Munos et al. (2016) for an overview.

When an on-policy learning algorithm is used, a buffer can be filled with experiences from roll-outs with the policy. After a learning update based on these experiences has been performed, the buffer is emptied and the process is repeated.

Input, activation and output normalization

The nonlinearities used in neural networks bound the outputs of the neurons to a certain range. For instance, the popular Rectified Linear Unit (ReLU) maps all non-positive inputs to zero. As a consequence, when calculating the derivatives of these nonlinearities with respect to their inputs, this derivative can be very small when the input is outside of a certain range. For the ReLU, the derivative of the activation with respect to all parameters that led to the activation is zero when the input to the ReLU was non-positive. As a consequence, none of the parameters that led to the activation will be updated, regardless of how sub-optimal the activation was. It is therefore important that the inputs to all neural network layers (whether they be the input to the network or the ouputs of previous layers) are within a sensible range.

When the properties of the inputs are unknown a priori and they cannot be normalized manually, adaptive normalization can be used. These techniques can also be used on subsequent layers. Normalization techniques include batch normalization (Ioffe and Szegedy, 2015), layer normalization (Ba et al., 2016) and weight normalization (Salimans and Kingma, 2016).

Similar considerations apply to the backward pass through a network during training. The gradients of the loss with respect to the parameters should not be too large, as an update based on large gradients can quickly cause the subsequent activations of the unit with the updated parameters to be outside of the range for which learning works well. Particularly, this means that while the scale of the reward function does not necessarily influence the stability of other forms of RL, DRL algorithms can be quite sensitive to this property (Henderson et al., 2017).

To ensure that the parameter gradients are within a sensible range, these gradients are often clipped. This changes the optimization objective, but prevents destructive updates. Additionally, when learning value functions, the reward function

can be scaled such that the resulting value function is of a sensible order of magnitude. Rewards are also sometimes clipped, although this changes the problem definition. Finally, the target values can be adaptively normalized during learning (van Hasselt et al., 2016).

2.3.2 Popular DRL algorithms

In this section we will discuss some of the more popular or historically relevant algorithms for deep reinforcement learning. These algorithms all address the challenges of performing RL with (deep) neural network function approximation by combining implementations of some of the ideas outlined in the previous section.

Neural Fitted Q iteration (NFQ)

An important early development in achieving convergent RL with neural network function approximation was the Neural Fitted Q iteration (NFQ) algorithm (Riedmiller, 2005). The algorithm uses a fixed experience buffer of previously obtained interaction samples from which to sample randomly. By calculating the target Q-values for all states at the start of each optimization iteration, and keeping them fixed throughout an iteration, the optimization is further helped to converge. A final measure to aid convergence was to add artificial experience samples to the database at the goal states, where the returns for the optimal policies were known.

Deep Q-network (DQN)

While good for convergence, the need for an a priori fixed set of experiences is limiting. While new experiences can be added to the NFQ buffer once a training iteration has completed, Mnih et al. (2015) proposed to continuously write experiences to an experience replay, and to sample experiences uniformly at random from this buffer at regular environment interaction intervals. Since the constant changes to the contents of the buffer and the learned Q-function mean that good targets cannot be calculated a priori, a copy θ^- of the Q-function parameters θ is kept in memory. The optimization targets (q-estimates) are calculated using a target network, which is a copy of the Q-function network using these older parameters θ^-. At regular intervals the target network parameters θ^- are updated to be equal to θ. Mnih et al. (2015) demonstrated their method using raw images as inputs. Their convolutional Deep Q-Network (DQN) achieved super-human performance on a number of Atari games, resulting in growing interest in the field of DRL.

The base DQN algorithm is simple to implement. Through various extensions, DQN can achieve competitive performance on domains with discrete actions (Hessel et al., 2017). Because of its simplicity and popularity, we use DQN in several chapters of this thesis. We therefore present the pseudo code in Algorithm 1.

Algorithm 1 DQN

Require: DNN value function estimator $\hat{Q}(o, a; \theta)$
Require: Exploration policy $\tilde{\pi}(\cdot, \hat{Q})$
Require: Parameter optimization algorithm
Require: Experience replay buffer $\mathcal{B}$
Require: Total number of training steps M
Require: Number of initial random interactions N_r
Require: Parameter update frequency U^{θ}
Require: Target parameter update frequency U^{-}
Require: Mini-batch size S

Randomly initialize θ
$\theta^{-} \leftarrow \theta$
while step $<$ M **do**
  $o \leftarrow$ reset environment, $\mathbb{T} \leftarrow 0$
  while $\mathbb{T} = 0$ **do**
    $a \leftarrow \tilde{\pi}(o; \theta)$ ▷ sample exploration action
    $o', r, \mathbb{T} \leftarrow$ environment step(a) ▷ execute action, observe transition
    store $\{o, a, o'r, \mathbb{T}\}$ in $\mathcal{B}$
    $o \leftarrow o'$
    step $\leftarrow$ step $+1$
    if step $> N_r$ **then**
      if step mod $U^{\theta} = 0$ **then**
        Sample a random batch of S transitions from $\mathcal{B}$
        for $i \in S$ **do**
          $q_i = r_i + (1 - \mathbb{T}_i) \cdot \gamma \max_{a'} \hat{Q}(o_i, a'; \theta^{-})$
        update θ to minimize $\mathcal{L}_{\mathcal{Q}} = \frac{1}{S} \sum_{i=1}^{S} (q_i - \hat{Q}(o_i, a_i; \theta))^2$
      if step mod $U^{-} = 0$ **then** ▷ $U^{-} \gg U^{\theta}$
        $\theta^{-} \leftarrow \theta$
return trained network $\hat{Q}(\cdot, \theta)$

Double DQN (DDQN)

Although methods that use bootstrapping are inherently biased, DQN suffers from a particular source of bias that can be reduced fairly easily. This form of bias is the overestimation of the returns which results from the maximization over the next step Q-values (line 15 of Algorithm 1). The max operator uses the same values to both select and evaluate the Q-values, which makes over-estimation of the values likely (Van Hasselt et al., 2016). To address this problem, the selection and evaluation can be decoupled. The original double Q-learning algorithm did this by learning two separate Q-functions, based on separate experiences (van Hasselt, 2010). One of these Q-functions is then used for the action selection while the other is used to determine the Q-value for that action. The Double Deep Q Network (DDQN) algorithm (Van Hasselt et al., 2016) uses the two separate Q-functions parameterized by θ and θ^- for the separation, such that the complexity of the algorithm is not increased. As in DQN, the target network is used to determine the value of the Q-function used for bootstrapping, while the on-line network is used to determine for which action the target Q-function is evaluated. This makes the optimization targets (line 15 of Algorithm 1):

$$q(o,a) = r + (1 - \mathbb{T}) \cdot \gamma \hat{Q} \left(o', \arg\max_{a'} \hat{Q}(o', a'; \theta); \theta^- \right). \tag{2.8}$$

This simple change was shown to improve the convergence and performance of the DQN algorithm.

Deep Deterministic Policy Gradient (DDPG)

For continuous action spaces, an actor-critic algorithm exists that is closely related to DQN. This Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap et al., 2016) uses a deterministic policy $a = \hat{\pi}(o; \theta_\pi)$, which is trained to (approximately) select those actions that maximize $\hat{Q}$. For convergence, target network copies of both the actor and the critic are used for the critic's optimization targets:

$$q(o,a) = r + (1 - \mathbb{T}) \cdot \gamma \hat{Q} \left(o', \pi(o'; \theta_\pi^-); \theta_q^- \right). \tag{2.9}$$

In this algorithm the target network parameters θ_q^-, θ_π^- slowly track the online parameters θ_q, θ_π using a low pass filter. They are updated after each optimization step according to:

$$\begin{aligned} \theta_q^- &\leftarrow (1-\tau)\theta_q^- + \tau\theta_q \\ \theta_\pi^- &\leftarrow (1-\tau)\theta_\pi^- + \tau\theta_\pi, \end{aligned}$$

with $\tau \ll 1$. To calculate the gradients for updating the policy parameters, the algorithm uses samples of the deterministic policy gradient (Silver et al., 2014):

$$\nabla_{\theta_\pi} J \approx \frac{1}{S} \sum_{i=1}^{S} \nabla_a \hat{Q}(o_i, \pi(o_i; \theta_\pi); \theta_q) \nabla_{\theta_\pi} \hat{\pi}(o_i; \theta_\pi). \tag{2.10}$$

The DDPG method additionally uses batch normalization layers (Ioffe and Szegedy, 2015).

DDPG is one of the simpler DRL algorithms allowing for continuous action spaces. Since the algorithm is off-policy, it additionally allows for experience replay, which together with the use of bootstrapping can lead to sample efficient learning. However, as Henderson et al. (2017) empirically show, its off-policy nature makes DDPG most suitable for domains with stable dynamics. Additionally, the bias in the policy gradient due to the exclusive reliance on a learned value function can limit the performance and convergence of the algorithm. As this algorithm is used in this thesis, the pseudo code is presented in Algorithm 2.

Trust Region Policy Optimization (TRPO)

While DDPG uses an off-policy critic to determine the policy gradient for a deterministic policy, Schulman et al. (2015a) introduced a policy-gradient method on the other end of the bias-variance spectrum. Their Trust Region Policy Optimization (TRPO) algorithm uses a large number of roll-outs with the current policy to obtain state-action pairs with Monte-Carlo estimates of their returns $q(o, a, \theta_\pi^-)$. The stochastic policy is then updated by optimizing for the conservative policy optimization objective (Kakade and Langford, 2002), while constraining the difference between the policy distribution after the optimization and the older policy distribution used to obtain the samples:

$$\max_{\theta_\pi} \mathbb{E} \left\{ \frac{\pi(a|o; \theta_\pi)}{\pi(a|o; \theta_\pi^-)} q(o, a; \theta_\pi^-) \right\} \tag{2.11}$$

$$\text{subject to } \mathbb{E} \left\{ D_{KL}(\pi(\cdot|o; \theta_\pi^-) \,||\, \pi(\cdot|o; \theta_\pi)) \right\} \leq c \tag{2.12}$$

where D_{KL} denotes the Kullback-Leibler divergence, and the expectations are with respect to the state distribution induced by the old policy. To perform the optimization, a linear approximation is made to the objective and a quadratic approximation is made to the constraint. The conjugate-gradient method is then used, followed by a line search, to calculate the next parameter values. The TRPO method is relatively complicated and sample inefficient, but does provide relatively reliable improvements to the policy.

Algorithm 2 DDPG

Require: DNN value function estimator $\hat{Q}(o, a; \theta_q)$
Require: DNN policy $\hat{\pi}(o; \theta_\pi)$
Require: Exploration policy $\tilde{\pi}(\cdot, \pi)$
Require: parameter optimization algorithm
Require: Experience replay buffer $\mathcal{B}$
Require: Total number of training steps M
Require: Number of initial random interactions N_r
Require: Target-parameters update-speed parameter τ
Require: Mini-batch size S

Randomly initialize θ_q, θ_π
$\theta_q^- \leftarrow \theta_q$, $\theta_\pi^- \leftarrow \theta_\pi$
while step < M **do**
    $o \leftarrow$ reset environment, $\mathbb{T} \leftarrow 0$
    while $\mathbb{T} = 0$ **do**
        $a \leftarrow \tilde{\pi}(o; \theta_\pi)$ ▷ sample exploration action
        $o', r, \mathbb{T} \leftarrow$ environment step(a) ▷ execute action, observe transition
        store $\{o, a, o' r, \mathbb{T}\}$ in $\mathcal{B}$
        $o \leftarrow o'$
        step $\leftarrow$ step $+1$
        if step $> N_r$ **then**
            Sample a random batch of S transitions from $\mathcal{B}$
            for $i \in S$ **do**
                $q_i = r_i + (1 - \mathbb{T}_i) \cdot \gamma \hat{Q}\left(o_i', \pi(o_i'; \theta_\pi^-); \theta_q^-\right)$
            update θ_q to minimize $\mathcal{L}_\mathcal{Q} = \frac{1}{S} \sum_{i=1}^{S} (q_i - \hat{Q}(o_i, a_i; \theta_q))^2$
            update θ_π using $\nabla_{\theta_\pi} J \approx \frac{1}{S} \sum_{i=1}^{S} \nabla_a \hat{Q}(o_i, \pi(o_i; \theta_\pi); \theta_q) \nabla_{\theta_\pi} \hat{\pi}(o_i; \theta_\pi)$
            $\theta_q^- \leftarrow (1 - \tau)\theta_q^- + \tau\theta_q$
            $\theta_\pi^- \leftarrow (1 - \tau)\theta_\pi^- + \tau\theta_\pi$
return trained policy $\hat{\pi}(\cdot, \theta_\pi)$

Generalized Advantage Estimation (GAE)

The stochastic policy gradient can be written as (Schulman et al., 2015b):

$$\nabla_{\theta_\pi} J = \mathbb{E}\{\sum_{k=0}^{\infty} q_k \ \nabla_{\theta_\pi} \log \pi(o_k, a_k; \theta_\pi)\}, \tag{2.13}$$

where q_k is an estimate of the return when taking action a_k in state s_k and following the policy afterwards. A trade-off between the bias and variance of the policy gradient estimates can be made by choosing how much q_k is based on observed rewards versus a learned value estimate, as discussed in Section 2.3.1. Additionally, the variance of the policy gradient can be reduced by subtracting a baseline from the return estimate (Greensmith et al., 2004). A common and close to optimal choice for the baseline is the state-value function. This makes q_k a sample from the advantage function:

$$\text{Adv}(o, a) = Q(o, a) - V(o),$$

which represents the advantage of taking action a in the state leading to observation o as opposed to the policy action $\pi(o)$. An n-step estimate of the advantage function is:

$$\text{adv}^{(n)}(o_k, a_k) = \sum_{i=0}^{n-1} \gamma^i r_{k+i} + \gamma^n \hat{V}(o_{k+n}; \theta_V) - \hat{V}(o_k; \theta_V) \tag{2.14}$$

To more optimally trade off the bias introduced by the imperfect learned value function for low n with the variance of estimators with high n, Schulman et al. (2015b) define a Generalized Advantage Estimator (GAE) as an exponentially weighted average of n-step advantage estimators:

$$\text{adv}^{GAE(\lambda)} := (1 - \lambda) \sum_{n=0}^{N} (\lambda^{n-1} \text{adv}^{(n)}) \tag{2.15}$$

The authors use the estimator with the TRPO algorithm. The value function is learned from Monte-Carlo estimations with trust-region updates as well.

Proximal Policy Optimization (PPO)

The constrained optimization of TRPO makes the algorithm relatively complicated and prevents using certain neural network architectures. In the Proximal Policy Optimization (PPO) algorithm, Schulman et al. (2017) therefore replace the hard constraint by a clipped version of the objective function, which ensures that for each state the potential gain from changing the state distribution is limited, while the potential loss is not. This makes it possible to optimize for the objective

(which uses the GAE) with SGD-based techniques, as well as to add additional terms to the objective. Specifically, a regression loss for the value function is added, which enables parameter sharing between the value function and the policy. Additionally, a loss based on the entropy of the policy is added to encourage exploration (Williams and Peng, 1991). PPO is a relatively simple algorithm that offers competitive performance.

Asynchronous Advantage Actor Critic (A3C)

Instead of collecting a large number of consecutive on-policy trajectories with a single policy, which are then batched together, Mnih et al. (2016) proposed the use of a number of parallel actors with global shared parameters. These actors all calculate updates with respect to the shared parameters, which they apply to the parameters asynchronously (Recht et al., 2011). To ensure the actors explore different parts of the state-action space so that the parameter updates better meet the i.i.d. assumption, each agent uses a different exploration policy. While a number of proposed algorithms benefited from the parallel actor setup, the most successful was the Asynchronous Advantage Actor Critic (A3C) algorithm. This algorithm takes a small number of steps, after which it calculates n-step advantage estimates (2.15) and value function estimates for these roll-out steps. These are then used to calculate gradients to update the policy (2.13) and the value function.

Actor Critic with Experience Replay (ACER)

The downside of the on-policy methods (TRPO, PPO, A3C) is that once a step has been made in policy space, re-evaluating the policy gradient requires discarding all previous experiences and running trials with the new policy. To increase the sample efficiency, it is desirable to combine the favorable convergence properties of the on-policy algorithms with the sample efficiency of off-policy algorithms.

One algorithm that does this is the Actor Critic with Experience Replay (ACER) algorithm of Wang et al. (2017). It uses the A3C algorithm as a base and combines it with a trust-region update scheme based on limiting the distance between the new policy parameters and those of a running average of recent policies. It then alternates between the standard on-policy updates of A3C and off-policy updates, where each parallel agent samples trajectories from a local experience buffer for the updates. Truncated importance sampling with a bias correction term is used to correct for the off-policy nature of the n-step trajectories. While the algorithm offers very competitive performance for both discrete and continuous actions, it is relatively complex.

Interpolated Policy Gradient (IPG)

Another way in which on-policy and off-policy algorithms can be combined is to simply interpolate between the biased yet sample efficient deterministic policy gradient obtained from an off-policy critic (2.10) and the unbiased yet sample inefficient on-policy Monte Carlo estimate of the policy gradient. This Interpolated Policy Gradient (IPG) method was proposed by Gu et al. (2017b), who found intermediate (but mostly on-policy) ratios to work best.

2.3.3 Extensions

The DRL algorithms discussed before mostly address the pitfalls of combining RL with DNNs. However, the use of DNNs also offers opportunities to go beyond simply performing RL with DNN function approximation. The functional decomposition of DNNs means that while later layers might compute very task specific features, earlier layers could represent much more general functions. For example, while later layers in a convolutional network might learn to recognize task specific objects, earlier layers might learn to detect edges or textures (Olah et al., 2017). These earlier layers might therefore easily generalize to new tasks and, equivalently, be trained through different objectives and using data obtained from separate tasks. Therefore, the deep learning assumptions make the combination of DRL with transfer learning and state-representation learning very interesting.

State representation learning

As discussed previously, learning a policy from observations can be seen as a combination of learning a mapping from the sensor data to a representation of the state of the environment as well as a subsequent mapping from the state representation to the actions. While the state representation can be learned implicitly through DRL, the number of required trial and error samples might be prohibitively expensive as the reward signal might contain only very indirect information on how to learn the state representation.

Instead, explicit State Representation Learning (SRL) objectives can be used before or during the RL phase. These objectives can enable learning from unlabeled sensor data, as well as limiting the parameter search space through the inclusion of prior knowledge. Auto-encoding is a popular SRL objective as it is fully unsupervised; through a compression objective salient details are extracted from observations that are highly redundant (Finn et al., 2016; Hinton and Salakhutdinov, 2006; Lange et al., 2012b). Besides the knowledge that observations are highly redundant, other priors include the fact that the state of the world only changes slowly over time (Wiskott and Sejnowski, 2002), as well as the fact that the state

should be predictive of immediate received rewards (Shelhamer et al., 2016). Additional priors relevant to physical domains were suggested by Jonschkowski and Brock (2015). Besides encoding general knowledge about the state of the world, it is possible to learn to encode the observations in a way that is suitable for control. One example is the work of Watter et al. (2015), which embeds images into a state-space in which actions have a (locally) linear effect. Another example is provided by Jonschkowski et al. (2017) who learned to encode the positions and velocities of relevant objects in an unsupervised manner. Jaderberg et al. (2017) proposed to learn, off-policy, additional value functions for optimizing pseudo rewards based on controlling the sensory observations and the activations of the neurons of the networks. The inclusion of SRL objectives in DRL can help learn representations and policies that generalize to unseen parts of the state-space more easily. We investigate this, and the question of how to combine SRL and RL in a stable manner in Chapter 5.

Transfer learning

Just as the generality of the functions encoded by the earlier layers of the policy and value function DNNs means that they can be trained with more than just RL updates, and generalize to unseen parts of the state-space, it also means that the encoded functions can be relevant to more than just the training task. This makes DRL suitable for transfer learning, where generalization needs to be performed to a new task, rather than just across the state-space of the training task. In this context, Parisotto et al. (2015) used DQN agents trained on several Atari games as teachers for a separate DQN agent that was trained to output similar actions, and have similar internal activations as the teachers. This was found to result in a weight initialization that sped up learning on new games significantly, given enough similarity between the new games and some of the training games. It is also possible to more explicitly parameterize representations for transfer. Universal Value Functions (UVFs) (Schaul et al., 2015) are one example where value functions are learned that generalize over both states and goal specifications. To improve the performance in domains where only reaching a goal results in obtaining a reward, Andrychowicz et al. (2017) proposed Hindsight Experience Replay (HER), which relabels a failed attempt to reach a certain goal as a successful attempt to reach another goal. Another representation that is suitable for transfer learning is the Successor Features (SF) representation (Barreto et al., 2017) which is based on successor representations (Dayan, 1993). These representations decouple the value function into a representation of the discounted state distribution induced by the policy and the rewards obtained in those states. Zhang et al. (2017) showed the use of this representation with DRL in the robotics domain.

Supervised policy representation learning

Sometimes the state of the environment is available for specific training cases, but not in general. For instance, a robot might be placed in a motion capture arena. In this case, it might be relatively simple to learn or calculate the correct actions for the states in the arena. Alternatively, it might be possible to solve the RL problem from specific initial states, but hard to learn a general policy for all initial states. In both of these scenarios, trajectories of observations and actions can be collected and supervised learning can be used to train DNN policies that generalize to the larger state-space, preventing many of the issues of DRL. One technique that applies this principle is Guided Policy Search (GPS), which adds a constraint to the local controllers on the deviation from the global policy, such that the local policies do not give solutions that the DNN can not learn to represent (Levine et al., 2016; Levine and Koltun, 2013).

3

The Effect of the Experience Buffer Contents

Parts of this chapter have previously been published in:

de Bruin, T., Kober, J., Tuyls, K., Babuška, R. (2018). *"Experience selection in deep reinforcement learning for control"*. the Journal of Machine Learning Research (JMLR).

de Bruin, T., Kober, J., Tuyls, K., Babuška, R. (2015). *"The importance of experience replay database composition in deep reinforcement learning"*. Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems (NIPS), 2015.

3.1 | Introduction

For any machine learning system, the data that are used to train the system are of paramount importance for its performance. In the application of supervised and unsupervised learning algorithms alike it is crucial to collect a training dataset with a distribution of examples that is representative of the distribution of inputs that is expected during the operation of the system. While generalization in deep neural networks is not well understood (Zhang et al., 2016), it is well known that relatively large amounts of diverse examples from the domain of the function that is to be learned are required. Besides these requirements on the training data set as a whole, the stochastic-gradient algorithms used for the optimization of the neural network parameters additionally require the data to be presented in an independent and identically distributed (i.i.d) order.

Reinforcement learning is a somewhat unusual form of machine learning, as the training dataset is generally not collected a priori. Instead, reinforcement learning algorithms receive a temporally correlated stream of data, the distribution of which is a direct function of the training process. Here too, sufficient coverage of the function domain is crucial. Not just to ensure that the learned function generalizes, but also to discover good policies to begin with. In RL this gives rise to a large body of work on advanced exploration strategies and the exploration-exploitation dilemma. To facilitate the use of neural network function approximation, the stream of experiences can be directed towards a buffer from which the experiences can be sampled in an order that breaks their temporal correlations. This reduces the violations of the i.i.d requirement of the stochastic gradient methods used to train the networks.

When applying deep reinforcement learning methods to robotics, there are constraints on the kind of policies that can be executed—and therefore the data that can be gathered. Due to the danger involved, it might not be possible to perform extensive exploration or even apply a policy directly that has not been sufficiently trained. Even when rigorous exploration *is* temporarily possible, it is often necessary to reduce the exploration intensity over time to prevent wear and to focus the newly gathered data around the sensitive equilibria that are often of most interest when controlling dynamical systems.

Given the possible limitations on the policies that can be executed on robots, as well as the expensive nature of robotic experience samples, we investigate off-policy reinforcement learning in this thesis. In this context we investigate the effects of the data distribution on deep reinforcement learning. In this chapter, we start by investigating how different aspects of control problems influence the need for different distributions of samples over the state-action space. In the next chapter

we will use this knowledge to propose several strategies for selecting experiences, from the stream of experiences that is observed, for training. These experience selection methods are developed to address specific problems stemming from the combination of reinforcement learning and deep learning under the constraints imposed in the robotics domain.

3.2 | Motivating example

We start by motivating the focus on the effects of the data distribution when combining reinforcement learning with deep neural network function approximation under robotic constraints. To do so, a reference tracking controller for a 2-link robot arm is learned using DDPG (Lillicrap et al., 2016, see Section 2.3.2 and Algorithm 2). The arm is depicted in Figure 3.1. It consists of two links that are connected through a motorized joint. The arm hangs down from another motorized joint. A detailed description of the control task is given in Appendix A, with algorithmic implementation details given in Appendix B. The control problem entails learning to move the end of the arm to reference positions by determining the voltages to the motors based on a low-dimensional observation.

Certain control signals, especially those with strong high-frequency components, have the potential to significantly increase wear on the system (Koryakovskiy et al., 2017). We therefore used both a low-pass filter on the applied control signals and the temporally-correlated Ohrnstein-Uhlenbeck noise process (detailed in Appendix A and Appendix B respectively). The observation of the system included the control voltages at the preceding time step to prevent the filtered control signal from making the problem non-Markovian.

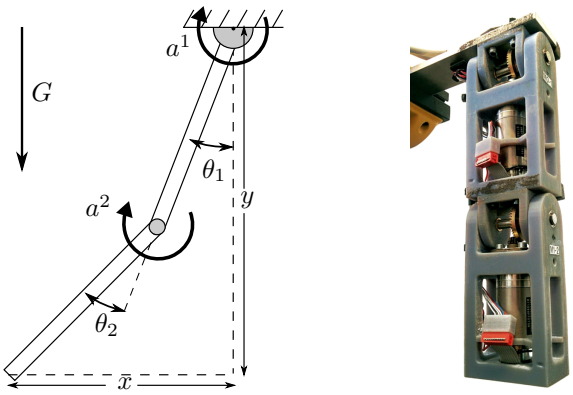


Figure 3.1: 2-link arm robot schematic (left) and photo (right).

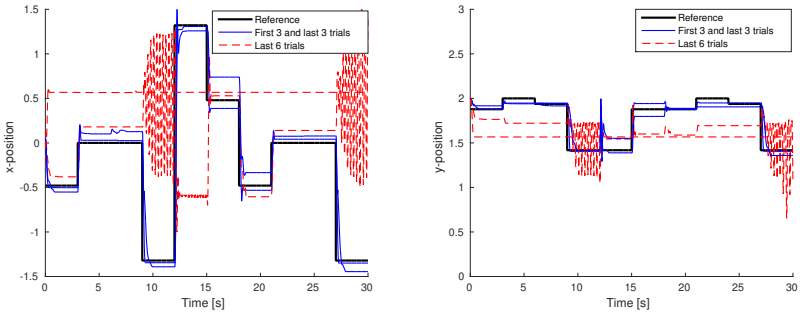


Figure 3.2: Effects of (not) keeping early samples in the experience buffer on a physical system. For both retention strategies, two policies were trained for 40 episodes of 30 seconds. An experience buffer that can hold experiences from 6 episodes was used. Keeping early experiences in memory can be seen to significantly improve the quality of the learned controller.

Despite these precautions, the exploration noise was still observed to cause increased wear on the system[1]. We therefore decay the magnitude of the exploration signal exponentially with every subsequent episode. This means that the experiences obtained during the initial exploratory phase, in which data is collected that is spread more uniformly over the state-action space, will quickly be outnumbered by experiences centered more closely around the policy. When a buffer is used that is large enough to contain all samples, uniform sampling will asymptotically reduce the importance of the initial samples to zero. When a smaller buffer is used that is overwritten in a First In First Out (FIFO) manner, the experiences will be overwritten within finite time.

To clearly demonstrate the continuing importance of the initial exploratory samples, we perform experiments with a small experience buffer that can hold 6 trials worth of experiences. We consider two strategies for overwriting this buffer. In the first, we simply overwrite the buffer in a FIFO manner. In the second method, only the second half of the buffer is overwritten, leaving the initial exploratory experiences in the buffer indefinitely.

The policies that resulted after 40 episodes were tested on the same sequence of reference positions. The response of the system in the four tests is shown in Figure 3.2. It can be seen that although the two learning runs which kept the initial episodes in the database produced useful policies, the method completely failed on both runs where the six most recent episodes were kept in memory instead.

[1]The more aggressive movements during exploration quickened the loosening of a screw on the arm. The gradual loosening altered the system dynamics and when the screw became too loose the arm was not accurately controllable.

These experiments motivate an investigation into which experiences should be in the experience buffer for deep reinforcement learning to work well. This investigation will be performed in simulation, as this allows for reproducible experiments, sufficient repetitions for statistically significant results and the exclusion of confounding factors such as screw looseness.

3.3 | Related work

When a learning system needs to learn a task from a set of examples, the order in which the examples are presented to the learner can be very important. One method to improve the learning performance on complex tasks is to gradually increase the difficulty of the examples that are presented. This concept is known as shaping (Skinner, 1958) in animal training and curriculum learning (Bengio et al., 2009) in machine learning.

Sometimes it is possible to generate training examples of just the right difficulty on-line. Recent machine learning examples of this principle include generative adversarial networks (Goodfellow et al., 2014) and self play in reinforcement learning (see for example the work by Silver et al. 2017). When the training examples are fixed, learning can be sped up by repeating those examples that the learning system is struggling with more often than those that it finds easy, as was shown for supervised learning by, among others, Hinton (2007) and Loshchilov and Hutter (2015). Additionally, the eventual performance of supervised-learning methods can be improved by re-sampling the training data proportionally to the difficulty of the examples, as done in the boosting technique (Freund et al., 1999; Valiant, 1984)

In on-line reinforcement learning, a set of examples is generally not available to start with. Instead, an agent interacts with its environment and observes a stream of experiences as a result. The experience replay technique was introduced to save those experiences in a buffer and replay them from that buffer to the learning system (Lin, 1992). The introduction of an experience buffer makes it possible to choose which examples should be presented to the learning system again. As in supervised learning, we can replay those experiences that induced the largest error (Schaul et al., 2016). Another option that has been investigated in the literature is to replay experiences that are associated with large immediate rewards more often (Narasimhan et al., 2015).

In off-policy reinforcement learning the question of which experiences to learn from extends beyond choosing how to sample from a buffer. It begins with determining which experiences should be in the buffer. Lipton et al. (2016) fill the buffer with

successful experiences from a pre-existing policy before learning starts. Other authors have investigated criteria to determine which experiences should be retained in a buffer of limited capacity when new experiences are observed. In this context, Pieters and Wiering (2016) have investigated keeping only experiences with the highest immediate rewards in the buffer, while this chapter will investigate the importance of sufficient diversity in the state-action space (see also de Bruin et al., 2016a,b).

Experience replay techniques, including those in this thesis, often take the stream of experiences that the agent observes as given and attempt to learn from this stream in an optimal way. Other authors have investigated ways to instill the desire to seek out information that is useful for the learning process directly into the agent's behavior (Bellemare et al., 2016a; Chentanez et al., 2004; Houthooft et al., 2016; Osband et al., 2016; Schmidhuber, 1991). Due to the classical exploration-exploitation dilemma, changing the agents behavior to obtain more informative experiences comes at the price of the agent acting less optimally according to the original reward function.

A safer alternative to actively seeking out *real* informative but potentially dangerous experiences is to learn, at least in part, from *synthetic* experiences. This can be done by using an a priori available environment model such as a physics simulator (Barrett et al., 2010; Rusu et al., 2016), or by learning a model from the stream of experiences itself and using that to generate experiences (Caarls and Schuitema, 2016; Gu et al., 2016; Kuvayev and Sutton, 1996; Sutton, 1991). The availability of a generative model still leaves the question of *which* experiences to generate. Prioritized sweeping bases updates again on surprise, as measured by the size of the change to the learned functions (Andre et al., 1997; Moore and Atkeson, 1993). Ciosek and Whiteson (2017) dynamically adjusted the distribution of experiences generated by a simulator to reduce the variance of learning updates.

Learning a model can reduce the sample complexity of a learning algorithm when learning the dynamics and reward functions is easy compared to learning the value function or policy. However, it is not straightforward to get improved performance in general. In contrast, the introduction of an experience replay buffer has shown to be both simple and very beneficial for many deep reinforcement learning techniques (Gu et al., 2017a; Lillicrap et al., 2016; Mnih et al., 2015; Wang et al., 2017). When a buffer is used, we can decide which experiences to have in the buffer and which experiences to sample from the buffer. In contrast to previous work on this topic we investigate, in the next chapter, the combined problem of experience retention and sampling. We also look at several different proxies for the usefulness of experiences and how prior knowledge about the specific reinforcement learning problem at hand can be used to choose between them, rather than attempting to

find a single universal experience-utility proxy. These proxies will be based on the examinations in this chapter.

3.4 | Experimental Benchmarks

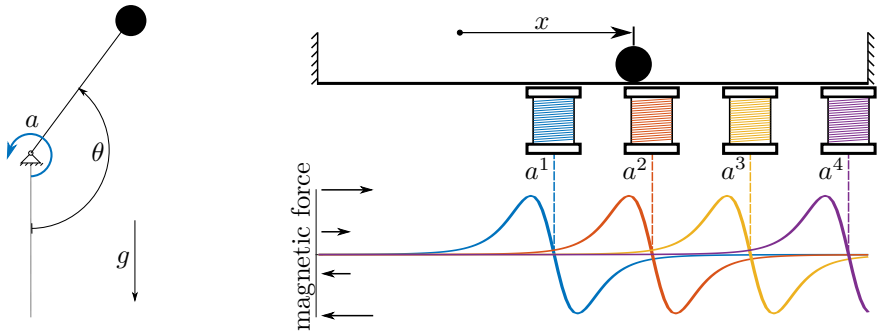
In this section, we discuss two simulated control tasks with limited observation and action dimensionalities. The low dimensional nature of these problems makes it possible to understand and visualize the effects of different data distributions and makes it feasible to perform statistically significant experiments. We test our findings on benchmarks with higher dimensional observation and action spaces in the next chapter.

The simulations are of a pendulum swing-up task and a magnetic manipulation problem. Both were previously discussed by Alibekov et al. (2018). Although both represent dynamical systems with a two dimensional state-space, it will be shown in Section 3.6 that they are quite different when it comes to the optimal experience selection strategy. Here, a high level description of these benchmarks is presented, while the full mathematical descriptions are given in Appendix A.

The first task is the classic underactuated pendulum swing-up problem, shown in Figure 3.3a. The pendulum starts out hanging down under gravity. The goal is to balance the pendulum in the upright position. The motor is torque limited such that a swing to one side is needed to build up momentum before swinging towards the upright position in the opposite direction. Once the pendulum is upright it needs to stabilize around this unstable equilibrium point. The observation of the problem o consists of normalized versions of the angle θ and angular velocity $\dot{\theta}$ of the pendulum. The action space is a normalized version of the voltage applied to the motor that applies a torque to the pendulum. A reward is given at every time-step, based on the absolute distance of the observed system state from the reference state of being upright with no rotational velocity.

The second benchmark is a magnetic manipulation (*magman*) task, in which the goal is to accurately position a steel ball on a 1-D track by dynamically changing a magnetic field. The relative magnitude and direction of the force that each magnet exerts on the ball is shown in Figure 3.3b. This force is linearly dependent on the actions, which represent the squared currents through the electromagnet coils. Normalized versions of the position x and velocity $\dot{x}$ form the observation-space of the problem. A reward is given at every time-step, based on the absolute distance of the observed system state from the (fixed) reference state.

In experiments where the buffer capacity $\mathcal{C}$ is limited, we take $\mathcal{C} = 10^4$ experiences, unless stated otherwise. All our experiments have episodes which last four seconds.



(a) Pendulum task **(b)** Magnetic manipulation task

Figure 3.3: Two of the benchmark problems considered in this thesis. In the pendulum task, an underactuated pendulum needs to be swung up and balanced in the upright position by controlling the torque applied by a motor. In the magnetic manipulation (magman) task, a steel ball (top) needs to be positioned by controlling the currents through four electromagnets. The magnetic forces exerted on the ball (bottom) can be seen to be a nonlinear function of the position. The forces scale linearly with the actions $a^1, ..., a^4$, which represent the normalized squared currents through the magnets.

Unless stated otherwise, a sampling frequency of 50 Hz is used, which means the buffer can store 50 episodes worth of experience tuples $\langle o_i, a_i, o'_i, r_i \rangle$.

Since we are especially interested in physical control problems where sustained exhaustive exploration is infeasible, the amount of exploration is reduced over time from its maximum at episode 1, to a minimum level from episode 500 onwards unless stated otherwise. At the minimum level, the amplitude of the exploration noise we add to the neural network policy is 10% of the amplitude at episode 1. Details of the exploration strategies used are given in Appendix B.

3.5 | Performance Measures

When we investigate the performance of the learning methods in this chapter and the next, we are interested in the effect that these methods might have on *three* aspects of the learning performance: *the learning stability, the maximum controller performance* and *the learning speed*. We define performance metrics for these aspects, related to the normalized mean reward per episode μ_r. The normalization is performed such that $\mu_r = 0$ is the performance achieved by a random controller, while $\mu_r = 1$ is the performance of the off-line dynamic programming method described in Appendix B.2.3. This baseline method is, at least for the noise-free tests, proven to be close to optimal.

The first learning performance aspect we consider is the *stability* of the learning process. We will be shown in this chapter (see also de Bruin et al., 2015, 2016a) that even when a good policy has already been learned, the learning process can become unstable and the performance can drop significantly when the properties of the training data change. We investigate to what extent different experience replay methods can help prevent this instability. We use the mean of μ_r over the last 100 episodes of each learning run, where the learning runs should have converged to good behavior already, as a measure of learning stability. We denote this measure by μ_r^{final}.

Although changing the data distribution might help stability, it could at the same time prevent us from accurately approximating the true optimal policy. Therefore we also report the *maximum performance* achieved per learning trial $\mu_r^{\max}$.

Finally, we want to know the effects of the experience selection methods on the *learning speed*. We therefore report the number of episodes before the learning method achieves a normalized mean reward per episode of $\mu_r = 0.8$ and denote this by Rise-time 0.8.

For these performance metrics we report the means and the 95% confidence bounds of those means over 50 trials for each experiment. The confidence bounds are based on bootstrapping (Efron, 1992).

3.6 | Main Contribution: Analysis of Experience utility

As previously noted by Narasimhan et al. (2015); Pieters and Wiering (2016); Schaul et al. (2016) and de Bruin et al. (2015, 2016a), when using experience replay, the criterion that determines which experiences are used to train the reinforcement learning agent can have a large impact on the performance of the method. The aim of this section is to investigate what makes an experience useful and how this usefulness depends on several identifiable characteristics of the control problem at hand.

In the following sections, we mention only some relevant aspects of our implementation of the deep reinforcement-learning methods, with more details given in Appendix B.

3.6.1 Exploration Decay

As mentioned in Section 3.2, it can be necessary to reduce the amount of exploration that is performed over time to prevent wear on physical systems. Here we investigate the effect this has on the two simulation benchmarks introduced in Section 3.4. Over the first 500 episodes of every trial, the strength of the exploration noise (B.1) is linearly decayed to a fraction of the initial strength. The results

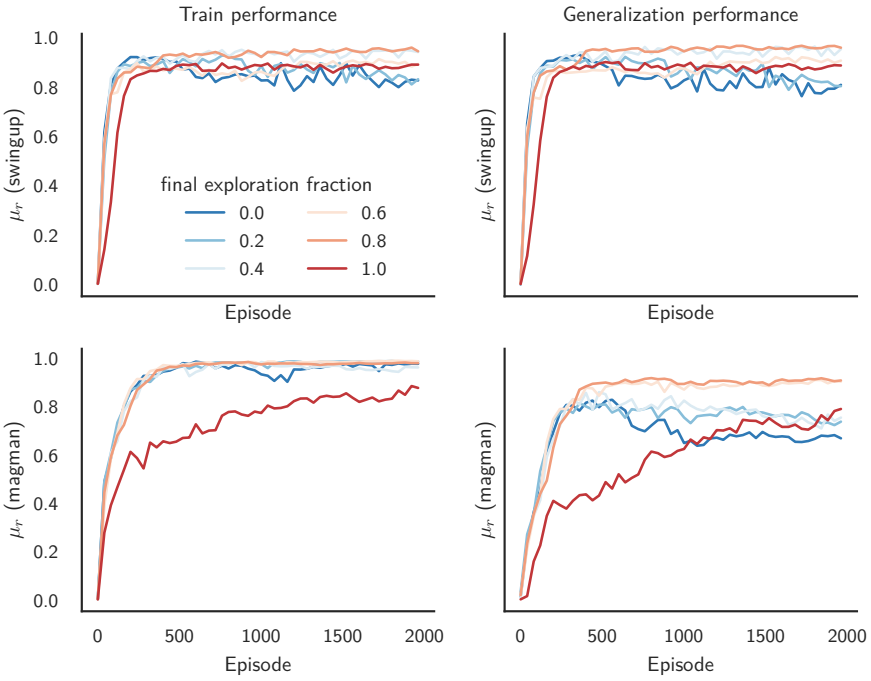


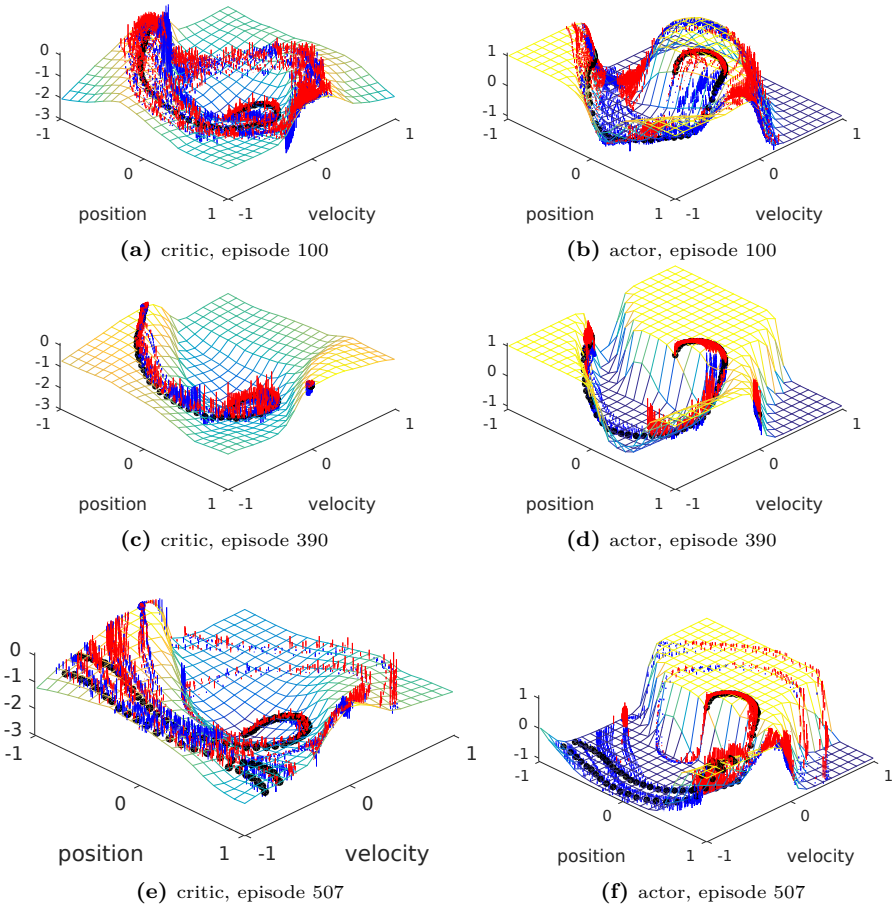
Figure 3.4: The effect of decaying the strength of the exploration noise. Only the means of 50 runs are shown for legibility. For the generalization tests, a larger set of initial states was used than during training.

are shown in Figure 3.4. The results show that for both benchmarks, decaying the exploration at least somewhat is necessary for good results. However, especially on the pendulum task, decaying the exploration too much results in a loss of performance over time. On the magman, this decay is only observed for the generalization tests.

3.6.2 Generalizability and Sample Diversity

One important aspect of the problem with reduced exploration, which at least partly explains the differences in training performance for the two methods on the two benchmarks in Figure 3.4, is the complexity of generalizing the value function and policy across the observation and action spaces.

For the pendulum task, learning actor and critic functions that generalize across the entire state and action spaces will be relatively simple, as a sufficiently deep neural network can efficiently exploit the symmetry in the value and policy functions (Montufar et al., 2014). Figure 3.5b shows the learned policy after 100



(a) critic, episode 100 **(b)** actor, episode 100

(c) critic, episode 390 **(d)** actor, episode 390

(e) critic, episode 507 **(f)** actor, episode 507

Figure 3.5: The critic $\hat{Q}\left(o, \hat{\pi}(o;\theta_\pi);\theta_Q\right)$ and actor $\hat{\pi}(o;\theta_\pi)$ functions trained on the pendulum swing up task. The surfaces represent the functions. The black dots show the trajectories through the state-action space resulting from following the current policy. The red and blue lines show respectively the positive and negative 'forces' that shape the surfaces caused by the experiences in the buffer: for the critic these are $\partial(q_i - \hat{Q}(o_i, a_i;\theta))^2/\partial\hat{Q}$ (note $a_i \neq \hat{\pi}(o_i;\theta_\pi)$). For the actor these forces represent $\partial\hat{Q}\left(o, \hat{\pi}(o;\theta_\pi);\theta_Q\right)/\partial a$. Animations of these graphs for different experience selection strategies are available at https://youtu.be/Hli1ky0bgT4. The episodes are chosen to illustrate the effect of reduced sample diversity described in Section 3.6.2.

episodes (where, as before, experiences are replaced in a Fist In First Out (FIFO) manner and sampling from the buffer is done uniformly at random). Due to the thorough initial exploration, the experiences in the buffer cover much of the state-action space. As a result, a policy has been learned that is capable of swinging the pendulum up and stabilizing it in both the clockwise and anticlockwise directions, although the current policy favors one direction over the other.

For the next 300 episodes this favored direction does not change and as the amount of exploration is decayed, the experiences in the buffer become less diverse and more centered around this favored trajectory through the state-action space. Even though the information on how to further improve the policy becomes increasingly local, the updates to the network parameters can cause the policy to be changed over the whole state space, as neural networks are global function approximators. This can be seen from Figure 3.5d, where the updates that further refine the policy for swinging up in the currently preferred direction have removed the previously obtained skill of swinging up in the opposite direction. The policy has suffered from *catastrophic forgetting* (Goodfellow et al., 2013) and has over-fitted to the currently preferred swing up direction.

For the pendulum swing up task, this over-fitting is particularly risky since the preferred swing up direction can and does change during learning, since both directions are equivalent with respect to the reward function. When this happens, the FIFO experience retention method can cause the data distribution in the buffer to change rapidly, which by itself can cause instability. In addition, the updates to the network parameters (calculated in lines 14-16 of Algorithm 2) now use the critic $\hat{Q}(o, a; \theta_Q)$ function in regions of the observation-action space that it has not been trained on in a while, resulting in potentially bad gradients. Both of these factors might destabilize the learning process. This can be seen in Figure 3.5f where, after the preferred swing up direction has rapidly changed a few times, the learning process is destabilized and the policy has deteriorated to the point that it no longer accomplishes the balancing task. By keeping a more diverse set of experiences in the buffer, this failure case can be prevented.

For the magman task, a policy that generalizes over the whole state-space might be harder to find. This is because the effects of the actions, shown as the colored lines in Figure 3.3b, are strongly nonlinear functions of the (position)-state. The actor and critic functions must therefore be very accurate for the states that are visited under the policy. Requiring the critic to explain all of the experiences that have been collected so far might limit the ability of the function approximators to achieve sufficient accuracy for the relevant states.

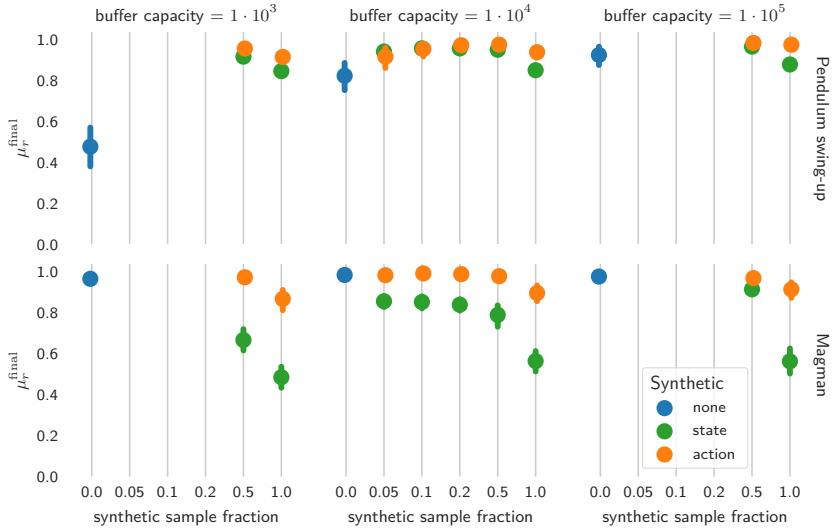


Figure 3.6: The effect on the mean performance during the last 100 episodes of the learning runs μ_r^{final} when changing a fraction of the observed experiences with synthetic experiences, for different buffer sizes. The addition of more diverse experiences can be seen to be most important for small buffers.

Buffer Size and Synthetic Sample Fraction

To further investigate its effects on the reinforcement learning performance, we will artificially alter the diversity of the experiences in the buffer. We will do so by performing the following experiment. We still decay the exploration and use an experience buffer of the same size, still overwritten in a FIFO manner with uniformly random sampling. However, with a certain probability, we make a change to an experience $\langle o_i, a_i, o_i', r_i \rangle$ before it is written to the buffer. We change either the observation o_i (and the corresponding underlying state) or the action a_i. The changed observations and actions are sampled uniformly at random from the observation and action spaces. When the observation is re-sampled the action is recalculated as the policy action for the new observation including exploration. In both cases, the next observation and the reward are recalculated to complete the altered experience. To calculate the next observation and reward, we use the real system model. This is not possible for most practical problems; it serves here merely as a way of gaining a better understanding of the need for sample diversity.

The results of performing this experiment for different sample alteration probabilities and buffer sizes are given in Figures 3.6 and 3.7. Interestingly, for the pendulum swing up task, changing some fraction of the experiences to be more

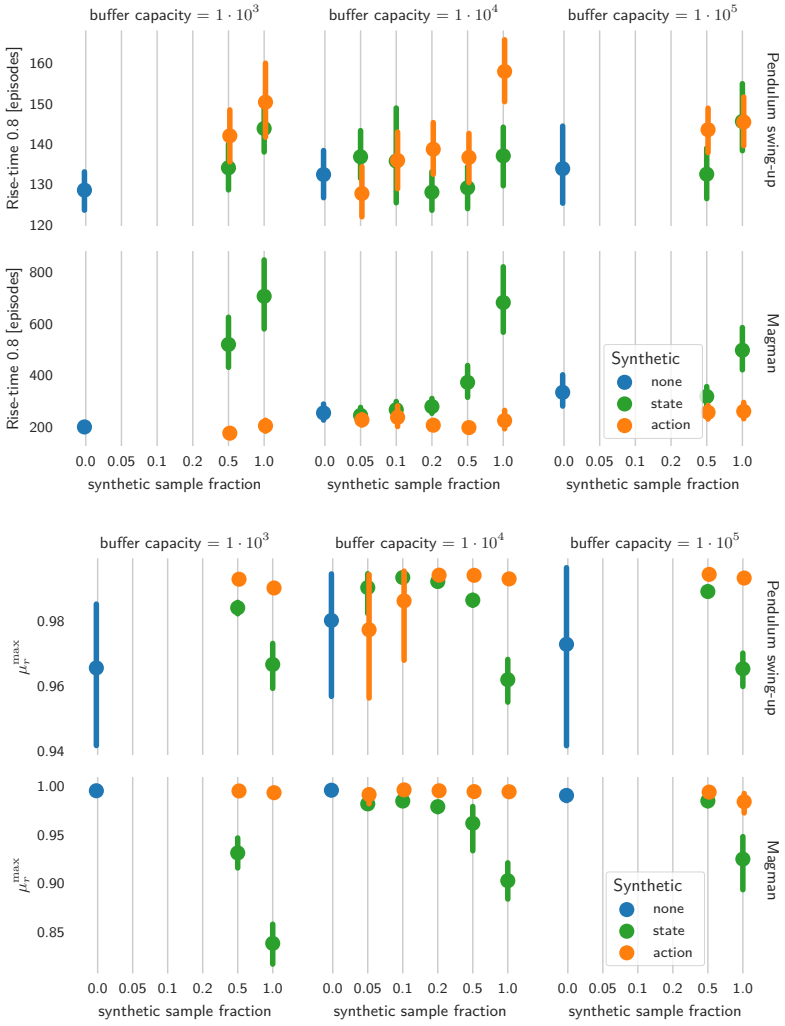


Figure 3.7: The effects on the learning performance when replacing a fraction of the observed experiences with synthetic experiences, for different buffer sizes. Increasing the diversity of the samples can be seen to increase the amount of training needed to find good policies, but also increase the eventual quality of the policy. The pendulum swingup task is most sensitive to the experience diversity. On the magman task, increased diversity in the state-space can limit the eventual performance.

diverse improves the stability of the learning method dramatically, regardless of whether the diversity is in the states or in the actions. The effect is especially noticeable for smaller experience buffers.

For the magman benchmark, as expected, having more diverse states reduces the performance significantly. Having a carefully chosen fraction of more diverse actions in the original states can however improve the stability and learning speed slightly. This can be explained from the fact that even though the effects of the actions are strongly nonlinear in the state-space, they are linear in the action space. Generalizing across the action space might thus be more straightforward and it is helped by having the training data spread out over this domain.

3.6.3 Reinforcement-Learning Algorithm

The need for experience diversity also depends on the algorithm that is used to learn from those experiences. In the rest of this chapter as well as the next one we exclusively consider the DDPG actor-critic algorithm, as the explicitly parameterized policy enables continuous actions, which makes it especially suitable for control. An alternative to using continuous actions is to discretize the action space. In this subsection, we compare the need for diverse data of the actor-critic DDPG algorithm (Lillicrap et al., 2016; Silver et al., 2014) to that of the closely related critic-only DQN algorithm (Mnih et al., 2015, see Section 2.3.2 and Algorithm 1). The experiments are performed on the pendulum benchmark, where the one dimensional action space is discretized uniformly into 15 actions. Results for the magman benchmark are omitted as the four dimensional action space makes discretization impractical.

For the actor-critic scheme to work, the critic needs to learn a general dependency of the Q-values on the observations and actions. For the DQN critic, this is not the case as the Q-values for different actions are separate. Although the processing of the observations is shared, the algorithm can learn at least partially independent value predictions for the different actions. These functions additionally do not need to be correct, as long as the optimal action for a given observation has a higher value than the sub-optimal actions. The requirement on the critic in the actor-critic algorithm is more stringent; there the derivative of the critic with respect to the actions needs to be correct.

These effects can be seen in Figure 3.8. The DDPG algorithm can make more efficient use of the observation-action space samples by learning a single value prediction, resulting in significantly faster learning than the DQN algorithm. The DDPG algorithm additionally benefits from more diverse samples, with the performance improving for higher fractions of randomly sampled observations or actions. The DQN algorithm conversely seems to suffer from a more uniform sampling of

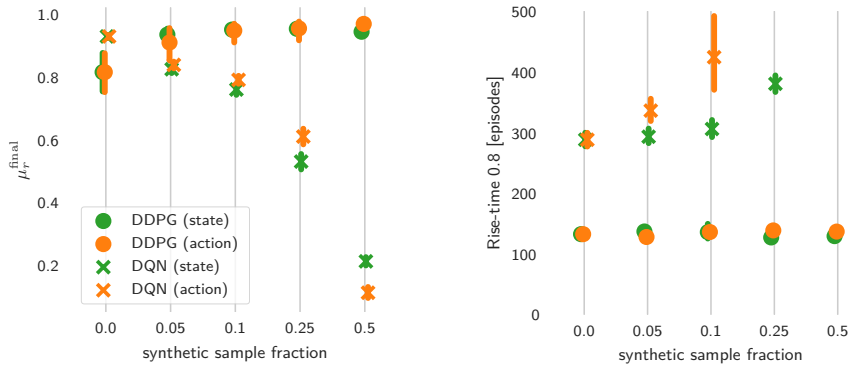


Figure 3.8: RL algorithm dependent effect of sample diversity on the pendulum benchmark. The effect on $\mu_r^{\max}$ is given in Figure C.4 of Appendix C. The benefits of increased diversity that were observed for the actor-critic learning DDPG algorithm are seen not to transfer to the critic only DQN algorithm.

the observation-action space. This could be because it is now tasked with learning accurate mappings from the observations to the state-action values for all actions. While doing so might not help to improve the predictions in the relevant parts of the observation-action space, it could increase the time required to learn the function and limit the function approximation capacity available for those parts of the observation-space where the values need to be accurate. Note again that learning precise Q-values for all actions over the whole observation-space is not needed, as long as the optimal action has the largest Q-value.

Sample Age

In the model-free setting it is not possible to add synthetic experiences to the buffer. Instead, in Chapter 4 we will introduce ways to select real experiences that have desirable properties and should be remembered for a longer time and replayed more often. This will inevitably mean that some experiences are used more often than others, which could have detrimental effects—such as that the learning agent could over-fit to those particular experiences.

To investigate the effects of adding older experiences for diversity, we perform the following experiment. As before, a FIFO buffer is used with a certain fraction of synthetic experiences. However, when a synthetic experience is about to be over-written, we only sample a new synthetic experience with a certain probability. Otherwise, the experience is left unchanged. The result of this experiment is shown in Figure 3.9. For the pendulum benchmark, old experiences only hurt when they were added to provide diversity in the action space in states that were

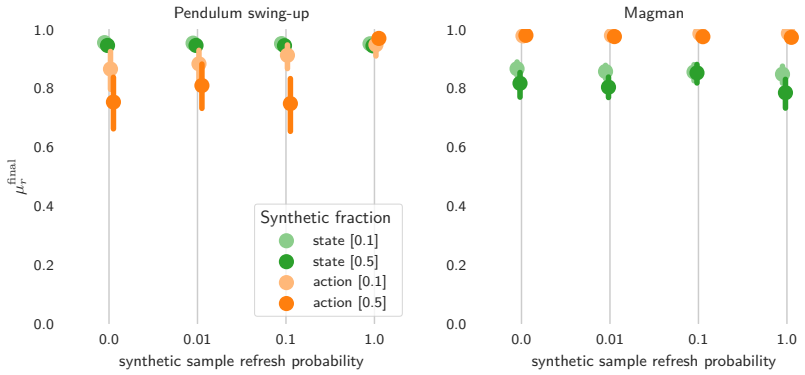


Figure 3.9: The effects of only updating the synthetic experiences with a certain probability each time they are overwritten. The effects on $\mu_r^{\max}$ and the rise-time are given in Figure C.6 in Appendix C. While over-fitting to individual samples does not seem to be an issue, keeping state-distribution current when adding synthetic actions is important on the pendulum benchmark.

visited by an older policy. For the magman benchmark the age of the synthetic experiences is not seen to affect the learning performance.

3.6.4 Sampling Frequency

An important property of control problems that can influence the need for experience diversity is the frequency at which the agent needs to produce control decisions. The sampling frequency of a task is something that is often considered a given property of the environment in reinforcement learning. For control tasks however, a sufficiently high sampling frequency can be crucial for the performance of the controller and for disturbance rejection (Franklin et al., 1998). At the same time, higher sampling frequencies can make reinforcement learning more difficult as the effect of taking an action for a single time-step diminishes for increasing sampling frequencies (Baird, 1994). Since the sampling rate can be an important hyperparameter to choose, we investigate whether changing it changes the diversity demands for the experiences to be replayed.

In Figure 3.10, the performance of the DDPG method is shown for different sampling frequencies, with and without synthetic samples. The first thing to note is that, as expected, low sampling frequencies limit the controller performance. Interestingly though, much of the performance loss on the pendulum at low frequencies can be prevented through increased sample diversity. This indicates that on this benchmark most of the performance loss at the tested control frequencies stems from the learning process rather than the fundamental control limitations. When increasing the sampling frequencies beyond our baseline frequency of 50Hz,

sample diversity becomes more important for both stability and performance. For the pendulum swing-up it can be seen that as sampling frequency increases further, increased diversity in the state/observation-space becomes more important. For the magman, adding synthetic action samples has clear benefits. This is very likely related to the idea that the effects of actions become harder to distinguish for higher sampling frequencies (Baird, 1994; de Bruin et al., 2016b).

There are several possible additional causes for the performance decrease at higher frequencies. The first is that by increasing the sampling frequency, we have increased the number of data points that are obtained and learned from per episode. Yet the amount of information that the data contains has not increased by the same amount. Since the buffer capacity is kept equal, the amount of information that the buffer contains has decreased and the learning rate has effectively increased. To compensate for these specific effects, experiments are performed in which samples are stochastically prevented from being written to the buffer with a probability proportional to the increase in sampling frequency. The results of these experiments are indicated with [DE] (dropped experiences) in Figure 3.11 and are indeed better, but still worse than the performance for lower sampling frequencies.

The second potential reason for the drop in performance is that we have changed the problem definition by changing the sampling frequency. This is because the forgetting factor γ determines how far into the future we consider the effects of our actions according to:

$$\gamma = e^{-\frac{T_s}{\tau_\gamma}},$$

where T_s is the sampling period in seconds and τ_γ is the lookahead horizon in seconds. To keep the same lookahead horizon, we recalculate γ, which is 0.95

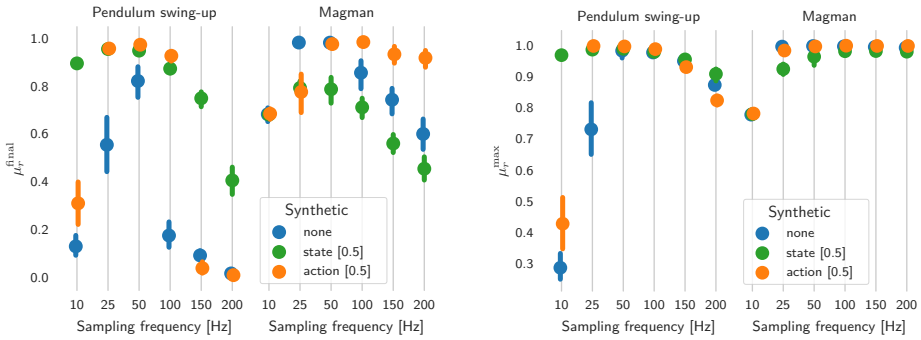


Figure 3.10: Sampling frequency dependent effect of adding synthetic experiences to the `FIFO[Uniform]` method. The effect on the rise time is given in Figure C.5 in Appendix C.

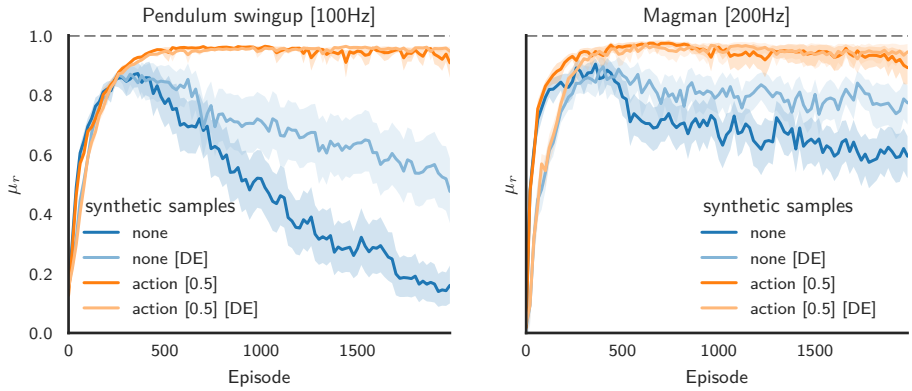


Figure 3.11: The effect of synthetic actions and stochastically preventing experiences from being written to the buffer [DE] on the benchmarks with increased sampling frequencies.

in our other experiments ($T_s = 0.02$), to be $\gamma_{\text{pendulum}} = 0.9747$ ($T_s = 0.01$) and $\gamma_{\text{magman}} = 0.9873$ ($T_s = 0.005$). To keep the scale of the Q functions the same, which prevents larger gradients, the rewards are scaled down. Correcting the lookahead horizon was found to hurt performance on both benchmarks. The likely cause of this is that higher values of γ increase the dependence on the biased estimation of Q over the unbiased immediate reward signal (see Equation (2.5)). This can cause instability (François-Lavet et al., 2015).

3.6.5 Noise

The final environment property that we consider is the presence of sensor and actuator noise. So far, the observations have been normalized versions of the exact environment state and the (de-normalized) chosen actions have been implemented without change. Now we consider adding Gaussian noise with a standard deviation $\sigma \in \{0, 0.01, 0.02, 0.05\}$ to both the normalized observations and actions. The results of performing these experiments are shown in Figure 3.12. The results indicate that the need for data diversity is not dependent on the presence of noise. However, in Section 4.6.3 it will be shown that the methods used to determine which experiences are useful *can* be affected by noise.

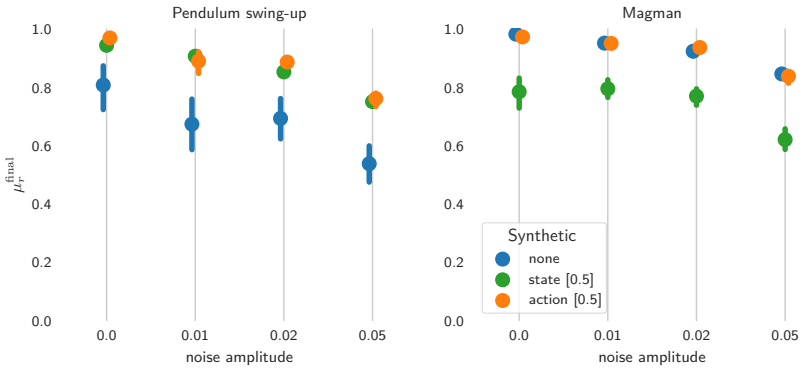


Figure 3.12: The effect of adding Gaussian noise to the observed states and applied actions on the need for diverse data.

3.7 | Summary

This chapter has presented an investigation into how different aspects of the reinforcement learning problem at hand influence the need for experience diversity. In Table 3.1 a summary is given of the investigated aspects and the strength of their effect on the need for experience diversity. While this chapter has used the true environment model to examine the potential benefits of diversity, the next chapter will propose strategies to obtain diverse experiences in ways that are feasible on real problems.

Property	Effect	Explanation
Benchmark	Very high	The need for diverse states and actions largely depends on the ease and importance of generalizing across the state-actions space, which is benchmark dependent.
RL algorithm	Very high	Generalizing across the action space is fundamental to actor-critic algorithms, but not to critic-only algorithms with discrete action spaces.
Sampling frequency	High	The stability of RL algorithms depends heavily on the sampling frequency. Experience diversity can help learning stability. Having diverse actions at higher frequencies might be crucial as the size of their effect on the observed returns diminishes.
Buffer size	Medium	Small buffers can lead to rapidly changing data distributions, which causes unstable learning. Large buffers have more inherent diversity.
Sample age	Low	Although retaining old samples could theoretically be problematic, these problems were not clearly observable the experiments in this chapter.
Noise	None	The presence of noise was not observed to influence the need for experience diversity, although it can influence experience selection strategies, as will be shown in the next chapter.

Table 3.1: The dependence of the need for diverse experiences on the investigated environment and reinforcement learning properties.

4

Experience Selection

Parts of this chapter have previously been published in:

de Bruin, T., Kober, J., Tuyls, K., Babuška, R. (2018). *"Experience selection in deep reinforcement learning for control"*. the Journal of Machine Learning Research (JMLR).

4.1 | Introduction

In the previous chapter we showed how the performance of deep reinforcement learning algorithms is influenced by the distribution of the experiences over the observation-action space. We investigated how several aspects of control problem at hand dictated a need for either more or less diverse experiences. We did this by assuming access to a perfect model of the environment, so that synthetic experiences could be generated. In this chapter we return to the standard model-free setting in which this is not the case. Instead, we assume that a stream of experiences, over which we might have little to no influence, is directed towards an experience buffer.

If we have access to a buffer with past experiences, an interesting question arises: how should we *sample* the experiences to be replayed from this buffer? It has been shown by Schaul et al. (2016) that a good answer to this question can significantly improve the performance of the reinforcement-learning algorithm.

However, even if we know how to sample from the experience buffer, two additional questions arise: what should the buffer capacity be and, once it is full, how do we decide which experiences should be *retained* in the buffer and which ones can be overwritten with new experiences? These questions are especially relevant when learning on systems with a limited storage capacity, for instance when dealing with high-dimensional inputs such as images. Finding a good answer to the question of which experiences to retain in the buffer becomes even more important when exploration is costly. This can be the case for physical systems such as robots, where exploratory actions cause wear or damage and risks need to be minimized (Garcıa and Fernández, 2015; Kober et al., 2013; Koryakovskiy et al., 2017; Tamar et al., 2016). It is also the case for tasks where a minimum level of performance needs to be achieved at all times (Banerjee and Peng, 2004) or when the policy that generates the experiences is out of our control (Schaal, 1999; Seo and Zhang, 2000).

We will refer to the combined problem of experience *retention* and experience *sampling* as experience *selection*. The questions of which experiences to sample and which experiences to retain in the buffer are related, since they both require a judgment on the utility of the experiences. The difference between them is that determining which experiences to sample requires a judgment on the instantaneous utility: from which experiences can the agent learn the most at the moment of sampling? In contrast, a decision on experience retention should be based on the expected long term utility of experiences. Experiences need to be retained in a way that prevents insufficient coverage of the state action space in the future, as experiences cannot be recovered once they have been discarded.

To know the true utility of an experience, it would be necessary to foresee the effects of having the reinforcement-learning agent learn from the experience at any given time. Since this is not possible, we instead investigate *proxies* for the experience utility that are cheap to obtain.

In this chapter, we investigate age, surprise (in the form of the temporal difference error), and the amplitude of the exploration noise as proxies for the utility of experiences. To motivate the need for *multiple* proxies, we will start by showing the performance of different experience selection methods on the pendulum and magman control benchmarks introduced in the previous chapter. We show how the current state-of-the-art experience selection method of Schaul et al. (2016), based on retaining a large number of experiences and sampling them according to their temporal difference error, compares on these benchmarks to sampling uniformly at random from the experiences of the most recent episodes. We show that the state-of-the-art method significantly outperforms the standard method on one benchmark while significantly *under*-performing on the other, seemingly similar benchmark.

The hardware limitations of the robotic systems that this thesis focuses on can impose constraints on the exploration policy and the number of experiences that can be stored in the buffer. These factors make the correct choice of experience sampling strategy especially important. As we show on additional, more complex benchmarks, even when sustained exploration *is* possible, it can be beneficial to be selective about which—and how many—experiences to retain in the buffer. The costs involved in operating a robot mean that it is generally infeasible to rely on an extensive hyper-parameter search to determine which experience selection strategy to use. We therefore want to understand how this choice can be made based on prior knowledge of the control task.

Note that for many of the experiments in this work most of the hyper-parameters of the deep reinforcement-learning algorithms are kept fixed. While it would be possible to improve the performance through a more extensive hyper-parameter search, our focus is on showing the relationships between the performance of the different methods and the properties of the control problems. While we do introduce new methods to address specific problems, the intended outcome of this work is to be able to make more informed choices regarding experience selection, rather than to promote any single method.

4.2 | Preliminaries

The experience replay techniques from the literature (some of which were briefly introduced in Chapter 2) will be discussed in more detail in this section. After-

wards, Section 4.3 will introduce the notation used to indicate experience selection strategies and Section 4.4 will demonstrate the need for multiple proxies. We will introduce our proposed methods in Section 4.5, experimentally test these methods in Section 4.6, and provide general recommendations for their use in Section 4.7.

4.2.1 Experience Replay

As discussed in Chapter 2 and Chapter 3, experience replay is crucial for making deep reinforcement learning work. The experience replay technique uses a buffer of past experiences, from which mini-batches are sampled for the paramter updates. Most commonly, experiences are written to the buffer in a First In First Out (FIFO) manner. When experiences are needed to train the neural networks, they are sampled uniformly at random from the buffer. This breaks the temporal correlations of the updates and restores the i.i.d. assumption of the optimization algorithms, which improves their performance (Mnih et al., 2015; Montavon et al., 2012). The increased learning stability that results comes in addition to the fact that experiences can be used multiple times for updates, increasing the sample efficiency of the algorithm.

4.2.2 Prioritized Experience Replay

Although sampling experiences uniformly at random from the experience buffer is an easy default, the performance of reinforcement-learning algorithms can be improved by choosing the experience samples used for training in a smarter way. Here, we summarize one of the variants of Prioritized Experience Replay (PER) that was introduced by Schaul et al. (2016). Our enhancements to experience replay are given in Section 4.5.

The PER technique is based on the idea that the temporal difference error

$$\delta_i = \hat{Q}(o_i, a_i; \theta_q) - q_i, \tag{4.1}$$

provides a good proxy for the instantaneous utility of an experience. Schaul et al. (2016) argue that, when the critic made a large error on an experience the last time it was used in an update, there is more to be learned from the experience. Therefore, its probability of being sampled again should be higher than that of an experience associated with a low temporal difference error.

In this work we consider the rank-based stochastic PER variant. In the method, the probability of sampling an experience i from the buffer is approximately given by:

$$P(i) \approx \frac{\left(\frac{1}{\text{rank}(i)}\right)^{\alpha}}{\sum_j \left(\frac{1}{\text{rank}(j)}\right)^{\alpha}}. \tag{4.2}$$

Here, rank(i) is the rank of sample i according to the absolute value of the temporal difference error $|\delta|$ according to (4.1), calculated when the experience was last used to train the critic. All experiences that have not yet been used for training have $\delta = \infty$, resulting in a large probability of being sampled. The parameter α determines how strongly the probability of sampling an experience depends on δ. We use $\alpha = 0.7$ as proposed by Schaul et al. (2016) and have included a sensitivity analysis for different buffer sizes in Appendix C. Note that the relation is only approximate as sampling from this probability distribution directly is inefficient. For efficient sampling, (4.2) is used to divide the buffer $\mathcal{B}$ into S segments of equal cumulative probability, where S is taken as the number of experiences per training mini batch. During training, one experience is sampled uniformly at random from each of the segments.

4.2.3 Importance Sampling

The estimation of an expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation. Schaul et al. (2016) proposed to compensate for the fact that the changed sampling procedure can affect the value of the expectation in (2.9) by multiplying the gradients of the critic with an Importance Sampling (IS) weight

$$\omega_i = \left(\frac{1}{\mathcal{C}}\frac{1}{P(i)}\right)^{\beta}. \tag{4.3}$$

Here, β allows scaling between not compensating at all ($\beta = 0$) and fully compensating for the changes in the sample distribution caused by the sampling strategy ($\beta = 1$). In our experiments, when IS is used, we follow Schaul et al. (2016) in scaling β linearly per episode from 0.5 at the start of a learning run to $\beta = 1$ at the end of the learning run. $\mathcal{C}$ indicates the capacity of the buffer.

Not all changes to the sampling distribution need to be compensated for. Since we use a deterministic policy gradient algorithm with a Q-learning critic, we do not need to compensate for the fact that the samples are obtained by a different policy than the one we are optimizing for (Silver et al., 2014). We can change the sampling distribution from the buffer, without compensating for the change, so long as these samples accurately represent the transition and reward functions.

Sampling based on the TD error can cause issues here, as infrequently occurring transitions or rewards will tend to be surprising. Replaying these samples more often will introduce a bias, which should be corrected through importance sampling.

However, the temporal difference error will also be partly caused by the function approximation error. These errors will be present even for a stationary sample

distribution after learning has converged. The errors will vary over the state-action space and their magnitude will be related to the sample density. Sampling based on *this* part of the temporal difference error will make the function approximation accuracy more consistent over the state-space. This effect might be unwanted when the learned controller will be tested on the same initial state distribution as it was trained on. In that case, it is preferable to have the function approximation accuracy be highest where the sample density is highest. However, when the aim is to train a controller that generalizes to a larger part of the state space, we might *not* want to use importance sampling to correct this effect. Note that importance sampling based on the sample distribution over the state space is heuristically motivated and based on function approximation considerations. The motivation does not stem from the reinforcement learning theory, where most methods assume that the Markov decision process is ergodic and that the initial state distribution does not factor into the optimal policy (Aslanides et al., 2017). In practice however, deep reinforcement-learning methods can be rather sensitive to the initial state distribution (Rajeswaran et al., 2017).

Unfortunately, we do not know to what extent the temporal difference error is caused by the stochasticity of the environment dynamics and to what extent it is caused by function approximation errors. We will empirically investigate the use of importance sampling in Section 4.6.4.

4.3 | Experience Selection Strategy Notation

We consider the problem of *experience selection*, which we have defined as the combination of *experience retention* and *experience sampling*. The experience retention strategy determines which experiences are discarded when new experiences are available to a full buffer. The sampling strategy determines which experiences are used in the updates of the reinforcement-learning algorithm. We use the following notation for the complete experience selection strategy: *retention strategy[sampling strategy]*. Our abbreviations for the retention and sampling strategies commonly used in deep RL that were introduced in Section 4.2 are given in Tables 4.1 and 4.2 respectively. The abbreviations used for the new or uncommonly used methods introduced in Section 4.5 are given there, in Tables 4.3 and 4.4.

Notation	Proxy	Explanation
`FIFO`	age	The oldest experiences are overwritten with new ones.
`FULL DB`	-	The buffer capacity $\mathcal{C}$ is chosen to be large enough to retain all experiences.

Table 4.1: Commonly used experience retention strategies for deep reinforcement learning.

Notation	Proxy	Explanation
`Uniform`	-	Experiences are sampled uniformly at random.
`PER`	surprise	Experiences are sampled using rank-based stochastic *prioritized experience replay* based on the temporal difference error. See Section 4.2.2.
`PER+IS`	surprise	Sampling as above, but with weighted *importance sampling* to compensate for the distribution changes caused by the sampling procedure. See Section 4.2.3.

Table 4.2: Experience sampling strategies from the literature.

4.4 | The Limitations of a Single Proxy

To motivate the need for multiple proxies for the utility of experiences rather than one universal proxy, we compare the performance of the two strategies from the literature on the benchmarks described in the previous chapter.

The first experience selection strategy tested is `FIFO[Uniform]`: overwriting the oldest experiences when the buffer is full and sampling uniformly at random from the buffer. We compare this strategy to the state-of-the-art prioritized experience replay method `FULL DB[PER]` by Schaul et al. (2016). Here, the buffer capacity $\mathcal{C}$ is chosen such that all experiences are retained during the entire learning run ($\mathcal{C} = N = 4 \times 10^5$ for this test).[1] The sampling strategy is the rank-based stochastic prioritized experience replay strategy as described in Section 4.2. The results of the experiments are shown in Figure 4.1.

Figure 4.1 shows that `FULL DB[PER]` method, which samples training batches based on the temporal difference error from a buffer that is large enough to contain all previous experiences, works well for the pendulum swing-up task. The method very reliably finds a near optimal policy. The `FIFO[Uniform]` method, which keeps only the experiences from the last 50 episodes in memory, performs much worse. As we demonstrated in the previous chapter, the performance degrades over time as the amount of exploration is reduced and the experiences in the buffer fail to cover the state-action space sufficiently.

If we look at the result on the magman benchmark in Figure 4.1, the situation is reversed. Compared to simply sampling uniformly from the most recent experiences, sampling from all previous experiences according to their temporal difference error limits the final performance significantly. As shown in Appendix C.1, this is not simply a matter of the function approximator capacity, as even much larger networks trained on all available data are outperformed by small networks trained on only recent data. When choosing an experience selection strategy for a reinforcement learning task, it seems therefore important to have some insights into how the characteristics of the task determine the need for specific kinds of experiences during training. We have investigated some of these characteristics in the previous chapter and will introduce experience selection methods based on the lessons learned in that chapter in the next section.

[1] Schaul et al. (2016) use a `FIFO` database with a capacity of 10^6 experiences. We here denote this as `FULL DB` since all our experiments use a smaller number of time-steps in total.

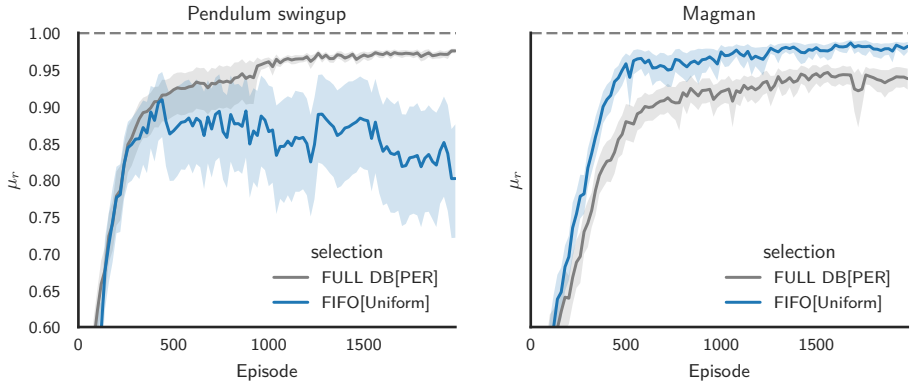


Figure 4.1: Comparison of the state-of-the-art (`FULL DB[PER]`) and the default method (`FIFO[Uniform]`) for experience selection on our two benchmark problems.

4.5 | Main Contribution: New Experience-Selection Strategies

For the reasons discussed in Section 3.3, we do not consider changing the stream of experiences that an agent observes by either changing the exploration or by generating synthetic experiences. Instead, to be able to replay experiences with desired properties, valuable experiences need to be identified, so that they can be retained in the buffer and replayed from it. In this section we look at how several proxies for the utility of experiences can be used in experience selection methods.

4.5.1 Experience Retention

Although we showed in Section 3.6.4 that high sampling rates might warrant dropping experiences, in general we assume that each new experience has at least some utility. Therefore, unless stated otherwise, we will always write newly obtained experiences to the buffer. When the buffer is full, this means that we need some metric that can be used to decide which experiences should be overwritten.

Experience Utility Proxies

A criterion used to manage the contents of an experience replay buffer should be cheap enough to calculate,[2] should be a good proxy for the usefulness of the experiences and should not depend on the learning process in a way that would

[2]We have discussed the need for experience diversity in Section 3.6.2 and we have previously proposed overwriting a buffer in a way that directly optimized for diversity (de Bruin et al., 2016a). However, calculating the experience density in the state-action space is very expensive and therefore prohibits using the method on anything but small-scale problems.

cause a feedback loop and possibly might destabilize that learning process. We consider three criteria for overwriting experiences.

Age: The default and simplest criterion is age. Since the policy is constantly changing and we are trying to learn its current effects, recent experiences might be more relevant than older ones. This (`FIFO`) criterion is computationally as cheap as it gets, since determining which experience to overwrite involves simply incrementing a buffer index. For smaller buffers, this does however make the buffer contents quite sensitive to the learning process, as a changing policy can quickly change the distribution of the experiences in the buffer. As seen in Figure 3.5, this can lead to instability.

Besides FIFO, we also consider reservoir sampling (Vitter, 1985). When the buffer is full, new experiences are added to it with a probability $\mathcal{C}/i$ where i is the index of the current experience. If the experience is written to the buffer, the experience it replaces is chosen uniformly at random. Note that this is the only retention strategy we consider that does not write all new experiences to the buffer. Reservoir sampling ensures that at every stage of learning, each experience observed so far has an equal probability of being in the buffer. As such, initial exploratory samples are kept in memory and the data distribution converges over time. These properties are shared with the `FULL DB` strategy, without needing the same amount of memory. The method might in some cases even improve the learning stability compared to using a full buffer, as the data distribution converges faster. However, when the buffer is too small this convergence can be premature, resulting in a buffer that does not adequately reflect the policy distribution. This can seriously compromise the learning performance.

Surprise: Another possible criterion is the unexpectedness of the experience, as measured by the temporal difference error δ from (4.1). The success of the Prioritized Experience Replay (PER) method of Schaul et al. (2016) shows that this can be a good proxy for the utility of experiences. Since the values have to be calculated to update the critic, the computational cost is very small if we accept that the utility values might not be current since they are only updated for experiences that are sampled. The criterion is however strongly linked with the learning process, as we are actively trying to minimize δ. This means that, when the critic is able to accurately predict the long term rewards of the policy in a certain region of the state-action space, these samples can be overwritten. If the predictions of the critic later become worse in this region, there is no way of getting these samples back. An additional problem might be that the error according to (4.1) will be

caused partially by state and actuator noise. Keeping experiences for which the temporal difference error is high might therefore cause the samples saved in the buffer to be more noisy than necessary.

Exploration: We introduce a new criterion based on the observation that problems can occur when the amount of exploration is reduced. On physical systems that are susceptible to damage or wear, or for tasks where adequate performance is required even during training, exploration can be costly. This means that preventing the problems caused by insufficiently diverse experiences observed in Section 3.6.2 simply by sustained thorough exploration might not be an option. We therefore view the amount of exploration performed during an experience as a proxy for its usefulness. We take the 1-norm of the deviation from the policy action to be the usefulness metric. In our experiments on the small scale benchmarks we follow the original DDPG paper (Lillicrap et al., 2016) in using an Ornstein-Uhlenbeck noise process added to the output of the policy network. The details of the implementation are given in Appendix B. In the experiments in Section 4.6.5 a copy of the policy network with noise added to the parameters is used to calculate the exploratory actions (Plappert et al., 2018).

For discrete actions, the cost of taking exploratory actions could be used as a measure of experience utility as well. The inverse of the probability of taking an action could be seen as a measure of the cost of the action. It could also be worth investigating the use of a low-pass filter, as a series of (semi)consecutive exploratory actions would be more likely to result in states that differ from the policy distribution in a meaningful way. These ideas are not tested here, as we only consider continuous actions in the remainder of this chapter.

Note that the size of the exploration signal is the deviation of the chosen action in a certain state from the policy action for that state. Since the policy evolves over time we could recalculate this measure of deviation from the policy actions per experience at a later time. Although we have investigated using this policy deviation proxy previously (de Bruin et al., 2016b), we found empirically that using the strength of the initial exploration yields better results. This can partly be explained by the fact that recalculating the policy deviation makes the proxy dependent on the learning process and partly by the fact that sequences with more exploration also result in different states being visited.

Notation	Proxy	Explanation
`Expl(`α`)`	Exploration	Experiences with the least exploration are stochastically overwritten with new ones.
`TDE(`α`)`	Surprise	Experiences with the smallest temporal difference error are stochastically overwritten with new ones.
`Resv`	Age	The buffer is overwritten such that each experience observed so far has an equal probability of being in the buffer.

Table 4.3: New and uncommon experience retention strategies considered in this work.

Stochastic Experience Retention Implementation

For the temporal difference error and exploration-based experience retention methods, keeping some experiences in the buffer indefinitely might lead to over-fitting to these samples. Additionally, although the overwrite metric we choose might provide a decent proxy for the usefulness of experiences, we might still want to be able to scale the extent to which we base the contents of the buffer on this proxy. We therefore use the same stochastic rank-based selection criterion of (4.2) suggested by Schaul et al. (2016), but now to determine which experience in the buffer is overwritten by a new experience. We denote this as `TDE(`α`)` for the temporal difference-based retention strategy and `Expl(`α`)` for the exploration-based policy. Here, α is the parameter in (4.2) which determines how strongly the buffer contents will be based on the chosen utility proxy. A sensitivity analysis of α for both Expl and PER is given in Appendix C.1. The notation used for the new experience retention strategies is given in Table 4.3.

4.5.2 Experience Sampling

For the choice of proxy when *sampling* experiences from the buffer, we consider the available methods from the literature: sampling either uniformly at random [`Uniform`], using stochastic rank-based prioritized experience replay [`PER`] and combining this with weighted importance sampling [`PER+IS`]. Given a buffer that contains useful experiences, these methods have shown to work well. We therefore focus on investigating how the experience retention and experience sampling strategies interact. In this context we introduce a weighted importance sampling method that accounts for the full experience selection strategy.

Importance sampling according to (4.3) can be used when performing prioritized experience replay from a buffer that contains samples with a distribution that is unbiased with respect to the environment dynamics. When this is not the case,

we might need to compensate for the effects of changing the contents of the buffer, potentially in addition to the current change in the sampling probability. The contents of the buffer might be the result of many subsequent retention probability distributions. Instead of keeping track of all of these, we compensate for both the retention and sampling probabilities by using the number of times an experience in the buffer has actually been replayed. When replaying an experience i for the K-th time, we relate the importance-weight to the probability under uniform sampling from a FIFO buffer of sampling an experience X times, where X is at least K: $\Pr(X \geq K|_{\texttt{FIFO[Uniform]}})$. We refer to this method as Full Importance Sampling (FIS) and calculate the weights according to :

$$\omega_i^{\text{FIS}} = \left(\frac{\Pr(X \geq K|_{\texttt{FIFO[Uniform]}})}{\left[\sum_{j=1}^{\lceil np \rceil} \Pr(X \geq j|_{\texttt{FIFO[Uniform]}})\right] / np} \right)^{\beta}.$$

Here, n is the lifetime of an experience for a FIFO retention strategy in the number of batch updates, which is the number of batch updates performed so far when the buffer is not yet full. The probability of sampling an experience during a batch update when sampling uniformly at random is denoted by p. Note that np is the expected number of replays per experience, which following Schaul et al. (2016) we take as 8 by choosing the number of batch updates per episode accordingly. As in Section 4.2.3 we use β to scale between not correcting for the changes and correcting fully. Since the probability of being sampled at least K times is always smaller than one for $K > 0$, we scale the weights such that the sum of the importance weights for the expected np replays under `FIFO[Uniform]` sampling is the same as when not using the importance weights ($n \cdot p \cdot 1$). The probability of sampling an experience at least K times under `FIFO[Uniform]` sampling is calculated using the binomial distribution:

$$\Pr(X \geq K|_{\texttt{FIFO[Uniform]}}) = 1 - \sum_{k=0}^{K} \binom{n}{k} p^k (1-p)^{n-k}.$$

Correcting fully ($\beta = 1$) for the changed distributions would make the updates as unbiased as those from the unbiased FIFO uniform distribution (Needell et al., 2016). However, since the importance weights of experiences that are repeatedly sampled for stability will quickly go to zero, it might also undo the stabilizing effects that were the intended outcome of changing the distribution in the first place. Additionally, as discussed in Section 4.2.3, the FIFO Uniform distribution is not the only valid distribution. As will be demonstrated in Section 4.6.4, it is

Notation	Proxy	Explanation
`Uniform + FIS`	-	Experiences are sampled uniformly at random, FIS (Section 4.5.2) is used to account for the distribution changes caused by the *retention* policy.
`PER+FIS`	Surprise	Experiences are sampled using rank based stochastic prioritized experience replay based on the temporal difference error. Full importance sampling is used to account for the distribution changes caused by both the retention and sampling policies.

Table 4.4: New experience sampling strategies considered in this work.

therefore important to determine whether compensating for the retention strategy is necessary before doing so.

The notation for the selection strategies with this form of importance sampling is given in Table 4.4.

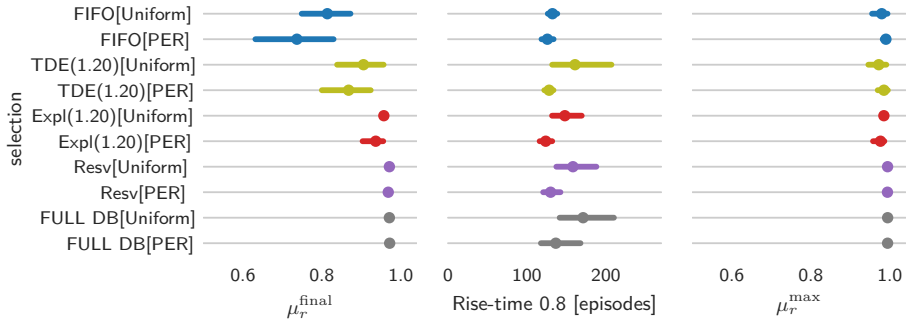
4.6 | Experience Selection Results

Using the experience retention and sampling methods discussed in Section 4.5, we revisit the scenarios discussed in Section 3.6. We first focus on the methods without importance sampling, which we discuss separately in Section 4.6.4. Besides the tests on the benchmarks of Section 3.4, we also show results on six additional benchmarks in Section 4.6.5. There we also discuss how to choose the size of the experience buffer.

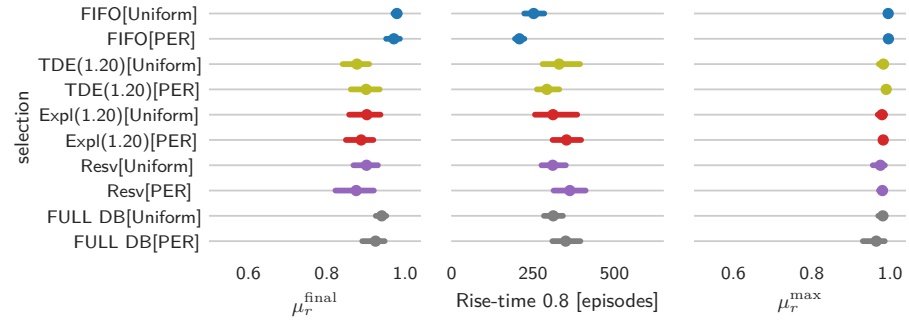
4.6.1 Basic Configuration

We start by investigating how these methods perform on the benchmarks in their basic configuration, with a sampling rate of 50 Hz and no sensor or actuator noise. The results are given in Figure 4.2 and show that it is primarily the combination of *retention* method and buffer size that determines the performance. It is again clear that this choice here depends on the benchmark. On the pendulum benchmark, where storing all experiences works well, the `Resv` method works equally well while storing only 10^4 experiences, which equals 50 of the 3000 episodes. On the magman benchmark, using a small buffer with only recent experiences works better than any other method.

Sampling according to the temporal difference error can be seen to benefit primarily the learning speed on the pendulum. On the magman, PER only speeds



(a) Swing-up



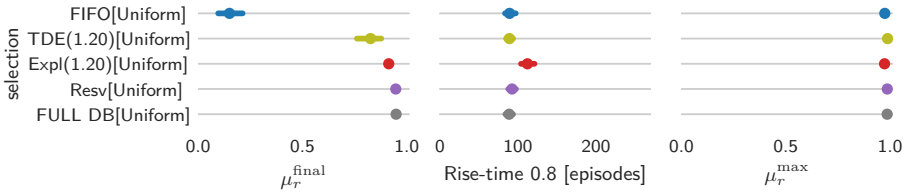
(b) Magman

Figure 4.2: Performance of the experience selection methods under the default conditions of moderate sampling frequencies and no state or actuator noise. A description of the performance measures is given in Section 3.5.

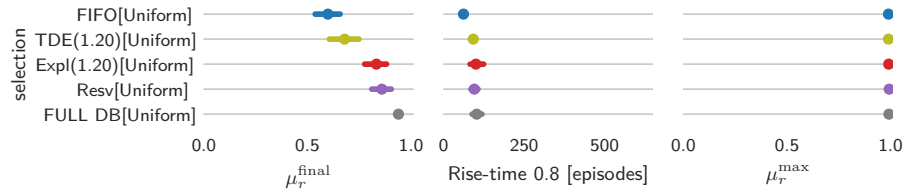
up the learning process when sampling from recent experiences. When sampling from diverse experiences, PER will attempt to make the function approximation errors more even across the state-action space, which as discussed before, hurts performance on this benchmark.

4.6.2 Effect of the Sampling Frequency

For higher sampling frequencies, the performance of the different experience selection methods is shown in Figure 4.3. We again see that higher sampling frequencies place different demands on the training data distribution. With the decreasing exploration, retaining the right experiences becomes important. This is most visible on the Magman benchmark where FIFO retention, which resulted in the best performance at the end of training for the base sampling frequency, now performs worst. Retaining all experiences works well on both benchmarks. When not all



(a) Swing-up [100 Hz]



(b) Magman [200 Hz]

Figure 4.3: Performance of the experience selection methods with increased sampling frequencies. Results are from 50 learning runs. A description of the performance measures is given in Section 3.5.

experiences can be retained, the reservoir retention method is still a good option here, with the exploration-based method a close second.

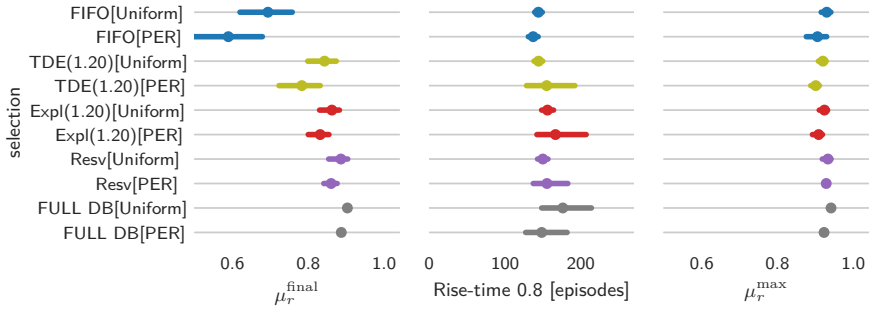
4.6.3 Sensor and Actuator Noise

We also test the performance of the methods in the presence of noise, similarly to Section 3.6.5. The main question here is how the noise might affect the methods that use the temporal difference error δ as the usefulness proxy. The concern is that these methods might favor noisy samples, since these samples might cause bigger errors. To test this we perform learning runs on the pendulum task while collecting statistics on all of the experiences in the mini-batches that are sampled for training. The mean absolute values of the noise in the experiences that are sampled are given in Table 4.5. It can be seen that the temporal difference error-based methods indeed promote noisy samples. The noise is highest for those dimensions that have the largest influence on the value of Q.

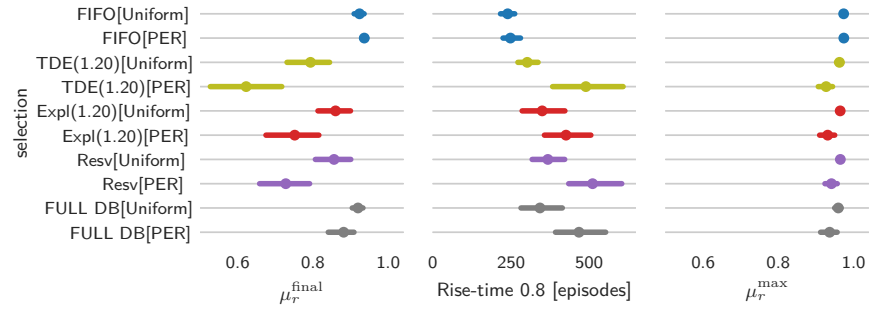
In Figure 4.4 the performance of the different methods on the two benchmarks with noise is given. The tendency to seek out noisy samples in the buffer is now clearly hurting the performance of PER sampling, as the performance with PER is consistently worse than with uniform sampling. For our chosen buffer size the retention strategy is still more influential and interestingly the TDE-based

	position	velocity	action
`Expl(1.0)[Uniform]`	$1.584 \cdot 10^{-2}$	$1.582 \cdot 10^{-2}$	$1.594 \cdot 10^{-2}$
`Expl(1.0)[PER]`	$1.654 \cdot 10^{-2}$	$1.630 \cdot 10^{-2}$	$1.595 \cdot 10^{-2}$
`TDE(1.0)[Uniform]`	$1.713 \cdot 10^{-2}$	$1.627 \cdot 10^{-2}$	$1.598 \cdot 10^{-2}$
`TDE(1.0)[PER]`	$1.846 \cdot 10^{-2}$	$1.743 \cdot 10^{-2}$	$1.594 \cdot 10^{-2}$

Table 4.5: Mean absolute magnitude of the noise per state-action dimension in the mini batches as a function of the experience selection procedure.



(a) Swing-up $[\sigma_s = \sigma_a = 0.02]$



(b) Magman $[\sigma_s = \sigma_a = 0.02]$

Figure 4.4: Performance of the experience selection methods with with sensor and actuator noise. Results are from 50 learning runs. A description of the performance measures is given in Section 3.5.

retention method does not seem to suffer as much here. The relative rankings of the retention strategies are similar to those without noise.

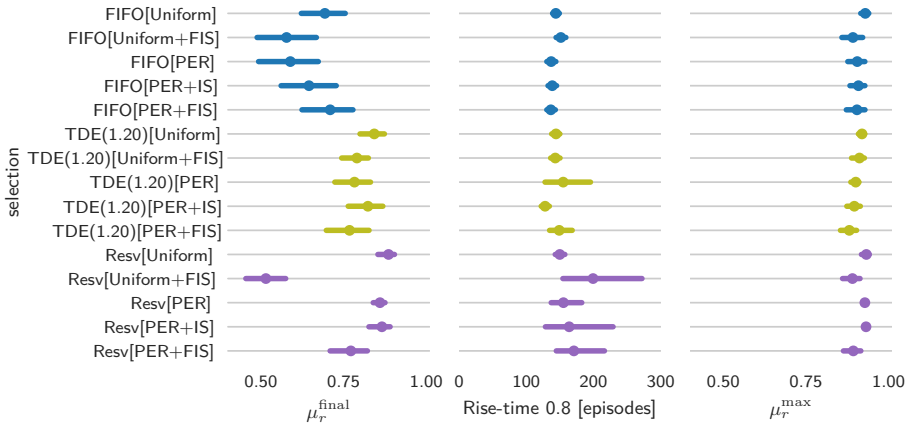
4.6.4 Importance Sampling

Finally, we investigate the different importance sampling strategies that were discussed in Sections 4.2.3 and 4.5.2. We do this by using the FIFO, TDE and Resv retention strategies as representative examples. We consider the benchmarks with noise, since as we discussed in Section 4.2.3, the stochasticity in the environment can make importance sampling more relevant. The results are shown in Figure 4.5. We discuss per retention strategy how the sample distribution is changed and whether the change introduces a bias that should be compensated for through importance sampling.

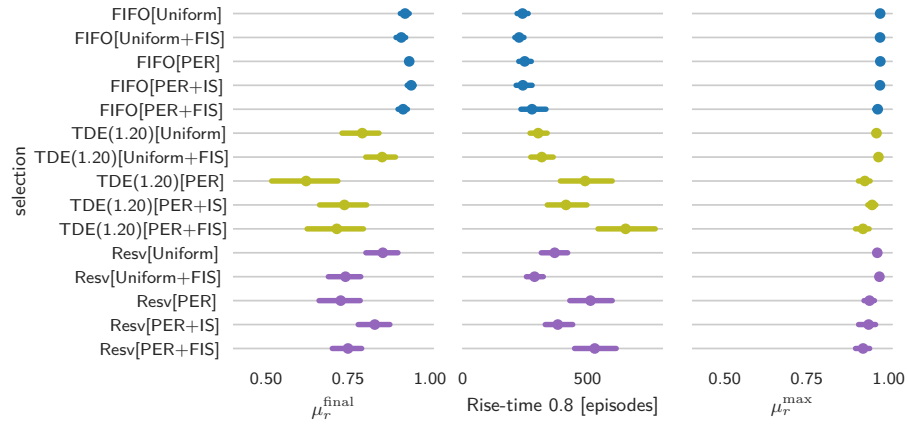
FIFO: This retention method results in an unbiased sample distribution. When combined with uniform sampling, there is no reason to compensate for the selection method. Doing so anyway (`FIFO[Uniform + FIS]`) results in downscaling the updates from experiences that happen to have been sampled more often than expected, effectively reducing the batch-size while not improving the distribution. The variance of the updates is therefore increased without reducing bias. This can be seen to hurt performance in Figure 4.5, especially on the swing-up task where sample diversity is most important. Using PER also hurts performance in the noisy setting as this sampling procedure *does* bias the sample distribution. Using importance sampling to compensate for just the sampling procedure (`FIFO[PER+IS]`) helps, but the resulting method is not clearly better than uniform sampling.

TDE: When the retention strategy is based on the temporal difference error, there is a reason to compensate for the bias in the sample distribution. It can be seen from Figure 4.5 however, that the full importance sampling scheme improves performance on the magman benchmark, but not on the swing-up task. The likely reason is again that importance sampling indiscriminately compensates for both the unwanted re-sampling of the environment dynamics and reward distributions as well as the beneficial re-sampling of the observation-action space distribution. The detrimental effects of compensating for the latter seem to outweigh the beneficial effects of compensating for the former on this benchmark where observation-action space diversity has been shown to be so crucial.

Resv: The reservoir retention method is not biased with respect to the reward function or the environment dynamics. Although the resulting distribution is strongly off-policy (assuming the policy has changed during learning), this



(a) Swing-up [$\sigma = 0.02$]



(b) Magman [$\sigma = 0.02$]

Figure 4.5: Performance of representative experience selection methods with and without importance sampling on the benchmarks with sensor and actuator noise. A description of the performance measures is given in Section 3.5.

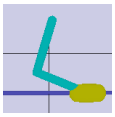
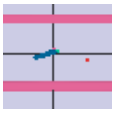
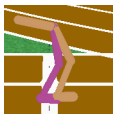
	InvDoublePnd	Reacher	Hopper	Walker2d	HalfCheetah	Ant
						
$\|o\|$	9	9	15	22	26	28
$\|a\|$	1	2	3	6	6	8

Table 4.6: The RoboSchool benchmarks considered in this section with the dimensionalities of their observation and action spaces.

does not present a problem for a deterministic policy gradient algorithm with Q-learning updates, other than that it might be harder to learn a function that generalizes to a larger part of the state space. When sampling uniformly, we do sample certain experiences, from early in the learning process, far more often than would be expected under a `FIFO[Uniform]` selection strategy. The FIS method compensates for this by weighing these experiences down, effectively reducing the size of both the buffer and the mini-batches. In Figure 4.5, this can be seen to severely hurt the performance on the swing-up problem, as well as the learning stability on the magman benchmark.

Interesting to note is that on these two benchmarks, for all three considered retention strategies, using importance sampling to compensate for the changes introduces by PER only improved the performance significantly when using PER resulted in poorer performance than not using PER. Similarly, using FIS to compensate for the changes introduced in the buffer distribution only improved the performance when those changes should not have been introduced to begin with.

4.6.5 Additional Benchmarks

The computational and conceptual simplicity of the two benchmarks used so far allowed for comprehensive tests and a good understanding of the characteristics of the benchmarks. However, we also saw that the right experience selection strategy is benchmark dependent. Furthermore, deep reinforcement learning yields most of its advantages over reinforcement learning with simpler function approximation on problems with higher dimensional observation and action spaces. To obtain a more complete picture we therefore perform additional tests on 6 benchmarks of varying complexity.

Benchmarks

In the interest of reproducibility, we use the open source RoboSchool (Klimov, 2017) benchmarks together with the openAI baselines (Dhariwal et al., 2017) implementation of DDPG. We have adapted the baselines code to include the experience selection methods considered in this section. Our adapted code is available online.[3]

The baselines version of DDPG uses Gaussian noise added to the parameters of the policy network for exploration (Plappert et al., 2018). In contrast to the other experiments in this work, the strength of the exploration is kept constant during the entire learning run. For the Expl method we still consider the 1-norm of the distance between the exploration policy action and the unperturbed policy action as the utility of the sample.

Results

For the benchmarks listed in Table 4.6, we compare the default `FULL DB[Uniform]` selection strategy in the baselines code to the alternative retention strategies considered in this work with uniform sampling. We show the maximum performance for these different retention strategies as a function of the buffer size in Figure 4.6. From the figure, it can be seen that on these noise-free benchmarks with constant exploration and moderate sampling frequencies, the gains obtained by using the considered non-standard experience selection strategies are limited. However, in spite of the limited number of trials performed due to the computational complexity, trends do emerge on most of the benchmarks. On all benchmarks, the best performance is seen not when retaining all experiences, but rather when learning from a smaller number of experiences. This is most visible on the reacher task, which involves learning a policy for a 2-DOF arm to move from one random location in its workspace to another random location. For this task, the best performance for all retention strategies is observed when retaining less than a tenth of all experiences.

For these noise-free benchmarks, the temporal difference error is an effective proxy for the utility of the experiences, resulting in the highest or close to the highest maximum performance on all benchmarks.

The exploration-based retention strategy was introduced to prevent problems when reducing exploration and for high sampling frequencies. Since the exploration is not decayed and the sampling frequencies are modest, there is no real benefit when applying this strategy to these benchmarks. However, it also does not seem to hurt performance compared to the age-based retention strategies. The constant

[3]The code is available at https://github.com/timdebruin/baselines-experience-selection.

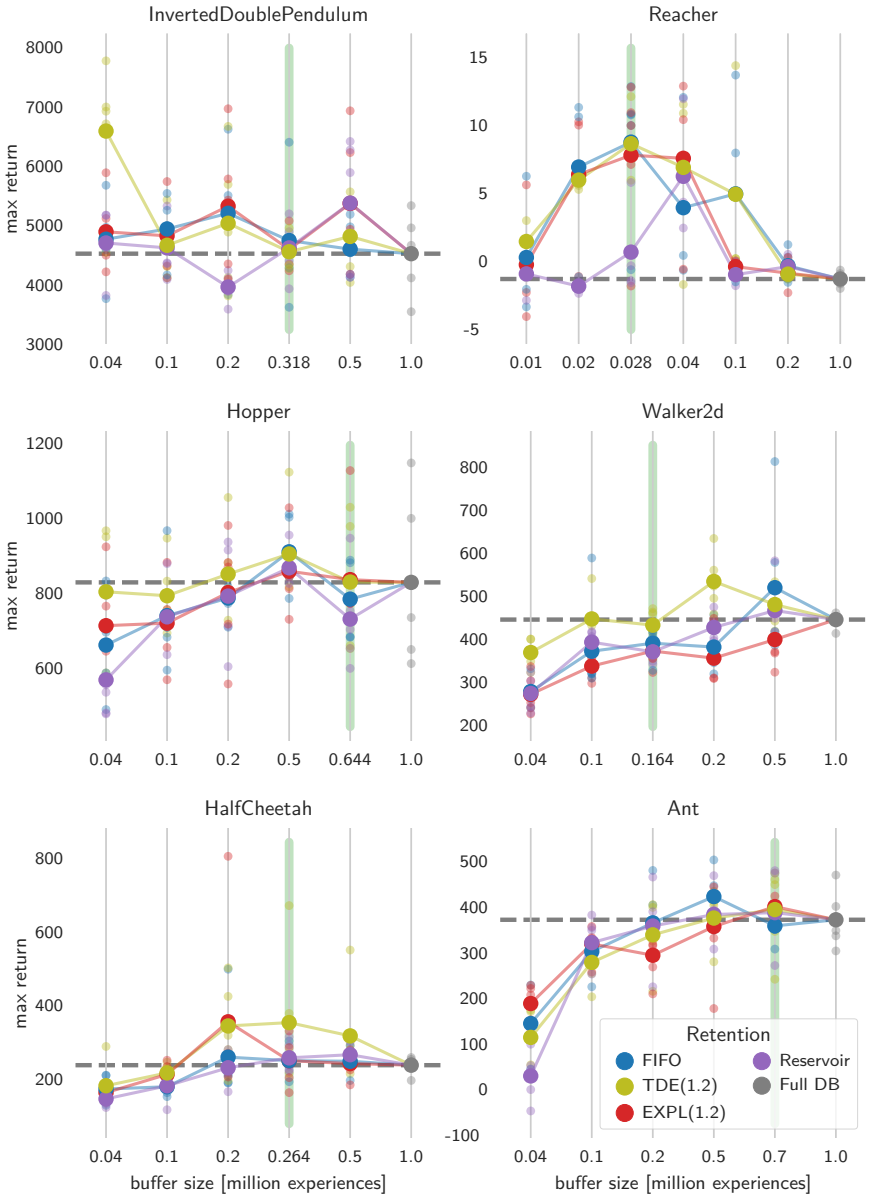


Figure 4.6: Maximum performance during a training run on the Roboschool benchmarks as a function of the retention strategy and buffer size. Results for the individual runs and their means are shown. In Appendix C, we additionally show the mean (Figure C.9) and final (Figure C.8) performance. Green lines indicate the rule of thumb buffer sizes of Figure 4.7.

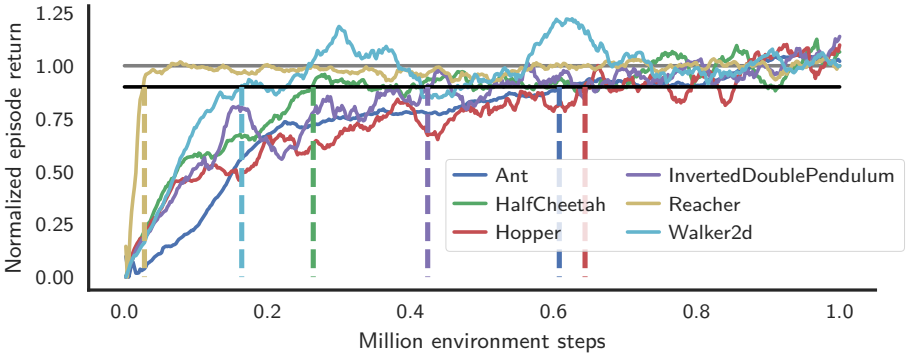


Figure 4.7: Learning curves of the `FULL DB` method on the different benchmarks, averaged over 5 trials. The curves are normalized by the final performance (the mean performance over the last $2 \cdot 10^5$ steps). Indicated are the number of steps needed to get to 90% of the final performance.

exploration on these benchmarks additionally means that the performance of FIFO and Reservoir retention are rather close, although due to premature convergence of the data distribution, reservoir retention does suffer the most when the buffer capacity is too low.

Figures 4.6, C.8 and C.9 show that when the right proxy for the utility of experiences is chosen, performance equal to and often exceeding that of retaining all experiences can be obtained while using only a fraction of the memory. This begs the question of how to choose the buffer size.

As it tends to result in more stable learning, retaining as many experiences as possible seems a sensible first choice for the buffer size. We therefore base our suggestion for subsequently tuning the buffer size on the learning curves of the `FULL DB`[`Uniform`] method. The complexity of the control task at hand determines the minimal number of environment steps required to learn a good policy, as well as the number of experiences that need to be retained in a buffer for decent learning performance. We propose to use the number of experiences needed to get to 90% of the final performance as a *rough empirical estimate* of the optimal buffer size. We show this rule of thumb in Figure 4.7 and have indicated the experiments with the proposed buffer sizes in Figure 4.6 with vertical green lines.

Instead of iteratively optimizing the buffer size over several reinforcement learning trials, extrapolation of the learning curve (Domhan et al., 2015) could also be used to limit the buffer capacity when the remaining learning performance increase is expected to be small. This would allow the method to work immediately for novel tasks.

4.7 | Conclusions and Recommendations

We have investigated how the characteristics of a control problem determine what experiences should be replayed when using experience replay in deep reinforcement learning.

We first investigated how factors such as the generalizability over the state-action space and the sampling frequency of the control problem influenced the balance between the need for on-policy experiences versus a broader coverage of the state-action space.

We then investigated a number of proxies for the utility of experiences which we used to both decide which experiences to retain in a buffer and how to sample from that buffer. We performed experiments that showed how these methods were affected by noise, increased sampling frequencies and how their performance varied across benchmarks and experience buffer sizes.

Based on these investigations we present a series of recommendations below for the three choices concerning experience selection: how to choose the capacity of the experience replay buffer, which experiences to retain in a buffer that has reached its capacity and how to sample from that buffer. These choices together should ensure that the experiences that are replayed to the reinforcement learning agent facilitate quick and stable learning of a policy with good performance. An example of applying the procedure outlined below on the Magman benchmark is given in Figure 4.8. Note the proposed methods are especially relevant when faced by issues that might occur in a physical-control setting, such as a need for constrained exploration, high or low sampling frequencies, the presence of noise and hardware limitations that place constraints on the experience buffer size. Section 4.6.5 showed that the potential gains might be limited for processes where these problems do not occur.

4.7.1 Choosing the Buffer Capacity

Although it is not the best retention strategy in most of the benchmarks we have considered, retaining as many experiences as possible is a good place to start. This tends to result in more stable learning, even if the eventual performance is not always optimal.

If the learning curve for the `FULL DB` experiments reaches a level of performance close to the performance after convergence in significantly fewer environment steps than there are experience samples in the buffer, it might be worthwhile reducing the size of the buffer. Our proposed rule of thumb is to to make the buffer size roughly equal to the number of environment steps needed to reach to 90% of the final performance level.

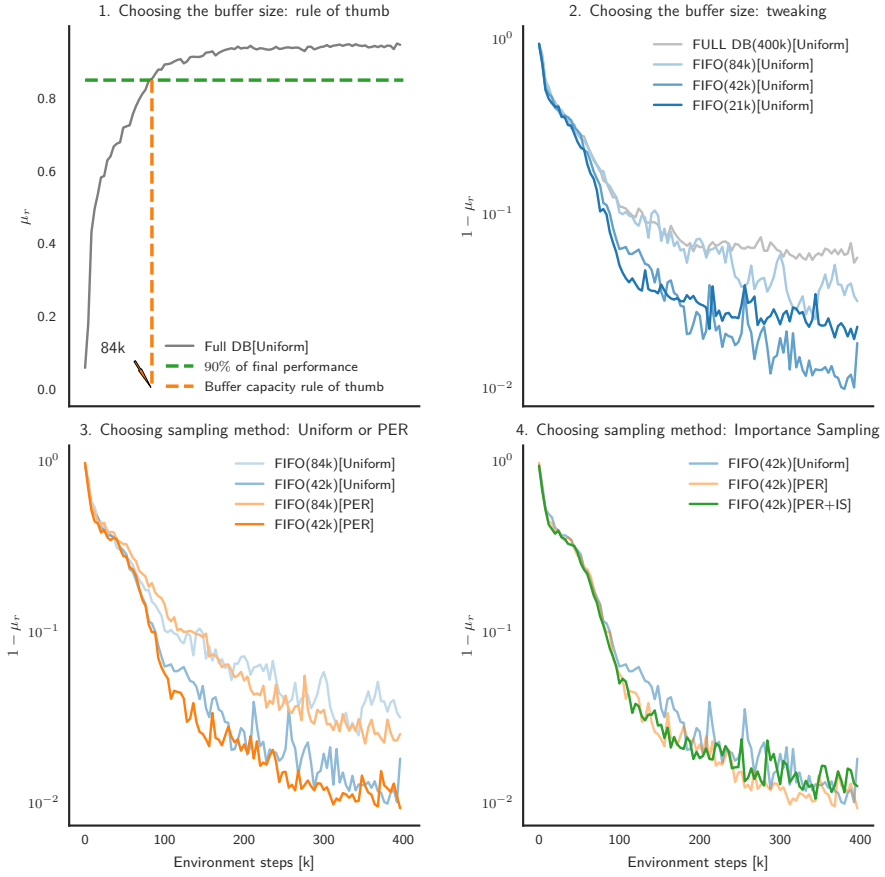


Figure 4.8: Demonstration of the proposed process for the magman benchmark. 1: Based on the performance of the Full DB[Uniform] method, the rule of thumb indicates a buffer capacity of 84×10^3 experiences. As there are no special circumstances such as high sampling frequencies and the magman requires a very precise policy that does not easily generalize due to the highly nonlinear behavior of the magnets, FIFO retention is used. 2: By exploring around the proposed buffer size, a buffer capacity of 42×10^3 experiences is chosen. 3: Sampling from the buffer based on the temporal difference error can help speed up and stabilize learning, but is very dependent on the experiences that are in the buffer to begin with. 4: Since the benchmark fully deterministic (noise free), importance sampling is not needed in this case and can be seen to undo some of the benefits of PER.

4.7.2 Choosing the Experience Utility Proxy

When not all experiences are retained in the buffer, a proxy for the utility of the individual experiences is needed to determine which experiences to retain and which to discard. In this work, we have discussed strategies based on several proxies and shown that the right strategy is problem dependent. Although finding the right one will likely require some experimentation, we discuss here what properties of the control problem at hand make certain strategies more likely or less likely to succeed.

FIFO: Although off-policy reinforcement-learning algorithms can learn from samples obtained by a different policy than the optimal policy that is being learned, the reality of deep reinforcement learning is that a finite amount of shared function approximation capacity is available to explain all of the training data. While simply using larger networks might help, we show in Appendix C.1 that learning only from more recent data (which corresponds more closely to the policy being learned) can work better. A large potential downfall presents itself when the policy suddenly changes in a way that changes the distribution of the states that are visited. As shown in Section 3.6.2, this can quickly destabilize the learning process. Extra care should be taken when using FIFO retention in combination with decaying exploration. This is especially true for problems where multiple policies are possible that give similar returns but distinct state/observation-space trajectories, such as swinging up a pendulum either clockwise or anti-clockwise.

TDE: The idea behind selecting certain experiences over others is that more can be learned from these samples. The temporal difference error is therefore an interesting proxy, especially during the early stages of the learning process when the error is mostly caused by the fact that the value function has not been accurately learned yet. In the experiments of Schaul et al. (2016) as well as in our own experiments, prioritizing experiences with larger TD errors was observed to improve both the speed of learning as well as the eventual performance in many cases. The downside of using the TD error as an experience utility proxy is that the error can also be caused by sensor and actuator noise, environment stochasticity or function approximation accuracy differences as a result of differences in the state-space coverage. We have shown in Section 4.6.3 how noise can hurt the performance of the algorithm when using this proxy and argued in Section 4.2.3 how this proxy introduces a harmful bias in the presence of environment stochasticity.

Exploration: We introduced an additional proxy based on the observation that, especially on physical systems, exploration can be costly. By using the strength

of the exploration signal as a proxy for the utility of the experience, some of the problems mentioned for the FIFO strategy when reducing exploration can be ameliorated. As shown in Section 3.6.4 and Section 4.6.2, sufficient diversity in the action space is most important when the dependency of the value function on the action is relatively small, such as for increased sampling frequencies. The downside of this strategy is that since it focuses on early experiences that are more off-policy, it can take longer for the true value function to be learned. Besides the impact on training speed, the focus on off-policy data can also limit the maximum controller performance.

Reservoir sampling: By using reservoir sampling as a retention strategy, the buffer contains, at all times, samples from all stages of learning. As with the exploration-based policy, this ensures that initial exploratory samples are retained which can significantly improve learning stability on domains where FIFO retention does not work. However, of the methods mentioned here, reservoir sampling is the one most severely impacted by a too small experience buffer, as the data distribution in the buffer will converge prematurely and will not cover the state-action space distribution of the optimal policy well enough.

4.7.3 Experience Sampling and Importance Sampling

The experiences that are used to learn from are not just determined by the buffer retention strategy, but also by the method of sampling experiences from the buffer. While the retention strategy needs to ensure that a good coverage of the state-action space is maintained in the buffer throughout learning, the sampling strategy can seek out those experiences that can result in the largest immediate improvement to the value function and policy. It can therefore be beneficial to *sample* based on the temporal difference error (as suggested by Schaul et al. 2016), which can improve learning speed and performance, while basing the *retention* strategy on a more stable criterion that either promotes stability or ensures that only samples from the relevant parts of the state-action space are considered by the sampling procedure.

As discussed in Sections 4.2.3 and 4.6.4, selecting experiences based on the temporal difference error in stochastic environments introduces a bias that should be compensated for through weighted importance sampling in order to make the learning updates (more) valid. While the other experience selection methods in this work change the distribution of the samples, these changed distributions are still valid for an off-policy deterministic gradient algorithm.

5

Integrating State Representation Learning into Deep Reinforcement Learning

Parts of this chapter have previously been published in:

de Bruin, T., Kober, J., Tuyls, K., Babuška, R. (2018). *"Integrating State Representation Learning into Deep Reinforcement Learning"*. IEEE Robotics and Automation Letters (RA-L / ICRA).

5.1 | Introduction

In the last two chapters we have investigated which experiences we would like to learn from. Now, we now turn our attention to the question of *what* to learn from them. One of the most exciting promises of using reinforcement learning for robot control is the fact that, instead of having to explicitly program the required behaviors, only a reward function that captures the success of the robot at performing the task is required. Unfortunately, while it has been shown that this reward function provides a signal that can be used to find a mapping directly from sensory signals to correct actuator commands (e.g., Finn et al., 2016; Mnih et al., 2015), the often uninformative nature of the reward signal contributes to the large amount of experiences required to learn good, generally applicable policies.

This problem is especially pronounced in the robotics domain, where much of the complexity of learning a new task lies in learning to perceive the world. Robot designers often equip their robots with several different types of sensors that estimate the state of the world through measuring different physical phenomena. Deep Learning has been shown to be very capable of extracting descriptive features from high-dimensional, multi-modal inputs (Ngiam et al., 2011). However, while reward functions describe the desirability of the state of the world, they often provide only vague and indirect information on how to distill that state from the raw sensory observations. This further increases the number of required samples, which when combined with the high operating cost of robots, makes using reinforcement learning in this domain infeasible in general.

In this chapter, an autonomous racing car is used as a simulation benchmark. Images, range-finder readings and velocity data are observed and rewards are given based on the velocity along the track-axis combined with the distance from the center of the track. With a wealth of high-dimensional sensor data and a scalar reward it is easy to learn the wrong causal relations. Was the negative reward due to the tree in the background, the combination of velocity and the distance to the track-edge or the color of the road markings? While some of these observations might be correlated to a good driving policy on one track, the learned policy will not necessarily work on a different circuit. Without a good compact and concise representation of the state of the problem, large amounts of diverse data will be required before the true causal connections outweigh the accidental ones and a general policy is found.

In order to make learning suitable state representations from the raw sensor data easier and faster, additional training criteria can be used to supplement the reinforcement learning objective (Caruana, 1993). These State Representation Learning (SRL) criteria can simplify the representation learning problem by encoding

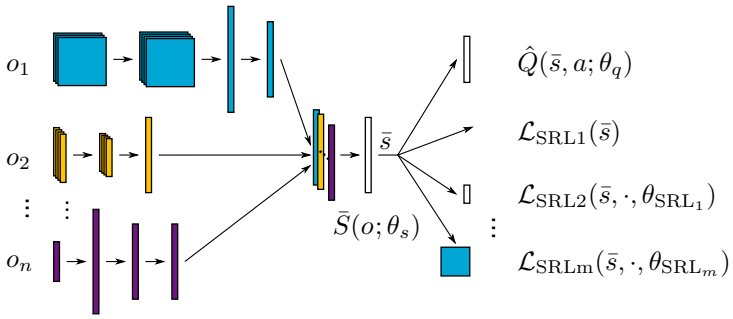


Figure 5.1: The general neural network architecture considered in this chapter. A shared state embedding $\bar{s} = \bar{S}(o;\theta_s)$ of the different sensory modalities o_m is learned. Both the reinforcement learning cost function (line 16 of Algorithm 1), as well as a number of state-representation learning cost terms $\mathcal{L}_{\text{SRL1},\ldots,\text{m}}$ are employed to shape the embedding during training.

prior knowledge and can help to regularize the learned representation by making it adhere to some fundamental properties from physics. Examples of state representation learning criteria include the classical auto-encoding objective (e.g., Finn et al., 2016; Hinton and Salakhutdinov, 2006; van Hoof et al., 2016; Lange et al., 2012b), predicting instantaneous rewards (e.g., Jaderberg et al., 2017; Jonschkowski and Brock, 2015; Munk et al., 2016), learning the (inverse) dynamics in the state embedding space (e.g., Agrawal et al., 2016; Shelhamer et al., 2016; Watter et al., 2015) or encoding the belief that state representations should change only slowly over time (e.g., Jonschkowski and Brock, 2015; Wiskott and Sejnowski, 2002), while being diverse in general (Jonschkowski et al., 2017).

These additional optimization criteria have the potential to aid reinforcement learning, and even to substitute for reinforcement learning when a shaped reward function is not available (Agrawal et al., 2016), as is often the case is real world settings. However, realizing this potential can be non-trivial. When the auxiliary optimization terms are added to the reinforcement learning objective naively, performance can easily be reduced rather than improved. Many of the works that have introduced new state-representation learning criteria have done so for purposes other than reinforcement learning (e.g., Agrawal et al., 2016; Shelhamer et al., 2016), or with the state-representation learning separated from the reinforcement learning (e.g., van Hoof et al., 2016; Jonschkowski et al., 2017; Lee et al., 2017; Munk et al., 2016). In this chapter we take a host of state-representation learning criteria from the literature. We then propose and compare different ways of integrating state-representation learning with popular deep reinforcement learning methods. Specifically, we make the following contributions:

- We propose and investigate a method for reducing the potential detrimental

effects of changing the state representation on-line.

- We combine several state-representation learning objectives from the literature and investigate their contributions.
- We compare the effects of pre-training the state-representation and policy on data from a related domain to learning both from scratch during the reinforcement learning trials.
- We learn a shared state representation from multi-modal sensor observations.
- We perform state representation learning and reinforcement learning simultaneously, allowing the reinforcement learning process to help shape the state representation while the state representation learning helps regularize that representation.

The rest of this chapter is organized as follows: in Section 5.2 we discuss the state representation learning objectives that we will use. Section 5.3 discusses the different methods we consider for combining these objectives in a way that maximally benefits the reinforcement learning process. To test these methods we perform experiments that are described in Section 5.4, and examine their results in Section 5.5. Further discussion and conclusions are given in Section 5.6.

5.2 | Learning Objectives

We consider a robot that observes an environment with one or more sensors. We indicate the individual sensory observations with o_m, where $m \in \mathcal{M}$ indicates the sensor modality with $\mathcal{M}$ the set of modalities of the sensors that the robot is equipped with. We denote with o the combined sensory signals of the robot.

To optimize the return R (2.1) obtained form a given state s, a policy has to be found that maps the observations to actions. In robotics applications, these sensory observations tend to be high dimensional, noisy and redundant, which makes learning a policy from them directly based only on rewards both costly and prone to over-fitting. Our aim therefore is to learn a mapping from the sensory observations to a low dimensional, concise representation of the task relevant aspects of the state: $\bar{s} = \bar{S}(o; \theta_s)$. This representation should allow learning a policy that generalizes better, while using less data. However, in contrast to many other works on representation learning, we consider reinforcement learning itself as a crucial state representation learning objective, and allow it to help shape the representation.

In the following subsections, we describe the different optimization criteria that we use to map the observations to state representations and the state representations to actions.

5.2.1 Reinforcement Learning

The first objective for which we optimize is the reinforcement learning objective. We use (Deep Double) Q-learning in this chapter (Mnih et al., 2015; Van Hasselt et al., 2016; Watkins, 1989, see also Section 2.3.2) for its simplicity and popularity. The algorithm is used to learn to approximate the value of the return (2.1) when taking action a after observing o and following the optimal policy from the subsequent time-step onwards. This function is learned by minimizing the squared temporal difference error (4.1) between the network predictions $\hat{Q}$ and the DDQN return estimates q (2.8):

$$\mathcal{L}_{RL} = \left(\hat{Q}\left(\bar{S}(o;\theta_s), a; \theta_q\right) - q(o,a)\right)^2. \tag{5.1}$$

5.2.2 Auto-encoding

Besides using the reinforcement learning objective to shape the state representation, we want to add additional objectives that encode some form of prior knowledge which can help simplify and regularize the state representation learning process by adding optimization targets and limiting the model search space. The most general prior knowledge that we encode is the knowledge that high dimensional sensory observations are often the result of a smaller number of relevant latent state variables (Finn et al., 2016; Ghadirzadeh et al., 2017; Hinton and Salakhutdinov, 2006; Lange et al., 2012b). Additionally, we use the knowledge that the different sensors on a robot all measure different physical effects of the same environment state. We encode these beliefs in two ways.

The first is through the network structure. While the different sensory modalities have their own encoders, these encodings are then fused and embedded into a shared state embedding space, as shown in Fig. 5.1. This embedding space is much lower dimensional than (some of) the sensory observations.

The second way in which we enable the state representation to encode significant aspects of the state is by reconstructing the observations of one or more of the sensory modalities from the shared embedding space, by minimizing the following loss:

$$\mathcal{L}_{AE^m} = \left\|\hat{o}_m\left(\bar{S}(o;\theta_s); \theta_{\mathrm{AE}^k}\right) - o_m\right\|^2, \tag{5.2}$$

where $\hat{o}_m(\bar{S}(o;\theta_s); \theta_{\mathrm{AE}^k})$ is the reconstruction of o_m made by a decoding layer in the network based on the state representation. In our autonomous car benchmark,

we might for instance expect the representation that is learned to include the curvature of the road, which would help explain much of the variation in both the images and the range-finder measurements.

5.2.3 Reward prediction

While auto-encoding is very general, and encourages encoding those factors that can help explain the significant variations in the observed sensor data, we might want the state-representation to specifically focus on those aspects of the environment state that are relevant to the task that needs to be performed. The second SRL objective we consider is therefore predicting the instantaneous rewards (Jaderberg et al., 2017; Jonschkowski and Brock, 2015; Munk et al., 2016). Doing this in addition to learning a value function helps especially when the rewards are sparse. However, even when this is not the case, this loss term is easier to optimize for than the temporal difference error (5.1) as changes to the policy will only change the data distribution and not the training targets.

We use the mean squared error as the reward prediction loss term:

$$\mathcal{L}_{rew} = \left(\hat{r}\left(\bar{S}(o;\theta_s), a, \bar{S}(o';\theta_s);\theta_r\right) - r\right)^2, \tag{5.3}$$

with $\hat{r}\left(\bar{S}(o;\theta_s), a, \bar{S}(o';\theta_s);\theta_r\right)$ the prediction of the network, based on the subsequent state representations and action, of the reward r. For the driving task, predicting the reward would encourage encoding the velocity of the car as well as the position and orientation of the car relative to the track axis, regardless of the current policy. Note however that this is not sufficient for a good racing policy, as we would also need properties like the distance to the next corner, which does not influence the instantaneous reward. This is why we still consider the temporal difference loss (5.1) to be an important state-representation learning criterion.

5.2.4 Slowness and diversity

It is also possible to encode knowledge about physics which should be applicable to state representation learning for robotics, regardless of the task (Jonschkowski and Brock, 2015). This physical prior knowledge can be encoded as loss functions that act directly on the learned state-space embedding. One popular physical prior is that states should not change quickly over short periods of time (Wiskott and Sejnowski, 2002). A potential downside of encoding this belief is that its optimum is a state representation that does not change at all, and therefore contains no information. We can counter this by adding an additional loss term that encourages diversity between non consecutive states (Jonschkowski et al., 2017). The slowness

$\mathcal{L}_{slow}$ and diversity $\mathcal{L}_{div}$ loss terms we use are respectively:

$$\mathcal{L}_{slow} = \left\|\bar{S}(o';\theta_s) - \bar{S}(o;\theta_s)\right\|^2, \tag{5.4}$$

$$\mathcal{L}_{div} = e^{-\|\bar{s}(o_x;\theta_s)-\bar{s}(o_y;\theta_s)\|^2}, \tag{5.5}$$

where o_x and o_y are non-consecutive observations. In practice we calculate the average of $\mathcal{L}_{div}$ over the experiences in a training mini-batch which is sampled uniformly at random from an experience replay buffer.

5.2.5 (Inverse) state dynamics

Since our eventual goal is to select actions based on the state representation, it can also be beneficial to make sure that the embedding specifically encodes those aspects of the world that can be changed by the robot's actions. This can be done by learning the inverse state-representation dynamics; predicting which action was responsible for the transition between two states (Agrawal et al., 2016; Shelhamer et al., 2016). We also learn the forward dynamics by giving a prediction $\hat{\bar{s}}'$ of the next state embedding $\bar{s}'$ based on the current state embedding and action. We assume the environment state to be (approximately) Markovian and in learning the forward dynamics we attempt to ensure that our state representation also has this property. Since we consider discrete actions in this chapter we use a classification loss term for the inverse dynamics $\mathcal{L}_{inv}$. The forward dynamics $\mathcal{L}_{fwd}$ are posed as a regression problem:

$$\mathcal{L}_{inv} = -\log\left(\hat{P}\left(a|\bar{S}(o;\theta_s), \bar{S}(o';\theta_s);\theta_{inv}\right)\right), \tag{5.6}$$

$$\mathcal{L}_{fwd} = \left\|\hat{\bar{S}}'\left(\bar{S}(o;\theta_s), a;\theta_{fwd}\right) - \bar{S}(o';\theta_s)\right\|^2. \tag{5.7}$$

5.3 | Main Contribution: Integration Methods

The loss functions from the literature that were reviewed in the previous section have been used in a number of different ways. In some works, state-representation learning was not explicitly combined with reinforcement learning (e.g., Agrawal et al., 2016; Watter et al., 2015). In others, the state-representation learning objectives were used during an initial pre-training phase while the state encoding was held (partially) fixed during the subsequent reinforcement learning phase (e.g., van Hoof et al., 2016; Jonschkowski et al., 2017; Lee et al., 2017; Munk et al., 2016). Yet others use the auxiliary optimization objectives during the reinforcement learning phase (e.g., Jonschkowski and Brock, 2015; Mirowski et al., 2017; Shelhamer et al., 2016) or even learn separate RL controllers that optimize for auxiliary tasks (Jaderberg et al., 2017). In this chapter, we use a large number of SRL objectives from the literature, and we propose and investigate different

ways of integrating them with popular deep RL algorithms. The main aim for the combined methods is finding control policies that generalize well, while minimizing the number of required environment interactions.

5.3.1 Simultaneous optimization

The first and most straightforward way of integrating state-representation learning with reinforcement learning that we consider is to simply add the SRL objectives to the RL loss $\mathcal{L}_{RL}$ and optimize for this new loss function $\mathcal{L}_{sim}$ instead of the standard loss function used in the RL algorithm:

$$\mathcal{L}_{sim} = \mathcal{L}_{RL} + c_{SRL}\left(c_{SRL_1}\mathcal{L}_{SRL_1} + \cdots + c_{SRL_n}\mathcal{L}_{SRL_n}\right), \tag{5.8}$$

where $c_{SRL_{1,\ldots,n}}$ are scaling constants for the individual SRL loss terms and c_{SRL} is an overall scaling term that trades off the SRL objectives with the RL objective. The individual loss scaling terms $c_{SRL_{1,\ldots,n}}$ are the same in all our experiments. They were chosen once, such that the 2-norms of the gradients of the loss terms with respect to the embedding vector are of the same order of magnitude during the early stages of learning. We do vary the overall scaling c_{SRL} to investigate the effects of the trade-off between strictly enforcing our state-representation knowledge and allowing the reinforcement learning to mostly dictate the representation. We refer to this method of optimizing *simultaneously* for all loss terms as `sim`.

During the learning process, we perform batch updates after each episode, with the number of updates dependent on the number of experiences obtained during the episode.

5.3.2 Alternating optimization with fixed Q values

One of the challenges of reinforcement learning, compared to supervised learning, is the fact that the training data distribution can change significantly as a result of a change in the policy. When using the `sim` method, this might cause some of the auxiliary loss terms to suddenly significantly change the state representation. This in turn can change the Q-values, which are dependent on the representation. As action gaps are generally small compared to the Q-values (Bellemare et al., 2016b), even small changes in these values could inadvertently change their ordering and, as a consequence, the policy. This could further destabilize the learning process.

To mitigate these effects we propose a second method, `alt`, in which the network parameters are updated in two *alternating* phases. In each cycle, we first predetermine the experiences that will be sampled from the experience buffer. For these experiences, we determine the predicted Q-values with the current network parameters: $\hat{Q}(\bar{S}(o;\theta_s), a; \theta_q^{\mathcal{I}})$ where $\mathcal{I}$ indicates the parameter update step at the start of the current cycle. We then first perform a number of update steps where

we optimize for the state representation learning objectives, while attempting to minimize the changes to the predicted Q-values:

$$\mathcal{L}_{alt} = c_{\text{Qfix}}\mathcal{L}_{\text{Qfix}} + c_{SRL}\left(c_{SRL_1}\mathcal{L}_{SRL_1} + \cdots + c_{SRL_n}\mathcal{L}_{SRL_n}\right), \tag{5.9}$$

with:

$$\mathcal{L}_{\text{Qfix}} = \left(\hat{Q}\left(\bar{S}(o;\theta_s), a; \theta_q\right) - \hat{Q}\left(\bar{S}(o;\theta_s), a; \theta_q^{\mathcal{I}}\right)\right)^2. \tag{5.10}$$

After these updates we perform the same number of updates with the regular reinforcement learning objective $\mathcal{L}_{RL}$ (5.1).

5.4 | Experiments

We performed experiments with the Torcs (Wymann et al., 2000; Yoshida, 2016) racing simulator. The aim is to complete a lap of a track as quickly as possible. We use three different sensory modalities:

1. $o_{\text{RGB}} \in \mathbb{R}^{12288}$: RGB images of 64 by 64 pixels, looking forward from the car.
2. $o_{\text{T}} \in \mathbb{R}^{19}$: Measurements of the distance to the track edge at 10 degree intervals covering the front of the car. When the car is off the track the measurements are -1.
3. $o_{\text{C}} \in \mathbb{R}^{5}$: The translational velocity of the car and the rotational velocities of each of the wheels.

For all experiments, the three different sensory modalities $o_{\text{RGB}}, o_{\text{T}}, o_{\text{C}}$ are embedded into a 30-dimensional shared state-representation space $\bar{s} \in \mathbb{R}^{30}$. We use the state-representation learning cost functions described in Section 5.2, with both o_{RGB} and o_{T} as auto-encoding targets. For o_{RGB} we reconstruct a down-sampled

Figure 5.2: Training and validation experiments are performed on four tracks. Pre-train data from a separate fifth track is used in some experiments.

image. We used the following neural network architecture. Three separate encoders were employed:

- For the camera images o_{RGB}, the same convolutional architecture as in Mnih et al. (2015) was used, without the final fully connected layer but with normalization layers (Ioffe and Szegedy, 2015).
- For the track (range finder) measurements o_{T} a fully connected encoder was used with two ReLU layers of 50 and 25 units respectively.
- The car observations o_{C} encoder, uses a single layer of 25 units with ReLU nonlinearities.

All encoders are followed by a single linear layer of size 30. The shared representation is obtained by averaging over the different modalities to get a single shared representation $\bar{s} \in \mathbb{R}^{30}$ (Coates and Bollegala, 2018). The decoders, for the Q-values, downsized (32x32x3) RGB targets, and the reconstruction of the track measurements all use an affine transformation of the state embedding.

As in other works, (e.g., Liu et al., 2017), we use a reward function that penalizes the distance from the center of the track as well as the velocity perpendicular to the track axis, while rewarding the velocity along the track axis:

$$r = v' \left(\cos(\alpha') - |\sin(\alpha')| - |d_c'|\right), \tag{5.11}$$

with v the velocity of the car, α the angle between the car and the track-axis and d_c the distance between the car and the middle of the track. The distance d_c is normalized such that $|d_c| = 1$ at the edge of the track. Episodes are ended ($\mathbb{T} = 1$) when the car starts pointing in the wrong direction ($\cos(\alpha) < 0$), when the car stops after an initial grace period or when a lap is completed.

We use a pool of four tracks for training and testing. When we train on one track, we test the generality of the learned controller on the remaining three. The hypothesis is that the SRL objectives will encourage a representation that allows the learned policies to generalize to new tracks as well. When we perform pre-training, the experiences are from a separate fifth track (see Figure 5.2). All reported experiments are based on two trials per training track. Reported training performance is therefore averaged over 8 runs, while test performance is averaged over 24 runs. To compare the algorithms across different tracks, we here define the *performance* as the mean reward observed during an episode, normalized between 0 and 1. For each track we define 0 as the mean reward during the first episode of the trials, when the controllers are untrained, and 1 as the best observed mean reward for any single episode over all experiments performed on the same track.

Additional implementation details are given in Appendix B.

5.5 | Results

We start by comparing the performance of plain reinforcement learning to that of the two algorithms that include state representation learning considered in this chapter. For both `sim` and `alt` we choose $c_{SRL} = 0.5$ such that the 2-norms of the gradients of the state representation learning terms in the cost function with respect to the state embedding vector are about half that of the reinforcement learning term. For `alt`, we consider a version with c_{Qfix} chosen such that the 2-norm of the gradient of this term is similar to that of the reinforcement learning cost term and a version with $c_{\text{Qfix}} = 0$.

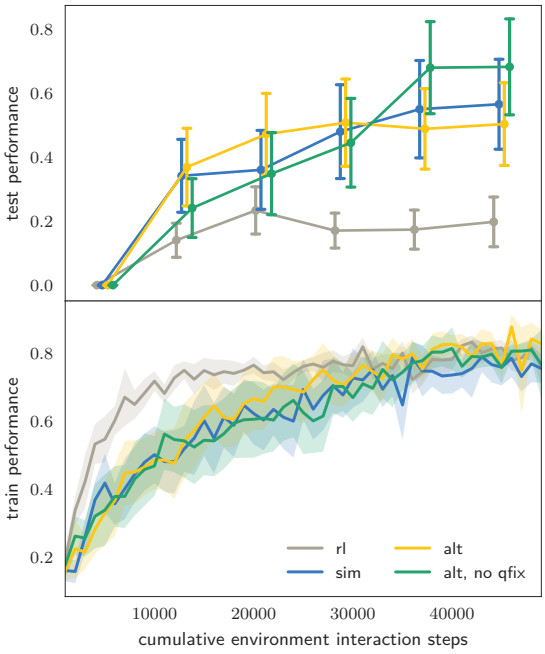


Figure 5.3: Normalized performance for the `rl`, `sim` and `alt` methods, as well as the `alt` method without the $\mathcal{L}_{\text{Qfix}}$ (5.10) loss term. For the test performance, every 10^4 steps the network parameters that gave the best training performance up to that point are evaluated on the test tracks, without any additional training on those tracks. The mean $\pm$ half a standard deviation of the performance criterion from Section 5.4 are shown.

From the results in Figure 5.3 it can be seen that all algorithms manage to find control policies that yield good performance on the tracks that they are trained on. When the learned controllers are tested on different tracks however, it becomes clear that the controllers trained by reinforcement learning alone do not generalize well. The performance when including the state-representation learning cost terms is significantly better. As can be seen from Table 5.1, the `alt` algorithm with the $\mathcal{L}_{\text{Qfix}}$ loss term consistently produced the best performing controllers on the training tracks, while the `alt` algorithm without this loss term gave the best generalization performance on three of the four tracks.

Table 5.1: Best performing algorithm (of `rl`, `sim` and `alt` with and without the $\mathcal{L}_{\text{Qfix}}$ (5.10) loss term) on each of the training track / evaluation track combinations.

evaluation:	track 1	track 2	track 3	track 4
training track 1	`alt`	`alt`(no qfix)	`alt`(no qfix)	`alt`
training track 2	`sim`	`alt`	`alt`(no qfix)	`alt`(no qfix)
training track 3	`sim`	`alt`(no qfix)	`alt`	`alt`(no qfix)
training track 4	`sim`	`alt`(no qfix)	`alt`(no qfix)	`alt`

Sensitivity to c_{SRL}

Since we are interested in the integration of RL and SRL, we have investigated the effect of changing the weight c_{SRL} of all SRL terms compared to the RL cost term. We observed that scaling c_{SRL} down from 0.5 to 0.25 and 0.05 for the `sim` method resulted in slightly better training performance, with no clear difference in test performance. Overall, the algorithms sensitivity to this hyper-parameter seems quite limited, as a change of an order of magnitude did not produce clear performance differences. Still, adapting the scale of the individual loss terms dynamically (van Hasselt et al., 2016) could be useful future work, as it could eliminate the tuning step altogether.

Learning speed

Besides the potential for regularizing the state representation in a way that benefits generalization across domains, the other main appeal of adding the extra state-representation learning objectives is making the optimization problem for the state-representation easier by providing more stable and simpler objectives. Other authors, such as Jaderberg et al. (2017), have found that the use of additional training objectives can speed up the learning process, also on the training domain. While Figure 5.3 showed that the `sim` and `alt` methods were able to find generalizing controllers more quickly, learning on the training domain was slower than that of plain `rl`.

A likely reason for these observations is that while some of the individual optimization problems posed by the SRL loss terms might be simple, we are optimizing for many of them at once, which makes the optimization more difficult. Additionally, by (softly) enforcing the physical priors we limit the space of suitable state representations, as the state representation should not only allow fitting the Q-values, but additionally adhere to the SRL constraints. While this results in the improved generalization performance, it makes finding a representation that allows decent training performance more challenging, as most SRL objectives do not limit the

parameter search space. To further investigate the cause of the slower learning we vary the experience reuse and perform an ablation study of the individual SRL loss components.

After each training episode, the newly observed experiences are added to the experience replay buffer and a number of optimization steps is performed. The number of updates is determined by the sample reuse hyper-parameter; the expectation of the number of times each experience is sampled as part of a mini-batch for an update. As the SRL objectives add richer training targets and serve as regularizers, we might expect that increasing the number of updates performed per new experience would be beneficial, especially compared to doing the same without the SRL objectives. To test this, we varied the sample reuse for both the `alt` and the `rl` algorithms from 16, which we use for all other experiments, to 8 and 32. The results are shown in Figure 5.4 and show that for our tests there is little to be gained from increasing the sample reuse beyond 16.

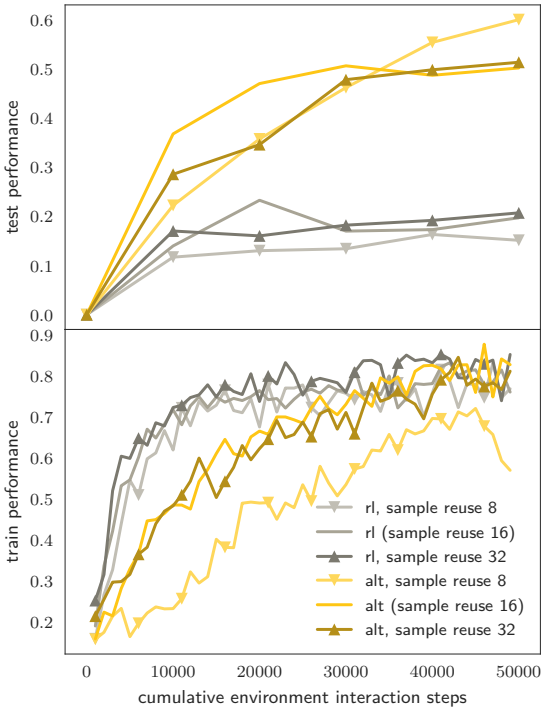


Figure 5.4: The influence of the sample-reuse hyper-parameter on the train and test performance of the `rl` and `alt` methods. For legibility only the means of the performance criterion from Section 5.4 are shown.

Individual SRL losses

We are also interested in how the individual losses contribute to both the speed of learning and the performance of the learned controllers. Figure 5.5 shows an ablation study in which, for the `alt` algorithm, each of the losses is separately turned off.

While all losses contribute to learning general control policies during the early stages of learning, the auto-encoding loss seems to hurt generalization in the later stages. Initially, this loss might help quickly shape the convolutional feature maps through the dense training targets. The input reconstruction objective is however the most general objective and the least specialized towards reinforcement learning. While the objective seems to help with the learning stability[1], it might hurt the test performance in the later stages, where it forces the state representation to capture information that, while it might help explain the variation in the sensor data on the training track, might not be relevant to the task at hand. The other losses all benefit the generalization performance.

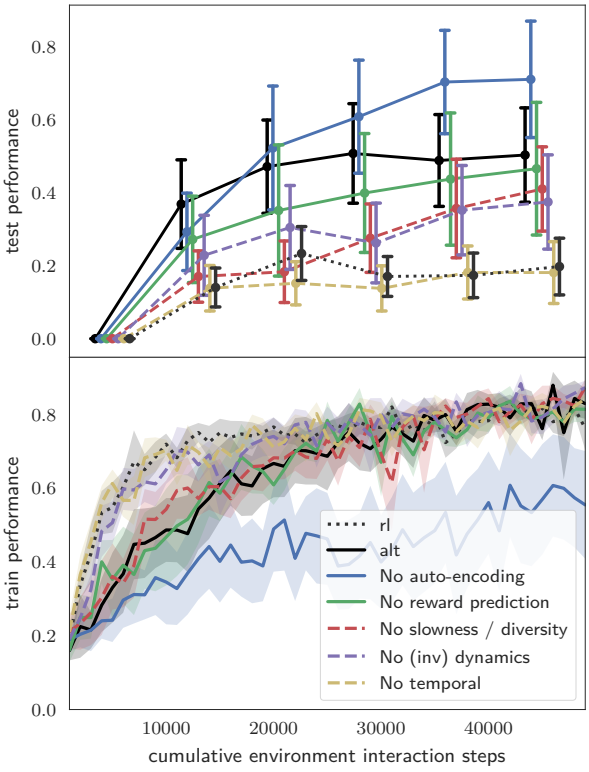


Figure 5.5: The effect on the performance of the `alt` algorithm of turning off individual SRL losses. The no temporal experiment excludes both the (inv) dynamics and the slowness / diversity losses. The performance with all SRL losses (`alt`) and with no SRL losses (`rl`) are shown for comparison. The mean ± half a standard deviation of the performance criterion from Section 5.4 are shown.

The (inverse) dynamics loss can be seen from Figure 5.5 to be the primary reason for the slower learning on the training domains, as excluding it yields a similar learning curve to the `rl` method. Leaving out either the (inverse) dynamics or the slowness and diversity loss terms results in a large drop in generalization performance.

[1]Note that since the test performance is evaluated every 10^4 steps for the best performing controllers up to that point, the test performance is less sensitive to the learning stability than the training performance.

MDP dynamics encoding

When leaving out both the slowness and diversity as well as the (inverse) dynamics losses (*no temporal* in Figure 5.5), the generalization performance degrades to the level of the plain `rl` method. This shows that, at least on the Torcs domain, explicitly learning to encode the temporal aspects of the environment into the representation of the state is the most beneficial for the generalization performance.

The incorporation of the MDP dynamics into the state representation to aid generalization is also the idea behind successor representations (Barreto et al., 2017; Dayan, 1993; Kulkarni et al., 2016). These methods use a prediction of the occupancy of future states as a representation of the current state, something that might be biologically plausible (Stachenfeld et al., 2017). While for successor representations the representation is a function of the MDP dynamics and the current policy, our state representation learning losses are all off-policy and only a function of the environment dynamics. However, the slowness objective does encourage successive states in the experience buffer to have similar representations.

Pre-training

So far we have investigated integrating state-representation learning directly into the reinforcement learning process. An alternative to this approach is to first learn a state-representation and to then perform reinforcement learning, either while keeping the representation fixed, or while allowing it to be adjusted further. We investigate the potential of pre-training using a fixed dataset obtained by a reinforcement learning controller on a separate track.

In Figure 5.6 the performance with pre-training is compared to the performance without. We pre-train using the SRL objectives with or without the RL loss. Subsequently, we train while either keeping the representation fixed (and only changing the parameters of the RL decoder), or we perform training with the `rl` or `alt` algorithms as usual. The results show that while pre-training enables quick adaption to a new track and can help generalization, the learned representation is not good enough to allow competitive performance without adaption of the representation in the on-line learning phase.

5.6 | Conclusions

We have investigated several ways of integrating State Representation Learning (SRL) objectives into standard deep Reinforcement Learning (RL). During all stages of learning, we allowed the reinforcement learning objective to help shape the state representation and we used the state representation learning objectives to regularize that representation.

The regularization resulted in a small control performance improvement on the training domain and a significant improvement on the test domain. While we combined state representation learning criteria from a number of different works, little effort was put into scaling their relative importance, and it was found that the methods were not sensitive to the hyperparameter that determined the ratio between the weights of the SRL and RL objectives. The *slowness and diversity* and the *(inverse) dynamics* SRL objectives were found to be most beneficial to the generalization performance, while the auto-encoding objective benefited the learning stability and speed at the cost of the eventual generalization performance.

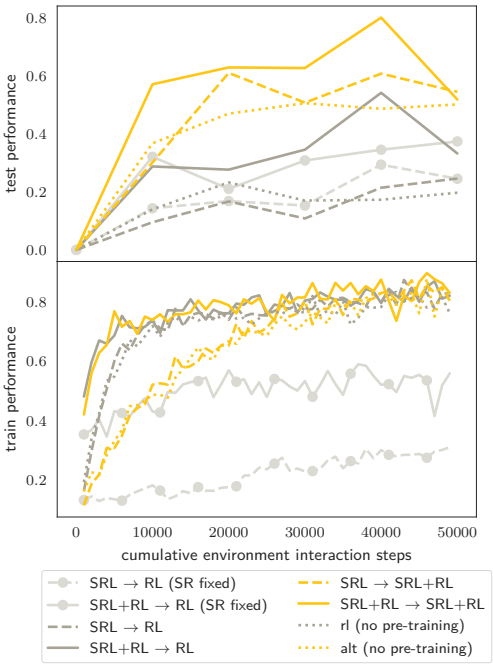


Figure 5.6: Performance when starting with a network pre-trained with data from a separate track. The notation is *pre-train method* → *train method*. SR fixed indicates that the state representation is kept fixed during the on-line learning phase and only the RL decoder is adapted. The means of the performance criterion from Section 5.4 are shown.

Compared to combining SRL and RL directly in a single update, our proposed method of alternating between these objectives yielded better performance. While limiting the changes to the value function predictions during the SRL updates consistently gave the best performance on the training domain, not doing so tended to result in better performance on the test domain.

6

Beyond Gradient-Based Optimization

Parts of this chapter are under review for the IFAC 2020 conference.

6.1 | Introduction

After having discussed in the preceding chapters *what* we would like to learn from *which* experiences, we now focus on *how* to learn these things.

As discussed in Chapter 2, deep neural network function approximation is suitable for robotics because of the functional decomposition of deep neural networks. This structure mirrors the hierarchical nature of the physical processes that generate the sensor data that robots base control decisions on (Lin et al., 2017). This makes DNNs more statistically and computationally efficient at processing these natural data than alternatives that do not have a hierarchical structure (Bengio et al., 2013).

While the DNN controllers are often trained end-to-end to map raw sensor data to actuator commands, the hierarchy of functions that is encoded by their layers could be thought of as representing two distinct sub-functions. The first is a mapping from the high-dimensional sensor data to a concise lower-dimensional representation of the task-relevant aspects of the environment state. The second is a mapping from this state representation to the action that needs to be taken in that state to accomplish the task at hand. In the previous chapter we made use of this decomposition by training the first mapping with additional SRL objectives.

To learn both functions, stochastic gradient-based optimization techniques are most commonly used. When good enough estimates of the true parameter gradients can be obtained, these techniques can efficiently find good values for the network parameters. In this chapter we will take the view that, for the mapping from observation to state representation, sufficiently good gradient estimates can indeed often be obtained. For an interesting class of problems many, if not most, of the parameters of the DNN controller will be used to encode this mapping. This mapping can be learned either implicitly through end-to-end reinforcement learning, or explicitly by using state representation learning objectives (as in the previous chapter and e.g. de Bruin et al., 2018b; Finn et al., 2016; Jonschkowski and Brock, 2015; Lange et al., 2012b). Intuitively, the sensory observations are a direct result of the latent state of the environment. While we do not have access to the true state, there are many objectives that will allow us to learn to infer the task-relevant aspects of this state relatively easily. These include reconstructing observations, predicting action-dependent changes to the state representation or observations, predicting the instantaneous rewards, and more. Even when hand crafted state representations are available, learned state representations can sometimes enable better policies (Levine et al., 2016).

The mapping from the state representation to the optimal action in that state often requires fewer parameters and can be simpler. However, this mapping can

still be much harder to *learn*. This is because the effect of any single action on the eventual task performance is often rather small, and there might be delays before the consequences of actions become apparent. Getting high-quality estimates of the gradients of the task performance with respect to the network parameters through the chosen actions is therefore difficult. These difficulties show up in different forms, depending on how the parameter gradients are obtained. Techniques using policy rollouts are able to get unbiased estimates of the policy gradient, but suffer from very high variance, while techniques using bootstrapping suffer from biased gradients (Marbach and Tsitsiklis, 2003; Schulman et al., 2015b). For both extremes, as well as the methods that trade off bias and variance by using both rollouts and bootstrapping, the low-quality parameter gradients can make the stochastic gradient optimization process diverge (Gu et al., 2017b; Henderson et al., 2017; Sutton and Barto, 2018). As discussed in Chapter 2, many different strategies have been used to deal with these problems—generally trading in learning speed and data efficiency for learning stability. Examples include target networks (Mnih et al., 2015), experience replay buffers (Lin et al., 2017; Mnih et al., 2015), trust region updates (Schulman et al., 2015a, 2017; Wang et al., 2017) and very large batch sizes (Bansal et al., 2017). While these techniques ameliorate the problem, DRL is still notoriously sensitive to hyperparameter tuning and prone to divergence (Henderson et al., 2017).

An alternative to these attempts to deal with low-quality parameter gradients is to use gradient-free optimization techniques, such as Evolutionary Strategies (ES) (e.g. Koutník et al., 2013; Salimans et al., 2017). Rather than trying to assign credit to the policy parameters through the individual actions that were taken, gradient-free techniques assign credit to parameter vectors directly based on entire trajectories. These techniques tend to be much less data efficient than gradient-based techniques, but they can lead to more stable convergence of the policy performance. They also have other benefits like their ability to optimize policies that need to make decisions at high sampling frequencies, and the fact that their training is highly parallelizable (Salimans et al., 2017).

In this work we combine the desirable aspects of both gradient-based and gradient-free optimization techniques for DNN controllers. We start with a gradient-based phase in which we use standard deep reinforcement learning techniques. This allows us to quickly learn a policy that is good enough to collect relevant training data and learn a state representation. We then freeze the state encoder part of the policy, and tune the final action-selection parameters further using a gradient-free technique. Since we are only tuning relatively small number of parameters, we use the CMA-ES algorithm (Hansen and Ostermeier, 2001). This algorithm is not only relatively sample efficient for an ES algorithm, but also includes a

natural way of decaying the amount of exploration (Stulp and Sigaud, 2012). This makes it possible to perfect the policy while reducing the probability of poor performances. Note that reducing the amount of exploration while training the entire policy—including the state encoder—using gradient-based optimization can lead to over-fitting and destabilize the learning process (see Chapters 3, 4 and de Bruin et al., 2018a).

6.2 | Related work

The method we propose in this chapter is perhaps most closely related to that of Ha and Schmidhuber (2018). In their work, a random policy is used to collect training data, which is used to train a state encoder using state-representation learning objectives. Then, the action-selection subnetwork of the policy is trained from scratch using CMA-ES. While we consider the use of state representation learning objectives *in addition* to reinforcement learning, we rely mainly on RL to learn the whole policy during the gradient-based learning phase. This not only results in more relevant training samples (enabling a better representation to be learned (Pérez Dattari et al., 2019)), but also a good initialization of the action-selection parameters. We show in Section 6.4 how these changes allow us to solve the CarRacing-v0 task using forty times fewer episodes, while still obtaining a considerably better final policy.

Evolutionary Strategies (ES) have also been used to optimize *all* parameters of deep neural network controllers (e.g. Hausknecht et al., 2014; Salimans et al., 2017). These methods are able to scale across many CPUs in an efficient way. They are however not very sample efficient. The fact that we only optimize a small number of parameters using ES helps us to limit the sample inefficiency of the method. It also allows us to use CMA-ES which does not scale to large parameter vectors, but is able to optimize small parameter vectors in a more sample-efficient manner than other ES strategies (Hansen and Ostermeier, 2001). We additionally show in Section 6.4 that even when we do optimize all parameters of a policy using ES (for small NNs), initializing the parameters using a short gradient-based optimization phase before starting the ES optimization can help to both speed up the learning as well as improve the eventual performance.

The idea that the last layer of a DNN is harder to train using gradient-based reinforcement learning than the preceding layers was previously explored by Levine et al. (2017). In their work, the instability of the DQN method was limited by performing least squares updates of the parameters of the final layer in addition to the standard gradient updates.

The authors of Plappert et al. (2018) also consider the final layers of the policy to represent the action selection part and add noise to these parameters to enable exploration. We use their technique during the gradient-based learning phase and update only these parameters through CMA-ES during the policy fine-tuning phase. Here, the CMA-ES can be understood as an optimization algorithm for both the exploration policy as well as the policy parameters (Stulp and Sigaud, 2012).

6.3 | Main Contribution: Optimization Method

In this work, the training of the deep neural network controller consists of two distinct phases; a gradient-based optimization phase and a gradient-free phase. The aim of the initial gradient-based learning phase is twofold: we want to efficiently learn a state representation that can be used for control and we want to find a good initialization of the action-selection subnetwork of the policy. After this phase is complete, we will trade in the learning speed for stability by further tuning the action-selection parameters using a gradient-free evolutionary strategy.

So far in this thesis, we have optimized for the discounted return (2.1) with $\gamma < 1$. However, in many episodic tasks—such as those in this chapter—we are interested in optimizing for the undiscounted return ($\gamma = 1$). We still do use the undiscounted return as a surrogate optimization objective during the gradient-based optimization phase, as this makes the optimization easier (Marbach and Tsitsiklis, 2003; Schulman et al., 2015b). The subsequent gradient-free phase will allow for stable convergence of the policy performance, while optimizing for the true (undiscounted return) objective.

6.3.1 DNN Controllers

The control policy that maps observations o to actions a is parameterized as a deep neural network with parameters θ: $a = \pi(o; \theta)$. The network consists of m layers, of which we consider the first n to represent the state encoder, which maps observations to a state representation: $\bar{s} = \bar{S}(o, \theta_s)$. The final $m - n$ layers are considered to represent the action selection subnetwork, which encodes the mapping from this state representation to the policy action: $a = \Psi(\bar{s}; \theta_a)$ with $\pi(o; \theta) = \Psi(\bar{S}(o; \theta_s); \theta_a)$. In addition to the policy action, several other predictions can be made based on the state representation $\bar{s}$. In the DRL methods considered in this work these include the return estimates $\hat{Q}(\bar{s}, a; \theta_q)$ and optionally additional state representation learning predictions, as shown in Figure 6.1.

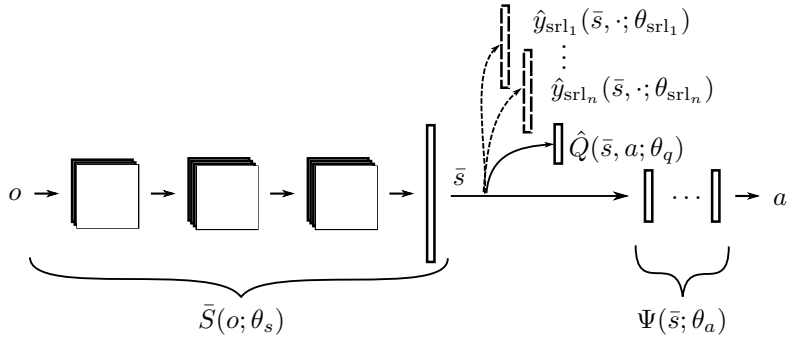


Figure 6.1: We learn DNN control policies that map observations to actions: $a = \pi(o; \theta)$. We consider the first n layers of m-layer DNN to encode a mapping from observations to a state representation: $\bar{s} = \bar{S}(o; \theta_s)$. The parameters θ_s that encode this mapping are learned through gradient-based optimization by fitting value functions, optionally supplemented with state representation learning objectives. The final $m - n$ layers encode the mapping from the state representation to actions: $a = \Psi(\bar{s}; \theta_a)$. The parameters θ_a are initialized during the gradient-based learning phase and then fine-tuned during a gradient-free optimization phase.

6.3.2 Gradient-based optimization

For the gradient-based optimization of θ we use two simple and popular deep reinforcement-learning algorithms. For policies with discrete actions we use the DQN algorithm (Algorithm 1, Mnih et al., 2015). For continuous actions we use the DDPG algorithm (Algorithm 2, Lillicrap et al., 2016). The experience samples $\{o, a, o', r, \mathbb{T}\}$ are collected by following an exploratory policy $\tilde{\pi}$ (defined below). These samples are stored in an experience buffer, from which they are sampled uniformly at random to calculate training targets $q(o, a)$ (line 15 of Algorithm 1 for DQN, line 14 of Algorithm 2 for DDPG). The observation-action estimation function $\hat{Q}(\bar{S}(o; \theta_s), a; \theta_q)$ is trained by minimizing the squared temporal difference error (5.1) through stochastic gradient descent. For the DQN algorithm, the return predictions for all actions are estimated for a given state representation $\bar{s}$ by a linear layer. In the DDPG algorithm, a neural network is used that takes both o and a as inputs and outputs the predicted expectation of the return.

For the DQN algorithm, we follow Plappert et al. (2018) in having an explicit policy head that is separate from the value function estimation (but uses the same state representation). This head is trained using the negative log likelihood objective to predict the action with the highest Q-value, given the state representation.

6.3.3 Parameter space exploration

During both the gradient-based and the gradient-free phase of the optimization, parameter space exploration is used to explore the state-action space. The ex-

ploratory policy is given by $\tilde{\pi} = \Psi(\bar{S}(o;\theta_s);\theta_{\tilde{a}})$. The parameters $\theta_{\tilde{a}}$, representing the exploratory version of the action-selection part of the policy, are re-sampled at the start of every $\mathcal{V}^{\text{th}}$ episode according to:

$$\theta_{\tilde{a}} \sim \mathcal{N}(\mu, \sigma C), \tag{6.1}$$

where the choice and evolution of μ, σ and C depend on the optimization phase.

Exploration during the gradient-based optimization phase

During the gradient-based (reinforcement learning) phase of the algorithm, a new exploratory policy is sampled every episode ($\mathcal{V} = 1$). The parameters are sampled from an isotropic mutation distribution ($C = I$) which is centered around the parameters of the current policy ($\mu = \theta_a$). The scaling of the parameter noise σ is adjusted according to the method of Plappert et al. (2018):

$$\sigma_{\nu+1} = \begin{cases} \alpha\sigma_\nu & \text{if } d(\pi, \tilde{\pi}) \leq \delta, \\ \frac{1}{\alpha}\sigma_\nu & \text{otherwise.} \end{cases} \tag{6.2}$$

To ensure that the scale of the parameters to which this noise is applied is not too different, layer normalization (Ba et al., 2016) is used on the perturbed layers (Plappert et al., 2018). The distance measure d and threshold δ relate the exploration scale σ to action space exploration, allowing for more intuitive hyperparameter choices. For DDPG, we follow Plappert et al. (2018) in using

$$d_{\text{DDPG}}(\pi, \widetilde{\pi}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \mathbb{E}\left[\left(\pi(o)_i - \widetilde{\pi}(o)_i\right)^2\right]},$$

where N is the dimensionality of the actions and the expectation is estimated over a batch of samples from the experience buffer. Using this distance measure, the threshold value can be chosen as $\delta \doteq \sigma_a$ to get exploration with the same standard deviation from the policy in the action space as normally distributed noise with a standard deviation of σ_a. For DQN, we do deviate from the method of (Plappert et al., 2018) and simply count the fraction of observations per episode for which $\pi \neq \tilde{\pi}$ which we compare directly to the desired epsilon greedy action space exploration fraction: $\delta = \epsilon$.

6.3.4 Gradient-free optimization phase

During the second phase of learning, we first restore all network parameters θ to the values θ^* that resulted in the highest undiscounted return so far. We then use the gradient-free CMA-ES (Hansen and Ostermeier, 2001) optimization procedure to further optimize the action-selection parameters θ_a.

We start by initializing a normal distribution (6.1) with:

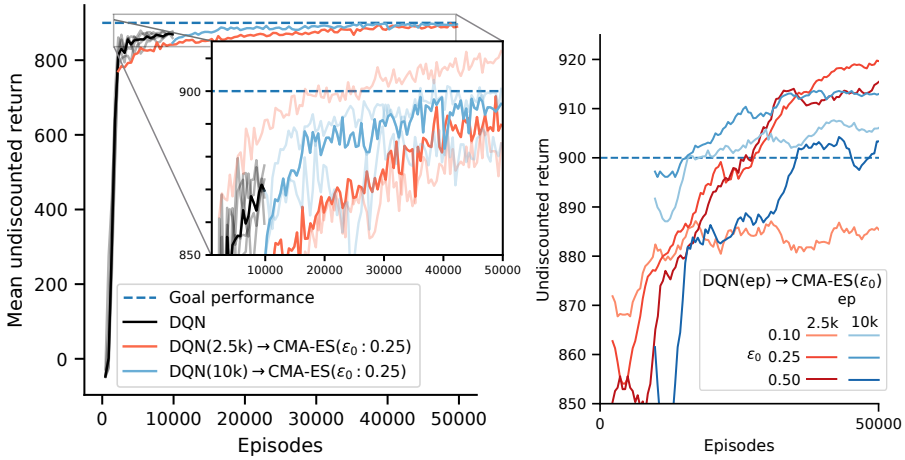
- $\mu_0 = \theta^*_{\tilde{a}}$ (the parameters that led to the best performance during training),
- σ_0: we use the procedure of (6.2) to adapt σ based on a desired exploration intensity in the action space.
- $C_0 = I$ (the mutation distribution starts out isotropic, but is adapted over time in contrast to the exploration during the gradient-based phase).

After this initialization, the CMA-ES algorithm then adapts μ, σ and C by iteratively sampling λ parameter vectors $\tilde{\theta}_a$ from the distribution and updating the distribution based on their fitness. For the evaluation of the sampled parameter values $\tilde{\theta}_a$, we perform $\mathcal{V}$ roll-outs of the exploration policy $\tilde{\pi}(\cdot; \theta_{s+\tilde{a}})$ and average the returns. The values of μ, σ and C are updated so that the parameters corresponding to the higher fitness scores are more likely under the updated distribution. In this update, previous updates are also taken into account to speed up the learning. For a more detailed description of the CMA-ES procedure we refer to Hansen et al. (2019); Hansen and Ostermeier (2001). One important aspect of the CMA-ES algorithm is that the intensity (σ) and shape C of the exploration are adapted automatically. During the final phase of learning, this can allow the optimization procedure to reduce the exploration intensity in a controlled manner (Stulp and Sigaud, 2012).

6.4 | Experiments

In the following we write `DRL(time)`→CMA-ES(`exploration`) to indicate that we use the `DRL` deep reinforcement learning algorithm (either DQN or DDPG) for `time` episodes or environment steps before switching to CMA-ES where we initialize σ_0 (6.1) to `exploration`. We use the CMA-ES implementation of Hansen et al. (2019). In all experiments, we use their default population size of $\lambda = 4 + \text{floor}(3\ln(n))$, where n is the number of elements in θ_a.

We start with experiments on the OpenAI Gym CarRacing-v0 benchmark (Brockman et al., 2016). In this task, the observation o is a top-down image of a car on a randomly generated racing track of which it needs to complete a lap as quickly as possible. Only a single image is provided at each time-step, where the car's translational and angular velocities, wheel speeds and steering wheel position are encoded as bars in the image. The task of the state encoder $\bar{S}(o; \theta_s)$ is therefore to decode this information, along with all other relevant information, and include it in the state representation $\bar{s}$. We discretize the action space (of throttle, breaking and steering inputs) into 7 actions (detailed in Appendix B). The aim of the action selection subnetwork $\Psi(\bar{s}; \theta_a)$ is to select the best action in a very reliable



(a) Average of the population

(b) Highest in the population

Figure 6.2: Mean undiscounted return over $\mathcal{V} = 16$ episodes on the CarRacing-v0 benchmark. (Median) results over 3 runs are shown.

way. Solving the task is defined as getting a mean score of at least 900 over 100 subsequent episodes, which means very few mistakes are allowed.

The need for a very precise and reliable policy based on a learned state representation makes this benchmark interesting for our proposed method. It also allows for a comparison with the related work of Ha and Schmidhuber (2018) who chose this benchmark for similar reasons. On this benchmark we use the DQN architecture (Mnih et al., 2015) with an added policy head (Plappert et al., 2018). We consider all but the final layer to be the state encoder $\bar{S}$, making the state representation 512 dimensional. This leaves a final layer with $\theta_a \in \mathbb{R}^{3591}$ for the action selection Ψ. For the CMA-ES phase we sample $\lambda = 28$ values of the parameter vector $\theta_{\tilde{a}}$ and evaluate each over $\mathcal{V} = 16$ episodes. Each iteration of the CMA-ES algorithm therefore consists of 448 episodes.

We use this benchmark to test the assumptions on which our method is based: that the state-encoder part of a good policy can quickly be learned through deep reinforcement learning and that stable convergence to a better performing policy can be achieved through gradient-free fine-tuning of the final action-selection parameters. Figure 6.2a shows both parts in action. The outer graph shows the relative speed with which the gradient-based DRL phase can learn the values of the 1.7 million parameters in θ_s and initialize the 3591 parameters in θ_a to a point where the task is performed reasonably well. The popout shows the stability with which the parameters of θ_a can subsequently be tuned further using the gradient-

Table 6.1: Performance over 100 test episodes on the CarRacing-v0 benchmark. Tested policies were those that set the highest (mean) undiscounted return during training for the first (of 3) training runs.

Method	Episodes	Score
SRL → CMA-ES (Ha and Schmidhuber, 2018)	1,843,200	906 ± 21
DQN(2500ep)	2,500	871 ± 86
DQN(2500ep) → CMA-ES($\epsilon_0 = 0.10$)	50,000	890 ± 34
DQN(2500ep) → CMA-ES($\epsilon_0 = 0.25$)	50,000	**918 ± 20**
DQN(2500ep) → CMA-ES($\epsilon_0 = 0.50$)	50,000	915 ± 28

free CMA-ES procedure, learning not just to solve the task but also beating what is to the best of our knowledge the highest reported score in the literature (Ha and Schmidhuber, 2018) while using considerably fewer episodes, as shown in Table 6.1.

Given that we have two distinct optimization phases, one important question is when we should switch from the gradient-based phase to the gradient-free phase. A closely related question is how much (initial) exploration around the action-selection parameters $\theta_{\tilde{a}}^*$ is beneficial. Figure 6.2b shows the mean undiscounted return for the best parameter vector $\theta_{\tilde{a}}$ sampled per iteration of the CMA-ES procedure, as a function of the number of DRL episodes and the initial exploration ϵ_0 that σ_0 is adapted to using (6.2). It can be seen that more gradient-based optimization steps might limit the potential for subsequent gradient free improvement. To see whether this effect is related to over-fitting in the state encoder or the action-selection parameters, we perform two experiments.

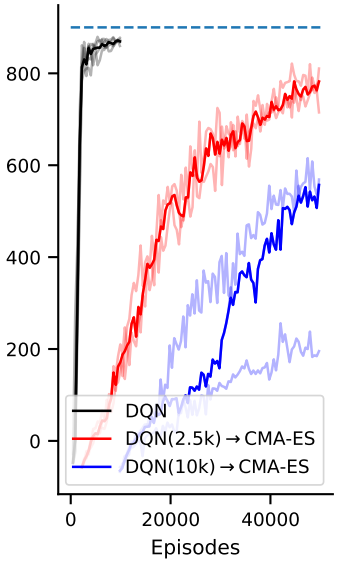


Figure 6.3: Mean undiscounted return over $\mathcal{V} = 16$ episodes on the CarRacing-v0 benchmark. Population average while training $\theta_{\tilde{a}}$ from scratch. Results from 3 runs are shown.

In the first we train θ_a from scratch using CMA-ES, rather than starting from $\theta_{\tilde{a}}^*$. This is shown in Figure 6.3. We see again that when using a state encoder that is trained for fewer episodes, the subsequent gradient-free optimization of the action selection subnetwork is easier. The sample efficiency benefit of starting from the pre-trained $\theta_{\tilde{a}}^*$ can also

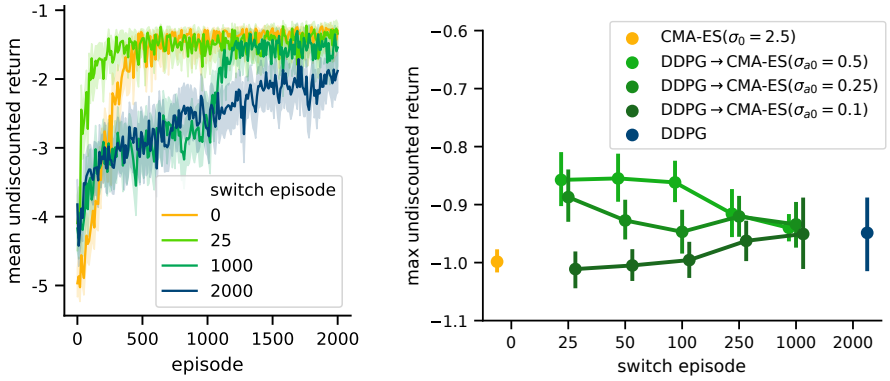
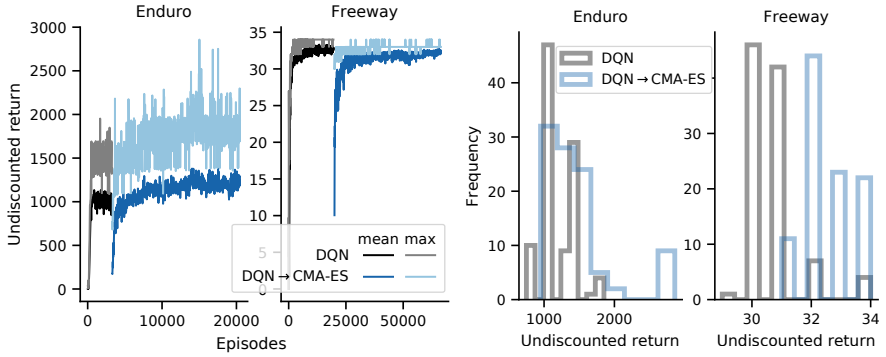


Figure 6.4: Magman benchmark results. Mean undiscounted return per episode for DDPG(switch episode)→CMA-ES($\sigma_{a0} = 0.25$) (left) and maximum undiscounted return per learning trial (right). Results are from 50 trials, with the 95% bootstrapped confidence bounds of the means shown.

be clearly seen when comparing Figure 6.3 with Figure 6.2a. As at least part of the limitation of the performance improvement when switching later seems to be related to over-fitting in the state encoder, we perform an experiment where state representation losses are added to the state encoder learning objective to help regularize the state representation (as in the previous chapter and de Bruin et al., 2018b). The result of this experiment is shown in Figure C.11 of the appendix. We indeed observe that switching later now leads to better performance.

Due to the computational complexity of these methods, these results were from three runs. For more statistically significant results we perform experiments on the Magman benchmark (de Bruin et al., 2018a). This benchmark is low-dimensional, but requires a very precise control policy with continuous actions. We use two smaller networks with 2 hidden layers of 64 units each for a policy and a Q-function. During the gradient-based optimization we use the DDPG algorithm to train these networks. During the CMA-ES phase, we optimize all parameters of the policy. The results, shown in Figure 6.4, again demonstrate the benefits of this two stage optimization. Switching early, with sufficient initial exploration, results in both faster learning as well as a higher maximum performance than either DDPG or CMA-ES alone.

Finally, we tested the method on two Atari games. We followed the exploration strategy and network architecture of (Plappert et al., 2018). This meant that in the RL phase, ϵ decayed linearly from 1 to 0.1 during the first one million episodes and remained constant afterwards. For the DQN architecture it meant



(a) Learning curves **(b)** Test performance

Figure 6.5: Train and test performance on two Atari games.

that the policy head was implemented as a single fully connected layer, right after the convolutional layers of the DQN architecture. The $\hat{Q}$ head still had the usual 512 dimensional fully connected intermediate layer. While we found that this architecture worked better during the DRL phase, it meant that the final action selection layer now contained more parameters than can feasibly be optimized using CMA-ES. Therefore, we trained a new policy head—which did have the 512 dimensional intermediate layer—after the DRL phase. This head was trained to minimize the KL divergence between its predictions and the predictions of the original policy head on samples from the replay buffer (Parisotto et al., 2015). We then used CMA-ES to optimize the final layer of this new policy head. When testing the controllers resulting in the highest undiscounted returns during both phases (evaluated with $\mathcal{V} = 1$), we again observe that the CMA-ES procedure was able to noticeably improve the policy performance by fine-tuning the final action-selection parameters. Results are shown in Figure 6.5 and Table 6.2. For the Enduro benchmark, where episodes can be very long and consequences of actions less immediate, the gradient-free optimization also resulted in the highest outright scores. On the freeway benchmark, with short episodes and more immediate consequences to actions, DRL was able to find a near optimal policy. The gradient-free optimization phase here found a policy that has a higher mean performance over 100 episodes, but did not obtain the maximum score.

Table 6.2: Performance over 100 test episodes on two Atari benchmarks.

	Enduro		Freeway	
Method	Steps	Mean ± SD	Steps	Mean ± SD
DQN(50m)	50m	1188 ± 240	50m	30.7 ± 0.9
DQN(50m) → CMA-ES($\epsilon_0 = 0.5$)	250m	**1483 ± 505**	100m	**32.6 ± 1.0**

6.5 | Conclusion and future work

In this chapter we combined gradient-based deep reinforcement learning methods with a gradient-free evolutionary strategy. We showed how a relatively short initial gradient-based phase was able to learn a good state representation and a decent action selection strategy relatively quickly, while a subsequent gradient-free fine-tuning of the action-selection parameters resulted in stable convergence to a policy performance not achieved with gradient-based optimization alone. Experiments on a small scale benchmark, where no state encoder needed to be trained, also showed how the combination of gradient-based and gradient-free optimization was able to learn more quickly, and find better performing policies, than either method alone.

The results suggests several avenues for future work. When we consider part of the network to represent a state encoder, we freeze this encoder during the gradient-free fine-tuning of the policy. However, if the improved policy that is found through gradient-free optimization visits significantly different states, it might be worth updating the state encoder using the data obtained with this new policy. On the other hand, if we do keep the state encoder frozen, we can consider using model compression (Hinton et al., 2015; Moniz et al., 2019) on this part of the network to speed up the evaluation of the gradient-free phase further.

When optimizing all parameters of a neural network, it might be interesting to investigate a tighter integration of deep reinforcement learning with parameter-space exploration and CMA-ES. For instance by introducing the ability of CMA-ES to scale and shape the exploration to the gradient-based optimization phase, or by allowing the DRL gradients to bias the update direction of the population mean of CMA-ES.

7

Conclusions & Outlook

This thesis has investigated how reinforcement learning using deep neural network function approximation can be made to work more efficiently under the constraints imposed by the robotics domain. In this chapter, we will summarize the main conclusions and contributions of the work, reflect on the state of the field, and outline possible avenues for future work.

7.1 | Conclusions

Before applying Deep Reinforcement Learning (DRL) to robotics an understanding is needed of how, when, and why the combination of deep learning and reinforcement learning works. To this end, *Chapter 2* has provided a review of existing literature. An important conclusion that can be drawn from this survey is that deep neural networks can not simply be seen as a plug-and-play function approximator to be used with traditional reinforcement learning algorithms. Instead, several properties of deep neural networks need to be properly accounted for to make the combination work. These properties include the fact that deep neural networks are global function approximators and that the stochastic estimates of their parameter gradients therefore need to be, in expectation, representative of all the relevant parts of the function domain. Besides this, the gradient estimates need to be accurate; with limited bias and variance. While this is true for other reinforcement learning methods and for supervised deep learning methods as well, the detrimental effects of poor gradient estimates can be especially severe in DRL. This is due to the fact that a poor local update can have global consequences that affect the training data distribution as well as the training targets. To obtain parameter gradient estimates that are of adequate accuracy and sufficiently representative of the function domain, several components are shared among many successful DRL methods. One of these components is the use of delayed targets, which reduce the correlations between the training targets and predictions. Another is the use of trust region updates, which reduce the probability of steps in the parameter space that lead to reduced performance. Possibly the most widely used

component that makes the combination of reinforcement learning with deep neural networks work are experience buffers, which ensure that parameter estimates are calculated that are representative of the function domain.

Despite their widespread use, experience replay techniques are often very simplistic. Most commonly, experiences are sampled uniformly at random from a buffer which contains a given number of the most recent experiences. This can cause problems in the robotics setting, due to the costly nature of exploration. When using standard experience replay techniques in combination with a decay in exploration, the coverage of the function domain by the experiences sampled from the buffer will shrink over time. *Chapter 3* investigates the effects this has on the performance of DRL methods. By assuming access to the true system dynamics, an investigation was carried out into what distribution over the state-action space was desirable for the learning algorithms to work well. It was found that the need for specific data distributions was most strongly related to the used RL algorithm, as well as the properties of the control problem at hand. Specifically, it was found that actor-critic algorithms can be sensitive to the diversity in the action space of the training data. When the training data become insufficiently diverse, previously learned successful behaviors can quickly be forgotten by these algorithms. This effect is stronger for control problems with high sampling frequencies, where the effect of a single action is smaller. It was additionally shown how the benefit of a diverse data distribution depends on the ease of generalizing the policy or value function over the state-space.

In *Chapter 4* this knowledge of desirable training data distributions was used to propose practical algorithms for managing the experience buffer. In this chapter it was no longer assumed that the dynamics model was available. Additionally, no influence over the stream of experiences observed by the agent was assumed, as it might be useful to learn from tele-operation, other agents, or experiences obtained while learning different tasks. Instead, the short and long-term utility of experiences needed to be determined in order to decide which experiences should be retained in the buffer as well as which experiences should be sampled from the buffer for the learning updates. It was concluded that there is no single metric to identify useful experiences. Instead, the use of proxies based on the age of the experience, the amount of exploration it represents, and the surprise it causes in the learning agent were proposed. It was found that prior knowledge about the control problem at hand could be used to choose among these proxies, select the size of the experience buffer and determine whether importance sampling should be used. We found that these decisions could help improve the stability and speed of the learning process and lead to better controller performance. Crucially, it was concluded that retaining the right experiences in memory makes it possible to

overcome the tendency of DRL algorithms to forget good behaviors when reducing exploration, an important step towards making these methods suitable for robotics.

Besides leading to challenges, the properties of deep neural networks also provide opportunities for DRL in the robotics domain. One source of these opportunities is the fact that DNNs encode functions that are made up of shared sub-functions. For an interesting class of problems, these sub-functions go from being very general to being very task specific when going from the input to the output of the network. Specifically, we can view the first part of a neural network as a state encoder, which extracts a compact and concise representation of the state of the world from the high dimensional sensor data. This part of the network can be trained not just in an end-to-end fashion with reinforcement learning, but also more explicitly by using State Representation Learning (SRL) objectives. *Chapter 5* investigated this option. A number of SRL objectives were added to the DRL training procedure. We showed how this leads to more general policies that work not just in the training domain but also generalize to test domains. Using just reinforcement learning, a policy is learned that over-fits to the training domain and fails to work in the test domains. While these SRL objectives can thus be viewed as effective regularizers of the state encoder, their inclusion in the training procedure can lead to problems. This is because the changes to the state representation that these objectives cause can change the policy predictions in unforeseen ways. To address this shortcoming, we proposed a method that alternates between improving the state representation—while optionally minimizing the changes to the RL predictions—and optimizing the RL predictions. We showed how this can improve the performance relative to simply optimizing for all objectives simultaneously, which was already significantly better than only optimizing for only the reinforcement learning objective.

Diverse data, combined with SRL and RL objectives, can enable the training of a good state encoder through gradient-based optimization strategies. Learning the mapping from the state representation to the optimal actions is often more of a challenge. While gradient-based DRL can quickly initialize this mapping to a point where reasonable performance is obtained, it is much more difficult to get good gradient estimates for this part of the network. This tends to cause instability in the optimization and prevents stable convergence towards an optimal policy. In *Chapter 6* a strategy was therefore proposed to optimize these final parameters in a gradient-free manner. While gradient-free methods (such as evolutionary strategies) have been used to train entire DNNs, these strategies tend to be much less data efficient than their gradient-based alternatives. On the other hand, the fact that a population of parameter vectors is maintained and that the performance of each one is tested empirically (rather than estimated) means that getting lost in

the gradient space is less likely. In several experiments we showed how, by starting from the solution found using the common gradient based methods and fine-tuning the final sub-mapping in a gradient-free way, the speed of gradient-based methods can be combined with the stability of gradient-free methods. In this way, better controllers were found than when using either method in isolation. The use of the CMA-ES method for fine-tuning only the final parameters also allowed for a natural decay of exploration, without the previously discussed diversity problems this would cause for gradient-based methods.

The work in this thesis suggests that deep reinforcement learning can be used to enable robots to learn new behaviors and fine-tune old ones, provided the combination of reinforcement learning and deep learning can be made to work in a stable and sample efficient manner. For the sake of sample efficiency, we have looked into off-policy reinforcement learning techniques. While these techniques allow for the use of data from an arbitrary policy (which could greatly increase the sample reusability), we showed how the data distribution that makes DRL work best is dependent on factors relating to the specific RL method as well as the benchmark. We then argued that for both the stability and sample efficiency of DRL methods, it is favorable to see a policy as being divided in separate state-encoder and action selection parts. Training the first part with additional SRL objectives enabled learning general policies more quickly. Using a different optimization technique for the latter part enabled more stable learning behavior. All of these parts help make deep reinforcement learning more stable and sample efficient. This in turn might make robots a little bit less dumb. However, until additional significant breakthroughs are achieved, general purpose robots that learn to perform novel tasks without being painstakingly reprogrammed will remain confined to fiction.

7.2 | Discussion and Outlook

The work that this thesis has reported was done during an exciting time for the deep reinforcement learning field. When the work started, the field had just had its first headline grabbing result of playing Atari games from pixels. This result led to much excitement about the potential of DRL and a flurry of research ensued. During this early phase, the excitement about the kind of problems that could suddenly be solved with these methods meant that the proper theoretical and empirical validation were sometimes overlooked. More recently, the field has been maturing and there has been a stronger call for reproducible, more thoroughly validated results.

One of the most important challenges for the field might be in figuring out what it takes to properly understand these methods and how to test them in a way that is

statistically sound and yet computationally feasible on a reasonable budget. The reason that difficulties are present here seems to stem from the complexity caused by the interdependent components of deep reinforcement learning; the model predictions (policy), the training data collection procedure, and the training targets. On one end of the spectrum, we can focus on a single component, abstract everything else away and perform a rigorous theoretical analysis. When we subsequently apply the theory obtained in this way to the full DRL method, we will more likely than not find that the interactions with the other components mean that the conclusions that were drawn from the analysis do not entirely transfer. On the other hand, we could leave the black box intact and do an empirical study. However, the large combinatorial space of the possible parameters who's influence we would like to understand means that statistically significant results can only be obtained for very simple problems. These problems might not be representative of the kind of problems for which the use of DRL actually makes sense. When we scale up to the kind of problems that make the usage of deep neural networks beneficial, it becomes computationally infeasible to get statistically significant results. Since these different options are problematic in different ways, different lessons can be taken from them. It therefore seems that for now, using at all of these problematic options—while staying aware of their respective biases—is the best option. In the future, the answer might have to be sought in new standardized benchmarks that specifically test for the different problems and opportunities of the DRL combination.

When the problem of how to answer DRL related research questions is solved, many interesting questions are left to be answered. DRL can be seen as reinforcement learning that happens to use DNN function approximation or as DNN training that happens to use an RL objective. While both vantage points can be very insightful, the latter might offer more low hanging fruit. The field of supervised deep learning is more popular and mature than that of DRL and new insights into the workings of DNNs are discovered regularly, often calling into question previous theories. It would be good to take the knowledge that is generated by this field and apply it to DRL. While this thesis has attempted to investigate how the best can be brought out of deep neural networks when they are trained—at least in part—through reinforcement learning, it has only scratched the surface of this topic.

One important theory about why and how deep learning works is the manifold hypothesis. It states that natural data in high-dimensional spaces are clustered around low-dimensional nonlinear manifolds. Deep neural networks are able to make predictions about data living in high-dimensional spaces because they learn a mapping to (and from) this manifold. Rather than simply training a DNN end-

to-end through reinforcement learning, the ideas in the later half of this thesis can be seen as treating the mappings to these manifolds separately from the mapping from the manifold to the control actions. In Chapter 5 we used additional SRL objectives to learn the mapping from the observations to the manifold, while Chapter 6 used a different optimizer for the mapping from the manifold to the actions. The separation could be taken further. It would be interesting to ask the questions related to the training data distribution that were asked in Chapter 3 separately for the state encoder (the mapping to the manifold) and the mapping from the manifold (or state) to the actions. In fact, we could take a step further back. While this thesis has motivated the use of deep neural networks for robotic control problems, this motivation mostly applied to the state encoder. It would be interesting to use a different (more stable and local) method for the state to action mapping. While the state encoder should be general, and able to deal with high dimensional data, the mapping from the learned state representation to actions might not have these constraints. If these parts are (partly) uncoupled, the action selection mapping could be performed by a local method on the manifold—trained with on-policy data—while the state encoder is still trained globally—potentially using data that is less closely related to the optimal policy than when the entire policy is trained with the same data.

To make DRL more feasible for robotics, the sample complexity of DRL algorithms will need to be brought down significantly further. To contribute to this goal, this thesis has focused on extracting as much knowledge as possible from collected data. An alternative to this is to increase the use of prior knowledge. This prior knowledge can relate to many aspects of the problems, such as a solution strategy (in the form of demonstrations, reward functions or additional optimization objectives), or the environment dynamics and tasks (through simulators and network structure). As argued, some of these sources will not be available in the situations where the tasks that need to be learned by a robot were not foreseen. Ideally, the included prior knowledge should therefore be very general. One such source of general prior knowledge about control problems are the classical control algorithms that DRL controllers are (partly) replacing. For instance, by embedding differentiable versions of filters and observers, the model search space could be reduced further without introducing too much of a task specific bias. Here again, the true underlying question is how, when, and why deep reinforcement learning works, and when would it be a good idea to (not) use it. Hopefully the next few years will bring more insights into these questions.

A

Benchmarks

This appendix provides additional information on the physical control set-ups and simulation benchmarks used in this thesis.

A.1 | 2-link robot arm

The physical robot arm setup depicted in Figure 3.1 consists of two links that are connected through a motorized joint. The arm hangs down from another motorized joint. The angle between the base and the first link is θ_1 and the angle of the second link with respect to the first link is θ_2. Both joints are physically constrained to $\theta_1, \theta_2 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. The control signals u_1, u_2 represent scaled versions of the voltages to the motors in the joints. To prevent damage to the physical setup due to jittering, the control signal to the setup is filtered with a low-pass filter:

$$u_k = 0.9u_{k-1} + 0.1a_k.$$

To ensure the Markov property is satisfied, the observation at time k consists of the angles and angular velocities of the joints and the preceding motor signals u_{k-1}), as well as the reference:

$$o_k = \begin{bmatrix} \theta_{1_k} & \theta_{2_k} & \dot{\theta_{1_k}} & \dot{\theta_{2_k}} & u_{1_{k-1}} & u_{2_{k-1}} & r_{x_k} & r_{y_k} \end{bmatrix}^T \tag{A.1}$$

The reference in A.1 is given in Cartesian coordinates whereas the state of the arm is observed in angular coordinates. It is left up to the neural networks to learn the mapping between the two and to deal with the fact that some reference positions can be reached through multiple arm configurations.

The reward for taking action a after observing o at time k is based on the distance between the Cartesian coordinates of the end of the second link at time-step $k+1$, as shown in Figure 3.1, and the reference position at time-step k:

$$r(o_k, a_k, o_{k+1}) = w_1 d_k^2 + w_2(1 - e^{-\alpha d_k^2}) + w_3 \|\dot{\theta_k}\|_2 \tag{A.2}$$

with

$$d_k^2 = \left\| \begin{bmatrix} r_{x_k} - x_{k+1} \\ r_{y_k} - y_{k+1} \end{bmatrix} \right\|_2^2,$$
$$x = \sin(\theta_1) + \sin(\theta_1 + \theta_2),$$
$$y = \cos(\theta_1) + \cos(\theta_1 + \theta_2).$$

The first term in (A.2) ensures that the initial learning is quick. Even when the policy is very bad the quadratic cost ensures that it is clear that moving towards the reference position is better than moving away from it. Using a quadratic term alone will not result in good final policies, however, since such a cost function is very flat close to the reference. The second term adds a steep drop in the cost very close to the reference to solve this problem. The third term discourages the highly oscillatory responses that might otherwise result. The constants in (A.2) that are used in the experiments are $w_1 = 0.2, w_2 = 0.4, w_3 = 8, \alpha = 100$.

The forgetting factor γ is calculated via:

$$\gamma = e^{-\frac{T_s}{\tau_\gamma}}$$

Where T_s is the sampling period and τ_γ is the look-ahead horizon in seconds. For this horizon a value of 2.5 seconds was used, which gives $\gamma = 0.996$ for the used control frequency of 100 Hz.

A.2 | Pendulum and Magman simulations

Here, a more detailed mathematical description is given of the pendulum swing-up and magnetic manipulation benchmarks. A high level description of these benchmarks was given in Section 3.4. Implementations of these benchmarks are available online.[1]

The benchmarks will be described based on their true physical environment states s and control signals u. In the main body of this thesis we instead deal with normalized observations and actions: $o \in [-1, 1]^n, a \in [-1, 1]^m$. See Figure 2.1 for a description of the symbols used.

The dynamics of both problems are defined as differential equations, which we use to calculate the next environment state s' as a function of the current state s and control signal u using the (fourth order) Runge-Kutta method. The reward is in both cases given by:

[1] https://github.com/timdebruin/CoR-control-benchmarks

$$r = -(W_1||s'| - s_{\text{ref}}| + W_2|u|). \tag{A.3}$$

In both cases a fixed reference state s_{ref} is used.

Pendulum Swing-Up

For the pendulum swing-up task, the state s is given by the angle $\theta \in [-\pi, \pi]$ and angular velocity $\dot{\theta}$ of a pendulum, which starts out hanging down under gravity $s_{k=0} = [\theta\ \dot{\theta}]^T = [0\ 0]^T$. For the normalization of the velocities, $\theta_{\min} = -30\,\text{rad}\,\text{s}^{-1}$ and $\theta_{\max} = 30\,\text{rad}\,\text{s}^{-1}$ are used. The action space is one dimensional: it is the voltage applied to a motor that exerts torque on the pendulum $u \in [-3, 3]$ V. The angular acceleration of the pendulum is given by:

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

Where $J = 9.41 \times 10^{-4}\,\text{kg}\,\text{m}^2$, $M = 5.5 \times 10^{-2}\,\text{kg}$, $g = 9.81\,\text{m}\,\text{s}^{-2}$, $l = 4.2 \times 10^{-2}\,\text{m}$, $b = 3 \times 10^{-6}\,\text{kg}\,\text{m}^2\,\text{s}^{-1}$, $K = 5.36 \times 10^{-2}\,\text{kg}\,\text{m}^2\,\text{s}^{-2}\,\text{A}^{-1}$ and $R = 9.5\,\text{V}\,\text{A}^{-1}$ are respectively the pendulum inertia, the pendulum mass, the acceleration due to gravity, pendulum length, viscous damping coefficient, the torque constant and the rotor resistance (Alibekov et al., 2018). For this task $W_1 = [50\ 1]$ and $W_2 = 10$ and $s_{\text{ref}} = [-\pi\ 0]^T = [\pi\ 0]^T$. The absolute value of the state is used in (A.3).

Magnetic Manipulation

In the magnetic manipulation problem, the action space represents the squared currents through four electromagnets under the track; $u^j \in [0,\ 0.6]A^2$ for $j = 1, 2, 3, 4$. The state of the problem is defined as the position $x \in [-0.035,\ 0.105]$ m of the ball relative to the center of the first magnet and the velocity $\dot{x}\,\text{m}\,\text{s}^{-1}$ of the ball: $s = [x\ \dot{x}]^T$. For the normalization of the velocities, $\dot{x}_{\min} = -0.4\,\text{m}\,\text{s}^{-1}$ and $\dot{x}_{\max} = 0.4\,\text{m}\,\text{s}^{-1}$ are used. When the position of the ball exceeds the bounds, the position is set to the bound and the velocity is set to $0.01\,\text{m}\,\text{s}^{-1}$ away from the wall. An additional reward of -1 is given for the time-step at which the collision occurred. The acceleration of the ball is given by:

$$\ddot{x} = -\frac{b}{m}\dot{x} + \frac{1}{m}\sum_{j=1}^{4} g\,(x, j)\,u^j,$$

with

$$g(x, j) = \frac{-c_1\ (x - 0.025j)}{\left((x - 0.025j)^2 + c_2\right)^3}.$$

Here, $g(x, j)$ is the nonlinear magnetic force equation, $m = 3.200 \times 10^{-2}\,\text{kg}$ the ball mass, and $b = 1.613 \times 10^{-2}\,\text{N}\,\text{s}\,\text{m}^{-1}$ the viscous friction of the ball on the

rail. The parameters c_1 and c_2 were empirically determined from experiments on a physical setup to be $c_1 = 5.520 \times 10^{-10}\,\mathrm{N\,m^5\,A^{-1}}$ and $c_2 = 1.750 \times 10^{-4}\,\mathrm{m^2}$ (Alibekov et al., 2018).

For the magnetic manipulation problem we take $W_1 = [100\ 5]$, $W_2 = [0\ 0\ 0\ 0]$, $s_{k=0} = [0\ 0]^T$ and $s_{\mathrm{ref}} = [0.035\ 0]^T$ in (A.3).

A.3 | CarRacing-v0

We use the de CarRacing-v0 benchmark of the OpenAI gym suite (Brockman et al., 2016), with the following adjustments:

Input pre-processing: To reduce the memory usage of the replay buffer, we convert the original 96x96x3 rgb images to 84x84 gray-scale images. Our initial tests showed that this did not affect the learning performance.

Action discretization: The original task has 3 continuous action dimensions: the steering angle $\in [-1, 1]$, accelerator $\in [0, 1]$ and break $\in [0, 1]$ inputs. We discretize the action space by using the following 7 actions:

Table A.1: Discrete actions used during the CarRacing experiments.

action	steering	acceleration	breaking
1	-1	0.2	0
2	-0.5	0.5	0
3	0	0.5	0
4	0	0.8	0
5	0	0	0.8
6	0.5	0.5	0
7	1	0.2	0

A.4 | Atari

We used the OpenAI gym interface (Brockman et al., 2016), together with the OpenAI baselines `wrap_deepmind` function[2] to the interface with the `EnduroNoFrameskip-v4` and `FreewayNoFrameskip-v4` environments. The `wrap_deepmind` function performs the following modifications to make the environments behave as in the original DQN paper (Mnih et al., 2015):

[2] https://github.com/openai/baselines

- On games with multiple lives, episodes end ($\mathbb{T} = 1$) when a life is lost, but the environment is only reset after all lives are lost.
- Rewards are clipped using the sign function to be one of $\{-1, 0, 1\}$.
- Observations are converted to 84x84 gray-scale images, and four subsequent images are stacked into one new observation.

B

Implementation details

This appendix describes additional implementation details of the experiments in this thesis.

B.1 | Physical arm experiments

The networks used for both the actor and the critic are fully connected networks with Rectified Linear Unit (ReLU) (Nair and Hinton, 2010) nonlinearities. Both networks have two hidden layers of equal size. In the critic network the action inputs come into the network before the second hidden layer. The number of hidden units in both networks is chosen such that they have around 10000 parameters each. This gives the actor network 94 neurons per hidden layer and the critic network 93 neurons per hidden layer. As in (Lillicrap et al., 2016), batch normalization (Ioffe and Szegedy, 2015) is used on the inputs to all layers of the actor network and all layers prior to the action input of the critic network. On the critic network, an $\mathcal{L}_2$ weight decay penalty of $0.5 \cdot 10^{-2}$ was used. The Adam (Kingma and Ba, 2015) optimization algorithm is used to train the neural networks. This optimization algorithm is appropriate for non-stationary objective functions and noisy gradients, which makes it suitable for reinforcement learning problems.

The experiments in this paper have been run in a growing batch setting (Lange et al., 2012a). Each trial consists of 40 episodes of 60 seconds. During these trials the experiences are added to the experience database. Between episodes the neural networks are updated based on the experiences in the complete database. For each separate trial in the experiment the initialization of the networks and the training references are unique and the database is reset. When the database is full the first experiences are overwritten in a first in first out manner.

The noise process that is used is an Ohrnstein-Uhlenbeck process:

$$\mathcal{O}(k) = \mathcal{O}(k-1) - \alpha\mathcal{O}(k-1) + \beta\mathcal{N}(0,1) \tag{B.1}$$

With $\alpha = 0.6$, $\beta = 0.4$ and $\mathcal{N}(0,1)$ is Gaussian noise with a mean of 0.0 and standard deviation 1.0. During training the references are determined stochastically

and in such a way that they can be reached with both joints $\theta_1, \theta_2 \in [-0.5, 0.5]$. They are changed periodically during the trials. After each learning run the final policy is evaluated with a fixed sequence of predetermined references in the same range.

B.2 | Experience buffer experiments (Chapters 3 and 4)

This section discusses the chosen hyperparameters of the methods discussed in Chapters 3 and 4. Only those hyperparameters that were not explicitly mentioned in the earlier sections of this work are mentioned here.

B.2.1 Neural Networks

This subsection describes the architecture and training procedure of the used neural networks.

Swing-up and Magman

To perform the experiments on these benchmarks, the DDPG method of Lillicrap et al. (2016) was reimplemented in Torch (Collobert et al., 2011a). For all experiments except for the control experiment in Appendix C, the actor and critic networks had the following configuration:

The actor is a fully connected network with two hidden layers, each with 50 units. The hidden layers have rectified linear activation functions. The output layer has hyperbolic tangent nonlinearities to map to the normalized action space.

The critic is a fully connected network with three hidden layers. The layers have rectified linear activation functions and 50, 50 and 20 units respectively. The observation is the input to the first hidden layer, while the action is concatenated with the output of the first hidden layer and used as input to the second hidden layer. The output layer is linear.

To train the networks, the ADAM optimization algorithm is used (Kingma and Ba, 2015). We use a batch size of 16 to calculate the gradients. For all experiments we use 0.9 and 0.999 as the exponential decay rates of the first and second order moment estimates respectively. The step-sizes used are 10^{-4} for the actor and 10^{-3} for the critic. We additionally use $\mathcal{L}_2$ regularization on the critic weights of 5×10^{-3}.

For the DQN experiments, a critic network similar to the DDPG critic was used. The critic only differs in the fact that instead of having actions as an input, the output size is increased to the number of discrete actions considered. The parameters θ^- of the target critic are updated to equal the online parameters θ every 200 batch updates.

Roboschool Benchmarks

For the experiments on the Roboschool benchmarks, we use a slightly modified version of the DDPG implementation in the openAI baselines (Dhariwal et al., 2017) repository. We have adapted the baselines code to include the experience selection methods considered in this section. Our adapted code is available online.[1] We here summarize the relevant differences from the implementation used on the simple benchmarks.

The actor and critic networks have two hidden layers with 64 units each. Layer normalization (Ba et al., 2016) is used in both networks after both hidden layers. The multiplier of the $\mathcal{L}_2$ regularization on the weights of the critic with is 1×10^{-2}. A batch size of 64 is used, with a sample reuse of 32. Training is performed every 100 environment steps, rather than after completed episodes.

B.2.2 Exploration

Swing-Up and Magman

We use an Ornstein-Uhlenbeck noise process (Uhlenbeck and Ornstein, 1930) (B.1) as advocated by Lillicrap et al. (2016). We use $\alpha = 5.14$, $\beta = 0.3$. Using this temporally correlated noise allows for more effective exploration in domains such as the pendulum swing-up. It also reduces the amount of damage on physical systems relative to uncorrelated noise (Koryakovskiy et al., 2017). For high frequencies, uncorrelated noise is unlikely to result in more than some small oscillations around the downward equilibrium position.

The noise signal is clipped between -1 and 1 after which it is added to the policy action, which is also clipped to get the normalized version of the control action a.

For the DQN experiments, epsilon greedy exploration was used with the probability of taking an action uniformly at random decaying linearly from $\epsilon = 0.7$ to $\epsilon = 0.01$ over the first 500 episodes.

Roboschool Benchmarks

For easy comparison to other work, we use the exploration strategy included in the baselines code. This means that for the Roboschool benchmarks we do not decay the strength of the exploration signal over time. Compared to our other benchmarks, the second difference is that the noise is added in the parameter space of the policy rather than directly in the action space (Plappert et al., 2018). The amplitude of the noise on the parameters is scaled such that the standard deviation of the exploration signal in action-space is 0.2.

[1] The code is available at https://github.com/timdebruin/baselines-experience-selection.

B.2.3 Baseline Controller

In this work we use the fuzzy Q-iteration algorithm of Buşoniu et al. (2010) as a baseline. This algorithm uses full knowledge of the system dynamics and reward function to compute a controller that has a proven bound on its sub-optimality for the deterministic (noise-free) case.

For the tests with sensor and actuator noise, the same controller as in the noise-free setting is used. To make the performance normalization (Section 3.5) fair, the performance of the controller is taken as the mean of 50 repetitions of taking the maximum obtained mean reward per episode over 1000 episodes with different realizations of the noise:

$$r_{\text{baseline with noise}} = \frac{1}{50}\sum_{i=1}^{50} \max(r^{\text{mean}}_{\text{episode i,1}}, \ldots, r^{\text{mean}}_{\text{episode i,1000}}).$$

Note that although this equalizes the chances of getting a favorable realization of the sensor and actuator noise sequences, it does not compensate for the fact that the fuzzy Q-iteration algorithm is unsuitable for noisy environments. Since the DDPG method used in this work can adjust the learned policy to the presence of noise in the environment, it outperforms the baseline in some situations. This is not an issue since we are interested in the relative performance of different experience selection strategies and only use the baseline as a reference point.

B.3 | State representation Learning (Chapter 5)

This section contains some additional details of the implementation of the experiments that were reported on in Chapter 5.

Optimization was done with the ADAM algorithm, using a learning rate of $3 \cdot 10^{-4}$ and $\beta_1 = 0.9, \beta_2 = 0.999$ (Kingma and Ba, 2015).

Epsilon greedy exploration was used with a $\epsilon = 0.15$ during training. To evaluate the performance of the learned controllers, tests were performed on the test track for 2000 environment steps with $\epsilon = 0.05$ to ensure some variation in the roll-outs and to test robustness. We used a discount factor of $\gamma = 0.95$. The considered actions were $a \in \{[-0.5, 0.2], [0.5, 0.2], [-0.2, 0.4], [0.2, 0.4], [0, 0.6], [0, -0.8]\}$, with the first dimension the steering command and the second the acceleration / break command. Experiences were stored in a replay buffer with a capacity of $2 \cdot 10^4$. We used a batch size $S = 16$ and a sample reuse of 16.

B.4 | Optimization (Chapter 6)

This section contains details of the implementation of the experiments that were reported on in Chapter 6.

B.4.1 CarRacing-v0

On this benchmark we use the standard DQN architecture apart from added layer normalization and a policy head, as represented in Table B.1:

Table B.1: Used DQN architecture for the CarRacing benchmark

Input: 84x84 8-bit gray scale images cast to 32 bit float and divided by 255	
Conv1: 8x8x32 (stride 4, ReLU)	
Conv2: 4x4x64 (stride 2, ReLU)	
Conv3: 3x3x64 (stride 1, ReLU)	
Fully connected + layer norm(Ba et al., 2016): 512 ReLU ($\bar{s}$)	
policy head: Stop Gradient policy head: Fully connected 7, softmax	$\hat{Q}$-head: Fully connected 7, linear

During the gradient based phase, the parameter noise magnitude σ is updated as described in the main paper to choose actions that differ from the policy with a probability $\epsilon = 0.1$. The first 200 episodes are performed with a random policy (actions are sampled uniformly at random in the action space) to fill the experience buffer.

For the DQN and DDPG algorithms we used the following hyper-parameters and implementation details (in addition to those mentioned in the main paper).

- DQN (Mnih et al., 2015):
  - optimizer: ADAM(Kingma and Ba, 2015)
    - learning rate: $\cdot 10^{-4}$
    - $\beta_1 = 0.9, \beta_2 = 0.999$
    - $\epsilon = 10^{-4}$
  - experience buffer size: 10^6 experiences
  - batch size: 32
  - parameter update every 4 environment steps, frozen parameter update every 10^4 environment steps
  - parameter gradient clipping: 10

  - loss functions:
    * TDE: Huber loss with $\delta = 1$
    * $0.01 \cdot ||\theta_{q,a}||_2$
    * cross entropy for policy head
  - $\gamma = 0.99$

- DDPG (Lillicrap et al., 2016):
  - optimizer: ADAM(Kingma and Ba, 2015)
    * learning rate: $\cdot 10^{-3}$ for the critic and $\cdot 10^{-4}$ for the actor
    * $\beta_1 = 0.9, \beta_2 = 0.999$
    * $\epsilon = 10^{-8}$
  - experience buffer size: 10^5 experiences
  - batch size: 64
  - parameter update every steps, frozen parameter update step with $\tau = 0.001$
  - parameter gradient clipping: 10
  - loss function:
    * TDE: Huber loss with $\delta = 1$
    * $0.01 \cdot ||\theta_q||_2$
  - $\gamma = 0.95$
  - $\sigma_a = 0.2$

For DQN, we do not allow the gradients of the policy head to back-propagate into the state encoder.

SRL experiment

For the experiment with state representation losses mentioned in the main paper, we added the following subnetworks and losses:

- Auto encoding:
  - Loss: 10 * $\|o - \hat{o}\|^2$
  - Subnetwork: $\bar{s} \rightarrow$ fully connected ELU layer with 441 units $\rightarrow$ reshaped to (21,21,1) $\rightarrow$ (3,3,32) transpose convolution with stride 2, ELU activation and batch normalization $\rightarrow$ (3,3,1) transpose convolution with stride 2, ELU activation and batch normalization $\rightarrow \hat{o}$.

- Reward prediction:
  - Loss: $0.5(r - \hat{r})^2$
  - Subnetwork: $\bar{s}$ concatenated with one hot representation of $a \rightarrow$ fully connected ELU layer with 32 units $\rightarrow$ fully connected linear layer with 1 unit $\rightarrow \hat{r}$.
- Forward dynamics:
  - Loss: $10 \left\| \bar{s}' - \hat{\bar{s}}' \right\|^2$
  - Subnetwork: $\bar{s}$ concatenated with one hot representation of $a \rightarrow$ fully connected ELU layer with 64 units $\rightarrow$ fully connected linear layer with 512 units $\rightarrow \hat{\bar{s}}'$.
- Inverse dynamics:
  - Loss: $1 \cdot$ cross entropy based on actually taken action a and the assigned probability.
  - Subnetwork: $\bar{s}$ concatenated with $\bar{s}' \rightarrow$ fully connected ELU layer with 64 units $\rightarrow$ softmax layer with 7 units.

C

Additional results

This section contains additional analyses and figures that were left out of the main body of the paper for brevity.

C.1 | Experience buffer experiments (Chapters 3 and 4)

Performance on the Magman Benchmark as a Function of Network Size

In the main body of the paper, a number of experiments are shown in which the performance of the magman benchmark is better with a small FIFO experience buffer than it is when retaining all experiences. As we use relatively small neural networks on the magman benchmark, it could be expected that at least part of the reason that training on all experiences results in poorer performance is that the function approximator simply does not have enough capacity to accurately cover the state-action space. We therefore compare the performance of the networks used on the magman benchmark in the main body of this work to that of the original DDPG architecture, which has more than 40 times as many parameters. Table C.1 compares the network architectures and the number of parameters of these architectures. It can be seen from Figure C.1 that, while the larger network is able to learn more successfully from the `FULL DB` buffer, it is outperformed by both the small and the large network using the FIFO buffer. The eventual performance

Architecture	hidden layer units	parameters swing-up	parameters magman
Small-critic	[50, 50, 20]	3791	3941
Small-actor	[50, 50]	2751	2904
DDPG original-critic	[400, 300]	122101	123001
DDPG original-actor	[400, 300]	121801	122704

Table C.1: The architectures of the networks compared in this section, with the number of parameters.

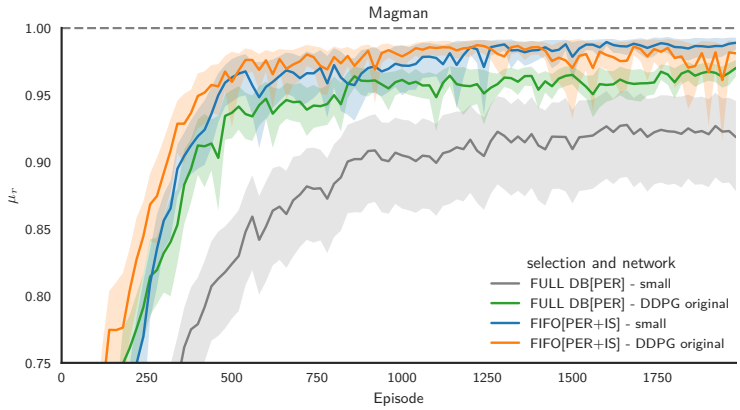


Figure C.1: Influence of network size on the performance of the magman benchmark, when retaining all 4×10^5 experiences (FULL DB) versus retaining only the last 10^4 experiences (FIFO). The small policy network used for most of the experiments on the magman has 2904 parameters, while the original DDPG network has 122704 parameters on the magman benchmark. In both cases the critic networks had slightly more parameters.

is best for our smaller network trained on a small buffer, although learning is somewhat faster with the larger network.

Sensitivity Analysis α

In both the PER sampling as well as the TDE and Expl retention methods, the parameter α (4.2) determines how strongly the used experience utility proxy influences the selection method. Here, we show the sensitivity of both Expl (Figure C.2) and PER (Figure C.3) with respect to this parameter.

In Figure C.2 it can be seen that on the Pendulum benchmark, where Expl retention has already been shown to aid stability, increasing α helps to improve the final performance more. This increased stability comes at the cost of somewhat reduced maximum performance. With PER sampling it does not seem to hurt the learning speed. On the Magman benchmark, where FIFO retention works better than Expl retention, increasing α (and thus relying more on the wrong proxy for the benchmark) hurts performance. Interesting to see is that compared to uniform sampling, PER speeds up the learning for low values of α, while it hurts for large values of α. This demonstrates again the need to choose both parts of experience selection with care.

In Figure C.3 it can again be seen that the benefits of PER are mostly to the speed of learning. Improvements to the maximum and final performance are possible

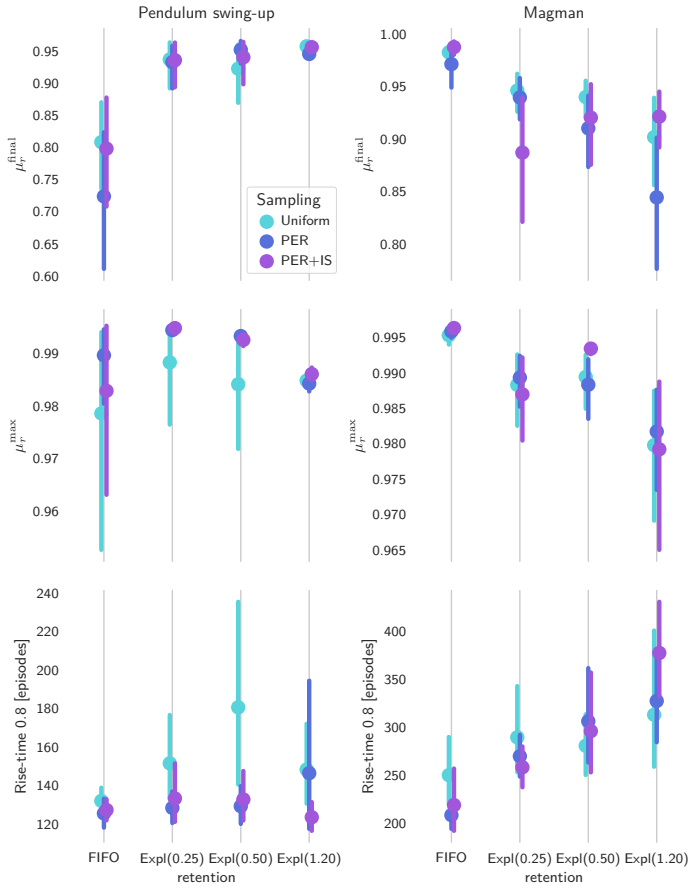


Figure C.2: Influence of α in the Expl algorithm for different sampling strategies.

when α is chosen correctly, but depend mostly on the contents of the buffer that PER is sampling from.

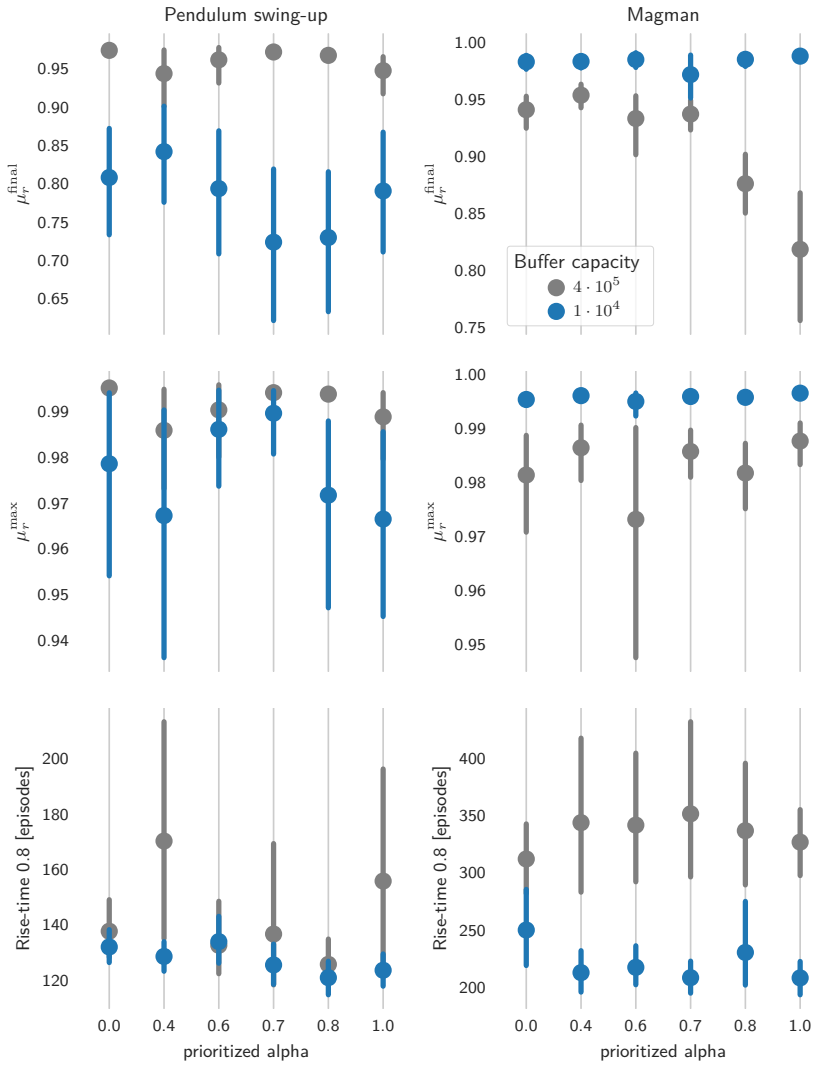


Figure C.3: Influence of α in the PER algorithm for the Full DB strategy (buffer capacity = 4×10^5) and FIFO retention (buffer capacity 1×10^{-4}).

Additional Figures

This subsection contains several figures that were left out of the main text of this work for brevity. They show the same experiments as Figures 3.7, 3.9, 4.6, according to the remaining performance criteria.

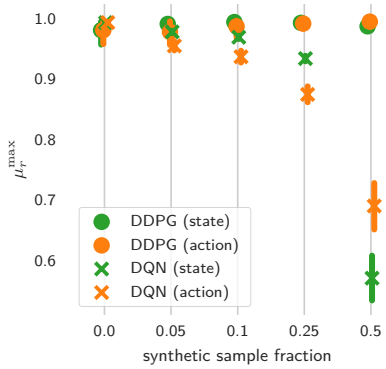


Figure C.4: RL algorithm dependent effect of adding synthetic experiences to the `FIFO`[`Uniform`] method on the maximum performance per episode $\mu_r^{\max}$ on the pendulum swing-up benchmark. The effect on the final performance and the rise-time is given in Figure 3.8.

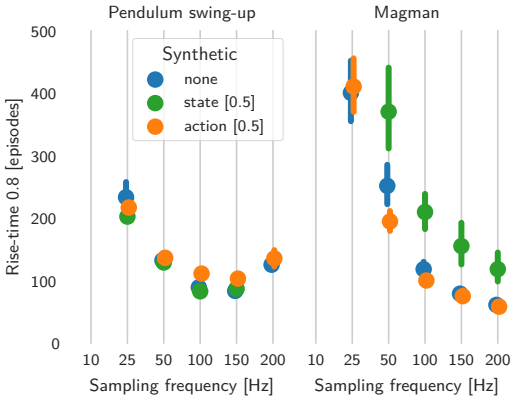
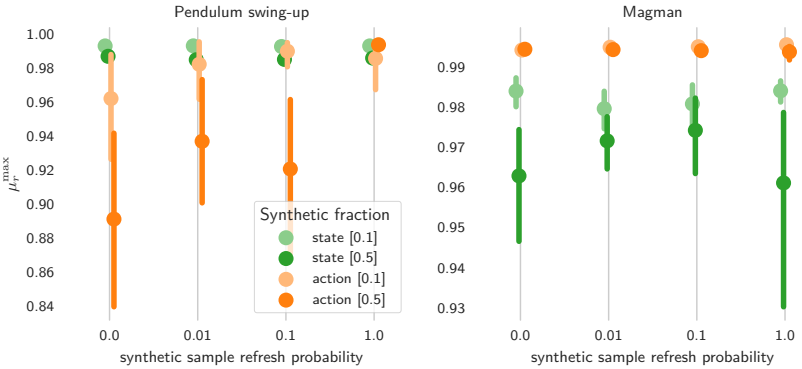
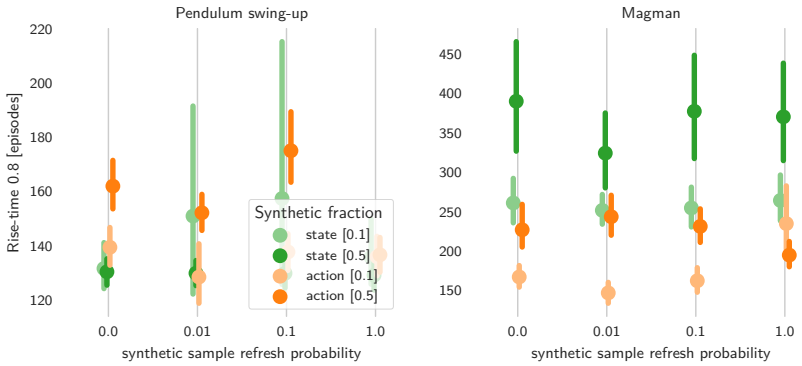


Figure C.5: Sampling frequency dependent effect on the learning speed of adding synthetic experiences to the `FIFO`[`Uniform`] method. The effect on the final and maximum performance is given in Figure 3.10.



(a) Effect on $\mu_r^{\max}$.



(b) Effect on Rise-time 0.8.

Figure C.6: The effects on the performance of the `FIFO[Uniform]` method when changing a fraction of the observed experiences with synthetic experiences, when the synthetic experiences are updated only with a certain probability each time they are overwritten. The effects on μ_r^{final} is shown in Figure 3.9.

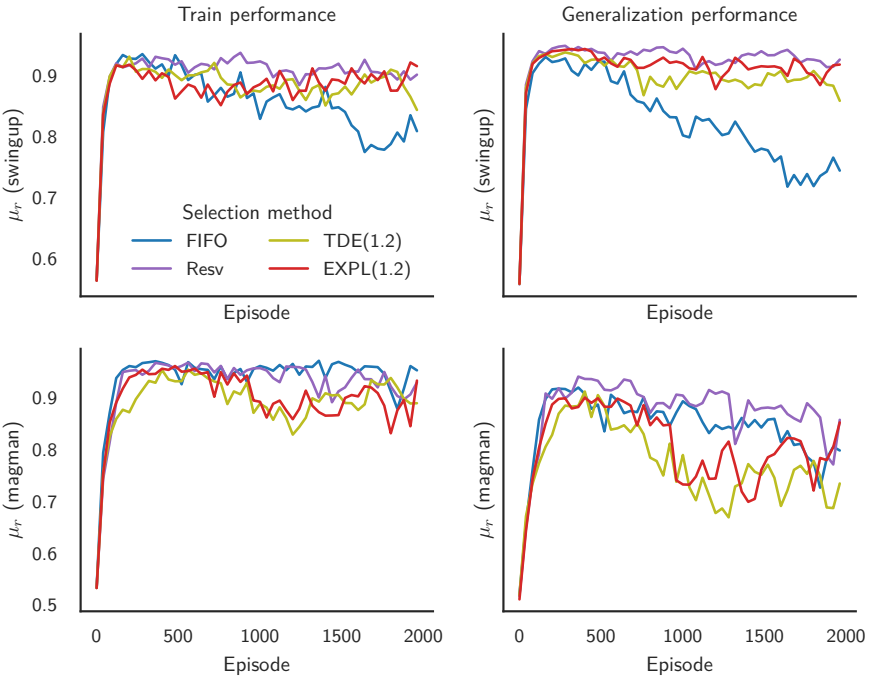


Figure C.7: Training and generalization performance of the experience retention methods proposed in Chapter 4 with `Uniform` sampling.

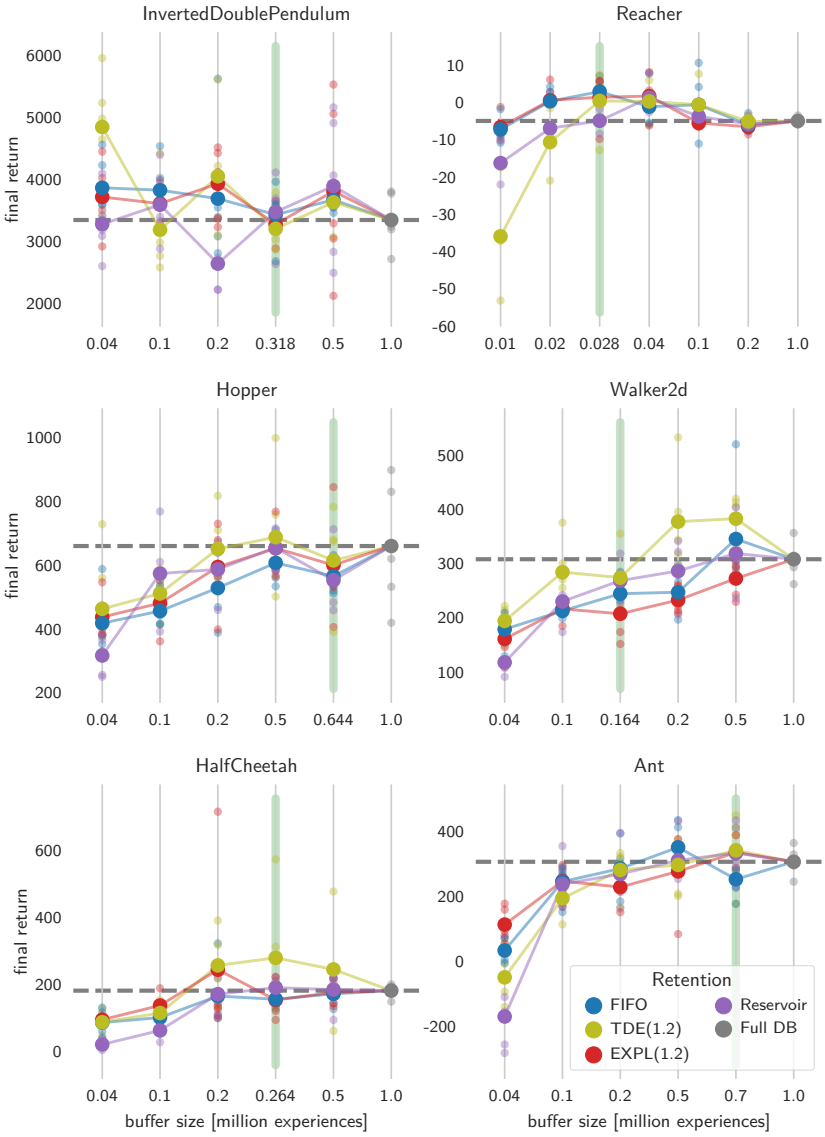


Figure C.8: Mean performance during the last 2×10^5 training steps of a 1×10^6 step training run on the Roboschool benchmarks as a function of the retention strategy and buffer size. Results for the individual runs and their means are shown.

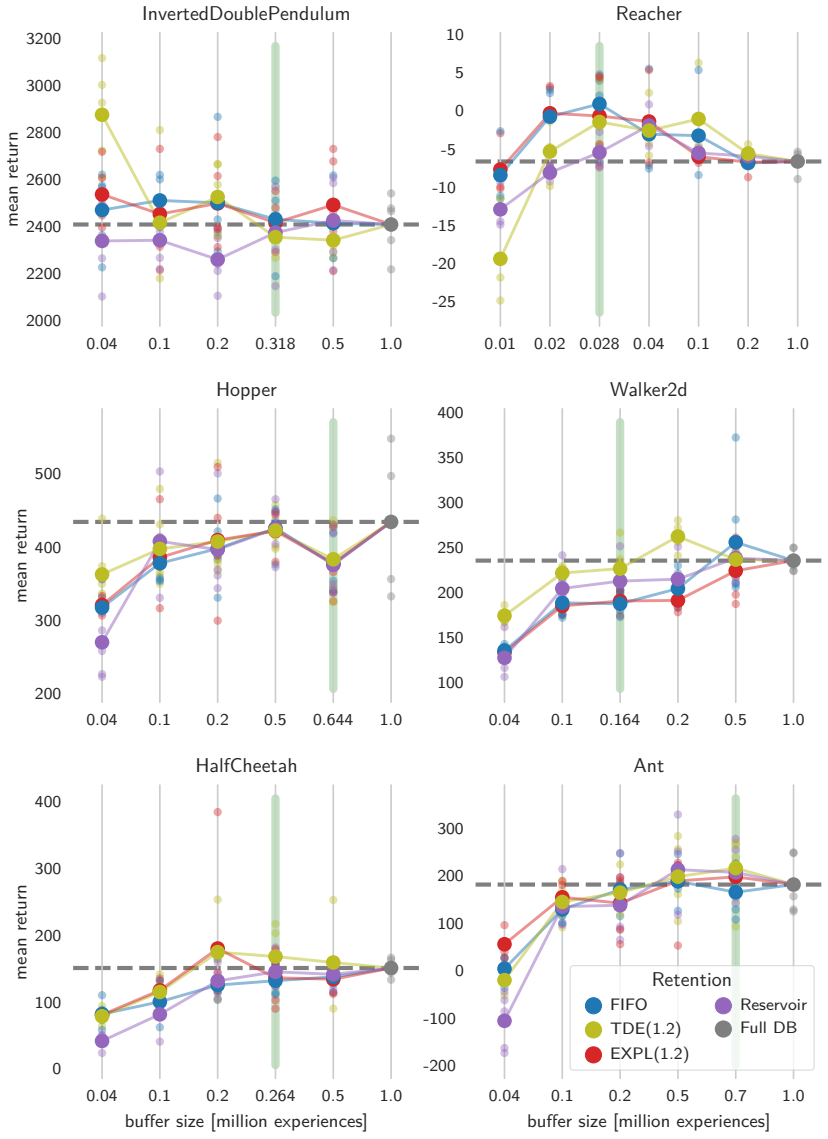


Figure C.9: Mean performance during the whole training run on the Roboschool benchmarks as a function of the retention strategy and buffer size. Results for the individual runs and their means are shown.

C.2 | Optimization (Chapter 6)

For Figure 6.2b of the main paper, we show in Figure C.10 how the mean and worst performance per iteration compared to the best performance per iteration. In Figure C.11, the same is shown with added SRL losses during the gradient based optimization phase. Both figures show the median of 3 runs.

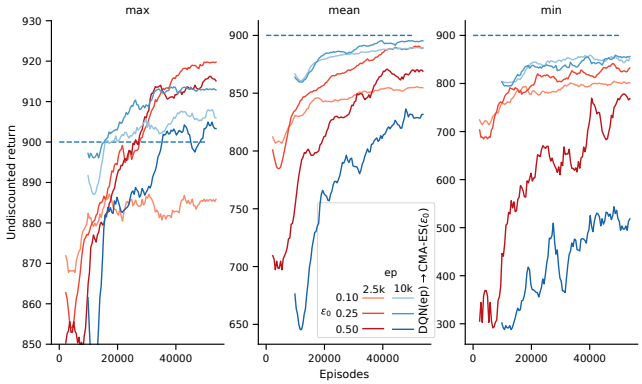


Figure C.10: Best, mean and worst performance per iteration on the CarRacing benchmark (accompanies Figure 6.2b of the main paper).

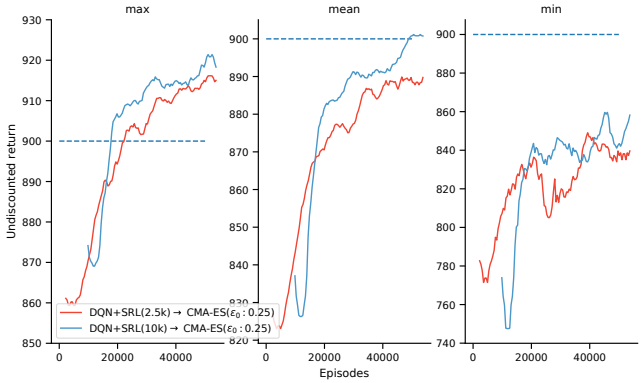


Figure C.11: Learning curves for the CarRacing experiment with added state representation learning losses.

References

Agrawal, P., Nair, A.V., Abbeel, P., Malik, J., and Levine, S. (2016). *"Learning to poke by poking: Experiential learning of intuitive physics"*. Neural Information Processing Systems (NIPS).

Alibekov, E., Kubalík, J., and Babuška, R. (2018). *"Policy derivation methods for critic-only reinforcement learning in continuous action spaces"*. Engineering Applications of Artificial Intelligence, 69, pp. 178–187.

Andre, D., Friedman, N., and Parr, R. (1997). *"Generalized prioritized sweeping"*. Advances In Neural Information Processing Systems (NIPS), pp. 1001–1007.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O.P., and Zaremba, W. (2017). *"Hindsight experience replay"*. Advances in Neural Information Processing Systems, pp. 5048–5058.

Aslanides, J., Leike, J., and Hutter, M. (2017). *"Universal reinforcement learning algorithms: Survey and experiments"*. International Joint Conference on Artificial Intelligence (IJCAI), pp. 1403–1410.

Ba, J.L., Kiros, J.R., and Hinton, G.E. (2016). *"Layer normalization"*. ArXiv preprint arXiv:1607.06450.

Baird, L.C. (1994). *"Reinforcement learning in continuous time: Advantage updating"*. World Congress on Computational Intelligence (WCCI), volume 4, pp. 2448–2453.

Banerjee, B. and Peng, J. (2004). *"Performance bounded reinforcement learning in strategic interactions"*. AAAI National Conference on Artificial Intelligence (AAAI), volume 4, pp. 2–7.

Bansal, T., Pachocki, J., Sidor, S., Sutskever, I., and Mordatch, I. (2017). *"Emergent complexity via multi-agent competition"*. arXiv preprint arXiv:1710.03748.

Barreto, A., Munos, R., Schaul, T., and Silver, D. (2017). *"Successor features for transfer in reinforcement learning"*. Neural Information Processing Systems (NIPS).

Barrett, S., Taylor, M., and Stone, P. (2010). *"Transfer learning for reinforcement learning on a physical robot"*. Adaptive Learning Agents Workshop, International Conference on Autonomous Agents and Multiagent Systems (AAMAS - ALA).

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R.

(2016a). *"Unifying count-based exploration and intrinsic motivation"*. Advances in Neural Information Processing Systems (NIPS), pp. 1471–1479.

Bellemare, M.G., Ostrovski, G., Guez, A., Thomas, P.S., and Munos, R. (2016b). *"Increasing the action gap: New operators for reinforcement learning."* Conf. Artificial Intelligence (AAAI).

Bengio, Y., Courville, A., and Vincent, P. (2013). *"Representation learning: A review and new perspectives"*. IEEE transactions on pattern analysis and machine intelligence, 35(8), pp. 1798–1828.

Bengio, Y., Delalleau, O., and Roux, N.L. (2006). *"The curse of highly variable functions for local kernel machines"*. Advances in neural information processing systems, pp. 107–114.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). *"Curriculum learning"*. International Conference on Machine Learning (ICML), pp. 41–48.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). *"Openai gym"*. CoRR, abs/1606.01540.

Bruin, T. de, Kober, J., Tuyls, K., and Babuška, R. (2018a). *"Experience selection in deep reinforcement learning for control"*. Journal of Machine Learning Research, 19(9), pp. 1–56.

Bruin, T. de, Kober, J., Tuyls, K., and Babuška, R. (2015). *"The importance of experience replay database composition in deep reinforcement learning"*. Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems (NIPS - DRLWS).

Bruin, T. de, Kober, J., Tuyls, K., and Babuška, R. (2016a). *"Improved deep reinforcement learning for robotics through distribution-based experience retention"*. International Conference on Intelligent Robots and Systems (IROS).

Bruin, T. de, Kober, J., Tuyls, K., and Babuška, R. (2016b). *"Off policy experience retention for deep actor critic learning"*. Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems (NIPS - DRLWS).

Bruin, T. de, Kober, J., Tuyls, K., and Babuška, R. (2018b). *"Integrating state representation learning into deep reinforcement learning"*. IEEE Robotics and Automation Letters, 3(3), pp. 1394–1401.

Buşoniu, L., Ernst, D., Babuška, R., and De Schutter, B. (2010). *"Approximate dynamic programming with a fuzzy parameterization"*. Automatica, 46(5), pp. 804–814.

Caarls, W. and Schuitema, E. (2016). *"Parallel online temporal difference learn-*

ing for motor control". IEEE Transactions on Neural Networks and Learning Systems, 27(7), pp. 1457–1468.

Carter, S., Armstrong, Z., Schubert, L., Johnson, I., and Olah, C. (2019). *"Activation atlas"*. Distill. Https://distill.pub/2019/activation-atlas.

Caruana, R. (1993). *"Multitask connectionist learning"*. Connectionist Models Summer School.

Chentanez, N., Barto, A.G., and Singh, S.P. (2004). *"Intrinsically motivated reinforcement learning"*. Advances in Neural Information Processing Systems (NIPS), pp. 1281–1288.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G.B., and LeCun, Y. (2015). *"The loss surfaces of multilayer networks"*. Artificial Intelligence and Statistics, pp. 192–204.

Ciosek, K. and Whiteson, S. (2017). *"OFFER: off-environment reinforcement learning"*. AAAI Conference on Artificial Intelligence (AAAI).

Coates, J. and Bollegala, D. (2018). *"Frustratingly easy meta-embedding–computing meta-embeddings by averaging source word embeddings"*. arXiv preprint arXiv:1804.05262.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011a). *"Torch7: A Matlab-like environment for machine learning"*. BigLearn Workshop, Advances in Neural Information Processing Systems (NIPS - BLWS).

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011b). *"Natural language processing (almost) from scratch"*. Journal of Machine Learning Research, 12(Aug), pp. 2493–2537.

Dayan, P. (1993). *"Improving generalization for temporal difference learning: The successor representation"*. Neural Computation, 5(4), pp. 613–624.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2017). *"OpenAI Baselines"*. https://github.com/openai/baselines.

Domhan, T., Springenberg, J.T., and Hutter, F. (2015). *"Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves."* International Joint Conference on Artificial Intelligence (IJCAI), volume 15, pp. 3460–3468.

Efron, B. (1992). *"Bootstrap methods: Another look at the jackknife"*. Breakthroughs in Statistics, pp. 569–593.

Finn, C., Tan, X.Y., Duan, Y., Darrell, T., Levine, S., and Abbeel, P. (2016).

"Deep spatial autoencoders for visuomotor learning". Int. Conf. Robotics and Automation (ICRA).

François-Lavet, V., Fonteneau, R., and Ernst, D. (2015). *"How to discount deep reinforcement learning: Towards new dynamic strategies"*. ArXiv preprint arXiv:1512.02011.

Franklin, G.F., Powell, D.J., and Workman, M.L. (1998). *"Digital control of dynamic systems"*, volume 3. Addison-Wesley Menlo Park.

Freund, Y., Schapire, R., and Abe, N. (1999). *"A short introduction to boosting"*. Journal of Japanese Society for Artificial Intelligence, 14(771-780), p. 1612.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *"The elements of statistical learning"*. 10. Springer series in statistics New York.

Garcıa, J. and Fernández, F. (2015). *"A comprehensive survey on safe reinforcement learning"*. Journal of Machine Learning Research, 16(1), pp. 1437–1480.

Ghadirzadeh, A., Maki, A., Kragic, D., and Björkman, M. (2017). *"Deep predictive policy training using reinforcement learning"*. arXiv:1703.00727.

Giusti, A., Guzzi, J., Ciresan, D., He, F.L., Rodriguez, J.P., Fontana, F., Faessler, M., Forster, C., Schmidhuber, J., Di Caro, G., Scaramuzza, D., and Gambardella, L. (2016). *"A machine learning approach to visual perception of forest trails for mobile robots"*. IEEE Robotics and Automation Letters.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *"Deep learning"*, volume 1. MIT press Cambridge.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). *"Generative adversarial nets"*. Advances in Neural Information Processing Systems (NIPS), pp. 2672–2680.

Goodfellow, I.J., Mirza, M., Da, X., Courville, A., and Bengio, Y. (2013). *"An empirical investigation of catastrophic forgeting in gradient-based neural networks"*. ArXiv preprint arXiv:1312.6211.

Greensmith, E., Bartlett, P.L., and Baxter, J. (2004). *"Variance reduction techniques for gradient estimates in reinforcement learning"*. Journal of Machine Learning Research, 5(Nov), pp. 1471–1530.

Gu, S., Lillicrap, T., Ghahramani, Z., Turner, R.E., and Levine, S. (2017a). *"Q-prop: sample-efficient policy gradient with an off-policy critic"*. International Conference on Learning Representations (ICLR).

Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. (2016). *"Continuous deep Q-learning with model-based acceleration"*. ArXiv preprint arXiv:1603.00748.

Gu, S., Lillicrap, T., Turner, R.E., Ghahramani, Z., Schölkopf, B., and Levine, S. (2017b). *"Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning"*. Advances in Neural Information Processing Systems, pp. 3849–3858.

Ha, D. and Schmidhuber, J. (2018). *"World models"*. arXiv preprint arXiv:1803.10122.

Haarnoja, T., Tang, H., Abbeel, P., and Levine, S. (2017). *"Reinforcement learning with deep energy-based policies"*. Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 1352–1361.

Hansen, N., Akimoto, Y., and Baudis, P. (2019). *"CMA-ES/pycma on Github"*. Zenodo, DOI:10.5281/zenodo.2559634. URL https://doi.org/10.5281/zenodo.2559634.

Hansen, N. and Ostermeier, A. (2001). *"Completely derandomized self-adaptation in evolution strategies"*. Evolutionary computation, 9(2), pp. 159–195.

Hasselt, H. van (2010). *"Double q-learning"*. Advances in Neural Information Processing Systems, pp. 2613–2621.

Hasselt, H. van, Guez, A., Hessel, M., Mnih, V., and Silver, D. (2016). *"Learning values across many orders of magnitude"*. Neural Information Processing Systems (NIPS).

Hausknecht, M., Lehman, J., Miikkulainen, R., and Stone, P. (2014). *"A neuroevolution approach to general atari game playing"*. IEEE Transactions on Computational Intelligence and AI in Games, 6(4), pp. 355–366.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2017). *"Deep reinforcement learning that matters"*. arXiv preprint arXiv:1709.06560.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). *"Rainbow: Combining improvements in deep reinforcement learning"*. arXiv preprint arXiv:1710.02298.

Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., et al. (2012). *"Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups"*. IEEE Signal processing magazine, 29(6), pp. 82–97.

Hinton, G., Vinyals, O., and Dean, J. (2015). *"Distilling the knowledge in a neural network"*. arXiv preprint arXiv:1503.02531.

Hinton, G.E. (2007). *"To recognize shapes, first learn to generate images"*. Progress

in Brain Research, 165, pp. 535–547.

Hinton, G.E. and Salakhutdinov, R.R. (2006). *"Reducing the dimensionality of data with neural networks"*. Science, 313(5786), pp. 504–507.

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). *"Gradient flow in recurrent nets: the difficulty of learning long-term dependencies"*.

Hoof, H. van, Chen, N., Karl, M., Smagt, P. van der, and Peters, J. (2016). *"Stable reinforcement learning with autoencoders for tactile and visual data"*. Int. Conf. Intelligent Robots and Systems (IROS).

Hornik, K. (1991). *"Approximation capabilities of multilayer feedforward networks"*. Neural networks, 4(2), pp. 251–257.

Houthooft, R., Chen, X., Duan, Y., Schulman, J., De Turck, F., and Abbeel, P. (2016). *"VIME: Variational information maximizing exploration"*. Advances in Neural Information Processing Systems (NIPS), pp. 1109–1117.

Ioffe, S. and Szegedy, C. (2015). *"Batch normalization: accelerating deep network training by reducing internal covariate shift"*. Int. Conf. Machine Learning (ICML).

Jaderberg, M., Mnih, V., Czarnecki, W.M., Schaul, T., Leibo, J.Z., Silver, D., and Kavukcuoglu, K. (2017). *"Reinforcement learning with unsupervised auxiliary tasks"*. Int. Conf. Learning Representations (ICLR).

Jonschkowski, R. and Brock, O. (2015). *"Learning state representations with robotic priors"*. Autonomous Robots, 39(3), pp. 407–428.

Jonschkowski, R., Hafner, R., Scholz, J., and Riedmiller, M. (2017). *"PVEs: Position-velocity encoders for unsupervised learning of structured state representations"*. New Frontiers for Deep Learning in Robotics Workshop at RSS.

Kakade, S. and Langford, J. (2002). *"Approximately optimal approximate reinforcement learning"*. ICML, volume 2, pp. 267–274.

Karras, T., Laine, S., and Aila, T. (2018). *"A style-based generator architecture for generative adversarial networks"*. arXiv preprint arXiv:1812.04948.

Kingma, D. and Ba, J. (2015). *"Adam: A method for stochastic optimization"*. International Conference for Learning Representations (ICLR).

Klimov, O. (2017). *"OpenAI Roboschool"*. https://github.com/openai/roboschool.

Kober, J., Bagnell, J.A., and Peters, J. (2013). *"Reinforcement learning in robotics: a survey"*. International Journal of Robotics Research (IJRR), 32(11), pp. 1238–1274.

Koryakovskiy, I., Vallery, H., Babuška, R., and Caarls, W. (2017). *"Evaluation of physical damage associated with action selection strategies in reinforcement learning"*. IFAC World Congress.

Koutník, J., Cuccu, G., Schmidhuber, J., and Gomez, F. (2013). *"Evolving large-scale neural networks for vision-based reinforcement learning"*. Proceedings of the 15th annual conference on Genetic and evolutionary computation, pp. 1061–1068.

Krizhevsky, A., Sutskever, I., and Hinton, G.E. (2012). *"Imagenet classification with deep convolutional neural networks"*. Advances in neural information processing systems, pp. 1097–1105.

Kulkarni, T.D., Saeedi, A., Gautam, S., and Gershman, S.J. (2016). *"Deep successor reinforcement learning"*. arXiv:1606.02396.

Kuvayev, L. and Sutton, R.S. (1996). *"Model-based reinforcement learning with an approximate, learned model"*. Yale Workshop on Adaptive Learning Systems.

Lange, S., Gabel, T., and Riedmiller, M. (2012a). *"Batch reinforcement learning"*. Reinforcement learning, pp. 45–73.

Lange, S., Riedmiller, M., and Voigtlander, A. (2012b). *"Autonomous reinforcement learning on raw visual input data in a real world application"*. Int. Joint Conf. Neural Networks (IJCNN).

Lee, A.X., Levine, S., and Abbeel, P. (2017). *"Learning visual servoing with deep features and fitted Q-iteration"*. Int. Conf. Learning Representations (ICLR).

Levine, N., Zahavy, T., Mankowitz, D.J., Tamar, A., and Mannor, S. (2017). *"Shallow updates for deep reinforcement learning"*. Advances in Neural Information Processing Systems, pp. 3135–3145.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). *"End-to-end training of deep visuomotor policies"*. Journal of Machine Learning Research, 17(39), pp. 1–40.

Levine, S. and Koltun, V. (2013). *"Guided policy search"*. International Conference on Machine Learning, pp. 1–9.

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

Lin, H.W., Tegmark, M., and Rolnick, D. (2017). *"Why does deep and cheap learning work so well?"* Journal of Statistical Physics, 168(6), pp. 1223–1247.

Lin, L.J. (1992). *"Self-improving reactive agents based on reinforcement learning,*

planning and teaching". Machine Learning, 8(3-4), pp. 293–321.

Lipton, Z.C., Gao, J., Li, L., Li, X., Ahmed, F., and Deng, L. (2016). *"Efficient exploration for dialogue policy learning with BBQ networks & replay buffer spiking"*. ArXiv preprint arXiv:1608.05081.

Liu, G.H., Siravuru, A., Prabhakar, S., Veloso, M., and Kantor, G. (2017). *"Learning end-to-end multimodal sensor policies for autonomous navigation"*. arXiv:1705.10422.

Loshchilov, I. and Hutter, F. (2015). *"Online batch selection for faster training of neural networks"*. ArXiv preprint arXiv:1511.06343.

Mania, H., Guy, A., and Recht, B. (2018). *"Simple random search provides a competitive approach to reinforcement learning"*. arXiv preprint arXiv:1803.07055.

Marbach, P. and Tsitsiklis, J.N. (2003). *"Approximate gradient methods in policy-space optimization of markov reward processes"*. Discrete Event Dynamic Systems, 13(1-2), pp. 111–148.

Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., et al. (2017). *"Learning to navigate in complex environments"*. Int. Conf. Learning Representations (ICLR).

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). *"Asynchronous methods for deep reinforcement learning"*. International Conference on Machine Learning, pp. 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al. (2015). *"Human-level control through deep reinforcement learning"*. Nature, 518(7540), pp. 529–533.

Moniz, J.R.A., Patra, B., and Garg, S. (2019). *"Compression and localization in reinforcement learning for atari games"*. arXiv preprint arXiv:1904.09489.

Montavon, G., Orr, G.B., and Müller, K.R., editors (2012). *"Neural networks: Tricks of the trade"*. Lecture Notes in Computer Science (LNCS). Springer, 2nd edition.

Montufar, G.F., Pascanu, R., Cho, K., and Bengio, Y. (2014). *"On the number of linear regions of deep neural networks"*. Advances in Neural Information Processing Systems (NIPS), pp. 2924–2932.

Moore, A.W. and Atkeson, C.G. (1993). *"Prioritized sweeping: reinforcement learning with less data and less time"*. Machine Learning, 13(1), pp. 103–130.

Munk, J., Kober, J., and Babuška, R. (2016). *"Learning state representation for*

deep actor-critic control". Conf. Decision and Control (CDC).

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. (2016). *"Safe and efficient off-policy reinforcement learning"*. Advances in Neural Information Processing Systems, pp. 1054–1062.

Nair, V. and Hinton, G.E. (2010). *"Rectified linear units improve restricted boltzmann machines"*. Int. Conf. Machine Learning (ICML).

Narasimhan, K., Kulkarni, T., and Barzilay, R. (2015). *"Language understanding for text-based games using deep reinforcement learning"*. Empirical Methods in Natural Language Processing (EMNLP).

Needell, D., Srebro, N., and Ward, R. (2016). *"Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm"*. Mathematical Programming, 155(1-2), pp. 549–573.

Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., and Ng, A.Y. (2011). *"Multimodal deep learning"*. Int. Conf. Machine Learning (ICML).

Olah, C., Mordvintsev, A., and Schubert, L. (2017). *"Feature visualization"*. Distill, 2(11), p. e7.

Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. (2016). *"Deep exploration via bootstrapped DQN"*. Advances In Neural Information Processing Systems (NIPS), pp. 4026–4034.

Parisotto, E., Ba, J.L., and Salakhutdinov, R. (2015). *"Actor-mimic: Deep multitask and transfer reinforcement learning"*. arXiv preprint arXiv:1511.06342.

Pérez Dattari, R., Celemin, C., Ruiz Del Solar, J., and Kober, J. (2019). *"Continuous control for high-dimensional state spaces: An interactive learning approach"*. IEEE International Conference on Robotics and Automation (ICRA).

Pieters, M. and Wiering, M.A. (2016). *"Q-learning with experience replay in a dynamic environment"*. Symposium Series on Computational Intelligence (SSCI), pp. 1–8.

Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R.Y., Chen, X., Asfour, T., Abbeel, P., and Andrychowicz, M. (2018). *"Parameter space noise for exploration"*. International Conference on Learning Representations (ICLR).

Precup, D., Sutton, R.S., and Singh, S.P. (2000). *"Eligibility traces for off-policy policy evaluation."* ICML, pp. 759–766.

Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Sohl-Dickstein, J. (2016). *"On the expressive power of deep neural networks"*. arXiv preprint arXiv:1606.05336.

Rajeswaran, A., Lowrey, K., Todorov, E.V., and Kakade, S.M. (2017). *"Towards generalization and simplicity in continuous control"*. Advances In Neural Information Processing Systems (NIPS), pp. 6550–6561.

Recht, B., Re, C., Wright, S., and Niu, F. (2011). *"Hogwild: A lock-free approach to parallelizing stochastic gradient descent"*. Advances in neural information processing systems, pp. 693–701.

Riedmiller, M. (2005). *"Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method"*. European Conference on Machine Learning, pp. 317–328.

Rusu, A.A., Vecerik, M., Rothörl, T., Heess, N., Pascanu, R., and Hadsell, R. (2016). *"Sim-to-real robot learning from pixels with progressive nets"*. ArXiv preprint arXiv:1610.04286.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). *"Evolution strategies as a scalable alternative to reinforcement learning"*. arXiv preprint arXiv:1703.03864.

Salimans, T. and Kingma, D.P. (2016). *"Weight normalization: A simple reparameterization to accelerate training of deep neural networks"*. Advances in Neural Information Processing Systems, pp. 901–909.

Schaal, S. (1999). *"Is imitation learning the route to humanoid robots?"* Trends in Cognitive Sciences, 3(6), pp. 233–242.

Schaul, T., Horgan, D., Gregor, K., and Silver, D. (2015). *"Universal value function approximators"*. International Conference on Machine Learning, pp. 1312–1320.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). *"Prioritized experience replay"*. International Conference on Learning Representations (ICLR).

Schmidhuber, J. (1991). *"A possibility for implementing curiosity and boredom in model-building neural controllers"*. From Animals to Animats: International Conference on Simulation of Adaptive Behavior (SAB).

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015a). *"Trust region policy optimization"*. International Conference on Machine Learning, pp. 1889–1897.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). *"High-dimensional continuous control using generalized advantage estimation"*. arXiv preprint arXiv:1506.02438.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). *"Prox-*

imal policy optimization algorithms". arXiv preprint arXiv:1707.06347.

Seo, Y.W. and Zhang, B.T. (2000). *"Learning user's preferences by analyzing web-browsing behaviors"*. International Conference on Autonomous Agents (ICAA), pp. 381–387.

Shelhamer, E., Mahmoudieh, P., Argus, M., and Darrell, T. (2016). *"Loss is its own reward: Self-supervision for reinforcement learning"*. arXiv:1612.07307.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). *"Mastering the game of go with deep neural networks and tree search"*. nature, 529(7587), pp. 484–489.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). *"Deterministic policy gradient algorithms"*. International Conference on Machine Learning (ICML).

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). *"Mastering the game of Go without human knowledge"*. Nature, 550(7676), pp. 354–359.

Skinner, B.F. (1958). *"Reinforcement today"*. American Psychologist, 13(3), p. 94.

Stachenfeld, K.L., Botvinick, M.M., and Gershman, S.J. (2017). *"The hippocampus as a predictive map"*. Nature Neuroscience, 20(11), pp. 1643–1653.

Stulp, F. and Sigaud, O. (2012). *"Path integral policy improvement with covariance matrix adaptation"*. Proceedings of the 29th International Conference on Machine Learning (ICML).

Sutton, R.S. (1991). *"Dyna, an integrated architecture for learning, planning, and reacting"*. ACM SIGART Bulletin, 2(4), pp. 160–163.

Sutton, R.S. and Barto, A.G. (2018). *"Reinforcement learning: An introduction"*. MIT press.

Sutton, R.S., McAllester, D.A., Singh, S.P., and Mansour, Y. (2000). *"Policy gradient methods for reinforcement learning with function approximation"*. Advances in neural information processing systems, pp. 1057–1063.

Tamar, A., Chow, Y., Ghavamzadeh, M., and Mannor, S. (2016). *"Sequential decision making with coherent risk"*. IEEE Transactions on Automatic Control.

Tieleman, T. and Hinton, G. (2012). *"Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude"*. COURSERA: Neural networks for machine learning, 4(2), pp. 26–31.

Tucker, G., Bhupatiraju, S., Gu, S., Turner, R.E., Ghahramani, Z., and Levine, S. (2018). *"The mirage of action-dependent baselines in reinforcement learning"*. arXiv preprint arXiv:1802.10031.

Uhlenbeck, G.E. and Ornstein, L.S. (1930). *"On the theory of the Brownian motion"*. Physical Review, 36(5), p. 823.

Valiant, L.G. (1984). *"A theory of the learnable"*. Communications of the ACM, 27(11), pp. 1134–1142.

Van Hasselt, H., Guez, A., and Silver, D. (2016). *"Deep reinforcement learning with double q-learning"*. Conf. Artificial Intelligence (AAAI).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., and Polosukhin, I. (2017). *"Attention is all you need"*. Advances in Neural Information Processing Systems, pp. 5998–6008.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). *"Show and tell: A neural image caption generator"*. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Vitter, J.S. (1985). *"Random sampling with a reservoir"*. ACM Transactions on Mathematical Software (TOMS), 11(1), pp. 37–57.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and Freitas, N. de (2017). *"Sample efficient actor-critic with experience replay"*. International Conference on Learning Representations (ICLR).

Watkins, C.J.C.H. (1989). *"Learning from delayed rewards"*. Ph.D. thesis, King's College, Cambridge.

Watter, M., Springenberg, J., Boedecker, J., and Riedmiller, M. (2015). *"Embed to control: A locally linear latent dynamics model for control from raw images"*. Neural Information Processing Systems (NIPS).

Williams, R.J. and Peng, J. (1991). *"Function optimization using connectionist reinforcement learning algorithms"*. Connection Science, 3(3), pp. 241–268.

Wiskott, L. and Sejnowski, T.J. (2002). *"Slow feature analysis: Unsupervised learning of invariances"*. Neural Computation, 14(4), pp. 715–770.

Wymann, B., Espié, E., Guionneau, C., Dimitrakakis, C., Coulom, R., and Sumner, A. (2000). *"Torcs, the open racing car simulator"*. Software available at http://torcs.sourceforge.net.

Yoshida, N. (2016). *"Gym-torcs"*. Software available at https://github.com/ugo-nama-kun/gym_torcs.

Zeghidour, N., Xu, Q., Liptchinsky, V., Usunier, N., Synnaeve, G., and Collobert, R. (2018). *"Fully convolutional speech recognition"*. arXiv preprint arXiv:1812.06864.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2016). *"Understanding deep learning requires rethinking generalization"*. arXiv preprint arXiv:1611.03530.

Zhang, J., Springenberg, J.T., Boedecker, J., and Burgard, W. (2017). *"Deep reinforcement learning with successor features for navigation across similar environments"*. Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on, pp. 2371–2378.

Acknowledgements

I had never considered the possibility of doing a PhD until I was asked if I would be interested in one. I didn't quite know what a PhD entailed but I did not think it was something that I would be good enough for—although I did not mention that at the time. Now, having nearly finished, I can happily look back on one of the best experiences of my life. For four years I got to do fun projects while being surrounded by smart, funny, kind, and generous people—all while somehow getting paid as well. To everyone I encountered during those years I would like to say a heartfelt *thank you*. In particular:

To Robert, Jens and Karl. Thank you for entrusting me with the project and recognizing my potential before I did so myself. I am very grateful for the freedom you gave me to deviate from your plans, and the fact that you managed to make me feel like an equal despite looking up to your academic accomplishments. Above all, I feel lucky to have had you as my (co-)promotors because you are all such genuinely nice people. Robert, after the MSc, I was very happy to work together for four more years. I have always enjoyed our discussions and the times you suddenly dropped by to show a new plot of an experiment you performed based on discussions we had. I think it is really cool to see how you lead a whole group while still doing experiments and taking the time to talk about the personal and professional details of the lives of the people you supervise. *Velmi vám děkuji*! Jens, in describing your supervision style to potential new PhD candidates I realized how perfectly you struck the balance; taking a hands off approach and giving us the freedom to find our own path, while always being available with helpful insights when asked. For me, that really made the PhD perfect. I also liked that you were truly "one of us". I will miss the shared lunches and the "ducklings" meetings. *Vielen Dank!* Karl, you were physically further away, and your perpetual out-of-office replies always claimed that you would be hard to reach. Yet I never found that to be the case. You always made time for meetings to discuss anything from papers to career advice—even when rushing to finish six conference papers simultaneously! *Heel erg bedankt!*

I mentioned that I did not know what a PhD entailed before I started one. In fact, it took me years before I felt like I knew what I should be doing. Fortunately, I did not go on the journey of discovery alone. In Andy, Cees and Ivan I had friends who's academic qualities were beyond any doubt to me. The fact that they were lost with me made me realize that maybe we weren't lost at all.

While being lost together is better than being lost alone, it also helps when someone who knows the path points you in the right direction. For this I especially want to thank Kim, who not only guided me through my MSc thesis—setting the example

of how to be a good PhD supervisor—but then continued to come back to share laughs, foosball matches, good talks and advice even after leaving the university. I also want to thank Subu, Jan-Maarten and Fahrid for patiently showing me the ropes in the early days.

I want to thank my office mates for the many laughs, talks, and what they taught me. They were also my first point of reference of what quality of work to strive for. There was Wouter, who came to join my project. The speed with which he absorbed the required knowledge was an eye-opener to me, and forced me to go faster myself. Besides laughs we also shared a very serious ongoing battle on the foosball table, where I was forced to improve a lot, although never quite enough. There was Linda, who due to her abundance in talents and accomplishments I prefer not to use as a reference point—but definitely enjoyed sharing an office with. There were Sebastiaan and Hongpeng, who set the bar when it came to positive attitude, Sjoerd who reminded me to always question the rules. There was the always friendly Carlos who with his COACH method that worked so annoyingly well and Ajith, Sherin and Charel, who showed how to never give up when faced with a seemingly insurmountable task like parsing the works of Friston. Besides the people in the office I would also like to thank Thomas, Reinier, Vahab, Mukunda, Bruno, Sander, Javier, Anqi and Sachin for being great colleagues.

Doing a PhD also involved teaching and supervising, which turned out to be a lot of fun. I very much enjoyed supervising Siddharth, Thijs, Vasos and Carlo, due to their enthusiasm and the creativity in their research. I enjoyed sharing TA duties with Sander, who set an excellent example.

In some sense universities are very weird places. To do research, it is necessary to be adaptive. And yet a university is a large bureaucratic institution. Thanks to Marieke, Heleen, Kiran, Hanneke, Karin and Hans not only does work get done, but it gets done in a really pleasant environment.

I also want to thank the people of the Intelligent and Interactive Systems group in Innsbruck for my fun research stay there: Justus and Connie for the hospitality, Philipp for going above and beyond in our work together and Jakob, Bart, Senka, Xiang, Athanasios and Erwan for making me feel instantly at home.

I want to thank my wonderful parents Paul and Judith for their continued support, cheerleading and feedback. For you, I will try to work on my Dutch.

Before agreeing to do a PhD I was thinking of moving to Australia to be with the girl of my dreams. Not only did she help convince me to do the PhD, but she came over and made me a happy person even during the infamous third year PhD slump. Therefore my final big tank you goes out to you, Rebecca. I am looking forward to all our future adventures!

About the author

14-02-1989 Born in Amsterdam, the Netherlands.

Education

2007–2012 BSc Mechanical Engineering
Delft University of Technology
Minor: Software Design

2013–2015 MSc Systems and Control
Delft University of Technology
Thesis: Railway Track Circuit Fault Diagnosis using Recurrent Neural Networks
Supervisor: Ir. K. Verbert
Supervisor: Prof. dr. R. Babuška

2015–2019 PhD.
Delft University of Technology
Thesis: Sample Efficient Deep Reinforcement Learning for Control
Promotor: Prof. dr. R. Babuška
Promotor: Prof. dr. K. Tuyls
Co-promotor: Dr.-Ing J. Kober

Awards

2016 Qualcomm Innovation Fellowship Europe

Visiting Scholar

2016 Intelligent and Interactive Systems
University of Innsbruck

List of publications

Journal papers

"Experience selection in deep reinforcement learning for control"
Tim de Bruin, Jens Kober, Karl Tuyls, Robert Babuška
the Journal of Machine Learning Research (JMLR), 2018

"Reinforcement learning for control:
Performance, stability, and deep approximators"
Lucian Buşoniu, **Tim de Bruin**, Domagoj Tolić, Jens Kober, Ivana Palunko
Annual Reviews in Control (ARC), 2018

"Integrating State Representation Learning into Deep Reinforcement Learning"
Tim de Bruin, Jens Kober, Karl Tuyls, Robert Babuška
IEEE Robotics and Automation Letters (RA-L / ICRA), 2018

Conference papers

"Improved Deep Reinforcement Learning for Robotics Through Distribution-based Experience Retention"
Tim de Bruin, Jens Kober, Karl Tuyls, Robert Babuška
EEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2016

Workshop papers

"Off-policy experience retention for deep actor-critic learning"
Tim de Bruin, Jens Kober, Karl Tuyls, Robert Babuška
Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems (NIPS), 2016

"The importance of experience replay database composition in deep reinforcement learning"
Tim de Bruin, Jens Kober, Karl Tuyls, Robert Babuška
Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems (NIPS), 2015